Programmez en orienté objet en PHP

Par Victor Thuillier (vyk12)

www.siteduzero.com

Dernière mise à jour le 8/01/2013

Sommaire

Programmez en orienté objet en PHP

Par  Victor Thuillier (vyk12)

Mise à jour : 08/01/2013
Difficulté : Intermédiaire Durée d'étude : 1 mois

Bienvenue dans ce tutoriel sur la programmation orientée objet (souvent abrégé par ses initiales « POO ») en PHP.

Ici, vous allez découvrir un nouveau moyen de penser votre code, un nouveau moyen de le concevoir. Vous allez le représenter de façon *orienté objet*, un moyen de conception inventé dans les années 1970 et qui prend de plus en plus de place aujourd'hui. La principale raison de ce succès est due à de nombreux avantages apportés par ce paradigme, comme une organisation plus cohérente de vos projets, une maintenance plus facile et une distribution de votre code plus aisée.

Cependant, avant de vous lancer dans ce (très) vaste domaine, vous devez avoir quelques connaissances au préalable.

Ce qui doit être acquis

Afin de suivre au mieux ce tutoriel, il est indispensable voire obligatoire :

- d'être à l'aise avec PHP et sa syntaxe. Si ce n'est pas le cas, le Site du Zéro propose un tutoriel ;
- d'avoir bien pratiqué ;
- d'être patient ;
- d'avoir PHP 5 sur son serveur. Je ne parlerai pas de POO en PHP 4 car sous cette version de PHP, certaines fonctions indispensables de la POO ne sont pas présentes (on ne peut donc pas vraiment parler de POO).

Si vous avez déjà pratiqué d'autres langages apportant la possibilité de programmer orienté objet, c'est un gros plus, surtout si vous savez programmer en Java (PHP a principalement tiré son modèle objet de ce langage).

Partie 1 : [Théorie] Les bases de la POO

Introduction à la POO

Alors ça y est, vous avez décidé de vous lancer dans la POO en PHP ? Sage décision ! Nous allons donc plonger dans ce vaste domaine par une introduction à cette nouvelle façon de penser : qu'est-ce que la POO ? En quoi ça consiste ? En quoi est-ce si différent de la méthode que vous employez pour développer votre site web ? Tant de questions auxquelles je vais répondre.

Cependant, puisque je sais que vous avez hâte de commencer, nous allons entamer sérieusement les choses en créant notre première **classe** dès la fin de ce chapitre. Vous commencerez ainsi vos premiers pas dans la POO en PHP !

Qu'est-ce que la POO ?

Il était une fois le procédural

Commençons ce cours en vous posant une question : comment est représenté votre code ? La réponse est unique : vous avez utilisé la « représentation procédurale » qui consiste à séparer le traitement des données des données elles-mêmes. Par exemple, vous avez un système de news sur votre site. D'un côté, vous avez les données (les news, une liste d'erreurs, une connexion à la BDD, etc.) et de l'autre côté vous avez une suite d'instructions qui viennent modifier ces données. Si je ne me trompe pas, c'est de cette manière que vous codez.

Cette façon de se représenter votre application vous semble sans doute la meilleure puisque c'est la seule que vous connaissez. D'ailleurs, vous ne voyez pas trop comment votre code pourrait être représenté de manière différente. Eh bien cette époque d'ignorance est révolue : voici maintenant la programmation orientée objet !

Puis naquit la programmation orientée objet

Alors, qu'est-ce donc que cette façon de représenter son code ? La POO, c'est tout simplement faire de son site un ensemble d'objets qui interagissent entre eux. En d'autres termes : tout est objet.

Définition d'un objet

Je suis sûr que vous savez ce que c'est. D'ailleurs, vous en avez pas mal à côté de vous : je suis sûr que vous avez un ordinateur, une lampe, une chaise, un bureau, ou que sais-je encore. Ce sont tous des objets. En programmation, les objets sont sensiblement la même chose.

L'exemple le plus pertinent quand on fait un cours sur la POO est d'utiliser l'exemple du personnage dans un jeu de combat. Ainsi, imaginons que nous ayons un objet `Personnage` dans notre application. Un personnage a des caractéristiques :

- une force ;
- une localisation ;
- une certaine expérience ;
- et enfin des dégâts.

Toutes ses caractéristiques correspondent à des valeurs. Comme vous le savez sûrement, les valeurs sont stockées dans des variables. C'est toujours le cas en POO. Ce sont des variables un peu spéciales, mais nous y reviendrons plus tard.

Mis à part ces caractéristiques, un personnage a aussi des capacités. Il peut :

- frapper un autre personnage ;
- gagner de l'expérience ;
- se déplacer.

Ces capacités correspondent à des fonctions. Comme pour les variables, ce sont des fonctions un peu spéciales et on y reviendra en temps voulu. En tout cas, le principe est là.

Vous savez désormais qu'on peut avoir des objets dans une application. Mais d'où sortent-ils ? Dans la vie réelle, un objet ne sort pas de nulle part. En effet, chaque objet est défini selon des caractéristiques et un plan bien précis. En POO, ces informations sont contenues dans ce qu'on appelle des **classes**.

Définition d'une classe

Comme je viens de le dire, les classes contiennent la définition des objets que l'on va créer par la suite. Prenons l'exemple le plus simple du monde : les gâteaux et leur moule. Le moule est unique. Il peut produire une quantité infinie de gâteaux. Dans ce cas-là, les gâteaux sont les *objets* et le moule est la *classe* : le moule va définir la forme du gâteau. La classe contient donc le plan de fabrication d'un objet et on peut s'en servir autant qu'on veut afin d'obtenir une infinité d'objets.

Concrètement, une classe, c'est quoi ?

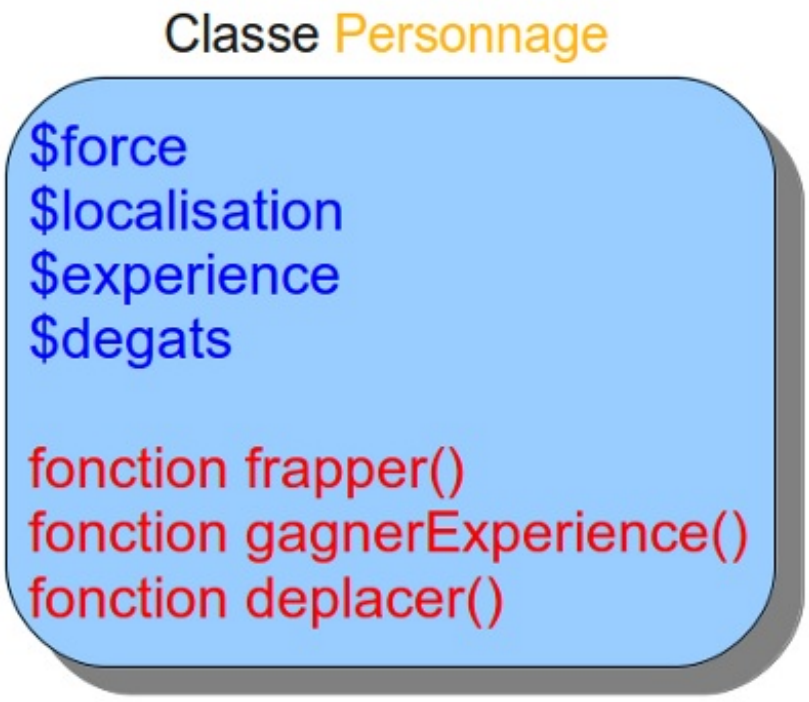
Une classe est une entité regroupant des variables et des fonctions. Chacune de ces fonctions aura accès aux variables de cette entité. Dans le cas du personnage, nous aurons une fonction `frapper()`. Cette fonction devra simplement modifier la variable `$degats` du personnage en fonction de la variable `$force`. Une classe est donc un regroupement logique de variables et fonctions que tout objet issu de cette classe possédera.

Définition d'une instance

Une instance, c'est tout simplement le résultat d'une *instanciation*. Une *instanciation*, c'est le fait d'*instancier* une classe. *Instancier* une classe, c'est se servir d'une classe afin qu'elle nous crée un objet. En gros, une instance est un objet.

Exemple : création d'une classe

Nous allons créer une classe `Personnage` (sous forme de schéma bien entendu). Celle-ci doit contenir la liste des variables et des fonctions que l'on a citées plus haut : c'est la base de tout objet `Personnage`. Chaque instance de cette classe possédera ainsi toutes ces variables et fonctions. Voici donc cette fameuse classe à la figure suivante.



Le schéma de notre classe

Vous voyez donc les variables et fonctions stockées dans la classe `Personnage`. Sachez qu'en réalité, on ne les appelle pas comme ça : il s'agit d'**attributs** (ou propriétés) et de **méthodes**. Un attribut désigne une variable et une méthode désigne une fonction.

Ainsi, tout objet `Personnage` aura ces attributs et méthodes. On pourra modifier ces attributs et invoquer ces méthodes sur notre objet afin de modifier ses caractéristiques ou son comportement.

Le principe d'encapsulation

L'un des gros avantages de la POO est que l'on peut masquer le code à l'utilisateur (l'utilisateur est ici celui qui se servira de la classe, pas celui qui chargera la page depuis son navigateur). Le concepteur de la classe a englobé dans celle-ci un code qui peut être assez complexe et il est donc inutile voire dangereux de laisser l'utilisateur manipuler ces objets sans aucune restriction. Ainsi, il est important d'interdire à l'utilisateur de modifier directement les attributs d'un objet.

Prenons l'exemple d'un avion où sont disponibles des centaines de boutons. Chacun de ces boutons constituent des actions que l'on peut effectuer sur l'avion. C'est l'*interface* de l'avion. Le pilote se moque de quoi est composé l'avion : son rôle est de le piloter. Pour cela, il va se servir des boutons afin de manipuler les composants de l'avion. Le pilote ne doit pas se charger de modifier manuellement ces composants : il pourrait faire de grosses bêtises.

Le principe est exactement le même pour la POO : l'utilisateur de la classe doit se contenter d'invoquer les méthodes en ignorant les attributs. Comme le pilote de l'avion, il n'a pas à les trifouiller. Pour instaurer une telle contrainte, on dit que les attributs sont **privés**. Pour l'instant, ceci peut sans doute vous paraître abstrait, mais nous y reviendrons. 

Bon, je pense que j'ai assez parlé, commençons par créer notre première classe !

Créer une classe

Syntaxe de base

Le but de cette section va être de traduire la figure précédente en code PHP. Avant cela, je vais vous donner la syntaxe de base de toute classe en PHP :

Code : PHP

```php
<?php
class Personnage // Présence du mot-clé class suivi du nom de la classe.
{
  // Déclaration des attributs et méthodes ici.
}
?>
```

Cette syntaxe est à retenir absolument. Heureusement, elle est simple.

Ce qu'on vient de faire est donc de créer le moule, le plan qui définira nos objets. On verra dans le prochain chapitre comment utiliser ce plan afin de créer un objet. Pour l'instant, contentons-nous de construire ce plan et de lui ajouter des fonctionnalités.

La déclaration d'attributs dans une classe se fait en écrivant le nom de l'attribut à créer, précédé de sa **visibilité**.

Visibilité d'un attribut ou d'une méthode

La visibilité d'un attribut ou d'une méthode indique à partir d'où on peut y avoir accès. Nous allons voir ici deux types de visibilité : `public` et `private`.

Le premier, `public`, est le plus simple. Si un attribut ou une méthode est `public`, alors on pourra y avoir accès depuis n'importe où, depuis l'intérieur de l'objet (dans les méthodes qu'on a créées), comme depuis l'extérieur. Je m'explique. Quand on crée un objet, c'est principalement pour pouvoir exploiter ses attributs et méthodes. L'extérieur de l'objet, c'est tout le code qui n'est pas *dans* votre classe. En effet, quand vous créerez un objet, cet objet sera représenté par une variable, et c'est à partir d'elle qu'on pourra modifier l'objet, appeler des méthodes, etc. Vous allez donc dire à PHP « dans cet objet, donne-moi cet attribut » ou « dans cet objet, appelle cette méthode » : c'est ça, appeler des attributs ou méthodes depuis l'extérieur de l'objet.

Le second, `private`, impose quelques restrictions. On n'aura accès aux attributs et méthodes *seulement* depuis l'intérieur de la classe, c'est-à-dire que seul le code voulant accéder à un attribut privé ou une méthode privée écrit(e) *à l'intérieur* de la classe fonctionnera. Sinon, une jolie erreur fatale s'affichera disant que vous ne pouvez pas accéder à telle méthode ou tel attribut parce qu'il ou elle est privé(e).

Là, ça devrait faire *tilt* dans votre tête : le principe d'encapsulation ! C'est de cette manière qu'on peut interdire l'accès à nos attributs.

Création d'attributs

Pour déclarer des attributs, on va donc les écrire entre les accolades, les uns à la suite des autres, en faisant précéder leurs noms du mot-clé `private`, comme ça :

Code : PHP

```php
<?php
class Personnage
{
  private $_force;         // La force du personnage
  private $_localisation; // Sa localisation
  private $_experience;   // Son expérience
  private $_degats;        // Ses dégâts
}
?>
```

Vous pouvez constater que chaque attribut est précédé d'un underscore (« _ »). Ceci est une notation qu'il est préférable de respecter (il s'agit de la notation PEAR) qui dit que chaque nom d'élément privé (ici il s'agit d'attributs, mais nous verrons plus tard qu'il peut aussi s'agir de méthodes) doit être précédé d'un underscore.

Vous pouvez initialiser les attributs lorsque vous les déclarez (par exemple, leur mettre une valeur de 0 ou autre). Exemple :

Code : PHP

```php
<?php
class Personnage
{
  private $_force = 50;             // La force du personnage, par
défaut à 50.
  private $_localisation = 'Lyon'; // Sa localisation, par défaut à
Lyon.
  private $_experience = 1;         // Son expérience, par défaut à
1.
  private $_degats = 0;             // Ses dégâts, par défaut à 0.
}
?>
```

La valeur que vous leur donnez par défaut doit être une expression constante. Par conséquent, leur valeur ne peut être issue d'un appel à une fonction (`private $_attribut = intval('azerty')`), d'une opération (`private $_attribut = 1 + 1`), d'une concaténation (`private $_attribut = 'Mon ' . 'super ' . 'attribut'`) ou d'une variable, superglobale ou non (`private $_attribut = $_SERVER['REQUEST_URI']`).

Création de méthodes

Pour la déclaration de méthodes, il suffit de faire précéder le mot-clé `function` à la visibilité de la méthode. Les types de visibilité des méthodes sont les mêmes que les attributs. Les méthodes n'ont en général pas besoin d'être masquées à l'utilisateur, vous les mettrez souvent en `public` (à moins que vous teniez absolument à ce que l'utilisateur ne puisse pas appeler cette méthode, par exemple s'il s'agit d'une fonction qui simplifie certaines tâches sur l'objet mais qui ne doit pas être appelée n'importe comment).

Code : PHP

```php
<?php
class Personnage
{
  private $_force;         // La force du personnage
  private $_localisation; // Sa localisation
```

```
  private $_experience;    // Son expérience
  private $_degats;        // Ses dégâts

  public function deplacer() // Une méthode qui déplacera le
personnage (modifiera sa localisation).
  {

  }

  public function frapper() // Une méthode qui frappera un
personnage (suivant la force qu'il a).
  {

  }

  public function gagnerExperience() // Une méthode augmentant
l'attribut $experience du personnage.
  {

  }
}
?>
```

Et voilà.

De même que l'underscore précédant les noms d'éléments privés, vous pouvez remarquer que le nom des classes commence par une majuscule. Il s'agit aussi du respect de la notation PEAR.

En résumé

- Une classe, c'est un ensemble de variables et de fonctions (attributs et méthodes).
- Un objet, c'est une *instance* de la classe pour pouvoir l'utiliser.
- Tous vos attributs doivent être privés. Pour les méthodes, peu importe leur visibilité. C'est ce qu'on appelle le principe d'encapsulation.
- On déclare une classe avec le mot-clé `class` suivi du nom de la classe, et enfin deux accolades ouvrantes et fermantes qui encercleront la liste des attributs et méthodes.

Utiliser la classe

Une classe, c'est bien beau mais, au même titre qu'un plan de construction pour une maison, si on ne sait pas comment se servir de notre plan pour construire notre maison, cela ne sert à rien ! Nous verrons donc ainsi comment se servir de notre **classe**, notre modèle de base, afin de créer des **objets** et pouvoir s'en servir.

Ce chapitre sera aussi l'occasion d'approfondir un peu le développement de nos classes : actuellement, la classe que l'on a créée contient des méthodes vides, chose un peu inutile vous avouerez. Pas de panique, on va s'occuper de tout ça !

Créer et manipuler un objet

Créer un objet

Nous allons voir comment créer un objet, c'est-à-dire que nous allons utiliser notre classe afin qu'elle nous fournisse un objet. Pour créer un nouvel objet, vous devez faire précéder le nom de la classe à instancier du mot-clé `new`, comme ceci :

Code : PHP

```php
<?php
$perso = new Personnage();
?>
```

Ainsi, `$perso` sera un objet de type `Personnage`. On dit que l'on *instancie* la classe `Personnage`, que l'on crée une instance de la classe `Personnage`.

Appeler les méthodes de l'objet

Pour appeler une méthode d'un objet, il va falloir utiliser un opérateur : il s'agit de l'opérateur `->` (une flèche composée d'un tiret suivi d'un chevron fermant). Celui-ci s'utilise de la manière suivante. À gauche de cet opérateur, on place **l'objet** que l'on veut utiliser. Dans l'exemple pris juste au-dessus, cet objet aurait été `$perso`. À droite de l'opérateur, on spécifie le nom de la méthode que l'on veut invoquer.

Et puisqu'un exemple vaut mieux qu'un long discours…

Code : PHP

```php
<?php
// Nous créons une classe « Personnage ».
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Nous déclarons une méthode dont le seul but est d'afficher un
texte.
  public function parler()
  {
    echo 'Je suis un personnage !';
  }
}

$perso = new Personnage();
$perso->parler();
```

La ligne 18 signifie donc « va chercher l'objet `$perso`, et invoque la méthode `parler()` sur cet objet ». Notez donc bien quelque part l'utilisation de cet opérateur : de manière générale, il sert à accéder à un élément de la classe. Ici, on s'en est servi pour atteindre une méthode, mais nous verrons plus tard qu'il nous permettra aussi d'atteindre un attribut.

Accéder à un élément depuis la classe

Je viens de vous faire créer un objet, et vous êtes maintenant capables d'appeler une méthode. Cependant, le contenu de cette dernière était assez simpliste : elle ne faisait qu'afficher un message. Ici, nous allons voir comment une méthode peut accéder aux attributs de l'objet.

Lorsque vous avez un objet, vous savez que vous pouvez invoquer des méthodes grâce à l'opérateur « -> ». Je vous ai par ailleurs dit que c'était également grâce à cet opérateur qu'on accédait aux attributs de la classe. Cependant, rappelez-vous : nos attributs sont privés. Par conséquent, ce code lèvera une erreur fatale :

Code : PHP

```php
<?php
class Personnage
{
  private $_force;
  private $_experience;
  private $_degats;
}

$perso = new Personnage();
$perso->_experience = $perso->_experience + 1; // Une erreur fatale est levée suite à cette instruction.
```

Ici, on essaye d'accéder à un attribut privé hors de la classe. Ceci est interdit, donc PHP lève une erreur. Dans notre exemple (qui essaye en vain d'augmenter de 1 l'expérience du personnage), il faudra demander à la classe d'augmenter l'expérience. Pour cela, nous allons écrire une méthode `gagnerExperience()` :

Code : PHP

```php
<?php
class Personnage
{
  private $_experience;

  public function gagnerExperience()
  {
    // Cette méthode doit ajouter 1 à l'expérience du personnage.
  }
}

$perso = new Personnage();
$perso->gagnerExperience();
```

Oui mais voilà : comment accéder à l'attribut `$_experience` dans notre méthode ? C'est là qu'intervient la pseudo-variable `$this`.

Dans notre script, `$perso` représente l'objet. Il est donc facile d'appeler une méthode à partir de cette variable. Mais dans notre méthode, nous n'avons pas cette variable pour accéder à notre attribut `$_experience` pour le modifier ! Du moins, c'est ce que vous croyez. En fait, un paramètre représentant l'objet est passé implicitement à chaque méthode de la classe. Regardez plutôt cet exemple :

Code : PHP

```php
<?php
class Personnage
{
  private $_experience = 50;
```

```
  public function afficherExperience()
  {
    echo $this->_experience;
  }
}

$perso = new Personnage();
$perso->afficherExperience();
```

Commentons la ligne 8. Vous avez sans doute reconnu la structure `echo` ayant pour rôle d'afficher une valeur. Ensuite, vous reconnaissez la variable `$this` dont je vous ai parlé : elle représente l'objet que nous sommes en train d'utiliser. Ainsi, dans ce script, *les variables `$this` et `$perso` représentent le même objet*. L'instruction surlignée veut donc dire : « Affiche-moi cette valeur : dans l'objet utilisé (donc `$perso`), donne-moi la valeur de l'attribut `$_experience`. »

Ainsi, je vais vous demander d'écrire la méthode `gagnerExperience()`. Cette méthode devra ajouter 1 à l'attribut `$_experience`. Je suis sûr que vous pouvez y arriver !

Voici la correction :

Code : PHP

```
<?php
class Personnage
{
  private $_experience = 50;

  public function afficherExperience()
  {
    echo $this->_experience;
  }

  public function gagnerExperience()
  {
    // On ajoute 1 à notre attribut $_experience.
    $this->_experience = $this->_experience + 1;
  }
}

$perso = new Personnage();
$perso->gagnerExperience();   // On gagne de l'expérience.
$perso->afficherExperience(); // On affiche la nouvelle valeur de
l'attribut.
```

Implémenter d'autres méthodes

Nous avons créé notre classe Personnage et déjà une méthode, `gagnerExperience()`. J'aimerais qu'on en implémente une autre : la méthode `frapper()`, qui devra infliger des dégâts à un personnage.

Pour partir sur une base commune, nous allons travailler sur cette classe :

Code : PHP

```
<?php
class Personnage
{
  private $_degats = 0; // Les dégâts du personnage.
  private $_experience = 0; // L'expérience du personnage.
  private $_force = 20; // La force du personnage (plus elle est
grande, plus l'attaque est puissante).
```

```
  public function gagnerExperience()
  {
    // On ajoute 1 à notre attribut $_experience.
    $this->_experience = $this->_experience + 1;
  }
}
```

Commençons par écrire notre méthode `frapper()`. Cette méthode doit accepter un argument : le personnage à frapper. La méthode aura juste pour rôle d'augmenter les dégâts du personnage passé en paramètre.

Pour vous aider à visualiser le contenu de la méthode, imaginez votre code manipulant des objets. Il doit ressembler à ceci :

Code : PHP

```
<?php
// On crée deux personnages
$perso1 = new Personnage();
$perso2 = new Personnage();

// Ensuite, on veut que le personnage n°1 frappe le personnage n°2.
$perso1->frapper($perso2);
```

Pour résumer :

- la méthode `frapper()` demande un argument : le personnage à frapper ;
- cette méthode augmente les dégâts du personnage à frapper en fonction de la force du personnage qui frappe.

On pourrait donc imaginer une classe ressemblant à ceci :

Code : PHP

```
<?php
class Personnage
{
  private $_degats; // Les dégâts du personnage.
  private $_experience; // L'expérience du personnage.
  private $_force; // La force du personnage (plus elle est grande,
plus l'attaque est puissante).

  public function frapper($persoAFrapper)
  {
    $persoAFrapper->_degats += $this->_force;
  }

  public function gagnerExperience()
  {
    // On ajoute 1 à notre attribut $_experience.
    $this->_experience = $this->_experience + 1;
  }
}
```

Commentons ensemble le contenu de cette méthode `frapper()`. Celle-ci comporte une instruction composée de deux parties :

1. La première consiste à dire à PHP que l'on veut assigner une nouvelle valeur à l'attribut `$_degats` du personnage à frapper.
2. La seconde partie consiste à donner à PHP la valeur que l'on veut assigner. Ici, nous voyons que cette valeur est atteinte par `$this->_force`.

Maintenant, grosse question : la variable `$this` fait-elle référence au personnage qui frappe ou au personnage frappé ? Pour répondre à cette question, il faut savoir sur quel objet est appelée la méthode. En effet, souvenez-vous que `$this` est une variable représentant l'objet à partir duquel on a appelé la méthode. Dans notre cas, on a appelé la méthode `frapper()` à partir du personnage qui frappe, donc `$this` représente le personnage qui frappe.

L'instruction contenue dans la méthode signifie donc : « Ajoute la valeur de la force du personnage qui frappe à l'attribut `$_degats` du personnage frappé. »

Maintenant, nous pouvons créer une sorte de petite mise en scène qui fait interagir nos personnages. Par exemple, nous pouvons créer un script qui fait combattre les personnages. Le personnage 1 frapperait le personnage 2 puis gagnerait de l'expérience, puis le personnage 2 frapperait le personnage 1 et gagnerait de l'expérience. Procédez étape par étape :

- créez deux personnages ;
- faites frapper le personnage 1 ;
- faites gagner de l'expérience au personnage 1 ;
- faites frapper le personnage 2 ;
- faites gagner de l'expérience au personnage 2.

Ce script n'est qu'une suite d'appels de méthodes. Chaque puce (sauf la première) correspond à l'appel d'une méthode, ce que vous savez faire. En conclusion, vous êtes aptes à créer ce petit script ! Voici la correction :

Code : PHP

```php
<?php
$perso1 = new Personnage(); // Un premier personnage
$perso2 = new Personnage(); // Un second personnage

$perso1->frapper($perso2); // $perso1 frappe $perso2
$perso1->gagnerExperience(); // $perso1 gagne de l'expérience

$perso2->frapper($perso1); // $perso2 frappe $perso1
$perso2->gagnerExperience(); // $perso2 gagne de l'expérience
?>
```

Cependant, un petit problème se pose. Puisque, à la base, les deux personnages ont le même niveau de dégâts, la même expérience et la même force, ils seront à la fin toujours égaux. Pour pallier ce problème, il faudrait pouvoir assigner des valeurs spécifiques aux deux personnages, afin que le combat puisse les différencier. Or, vous ne pouvez pas accéder aux attributs en-dehors de la classe ! Pour savoir comment résoudre ce problème, je vais vous apprendre deux nouveaux mots : **accesseur** et **mutateur**. Mais avant, j'aimerais faire une petite parenthèse.

Exiger des objets en paramètre

Reprenons l'exemple du code auquel nous sommes arrivés et concentrons-nous sur la méthode `frapper()`. Celle-ci accepte un argument : un personnage à frapper. Cependant, qu'est-ce qui vous garantit qu'on passe effectivement un personnage à frapper ? On pourrait très bien passer un argument complètement différent, comme un nombre par exemple :

Code : PHP

```php
<?php
$perso = new Personnage();
$perso->frapper(42);
?>
```

Et là, qu'est-ce qui se passe ? Une erreur est générée car, à l'intérieur de la méthode `frapper()`, nous essayons d'appeler une méthode sur le paramètre qui n'est pas un objet. C'est comme si on avait fait ça :

Code : PHP

```php
<?php
$persoAFrapper = 42;
$persoAFrapper->_degats += 50; // Le nombre 50 est arbitraire, il
est censé représenter une force.
?>
```

Ce qui n'a aucun sens. Il faut donc s'assurer que le paramètre passé est bien un personnage, sinon PHP arrête tout et n'exécute pas la méthode. Pour cela, il suffit d'ajouter un seul mot : le nom de la classe dont le paramètre doit être un objet. Dans notre cas, si le paramètre doit être un objet de type `Personnage`, alors il faudra ajouter le mot-clé `Personnage`, juste avant le nom du paramètre, comme ceci :

Code : PHP

```php
<?php
class Personnage
{
  // …

  public function frapper(Personnage $persoAFrapper)
  {
    // …
  }
}
```

Grâce à ça, vous êtes sûrs que la méthode `frapper()` ne sera exécutée *que* si le paramètre passé est de type `Personnage`, sinon PHP interrompt tout le script. Vous pouvez donc appeler les méthodes de l'objet sans crainte qu'un autre type de variable soit passé en paramètre.

Le type de la variable à spécifier doit obligatoirement être un nom de classe ou alors un tableau. Si vous voulez exiger un tableau, faites précéder le nom du paramètre devant être un tableau par le mot-clé *array* comme ceci : `public function frapper(array $coups)`. Vous ne pouvez pas exiger autre chose : par exemple, il est impossible d'exiger un nombre entier ou une chaîne de caractères de cette façon.

Les accesseurs et mutateurs

Comme vous le savez, le principe d'encapsulation nous empêche d'accéder directement aux attributs de notre objet puisqu'ils sont privés : seule la classe peut les lire et les modifier. Par conséquent, si vous voulez récupérer un attribut, il va falloir le demander à la classe, de même si vous voulez les modifier.

Accéder à un attribut : l'accesseur

À votre avis, comment peut-on faire pour récupérer la valeur d'un attribut ? La solution est simple : nous allons implémenter des méthodes dont le seul rôle sera de nous donner l'attribut qu'on leur demande ! Ces méthodes ont un nom bien spécial : ce sont des **accesseurs** (ou *getters*). Par convention, ces méthodes portent le même nom que l'attribut dont elles renvoient la valeur. Par exemple, voici la liste des accesseurs de notre classe `Personnage` :

Code : PHP

```php
<?php
class Personnage
{
  private $_force;
  private $_experience;
  private $_degats;

  public function frapper(Personnage $persoAFrapper)
  {
    $persoAFrapper->_degats += $this->_force;
  }
```

```
  public function gagnerExperience()
  {
    // Ceci est un raccourci qui équivaut à écrire « $this->_experience = $this->_experience + 1 »
    // On aurait aussi pu écrire « $this->_experience += 1 »
    $this->_experience++;
  }

  // Ceci est la méthode degats() : elle se charge de renvoyer le contenu de l'attribut $_degats.
  public function degats()
  {
    return $this->_degats;
  }

  // Ceci est la méthode force() : elle se charge de renvoyer le contenu de l'attribut $_force.
  public function force()
  {
    return $this->_force;
  }

  // Ceci est la méthode experience() : elle se charge de renvoyer le contenu de l'attribut $_experience.
  public function experience()
  {
    return $this->_experience;
  }
}
```

Modifier la valeur d'un attribut : les mutateurs

Maintenant, comment cela se passe-t-il si vous voulez modifier un attribut ? Encore une fois, il va falloir que vous demandiez à la classe de le faire pour vous. Je vous rappelle que le principe d'encapsulation est là pour vous empêcher d'assigner un mauvais type de valeur à un attribut : si vous demandez à votre classe de le faire, ce risque est supprimé car la classe « contrôle » la valeur des attributs. Comme vous l'aurez peut-être deviné, ce sera par le biais de méthodes que l'on demandera à notre classe de modifier tel attribut.

La classe doit impérativement contrôler la valeur afin d'assurer son intégrité car, si elle ne le fait pas, on pourra passer n'importe quelle valeur à la classe et le principe d'encapsulation n'est plus respecté ! Ces méthodes ont aussi un nom spécial : il s'agit de **mutateurs** (ou *setters*). Ces méthodes sont de la forme `setNomDeLAttribut()`. Voici la liste des mutateurs (ajoutée à la liste des accesseurs) de notre classe `Personnage` :

Code : PHP

```
<?php
class Personnage
{
  private $_force;
  private $_experience;
  private $_degats;

  public function frapper(Personnage $persoAFrapper)
  {
    $persoAFrapper->_degats += $this->_force;
  }

  public function gagnerExperience()
  {
    $this->_experience++;
  }

  // Mutateur chargé de modifier l'attribut $_force.
  public function setForce($force)
```

```
    {
      if (!is_int($force)) // S'il ne s'agit pas d'un nombre entier.
      {
        trigger_error('La force d\'un personnage doit être un nombre
entier', E_USER_WARNING);
        return;
      }

      if ($force > 100) // On vérifie bien qu'on ne souhaite pas
assigner une valeur supérieure à 100.
      {
        trigger_error('La force d\'un personnage ne peut dépasser
100', E_USER_WARNING);
        return;
      }

      $this->_force = $force;
    }

    // Mutateur chargé de modifier l'attribut $_experience.
    public function setExperience($experience)
    {
      if (!is_int($experience)) // S'il ne s'agit pas d'un nombre
entier.
      {
        trigger_error('L\'expérience d\'un personnage doit être un
nombre entier', E_USER_WARNING);
        return;
      }

      if ($experience > 100) // On vérifie bien qu'on ne souhaite pas
assigner une valeur supérieure à 100.
      {
        trigger_error('L\'expérience d\'un personnage ne peut dépasser
100', E_USER_WARNING);
        return;
      }

      $this->_experience = $experience;
    }

    // Ceci est la méthode degats() : elle se charge de renvoyer le
contenu de l'attribut $_degats.
    public function degats()
    {
      return $this->_degats;
    }

    // Ceci est la méthode force() : elle se charge de renvoyer le
contenu de l'attribut $_force.
    public function force()
    {
      return $this->_force;
    }

    // Ceci est la méthode experience() : elle se charge de renvoyer
le contenu de l'attribut $_experience.
    public function experience()
    {
      return $this->_experience;
    }
}
```

Voilà ce que j'avais à dire concernant ces accesseurs et mutateurs. Retenez bien ces définitions, vous les trouverez dans la plupart des classes !

Retour sur notre script de combat

Maintenant que nous avons vu ce qu'étaient des accesseurs et des mutateurs, nous pouvons améliorer notre script de combat. Pour commencer, je vais vous demander d'afficher, à la fin du script, la force, l'expérience et le niveau de dégâts de chaque personnage.

Voici la correction :

Code : PHP

```php
<?php
$perso1 = new Personnage();  // Un premier personnage
$perso2 = new Personnage();  // Un second personnage

$perso1->frapper($perso2);  // $perso1 frappe $perso2
$perso1->gagnerExperience(); // $perso1 gagne de l'expérience

$perso2->frapper($perso1);  // $perso2 frappe $perso1
$perso2->gagnerExperience(); // $perso2 gagne de l'expérience

echo 'Le personnage 1 a ', $perso1->force(), ' de force,
contrairement au personnage 2 qui a ', $perso2->force(), ' de
force.<br />';
echo 'Le personnage 1 a ', $perso1->experience(), ' d\'expérience,
contrairement au personnage 2 qui a ', $perso2->experience(), '
d\'expérience.<br />';
echo 'Le personnage 1 a ', $perso1->degats(), ' de dégâts,
contrairement au personnage 2 qui a ', $perso2->degats(), ' de
dégâts.<br />';
```

Comme nous l'avions dit, les valeurs finales des deux personnages sont identiques. Pour pallier ce problème, nous allons modifier, juste après la création des personnages, la valeur de la force et de l'expérience des deux personnages. Vous pouvez par exemple favoriser un personnage en lui donnant une plus grande force et une plus grande expérience par rapport au deuxième.

Voici la correction que je vous propose (peu importe les valeurs que vous avez choisies, l'essentiel est que vous ayez appelé les bonnes méthodes) :

Code : PHP

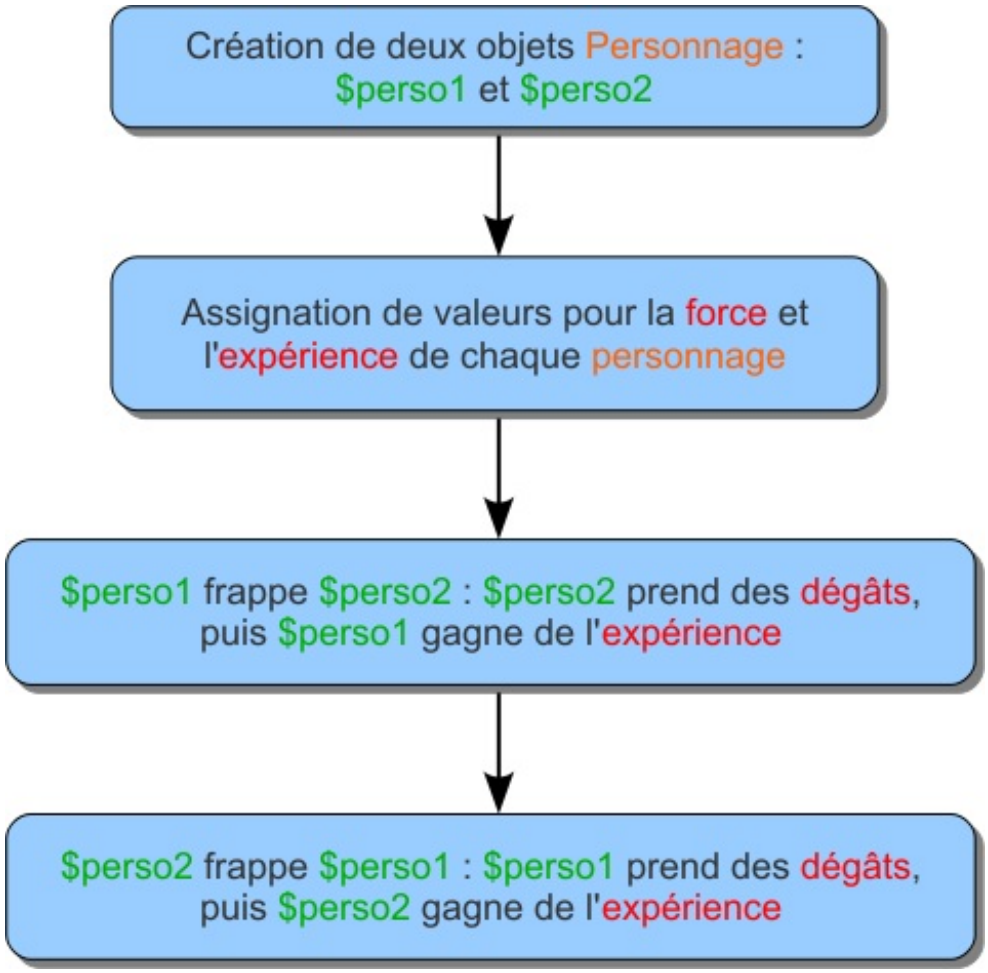
```php
<?php
$perso1 = new Personnage(); // Un premier personnage
$perso2 = new Personnage(); // Un second personnage

$perso1->setForce(10);
$perso1->setExperience(2);

$perso2->setForce(90);
$perso2->setExperience(58);

$perso1->frapper($perso2);  // $perso1 frappe $perso2
$perso1->gagnerExperience(); // $perso1 gagne de l'expérience

$perso2->frapper($perso1);  // $perso2 frappe $perso1
$perso2->gagnerExperience(); // $perso2 gagne de l'expérience

echo 'Le personnage 1 a ', $perso1->force(), ' de force,
contrairement au personnage 2 qui a ', $perso2->force(), ' de
force.<br />';
echo 'Le personnage 1 a ', $perso1->experience(), ' d\'expérience,
contrairement au personnage 2 qui a ', $perso2->experience(), '
d\'expérience.<br />';
echo 'Le personnage 1 a ', $perso1->degats(), ' de dégâts,
contrairement au personnage 2 qui a ', $perso2->degats(), ' de
dégâts.<br />';
```

Ce qui affichera :

Résultat affiché par le script

Comme vous le voyez, à la fin, les deux personnages n'ont plus les mêmes caractéristiques !

Pour bien être sûr que vous me suiviez toujours, je vous ai fait un schéma résumant le déroulement du script.



Déroulement du script

Le constructeur

Vous vous demandez peut-être à quoi servent les parenthèses juste après `Personnage` lorsque vous créez un objet ? C'est ce que je vais vous expliquer juste après vous avoir expliqué ce qu'est un constructeur.

Le constructeur est la méthode appelée dès que vous créez l'objet avec la technique présentée ci-dessus. Cette méthode peut demander des paramètres, auquel cas nous devrons les placer entre les parenthèses que vous voyez après le nom de la classe.

Effectuons un retour sur notre classe `Personnage`. Ajoutons-lui un constructeur. Ce dernier ne peut pas avoir n'importe quel nom (sinon, comment PHP sait quel est le constructeur ?). Il a tout simplement le nom suivant : `__construct`, avec deux underscores au début.

Comme son nom l'indique, le constructeur sert à *construire* l'objet. Ce que je veux dire par là, c'est que si des attributs doivent être initialisés ou qu'une connexion à la BDD doit être faite, c'est par ici que ça se passe. Comme dit plus haut, le constructeur est exécuté *dès la création de l'objet* et par conséquent, aucune valeur ne doit être retournée, même si ça ne génèrera aucune erreur. Bien sûr, et comme nous l'avons vu, une classe fonctionne très bien sans constructeur, il n'est en rien obligatoire ! Si vous n'en spécifiez pas, cela revient au même que si vous en aviez écrit un vide (sans instruction à l'intérieur).

Code : PHP

```php
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  public function __construct($force, $degats) // Constructeur demandant 2 paramètres
  {
    echo 'Voici le constructeur !'; // Message s'affichant une fois que tout objet est créé.
    $this->setForce($force); // Initialisation de la force.
    $this->setDegats($degats); // Initialisation des dégâts.
    $this->_experience = 1; // Initialisation de l'expérience à 1.
  }

  // Mutateur chargé de modifier l'attribut $_force.
  public function setForce($force)
  {
    if (!is_int($force)) // S'il ne s'agit pas d'un nombre entier.
    {
      trigger_error('La force d\'un personnage doit être un nombre entier', E_USER_WARNING);
      return;
    }

    if ($force > 100) // On vérifie bien qu'on ne souhaite pas assigner une valeur supérieure à 100.
    {
      trigger_error('La force d\'un personnage ne peut dépasser 100', E_USER_WARNING);
      return;
    }

    $this->_force = $force;
  }

  // Mutateur chargé de modifier l'attribut $_degats.
  public function setDegats($degats)
  {
    if (!is_int($degats)) // S'il ne s'agit pas d'un nombre entier.
    {
      trigger_error('Le niveau de dégâts d\'un personnage doit être un nombre entier', E_USER_WARNING);
      return;
    }

    $this->_degats = $degats;
  }
}
?>
```

Notez que je n'ai pas réécrit toutes les méthodes, ce n'est pas le but de ce que je veux vous montrer ici.

Ici, le constructeur demande la force et les dégâts initiaux du personnage que l'on vient de créer. Il faudra donc lui spécifier ceci en paramètre :

Code : PHP

```php
<?php
$perso1 = new Personnage(60, 0); // 60 de force, 0 dégât
$perso2 = new Personnage(100, 10); // 100 de force, 10 dégâts
?>
```

Et à la sortie s'affichera à la suite :

Résultat affiché par le script

Notez quelque chose d'important dans la classe : dans le constructeur, les valeurs sont initialisées en appelant les mutateurs correspondant. En effet, si on assignait directement ces valeurs avec les arguments, le principe d'encapsulation ne serait plus respecté et n'importe quel type de valeur pourrait être assigné !

Ne mettez jamais la méthode `__construct` avec le type de visibilité `private` car elle ne pourra jamais être appelée, vous ne pourrez donc pas *instancier* votre classe ! Cependant, sachez qu'il existe certains cas particuliers qui nécessitent le constructeur en privé, mais ce n'est pas pour tout de suite.

Notez que si la classe n'a pas implémenté de constructeur ou si le constructeur ne requiert aucun argument, alors les parenthèses placées après le nom de la classe lorsque vous l'instancierez sont inutiles. Ainsi, vous pourrez faire `$classe = new MaClasse;`. C'est d'ailleurs sans parenthèses que j'instancierai les classes sans constructeur ou avec constructeur sans argument désormais.

L'auto-chargement de classes

Pour une question d'organisation, il vaut mieux créer un fichier par classe. Vous appelez votre fichier comme bon vous semble et placez votre classe dedans. Pour ma part, mes fichiers sont toujours appelés « MaClasse.class.php ». Ainsi, si je veux pouvoir utiliser la classe `MaClasse`, je n'aurais qu'à inclure ce fichier :

Code : PHP

```php
<?php
require 'MaClasse.class.php'; // J'inclus la classe.

$objet = new MaClasse(); // Puis, seulement après, je me sers de ma classe.
```

Maintenant, imaginons que vous ayez plusieurs dizaines de classes… Pas très pratique de les inclure une par une ! Vous vous retrouverez avec des dizaines d'inclusions, certaines pouvant même être inutile si vous ne vous servez pas de toutes vos classes. Et c'est là qu'intervient l'auto-chargement des classes. Vous pouvez créer dans votre fichier principal (c'est-à-dire celui où vous créerez une instance de votre classe) une ou plusieurs fonction(s) qui tenteront de charger le fichier déclarant la classe. Dans la plupart des cas, une seule fonction suffit. Ces fonctions doivent accepter un paramètre, c'est le nom de la classe qu'on doit tenter de charger. Par exemple, voici une fonction qui aura pour rôle de charger les classes :

Code : PHP

```php
<?php
function chargerClasse($classe)
{
  require $classe . '.class.php'; // On inclut la classe correspondante au paramètre passé.
}
?>
```

Essayons maintenant de créer un objet pour voir si il sera chargé automatiquement (je prends pour exemple la classe `Personnage` et prends en compte le fait qu'un fichier `Personnage.class.php` existe).

Code : PHP

```php
<?php
function chargerClasse($classe)
{
  require $classe . '.class.php'; // On inclut la classe correspondante au paramètre passé.
}

$perso = new Personnage(); // Instanciation de la classe Personnage qui n'est pas déclarée dans ce fichier.
?>
```

Et là… Bam ! Erreur fatale ! La classe n'a pas été trouvée, elle n'a donc pas été chargée… Normal quand on y réfléchi ! PHP ne sait pas qu'il doit appeler cette fonction lorsqu'on essaye d'instancier une classe non déclarée. On va donc utiliser la fonction `spl_autoload_register` en spécifiant en premier paramètre le nom de la fonction à charger :

Code : PHP

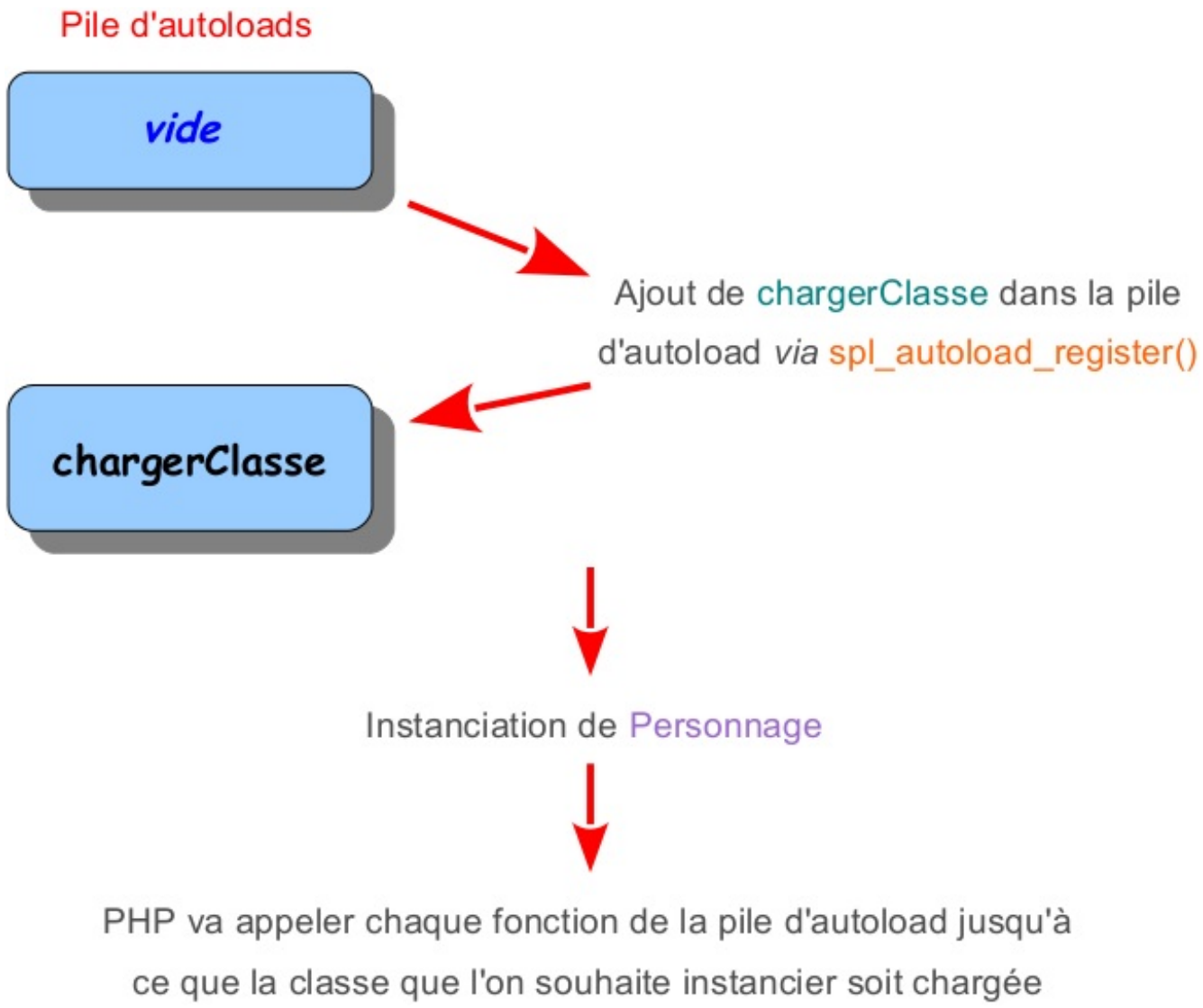
```php
<?php
function chargerClasse($classe)
{
  require $classe . '.class.php'; // On inclut la classe correspondante au paramètre passé.
}

spl_autoload_register('chargerClasse'); // On enregistre la fonction en autoload pour qu'elle soit appelée dès qu'on instanciera une classe non déclarée.

$perso = new Personnage();
?>
```

Et là, comme par magie, aucune erreur ne s'affiche ! Notre auto-chargement a donc bien fonctionné.

Décortiquons ce qui s'est passé. En PHP, il y a ce qu'on appelle une « pile d'autoloads ». Cette pile contient une liste de fonctions. Chacune d'entre elles sera appelée automatiquement par PHP lorsque l'on essaye d'instancier une classe non déclarée.

Nous avons donc ici ajouté notre fonction à la pile d'autoloads afin qu'elle soit appelée à chaque fois qu'on essaye d'instancier une classe non déclarée.

Schématiquement, la figure suivante montre ce qui s'est passé.



L'autoload en PHP

Sachez que vous pouvez enregistrer autant de fonctions en autoload que vous le voulez avec `spl_autoload_register`. Si vous en enregistrez plusieurs, elles seront appelées dans l'ordre de leur enregistrement jusqu'à ce que la classe soit chargée. Pour y parvenir, il suffit d'appeler `spl_autoload_register` pour chaque fonction à enregistrer.

Voici un chapitre aussi essentiel que le premier et toujours aussi riche en nouveautés fondant les bases de la POO. Prenez bien le temps de lire et relire ce chapitre si vous êtes un peu perdus, sinon vous ne pourrez jamais suivre !

En résumé

- Un objet se crée grâce à l'opérateur `new`.
- L'accès à un attribut ou à une méthode d'un objet se fait grâce à l'opérateur « `->` ».
- Pour lire ou modifier un attribut, on utilise des *accesseurs* et des *mutateurs*.
- Le constructeur d'une classe a pour rôle principal d'initialiser l'objet en cours de création, c'est-à-dire d'initialiser la valeur des attributs (soit en assignant directement des valeurs spécifiques, soit en appelant diverses méthodes).
- Les classes peuvent être chargées dynamiquement (c'est-à-dire sans avoir explicitement inclus le fichier la déclarant) grâce à l'auto-chargement de classe (utilisation de `spl_autoload_register`).

L'opérateur de résolution de portée

L'opérateur de résolution de portée (« `::` »), appelé « double deux points » (« *Scope Resolution Operator* » en anglais), est utilisé pour appeler des éléments appartenant à telle classe et non à tel objet. En effet, nous pouvons définir des attributs et méthodes appartenant à la classe : ce sont des éléments **statiques**. Nous y reviendrons en temps voulu dans une partie dédiée à ce sujet.

Parmi les éléments appartenant à la classe (et donc appelés via cet opérateur), il y a aussi les **constantes de classe**, sortes d'attributs dont la valeur est constante, c'est-à-dire qu'elle ne change pas. Nous allons d'ailleurs commencer par ces constantes de classe.

Cet opérateur est aussi appelé « Paamayim Nekudotayim ». Mais rassurez-vous, je ne vais pas vous demander de le retenir (si vous y arrivez, bien joué).

Les constantes de classe

Commençons par les constantes de classe. Le principe est à peu près le même que lorsque vous créez une constante à l'aide de la fonction `define`. Les constantes de classe permettent d'éviter tout code *muet*. Voici un code muet :

Code : PHP

```php
<?php
$perso = new Personnage(50);
?>
```

Pourquoi est-il muet ? Tout simplement parce qu'on ne sait pas à quoi « 50 » correspond. Qu'est-ce que cela veut dire ? Étant donné que je viens de réaliser le script, je sais que ce « 50 » correspond à la force du personnage. Cependant, ce paramètre ne peut prendre que 3 valeurs possibles :

- 20, qui veut dire que le personnage aura une faible force ;
- 50, qui veut dire que le personnage aura une force moyenne ;
- 80, qui veut dire que le personnage sera très fort.

Au lieu de passer ces valeurs telles quelles, on va plutôt passer une **constante** au constructeur. Ainsi, quand on lira le code, on devinera facilement que l'on passe une force moyenne au constructeur. C'est bien plus facile à comprendre qu'un nombre quelconque.

Une constante est une sorte d'attribut appartenant à la classe dont la valeur ne change jamais. Ceci est peut-être un peu flou, c'est pourquoi nous allons passer à la pratique. Pour déclarer une constante, vous devez faire précéder son nom du mot-clé `const`. Faites bien attention, une constante ne prend pas de « $ » devant son nom ! Voici donc comment créer une constante :

Code : PHP

```php
<?php
class Personnage
{
  // Je rappelle : tous les attributs en privé !

  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Déclarations des constantes en rapport avec la force.

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;

  public function __construct()
  {

  }
```

```
  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }
}
?>
```

Et voilà ! Facile n'est-ce pas ?

Bien sûr, vous pouvez assigner à ces constantes d'autres valeurs.

Et quel est le rapport avec tes « double deux points » ?

Contrairement aux attributs, vous ne pouvez accéder à ces valeurs *via* l'opérateur « `->` » depuis un objet (ni `$this` ni `$perso` ne fonctionneront) mais avec l'opérateur « `::` » car une constante appartient à la classe et non à un quelconque objet.

Pour accéder à une constante, vous devez spécifier le nom de la classe, suivi du symbole double deux points, suivi du nom de la constante. Ainsi, on pourrait imaginer un code comme celui-ci :

Code : PHP

```
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Déclarations des constantes en rapport avec la force.

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;

  public function __construct($forceInitiale)
  {
    // N'oubliez pas qu'il faut assigner la valeur d'un attribut
uniquement depuis son setter !
    $this->setForce($forceInitiale);
  }

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
```

```
  {

  }

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit une « FORCE_PETITE »,
soit une « FORCE_MOYENNE », soit une « FORCE_GRANDE ».
    if (in_array($force, array(self::FORCE_PETITE,
self::FORCE_MOYENNE, self::FORCE_GRANDE)))
    {
      $this->_force = $force;
    }
  }
}
?>
```

Et lors de la création de notre personnage :

Code : PHP

```
<?php
// On envoie une « FORCE_MOYENNE » en guise de force initiale.
$perso = new Personnage(Personnage::FORCE_MOYENNE);
?>
```

Notez qu'ici les noms de constantes sont en majuscules : c'est encore et toujours une convention de dénomination.

Reconnaissez que ce code est plus lisible que celui montré au début de cette sous-partie. 

Les attributs et méthodes statiques

Les méthodes statiques

Comme je l'ai brièvement dit dans l'introduction, les méthodes statiques sont des méthodes qui sont faites pour agir sur une classe et non sur un objet. Par conséquent, je ne veux voir aucun `$this` dans la méthode ! En effet, la méthode n'étant appelée sur aucun objet, il serait illogique que cette variable existe. Souvenez-vous : `$this` est un paramètre implicite. Cela n'est vrai que pour les méthodes appelées sur un objet !

Même si la méthode est dite « statique », il est possible de l'appeler depuis un objet (`$obj->methodeStatique()`), mais, même dans ce contexte, la variable `$this` ne sera toujours pas passée !

Pour déclarer une méthode statique, vous devez faire précéder le mot-clé `function` du mot-clé `static` après le type de visibilité.

Code : PHP

```
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;
```

```php
  public function __construct($forceInitiale)
  {
    $this->setForce($forceInitiale);
  }

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit une « FORCE_PETITE », soit une « FORCE_MOYENNE », soit une « FORCE_GRANDE ».
    if (in_array($force, array(self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE)))
    {
      $this->_force = $force;
    }
  }

  // Notez que le mot-clé static peut être placé avant la visibilité de la méthode (ici c'est public).
  public static function parler()
  {
    echo 'Je vais tous vous tuer !';
  }
}
?>
```

Et dans le code, vous pourrez faire :

Code : PHP

```php
<?php
Personnage::parler();
?>
```

Comme je l'ai dit plus haut, vous pouvez aussi appeler la méthode depuis un objet, mais cela ne changera rien au résultat final :

Code : PHP

```php
<?php
$perso = new Personnage(Personnage::FORCE_GRANDE);
$perso->parler();
?>
```

Cependant, préférez appeler la méthode avec l'opérateur « `::` » comme le montre le premier de ces deux codes. De cette façon, on voit directement de quelle classe on décide d'invoquer la méthode. De plus, appeler de cette façon une méthode statique évitera

une erreur de degré `E_STRICT`.

Je me répète mais j'insiste là-dessus : il n'y a pas de variable `$this` dans la méthode dans la mesure où la méthode est invoquée afin d'agir sur la classe et non sur un quelconque objet !

Les attributs statiques

Le principe est le même, c'est-à-dire qu'un attribut statique appartient à la classe et non à un objet. Ainsi, tous les objets auront accès à cet attribut et cet attribut aura la même valeur pour tous les objets.

La déclaration d'un attribut statique se fait en faisant précéder son nom du mot-clé `static`, comme ceci :

Code : PHP

```php
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  const FORCE_PETITE = 20;
  const FORCE_MOYENNE = 50;
  const FORCE_GRANDE = 80;

  // Variable statique PRIVÉE.
  private static $_texteADire = 'Je vais tous vous tuer !';

  public function __construct($forceInitiale)
  {
    $this->setForce($forceInitiale);
  }

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit un « FORCE_PETITE », soit une « FORCE_MOYENNE », soit une « FORCE_GRANDE ».
    if (in_array($force, array(self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE)))
    {
      $this->_force = $force;
    }
  }

  public static function parler()
  {
    echo self::$_texteADire; // On donne le texte à dire.
  }
}
```

```
?>
```

Quelques nouveautés dans ce code nécessitent des explications. Premièrement, à quoi sert un attribut statique ?

Nous avons vu que les méthodes statiques sont faites pour agir sur la classe. D'accord, mais qu'est-ce qu'on peut faire sur une classe ? Et bien tout simplement *modifier les attributs de celle-ci* car, comme je l'ai déjà dit, des attributs appartenant à une classe ne sont autre que des attributs statiques ! Les attributs statiques servent en particulier à avoir des attributs *indépendants de tout objet*. Ainsi, vous aurez beau créer des tas d'objets, votre attribut aura toujours la même valeur (sauf si l'objet modifie sa valeur, bien sûr). Mieux encore : si l'un des objets modifie sa valeur, tous les autres objets qui accéderont à cet attribut obtiendront la nouvelle valeur ! C'est logique quand on y pense, car un attribut statique appartenant à la classe, il n'existe qu'en un seul exemplaire. Si on le modifie, tout le monde pourra accéder à sa nouvelle valeur.

La ligne 47 commence avec le mot-clé `self`, ce qui veut dire (en gros) « moi-même » (= la classe). Notre ligne veut donc dire : « Dans moi-même, donne-moi l'attribut statique `$_texteADire`. » (Je sais ce n'est pas bien français mais c'est la meilleure traduction mot à mot que j'ai pu trouver.)

N'oubliez pas de mettre un « $ » devant le nom de l'attribut. C'est souvent source d'erreur donc faites bien attention.

Nous allons maintenant faire un petit exercice. Je veux que vous me fassiez une classe toute bête qui ne sert à rien. Seulement, à la fin du script, je veux pouvoir afficher le nombre de fois où la classe a été instanciée. Pour cela, vous aurez besoin d'un attribut appartenant à la classe (admettons `$_compteur`) qui est incrémenté dans le constructeur.

Voici la correction :

Code : PHP

```php
<?php
class Compteur
{
  // Déclaration de la variable $compteur
  private static $_compteur = 0;

  public function __construct()
  {
    // On instancie la variable $compteur qui appartient à la
classe (donc utilisation du mot-clé self).
    self::$_compteur++;
  }

  public static function getCompteur() // Méthode statique qui
renverra la valeur du compteur.
  {
    return self::$_compteur;
  }
}

$test1 = new Compteur;
$test2 = new Compteur;
$test3 = new Compteur;

echo Compteur::getCompteur();
?>
```

Eh oui, le retour du mot-clé `self`… Pourquoi pas `$this` ? Souvenez-vous : on n'accède pas à un attribut statique avec `$this` mais avec `self` ! `self` représente la classe tandis que `$this` représente l'objet actuellement créé. Si un attribut statique est modifié, il n'est pas modifié uniquement dans l'objet créé mais dans la structure complète de la classe ! Je me répète, mais il est essentiel que vous compreniez ça, c'est très important.

En résumé

- L'opérateur « `->` » permet d'accéder à un élément de tel objet, tandis que l'opérateur « `::` » permet d'accéder à un élément de telle classe.
- Au sein d'une méthode, on accède à l'objet grâce à la pseudo-variable `$this`, tandis qu'on accède à la classe grâce au mot-clé `self`.
- Les attributs et méthodes statiques ainsi que les constantes de classe sont des éléments propres à la classe, c'est-à-dire qu'il n'est pas utile de créer un objet pour s'en servir.
- Les constantes de classe sont utiles pour éviter d'avoir un code muet, c'est-à-dire un code qui, sans commentaire, ne nous informe pas vraiment sur son fonctionnement.
- Les attributs et méthodes statiques sont utiles lorsque l'on ne veut pas avoir besoin d'un objet pour s'en servir.

Manipulation de données stockées

Tout site web dynamique doit être capable de sauvegarder des données afin de les utiliser ultérieurement. Comme vous le savez sans doute, l'un des moyens les plus simples et efficaces est la base de données permettant de sauvegarder des centaines de données et de les récupérer en un temps très court.

Cependant, la gestion des données stockées en BDD peut être assez difficile à cerner en POO. Le débutant ne sait en général pas par où commencer et se pose tout un tas de questions : où créer la connexion avec la BDD ? Où placer les requêtes ? Comment traiter les valeurs retournées ? Nous allons justement voir tout cela ensemble. Accrochez-vous bien, un TP suivra pour voir si vous avez bien tout compris !

Une entité, un objet

Rappels sur la structure d'une BDD

Commençons ce chapitre par observer la structure d'une table en BDD et, plus précisément, la manière dont sont stockées les données. Comme vous le savez sûrement, une table n'est autre qu'un grand tableau où sont organisées les données :

	id	nom	forcePerso	degats	niveau	experience
	16	Vyk12	5	55	4	20
	18	Tchaper	5	5	1	0
	19	Neo	5	0	1	20

Exemple d'une table de personnages

Nous voyons ici que notre table contient 3 personnages. Ces données sont stockées sous forme d'entrées. C'est sous cette forme que le gestionnaire de la BDD (MySQL, PostgreSQL, etc.) manipule les données. Si l'on regarde du côté de l'application qui les manipule (ici, notre script PHP), on se rend compte que c'est sous une forme similaire que les données sont récupérées. Cette forme utilisée, vous la connaissez bien : vous utilisiez jusqu'à présent des tableaux pour les manipuler. Exemple :

Code : PHP

```php
<?php
// On admet que $db est un objet PDO
$request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages');

while ($perso = $request->fetch(PDO::FETCH_ASSOC)) // Chaque entrée sera récupérée et placée dans un array.
{
  echo $perso['nom'], ' a ', $perso['forcePerso'], ' de force, ', $perso['degats'], ' de dégâts, ', $perso['experience'], ' d\'expérience et est au niveau ', $perso['niveau'];
}
```

Ça, si vous avez déjà fait un site internet de A à Z, vous avez du l'écrire pas mal de fois !

Pour les exemples, et dans la suite du cours, j'utilise l'API PDO pour accéder à la base de données. Je vous conseille de lire le cours de M@teo21 sur PDO pour être à l'aise avec !

Maintenant, puisque nous programmons de manière orientée objet, nous voulons travailler seulement avec des objets (c'est le principe même de la POO, rappelons-le)et non plus avec des tableaux. En d'autres termes, il va falloir transformer le tableau que l'on reçoit en objet.

Travailler avec des objets

Avant de travailler avec des objets, encore faut-il savoir de quelle classe ils sont issus. Dans notre cas, nous voulons des objets représentant des personnages. On aura donc besoin d'une classe `Personnage` !

Code : PHP

```php
<?php
class Personnage
{

}
?>
```

Vient maintenant une question embêtante dans votre tête pour construire cette classe : « Je commence par où ? »

Une classe est composée de deux parties (éventuellement trois) :

- une partie déclarant les attributs. Ce sont les *caractéristiques* de l'objet ;
- une partie déclarant les méthodes. Ce sont les *fonctionnalités* de chaque objet ;
- éventuellement, une partie déclarant les constantes de classe. Nous nous en occuperons en temps voulu.

Lorsque l'on veut construire une classe, il va donc falloir *systématiquement* se poser les mêmes questions :

- Quelles seront les caractéristiques de mes objets ?
- Quelles seront les fonctionnalités de mes objets ?

Les réponses à ces questions aboutiront à la réalisation du plan de la classe, le plus difficile. 

Commençons donc à réfléchir sur le plan de notre classe en répondant à la première question : « Quelles seront les caractéristiques de mes objets ? » Pour vous mettre sur la piste, regardez de nouveau le tableau de la BDD contenant les entrées, cela vous donnera peut-être la réponse… Et oui, les attributs de notre classe (les caractéristiques) nous sont offerts sur un plateau ! Ils sont listés en haut du tableau : `id`, `nom`, `forcePerso`, `degats`, `niveau` et `experience`.

Écrivons maintenant notre classe :

Code : PHP

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;
}
?>
```

Il faut bien sûr implémenter (écrire) les *getters* et *setters* qui nous permettront d'accéder et de modifier les valeurs de notre objet. Pour rappel, un *getter* est une méthode chargée de renvoyer la valeur d'un attribut, tandis qu'un *setter* une méthode chargée d'assigner une valeur à un attribut **en vérifiant son intégrité** (si vous assignez la valeur sans aucun contrôle, vous perdez tout l'intérêt qu'apporte le principe d'encapsulation).

Pour construire nos setters, il faut donc nous pencher sur les valeurs possibles de chaque attribut :

- les valeurs possibles de l'identifiant sont tous les nombres entiers strictement positifs ;
- les valeurs possibles pour le nom du personnage sont toutes les chaînes de caractères ;
- les valeurs possibles pour la force du personnage sont tous les nombres entiers allant de 1 à 100 ;
- les valeurs possibles pour les dégâts du personnage sont tous les nombres entiers allant de 0 à 100 ;
- les valeurs possibles pour le niveau du personnage sont tous les nombres entiers allant de 1 à 100 ;
- les valeurs possibles pour l'expérience du personnage sont tous les nombres entiers allant de 1 à 100.

On en déduit donc le code de chaque setter. Voici donc ce que donnerait notre classe avec ses getters et setters :

Code : PHP

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;

  // Liste des getters

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

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

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

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

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

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

  // Liste des setters

  public function setId($id)
  {
    // On convertit l'argument en nombre entier.
    // Si c'en était déjà un, rien ne changera.
    // Sinon, la conversion donnera le nombre 0 (à quelques
exceptions près, mais rien d'important ici).
    $id = (int) $id;

    // On vérifie ensuite si ce nombre est bien strictement
positif.
    if ($id > 0)
    {
      // Si c'est le cas, c'est tout bon, on assigne la valeur à
l'attribut correspondant.
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
```

```
    // On vérifie qu'il s'agit bien d'une chaîne de caractères.
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }

  public function setForcePerso($forcePerso)
  {
    $forcePerso = (int) $forcePerso;

    if ($forcePerso >= 1 && $forcePerso <= 100)
    {
      $this->_forcePerso = $forcePerso;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setNiveau($niveau)
  {
    $niveau = (int) $niveau;

    if ($niveau >= 1 && $niveau <= 100)
    {
      $this->_niveau = $niveau;
    }
  }

  public function setExperience($experience)
  {
    $experience = (int) $experience;

    if ($experience >= 1 && $experience <= 100)
    {
      $this->_experience = $experience;
    }
  }
}
?>
```

Reprenons notre code de tout à l'heure, celui qui nous permettait de lire la BDD. Modifions-le un peu pour qu'on puisse manipuler des objets et non des tableaux :

Code : PHP

```
<?php
// On admet que $db est un objet PDO.
$request = $db->query('SELECT id, nom, forcePerso, degats, niveau,
experience FROM personnages');

while ($donnees = $request->fetch(PDO::FETCH_ASSOC)) // Chaque
entrée sera récupérée et placée dans un array.
{
  // On passe les données (stockées dans un tableau) concernant le
personnage au constructeur de la classe.
  // On admet que le constructeur de la classe appelle chaque
setter pour assigner les valeurs qu'on lui a données aux attributs
```

```
        correspondants.
          $perso = new Personnage($donnees);

          echo $perso->nom(), ' a ', $perso->forcePerso(), ' de force, ',
        $perso->degats(), ' de dégâts, ', $perso->experience(), '
        d\'expérience et est au niveau ', $perso->niveau();
        }
```

Et quelles seront les méthodes de nos classes ?

Les méthodes concernent des fonctionnalités que possède l'objet, des actions qu'il peut effectuer. Voyez-vous des fonctionnalités intéressantes à implémenter ? Pour les opérations basiques que l'on effectue, il n'y en a pas besoin. En effet, nous voulons juste créer des objets et assigner des valeurs aux attributs, donc hormis les getters et setters, aucune autre méthode n'est nécessaire !

D'accord, nous avons des objets maintenant. Mais à quoi cela sert-il, concrètement ?

Il est sûr que dans cet exemple précis, cela ne sert à rien. Mais vous verrez plus tard qu'il est beaucoup plus intuitif de travailler avec des objets et par conséquent beaucoup plus pratique, notamment sur de grosses applications où de nombreux objets circulent un peu dans tous les sens.

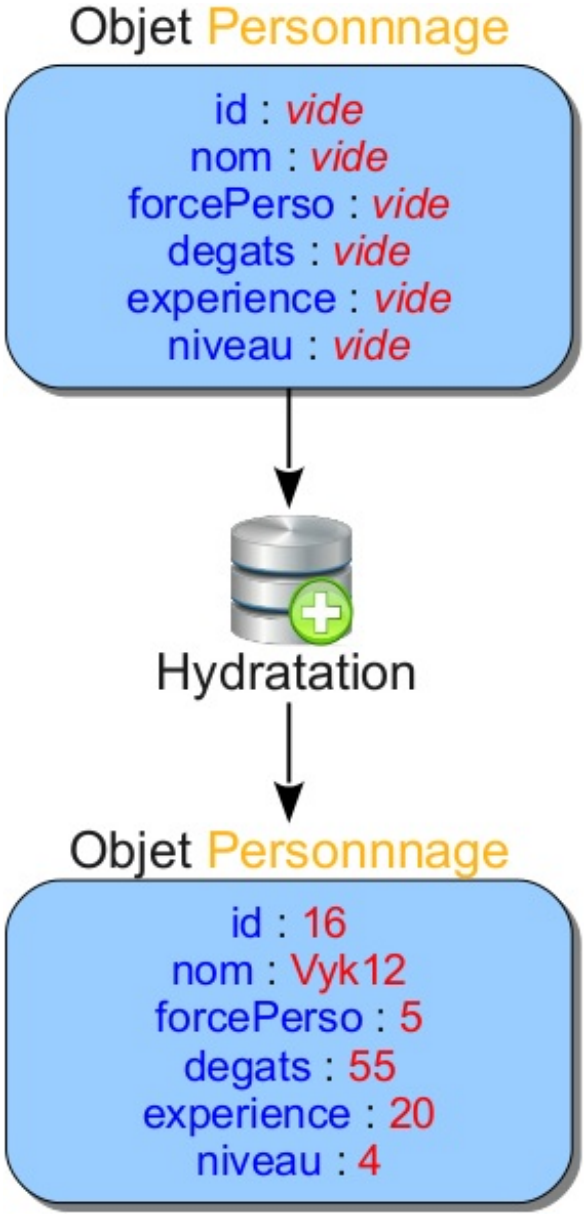
L'hydratation

La théorie de l'hydratation

L'hydratation est un point essentiel dans le domaine de la POO, notamment lorsqu'on utilise des objets représentant des données stockées. Cette notion peut vite devenir compliquée et créer des interrogations pour un développeur débutant si elle est abordée de manière approximative, alors que des explications claires prouvent qu'il n'y a rien de compliqué là-dedans.

Quand on vous parle d'hydratation, c'est qu'on parle d'« objet à hydrater ». Hydrater un objet, c'est tout simplement lui apporter ce dont il a besoin pour fonctionner. En d'autres termes plus précis, hydrater un objet revient à lui fournir des données correspondant à ses attributs pour qu'il assigne les valeurs souhaitées à ces derniers. L'objet aura ainsi des attributs valides et sera en lui-même valide. On dit que l'objet a ainsi été hydraté.

Schématiquement, une hydratation se produit comme ceci :



Hydratation d'un objet

Au début, nous avons un objet `Personnage` dont les attributs sont vides. Comme le schéma le représente, l'hydratation consiste à assigner des valeurs aux attributs. Ainsi l'objet est fonctionnel car il contient des attributs valides : nous avons donc bien hydraté l'objet.

L'hydratation en pratique

Comme on l'a vu, hydrater un objet revient à assigner des valeurs à ses attributs. Qu'avons-nous besoin de faire pour réaliser une telle chose ? Il faut ajouter à l'objet l'action de s'hydrater. Et qui dit action dit méthode !

Nous allons donc écrire notre fonction `hydrate()` :

Code : PHP

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;
```

```
  // Un tableau de données doit être passé à la fonction (d'où le
préfixe « array »).
public function hydrate(array $donnees)
{

}

  public function id() { return $this->_id; }
  public function nom() { return $this->_nom; }
  public function forcePerso() { return $this->_forcePerso; }
  public function degats() { return $this->_degats; }
  public function niveau() { return $this->_niveau; }
  public function experience() { return $this->_experience; }

  public function setId($id)
  {
    // L'identifiant du personnage sera, quoi qu'il arrive, un
nombre entier.
    $this->_id = (int) $id;
  }

  public function setNom($nom)
  {
    // On vérifie qu'il s'agit bien d'une chaîne de caractères.
    // Dont la longueur est inférieure à 30 caractères.
    if (is_string($nom) && strlen($nom) <= 30)
    {
      $this->_nom = $nom;
    }
  }

  public function setForcePerso($forcePerso)
  {
    $forcePerso = (int) $forcePerso;

    // On vérifie que la force passée est comprise entre 0 et 100.
    if ($forcePerso >= 0 && $forcePerso <= 100)
    {
      $this->_forcePerso = $forcePerso;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    // On vérifie que les dégâts passés sont compris entre 0 et
100.
    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setNiveau($niveau)
  {
    $niveau = (int) $niveau;

    // On vérifie que le niveau n'est pas négatif.
    if ($niveau >= 0)
    {
      $this->_niveau = $niveau;
    }
  }

  public function setExperience($exp)
  {
    $exp = (int) $exp;
```

```
        // On vérifie que l'expérience est comprise entre 0 et 100.
        if ($exp >= 0 && $exp <= 100)
        {
          $this->_experience = $exp;
        }
      }
    }
    ?>
```

Vient maintenant l'apect le plus important : que doit-on mettre dans cette méthode ? Souvenez-vous de sa fonction : elle doit hydrater l'objet, c'est-à-dire « assigner les valeurs passées en paramètres aux attributs correspondant ». Cependant, je vous le dis avant que vous fassiez fausse route : il ne faut pas assigner ces valeurs directement, comme ceci :

Code : PHP

```
<?php
// …
public function hydrate(array $donnees)
{
  if (isset($donnees['id']))
  {
    $this->_id = $donnees['id'];
  }

  if (isset($donnees['nom']))
  {
    $this->_nom = $donnees['nom'];
  }

  // …
}
// …
?>
```

La principale raison pour laquelle c'est une mauvaise façon de procéder, est qu'en agissant ainsi vous violez le principe d'encapsulation. En effet, de cette manière, vous ne contrôlez pas l'intégrité des valeurs. Qui vous garantit que le tableau contiendra un nom valide ? Rien du tout! Comment contrôler alors l'intégrité de ces valeurs ? Regardez un peu votre classe, la réponse est sous vos yeux… C'est le rôle des setters de faire ces vérifications ! Il faudra donc les appeler au lieu d'assigner les valeurs directement :

Code : PHP

```
<?php
// …
public function hydrate(array $donnees)
{
  if (isset($donnees['id']))
  {
    $this->setId($donnees['id']);
  }

  if (isset($donnees['nom']))
  {
    $this->setNom($donnees['nom']);
  }

  // …
}
// …
?>
```

Théoriquement, nous avons terminé le travail. Cependant, cela n'est pas très flexible : si vous ajoutez un attribut (et donc son setter correspondant), il faudra modifier votre méthode `hydrate()` pour ajouter la possibilité d'assigner cette valeur. De plus, avec les 6 attributs actuellement créés, il est long de vérifier si la clé existe puis d'appeler la méthode correspondante.

Nous allons donc procéder plus rapidement : nous allons créer une boucle qui va parcourir le tableau passé en paramètre. Si le setter correspondant à la clé existe, alors on appelle ce setter en lui passant la valeur en paramètre. En langage naturel, l'algorithme peut s'écrire de cette façon :

Code : Autre

```
PARCOURS du tableau $donnees (avec pour clé $cle et pour valeur $valeur)
  On assigne à $setter la valeur « 'set'.$cle », en mettant la première lettre de
  SI la méthode $setter de notre classe existe ALORS
    On invoque $setter($valeur)
  FIN SI
FIN PARCOURS
```

Pas de panique, je vais vous expliquer chaque ligne. Dans votre tête, prenez un exemple de valeur pour `$donnees` :

Code : PHP

```php
<?php
$donnees = array(
  'id' => 16,
  'nom' => 'Vyk12',
  'forcePerso' => 5,
  'degats' => 55,
  'niveau' => 4,
  'experience' => 20
);
?>
```

Vous voyez bien que chaque clé correspond à un attribut de notre objet, à qui on assigne une valeur précise. Dans notre méthode `hydrate()`, lorsqu'on parcourt notre tableau, on a successivement la clé `id`, puis `nom`, puis `forcePerso`, etc., avec leur valeur correspondante.

En récupérant le nom de l'attribut, il est facile de déterminer le setter correspondant. En effet, chaque setter a pour nom `setNomDeLAttribut`.

Il est important de préciser que la première lettre du nom de l'attribut doit être en majuscule. Par exemple, le setter correspondant à `nom` est `setNom`.

Pour l'instant notre méthode ressemble à ceci :

Code : PHP

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);
    // …
  }
}
// …
```

```
?>
```

Il faut maintenant vérifier que la méthode existe. Pour cela, nous allons utiliser la fonction method_exists(). Elle prend en premier paramètre le nom de la classe ou une instance de cette classe, et en deuxième paramètre le nom de la méthode qui nous intéresse. La méthode renvoie `true` si la méthode existe, sinon `false`.

Je vous laisse ajouter cette condition qui permet de voir si le setter correspondant existe.

Voici la correction :

Code : PHP

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);

    if (method_exists($this, $method))
    {
      // …
    }
  }
}
// …
?>
```

Et maintenant, il ne reste plus qu'à appeler le setter à l'intérieur de la condition ! Pour cela, je vais vous apprendre une petite astuce. Il est possible d'appeler une méthode dynamiquement, c'est-à-dire appeler une méthode dont le nom n'est pas connu à l'avance (en d'autres termes, le nom est connu seulement pendant l'exécution et est donc stocké dans une variable). Pour ce faire, rien de plus simple ! Regardez ce code :

Code : PHP

```php
<?php
class A
{
  public function hello()
  {
    echo 'Hello world !';
  }
}

$a = new A;
$method = 'hello';

$a->$method(); // Affiche : « Hello world ! »
?>
```

Il est bien entendu possible de passer des arguments à la méthode.

Vous êtes maintenant capables de créer la dernière instruction dans notre méthode ! Pour rappel, celle-ci doit invoquer le setter dont le nom est contenu dans la variable `$method`.

Code : PHP

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    // On récupère le nom du setter correspondant à l'attribut.
    $method = 'set'.ucfirst($key);

    // Si le setter correspondant existe.
    if (method_exists($this, $method))
    {
      // On appelle le setter.
      $this->$method($value);
    }
  }
}
// …
?>
```

Cette fonction est très importante, vous la retrouverez dans de nombreux codes (parfois sous des formes différentes) provenant de plusieurs développeurs. Gardez-là donc dans un coin de votre tête.

Il est courant d'implémenter un constructeur à ces classes demandant un tableau de valeurs pour qu'il appelle ensuite la fonction d'hydratation afin que l'objet soit hydraté dès sa création, comme on l'a vu au début de ce chapitre.

Gérer sa BDD correctement

On vient de voir jusqu'à présent comment gérer les données que les requêtes nous renvoient, mais où placer ces requêtes ? Notre but est de programmer orienté objet, donc nous voulons le moins de code possible en-dehors des classes pour mieux l'organiser. Beaucoup de débutants sont tentés de placer les requêtes dans des méthodes de la classe représentant une entité de la BDD. Par exemple, dans le cas du personnage, je parie que la plupart d'entre vous seraient tentés d'écrire les méthodes `add()` et `update()` dans la classe `Personnage`, ayant respectivement pour rôle d'exécuter une requête pour ajouter et modifier un personnage en BDD.

Arrêtez-vous là tout de suite ! Je vais vous expliquer pourquoi vous faites fausse route, puis vous montrerai la bonne marche à suivre.

Une classe, un rôle

En POO, il y a une phrase très importante qu'il faut que vous ayez constamment en tête : « Une classe, un rôle. » Maintenant, répondez clairement à cette question : « Quel est le rôle d'une classe comme `Personnage` » ?

Un objet instanciant une classe comme `Personnage` a pour rôle de *représenter* une ligne présente en BDD. Le verbe « représenter » est ici très important. En effet, « représenter » est très différent de « gérer ». Une ligne de la BDD ne peut pas s'auto-gérer ! C'est comme si vous demandiez à un ouvrier ayant construit un produit de le commercialiser : l'ouvrier est tout à fait capable de le construire, c'est son *rôle*, mais il ne s'occupe pas du tout de sa gestion, il en est incapable. Il faut donc qu'une deuxième personne intervienne, un commercial, qui va s'occuper de vendre ce produit.

Pour revenir à nos objets d'origine, nous aurons donc besoin de quelque chose qui va s'occuper de les gérer. Ce quelque chose, vous l'aurez peut-être deviné, n'est autre qu'un objet. Un objet gérant des entités issues d'une BDD est généralement appelé un « manager ».

Comme un manager ne fonctionne pas sans support de stockage (dans notre cas, une BDD), on va prendre un exemple concret en créant un gestionnaire pour nos personnages, qui va donc se charger d'en ajouter, d'en modifier, d'en supprimer et d'en récupérer. Puisque notre classe est un gestionnaire de personnages, je vous propose de la nommer `PersonnagesManager`. Cependant, rappelez-vous les questions que l'on doit se poser pour établir le plan de notre classe :

- Quelles seront les caractéristiques de mes objets ?
- Quelles seront les fonctionnalités de mes objets ?

Les caractéristiques d'un manager

Partie délicate, car cela est moins intuitif que de trouver les caractéristiques d'un personnage. Pour trouver la réponse, vous devrez passer par une autre question (ce serait trop simple de toujours répondre à la même question !) : « De quoi a besoin un manager pour fonctionner ? »

Même si la réponse ne vous vient pas spontanément à l'esprit, vous la connaissez : elle a besoin d'une connexion à la BDD pour pouvoir exécuter des requêtes. En utilisant PDO, vous devriez savoir que la connexion à la BDD est représentée par un objet, un *objet d'accès à la BDD* (ou DAO pour *Database Access Object*). Vous l'avez peut-être compris : notre manager aura besoin de cet objet pour fonctionner, donc notre classe aura un attribut stockant cet objet.

Notre classe a-t-elle besoin d'autre chose pour fonctionner ?

Non, c'est tout. Vous pouvez donc commencer à écrire votre classe :

Code : PHP

```php
<?php
class PersonnagesManager
{
  private $_db;
}
?>
```

Les fonctionnalités d'un manager

Pour pouvoir gérer au mieux des entités présentes en BDD (ici, nos personnages), il va falloir quelques fonctionnalités de base. Quelles sont-elles ?

Un manager doit pouvoir :

- enregistrer une nouvelle entité ;
- modifier une entité ;
- supprimer une entité ;
- sélectionner une entité.

C'est le fameux CRUD (*Create, Read, Update, Delete*). N'oublions pas aussi d'ajouter un setter pour notre manager afin de pouvoir modifier l'attribut `$_db` (pas besoin d'accesseur). La création d'un constructeur sera aussi indispensable si nous voulons assigner à cet attribut un objet PDO dès l'instanciation du manager.

Nous allons donc écrire notre premier manager en écrivant les méthodes correspondant à ces fonctionnalités. Écrivez dans un premier temps les méthodes en ne les remplissant que de commentaires décrivant les instructions qui y seront écrites. Cela vous permettra d'organiser clairement le code dans votre tête.

Code : PHP

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO.

  public function __construct($db)
  {
    $this->setDb($db);
  }

  public function add(Personnage $perso)
```

```
  {
    // Préparation de la requête d'insertion.
    // Assignation des valeurs pour le nom, la force, les dégâts,
l'expérience et le niveau du personnage.
    // Exécution de la requête.
  }

  public function delete(Personnage $perso)
  {
    // Exécute une requête de type DELETE.
  }

  public function get($id)
  {
    // Exécute une requête de type SELECT avec une clause WHERE, et
retourne un objet Personnage.
  }

  public function getList()
  {
    // Retourne la liste de tous les personnages.
  }

  public function update(Personnage $perso)
  {
    // Prépare une requête de type UPDATE.
    // Assignation des valeurs à la requête.
    // Exécution de la requête.
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
?>
```

Une fois cette étape accomplie, vous pouvez remplacer les commentaires par les instructions correspondantes. C'est la partie la plus facile car vous l'avez déjà fait de nombreuses fois en procédural : ce ne sont que des requêtes à taper et des valeurs à retourner.

Code : PHP

```
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

  public function __construct($db)
  {
    $this->setDb($db);
  }

  public function add(Personnage $perso)
  {
    $q = $this->_db->prepare('INSERT INTO personnages SET nom =
:nom, forcePerso = :forcePerso, degats = :degats, niveau = :niveau,
experience = :experience');

    $q->bindValue(':nom', $perso->nom());
    $q->bindValue(':forcePerso', $perso->forcePerso(),
PDO::PARAM_INT);
    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':niveau', $perso->niveau(), PDO::PARAM_INT);
    $q->bindValue(':experience', $perso->experience(),
PDO::PARAM_INT);
```

```
    $q->execute();
  }

  public function delete(Personnage $perso)
  {
    $this->_db->exec('DELETE FROM personnages WHERE id = '.$perso->id());
  }

  public function get($id)
  {
    $id = (int) $id;

    $q = $this->_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages WHERE id = '.$id);
    $donnees = $q->fetch(PDO::FETCH_ASSOC);

    return new Personnage($donnees);
  }

  public function getList()
  {
    $persos = array();

    $q = $this->_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages ORDER BY nom');

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      $persos[] = new Personnage($donnees);
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->_db->prepare('UPDATE personnages SET forcePerso = :forcePerso, degats = :degats, niveau = :niveau, experience = :experience WHERE id = :id');

    $q->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM_INT);
    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':niveau', $perso->niveau(), PDO::PARAM_INT);
    $q->bindValue(':experience', $perso->experience(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
?>
```

Essayons tout ça !

Faisons un petit test pour s'assurer que tout cela fonctionne.

Assurez-vous de bien avoir créé une table `personnages` dans votre BDD.

Commençons par créer un objet `Personnage` :

Code : PHP

```php
<?php
$perso = new Personnage(array(
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
));
?>
```

Maintenant, comment l'ajouter en BDD ? Il faudra d'abord créer une instance de notre manager, *en lui passant une instance de PDO*.

Code : PHP

```php
<?php
$perso = new Personnage(array(
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
));

$db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
$manager = new PersonnagesManager($db);
?>
```

Nous n'avons plus qu'une instruction à entrer : celle ordonnant l'ajout du personnage en BDD. Et c'est tout !

Code : PHP

```php
<?php
$perso = new Personnage(array(
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
));

$db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
$manager = new PersonnagesManager($db);

$manager->add($perso);
?>
```

Il est maintenant temps de mettre en pratique ce que vous venez d'apprendre. Cependant, avant de continuer, assurez-vous bien de tout avoir compris, sinon vous serez perdus car vous ne comprendrez pas le code du TP !

En résumé

- Du côté du script PHP, chaque enregistrement de la base de données est représenté par un objet possédant une liste

d'attributs identique à la liste des colonnes de la table.
- Il doit être possible d'*hydrater* chacun des objets grâce à une méthode `hydrate` qui a pour rôle d'assigner les valeurs passées en paramètre aux attributs correspondant.
- La communication avec la BDD se fait par le biais d'un objet différent de l'objet représentant l'enregistrement (une classe = un rôle). Un tel objet est un *manager*.
- Un *manager* peut stocker les objets en BDD, mais peut tout-à-fait les stocker sur un autre support (fichier XML, fichier texte, etc.).

TP : Mini-jeu de combat

Voici un petit TP qui mettra en pratique ce que l'on vient de voir. Celui-ci est étroitement lié avec le précédent chapitre. Ainsi, je vous conseille d'avoir bien compris ce dernier avant d'aborder ce TP, sinon vous n'arriverez sans doute pas au résultat tout seul.

En tout, vous aurez 3 TP de la sorte dont le niveau de difficulté sera croissant. Puisque celui-ci est votre premier, tout sera expliqué dans les moindres détails.

Ce qu'on va faire

Pour information, j'utilise ici PDO. Si vous ne savez toujours pas maitriser cette API, alors allez lire le tutoriel *PDO : Interface d'accès aux BDD*. Vous avez les connaissances requises pour pouvoir manipuler cette bibliothèque.

Cahier des charges

Ce que nous allons réaliser est très simple. Nous allons créer une sorte de jeu. Chaque visiteur pourra créer un personnage (pas de mot de passe requis pour faire simple) avec lequel il pourra frapper d'autres personnages. Le personnage frappé se verra infliger un certain degré de dégâts.

Un personnage est défini selon 2 caractéristiques :

- Son nom (unique).
- Ses dégâts.

Les dégâts d'un personnage sont compris entre 0 et 100. Au début, il a bien entendu 0 de dégât. Chaque coup qui lui sera porté lui fera prendre 5 points de dégâts. Une fois arrivé à 100 points de dégâts, le personnage est mort (on le supprimera alors de la BDD).

Notions utilisées

Voici une petite liste vous indiquant les points techniques que l'on va mettre en pratique avec ce TP :

- Les attributs et méthodes ;
- l'*instanciation* de la classe ;
- les constantes de classe ;
- et surtout, tout ce qui touche à la **manipulation de données stockées**.

Cela dit, et c'est je pense ce qui sera le plus difficile, ce qui sera mis en valeur ici sera la **conception** d'un mini-projet, c'est-à-dire que l'on travaillera surtout ici votre capacité à programmer orienté objet. Cependant, ne prenez pas trop peur : nous avons effectué une importante partie théorique lors du précédent chapitre, et rien de nouveau n'apparaîtra.

Vous avez donc maintenant une idée de ce qui vous attend. Cependant, ceci étant votre premier TP concernant la POO, il est fort probable que vous ne sachiez pas par où commencer, et c'est normal ! nous allons donc réaliser le TP ensemble : je vais vous expliquer comment **penser** un mini-projet et vous poser les bonnes questions.

Pré-conception

Avant de nous attaquer au cœur du script, nous allons réfléchir à son organisation. De quoi aura-t-on besoin ? Puisque nous travaillerons avec des personnages, nous aurons besoin de les stocker pour qu'ils puissent durer dans le temps. L'utilisation d'une base de données sera donc indispensable.

Le script étant simple, nous n'aurons qu'une table **personnages** qui aura différents champs. Pour les définir, réfléchissez à ce qui caractérise un personnage. Ainsi nous connaissons déjà 2 champs de cette table que nous avons définis au début : **nom** et **dégâts** . Et bien sûr, n'oublions pas le plus important : l'identifiant du personnage ! Chaque personnage doit posséder un identifiant unique qui permet ainsi de le rechercher plus rapidement (au niveau performances) qu'avec son nom.

Vous pouvez donc ainsi créer votre table tous seuls *via* PhpMyAdmin. Si vous n'êtes pas sûrs de vous, je vous laisse le code SQL créant cette table :

Code : SQL

```
CREATE TABLE IF NOT EXISTS `personnages` (
```

```
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `nom` varchar(50) COLLATE latin1_general_ci NOT NULL,
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  UNIQUE KEY `nom` (`nom`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci;
```

Voir le résultat que vous obtiendrez

Le résultat présenté ici contient les améliorations proposées en fin de chapitre.

Première étape : le personnage

Nous allons commencer le TP. Pour rappel, nous allons réaliser un petit jeu mettant en scène des personnages qui peuvent combattre. Qui dit personnages dit **objets** `Personnage` (je pense que ça, vous l'avez deviné, puisque nous avons travaillé dessus durant les premiers chapitres).

Arrive maintenant un moment délicat dans votre tête : **"Par où je commence ?"**

Souvenez-vous de la première partie du précédent chapitre. Pour construire une classe, vous devez répondre à deux questions qui vont vous permettre d'établir le plan de votre classe :

- Quelles seront les caractéristiques de mes objets ?
- Quelles seront les fonctionnalités de mes objets ?

Voici comment procéder. Nous allons dans un premier temps dresser la liste des caractéristiques du personnage pour ensuite se pencher sur ses fonctionnalités.

Rappel : le traitement des données (c'est-à-dire l'exécution de requêtes permettant d'aller effectuer des opérations en BDD) se fera **dans une autre classe**. Ne vous en préoccupez donc pas maintenant !

Les caractéristiques du personnage

Essayez de vous souvenir quelles sont les caractéristiques d'un personnage (ou plus globalement celles d'une entité représentant un enregistrement présent en BDD). Nous l'avons vu dans la première partie du précédent chapitre.

Les attributs d'un objet représentant un enregistrement présent en BDD correspondent aux noms des champs de cet enregistrement.

Si vous n'avez pas retenu cette information, notez-là quelque part, vous en aurez besoin un très grand nombre de fois.

Maintenant que nous avons déterminé les caractéristiques d'un objet `Personnage`, nous savons quels attributs placer dans notre classe. Voici une première écriture de notre classe `Personnage` :

Code : PHP

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;
}
```

Jusque là tout va bien. Tournons-nous maintenant vers les fonctionnalités que doit posséder un personnage.

Les fonctionnalités d'un personnage

Comme nous l'avons dit tout-à-l'heure, pour obtenir les méthodes d'un objet, il faut se demander quelles seront les fonctionnalités de ces entités. Ici, **que pourra faire un personnage** ? Relisez les consignes du début de chapitre et répondez clairement à la question.

Un personnage doit pouvoir :

- **Frapper** un autre personnage ;
- **recevoir** des dégâts.

À chaque fonctionnalité correspond une méthode. Écrivez ces méthodes dans la classe en mettant en commentaire ce qu'elles doivent faire.

Ne vous étonnez pas si vous avez un résultat bien différent du mien. Le contenu des méthodes importe peu car il est issu de mon imagination et il est normal que vous n'ayez pas pensé le script comme moi. Gardez votre version, n'essayez pas de copier/coller mon code.

Code : PHP

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
    // Si c'est le cas, on stoppe tout en renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage a bien été frappé.
  }
}
```

Tout ceci n'est que du français, nous sommes encore à l'étape de **réflexion**. Si vous sentez que ça va trop vite, relisez le début du TP et suivez bien les étapes. C'est très important car c'est en général cette étape qui pose problème : la **réflexion** ! Beaucoup de débutants foncent tête baissée sans rendre le temps de concevoir correctement les méthodes au début et se plantent royalement. Prenez donc bien votre temps.

Normalement, des petites choses doivent vous chiffonner.

Pour commencer, la première chose qui doit vous interpeller se situe dans la méthode `frapper()`, lorsqu'il faut vérifier qu'on ne se frappe pas soi-même. Pour cela, il suffit de comparer l'identifiant du personnage qui attaque avec l'identifiant du personnage ciblé. En effet, si l'identifiant est le même, alors il s'agira d'un seul et même personnage.

Ensuite, à certains moments dans le code, il est dit que la méthode « renverra une valeur signifiant que le personnage a bien été frappé » par exemple. Qu'est-ce que cela peut bien signifier ? Ce genre de commentaire laissera place à des **constantes de classe**

(si vous l'aviez deviné, bien joué !). Si vous regardez bien le code, vous verrez 3 commentaires de la sorte :

- Méthode `frapper()` : *« [...] renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque »* → la méthode renverra la valeur de la constante `CEST_MOI` ;
- Méthode `recevoirDegats()` : *« la méthode renverra une valeur signifiant que le personnage a été tué »* → la méthode renverra la valeur de la constante `PERSONNAGE_TUE` ;
- Méthode `recevoirDegats()` : *« elle renverra une valeur signifiant que le personnage a bien été frappé »* → la méthode renverra la valeur de la constante `PERSONNAGE_FRAPPE`.

Vous pouvez ajouter ces constantes à votre classe.

Code : PHP

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  const CEST_MOI = 1;
  const PERSONNAGE_TUE = 2;
  const PERSONNAGE_FRAPPE = 3;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
      // Si c'est le cas, on stoppe tout en renvoyant une valeur
signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des
dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une
valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage
a bien été frappé.
  }
}
```

Les getters et setters

Actuellement, les attributs de nos objets sont inaccessibles. Il faut créer des *getters* pour pouvoir les lire, et des *setters* pour pouvoir modifier leurs valeurs. Nous allons aller un peu plus rapidement que les précédentes méthodes en écrivant directement le contenu de celles-ci car une ou deux instructions suffisent en général.

N'oubliez pas de vérifier l'intégrité des données dans les *setters* sinon ils perdent tout leur intérêt !

Code : PHP

```php
<?php
class Personnage
{
  private $_id,
```

```
            $_degats,
            $_nom;

  const CEST_MOI = 1;
  const PERSONNAGE_TUE = 2;
  const PERSONNAGE_FRAPPE = 3;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
      // Si c'est le cas, on stoppe tout en renvoyant une valeur
signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des
dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une
valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage
a bien été frappé.
  }

  public function degats()
  {
    return $this->_degats;
  }

  public function id()
  {
    return $this->_id;
  }

  public function nom()
  {
    return $this->_nom;
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setId($id)
  {
    $id = (int) $id;

    if ($id > 0)
    {
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }
```

```
    }
```

Hydrater ses objets

Deuxième point essentiel que nous allons ré-exploiter dans ce TP : l'hydratation.

Je n'ai pas d'explication supplémentaire à ajouter, tout est dit dans le précédent chapitre. La méthode créée dans ce dernier est d'ailleurs réutilisable ici. Au lieu de regarder la correction en vous replongeant dans le chapitre précédent, essayez plutôt de réécrire la méthode. Pour rappel, celle-ci doit permettre d'assigner aux attributs de l'objet les valeurs correspondantes, passées en paramètre dans un tableau. Si vous avez essayé de réécrire la méthode, voici le résultat que vous auriez du obtenir :

Code : PHP

```php
<?php
class Personnage
{
  // ...

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  // ...
}
```

Il ne manque plus qu'à implémenter le constructeur pour qu'on puisse directement hydrater notre objet lors de l'instanciation de la classe. Pour cela, ajoutez un paramètre : `$donnees`. Appelez ensuite directement la méthode `hydrate()`.

Code : PHP

```php
<?php
class Personnage
{
  // ...

  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
  }

  // ...
}
```

Codons le tout !

La partie réflexion est terminé, il est maintenant temps d'écrire notre classe `Personnage` ! Voici la correction que je vous propose :

Code : PHP

```php
<?php
class Personnage
{
  private $_degats,
          $_id,
          $_nom;

  const CEST_MOI = 1; // Constante renvoyée par la méthode
`frapper` si on se frappe soi-même.
  const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode
`frapper` si on a tué le personnage en le frappant.
  const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode
`frapper` si on a bien frappé le personnage.


  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
  }

  public function frapper(Personnage $perso)
  {
    if ($perso->id() == $this->_id)
    {
      return self::CEST_MOI;
    }

    // On indique au personnage qu'il doit recevoir des dégâts.
    // Puis on retourne la valeur renvoyée par la méthode :
self::PERSONNAGE_TUE ou self::PERSONNAGE_FRAPPE
    return $perso->recevoirDegats();
  }

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  public function recevoirDegats()
  {
    $this->_degats += 5;

    // Si on a 100 de dégâts ou plus, on dit que le personnage a
été tué.
    if ($this->_degats >= 100)
    {
      return self::PERSONNAGE_TUE;
    }

    // Sinon, on se contente de dire que le personnage a bien été
frappé.
    return self::PERSONNAGE_FRAPPE;
  }


  // GETTERS //


  public function degats()
  {
    return $this-> degats;
```

```
    }

    public function id()
    {
      return $this->_id;
    }

    public function nom()
    {
      return $this->_nom;
    }

    public function setDegats($degats)
    {
      $degats = (int) $degats;

      if ($degats >= 0 && $degats <= 100)
      {
        $this->_degats = $degats;
      }
    }

    public function setId($id)
    {
      $id = (int) $id;

      if ($id > 0)
      {
        $this->_id = $id;
      }
    }

    public function setNom($nom)
    {
      if (is_string($nom))
      {
        $this->_nom = $nom;
      }
    }
  }
```

Prenez le temps de bien comprendre ce code et de lire les commentaires. Il est important que vous cerniez son fonctionnement pour savoir ce que vous faites. Cependant, si vous ne comprenez pas toutes les instructions placées dans les méthodes, ne vous affolez pas : si vous avez compris globalement le rôle de chacune d'elles, vous n'aurez pas de handicap pour suivre la prochaine sous-partie.

Seconde étape : stockage en base de données

Attaquons-nous maintenant à la deuxième grosse partie de ce TP, celle consistant à pouvoir stocker nos personnages dans une base de données. Grande question maintenant : **comment faire ?**

Nous avons répondu à cette question dans la troisième partie du précédent chapitre. Au cas où certains seraient toujours tentés de placer les requêtes qui iront chercher les personnages en BDD dans la classe `Personnage`, je vous arrête tout de suite et vous fais un bref rappel avec cette phrase que vous avez déjà rencontrée : **une classe, un rôle**.

J'espère que vous l'avez retenue cette fois-ci !

Vous souvenez-vous de ce que cela signifie ? Quel est le rôle de notre classe `Personnage` ? Où placer nos requêtes ?

La classe `Personnage` a pour rôle de **représenter** un personnage présent en BDD. Elle n'a en aucun cas pour rôle de les **gérer**. Cette gestion sera le rôle d'une autre classe, communément appelée **manager**. Dans notre cas, notre gestionnaire de personnage sera tout simplement nommée `PersonnagesManager`.

Comment va-t-on faire pour construire ces classes ? Quelles questions va-t-on se poser ?

- Quelles seront les caractéristiques d'un manager ?
- Quelles seront les fonctionnalités d'un manager ?

Les caractéristiques d'un manager

Encore une fois, ce point a été abordé dans la troisième partie du précédent chapitre. Allez y faire un tour si vous avez un trou de mémoire ! Voici le code de la classe contenant sa (grande) liste d'attributs.

Code : PHP

```php
<?php
class PersonnagesManager
{
  private $_db;
}
```

Les fonctionnalités d'un manager

Dans la troisième partie du précédent chapitre, nous avons vu quelques fonctionnalités de base. Notre manager pouvait :

- Enregistrer un nouveau personnage ;
- modifier un personnage ;
- supprimer un personnage ;
- sélectionner un personnage.

Cependant, ici, nous pouvons ajouter quelques fonctionnalités qui pourront nous être utiles :

- Compter le nombre de personnages ;
- récupérer une liste de plusieurs personnages ;
- savoir si un personnage existe.

Cela nous fait ainsi 7 méthodes à implémenter !

Rappels du précédent chapitre : n'oubliez pas d'ajouter un *setter* pour notre manager afin de pouvoir modifier l'attribut `$_db`. La création d'un constructeur sera aussi indispensable si nous voulons assigner à cet attribut un objet PDO dès l'instanciation du manager.

Comme d'habitude, écrivez le nom des méthodes en ajoutant des commentaires sur ce que doit faire la méthode.

Code : PHP

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

  public function __construct($db)
  {
    $this->setDb($db);
  }
```

```
  public function add(Personnage $perso)
  {
    // Préparation de la requête d'insertion.
    // Assignation des valeurs pour le nom du personnage.
    // Exécution de la requête.

    // Hydratation du personnage passé en paramètre avec
assignation de son identifiant et des dégâts initiaux (= 0).
  }

  public function count()
  {
    // Exécute une requête COUNT() et retourne le nombre de
résultats retourné.
  }

  public function delete(Personnage $perso)
  {
    // Exécute une requête de type DELETE.
  }

  public function exists($info)
  {
    // Si le paramètre est un entier, c'est qu'on a fourni un
identifiant.
      // On exécute alors une requête COUNT() avec une clause
WHERE, et on retourne un boolean.

    // Sinon c'est qu'on a passé un nom.
    // Exécution d'une requête COUNT() avec une clause WHERE, et
retourne un boolean.
  }

  public function get($info)
  {
    // Si le paramètre est un entier, on veut récupérer le
personnage avec son identifiant.
      // Exécute une requête de type SELECT avec une clause WHERE,
et retourne un objet Personnage.

    // Sinon, on veut récupérer le personnage avec son nom.
    // Exécute une requête de type SELECT avec une clause WHERE, et
retourne un objet Personnage.
  }

  public function getList($nom)
  {
    // Retourne la liste des personnages dont le nom n'est pas
$nom.
    // Le résultat sera un tableau d'instances de Personnage.
  }

  public function update(Personnage $perso)
  {
    // Prépare une requête de type UPDATE.
    // Assignation des valeurs à la requête.
    // Exécution de la requête.
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Codons le tout !

Normalement, l'écriture des méthodes devrait être plus facile que dans la précédente partie. En effet, ici, il n'y a que des requêtes à écrire : si vous savez utiliser PDO, vous ne devriez pas avoir de mal.

Code : PHP

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

  public function __construct($db)
  {
    $this->setDb($db);
  }

  public function add(Personnage $perso)
  {
    $q = $this->_db->prepare('INSERT INTO personnages SET nom =
:nom');
    $q->bindValue(':nom', $perso->nom());
    $q->execute();

    $perso->hydrate(array(
      'id' => $this->_db->lastInsertId(),
      'degats' => 0,
    ));
  }

  public function count()
  {
    return $this->_db->query('SELECT COUNT(*) FROM personnages')-
>fetchColumn();
  }

  public function delete(Personnage $perso)
  {
    $this->_db->exec('DELETE FROM personnages WHERE id = '.$perso-
>id());
  }

  public function exists($info)
  {
    if (is_int($info)) // On veut voir si tel personnage ayant pour
id $info existe.
    {
      return (bool) $this->_db->query('SELECT COUNT(*) FROM
personnages WHERE id = '.$info)->fetchColumn();
    }

    // Sinon, c'est qu'on veut vérifier que le nom existe ou pas.

    $q = $this->_db->prepare('SELECT COUNT(*) FROM personnages WHERE
nom = :nom');
    $q->execute(array(':nom' => $info));

    return (bool) $q->fetchColumn();
  }

  public function get($info)
  {
    if (is_int($info))
    {
      $q = $this->_db->query('SELECT id, nom, degats FROM
personnages WHERE id = '.$info);
      $donnees = $q->fetch(PDO::FETCH_ASSOC);

      return new Personnage($donnees);
    }
    else
```

```
    {
      $q = $this->_db->prepare('SELECT id, nom, degats FROM
personnages WHERE nom = :nom');
      $q->execute(array(':nom' => $info));

      return new Personnage($q->fetch(PDO::FETCH_ASSOC));
    }
  }

  public function getList($nom)
  {
    $persos = array();

    $q = $this->_db->prepare('SELECT id, nom, degats FROM
personnages WHERE nom <> :nom ORDER BY nom');
    $q->execute(array(':nom' => $nom));

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      $persos[] = new Personnage($donnees);
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->_db->prepare('UPDATE personnages SET degats =
:degats WHERE id = :id');

    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Troisième étape : utilisation des classes

J'ai le plaisir de vous annoncer que vous avez fait le plus gros du travail ! Maintenant, nous allons juste utiliser nos classes en les instanciant et en invoquant les méthodes souhaitées sur nos objets. Le plus difficile ici est de se mettre d'accord sur le déroulement du jeu.

Celui-ci étant simple, nous n'aurons besoin que d'un seul fichier. Commençons par le début : que doit afficher notre mini-jeu lorsqu'on ouvre la page pour la première fois ? Il doit afficher un petit formulaire nous demandant le nom du personnage qu'on veut créer ou utiliser.

Code : PHP

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
  </head>
  <body>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage"
name="creer" />
```

```
        <input type="submit" value="Utiliser ce personnage"
name="utiliser" />
      </p>
    </form>
  </body>
</html>
```

Vient ensuite la partie traitement. Deux cas peuvent se présenter :

- Le joueur a cliqué sur **Créer ce personnage**. Le script devra créer un objet `Personnage` en passant au constructeur un tableau contenant une entrée (le nom du personnage). Il faudra ensuite s'assurer que le personnage ait un nom valide et qu'il n'existe pas déjà. Après ces vérifications, l'enregistrement en BDD pourra se faire.
- Le joueur a cliqué sur **Utiliser ce personnage**. Le script devra vérifier si le personnage existe bien en BDD. Si c'est le cas, on le récupère de la BDD.

Pour savoir si le nom du personnage est valide, il va falloir implémenter une méthode `nomValide()` (je vous laisse réfléchir à quelle classe) qui retournera `true` ou `false` suivant si le nom est valide ou pas. Un nom est valide s'il n'est pas vide.

Cependant, avant de faire cela, il va falloir préparer le terrain.

- Un *autoload* devra être créé (bien que non indispensable puisqu'il n'y a que deux classes).
- Une instance de PDO devra être créée.
- Une instance de notre manager devra être créée.

Puisque notre manager a été créé, pourquoi ne pas afficher en haut de page le nombre de personnages créés ? 

Code : PHP

```
<?php
// On enregistre notre autoload.
function chargerClasse($classname)
{
  require $classname.'.class.php';
}

spl_autoload_register('chargerClasse');

$db = new PDO('mysql:host=localhost;dbname=combats', 'root', '');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING); // On
émet une alerte à chaque fois qu'une requête a échoué.

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu
créer un personnage.
{
  $perso = new Personnage(array('nom' => $_POST['nom'])); // On crée
un nouveau personnage.

  if (!$perso->nomValide())
  {
    $message = 'Le nom choisi est invalide.';
    unset($perso);
  }
  elseif ($manager->exists($perso->nom()))
  {
    $message = 'Le nom du personnage est déjà pris.';
    unset($perso);
```

```
    }
    else
    {
      $manager->add($perso);
    }
  }

  elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on
a voulu utiliser un personnage.
  {
    if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
    {
      $perso = $manager->get($_POST['nom']);
    }
    else
    {
      $message = 'Ce personnage n\'existe pas !'; // S'il n'existe
pas, on affichera ce message.
    }
  }
  ?>
  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
    <head>
      <title>TP : Mini jeu de combat</title>

      <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
    </head>
    <body>
      <p>Nombre de personnages créés : <?php echo $manager->count(); ?
></p>
  <?php
  if (isset($message)) // On a un message à afficher ?
    echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
  ?>
      <form action="" method="post">
        <p>
          Nom : <input type="text" name="nom" maxlength="50" />
          <input type="submit" value="Créer ce personnage"
name="creer" />
          <input type="submit" value="Utiliser ce personnage"
name="utiliser" />
        </p>
      </form>
    </body>
  </html>
```

Au cas où, je vous donne la méthode `nomValide()` de la classe `Personnage`. J'espère cependant que vous y êtes arrivés, un simple contrôle avec `empty()` et le tour est joué.

Code : PHP

```
<?php
class Personnage
{
  // ...

  public function nomValide()
  {
    return !empty($this->_nom);
  }

  // ...
}
```

Une fois que nous avons un personnage, que se passera-t-il ? Il faut en effet cacher ce formulaire et laisser place à d'autres informations. Je vous propose d'afficher, dans un premier temps, les informations du personnage sélectionné (son nom et ses dégâts), puis, dans un second temps, la liste des autres personnages avec leurs informations. Il devra être possible de cliquer sur le nom du personnage pour le frapper.

Code : PHP

```php
<?php
// On enregistre notre autoload.
function chargerClasse($classname)
{
  require $classname.'.class.php';
}

spl_autoload_register('chargerClasse');

$db = new PDO('mysql:host=localhost;dbname=combats', 'root', '');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING); // On
émet une alerte à chaque fois qu'une requête a échoué.

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu
créer un personnage.
{
  $perso = new Personnage(array('nom' => $_POST['nom'])); // On crée
un nouveau personnage.

  if (!$perso->nomValide())
  {
    $message = 'Le nom choisi est invalide.';
    unset($perso);
  }
  elseif ($manager->exists($perso->nom()))
  {
    $message = 'Le nom du personnage est déjà pris.';
    unset($perso);
  }
  else
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on
a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe
pas, on affichera ce message.
  }
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
```

```
  </head>
  <body>
    <p>Nombre de personnages créés : <?php echo $manager->count(); ?
></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?php echo htmlspecialchars($perso->nom()); ?><br />
        Dégâts : <?php echo $perso->degats(); ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
    echo '<a href="?frapper=', $unPerso->id(), '">',
htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso-
>degats(), ')<br />';
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage"
name="creer" />
        <input type="submit" value="Utiliser ce personnage"
name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
```

Maintenant, quelque chose devrait vous titiller. En effet, si on recharge la page, on atterrira à nouveau sur le formulaire. Nous allons donc devoir utiliser le système de **sessions**. La première chose à faire sera alors de démarrer la session au début du script, juste après la déclaration de l'*autoload*. La session démarrée, nous pouvons aisément sauvegarder notre personnage. Pour cela, il nous faudra enregistrer le personnage en session (admettons dans `$_SESSION['perso']`) tout à la fin du code. Cela nous permettra, au début du script, de récupérer le personnage sauvegardé et de continuer le jeu.

Pour des raisons pratiques, il est préférable d'ajouter un lien de déconnexion pour qu'on puisse utiliser un autre personnage si on le souhaite. Ce lien aura pour effet de conduire à un `session_destroy()`.

Code : PHP

```php
<?php
// On enregistre notre autoload.
function chargerClasse($classname)
{
  require $classname.'.class.php';
}

spl_autoload_register('chargerClasse');

session_start(); // On appelle session_start() APRÈS avoir enregistré l'autoload.

if (isset($_GET['deconnexion']))
{
  session_destroy();
  header('Location: .');
  exit();
}

if (isset($_SESSION['perso'])) // Si la session perso existe, on restaure l'objet.
{
  $perso = $_SESSION['perso'];
}

$db = new PDO('mysql:host=localhost;dbname=combats', 'root', '');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING); // On émet une alerte à chaque fois qu'une requête a échoué.

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un personnage.
{
  $perso = new Personnage(array('nom' => $_POST['nom'])); // On crée un nouveau personnage.

  if (!$perso->nomValide())
  {
    $message = 'Le nom choisi est invalide.';
    unset($perso);
  }
  elseif ($manager->exists($perso->nom()))
  {
    $message = 'Le nom du personnage est déjà pris.';
    unset($perso);
  }
  else
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
```

```
    {
      $message = 'Ce personnage n\'existe pas !'; // S'il n'existe
pas, on affichera ce message.
    }
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?php echo $manager->count(); ?
></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?php echo htmlspecialchars($perso->nom()); ?><br />
        Dégâts : <?php echo $perso->degats(); ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
    echo '<a href="?frapper=', $unPerso->id(), '">',
htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso-
>degats(), ')<br />';
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage"
name="creer" />
        <input type="submit" value="Utiliser ce personnage"
name="utiliser" />
```

```
        </p>
      </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans
une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

Il reste maintenant une dernière partie à développer : celle qui s'occupera de frapper un personnage. Puisque nous avons déjà écrit tout le code faisant l'interaction entre l'attaquant et la cible, vous verrez que nous n'aurons presque rien à écrire.

Comment doit se passer la phase de traitement ? Avant toute chose, il faut bien vérifier que le joueur est connecté et que la variable `$perso` existe, sinon nous n'iront pas bien loin. Seconde vérification : il faut demander à notre manager si le personnage que l'on veut frapper existe bien. Si ces deux conditions sont vérifiées, alors on peut lancer l'attaque.

Pour lancer l'attaque, il va falloir récupérer le personnage à frapper grâce à notre manager. Ensuite, il suffira d'invoquer la méthode permettant de frapper le personnage.

Cependant, nous n'allons pas nous arrêter là. N'oubliez pas que cette méthode peut retourner 3 valeurs différentes :

- `Personnage::CEST_MOI`. Le personnage a voulu se frapper lui-même.
- `Personnage::PERSONNAGE_FRAPPE`. Le personnage a bien été frappé.
- `Personnage::PERSONNAGE_TUE`. Le personnage a été tué.

Il va donc falloir afficher un message en fonction de cette valeur retournée. Aussi, seuls 2 de ces cas nécessitent une mise à jour de la BDD : si le personnage a été frappé ou s'il a été tué. En effet, si on a voulu se frapper soi-même, aucun des deux personnages impliqués n'a été modifié.

Code : PHP

```
<?php
// On enregistre notre autoload.
function chargerClasse($classname)
{
  require $classname.'.class.php';
}

spl_autoload_register('chargerClasse');

session_start(); // On appelle session_start() APRÈS avoir
enregistré l'autoload.

if (isset($_GET['deconnexion']))
{
  session_destroy();
  header('Location: .');
  exit();
}

$db = new PDO('mysql:host=localhost;dbname=combats', 'root', '');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING); // On
émet une alerte à chaque fois qu'une requête a échoué.

$manager = new PersonnagesManager($db);

if (isset($_SESSION['perso'])) // Si la session perso existe, on
restaure l'objet.
```

```
{
  $perso = $_SESSION['perso'];
}

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu
créer un personnage.
{
  $perso = new Personnage(array('nom' => $_POST['nom'])); // On crée
un nouveau personnage.

  if (!$perso->nomValide())
  {
    $message = 'Le nom choisi est invalide.';
    unset($perso);
  }
  elseif ($manager->exists($perso->nom()))
  {
    $message = 'Le nom du personnage est déjà pris.';
    unset($perso);
  }
  else
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on
a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe
pas, on affichera ce message.
  }
}

elseif (isset($_GET['frapper'])) // Si on a cliqué sur un
personnage pour le frapper.
{
  if (!isset($perso))
  {
    $message = 'Merci de créer un personnage ou de vous
identifier.';
  }

  else
  {
    if (!$manager->exists((int) $_GET['frapper']))
    {
      $message = 'Le personnage que vous voulez frapper n\'existe
pas !';
    }

    else
    {
      $persoAFrapper = $manager->get((int) $_GET['frapper']);

      $retour = $perso->frapper($persoAFrapper); // On stocke dans
$retour les éventuelles erreurs ou messages que renvoie la méthode
frapper.

      switch ($retour)
      {
        case Personnage::CEST_MOI :
          $message = 'Mais... pourquoi voulez-vous vous frapper ???
';
```

```
            break;

          case Personnage::PERSONNAGE_FRAPPE :
            $message = 'Le personnage a bien été frappé !';

            $manager->update($perso);
            $manager->update($persoAFrapper);

            break;

          case Personnage::PERSONNAGE_TUE :
            $message = 'Vous avez tué ce personnage !';

            $manager->update($perso);
            $manager->delete($persoAFrapper);

            break;
        }
      }
    }
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?php echo $manager->count(); ?
></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?php echo htmlspecialchars($perso->nom()); ?><br />
        Dégâts : <?php echo $perso->degats(); ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
  {
    echo '<a href="?frapper=', $unPerso->id(), '">',
htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso-
```

```
>degats(), ')<br />';
  }
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage"
name="creer" />
        <input type="submit" value="Utiliser ce personnage"
name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans
une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

Et voilà, vous avez un jeu opérationnel.

Améliorations possibles

Ce code est très basique, beaucoup d'améliorations sont possibles. En voici quelques unes :

- Un système de **niveau**. Vous pourriez très bien assigner à chaque personnage un niveau de 1 à 100. Le personnage bénéficierait aussi d'une expérience allant de 0 à 100. Lorsque l'expérience atteint 100, le personnage passe au niveau suivant.

  Indice : le **niveau** et l'**expérience** deviendraient des caractéristiques du personnage, donc... Pas besoin de vous le dire, je suis sûr que vous savez ce que ça signifie !

- Un système de **force**. La force du personnage pourrait augmenter en fonction de son niveau, et les dégâts infligés à la victime seront donc plus importants.

  Indice : de même, la **force** du personnage serait aussi une caractéristique du personnage.

- Un système de **limitation**. En effet, un personnage peut en frapper autant qu'il veut dans un laps de temps indéfini. Pourquoi ne pas le limiter à 3 coups par jour ?

  Indice : il faudrait que vous stockiez le **nombre de coups** portés par le personnage, ainsi que la **date du dernier coup porté**. Cela ferait donc deux nouveaux champs en BDD, et deux nouvelles caractéristiques pour le personnage !

- Un système de **retrait de dégâts**. Chaque jour, si l'utilisateur se connecte, il pourrait voir ses dégâts se soustraire de 10 par exemple.

  Indice : il faudrait stocker la date de dernière connexion. À chaque connexion, vous regarderiez cette date. Si elle est inférieure à 24h, alors vous ne feriez rien. Sinon, vous retireriez 10 de dégâts au personnage puis mettriez à jour cette date de dernière connexion.

Et la liste peut être longue ! Je vous encourage vivement à essayer d'implémenter ces fonctionnalités et à laisser libre court à votre imagination, vous progresserez bien plus. 

L'héritage

L'héritage en POO (que ce soit en C++, Java ou autre langage utilisant la POO) est une technique très puissante et extrêmement pratique. Ce chapitre sur l'héritage est **le** chapitre à connaitre par cœur (ou du moins, le mieux possible). Pour être bien sûr que vous ayez compris le principe, un TP vous attend au prochain chapitre.

Allez, j'arrête de vous mettre la pression, allons-y !

Notion d'héritage

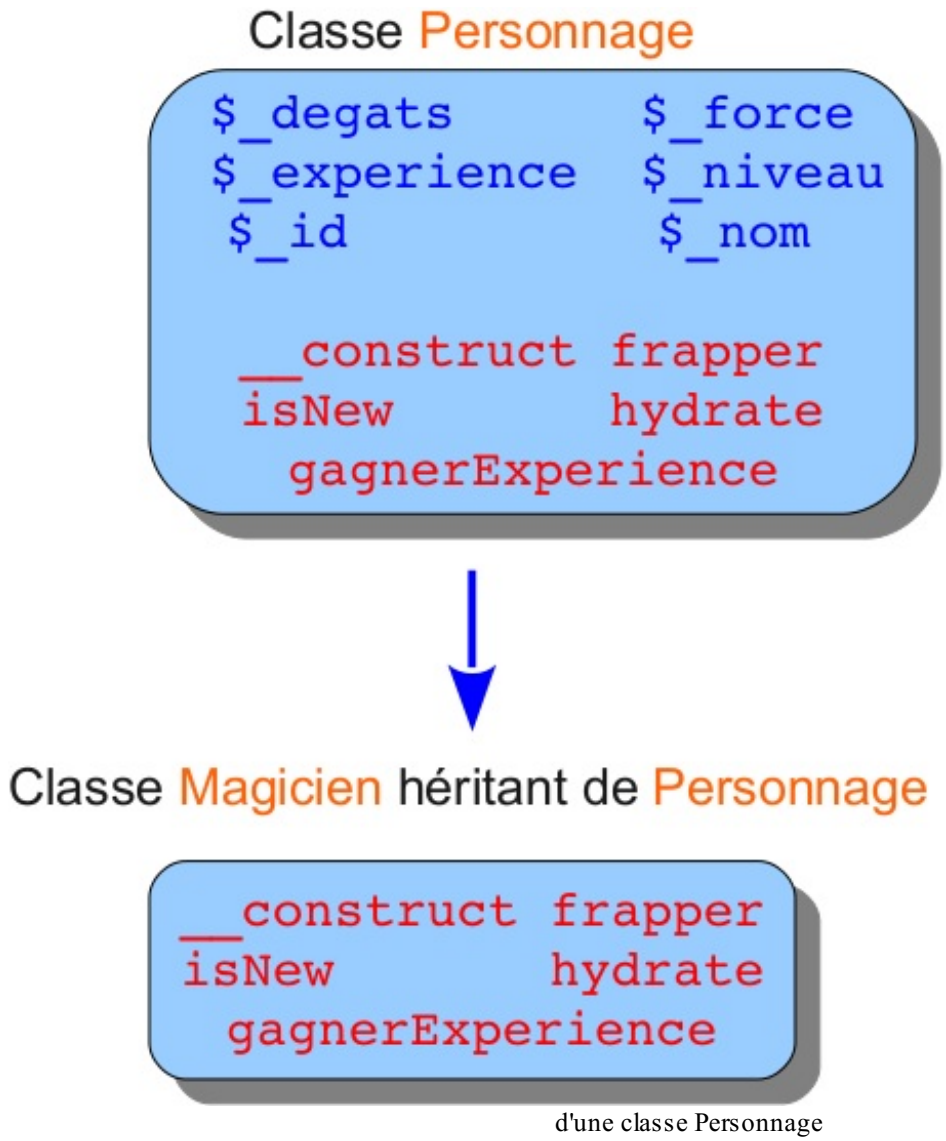
Définition

Quand on parle **d'héritage**, c'est qu'on dit qu'une classe B hérite d'une classe A. La classe A est donc considérée comme la classe **mère** et la classe B est considérée comme la classe **fille**.

Concrètement, l'héritage, c'est quoi ?

Lorsqu'on dit que la classe B **hérite** de la classe A, c'est que la classe B **hérite** de tous les attributs et méthodes de la classe A. Si l'on déclare des méthodes dans la classe A, et qu'on crée une instance de la classe B, alors on pourra appeler n'importe quelle méthode déclarée dans la classe A **du moment qu'elle est publique**.

Schématiquement, une classe B héritant d'une classe A peut être représentée comme ceci (figure suivante).



Exemple d'une classe Magicien héritant d'une classe Personnage

Vous voyez que la classe `Magicien` a hérité de **toutes** les méthodes et d'**aucun** attribut de la classe `Personnage`. Souvenez-vous : toutes les méthodes sont publiques et tous les attributs sont privés. En fait, les attributs privés ont bien été hérités aussi,

mais notre classe `Magicien` ne pourra s'en servir, c'est la raison pour laquelle je ne les ai pas représentés. Il n'y a que les méthodes de la classe **parente** qui auront accès à ces attributs. C'est comme pour le principe d'encapsulation : ici, les éléments privés sont **masqués**. On dit que la classe `Personnage` est la classe **mère** et que la classe `Magicien` est la classe **fille**.

Je n'ai pas mis toutes les méthodes du dernier TP dans ce schéma pour ne pas le surcharger, mais en réalité, toutes les méthodes ont bien été héritées.

Quand est-ce que je sais si telle classe doit hériter d'une autre ?

Soit deux classes `A` et `B`. Pour qu'un héritage soit possible, il faut que vous puissiez dire que `A` *est un* `B`. Par exemple, un magicien est un personnage, donc héritage. Un chien est un animal, donc héritage aussi. Bref, vous avez compris le principe.

Procéder à un héritage

Pour procéder à un héritage (c'est-à-dire faire en sorte qu'une classe hérite des attributs et méthodes d'une autre classe), il suffit d'utiliser le mot-clé `extends`. Vous déclarez votre classe comme d'habitude (`class MaClasse`) en ajoutant `extends NomDeLaClasseAHeriter` comme ceci :

Code : PHP

```php
<?php
class Personnage // Création d'une classe simple.
{

}

class Magicien extends Personnage // Notre classe Magicien hérite
des attributs et méthodes de Personnage.
{

}
?>
```

Comme dans la réalité, une mère peut avoir plusieurs filles, mais une fille ne peut avoir plusieurs mères. La seule différence avec la vie réelle, c'est qu'une mère ne peut avoir une infinité de filles.

Ainsi, on pourrait créer des classes `Magicien`, `Guerrier`, `Brute`, etc. qui héritent toutes de `Personnage` : la classe `Personnage` sert de **modèle**.

Code : PHP

```php
<?php
class Personnage // Création d'une classe simple.
{

}

// Toutes les classes suivantes hériteront de Personnage.

class Magicien extends Personnage
{

}

class Guerrier extends Personnage
{

}

class Brute extends Personnage
```

```
{

}
?>
```

Ainsi, toutes ces nouvelles classes auront les mêmes attributs et méthodes que `Personnage`.

Super, tu me crées des classes qui sont exactement les mêmes qu'une autre… Très utile ! En plus, tout ce qui est privé je n'y ai pas accès donc…

Nous sommes d'accord, si l'héritage c'était ça, ce serait le concept le plus idiot et le plus inutile de la POO.

Chaque classe peut créer des attributs et méthodes **qui lui seront propres**, et c'est là toute la puissance de l'héritage : toutes les classes que l'on a créées plus haut peuvent avoir des attributs et méthodes **en plus** des attributs et méthodes hérités. Pour cela, rien de plus simple. Il vous suffit de créer des attributs et méthodes comme nous avons appris jusqu'à maintenant. Un exemple ?

Code : PHP

```php
<?php
class Magicien extends Personnage
{
  private $_magie; // Indique la puissance du magicien sur 100, sa capacité à produire de la magie.

  public function lancerUnSort($perso)
  {
    $perso->recevoirDegats($this->_magie); // On va dire que la magie du magicien représente sa force.
  }
}
?>
```

Ainsi, la classe `Magicien` aura, **en plus des attributs et méthodes hérités**, un attribut `$magie` et une méthode `lancerUnSort`.

Si vous essayez d'accéder à un attribut privé de la classe parente, aucune erreur fatale ne s'affichera, seulement une notice si vous les avez activées disant que l'attribut n'existe pas.

Surcharger les méthodes

On vient de créer une classe `Magicien` héritant de toutes les méthodes de la classe `Personnage`. Que diriez-vous si l'on pouvait **réécrire** certaines méthodes, afin de modifier leur comportement ? Pour cela, il vous suffit de déclarer à nouveau la méthode et d'écrire ce que bon vous semble à l'intérieur.

Un problème se pose pourtant. Si vous voulez accéder à un attribut de la classe parente pour le modifier, vous ne pourrez pas car notre classe `Magicien` n'a pas accès aux attributs de sa classe mère `Personnage` puisqu'ils sont tous privés.

Nous allons maintenant essayer de surcharger la méthode `gagnerExperience` afin de modifier l'attribut stockant la magie (admettons **`$_magie`**) lorsque, justement, on gagne de l'expérience. Problème : si on la réécrit, nous écrasons toutes les instructions présentes dans la méthode de la classe parente (`Personnage`), ce qui aura pour effet de ne pas faire gagner d'expérience à notre magicien mais juste de lui augmenter sa magie. Solution : appeler la méthode `gagnerExperience` de la classe parente, puis ensuite ajouter les instructions modifiant la magie.

Il suffit pour cela d'utiliser le mot-clé `parent` suivi du symbole double deux points (le revoilà celui-là) suivi lui-même du nom de la méthode à appeler.

Code : PHP

```php
<?php
class Magicien extends Personnage
{
  private $_magie; // Indique la puissance du magicien sur 100, sa capacité à produire de la magie.

  public function lancerUnSort($perso)
  {
    $perso->recevoirDegats($this->_magie); // On va dire que la magie du magicien représente sa force.
  }

  public function gagnerExperience()
  {
    // On appelle la méthode gagnerExperience() de la classe parente
parent::gagnerExperience();

    if ($this->_magie < 100)
    {
      $this->_magie += 10;
    }
  }
}
?>
```

Notez que si la méthode parente retourne une valeur, vous pouvez la récupérer comme si vous appeliez une méthode normalement. Exemple :

Code : PHP

```php
<?php
class A
{
  public function test()
  {
    return 'test';
  }
}

class B extends A
{
  public function test()
  {
    $retour = parent::test();

    echo $retour; // Affiche 'test'
  }
}
```

Comme vous pouvez le constater, j'ai fait appel aux getters et setters correspondant à l'attribut **`$_magie`**. Pourquoi ? Car les classes enfant n'ont pas accès aux éléments privés, il fallait donc que la classe parente le fasse pour moi ! Il n'y a que les méthodes de la classe parente non réécrites qui ont accès aux éléments privés. À partir du moment où l'on réécrit une méthode de la classe parente, la méthode appartient à la classe fille et n'a donc plus accès aux éléments privés.

Si vous surchargez une méthode, sa visibilité doit être la même que dans la classe parente ! Si tel n'est pas le cas, une erreur fatale sera levée. Par exemple, vous ne pouvez surcharger une méthode publique en disant qu'elle est privée.

Héritez à l'infini !

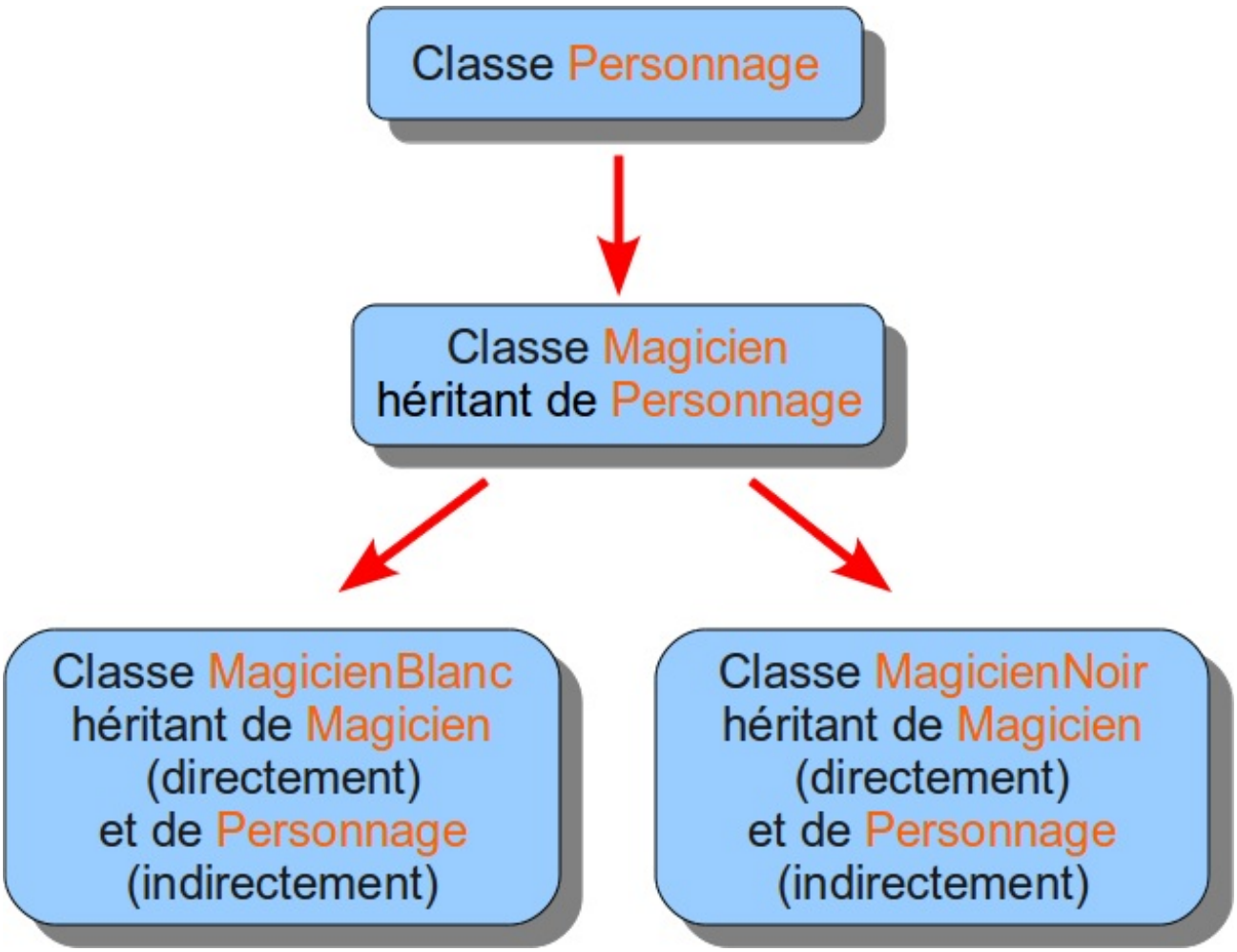
Toute classe en POO peut être héritée si elle ne spécifie pas le contraire, **vraiment toutes**. Vous pouvez ainsi reproduire un arbre réel avec autant de classes héritant les unes des autres que vous le souhaitez.

Pour reprendre l'exemple du magicien dans le cours sur la POO en C++ de M@teo21, on peut créer deux autres classes `MagicienBlanc` et `MagicienNoir` qui héritent toutes les deux de `Magicien`. Exemple :

Code : PHP

```php
<?php
class Personnage // Classe Personnage de base.
{

}

class Magicien extends Personnage // Classe Magicien héritant de
Personnage.
{

}

class MagicienBlanc extends Magicien // Classe MagicienBlanc
héritant de Magicien.
{

}

class MagicienNoir extends Magicien // Classe MagicienNoir héritant
de Magicien.
{

}
?>
```

Et un petit schéma qui reproduit ce code (figure suivante).



Exemple d'un héritage multiple

Ainsi, les classes `MagicienBlanc` et `MagicienNoir` hériteront de tous les attributs et de toutes les méthodes des classes `Magicien` et `Personnage`.

Un nouveau type de visibilité : protected

Je vais à présent vous présenter le dernier type de visibilité existant en POO : il s'agit de `protected`. Ce type de visibilité est, au niveau restrictif, à placer entre `public` et `private`.

Je vous rappelle brièvement les rôles de ces deux portées de visibilité :

- `Public` : ce type de visibilité nous laisse beaucoup de liberté. On peut accéder à l'attribut ou à la méthode de **n'importe où**. Toute classe fille aura accès aux éléments publics.
- `Private` : ce type est celui qui nous laisse le moins de liberté. On ne peut accéder à l'attribut ou à la méthode **que depuis l'intérieur de la classe qui l'a créé**. Toute classe fille n'aura pas accès aux éléments privés.

Le type de visibilité `protected` est en fait une petite modification du type `private` : il a **exactement** les mêmes effets que `private`, à l'exception que toute classe fille aura accès aux éléments protégés.

Exemple :

Code : PHP

```php
<?php
class ClasseMere
{
  protected $attributProtege;
  private $_attributPrive;

  public function __construct()
```

```
    {
      $this->attributProtege = 'Hello world !';
      $this->_attributPrive = 'Bonjour tout le monde !';
    }
}

class ClasseFille extends ClasseMere
{
  public function afficherAttributs()
  {
    echo $this->attributProtege; // L'attribut est protégé, on a
donc accès à celui-ci.
    echo $this->_atributPrive; // L'attribut est privé, on n'a pas
accès celui-ci, donc rien ne s'affichera (mis à part une notice si
vous les avez activées).
  }
}

$obj = new ClasseFille;

echo $obj->attributProtege; // Erreur fatale.
echo $obj->_attributPrive; // Rien ne s'affiche (ou une notice si
vous les avez activées).

$obj->afficherAttributs(); // Affiche « Hello world ! » suivi de
rien du tout ou d'une notice si vous les avez activées.
?>
```

Comme vous pouvez le constater, il n'y a pas d'*underscores* précédant les noms d'éléments protégés. C'est encore une fois la notation PEAR qui nous dit que les noms d'éléments protégés ne sont pas protégés de ce caractère.

Et pour le principe d'encapsulation, j'utilise quoi ? `private` ou `protected` ?

La portée `private` est, selon moi, bien trop restrictive et contraignante. Elle empêche toute classe enfant d'accéder aux attributs et méthodes privées alors que cette dernière en a souvent besoin. De manière générale, je vous conseille donc de toujours mettre `protected` au lieu de `private`, à moins que vous teniez absolument à ce que la classe enfant ne puisse y avoir accès. Cependant, je trouve cela inutile dans le sens où la classe enfant a été créée par un développeur, donc quelqu'un qui sait ce qu'il fait et qui par conséquent doit pouvoir modifier à souhait tous les attributs, contrairement à l'utilisateur de la classe.

Imposer des contraintes

Il est possible de mettre en place des contraintes. On parlera alors d'**abstraction** ou de **finalisation** suivant la contrainte instaurée.

Abstraction

Classes abstraites

On a vu jusqu'à maintenant que l'on pouvait *instancier* n'importe quelle classe afin de pouvoir exploiter ses méthodes. On va maintenant découvrir comment **empêcher quiconque d'*instancier* telle classe**.

Mais à quoi ça sert de créer une classe si on ne peut pas s'en servir ?

On ne pourra pas se servir **directement** de la classe. La seule façon d'exploiter ses méthodes est de créer une classe **héritant de la classe abstraite**.

Vous vous demandez sans doute à quoi cela peut bien servir. L'exemple que je vais prendre est celui du personnage et de ses classes filles. Dans ce que nous venons de faire, nous ne créerons **jamais** d'objet `Personnage`, mais uniquement des objets `Magicien`, `Guerrier`, `Brute`, etc. En effet, à quoi cela nous servirait d'instancier la classe `Personnage` si notre but est de créer un tel type de personnage ?

Nous allons donc considérer la classe `Personnage` comme étant une classe **modèle** dont toute classe fille possédera les méthodes et attributs.

Pour déclarer une classe abstraite, il suffit de faire précéder le mot-clé `class` du mot-clé `abstract` comme ceci :

Code : PHP

```php
<?php
abstract class Personnage // Notre classe Personnage est abstraite.
{

}

class Magicien extends Personnage // Création d'une classe Magicien héritant de la classe Personnage.
{

}

$magicien = new Magicien; // Tout va bien, la classe Magicien n'est pas abstraite.
$perso = new Personnage; // Erreur fatale car on instancie une classe abstraite.
?>
```

Simple et court à retenir, il suffit juste de se souvenir où l'on doit le placer.

Ainsi, si vous essayez de créer une instance de la classe `Personnage`, une erreur fatale sera levée. Ceci nous garantit que l'on ne créera jamais d'objet `Personnage` (suite à une étourderie par exemple).

Méthodes abstraites

Si vous décidez de rendre une méthode abstraite en plaçant le mot-clé `abstract` juste avant la visibilité de la méthode, vous forcerez toutes les classes filles à écrire cette méthode. Si tel n'est pas le cas, une erreur fatale sera levée. Puisque l'on force la classe fille à écrire la méthode, on ne doit spécifier aucune instruction dans la méthode, on déclarera juste son prototype (visibilité + `function` + nomDeLaMethode + parenthèses avec ou sans paramètres + **point-virgule**).

Code : PHP

```php
<?php
abstract class Personnage
{
  // On va forcer toute classe fille à écrire cette méthode car chaque personnage frappe différemment.
  abstract public function frapper(Personnage $perso);

  // Cette méthode n'aura pas besoin d'être réécrite.
  public function recevoirDegats()
  {
    // Instructions.
  }
}

class Magicien extends Personnage
{
  // On écrit la méthode « frapper » du même type de visibilité que la méthode abstraite « frapper » de la classe mère.
  public function frapper(Personnage $perso)
  {
    // Instructions.
  }
}
?>
```

Pour définir une méthode comme étant abstraite, il faut que la classe elle-même soit abstraite !

Finalisation

Classes finales

Le concept des classes et méthodes finales est exactement l'inverse du concept d'abstraction. Si une classe est finale, vous ne pourrez pas créer de classe fille héritant de cette classe.

Pour ma part, je ne rends jamais mes classes finales (au même titre que, à quelques exceptions près, je mets toujours mes attributs en `protected`) pour me laisser la liberté d'hériter de n'importe quelle classe.

Pour déclarer une classe finale, vous devez placer le mot-clé `final` juste avant le mot-clé `class`, comme `abstract`.

Code : PHP

```php
<?php
// Classe abstraite servant de modèle.

abstract class Personnage
{

}

// Classe finale, on ne pourra créer de classe héritant de
Guerrier.

final class Guerrier extends Personnage
{

}

// Erreur fatale, car notre classe hérite d'une classe finale.

class GentilGuerrier extends Guerrier
{

}
?>
```

Méthodes finales

Si vous déclarez une méthode finale, toute classe fille de la classe comportant cette méthode finale héritera de cette méthode **mais ne pourra la surcharger**. Si vous déclarez votre méthode `recevoirDegats` en tant que méthode finale, vous ne pourrez la surcharger.

Code : PHP

```php
<?php
abstract class Personnage
{
  // Méthode normale.

  public function frapper(Personnage $perso)
  {
    // Instructions.
  }
```

```php
  // Méthode finale.

  final public function recevoirDegats()
  {
    // Instructions.
  }
}

class Guerrier extends Personnage
{
  // Aucun problème.

  public function frapper(Personnage $perso)
  {
    // Instructions.
  }

  // Erreur fatale car cette méthode est finale dans la classe
parente.

  public function recevoirDegats()
  {
    // Instructions.
  }
}
?>
```

Résolution statique à la volée

Fonctionnalité disponible depuis PHP 5.3 !

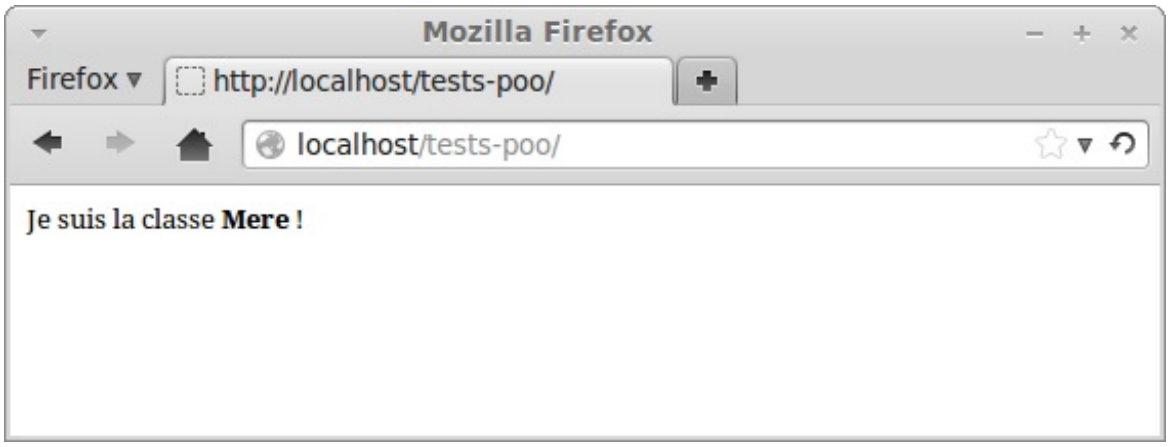
Cette sous-partie va vous montrer une possibilité intéressante de la POO en PHP : la résolution statique à la volée. C'est une notion un peu complexe à comprendre au premier abord donc n'hésitez pas à relire cette partie autant de fois que nécessaire.

Effectuons d'abord un petit flash-back sur `self::`. Vous vous souvenez à quoi il sert ? À appeler un attribut, une méthode statique ou une constante **de la classe dans laquelle est contenu `self::`**. Ainsi, si vous testez ce code :

Code : PHP

```php
<?php
class Mere
{
  public static function lancerLeTest()
  {
    self::quiEstCe();
  }

  public function quiEstCe()
  {
    echo 'Je suis la classe <strong>Mere</strong> !';
  }
}

class Enfant extends Mere
{
  public function quiEstCe()
  {
    echo 'Je suis la classe <strong>Enfant</strong> !';
  }
}

Enfant::lancerLeTest();
?>
```

À l'écran s'affichera :



Résultat affiché par le script

Mais qu'est-ce qui s'est passé ???

- Appel de la méthode `lancerLeTest` de la classe `Enfant` ;
- la méthode n'a pas été réécrite, on va donc « chercher » la méthode `lancerLeTest` de la classe mère ;
- appel de la méthode `quiEstCe` **de la classe `Mere`**.

Pourquoi c'est la méthode `quiEstCe` de la classe parente qui a été appelée ? Pourquoi pas celle de la classe fille puisqu'elle a été récrite ?

Tout simplement parce que `self::` fait appel à la méthode statique **de la classe dans laquelle est contenu `self::`**, donc de la classe parente.

Et la résolution statique à la volée dans tout ça ?

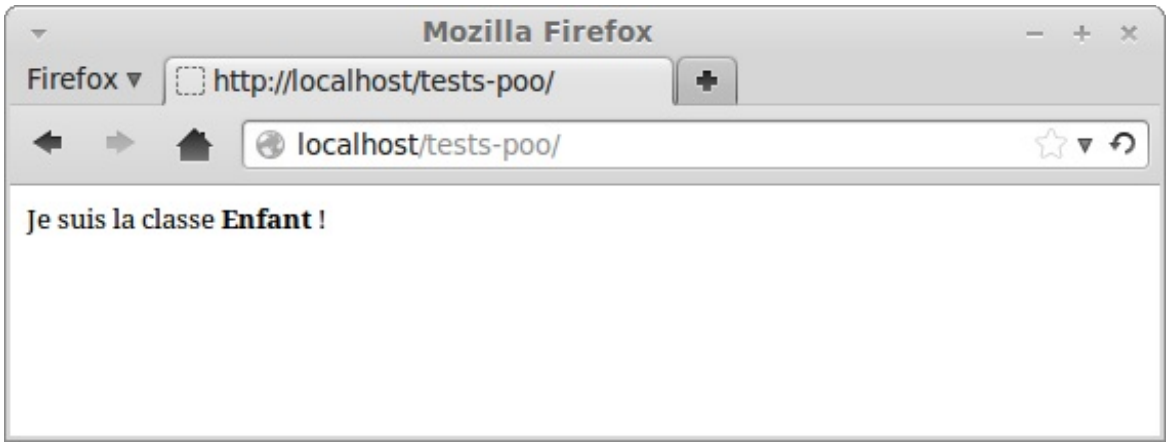
Tout tourne autour de l'utilisation de `static::`. `static::` a exactement le même effet que `self::`, **à l'exception près que `static::` appelle l'élément de la classe qui est appelée pendant l'exécution**. C'est-à-dire que si j'appelle la méthode `lancerLeTest` depuis la classe `Enfant` et que dans cette méthode j'utilise `static::` au lieu de `self::`, c'est la méthode `quiEstCe` de la classe `Enfant` qui sera appelée et non de la classe `Mere` !

Code : PHP

```php
<?php
class Mere
{
  public static function lancerLeTest()
  {
static::quiEstCe();
  }

  public function quiEstCe()
  {
    echo 'Je suis la classe <strong>Mere</strong> !';
  }
}

class Enfant extends Mere
```

```
{
  public function quiEstCe()
  {
    echo 'Je suis la classe <strong>Enfant</strong> !';
  }
}

Enfant::lancerLeTest();
?>
```

Ce qui donnera :



Résultat affiché par le script

Notez que tous les exemples ci-dessus utilisent des méthodes qui sont appelées dans un contexte statique. J'ai fait ce choix car pour ce genre de tests, il était inutile d'*instancier* la classe, mais sachez bien que la résolution statique à la volée a exactement le même effet quand on crée un objet puis qu'on appelle une méthode de celui-ci. Il n'est donc pas du tout obligatoire de rendre les méthodes statiques pour pouvoir y placer `static::`. Ainsi, si vous testez ce code, à l'écran s'affichera la même chose que précédemment.

Code : PHP

```
<?php
class Mere
{
  public function lancerLeTest()
  {
    static::quiEstCe();
  }

  public function quiEstCe()
  {
    echo 'Je suis la classe « Mere » !';
  }
}

class Enfant extends Mere
{
  public function quiEstCe()
  {
    echo 'Je suis la classe « Enfant » !';
  }
}

$e = new Enfant;
$e->lancerLeTest();
?>
```

Cas complexes

Au premier abord, vous n'avez peut-être pas tout compris. Si tel est le cas, ne lisez pas la suite cela risquerait de vous embrouiller davantage. Prenez bien le temps de comprendre ce qui est écrit plus haut puis vous pourrez continuer.

Comme le spécifie le titre, il y a quelques cas complexes (des pièges en quelque sorte). Imaginons trois classes A, B et C qui héritent chacune d'une autre (A est la grand-mère, B la mère et C la fille). En PHP, on dirait plutôt :

Code : PHP

```php
<?php
class A
{

}

class B extends A
{

}

class C extends B
{

}
?>
```

Nous allons implémenter dans chacune des classes une méthode qui aura pour rôle d'afficher le nom de la classe pour pouvoir effectuer quelques tests.

Code : PHP

```php
<?php
class A
{
  public function quiEstCe()
  {
    echo 'A';
  }
}

class B extends A
{
  public function quiEstCe()
  {
    echo 'B';
  }
}

class C extends B
{
  public function quiEstCe()
  {
    echo 'C';
  }
}
?>
```

Créons une méthode de test dans la classe B. Pourquoi dans cette classe ? Parce qu'elle hérite à la fois de A et est héritée par C,

son cas est donc intéressant à étudier.

Appelons maintenant cette méthode depuis la classe C dans un contexte statique (nul besoin de créer d'objet, mais ça marche tout aussi bien).

Code : PHP

```php
<?php
class A
{
  public function quiEstCe()
  {
    echo 'A';
  }
}

class B extends A
{
  public static function test()
  {

  }

  public function quiEstCe()
  {
    echo 'B';
  }
}

class C extends B
{
  public function quiEstCe()
  {
    echo 'C';
  }
}

C::test();
?>
```

Plaçons un peu de code dans cette méthode, sinon c'est pas drôle.

Nous allons essayer d'appeler la méthode parente `quiEstCe`. Là, il n'y a pas de piège, pas de résolution statique à la volée, donc à l'écran s'affichera « A » :

Code : PHP

```php
<?php
class A
{
  public function quiEstCe()
  {
    echo 'A';
  }
}

class B extends A
{
  public static function test()
  {
    parent::quiEstCe();
  }

  public function quiEstCe()
```

```
    {
      echo 'B';
    }
}

class C extends B
{
  public function quiEstCe()
  {
    echo 'C';
  }
}

C::test();
?>
```

Maintenant, créons une méthode dans la classe A qui sera chargée d'appeler la méthode `quiEstCe` avec `static::`. Là, si vous savez ce qui va s'afficher, vous avez tout compris ! 

Code : PHP

```
<?php
class A
{
  public function appelerQuiEstCe()
  {
    static::quiEstCe();
  }

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

class B extends A
{
  public static function test()
  {
    parent::appelerQuiEstCe();
  }

  public function quiEstCe()
  {
    echo 'B';
  }
}

class C extends B
{
  public function quiEstCe()
  {
    echo 'C';
  }
}

C::test();
?>
```

Alors ? Vous avez une petite idée ? À l'écran s'affichera… **C** ! Décortiquons ce qui s'est passé :

- Appel de la méthode `test` de la classe `C` ;

- la méthode n'a pas été réécrite, on appelle donc la méthode `test` de la classe `B` ;
- on appelle maintenant la méthode `appelerQuiEstCe` de la classe `A` (avec `parent::`) ;
- résolution statique à la volée : on appelle la méthode `quiEstCe` de la classe qui a appelé la méthode `appelerQuiEstCe` ;
- la méthode `quiEstCe` de la classe `C` est donc appelée car c'est depuis la classe `C` qu'on a appelé la méthode `test`.

C'est très compliqué mais fondamental à comprendre.

Remplaçons maintenant `parent::` par `self::` :

Code : PHP

```php
<?php
class A
{
  public function appelerQuiEstCe()
  {
    static::quiEstCe();
  }

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

class B extends A
{
  public static function test()
  {
    self::appelerQuiEstCe();
  }

  public function quiEstCe()
  {
    echo 'B';
  }
}

class C extends B
{
  public function quiEstCe()
  {
    echo 'C';
  }
}

C::test();
?>
```

Et là, qu'est-ce qui s'affiche à l'écran ? Et bien toujours **C** ! Le principe est **exactement** le même que le code plus haut.

Si vous cassez la chaîne en appelant une méthode depuis une instance ou statiquement du genre `Classe::methode()`, la méthode appelée par `static::` sera celle de la classe contenant ce code ! Ainsi, le code suivant affichera « A ».

Code : PHP

```php
<?php
class A
{
```

```
  public static function appelerQuiEstCe()
  {
    static::quiEstCe();
  }

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

class B extends A
{
  public static function test()
  {
    // On appelle « appelerQuiEstCe » de la classe « A »
normalement.
    A::appelerQuiEstCe();
  }

  public function quiEstCe()
  {
    echo 'B';
  }
}

class C extends B
{
  public function quiEstCe()
  {
    echo 'C';
  }
}

C::test();
?>
```

Utilisation de *static::* dans un contexte non statique

L'utilisation de `static::` dans un contexte non statique se fait de la même façon que dans un contexte statique. Je vais prendre l'exemple de la documentation pour illustrer mes propos :

Code : PHP

```
<?php
class TestChild extends TestParent
{
  public function __construct()
  {
    static::qui();
  }

  public function test()
  {
    $o = new TestParent();
  }

  public static function qui()
  {
    echo 'TestChild';
  }
}

class TestParent
{
  public function __construct()
```

```
    {
      static::qui();
    }

    public static function qui()
    {
      echo 'TestParent';
    }
}

$o = new TestChild;
$o->test();
?>
```

À l'écran s'affichera « TestChild » suivi de « TestParent ». Je vous explique ce qui s'est passé si vous n'avez pas tout suivi :

- Création d'une instance de la classe `TestChild` ;
- appel de la méthode `qui` de la classe `TestChild` puisque c'est la méthode `__construct` de la classe `TestChild` qui a été appelée ;
- appel de la méthode `test` de la classe `TestChild` ;
- création d'une instance de la classe `TestParent` ;
- appel de la méthode `qui` de la classe `TestParent` puisque c'est la méthode `__construct` de cette classe qui a été appelée.

Ouf ! Enfin terminé ! 

N'hésitez pas à le relire autant de fois que nécessaire afin de bien comprendre cette notion d'héritage et toutes les possibilités que ce concept vous offre. Ne soyez pas pressés de continuer si vous n'avez pas tout compris, sinon vous allez vous planter au TP. 

En résumé

- Nous pouvons parler d'héritage entre une classe A et une classe B si et seulement si nous pouvons dire « B est un A » (dans ce cas-là, B hérite de A).
- Une classe héritant d'une autre a accès à tous ses attributs et méthodes publics ou protégés.
- La visibilité `protected` est équivalente à la visibilité `private` à la différence près qu'un élément protégé est accessible par les classes filles, contrairement aux éléments privés.
- Il est possible d'interdire l'instanciation d'une classe grâce au mot-clé `abstract` (utile pour poser un modèle de base commun à plusieurs classes) et l'héritage d'une classe grâce au mot-clé `final`.
- Il est possible d'obliger ses classes filles à implémenter une méthode grâce au mot-clé `abstract` et de leur interdire la réécriture d'une méthode grâce au mot-clé `final`.
- La résolution statique à la volée permet de savoir quelle classe a été initialement appelée pour invoquer la méthode dans laquelle on se trouve.

TP : Des personnages spécialisés

Avez-vous bien tout retenu du dernier chapitre ? C'est ce que l'on va vérifier maintenant ! En effet, l'héritage est un point très important de la POO qu'il est essentiel de maîtriser, c'est pourquoi je souhaite insister dessus à travers ce TP. Ce dernier est en fait une modification du premier (celui qui mettait en scène notre personnage). Nous allons donc ajouter une possibilité supplémentaire au script : le choix du personnage. Cette modification vous permettra de revoir ce que nous avons abordé depuis le dernier TP, à savoir :

- L'héritage ;
- la portée `protected` ;
- l'abstraction.

Je ne mettrai pas en pratique la résolution statique à la volée car elle ne nous est pas utile ici. Aussi, la finalisation n'est pas utilisée car c'est davantage une contrainte inutile qu'autre chose dans cette situation.

Ce que nous allons faire

Cahier des charges

Je veux que nous ayons le choix de créer un certain type de personnage qui aura certains avantages. Il ne doit pas être possible de créer un personnage « normal » (donc il devra être impossible d'instancier la classe `Personnage`). Comme précédemment, la classe `Personnage` aura la liste des colonnes de la table en guise d'attributs.

Je vous donne une liste de personnages différents qui pourront être créés. Chaque personnage a un atout différent sous forme d'entier.

- Un magicien. Il aura une nouvelle fonctionnalité : celle de lancer un sort qui aura pour effet d'endormir un personnage pendant `$atout * 6` heures (l'attribut `$atout` représente la dose de magie du personnage).
- Un guerrier. Lorsqu'un coup lui est porté, il devra avoir la possibilité de parer le coup en fonction de sa protection (son atout).

Ceci n'est qu'une petite liste de départ. Libre à vous de créer d'autres personnages.

Comme vous le voyez, chaque personnage possède un **atout**. Cet atout devra être augmenté lorsque le personnage est amené à s'en servir (c'est-à-dire lorsque le magicien lance un sort ou que le guerrier subit des dégâts).

Des nouvelles fonctionnalités pour chaque personnage

Étant donné qu'un magicien peut endormir un personnage, il est nécessaire d'implémenter deux nouvelles fonctionnalités :

- Celle consistant à savoir si un personnage est endormi ou non (nécessaire lorsque ledit personnage voudra en frapper un autre : s'il est endormi, ça ne doit pas être possible).
- Celle consistant à obtenir la date du réveil du personnage sous la forme « XX heures, YY minutes et ZZ secondes », qui s'affichera dans le cadre d'information du personnage s'il est endormi.

La base de données

La structure de la BDD ne sera pas la même. En effet, chaque personnage aura un attribut en plus, et surtout, il faut savoir de quel personnage il s'agit (magicien ou guerrier). Nous allons donc créer une colonne **type** et une colonne **atout** (l'attribut qu'il a en plus). Une colonne **timeEndormi** devra aussi être créée pour stocker le *timestamp* auquel le personnage se réveillera s'il a été ensorcelé. Je vous propose donc cette nouvelle structure (j'ai juste ajouté trois nouveaux champs en fin de table) :

Code : SQL

```sql
CREATE TABLE IF NOT EXISTS `personnages_v2` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `nom` varchar(50) COLLATE latin1_general_ci NOT NULL,
```

```
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',
  `timeEndormi` int(10) unsigned NOT NULL DEFAULT '0',
  `type` enum('magicien','guerrier') COLLATE latin1_general_ci NOT
NULL,
  `atout` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci;
```

Le coup de pouce du démarrage

Les modifications que je vous demande peuvent vous donner mal à la tête, c'est pourquoi je me propose de vous mettre sur la voie. Procédons classe par classe.

Le personnage

Là où il y a peut-être une difficulté, c'est pour déterminer l'attribut `$type`. En effet, où doit-on assigner cette valeur ? Qui doit le faire ?

Pour des raisons évidentes de risques de bugs, ce ne sera pas à l'utilisateur d'assigner la valeur « guerrier » ou « magicien » à cet attribut, mais à la classe elle-même. Cependant, il serait redondant dans chaque classe fille de **Personnage** de faire un `<?php $this->type = 'magicien'` par exemple, alors on va le faire une bonne fois pour toute dans la classe **Personnage**.

Comment cela est-il possible ?

Grâce à une fonction bien pratique (nommée get_class()), il est possible d'obtenir le nom d'une classe à partir d'une instance de cette classe. Par exemple, si on instancie une classe `Guerrier`, alors `get_class($instance)` renverra « Guerrier ». Ici, nous nous situons dans le constructeur de la classe `Personnage` : l'instance du personnage sera donc représentée par... `$this` ! Eh oui, `get_class($this)`, dans le constructeur de `Personnage`, ne renverra pas « Personnage », mais « Guerrier » ou « Magicien ». Pourquoi ? Car `get_class()` ne renvoie pas le nom de la classe dans laquelle elle est appelée, mais le nom de la classe instanciée par l'objet passé en argument. Or, l'instance `$this` n'est autre qu'une instance de `Guerrier` ou `Magicien`.

Pour des raisons pratiques, le type du personnage doit être en minuscules. Un petit appel à strtolower() sera donc indispensable.

Le magicien

Qui dit nouvelle fonctionnalité dit nouvelle méthode. Votre classe `Magicien` devra donc implémenter une nouvelle méthode permettant de lancer un sort. Celle-ci devra vérifier plusieurs points :

- La cible à ensorceler n'est pas le magicien qui lance le sort.
- Le magicien possède encore de la magie (l'atout n'est pas à 0).

Le guerrier

Ce qu'on cherche à faire ici est **modifier** le comportement du personnage lorsqu'il subit des dégâts. Nous allons donc **modifier** la méthode qui se charge d'ajouter des dégâts au personnage. Cette méthode procédera de la sorte :

- Elle calculera d'abord la valeur de l'atout.
- Elle augmentera les dégâts en prenant soin de prendre en compte l'atout.
- Elle indiquera si le personnage a été frappé ou tué.

Tu nous parles d'un atout depuis tout à l'heure, mais comment est-ce qu'on le détermine ?

L'atout du magicien et du guerrier se déterminent de la même façon :

- Si les dégâts sont compris entre 0 et 25, alors l'atout sera de 4.
- Si les dégâts sont compris entre 25 et 50, alors l'atout sera de 3.
- Si les dégâts sont compris entre 50 et 75, alors l'atout sera de 2.
- Si les dégâts sont compris entre 75 et 90, alors l'atout sera de 1.
- Sinon, il sera de 0.

Comment sont-ils exploités ?

Du côté du guerrier, j'utilise une simple formule : la dose de dégâts reçu ne sera pas de 5, mais de `5 - $atout`. Du côté du magicien, là aussi j'utilise une simple formule : il endort sa victime pendant `($this->atout * 6) * 3600` secondes.

Voir le résultat que vous devez obtenir

Comme au précédent TP, le résultat comporte toutes les améliorations proposées en fin de chapitre.

Correction

Nous allons maintenant corriger le TP. Les codes seront d'abord précédés d'explications pour bien mettre au clair ce qui était demandé.

Commençons d'abord par notre classe `Personnage`. Celle-ci devait implémenter deux nouvelles méthodes (sans compter les *getters* et *setters*) : `estEndormi()` et `reveil()`. Aussi il ne fallait pas oublier de modifier la méthode `frapper()` afin de bien vérifier que le personnage qui frappe n'était pas endormi ! Enfin, il fallait assigner la bonne valeur à l'attribut `$type` dans le constructeur.

Il ne fallait pas mettre de *setter* pour l'attribut `$type`. En effet, le type d'un personnage est constant (un magicien ne se transforme pas en guerrier). Il est défini dans le constructeur et l'utilisateur ne pourra pas changer sa valeur (imaginez le non-sens qu'il y aurait si l'utilisateur mettait « magicien » à l'attribut `$type` d'un objet `Guerrier`).

Code : PHP - Personnage.class.php

```php
<?php
abstract class Personnage
{
  protected $atout,
            $degats,
            $id,
            $nom,
            $timeEndormi,
            $type;

  const CEST_MOI = 1; // Constante renvoyée par la méthode `frapper` si on se frappe soit-même.
  const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode `frapper` si on a tué le personnage en le frappant.
  const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode `frapper` si on a bien frappé le personnage.
  const PERSONNAGE_ENSORCELE = 4; // Constante renvoyée par la méthode `lancerUnSort` (voir classe Magicien) si on a bien ensorcelé un personnage.
  const PAS_DE_MAGIE = 5; // Constante renvoyée par la méthode `lancerUnSort` (voir classe Magicien) si on veut jeter un sort alors que la magie du magicien est à 0.
  const PERSO_ENDORMI = 6; // Constante renvoyée par la méthode `frapper` si le personnage qui veut frapper est endormi.

  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
    $this->type = strtolower(get_class($this));
  }
```

```
  public function estEndormi()
  {
    return $this->timeEndormi > time();
  }

  public function frapper(Personnage $perso)
  {
    if ($perso->id == $this->id)
    {
      return self::CEST_MOI;
    }

    if ($this->estEndormi())
    {
      return self::PERSO_ENDORMI;
    }

    // On indique au personnage qu'il doit recevoir des dégâts.
    // Puis on retourne la valeur renvoyée par la méthode :
self::PERSONNAGE_TUE ou self::PERSONNAGE_FRAPPE.
    return $perso->recevoirDegats();
  }

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  public function nomValide()
  {
    return !empty($this->nom);
  }

  public function recevoirDegats($force)
  {
    $this->degats += 5;

    // Si on a 100 de dégâts ou plus, on supprime le personnage de
la BDD.
    if ($this->degats >= 100)
    {
      return self::PERSONNAGE_TUE;
    }

    // Sinon, on se contente de mettre à jour les dégâts du
personnage.
    return self::PERSONNAGE_FRAPPE;
  }

  public function reveil()
  {
    $secondes = $this->timeEndormi;
    $secondes -= time();

    $heures = floor($secondes / 3600);
    $secondes -= $heures * 3600;
    $minutes = floor($secondes / 60);
    $secondes -= $minutes * 60;

    $heures .= $heures <= 1 ? ' heure' : ' heures';
    $minutes .= $minutes <= 1 ? ' minute' : ' minutes';
```

```
        $secondes .= $secondes <= 1 ? ' seconde' : ' secondes';

        return $heures . ', ' . $minutes . ' et ' . $secondes;
    }

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

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

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

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

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

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

    public function setAtout($atout)
    {
        $atout = (int) $atout;

        if ($atout >= 0 && $atout <= 100)
        {
            $this->atout = $atout;
        }
    }

    public function setDegats($degats)
    {
        $degats = (int) $degats;

        if ($degats >= 0 && $degats <= 100)
        {
            $this->degats = $degats;
        }
    }

    public function setId($id)
    {
        $id = (int) $id;

        if ($id > 0)
        {
            $this->id = $id;
        }
    }

    public function setNom($nom)
    {
        if (is_string($nom))
        {
            $this->nom = $nom;
```

```
    }
  }

  public function setTimeEndormi($time)
  {
    $this->timeEndormi = (int) $time;
  }
}
```

Penchons-nous maintenant vers la classe `Guerrier`. Celle-ci devait modifier la méthode `recevoirDegats()` afin d'ajouter une parade lors d'une attaque.

Code : PHP - Guerrier.class.php

```
<?php
class Guerrier extends Personnage
{
  public function recevoirDegats($force)
  {
    if ($this->degats >= 0 && $this->degats <= 25)
    {
      $this->atout = 4;
    }
    elseif ($this->degats > 25 && $this->degats <= 50)
    {
      $this->atout = 3;
    }
    elseif ($this->degats > 50 && $this->degats <= 75)
    {
      $this->atout = 2;
    }
    elseif ($this->degats > 50 && $this->degats <= 90)
    {
      $this->atout = 1;
    }
    else
    {
      $this->atout = 0;
    }

    $this->degats += 5 - $this->atout;

    // Si on a 100 de dégâts ou plus, on supprime le personnage de
la BDD.
    if ($this->degats >= 100)
    {
      return self::PERSONNAGE_TUE;
    }

    // Sinon, on se contente de mettre à jour les dégâts du
personnage.
    return self::PERSONNAGE_FRAPPE;
  }
}
```

Enfin, il faut maintenant s'occuper du magicien. La classe le représentant devait ajouter une nouvelle fonctionnalité : celle de pouvoir lancer un sort.

Code : PHP - Magicien.class.php

```
<?php
class Magicien extends Personnage
{
```

```
  public function lancerUnSort(Personnage $perso)
  {
    if ($this->degats >= 0 && $this->degats <= 25)
    {
      $this->atout = 4;
    }
    elseif ($this->degats > 25 && $this->degats <= 50)
    {
      $this->atout = 3;
    }
    elseif ($this->degats > 50 && $this->degats <= 75)
    {
      $this->atout = 2;
    }
    elseif ($this->degats > 50 && $this->degats <= 90)
    {
      $this->atout = 1;
    }
    else
    {
      $this->atout = 0;
    }

    if ($perso->id == $this->id)
    {
      return self::CEST_MOI;
    }

    if ($this->atout == 0)
    {
      return self::PAS_DE_MAGIE;
    }

    if ($this->estEndormi())
    {
      return self::PERSO_ENDORMI;
    }

    $perso->timeEndormi = time() + ($this->atout * 6) * 3600;

    return self::PERSONNAGE_ENSORCELE;
  }
}
```

Passons maintenant au manager. Nous allons toujours garder un seul manager parce que nous gérons toujours des personnages ayant la même structure. Les modifications sont mineures : il faut juste ajouter les deux nouveaux champs dans les requêtes et instancier la bonne classe lorsqu'on récupère un personnage.

Code : PHP - PersonnagesManager.class.php

```
<?php
class PersonnagesManager
{
  private $db; // Instance de PDO

  public function __construct($db)
  {
    $this->db = $db;
  }

  public function add(Personnage $perso)
  {
    $q = $this->db->prepare('INSERT INTO personnages_v2 SET nom =
:nom, type = :type');

    $q->bindValue(':nom', $perso->nom());
```

```
    $q->bindValue(':type', $perso->type());

    $q->execute();

    $perso->hydrate(array(
      'id' => $this->db->lastInsertId(),
      'degats' => 0,
      'atout' => 0
    ));
  }

  public function count()
  {
    return $this->db->query('SELECT COUNT(*) FROM personnages_v2')->fetchColumn();
  }

  public function delete(Personnage $perso)
  {
    $this->db->exec('DELETE FROM personnages_v2 WHERE id = '.$perso->id());
  }

  public function exists($info)
  {
    if (is_int($info)) // On veut voir si tel personnage ayant pour id $info existe.
    {
      return (bool) $this->db->query('SELECT COUNT(*) FROM personnages_v2 WHERE id = '.$info)->fetchColumn();
    }

    // Sinon, c'est qu'on veut vérifier que le nom existe ou pas.

    $q = $this->db->prepare('SELECT COUNT(*) FROM personnages_v2 WHERE nom = :nom');
    $q->execute(array(':nom' => $info));

    return (bool) $q->fetchColumn();
  }

  public function get($info)
  {
    if (is_int($info))
    {
      $q = $this->db->query('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE id = '.$info);
      $perso = $q->fetch(PDO::FETCH_ASSOC);
    }

    else
    {
      $q = $this->db->prepare('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE nom = :nom');
      $q->execute(array(':nom' => $info));

      $perso = $q->fetch(PDO::FETCH_ASSOC);
    }

    switch ($perso['type'])
    {
      case 'guerrier': return new Guerrier($perso);
      case 'magicien': return new Magicien($perso);
      default: return null;
    }
  }

  public function getList($nom)
  {
    $persos = array();
```

```
    $q = $this->db->prepare('SELECT id, nom, degats, timeEndormi,
type, atout FROM personnages_v2 WHERE nom <> :nom ORDER BY nom');
    $q->execute(array(':nom' => $nom));

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      switch ($donnees['type'])
      {
        case 'guerrier': $persos[] = new Guerrier($donnees); break;
        case 'magicien': $persos[] = new Magicien($donnees); break;
      }
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->db->prepare('UPDATE personnages_v2 SET degats =
:degats, timeEndormi = :timeEndormi, atout = :atout WHERE id =
:id');

    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':timeEndormi', $perso->timeEndormi(),
PDO::PARAM_INT);
    $q->bindValue(':atout', $perso->atout(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }
}
```

Enfin, finissons par la page d'index qui a légèrement changé. Quelles sont les modifications ?

- Le formulaire doit proposer au joueur de choisir le type du personnage qu'il veut créer.
- Lorsqu'on crée un personnage, le script doit créer une instance de la classe désirée et non de `Personnage`.
- Lorsqu'on veut frapper un personnage, on vérifie que l'attaquant n'est pas endormi.
- Un lien permettant d'ensorceler un personnage doit être ajouté pour les magiciens ainsi que l'antidote qui va avec.

Code : PHP - index.php

```
<?php
function chargerClasse($classe)
{
  require $classe . '.class.php';
}

spl_autoload_register('chargerClasse');

session_start();

if (isset($_GET['deconnexion']))
{
  session_destroy();
  header('Location: .');
  exit();
}

$db = new PDO('mysql:host=localhost;dbname=combats', 'root', '');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);

$manager = new PersonnagesManager($db);
```

```
if (isset($_SESSION['perso'])) // Si la session perso existe, on
restaure l'objet.
{
  $perso = $_SESSION['perso'];
}

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu
créer un personnage.
{
  switch ($_POST['type'])
  {
    case 'magicien' :
      $perso = new Magicien(array('nom' => $_POST['nom']));
      break;

    case 'guerrier' :
      $perso = new Guerrier(array('nom' => $_POST['nom']));
      break;

    default :
      $message = 'Le type du personnage est invalide.';
      break;
  }

  if (isset($perso)) // Si le type du personnage est valide, on a
créé un personnage.
  {
    if (!$perso->nomValide())
    {
      $message = 'Le nom choisi est invalide.';
      unset($perso);
    }
    elseif ($manager->exists($perso->nom()))
    {
      $message = 'Le nom du personnage est déjà pris.';
      unset($perso);
    }
    else
    {
      $manager->add($perso);
    }
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on
a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe
pas, on affichera ce message.
  }
}

elseif (isset($_GET['frapper'])) // Si on a cliqué sur un
personnage pour le frapper.
{
  if (!isset($perso))
  {
    $message = 'Merci de créer un personnage ou de vous
identifier.';
  }

  else
  {
    if (!$manager->exists((int) $_GET['frapper']))
```

```
        {
          $message = 'Le personnage que vous voulez frapper n\'existe
pas !';
        }

        else
        {
          $persoAFrapper = $manager->get((int) $_GET['frapper']);
          $retour = $perso->frapper($persoAFrapper); // On stocke dans
$retour les éventuelles erreurs ou messages que renvoie la méthode
frapper.

          switch ($retour)
          {
            case Personnage::CEST_MOI :
              $message = 'Mais... pourquoi voulez-vous vous frapper ???
';
              break;

            case Personnage::PERSONNAGE_FRAPPE :
              $message = 'Le personnage a bien été frappé !';

              $manager->update($perso);
              $manager->update($persoAFrapper);

              break;

            case Personnage::PERSONNAGE_TUE :
              $message = 'Vous avez tué ce personnage !';

              $manager->update($perso);
              $manager->delete($persoAFrapper);

              break;

            case Personnage::PERSO_ENDORMI :
              $message = 'Vous êtes endormi, vous ne pouvez pas frapper
de personnage !';
              break;
          }
        }
      }
    }

    elseif (isset($_GET['ensorceler']))
    {
      if (!isset($perso))
      {
        $message = 'Merci de créer un personnage ou de vous
identifier.';
      }

      else
      {
        // Il faut bien vérifier que le personnage est un magicien.
        if ($perso->type() != 'magicien')
        {
          $message = 'Seuls les magiciens peuvent ensorceler des
personnages !';
        }

        else
        {
          if (!$manager->exists((int) $_GET['ensorceler']))
          {
            $message = 'Le personnage que vous voulez frapper n\'existe
pas !';
          }

          else
```

```
        {
          $persoAEnsorceler = $manager->get((int)
$_GET['ensorceler']);
          $retour = $perso->lancerUnSort($persoAEnsorceler);

          switch ($retour)
          {
            case Personnage::CEST_MOI :
              $message = 'Mais... pourquoi voulez-vous vous ensorceler
???';
              break;

            case Personnage::PERSONNAGE_ENSORCELE :
              $message = 'Le personnage a bien été ensorcelé !';

              $manager->update($perso);
              $manager->update($persoAEnsorceler);

              break;

            case Personnage::PAS_DE_MAGIE :
              $message = 'Vous n\'avez pas de magie !';
              break;

            case Personnage::PERSO_ENDORMI :
              $message = 'Vous êtes endormi, vous ne pouvez pas lancer
de sort !';
              break;
          }
        }
      }
    }
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>TP : Mini jeu de combat - Version 2</title>

    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?php echo $manager->count(); ?
></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Type : <?php echo ucfirst($perso->type()); ?><br />
        Nom : <?php echo htmlspecialchars($perso->nom()); ?><br />
        Dégâts : <?php echo $perso->degats(); ?><br />
<?php
// On affiche l'atout du personnage suivant son type.
switch ($perso->type())
{
  case 'magicien' :
    echo 'Magie : ';
    break;
```

```
      case 'guerrier' :
        echo 'Protection : ';
        break;
    }

    echo $perso->atout();
    ?>
          </p>
        </fieldset>

        <fieldset>
          <legend>Qui attaquer ?</legend>
          <p>
    <?php
    // On récupère tous les personnages par ordre alphabétique, dont le
    nom est différent de celui de notre personnage (on va pas se
    frapper nous-même :p).
    $retourPersos = $manager->getList($perso->nom());

    if (empty($retourPersos))
    {
      echo 'Personne à frapper !';
    }

    else
    {
      if ($perso->estEndormi())
      {
        echo 'Un magicien vous a endormi ! Vous allez vous réveiller
    dans ', $perso->reveil(), '.';
      }

      else
      {
        foreach ($retourPersos as $unPerso)
        {
          echo '<a href="?frapper=', $unPerso->id(), '">',
    htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso-
    >degats(), ' | type : ', $unPerso->type(), ')';

          // On ajoute un lien pour lancer un sort si le personnage est
    un magicien.
          if ($perso->type() == 'magicien')
          {
            echo ' | <a href="?ensorceler=', $unPerso->id(), '">Lancer
    un sort</a>';
          }

          echo '<br />';
        }
      }
    }
    ?>
          </p>
        </fieldset>
    <?php
    }
    else
    {
    ?>
        <form action="" method="post">
          <p>
            Nom : <input type="text" name="nom" maxlength="50" /> <input
    type="submit" value="Utiliser ce personnage" name="utiliser" /><br
    />
            Type :
            <select name="type">
              <option value="magicien">Magicien</option>
              <option value="guerrier">Guerrier</option>
```

```
            </select>
            <input type="submit" value="Créer ce personnage"
name="creer" />
        </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans
une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

Alors, vous commencez à comprendre toute la puissance de la POO et de l'héritage ? Avec une telle structure, vous pourrez à tout moment décider de créer un nouveau personnage très simplement ! Il vous suffit de créer une nouvelle classe, d'ajouter le type du personnage à l'énumération « type » en BDD et de modifier un petit peu le fichier **index.php** et votre personnage voit le jour !

Améliorations possibles

Comme au précédent TP, beaucoup d'améliorations sont possibles, à commencer par celles déjà exposées dans le chapitre dudit TP :

- Un système de **niveau**. Vous pourriez très bien assigner à chaque personnage un niveau de 1 à 100. Le personnage bénéficierait aussi d'une expérience allant de 0 à 100. Lorsque l'expérience atteint 100, le personnage passe au niveau suivant.

  Indice : le **niveau** et l'**expérience** deviendraient des caractéristiques du personnage, donc... Pas besoin de vous le dire, je suis sûr que vous savez ce que ça signifie !

- Un système de **force**. La force du personnage pourrait augmenter en fonction de son niveau, les dégâts infligés à la victime seront donc plus importants.

  Indice : de même, la **force** du personnage serait aussi une caractéristique du personnage.

- Un système de **limitation**. En effet, un personnage peut en frapper autant qu'il veut dans un laps de temps indéfini. Pourquoi ne pas le limiter à 3 coups par jour ?

  Indice : il faudrait que vous stockiez le **nombre de coups** portés par le personnage, ainsi que la **date du dernier coup porté**. Cela ferait donc deux nouveaux champs en BDD, et deux nouvelles caractéristiques pour le personnage !

- Un système de **retrait de dégâts**. Chaque jour, si l'utilisateur se connecte, il pourrait voir ses dégâts se soustraire de 10 par exemple.

  Indice : il faudrait stocker la date de dernière connexion. À chaque connexion, vous regarderiez cette date. Si elle est inférieure à 24h, alors vous ne feriez rien. Sinon, vous retireriez 10 de dégâts au personnage puis mettriez à jour cette date de dernière connexion.

Pour reprendre cet esprit, vous pouvez vous entraîner à créer d'autres personnages, comme une **brute** par exemple. Son atout dépendrait aussi de ses dégâts et viendrait augmenter sa force lors d'une attaque.

La seule différence avec le premier TP est l'apparition de nouveaux personnages, les seules améliorations différentes possibles sont donc justement la création de nouveaux personnages. Ainsi je vous encourage à imaginer tout un tas de différents personnages tous aussi farfelus les uns que les autres : cela vous fera grandement progresser et cela vous aidera à vous familiariser avec l'héritage !

Les méthodes magiques

Nous allons terminer cette partie par un chapitre assez simple. Dans ce chapitre, nous allons nous pencher sur une possibilité que nous offre le langage : il s'agit des méthodes magiques. Ce sont de petites bricoles bien pratiques dans certains cas.

Le principe

Vous devez sans doute vous poser une grande question à la vue du titre du chapitre : mais qu'est-ce que c'est qu'une méthode magique ?

Une méthode magique est une méthode qui, si elle est présente dans votre classe, sera appelée lors de tel ou tel évènement. Si la méthode n'existe pas et que l'évènement est exécuté, aucun effet « spécial » ne sera ajouté, l'évènement s'exécutera normalement. Le but des méthodes magiques est d'intercepter un évènement, dire de faire ceci ou cela et retourner une valeur utile pour l'évènement si besoin il y a.

Bonne nouvelle : vous connaissez déjà une méthode magique !

Si si, cherchez bien au fond de votre tête… Et oui, la méthode `__construct` est magique ! Comme nous l'avons vu plus haut, chaque méthode magique s'exécute au moment où un évènement est lancé. L'évènement qui appelle la méthode `__construct` est **la création de l'objet**.

Dans le même genre que `__construct` on peut citer `__destruct` qui, elle, sera appelée lors de la **destruction de l'objet**. Assez intuitif, mais voici un exemple au cas où :

Code : PHP

```php
<?php
class MaClasse
{
  public function __construct()
  {
    echo 'Construction de MaClasse';
  }

  public function __destruct()
  {
    echo 'Destruction de MaClasse';
  }
}

$obj = new MaClasse;
?>
```

Ainsi, vous verrez les deux messages écrits ci-dessus à la suite.

Surcharger les attributs et méthodes

Parlons maintenant des méthodes magiques liées à la surcharge des attributs et méthodes.

Euh, deux secondes là… C'est quoi la « surcharge des attributs et méthodes » ??

Vous avez raison, il serait d'abord préférable d'expliquer ceci. La surcharge d'attributs ou méthodes consiste à prévoir le cas où l'on appelle un attribut ou méthode qui n'existe pas ou du moins, auquel on n'a pas accès (par exemple, si un attribut ou une méthode est privé(e)). Dans ce cas-là, on a… voyons… 6 méthodes magiques à notre disposition !

« __set » et « __get »

Commençons par étudier ces deux méthodes magiques. Leur principe est le même, leur fonctionnement est à peu près semblable, c'est juste l'événement qui change.

Commençons par `__set`. Cette méthode est appelée lorsque l'on essaye d'assigner une valeur à un attribut auquel on n'a pas accès ou qui n'existe pas. Cette méthode prend deux paramètres : le premier est le nom de l'attribut auquel on a tenté d'assigner

une valeur, le second paramètre est la valeur que l'on a tenté d'assigner à l'attribut. Cette méthode ne retourne rien. Vous pouvez simplement faire ce que bon vous semble.

Exemple :

Code : PHP

```php
<?php
class MaClasse
{
  private $unAttributPrive;

  public function __set($nom, $valeur)
  {
    echo 'Ah, on a tenté d\'assigner à l\'attribut <strong>', $nom, '</strong> la valeur <strong>', $valeur, '</strong> mais c\'est pas possible !<br />';
  }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';
?>
```

À la sortie s'affichera :

Résultat affiché par le script

Tenez, petit exercice, stockez dans un tableau tous les attributs (avec leurs valeurs) que nous avons essayé de modifier ou créer.

Solution :

Code : PHP

```php
<?php
class MaClasse
{
  private $attributs = array();
  private $unAttributPrive;

  public function __set($nom, $valeur)
  {
    $this->attributs[$nom] = $valeur;
  }

  public function afficherAttributs()
```

```
    {
      echo '<pre>', print_r($this->attributs, true), '</pre>';
    }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

$obj->afficherAttributs();
?>
```

Pas compliqué à faire, mais cela permet de pratiquer un peu.

Parlons maintenant de `__get`. Cette méthode est appelée lorsque l'on essaye d'accéder à un attribut qui n'existe pas ou auquel on n'a pas accès. Elle prend un paramètre : le nom de l'attribut auquel on a essayé d'accéder. Cette méthode peut retourner ce qu'elle veut (ce sera, en quelque sorte, la valeur de l'attribut inaccessible).

Exemple :

Code : PHP

```
<?php
class MaClasse
{
  private $unAttributPrive;

  public function __get($nom)
  {
    return 'Impossible d\'accéder à l\'attribut <strong>' . $nom .
'</strong>, désolé !<br />';
  }
}

$obj = new MaClasse;

echo $obj->attribut;
echo $obj->unAttributPrive;
?>
```

Ce qui va afficher :

Résultat affiché par le script

Encore un exercice.
Combinez l'exercice précédent en vérifiant si l'attribut auquel on a tenté d'accéder est contenu dans le tableau de stockage

d'attributs. Si tel est le cas, on l'affiche, sinon, on ne fait rien.

Solution :

Code : PHP

```php
<?php
class MaClasse
{
  private $attributs = array();
  private $unAttributPrive;

  public function __get($nom)
  {
    if (isset($this->attributs[$nom]))
    {
      return $this->attributs[$nom];
    }
  }

  public function __set($nom, $valeur)
  {
    $this->attributs[$nom] = $valeur;
  }

  public function afficherAttributs()
  {
    echo '<pre>', print_r($this->attributs, true), '</pre>';
  }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

echo $obj->attribut;
echo $obj->autreAtribut;
?>
```

Étant donné que tous vos attributs doivent être privés, vous pouvez facilement les mettre en «lecture seule » grâce à `__get`. L'utilisateur aura accès aux attributs, mais ne pourra pas les modifier.

« __isset » et « __unset »

La première méthode `__isset` est appelée lorsque l'on appelle la fonction `isset` sur un attribut qui n'existe pas ou auquel on n'a pas accès. Étant donné que la fonction initiale `isset` renvoie `true` ou `false`, la méthode magique `__isset` doit renvoyer un booléen. Cette méthode prend un paramètre : le nom de l'attribut que l'on a envoyé à la fonction `isset`. Vous pouvez par exemple utiliser la classe précédente en implémentant la méthode `__isset`, ce qui peut nous donner :

Code : PHP

```php
<?php
class MaClasse
{
  private $attributs = array();
  private $unAttributPrive;

  public function __set($nom, $valeur)
  {
    $this->attributs[$nom] = $valeur;
  }
```

```php
    public function __get($nom)
    {
      if (isset($this->attributs[$nom]))
      {
        return $this->attributs[$nom];
      }
    }

    public function __isset($nom)
    {
      return isset($this->attributs[$nom]);
    }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

if (isset($obj->attribut))
{
  echo 'L\'attribut <strong>attribut</strong> existe !<br />';
}
else
{
  echo 'L\'attribut <strong>attribut</strong> n\'existe pas !<br
/>';
}

if (isset($obj->unAutreAttribut))
{
  echo 'L\'attribut <strong>unAutreAttribut</strong> existe !';
}
else
{
  echo 'L\'attribut <strong>unAutreAttribut</strong> n\'existe pas
!';
}
?>
```

Ce qui affichera :

Résultat affiché par le script

Pour `__unset`, le principe est le même. Cette méthode est appelée lorsque l'on tente d'appeler la fonction `unset` sur un attribut inexistant ou auquel on n'a pas accès. On peut facilement implémenter `__unset` à la classe précédente de manière à supprimer l'entrée correspondante dans notre tableau `$attributs`. Cette méthode ne doit rien retourner.

Code : PHP

```php
<?php
class MaClasse
{
  private $attributs = array();
  private $unAttributPrive;

  public function __set($nom, $valeur)
  {
    $this->attributs[$nom] = $valeur;
  }

  public function __get($nom)
  {
    if (isset($this->attributs[$nom]))
    {
      return $this->attributs[$nom];
    }
  }

  public function __isset($nom)
  {
    return isset($this->attributs[$nom]);
  }

  public function __unset($nom)
  {
    if (isset($this->attributs[$nom]))
    {
      unset($this->attributs[$nom]);
    }
  }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

if (isset($obj->attribut))
{
  echo 'L\'attribut <strong>attribut</strong> existe !<br />';
}
else
{
  echo 'L\'attribut <strong>attribut</strong> n\'existe pas !<br
/>';
}

unset($obj->attribut);

if (isset($obj->attribut))
{
  echo 'L\'attribut <strong>attribut</strong> existe !<br />';
}
else
{
  echo 'L\'attribut <strong>attribut</strong> n\'existe pas !<br
/>';
}

if (isset($obj->unAutreAttribut))
{
  echo 'L\'attribut <strong>unAutreAttribut</strong> existe !';
}
else
{
  echo 'L\'attribut <strong>unAutreAttribut</strong> n\'existe pas
!';
}
?>
```

Ce qui donnera :

Résultat affiché par le script

Finissons par « __call » et « __callStatic »

Bien que la méthode magique `__call` soit disponible sous PHP 5.1, la méthode `__callStatic` n'est disponible que sous PHP 5.3 !

Bien. Laissons de côté les attributs pour le moment et parler cette fois-ci des méthodes que l'on appelle alors qu'on n'y a pas accès (soit elle n'existe pas, soit elle est privée). La méthode `__call` sera appelée lorsque l'on essayera d'appeler une telle méthode. Elle prend deux arguments : le premier est le nom de la méthode que l'on a essayé d'appeler et le second est la liste des arguments qui lui ont été passés (sous forme de tableau).

Exemple :

Code : PHP

```php
<?php
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode($arguments, '</strong>, <strong>'), '</strong>';
  }
}

$obj = new MaClasse;

$obj->methode(123, 'test');
?>
```

Résultat :

Résultat affiché par le script

Et si on essaye d'appeler une méthode qui n'existe pas statiquement ? Et bien, erreur fatale ! Sauf si vous utilisez `__callStatic`. Cette méthode est appelée lorsque vous appelez une méthode dans un contexte statique alors qu'elle n'existe pas. La méthode magique `__callStatic` doit obligatoirement être `static` !

Code : PHP

```php
<?php
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode ($arguments, '</strong>, <strong>'), '</strong><br />';
  }

  public static function __callStatic($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée dans un contexte statique alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode ($arguments, '</strong>, <strong>'), '</strong><br />';
  }
}

$obj = new MaClasse;

$obj->methode(123, 'test');

MaClasse::methodeStatique(456, 'autre test');
?>
```

Résultat :

Résultat affiché par le script

Linéariser ses objets

Voici un point important de ce chapitre sur lequel je voudrais m'arrêter un petit instant : la linéarisation des objets. Pour suivre cette partie, je vous recommande chaudement ce tutoriel qui vous expliquera de manière générale ce qu'est la linéarisation et vous fera pratiquer sur des exemples divers. Lisez le jusqu'à la partie **Encore plus fort !**, et je vais mieux vous expliquer à partir de là !

Posons le problème

Vous avez un système de sessions sur votre site avec une classe `Connexion`. Cette classe, comme son nom l'indique, aura pour rôle d'établir une connexion à la BDD. Vous aimeriez bien stocker l'objet créé dans une variable `$_SESSION` mais vous ne savez pas comment faire.

Ben si ! On fait `$_SESSION['connexion'] = $objetConnexion` et puis voilà !

Oui, ça fonctionne, mais savez-vous vraiment ce qui se passe quand vous effectuez une telle opération ? Ou plutôt, ce qui se passe **à la fin du script** ? En fait, à la fin du script, le tableau de session est **linéarisé** automatiquement. **Linéariser** signifie que l'on *transforme* une variable en chaîne de caractères selon un format bien précis. Cette chaîne de caractères pourra, quand on le souhaitera, être transformée dans l'autre sens (c'est-à-dire qu'on va restituer son état d'origine). Pour bien comprendre ce principe, on va linéariser nous-mêmes notre objet. Voici ce que nous allons faire :

- Création de l'objet (`$objetConnexion = new Connexion;`);
- transformation de l'objet en chaîne de caractères (`$_SESSION['connexion'] = serialize($objetConnexion);`);
- changement de page ;
- transformation de la chaîne de caractères en objet (`$objetConnexion = unserialize($_SESSION['connexion']);`).

Des explications s'imposent.

Les nouveautés rencontrées ici sont l'apparition de deux nouvelles fonctions : `serialize` et `unserialize`.

La première fonction, `serialize`, retourne l'objet passé en paramètre sous forme de chaîne de caractères. Vous vous demandez sans doutes comment on peut transformer un objet en chaîne de caractères : la réponse est toute simple. Quand on y réfléchit, un objet c'est quoi ? C'est un ensemble d'attributs, tout simplement. Les méthodes ne sont pas stockées dans l'objet, c'est la classe qui s'en occupe. Notre chaîne de caractères contiendra donc juste quelque chose comme : « Objet `MaClasse` contenant les attributs `unAttribut` qui vaut "Hello world !", `autreAttribut` qui vaut "Vive la linéarisation", `dernierAttribut` qui vaut "Et un dernier pour la route !" ». Ainsi, vous pourrez conserver votre objet dans une variable sous forme de chaîne de caractères. Si vous affichez cette chaîne par un `echo` par exemple, vous n'arriverez sans doute pas à déchiffrer l'objet, c'est normal, ce n'est pas aussi simple que la chaîne que j'ai montrée à titre d'exemple . Cette fonction est **automatiquement** appelée sur l'array `$_SESSION` à la fin du script, notre objet est donc automatiquement linéarisé à la fin du script. C'est uniquement dans un but didactique que nous linéarisons manuellement.

La seconde fonction, `unserialize`, retourne la chaîne de caractères passée en paramètre sous forme d'objet. En gros, cette fonction lit la chaîne de caractères, crée une instance de la classe correspondante et assigne à chaque attribut la valeur qu'ils

avaient. Ainsi, vous pourrez utiliser l'objet retourné (appel de méthodes, attributs et diverses opérations) comme avant. Cette fonction est automatiquement appelée dès le début du script pour restaurer le tableau de sessions précédemment enregistré dans le fichier. Sachez toutefois que si vous avez linéarisé un objet manuellement, il ne sera **jamais** restauré automatiquement.

Et quel est le rapport avec tes méthodes magiques ?

En fait, les fonctions citées ci-dessus (`serialize` et `unserialize`) ne se contentent pas de transformer le paramètre qu'on leur passe en autre chose : elles vérifient si, dans l'objet passé en paramètre (pour `serialize`), il y a une méthode `__sleep`, auquel cas elle est exécutée. Si c'est `unserialize` qui est appelée, la fonction vérifie si l'objet obtenu comporte une méthode `__wakeup`, auquel cas elle est appelée.

« serialize » et « __sleep »

La méthode magique `__sleep` est utilisée pour nettoyer l'objet ou pour sauver des attributs. Si la méthode magique `__sleep` n'existe pas, tous les attributs seront sauvés. Cette méthode doit renvoyer un tableau avec les noms des attributs à sauver. Par exemple, si vous voulez sauver `$serveur` et `$login`, la fonction devra retourner `array('serveur', 'login');`.

Voici ce que pourrait donner notre classe `Connexion` :

Code : PHP

```php
<?php
class Connexion
{
  protected $pdo, $serveur, $utilisateur, $motDePasse, $dataBase;

  public function __construct($serveur, $utilisateur, $motDePasse,
$dataBase)
  {
    $this->serveur = $serveur;
    $this->utilisateur = $utilisateur;
    $this->motDePasse = $motDePasse;
    $this->dataBase = $dataBase;

    $this->connexionBDD();
  }

  protected function connexionBDD()
  {
    $this->pdo = new PDO('mysql:host='.$this-
>serveur.';dbname='.$this->dataBase, $this->utilisateur, $this-
>motDePasse);
  }

  public function __sleep()
  {
    // Ici sont à placer des instructions à exécuter juste avant la
linéarisation.
    // On retourne ensuite la liste des attributs qu'on veut
sauver.
    return array('serveur', 'utilisateur', 'motDePasse',
'dataBase');
  }
}
?>
```

Ainsi, vous pourrez faire ceci :

Code : PHP

```php
<?php
```

```
$connexion = new Connexion('localhost', 'root', '', 'tests');

$_SESSION['connexion'] = serialize($connexion);
?>
```

« unserialize » et « __wakeup »

Maintenant, nous allons simplement implémenter la fonction `__wakeup`. Qu'allons-nous mettre dedans ?

Rien de compliqué… Nous allons juste appeler la méthode `connexionBDD` qui se chargera de nous connecter à notre base de données puisque les identifiants, serveur et nom de la base ont été sauvegardés et ainsi restaurés à l'appel de la fonction `unserialize` !

Code : PHP

```
<?php
class Connexion
{
  protected $pdo, $serveur, $utilisateur, $motDePasse, $dataBase;

  public function __construct($serveur, $utilisateur, $motDePasse,
$dataBase)
  {
    $this->serveur = $serveur;
    $this->utilisateur = $utilisateur;
    $this->motDePasse = $motDePasse;
    $this->dataBase = $dataBase;

    $this->connexionBDD();
  }

  protected function connexionBDD()
  {
    $this->pdo = new PDO('mysql:host='.$this-
>serveur.';dbname='.$this->dataBase, $this->utilisateur, $this-
>motDePasse);
  }

  public function __sleep()
  {
    return array('serveur', 'utilisateur', 'motDePasse',
'dataBase');
  }

  public function __wakeup()
  {
    $this->connexionBDD();
  }
}
?>
```

Pratique, hein ? 

Maintenant que vous savez ce qui se passe quand vous enregistrez un objet dans une entrée de session, je vous autorise à ne plus appeler `serialize` et `unserialize`. 

Ainsi, ce code fonctionne parfaitement :

Code : PHP

```
<?php
```

```php
session_start();

if (!isset($_SESSION['connexion']))
{
  $connexion = new Connexion('localhost', 'root', '', 'tests');
  $_SESSION['connexion'] = $connexion;

  echo 'Actualisez la page !';
}

else
{
  echo '<pre>';
  var_dump($_SESSION['connexion']); // On affiche les infos concernant notre objet.
  echo '</pre>';
}
?>
```

Vous voyez donc, en testant ce code, que notre objet a bel et bien été sauvegardé comme il fallait, et que tous les attributs ont été sauvés. Bref, c'est magique. 

Étant donné que notre objet est restauré automatiquement lors de l'appel de `session_start()`, la classe correspondante doit être déclarée **avant**, sinon l'objet désérialisé sera une instance de `__PHP_Incomplete_Class_Name`, classe qui ne contient aucune méthode (cela produira donc un objet inutile). Si vous avez un autoload qui chargera la classe automatiquement, il sera appelé.

Autres méthodes magiques

Voici les dernières méthodes magiques que vous n'avez pas vues. Je parlerai ici de `__toString`, `__set_state` et `__invoke`.

« __toString »

La méthode magique `__toString` est appelée lorsque l'objet est amené à être converti en chaîne de caractères. Cette méthode doit retourner la chaîne de caractères souhaitée.

Exemple :

Code : PHP

```php
<?php
class MaClasse
{
  protected $texte;

  public function __construct($texte)
  {
    $this->texte = $texte;
  }

  public function __toString()
  {
    return $this->texte;
  }
}

$obj = new MaClasse('Hello world !');

// Solution 1 : le cast

$texte = (string) $obj;
var_dump($texte); // Affiche : string(13) "Hello world !".

// Solution 2 : directement dans un echo
```

```
echo $obj; // Affiche : Hello world !
?>
```

Pas mal, hein ? 

« __set_state »

La méthode magique `__set_state` est appelée lorsque vous appelez la fonction `var_export` en passant votre objet à exporter en paramètre. Cette fonction `var_export` a pour rôle d'exporter la variable passée en paramètre sous forme de code PHP (chaîne de caractères). Si vous ne spécifiez pas de méthode `__set_state` dans votre classe, une erreur fatale sera levée.

Notre méthode `__set_state` prend un paramètre, la liste des attributs ainsi que leur valeur dans un tableau associatif (`array('attribut' => 'valeur')`). Notre méthode magique devra retourner l'objet à exporter. Il faudra donc créer un nouvel objet et lui assigner les valeurs qu'on souhaite, puis le retourner.

Ne jamais retourner `$this`, car cette variable n'existera pas dans cette méthode ! `var_export` reportera donc une valeur nulle.

Puisque la fonction `var_export` retourne du code PHP valide, on peut utiliser la fonction `eval` qui exécute du code PHP sous forme de chaîne de caractères qu'on lui passe en paramètre.

Par exemple, pour retourner un objet en sauvant ses attributs, on pourrait faire :

Code : PHP

```
<?php
class Export
{
  protected $chaine1, $chaine2;

  public function __construct($param1, $param2)
  {
    $this->chaine1 = $param1;
    $this->chaine2 = $param2;
  }

  public function __set_state($valeurs) // Liste des attributs de
l'objet en paramètre.
  {
    $obj = new Export($valeurs['chaine1'], $valeurs['chaine2']); //
On crée un objet avec les attributs de l'objet que l'on veut
exporter.
    return $obj; // on retourne l'objet créé.
  }
}

$obj1 = new Export('Hello ', 'world !');

eval('$obj2 = ' . var_export ($obj1, true) . ';'); // On crée un
autre objet, celui-ci ayant les mêmes attributs que l'objet
précédent.

echo '<pre>', print_r ($obj2, true), '</pre>';
?>
```

Le code affichera donc :

Résultat affiché par le script

« __invoke »

Disponible depuis PHP 5.3

Que diriez-vous de pouvoir utiliser l'objet comme fonction ? Vous ne voyez pas ce que je veux dire ? Je comprends.

Voici un code qui illustrera bien le tout :

Code : PHP

```php
<?php
$obj = new MaClasse;
$obj('Petit test'); // Utilisation de l'objet comme fonction.
?>
```

Essayez ce code et… BAM ! Une erreur fatale (c'est bizarre). Plus sérieusement, pour résoudre ce problème, nous allons devoir utiliser la méthode magique `__invoke`. Elle est appelée dès qu'on essaye d'utiliser l'objet comme fonction (comme on vient de faire). Cette méthode comprend autant de paramètres que d'arguments passés à la fonction.

Exemple :

Code : PHP

```php
<?php
class MaClasse
{
  public function __invoke($argument)
  {
    echo $argument;
  }
}

$obj = new MaClasse;

$obj(5); // Affiche « 5 ».
?>
```

Ce tutoriel portant sur les méthodes magiques s'arrête ici. Je parlerai de la méthode `__clone` lors du clonage d'objets en deuxième partie.

En résumé

- Les méthodes magiques sont des méthodes qui sont appelées automatiquement lorsqu'un certain évènement est déclenché.
- Toutes les méthodes magiques commencent par deux *underscores*, évitez donc d'appeler vos méthodes suivant ce même modèle.
- Les méthodes magiques dont vous vous servirez le plus souvent sont `__construct`, `__set`, `__get` et `__call`. Les autres sont plus « gadget » et vous les rencontrerez moins souvent.

Partie 2 : [Théorie] Techniques avancées

Les objets en profondeur

Je suis sûr qu'actuellement, vous pensez que lorsqu'on fait un `$objet = new MaClasse;`, la variable `$objet` contient l'objet que l'on vient de créer. Personne ne peut vous en vouloir puisque personne ne vous a dit que c'était faux. Et bien je vous le dis maintenant : comme nous le verrons dans ce chapitre, une telle variable ne contient pas l'objet à proprement parler ! Ceci explique ainsi quelques comportements bizarres que peut avoir PHP avec les objets. Nous allons ainsi parler de la dernière méthode magique que je vous avais volontairement cachée.

Une fois tout ceci expliqué, nous jouerons un peu avec nos objets en les parcourant, à peu près de la même façon qu'avec des tableaux.

Comme vous le verrez, les objets réservent bien des surprises !

Un objet, un identifiant

Je vais commencer cette partie en vous faisant une révélation : quand vous instanciez une classe, la variable stockant l'objet ne stocke en fait pas l'objet lui-même, mais un identifiant qui représente cet objet. C'est-à-dire qu'en faisant `$objet = new Classe;`, "`$objet` ne contient pas l'objet lui-même, mais son identifiant unique. C'est un peu comme quand vous enregistrez des informations dans une BDD : la plupart du temps, vous avez un champ "id" unique qui représente l'entrée. Quand vous faites une requête SQL, vous sélectionnez l'élément en fonction de son id. Et bien là, c'est pareil : quand vous accédez à un attribut ou à une méthode de l'objet, PHP regarde l'identifiant contenu dans la variable, va chercher l'objet correspondant et effectue le traitement nécessaire. Il est très important que vous compreniez cette idée, sinon vous allez être complètement perdus pour la suite du chapitre.

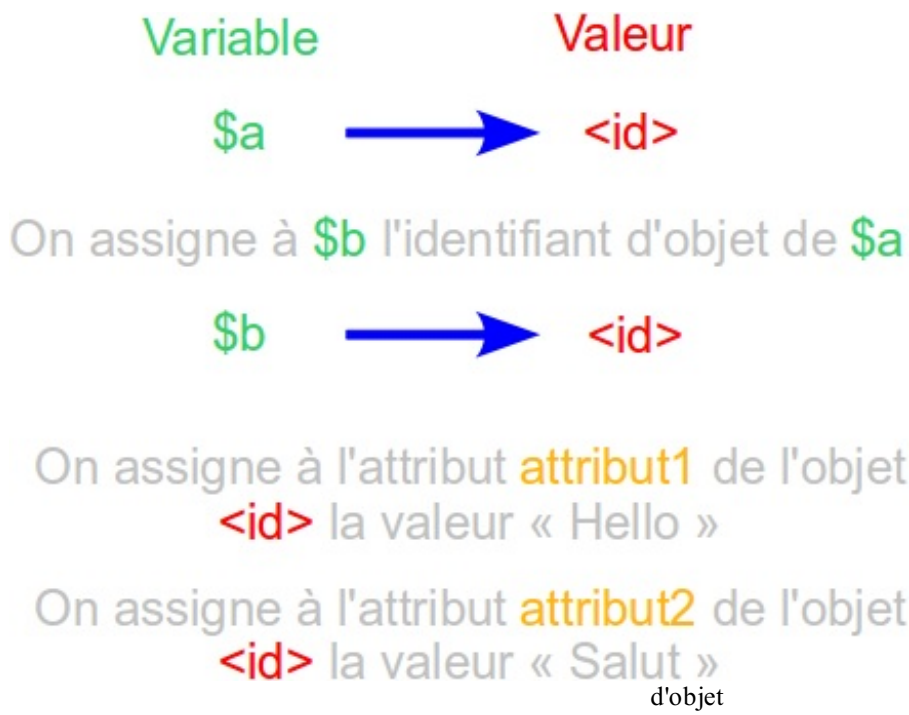
Nous avons donc vu que la variable `$objet` contenait l'identifiant de l'objet qu'elle a instancié. Vérifions cela :

Code : PHP

```php
<?php
class MaClasse
{
  public $attribut1;
  public $attribut2;
}

$a = new MaClasse;

$b = $a; // On assigne à $b l'identifiant de $a, donc $a et $b représentent le même objet.

$a->attribut1 = 'Hello';
echo $b->attribut1; // Affiche Hello.

$b->attribut2 = 'Salut';
echo $a->attribut2; // Affiche Salut.
?>
```

Je commente plus en détail la ligne 10 pour ceux qui sont un peu perdus. Nous avons dit plus haut que `$a` ne contenait pas l'objet lui-même mais son identifiant (un identifiant d'objet). `$a` contient donc l'identifiant représentant l'objet créé. Ensuite, on assigne à `$b` la valeur de `$a`. Donc qu'est-ce que `$b` vaut maintenant ? Et bien la même chose que `$a`, à savoir **l'identifiant qui représente l'objet** ! `$a` et "`$b` font donc référence **à la même instance**.

Schématiquement, on peut représenter le code ci-dessus comme ceci :



Exemple de conséquences des identifiants d'objet

Comme vous le voyez sur l'image, en réalité, il n'y a qu'un seul objet, qu'un seul identifiant, mais deux variables contenant exactement le même identifiant d'objet. Tout ceci peut sembler abstrait, donc allez à votre rythme pour bien comprendre.

Maintenant que l'on sait que ces variables ne contiennent pas d'objet mais un **identifiant d'objet**, vous êtes censés savoir que lorsqu'un objet est passé en paramètre à une fonction ou renvoyé par une autre, on ne passe pas une copie de l'objet mais une copie de son identifiant ! Ainsi, vous n'êtes pas obligé de passer l'objet en référence, car vous passerez une référence de l'identifiant de l'objet. Inutile, donc.

Cependant un problème se pose. Comment faire pour copier un objet ? Comment faire pour pouvoir copier tous ses attributs et valeurs dans un nouvel objet **unique** ? On a vu qu'on ne pouvait pas faire un simple `$objet1 = $objet2` pour arriver à cela. Comme vous vous en doutez peut-être, c'est là qu'intervient le clonage d'objet.

Pour cloner un objet, c'est assez simple. Il faut utiliser le mot-clé *clone* juste avant l'objet à copier. Exemple :

Code : PHP

```php
<?php
$copie = clone $origine; // On copie le contenu de l'objet $origine dans l'objet $copie.
?>
```

C'est aussi simple que cela. Ainsi les deux objets contiennent des identifiants différents : par conséquent, si on veut modifier l'un d'eux, on peut le faire sans qu'aucune propriété de l'autre ne soit modifiée.

Il n'était pas question d'une méthode magique ?

Si si, j'y viens.

Lorsque vous clonez un objet, la méthode `__clone` de celui-ci sera appelée (du moins, si vous l'avez définie). Vous ne pouvez pas appeler cette méthode directement. C'est la méthode `__clone` de l'objet à cloner qui est appelée, pas la méthode `__clone` du nouvel objet créé.

Vous pouvez utiliser cette méthode pour modifier certains attributs pour l'ancien objet, ou alors incrémenter un compteur

d'instances par exemple.

Code : PHP

```php
<?php
class MaClasse
{
  private static $instances = 0;

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

  public function __clone()
  {
    self::$instances++;
  }

  public static function getInstances()
  {
    return self::$instances;
  }
}

$a = new MaClasse;
$b = clone $a;

echo 'Nombre d\'instances de MaClasse : ', MaClasse::getInstances();
?>
```

Ce qui affichera :

Résultat affiché par le script

Comparons nos objets

Nous allons maintenant voir comment comparer deux objets. C'est très simple, il suffit de faire comme vous avez toujours fait en comparant des chaînes de caractères ou des nombres. Voici un exemple :

Code : PHP

```php
<?php
if ($objet1 == $objet2)
{
  echo '$objet1 et $objet2 sont identiques !';
}
else
{
  echo '$objet1 et $objet2 sont différents !';
}
?>
```

Cette partie ne vous expliquera donc pas comment comparer des objets mais la démarche que PHP exécute pour les comparer et les effets que ces comparaisons peuvent produire.

Reprenons le code ci-dessus. Pour que la condition renvoie *true*, il faut que `$objet1` et `$objet2` aient les mêmes attributs et les mêmes valeurs, mais également que les deux objets soient des instances de la même classe. C'est-à-dire que même s'ils ont les mêmes attributs et valeurs mais que l'un est une instance de la classe A et l'autre une instance de la classe B, la condition renverra *false*.

Exemple :

Code : PHP

```php
<?php
class A
{
  public $attribut1;
  public $attribut2;
}

class B
{
  public $attribut1;
  public $attribut2;
}

$a = new A;
$a->attribut1 = 'Hello';
$a->attribut2 = 'Salut';

$b = new B;
$b->attribut1 = 'Hello';
$b->attribut2 = 'Salut';

$c = new A;
$c->attribut1 = 'Hello';
$c->attribut2 = 'Salut';

if ($a == $b)
{
  echo '$a == $b';
}
else
{
  echo '$a != $b';
}

echo '<br />';

if ($a == $c)
{
  echo '$a == $c';
}
else
{
  echo '$a != $c';
}
?>
```

Si vous avez bien suivi, vous savez ce qui va s'afficher, à savoir :

Résultat affiché par le script

Comme on peut le voir, `$a` et `$b` ont beau avoir les mêmes attributs et les mêmes valeurs, ils ne sont pas identiques car ils ne sont pas des instances de la même classe. Par contre, `$a` et `$c` sont bien identiques.

Parlons maintenant de l'opérateur === qui permet de vérifier que deux objets sont **strictement** identiques. Vous n'avez jamais entendu parler de cet opérateur ? Allez lire ce tutoriel !

Cet opérateur vérifiera si les deux objets font référence vers la même instance. Il vérifiera donc que les deux identifiants d'objets comparés sont les mêmes. Allez relire la première partie de ce chapitre si vous êtes un peu perdu.

Faisons quelques tests pour être sûr que vous avez bien compris :

Code : PHP

```php
<?php
class A
{
  public $attribut1;
  public $attribut2;
}

$a = new A;
$a->attribut1 = 'Hello';
$a->attribut2 = 'Salut';

$b = new A;
$b->attribut1 = 'Hello';
$b->attribut2 = 'Salut';

$c = $a;

if ($a === $b)
{
  echo '$a === $b';
}
else
{
  echo '$a !== $b';
}

echo '<br />';

if ($a === $c)
{
  echo '$a === $c';
}
else
{
  echo '$a !== $c';
}
?>
```

Et à l'écran s'affichera :

Résultat affiché par le script

On voit donc que cette fois ci, la condition qui renvoyait *true* avec l'opérateur == renvoie maintenant *false*. `$a` et `$c` font référence à la même instance, la condition renvoie donc *true*.

Parcourons nos objets

Finissons en douceur en voyant comment parcourir nos objets et en quoi cela consiste.

Le fait de parcourir un objet consiste à lire tous les attributs **visibles** de l'objet. Qu'est-ce que cela veut dire ? Ceci veut tout simplement dire que vous ne pourrez pas lire les attributs privés ou protégés en dehors de la classe, mais l'inverse est tout à fait possible. Je ne vous apprends rien de nouveau me direz-vous, mais ce rappel me semblait important pour vous expliquer le parcours d'objets.

Qui dit "parcours" dit "boucle". Quelle boucle devrons-nous utiliser pour parcourir un objet ? Et bien la même boucle que pour parcourir un tableau... J'ai nommé *foreach* !

Son utilisation est d'une simplicité remarquable (du moins, si vous savez parcourir un tableau). Sa syntaxe est la même. Il y en a deux possibles :

- `foreach ($objet as $valeur)` : `$valeur` sera la valeur de l'attribut actuellement lu.
- `foreach ($objet as $attribut => $valeur)` : `$attribut` aura pour valeur le nom de l'attribut actuellement lu et `$valeur` sera sa valeur.

Vous ne devez sans doute pas être dépaysé, il n'y a presque rien de nouveau. Comme je vous l'ai dit, la boucle *foreach* parcourt les attributs visibles. Faisons quelques tests. Normalement, vous devez déjà anticiper le bon résultat (enfin, j'espère, mais si vous êtes tombé à côté de la plaque ce n'est pas un drame !).

Code : PHP

```php
<?php
class MaClasse
{
  public $attribut1 = 'Premier attribut public';
  public $attribut2 = 'Deuxième attribut public';

  protected $attributProtege1 = 'Premier attribut protégé';
  protected $attributProtege2 = 'Deuxième attribut protégé';

  private $attributPrive1 = 'Premier attribut privé';
  private $attributPrive2 = 'Deuxième attribut privé';

  function listeAttributs()
  {
    foreach ($this as $attribut => $valeur)
```

```
        {
          echo '<strong>', $attribut, '</strong> => ', $valeur, '<br
/>';
        }
      }
    }

    class Enfant extends MaClasse
    {
      function listeAttributs() // Redéclaration de la fonction pour
    que ce ne soit pas celle de la classe mère qui soit appelée.
      {
        foreach ($this as $attribut => $valeur)
        {
          echo '<strong>', $attribut, '</strong> => ', $valeur, '<br
/>';
        }
      }
    }

    $classe = new MaClasse;
    $enfant = new Enfant;

    echo '---- Liste les attributs depuis l\'intérieur de la classe
    principale ----<br />';
    $classe->listeAttributs();

    echo '<br />---- Liste les attributs depuis l\'intérieur de la
    classe enfant ----<br />';
    $enfant->listeAttributs();

    echo '<br />---- Liste les attributs depuis le script global ----<br
    />';

    foreach ($classe as $attribut => $valeur)
    {
      echo '<strong>', $attribut, '</strong> => ', $valeur, '<br />';
    }
    ?>
```

Ce qui affichera :

---- Liste les attributs depuis l'intérieur de la classe principale ----

- **attribut1** => Premier attribut public
- **attribut2** => Deuxième attribut public
- **attributProtege1** => Premier attribut protégé
- **attributProtege2** => Deuxième attribut protégé
- **attributPrive1** => Premier attribut privé
- **attributPrive2** => Deuxième attribut privé

---- Liste les attributs depuis l'intérieur de la classe enfant ----

- **attribut1** => Premier attribut public
- **attribut2** => Deuxième attribut public
- **attributProtege1** => Premier attribut protégé
- **attributProtege2** => Deuxième attribut protégé

---- Liste les attributs depuis le script global ----

- **attribut1** => Premier attribut public
- **attribut2** => Deuxième attribut public

J'ai volontairement terminé ce chapitre par le parcours d'objets. Pourquoi ? Car dans le prochain chapitre nous verrons comment modifier le comportement de l'objet quand il est parcouru grâce aux **interfaces** ! Celles-ci permettent de réaliser beaucoup de choses pratiques, mais je ne vous en dis pas plus. 

En résumé

- Une variable ne contient jamais d'objet à proprement parler, mais leurs identifiants.
- Pour dupliquer un objet, l'opérateur = n'a donc pas l'effet désiré : il faut **cloner** l'objet grâce à l'opérateur `clone`.
- Pour comparer deux objets, l'opérateur == vérifie que les deux objets sont issus de la même classe et que les valeurs de chaque attribut sont identiques, tandis que l'opérateur === vérifie que les deux identifiants d'objet sont les mêmes.
- Il est possible de parcourir un objet grâce la structure `foreach` : ceci aura pour effet de lister tous les attributs auxquels la structure a accès (par exemple, si la structure est située à l'extérieur de la classe, seuls les attributs publics seront listés).

Les interfaces

Si, dans l'une de vos méthodes, on vous passe un objet quelconque, il vous est impossible de savoir si vous pouvez invoquer telle ou telle méthode sur ce dernier pour la simple et bonne raison que vous n'êtes pas totalement sûr que ces méthodes existent. En effet, si vous ne connaissez pas la classe dont l'objet est l'instance, vous ne pouvez pas vérifier l'existence de ces méthodes.

En PHP, il existe un moyen d'imposer une **structure** à nos classes, c'est-à-dire d'obliger certaines classes à implémenter certaines méthodes. Pour y arriver, nous allons nous servir des **interfaces**. Une fois toute la théorie posée, nous allons nous servir d'interfaces pré-définies afin de jouer un petit peu avec nos objets.

Présentation et création d'interfaces

Le rôle d'une interface

Techniquement, une interface est une classe entièrement abstraite. Son rôle est de décrire un comportement à notre objet. Les interfaces ne doivent pas être confondues avec l'héritage : l'héritage représente un sous-ensemble (exemple : un magicien est un sous-ensemble d'un personnage). Ainsi, une voiture et un personnage n'ont aucune raison d'hériter d'une même classe. Par contre, une voiture et un personnage peuvent tous les deux se déplacer, donc une interface représentant ce point commun pourra être créée.

Créer une interface

Une interface se déclare avec le mot-clé `interface`, suivi du nom de l'interface, suivi d'une paire d'accolades. C'est entre ces accolades que vous listerez des méthodes. Par exemple, voici une interface pouvant représenter le point commun évoqué ci-dessus :

Code : PHP

```php
<?php
interface Movable
{
  public function move($dest);
}
?>
```

1. Toutes les méthodes présentes dans une interface doivent être publiques.
2. Une interface ne peut pas lister de méthodes abstraites ou finales.
3. Une interface ne peut pas avoir le même nom qu'une classe et vice-versa.

Implémenter une interface

Cette interface étant toute seule, elle est un peu inutile. Il va donc falloir *implémenter* l'interface à notre classe grâce au mot-clé `implements` ! La démarche à exécuter est comme quand on faisait hériter une classe d'une autre, à savoir :

Code : PHP

```php
<?php
class Personnage implements Movable
{

}
?>
```

Essayez ce code et... observez le résultat :

Résultat affiché par le script

Et oui, une erreur fatale est générée car notre classe `Personnage` n'a pas implémenté la méthode présente dans l'interface `Movable`. Pour que ce code ne génère aucune erreur, il faut qu'il y ait au minimum ce code :

Code : PHP

```php
<?php
class Personnage implements Movable
{
  public function move($dest)
  {

  }
}
?>
```

Et là... l'erreur a disparu !

Vous pouvez très bien, dans votre classe, définir une méthode comme étant abstraite ou finale.

Si vous héritez une classe et que vous implémentez une interface, alors vous devez d'abord spécifier la classe à hériter avec le mot-clé `extends` **puis** les interfaces à implémenter avec le mot-clé `implements`.

Une interface vous oblige à écrire toutes ses méthodes, mais vous pouvez en rajouter autant que vous voulez.

Vous pouvez très bien implémenter plus d'une interface par classe, **à condition que celles-ci n'aient aucune méthode portant le même nom** ! Exemple :

Code : PHP

```php
<?php
interface iA
{
  public function test1();
}

interface iB
{
  public function test2();
}

class A implements iA, iB
{
```

```
  // Pour ne générer aucune erreur, il va falloir écrire les
méthodes de iA et de iB.

  public function test1()
  {

  }

  public function test2()
  {

  }
}
?>
```

Les constantes d'interfaces

Les constantes d'interfaces fonctionnent exactement comme les constantes de classes. Elles ne peuvent être écrasées par des classes qui implémentent l'interface. Exemple :

Code : PHP

```
<?php
interface iInterface
{
  const MA_CONSTANTE = 'Hello !';
}

echo iInterface::MA_CONSTANTE; // Affiche Hello !

class MaClasse implements iInterface
{

}

echo MaClasse::MA_CONSTANTE; // Affiche Hello !
?>
```

Hériter ses interfaces

Comme pour les classes, vous pouvez hériter vos interfaces grâce à l'opérateur `extends`. Vous ne pouvez réécrire ni une méthode ni une constante qui a déjà été listée dans l'interface parente. Exemple :

Code : PHP

```
<?php
interface iA
{
  public function test1();
}

interface iB extends iA
{
  public function test1 ($param1, $param2); // Erreur fatale :
impossible de réécrire cette méthode.
}

interface iC extends iA
{
  public function test2();
}

class MaClasse implements iC
{
  // Pour ne générer aucune erreur, on doit écrire les méthodes de
```

```
    iC et aussi de iA.

      public function test1()
      {

      }

      public function test2()
      {

      }
    }
    ?>
```

Contrairement aux classes, les interfaces peuvent hériter de plusieurs interfaces à la fois. Il vous suffit de séparer leur nom par une virgule. Exemple :

Code : PHP

```
<?php
interface iA
{
  public function test1();
}

interface iB
{
  public function test2();
}

interface iC extends iA, iB
{
  public function test3();
}
?>
```

Dans cet exemple, si on imagine une classe implémentant iC, celle-ci devra implémenter les trois méthodes `test1`, `test2` et `test3`.

Interfaces prédéfinies

Nous allons maintenant aborder les interfaces prédéfinies. Grâce à certaines, nous allons pouvoir modifier le comportement de nos objets ou réaliser plusieurs choses pratiques. Il y a beaucoup d'interfaces prédéfinies, je ne vous les présenterai pas toutes, seulement quatre d'entre elles. Déjà, avec celles-ci, nous allons pouvoir réaliser de belles choses, et puis vous êtes libres de lire la documentation pour découvrir toutes les interfaces. Nous allons essayer ici de créer un « tableau-objet ».

L'interface `Iterator`

Commençons d'abord par l'interface `Iterator`. Si votre classe implémente cette interface, alors vous pourrez modifier le comportement de votre objet lorsqu'il est parcouru. Cette interface comporte 5 méthodes :

- `current` : renvoie l'élément courant ;
- `key` : retourne la clé de l'élément courant ;
- `next` : déplace le pointeur sur l'élément suivant ;
- `rewind` : remet le pointeur sur le premier élément ;
- `valid` : vérifie si la position courante est valide.

En écrivant ces méthodes, on pourra renvoyer la valeur qu'on veut, et pas forcément la valeur de l'attribut actuellement lu. Imaginons qu'on ait un attribut qui soit un array. On pourrait très bien créer un petit script qui, au lieu de parcourir l'objet,

parcourt le tableau ! Je vous laisse essayer. Vous aurez besoin d'un attribut `$position` qui stocke la position actuelle.

Correction :

Code : PHP

```php
<?php
class MaClasse implements Iterator
{
  private $position = 0;
  private $tableau = array('Premier élément', 'Deuxième élément',
'Troisième élément', 'Quatrième élément', 'Cinquième élément');

  /**
* Retourne l'élément courant du tableau.
*/
  public function current()
  {
    return $this->tableau[$this->position];
  }

  /**
* Retourne la clé actuelle (c'est la même que la position dans
notre cas).
*/
  public function key()
  {
    return $this->position;
  }

  /**
* Déplace le curseur vers l'élément suivant.
*/
  public function next()
  {
    $this->position++;
  }

  /**
* Remet la position du curseur à 0.
*/
  public function rewind()
  {
    $this->position = 0;
  }

  /**
* Permet de tester si la position actuelle est valide.
*/
  public function valid()
  {
    return isset($this->tableau[$this->position]);
  }
}

$objet = new MaClasse;

foreach ($objet as $key => $value)
{
  echo $key, ' => ', $value, '<br />';
}
?>
```

Ce qui affichera :

Résultat affiché par le script

Alors, pratique non ?

L'interface SeekableIterator

Cette interface hérite de l'interface `Iterator`, on n'aura donc pas besoin d'implémenter les deux à notre classe.

`SeekableIterator` ajoute une méthode à la liste des méthodes d'`Iterator` : la méthode `seek`. Cette méthode permet de placer le curseur interne à une position précise. Elle demande donc un argument : la position du curseur à laquelle il faut le placer. Je vous déconseille de modifier directement l'attribut `$position` afin d'assigner directement la valeur de l'argument à `$position`. En effet, qui vous dit que la valeur de l'argument est une position valide ?

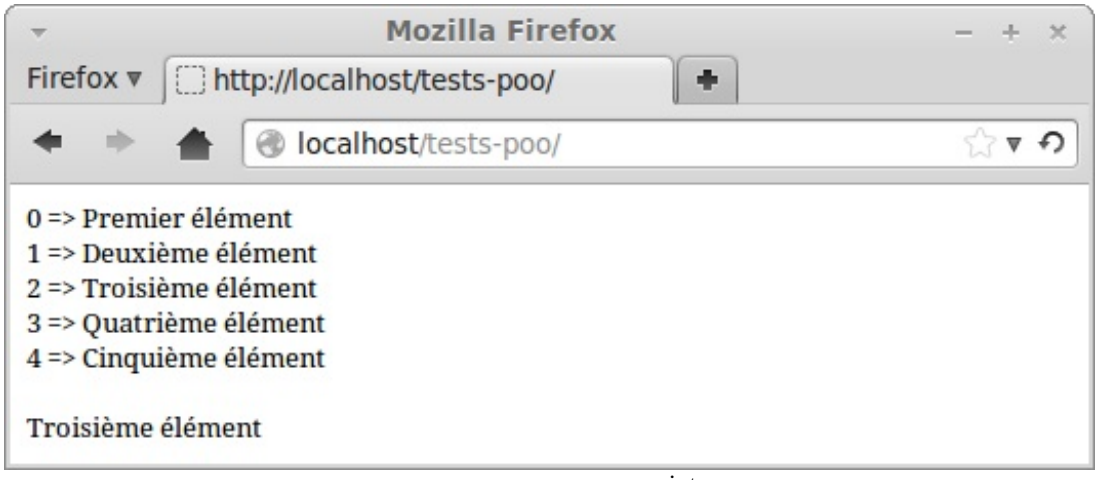
Je vous laisse réfléchir quant à l'implémentation de cette méthode. Voici la correction (j'ai repris la dernière classe) :

Code : PHP

```php
<?php
class MaClasse implements SeekableIterator
{
  private $position = 0;
  private $tableau = array('Premier élément', 'Deuxième élément',
'Troisième élément', 'Quatrième élément', 'Cinquième élément');

  /**
* Retourne l'élément courant du tableau.
*/
  public function current()
  {
    return $this->tableau[$this->position];
  }

  /**
* Retourne la clé actuelle (c'est la même que la position dans
notre cas).
*/
  public function key()
  {
    return $this->position;
  }

  /**
* Déplace le curseur vers l'élément suivant.
*/
  public function next()
  {
    $this->position++;
  }

  /**
* Remet la position du curseur à 0.
```

```
    */
      public function rewind()
      {
        $this->position = 0;
      }

      /**
    * Déplace le curseur interne.
    */
      public function seek($position)
      {
        $anciennePosition = $this->position;
        $this->position = $position;

        if (!$this->valid())
        {
          trigger_error('La position spécifiée n\'est pas valide',
E_USER_WARNING);
          $this->position = $anciennePosition;
        }
      }

      /**
    * Permet de tester si la position actuelle est valide.
    */
      public function valid()
      {
        return isset($this->tableau[$this->position]);
      }
    }

    $objet = new MaClasse;

    foreach ($objet as $key => $value)
    {
      echo $key, ' => ', $value, '<br />';
    }

    $objet->seek(2);
    echo '<br />', $objet->current();
    ?>
```

Ce qui affichera :



Résultat affiché par le script

L'interface ArrayAccess

Nous allons enfin, grâce à cette interface, pouvoir placer des crochets à la suite de notre objet avec la clé à laquelle accéder, comme sur un vrai tableau ! L'interface `ArrayAccess` liste quatre méthodes :

- `offsetExists` : méthode qui vérifiera l'existence de la clé entre crochets lorsque l'objet est passé à la fonction `isset` ou `empty` (cette valeur entre crochet est passé à la méthode en paramètre) ;
- `offsetGet` : méthode appelée lorsqu'on fait un simple `$obj['clé']`. La valeur 'clé' est donc passée à la méthode `offsetGet` ;
- `offsetSet` : méthode appelée lorsqu'on assigne une valeur à une entrée. Cette méthode reçoit donc deux arguments, la valeur de la clé et la valeur qu'on veut lui assigner.
- `offsetUnset` : méthode appelée lorsqu'on appelle la fonction `unset` sur l'objet avec une valeur entre crochets. Cette méthode reçoit un argument, la valeur qui est mise entre les crochets.

Maintenant, votre mission est d'implémenter cette interface et de gérer l'attribut `$tableau` grâce aux quatre méthodes. C'est parti ! 

Correction :

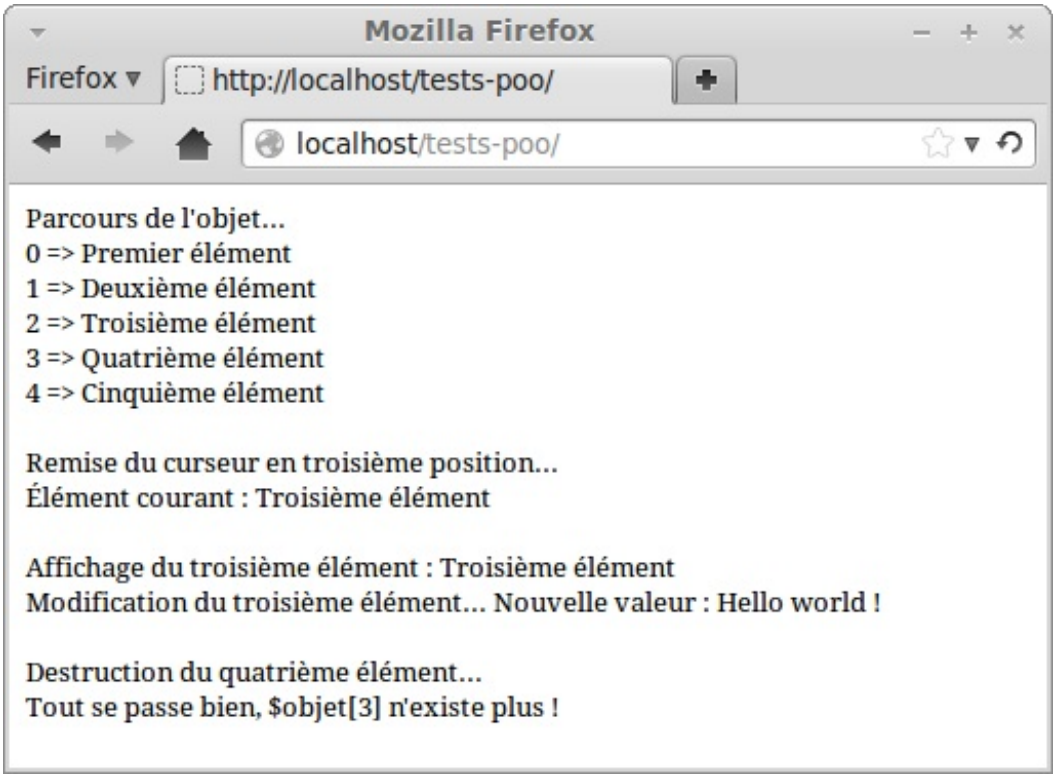
Code : PHP

```php
<?php
class MaClasse implements SeekableIterator, ArrayAccess
{
  private $position = 0;
  private $tableau = array('Premier élément', 'Deuxième élément',
'Troisième élément', 'Quatrième élément', 'Cinquième élément');


  /* MÉTHODES DE L'INTERFACE SeekableIterator */


  /**
* Retourne l'élément courant du tableau.
*/
  public function current()
  {
    return $this->tableau[$this->position];
  }

  /**
* Retourne la clé actuelle (c'est la même que la position dans
notre cas).
*/
  public function key()
  {
    return $this->position;
  }

  /**
* Déplace le curseur vers l'élément suivant.
*/
  public function next()
  {
    $this->position++;
  }

  /**
* Remet la position du curseur à 0.
*/
  public function rewind()
  {
    $this->position = 0;
  }

  /**
```

```
    * Déplace le curseur interne.
    */
      public function seek($position)
      {
        $anciennePosition = $this->position;
        $this->position = $position;

        if (!$this->valid())
        {
          trigger_error('La position spécifiée n\'est pas valide',
    E_USER_WARNING);
          $this->position = $anciennePosition;
        }
      }

      /**
    * Permet de tester si la position actuelle est valide.
    */
      public function valid()
      {
        return isset($this->tableau[$this->position]);
      }


      /* MÉTHODES DE L'INTERFACE ArrayAccess */


      /**
    * Vérifie si la clé existe.
    */
      public function offsetExists($key)
      {
        return isset($this->tableau[$key]);
      }

      /**
    * Retourne la valeur de la clé demandée.
    * Une notice sera émise si la clé n'existe pas, comme pour les
    vrais tableaux.
    */
      public function offsetGet($key)
      {
        return $this->tableau[$key];
      }

      /**
    * Assigne une valeur à une entrée.
    */
      public function offsetSet($key, $value)
      {
        $this->tableau[$key] = $value;
      }

      /**
    * Supprime une entrée et émettra une erreur si elle n'existe pas,
    comme pour les vrais tableaux.
    */
      public function offsetUnset($key)
      {
        unset($this->tableau[$key]);
      }
    }

    $objet = new MaClasse;

    echo 'Parcours de l\'objet...<br />';
    foreach ($objet as $key => $value)
    {
      echo $key, ' => ', $value, '<br />';
    }
```

```php
echo '<br />Remise du curseur en troisième position...<br />';
$objet->seek(2);
echo 'Élément courant : ', $objet->current(), '<br />';

echo '<br />Affichage du troisième élément : ', $objet[2], '<br />';
echo 'Modification du troisième élément... ';
$objet[2] = 'Hello world !';
echo 'Nouvelle valeur : ', $objet[2], '<br /><br />';

echo 'Destruction du quatrième élément...<br />';
unset($objet[3]);

if (isset($objet[3]))
{
  echo '$objet[3] existe toujours... Bizarre...';
}
else
{
  echo 'Tout se passe bien, $objet[3] n\'existe plus !';
}
?>
```

Ce qui affiche :



Résultat affiché par le script

Alors, on se rapproche vraiment du comportement d'un tableau, n'est-ce pas ? On peut faire tout ce qu'on veut, comme sur un tableau ! Enfin, il manque juste un petit quelque chose pour que ce soit absolument parfait...

L'interface Countable

Et voici la dernière interface que je vous présenterai. Elle contient une méthode : la méthode `count`. Celle-ci doit obligatoirement renvoyer un entier qui sera la valeur renvoyée par la fonction `count` appelée sur notre objet. Cette méthode n'est pas bien compliquée à implémenter, il suffit juste de retourner le nombre d'entrées de notre tableau.

Correction :

Code : PHP

```php
<?php
class MaClasse implements SeekableIterator, ArrayAccess, Countable
{
  private $position = 0;
  private $tableau = array('Premier élément', 'Deuxième élément',
'Troisième élément', 'Quatrième élément', 'Cinquième élément');


  /* MÉTHODES DE L'INTERFACE SeekableIterator */


  /**
* Retourne l'élément courant du tableau.
*/
  public function current()
  {
    return $this->tableau[$this->position];
  }

  /**
* Retourne la clé actuelle (c'est la même que la position dans
notre cas).
*/
  public function key()
  {
    return $this->position;
  }

  /**
* Déplace le curseur vers l'élément suivant.
*/
  public function next()
  {
    $this->position++;
  }

  /**
* Remet la position du curseur à 0.
*/
  public function rewind()
  {
    $this->position = 0;
  }

  /**
* Déplace le curseur interne.
*/
  public function seek($position)
  {
    $anciennePosition = $this->position;
    $this->position = $position;

    if (!$this->valid())
    {
      trigger_error('La position spécifiée n\'est pas valide',
E_USER_WARNING);
      $this->position = $anciennePosition;
    }
  }

  /**
* Permet de tester si la position actuelle est valide.
*/
  public function valid()
  {
    return isset($this->tableau[$this->position]);
```

```
  }


  /* MÉTHODES DE L'INTERFACE ArrayAccess */


  /**
* Vérifie si la clé existe.
*/
  public function offsetExists($key)
  {
    return isset($this->tableau[$key]);
  }

  /**
* Retourne la valeur de la clé demandée.
* Une notice sera émise si la clé n'existe pas, comme pour les
vrais tableaux.
*/
  public function offsetGet($key)
  {
    return $this->tableau[$key];
  }

  /**
* Assigne une valeur à une entrée.
*/
  public function offsetSet($key, $value)
  {
    $this->tableau[$key] = $value;
  }

  /**
* Supprime une entrée et émettra une erreur si elle n'existe pas,
comme pour les vrais tableaux.
*/
  public function offsetUnset($key)
  {
    unset($this->tableau[$key]);
  }


  /* MÉTHODES DE L'INTERFACE Countable */


  /**
* Retourne le nombre d'entrées de notre tableau.
*/
  public function count()
  {
    return count($this->tableau);
  }
}

$objet = new MaClasse;

echo 'Parcours de l\'objet...<br />';
foreach ($objet as $key => $value)
{
  echo $key, ' => ', $value, '<br />';
}

echo '<br />Remise du curseur en troisième position...<br />';
$objet->seek(2);
echo 'Élément courant : ', $objet->current(), '<br />';

echo '<br />Affichage du troisième élément : ', $objet[2], '<br />';
echo 'Modification du troisième élément... ';
$objet[2] = 'Hello world !';
echo 'Nouvelle valeur : ', $objet[2], '<br /><br />';
```

```
echo 'Actuellement, mon tableau comporte ', count($objet), '
entrées<br /><br />';

echo 'Destruction du quatrième élément...<br />';
unset($objet[3]);

if (isset($objet[3]))
{
  echo '$objet[3] existe toujours... Bizarre...';
}
else
{
  echo 'Tout se passe bien, $objet[3] n\'existe plus !';
}

echo '<br /><br />Maintenant, il n\'en comporte plus que ',
count($objet), ' !';
?>
```

Ce qui affichera :

Résultat affiché par le script

Bonus : la classe `ArrayIterator`

Je dois vous avouer quelque chose : la classe que nous venons de créer pour pouvoir créer des « objets-tableaux » existe déjà. En effet, PHP possède nativement une classe nommée `ArrayIterator`. Comme notre précédente classe, celle-ci implémente les quatre interfaces qu'on a vues.

Mais pourquoi tu nous as fais faire tout ça ?!

Pour vous faire pratiquer, tiens.

Sachez que réécrire des classes ou fonctions natives de PHP est un **excellent** exercice (et c'est valable pour tous les langages de programmation). Je ne vais pas m'attarder sur cette classe, étant donné qu'elle s'utilise exactement comme la nôtre. Elle possède les mêmes méthodes, à une différence près : cette classe implémente un constructeur qui accepte un tableau en guise d'argument. Comme vous l'aurez deviné, c'est ce tableau qui sera « transformé » en objet. Ainsi, si vous faites un `echo $monInstanceArrayIterator['cle']`, alors à l'écran s'affichera l'entrée qui a pour clé `cle` du tableau passé en paramètre.

En résumé

- Une interface représente un **comportement**.
- Les interfaces ne sont autre que des classes 100% abstraites.
- Une interface s'utilise dans une classe grâce au mot-clé `implements`.
- Il est possibles d'hériter ses interfaces grâce au mot-clé `extends`.
- Il existe tout un panel d'interfaces pré-définies vous permettant de réaliser tout un tas de fonctionnalités intéressantes, comme la création « d'objets-tableaux ».

Les exceptions

Actuellement, vous connaissez les erreurs fatales, les alertes, les erreurs d'analyse ou encore les notices. Nous allons découvrir dans ce chapitre une façon différente de gérer les erreurs. Nous allons, en quelque sorte, créer **nos propres types d'erreurs**. Les *exceptions* sont des erreurs assez différentes qui ne fonctionnent pas de la même manière. Comme vous le verrez, cette nouvelle façon de gérer ses erreurs est assez pratique. Par exemple, vous pouvez **attraper** l'erreur pour l'afficher comme vous voulez plutôt que d'avoir un fameux et plutôt laid **Warning**.

Cela ne s'arrêtera pas là. Puisque la gestion d'erreurs est assez importante sur un site, je dédie une partie de ce chapitre au moyen de gérer ses erreurs facilement et proprement, que ce soit les erreurs que vous connaissez déjà ou les exceptions.

Une différente gestion des erreurs

Les exceptions, comme nous l'avons évoqué dans l'introduction de ce chapitre, sont une façon différente de gérer les erreurs. Celles-ci sont en fait des erreurs lancées par PHP lorsque quelque chose qui ne va pas est survenu. Nous allons commencer par lancer nos propres exceptions. Pour cela, on va devoir s'intéresser à la classe `Exception`.

Lancer une exception

Une exception peut être lancée depuis n'importe où dans le code. Quand on lance une exception, on doit, en gros, lancer une instance de la classe `Exception`. Cet objet lancé contiendra le message d'erreur ainsi que son code. Pensez à spécifier au moins le message d'erreur, bien que celui-ci soit facultatif. Je ne vois pas l'intérêt de lancer une exception sans spécifier l'erreur rencontrée. Pour le code d'erreur, il n'est pas (pour l'instant) très utile. Libre à vous de le spécifier ou pas. Le troisième et dernier argument est l'exception précédente. Là aussi, spécifiez-là si vous le souhaitez, mais ce n'est pas indispensable.

Passons à l'acte. Nous allons créer une simple fonction qui aura pour rôle d'additionner un nombre avec un autre. Si l'un des deux nombres n'est pas numérique, alors on lancera une exception de type `Exception` à l'aide du mot `throw` (= lancer). On va donc **lancer** une **nouvelle** `Exception`. Le constructeur de la classe `Exception` demande en paramètre le message d'erreur, son code et l'exception précédente. Ces trois paramètres sont facultatifs.

Code : PHP

```php
<?php
function additionner($a, $b)
{
  if (!is_numeric($a) OR !is_numeric($b))
  {
    // On lance une nouvelle exception grâce à throw et on instancie directement un objet de la classe Exception.
    throw new Exception('Les deux paramètres doivent être des nombres');
  }

  return $a + $b;
}

echo additionner(12, 3), '<br />';
echo additionner('azerty', 54), '<br />';
echo additionner(4, 8);
?>
```

Et là, vous avez :

Résultat affiché par le script

Et voilà votre première exception qui apparaît, d'une façon assez désagréable, devant vos yeux ébahis. Décortiquons ce que PHP veut nous dire.

Premièrement, il génère une erreur fatale. Et oui, une exception non attrapée génère automatiquement une erreur fatale. Nous verrons plus tard ce que signifie « attraper ».

Deuxièmement, il nous dit *Uncaught exception 'Exception' with message 'Les deux paramètres doivent être des nombres'* ce qui signifie « *Exception 'Exception' non attrapée avec le message 'Les deux paramètres doivent être des nombres'* ». Ce passage se passera de commentaire, la traduction parle d'elle-même : on n'a pas attrapé l'exception 'Exception' (= le nom de la classe instanciée par l'objet qui a été lancé) avec tel message (ici, c'est le message spécifié dans le constructeur).

Et pour finir, PHP nous dit où a été lancée l'exception, depuis quelle fonction, à quelle ligne, etc.

Maintenant, puisque PHP n'a pas l'air content que l'on n'ait pas « attrapé » cette exception, et bien c'est ce que nous allons faire.

Ne lancez jamais d'exception dans un destructeur. Si vous faites une telle chose, vous aurez une erreur fatale : Exception thrown without a stack frame in Unknown on line 0. Cette erreur peut aussi être lancée dans un autre cas évoqué plus tard.

Attraper une exception

Afin d'attraper une exception, il faut d'abord qu'elle soit lancée. Le problème, c'est qu'on ne peut pas dire à PHP que toutes les exceptions lancées doivent être attrapées : c'est à nous de lui dire que l'on va **essayer** d'effectuer telle ou telle instruction et, si une exception est lancée, alors on **attrapera** celle-ci afin qu'aucune erreur fatale ne soit lancée et que de tels messages ne s'affichent plus.

Nous allons dès à présent placer nos instructions dans un bloc `try`. Celles-ci seront à placer entre une paire d'accolades. Qui dit bloc `try` dit aussi bloc `catch` car l'un ne va pas sans l'autre (si vous mettez l'un sans l'autre, une erreur d'analyse sera levée). Ce bloc `catch` a une petite particularité. Au lieu de placer `catch` suivi directement des deux accolades, nous allons devoir spécifier, entre une paire de parenthèses placée entre `catch` et l'accolade ouvrante, le type d'exception à attraper suivi d'une variable qui représentera cette exception. C'est à partir d'elle que l'on pourra récupérer le message ou le code d'erreur grâce aux méthodes de la classe.

Commençons en douceur en attrapant simplement toute exception `Exception` :

Code : PHP

```php
<?php
function additionner($a, $b)
{
  if (!is_numeric($a) OR !is_numeric($b))
  {
    throw new Exception('Les deux paramètres doivent être des nombres'); // On lance une nouvelle exception si l'un des deux paramètres n'est pas un nombre.
```

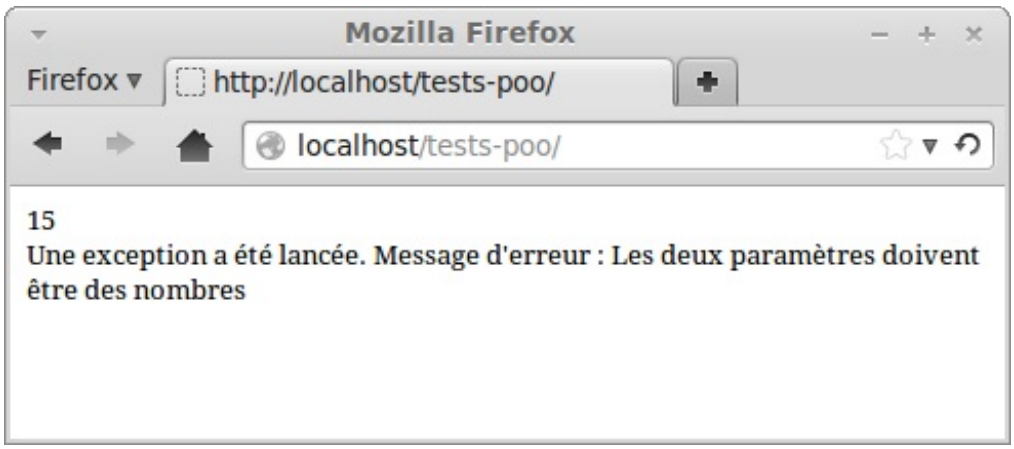
```
  }

  return $a + $b;
}

try // On va essayer d'effectuer les instructions situées dans ce
bloc.
{
  echo additionner(12, 3), '<br />';
  echo additionner('azerty', 54), '<br />';
  echo additionner(4, 8);
}

catch (Exception $e) // On va attraper les exceptions "Exception"
s'il y en a une qui est levée.
{

}
?>
```

Et là, miracle, vous n'avez plus que **15** qui s'affiche, et plus d'erreur ! Par contre, les deux autres résultats ne sont pas affichés, et il serait intéressant de savoir pourquoi. Nous allons afficher le message d'erreur. Pour ce faire, il faut appeler la méthode `getMessage()`. Si vous souhaitez récupérer le code d'erreur, il faut appeler `getCode()`.

Code : PHP

```
<?php
function additionner($a, $b)
{
  if (!is_numeric($a) OR !is_numeric($b))
  {
    throw new Exception('Les deux paramètres doivent être des
nombres');
  }

  return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans
ce bloc.
{
  echo additionner(12, 3), '<br />';
  echo additionner('azerty', 54), '<br />';
  echo additionner(4, 8);
}

catch (Exception $e) // On va attraper les exceptions "Exception"
s'il y en a une qui est levée
{
  echo 'Une exception a été lancée. Message d\'erreur : ', $e-
>getMessage();
}
?>
```

Ce qui affichera :



Résultat affiché par le script

Comme vous pouvez le constater, la troisième instruction du bloc `try` n'a pas été exécutée. C'est normal puisque la deuxième instruction a interrompu la lecture du bloc. Si vous interceptez les exceptions comme nous l'avons fait, alors le script n'est pas interrompu. En voici la preuve :

Code : PHP

```php
<?php
function additionner($a, $b)
{
  if (!is_numeric($a) OR !is_numeric($b))
  {
    throw new Exception('Les deux paramètres doivent être des nombres');
  }

  return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans ce bloc.
{
  echo additionner(12, 3), '<br />';
  echo additionner('azerty', 54), '<br />';
  echo additionner(4, 8);
}

catch (Exception $e) // Nous allons attraper les exceptions "Exception" s'il y en a une qui est levée.
{
  echo 'Une exception a été lancée. Message d\'erreur : ', $e->getMessage();
}

echo 'Fin du script'; // Ce message s'affiche, ça prouve bien que le script est exécuté jusqu'au bout.
?>
```

Vous vous demandez sans doute pourquoi on doit spécifier le nom de l'exception à intercepter puisque c'est toujours une instance de la classe `Exception`. En fait, la classe `Exception` est la classe de base pour toute exception qui doit être lancée, ce qui signifie que l'on peut lancer n'importe quelle autre instance d'une classe, du moment **qu'elle hérite de la classe `Exception`** !

Des exceptions spécialisées

Hériter la classe Exception

PHP nous offre la possibilité d'hériter la classe `Exception` afin de personnaliser nos exceptions. Par exemple, nous pouvons créer une classe `MonException` qui réécrira des méthodes de la classe `Exception` ou en créera de nouvelles qui lui seront propres.

Avant de foncer tête baissée dans l'écriture de notre classe personnalisée, encore faut-il savoir quelles méthodes et attributs seront disponibles. Voici la liste des attributs et méthodes de la classe `Exception` tirée de la documentation :

Code : PHP

```php
<?php
class Exception
{
  protected $message = 'exception inconnu'; // Message de
l'exception.
  protected $code = 0; // Code de l'exception défini par
l'utilisateur.
  protected $file; // Nom du fichier source de l'exception.
  protected $line; // Ligne de la source de l'exception.

  final function getMessage(); // Message de l'exception.
  final function getCode(); // Code de l'exception.
  final function getFile(); // Nom du fichier source.
  final function getLine(); // Ligne du fichier source.
  final function getTrace(); // Un tableau de backtrace().
  final function getTraceAsString(); // Chaîne formattée de trace.

  /* Remplacable */
  function __construct ($message = NULL, $code = 0);
  function __toString(); // Chaîne formatée pour l'affichage.
}
?>
```

Ainsi, nous voyons que l'on a accès aux attributs protégés de la classe et qu'on peut réécrire les méthodes `__construct` et `__toString`. Toutes les autres méthodes sont finales, nous n'avons donc pas le droit de les réécrire.

Nous allons donc créer notre classe `MonException` qui, par exemple, réécrira le constructeur en rendant obligatoire le premier argument ainsi que la méthode `__toString` pour n'afficher que le message d'erreur (c'est uniquement ça qui nous intéresse).

Code : PHP

```php
<?php
class MonException extends Exception
{
  public function __construct($message, $code = 0)
  {
    parent::__construct($message, $code);
  }

  public function __toString()
  {
    return $this->message;
  }
}
?>
```

Maintenant, comme vous l'aurez peut-être deviné, nous n'allons pas lancer d'exception `Exception` mais une exception de type `MonException`.

Dans notre script, nous allons désormais attraper uniquement les exceptions `MonException`, ce qui éclaircira le code car c'est une manière de se dire que l'on ne travaille, dans le bloc `try`, qu'avec des instructions susceptibles de lancer des exceptions de type `MonException`. Exemple :

Code : PHP

```php
<?php
class MonException extends Exception
{
  public function __construct($message, $code = 0)
  {
    parent::__construct($message, $code);
  }

  public function __toString()
  {
    return $this->message;
  }
}

function additionner($a, $b)
{
  if (!is_numeric($a) OR !is_numeric($b))
  {
    throw new MonException('Les deux paramètres doivent être des
nombres'); // On lance une exception "MonException".
  }

  return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans
ce bloc.
{
  echo additionner(12, 3), '<br />';
  echo additionner('azerty', 54), '<br />';
  echo additionner(4, 8);
}

catch (MonException $e) // Nous allons attraper les exceptions
"MonException" s'il y en a une qui est levée.
{
  echo $e; // On affiche le message d'erreur grâce à la méthode
__toString que l'on a écrite.
}

echo '<br />Fin du script'; // Ce message s'affiche, ça prouve bien
que le script est exécuté jusqu'au bout.
?>
```

Ainsi, nous avons attrapé uniquement les exceptions de type `MonException`. Essayez de lancer une exception `Exception` à la place et vous verrez qu'elle ne sera pas attrapée. Si vous décidez d'attraper, dans le bloc `catch`, les exceptions `Exception`, alors **toutes** les exceptions seront attrapées car elles héritent toutes de cette classe. En fait, quand vous héritez une classe d'une autre et que vous décidez d'attraper les exceptions de la classe parente, alors celles de la classe enfant le seront aussi.

Emboîter plusieurs blocs `catch`

Il est possible d'emboîter plusieurs blocs `catch`. En effet, vous pouvez mettre un premier bloc attrapant les exceptions `MonException` suivi d'un deuxième attrapant les exceptions `Exception`. Si vous effectuez une telle opération et qu'une exception est lancée, alors PHP ira dans le premier bloc pour voir si ce type d'exception doit être attrapé, si tel n'est pas le cas il va dans le deuxième, etc., jusqu'à ce qu'il tombe sur un bloc qui l'attrape. Si aucun ne l'attrape, alors une erreur fatale est levée. Exemple :

Code : PHP

```php
<?php
class MonException extends Exception
{
  public function __construct($message, $code = 0)
  {
```

```
        parent::__construct($message, $code);
    }

    public function __toString()
    {
        return $this->message;
    }
}

function additionner($a, $b)
{
    if (!is_numeric($a) OR !is_numeric($b))
    {
        throw new MonException('Les deux paramètres doivent être des
nombres'); // On lance une exception "MonException".
    }

    if (func_num_args() > 2)
    {
        throw new Exception('Pas plus de deux arguments ne doivent être
passés à la fonction'); // Cette fois-ci, on lance une exception
"Exception".
    }

    return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans
ce bloc.
{
    echo additionner(12, 3), '<br />';
    echo additionner(15, 54, 45), '<br />';
}

catch (MonException $e) // Nous allons attraper les exceptions
"MonException" s'il y en a une qui est levée.
{
    echo '[MonException] : ', $e; // On affiche le message d'erreur
grâce à la méthode __toString que l'on a écrite.
}

catch (Exception $e) // Si l'exception n'est toujours pas attrapée,
alors nous allons essayer d'attraper l'exception "Exception".
{
    echo '[Exception] : ', $e->getMessage(); // La méthode
__toString() nous affiche trop d'informations, nous voulons juste le
message d'erreur.
}

echo '<br />Fin du script'; // Ce message s'affiche, cela prouve
bien que le script est exécuté jusqu'au bout.
?>
```

Cette fois-ci, aucune exception `MonException` n'est lancée, mais une exception `Exception` l'a été. PHP va donc effectuer les opérations demandées dans le deuxième bloc `catch`.

Exemple concret : la classe PDOException

Vous connaissez sans doute la bibliothèque PDO (lire le tutoriel à ce sujet). Dans ce tutoriel, il vous est donné une technique pour capturer les erreurs générées par PDO : comme vous le savez maintenant, ce sont des exceptions ! Et PDO a sa classe d'exception : `PDOException`. Celle-ci n'hérite pas directement de la classe `Exception` mais de `RuntimeException`. Cette classe n'a rien de plus que sa classe mère, il s'agit juste d'une classe qui est instanciée pour émettre une exception lors de l'exécution du script. Il existe une classe pour chaque circonstance dans laquelle l'exception est lancée : vous trouverez la liste des exceptions ici.

Cette classe `PDOException` est donc la classe personnalisée pour émettre une exception par la classe PDO ou PDOStatement.

Sachez d'ailleurs que si une extension orientée objet doit émettre une erreur, elle émettra une exception.

Bref, voici un exemple d'utilisation de `PDOException` :

Code : PHP

```php
<?php
try
{
  $db = new PDO('mysql:host=localhost;dbname=tests', 'root', ''); // Tentative de connexion.
  echo 'Connexion réussie !'; // Si la connexion a réussi, alors cette instruction sera exécutée.
}

catch (PDOException $e) // On attrape les exceptions PDOException.
{
  echo 'La connexion a échoué.<br />';
  echo 'Informations : [', $e->getCode(), '] ', $e->getMessage(); // On affiche le n° de l'erreur ainsi que le message.
}
?>
```

Exceptions pré-définies

Il existe toute une quantité d'exceptions pré-définies. Vous pouvez obtenir cette liste sur la documentation. Au lieu de lancer tout le temps une exception en instanciant `Exception`, il est préférable d'instancier la classe adaptée à la situation. Par exemple, reprenons le code proposé en début de chapitre :

Code : PHP

```php
<?php
function additionner($a, $b)
{
  if (!is_numeric($a) OR !is_numeric($b))
  {
    // On lance une nouvelle exception grâce à throw et on instancie directement un objet de la classe Exception.
    throw new Exception('Les deux paramètres doivent être des nombres');
  }

  return $a + $b;
}

echo additionner(12, 3), '<br />';
echo additionner('azerty', 54), '<br />';
echo additionner(4, 8);
```

La classe à instancier ici est celle qui doit l'être lorsqu'un paramètre est invalide. On regarde la documentation, et on tombe sur `InvalidArgumentException`. Le code donnerait donc :

Code : PHP

```php
<?php
function additionner($a, $b)
{
  if (!is_numeric($a) OR !is_numeric($b))
  {
    throw new InvalidArgumentException('Les deux paramètres doivent être des nombres');
```

```
    }

    return $a + $b;
}

echo additionner(12, 3), '<br />';
echo additionner('azerty', 54), '<br />';
echo additionner(4, 8);
```

Cela permet de mieux se repérer dans le code et surtout de mieux cibler les erreurs grâce aux multiples blocs `catch`. 

Gérer les erreurs facilement

Convertir les erreurs en exceptions

Nous allons voir une dernière chose avant de passer au prochain chapitre : la manière de convertir les erreurs fatales, alertes et notices en exceptions. Pour cela, nous allons avoir besoin de la fonction set_error_handler. Celle-ci permet d'enregistrer une fonction en callback qui sera appelée à chaque fois que l'une de ces trois erreurs sera lancée. Il n'y a pas de rapport direct avec les exceptions : c'est à nous de l'établir.

Notre fonction, que l'on nommera `error2exception` par exemple, doit demander entre deux et cinq paramètres :

- Le numéro de l'erreur **(obligatoire)**.
- Le message d'erreur **(obligatoire)**.
- Le nom du fichier dans lequel l'erreur a été lancée.
- Le numéro de la ligne à laquelle l'erreur a été identifiée.
- Un tableau avec toutes les variables qui existaient jusqu'à ce que l'erreur soit rencontrée.

Nous n'allons pas prêter attention au dernier paramètre, juste aux quatre premiers. Nous allons créer notre propre classe `MonException` qui hérite non pas de `Exception` mais de `ErrorException`. Bien sûr, comme je l'ai déjà dit plus haut, toutes les exceptions héritent de la classe `Exception` : `ErrorException` n'échappe pas à la règle et hérite de celle-ci.

La fonction `set_error_handler` demande deux paramètres. Le premier est la fonction à appeler, et le deuxième, les erreurs à intercepter. Par défaut, ce paramètre intercepte toutes les erreurs, y compris les erreurs strictes.

Le constructeur de la classe `ErrorException` demande cinq paramètres, tous facultatifs :

- Le message d'erreur.
- Le code de l'erreur.
- La sévérité de l'erreur (erreur fatale, alerte, notice, etc.) représentées par des constantes pré-définies.
- Le fichier où l'erreur a été rencontrée.
- La ligne à laquelle l'erreur a été rencontrée.

Voici à quoi pourrait ressembler le code de base :

Code : PHP

```
<?php
class MonException extends ErrorException
{
  public function __toString()
  {
    switch ($this->severity)
    {
      case E_USER_ERROR : // Si l'utilisateur émet une erreur
fatale;
        $type = 'Erreur fatale';
        break;
```

```
            case E_WARNING : // Si PHP émet une alerte.
            case E_USER_WARNING : // Si l'utilisateur émet une alerte.
              $type = 'Attention';
              break;

            case E_NOTICE : // Si PHP émet une notice.
            case E_USER_NOTICE : // Si l'utilisateur émet une notice.
              $type = 'Note';
              break;

            default : // Erreur inconnue.
              $type = 'Erreur inconnue';
              break;
        }

        return '<strong>' . $type . '</strong> : [' . $this->code . '] '
. $this->message . '<br /><strong>' . $this->file . '</strong> à la
ligne <strong>' . $this->line . '</strong>';
    }
}

function error2exception($code, $message, $fichier, $ligne)
{
  // Le code fait office de sévérité.
  // Reportez-vous aux constantes prédéfinies pour en savoir plus.
  // http://fr2.php.net/manual/fr/errorfunc.constants.php
  throw new MonException($message, 0, $code, $fichier, $ligne);
}

set_error_handler('error2exception');
?>
```

Vous voyez que dans la méthode `__toString` je mettais à chaque fois E_X et E_USER_X. Les erreurs du type E_X sont générées par PHP et les erreurs E_USER_X sont générées par l'utilisateur grâce à trigger_error. Les erreurs E_ERROR (donc les erreurs fatales générées par PHP) ne peuvent être interceptées, c'est la raison pour laquelle je ne l'ai pas placé dans le switch.

Ensuite, à vous de faire des tests, vous verrez bien que ça fonctionne à merveille. Mais gardez bien ça en tête : avec ce code, toutes les erreurs (même les notices) qui ne sont pas dans un bloc `try` **interrompront le script** car elles émettront une exception !

On aurait très bien pu utiliser la classe `Exception` mais `ErrorException` a été conçu exactement pour ce genre de chose. Nous n'avons pas besoin de créer d'attribut stockant la sévérité de l'erreur ou de réécrire le constructeur pour y stocker le nom du fichier et la ligne à laquelle s'est produite l'erreur.

Personnaliser les exceptions non attrapées

Nous avons réussi à transformer toutes nos erreurs en exceptions en les interceptant grâce à `set_error_handler`. Étant donné que la moindre erreur lèvera une exception, il serait intéressant de *personnaliser* l'erreur générée par PHP. Ce que je veux dire par là, c'est qu'une exception non attrapée génère une longue et laide erreur fatale. Nous allons donc, comme pour les erreurs, **intercepter** les exceptions grâce à `set_exception_handler`. Cette fonction demande un seul argument : le nom de la fonction à appeler lorsqu'une exception est lancée. La fonction de callback doit accepter un argument : c'est un objet représentant l'exception.

Voici un exemple d'utilisation en reprenant le précédent code :

Code : PHP

```
<?php
class MonException extends ErrorException
{
  public function __toString()
  {
    switch ($this->severity)
    {
```

```
            case E_USER_ERROR : // Si l'utilisateur émet une erreur
        fatale.
                $type = 'Erreur fatale';
                break;

            case E_WARNING : // Si PHP émet une alerte.
            case E_USER_WARNING : // Si l'utilisateur émet une alerte.
                $type = 'Attention';
                break;

            case E_NOTICE : // Si PHP émet une notice.
            case E_USER_NOTICE : // Si l'utilisateur émet une notice.
                $type = 'Note';
                break;

            default : // Erreur inconnue.
                $type = 'Erreur inconnue';
                break;
        }

        return '<strong>' . $type . '</strong> : [' . $this->code . '] '
    . $this->message . '<br /><strong>' . $this->file . '</strong> à la
    ligne <strong>' . $this->line . '</strong>';
      }
    }

    function error2exception($code, $message, $fichier, $ligne)
    {
      // Le code fait office de sévérité.
      // Reportez-vous aux constantes prédéfinies pour en savoir plus.
      // http://fr2.php.net/manual/fr/errorfunc.constants.php
      throw new MonException($message, 0, $code, $fichier, $ligne);
    }

    function customException($e)
    {
    echo 'Ligne ', $e->getLine(), ' dans ', $e->getFile(), '<br
    /><strong>Exception lancée</strong> : ', $e->getMessage();
    }

    set_error_handler('error2exception');
    set_exception_handler('customException');
    ?>
```

Je dis bien que l'exception est **interceptée** et non **attrapée** ! Cela signifie que l'on attrape l'exception, qu'on effectue des opérations puis qu'on la relâche. Le script, une fois `customException` appelé, est automatiquement **interrompu**.

Ne lancez jamais d'exception dans votre gestionnaire d'exception (ici `customException`). En effet, cela créerait une boucle infinie puisque votre gestionnaire lance lui-même une exception. L'erreur lancée est la même que celle vue précédemment : il s'agit de l'erreur fatale « Exception thrown without a stack frame in Unknown on line 0 ».

En résumé

- Une exception est une erreur que l'on peut attraper grâce aux mots-clé `try` et `catch`.
- Une exception est une erreur que l'on peut personnaliser, que ce soit au niveau de son affichage ou au niveau de ses renseignements (fichier concerné par l'erreur, le numéro de la ligne, etc.).
- Une exception se lance grâce au mot-clé `throw`.
- Il est possible d'hériter des exceptions entre elles.
- Il existe un certain nombre d'exceptions déjà disponibles dont il ne faut pas hésiter à se servir pour respecter la logique du code.

Les traits

Depuis sa version 5.4, PHP intègre un moyen de réutiliser le code d'une méthode dans deux classes indépendantes. Cette fonctionnalité permet ainsi de repousser les limites de l'héritage simple (pour rappel, en PHP, une classe ne peut hériter que d'une et une seule classe mère). Nous allons donc nous pencher ici sur les **traits** afin de pallier le problème de duplication de méthode.

Le principe des traits

Pour suivre ce chapitre, il vous faut PHP 5.4 d'installé sur votre serveur. Si vous tournez sous Ubuntu avec un serveur LAMP et que les dépôts officiels ne sont pas encore mis à jour, je vous invite à visiter cette page.

Posons le problème

Admettons que vous ayez deux classes, `Writer` et `Mailer`. La première est chargée d'écrire du texte dans un fichier, tandis que la seconde envoie un texte par mail. Cependant, il est agréable de mettre en forme le texte. Pour cela, vous décidez de **formater** le texte en HTML. Or, un problème se pose : vous allez devoir effectuer la même opération (celle de formater en HTML) dans deux classes complètement différentes et indépendantes :

Code : PHP

```php
<?php
class Writer
{
  public function write($text)
  {
    $text = '<p>Date : '.date('d/m/Y').'</p>'."\n".
            '<p>'.nl2br($text).'</p>';
    file_put_contents('fichier.txt', $text);
  }
}
```

Code : PHP

```php
<?php
class Mailer
{
  public function send($text)
  {
    $text = '<p>Date : '.date('d/m/Y').'</p>'."\n".
            '<p>'.nl2br($text).'</p>';
    mail('login@fai.tld', 'Test avec les traits', $text);
  }
}
```

Ici, le code est petit et la duplication n'est donc pas énorme, mais elle est belle et bien présente. Dans une application de plus grande envergure, ce code pourrait être dupliqué pas mal de fois. Imaginez alors que vous décidiez de formater autrement votre texte : catastrophe, il va falloir modifier chaque partie du code qui formatait du texte ! Penchons-nous donc vers les **traits** pour résoudre ce problème.

Résoudre le problème grâce aux traits

Syntaxe de base

Comme vous vous en doutez peut-être, les traits sont un moyen **d'externaliser** du code. Plus précisément, les traits définissent des méthodes que les classes peuvent utiliser. Avant de résoudre le problème évoqué, nous allons nous pencher sur la syntaxe des traits pour pouvoir s'en servir.

Regardez ce code :

Code : PHP

```php
<?php
trait MonTrait
{
  public function hello()
  {
    echo 'Hello world !';
  }
}

class A
{
  use MonTrait;
}

class B
{
  use MonTrait;
}

$a = new A;
$a->hello(); // Affiche « Hello world ! ».

$b = new b;
$b->hello(); // Affiche aussi « Hello world ! ».
```

Commentons ce code. Dans un premier temps, nous définissons un **trait**. Un trait, comme vous pouvez le constater, n'est autre qu'une mini-classe. Dedans, nous n'avons déclaré qu'une seule méthode. Ensuite, nous déclarons deux classes, chacune utilisant le trait que nous avons créé. Comme vous pouvez le constater, l'utilisation d'un trait dans une classe se fait grâce au mot-clé `use`. En utilisant ce mot-clé, toutes les méthodes du trait vont être **importées** dans la classe. Comme en témoigne le code de test en fin de fichier, les deux classes possèdent bien une méthode `hello()` et celle-ci affiche bien *« Hello world ! »*.

Retour sur notre formateur

Ce que l'on va faire maintenant, c'est retirer le gros défaut de notre code : grâce aux traits, nous ferons disparaître la duplication de notre code. Ce que je vous propose de faire, c'est de créer un trait qui va contenir une méthode `format($text)` qui va formater le texte passé en argument en HTML.

Procédons étape par étape. Commençons par créer notre trait (il ne contient qu'une méthode chargée de formater le texte en HTML) :

Code : PHP

```php
<?php
trait HTMLFormater
{
  public function format($text)
  {
    return '<p>Date : '.date('d/m/Y').'</p>'."\n".
           '<p>'.nl2br($text).'</p>';
  }
}
```

Maintenant, nous allons modifier nos classes `Writer` et `Mailer` afin qu'elles utilisent ce trait pour formater le texte qu'elles exploiteront. Pour y arriver, je vous rappelle qu'il vous faut utiliser le mot-clé `use` pour **importer** toutes les méthodes du trait (ici, il n'y en a qu'une) dans la classe. Vous pourrez ainsi utiliser les méthodes comme si elles étaient déclarées dans la classe. Voici la correction :

Code : PHP

```php
<?php
class Writer
{
```

```
  use HTMLFormater;

  public function write($text)
  {
    file_put_contents('fichier.txt', $this->format($text));
  }
}
```

Code : PHP

```
<?php
class Mailer
{
  use HTMLFormater;

  public function send($text)
  {
    mail('login@fai.tld', 'Test avec les traits', $this-
>format($text));
  }
}
```

Libre à vous de tester vos classes ! Par exemple, essayez d'exécuter ce code :

Code : PHP

```
<?php
$w = new Writer;
$w->write('Hello world!');

$m = new Mailer;
$m->send('Hello world!');
```

Nous venons ici de supprimer la duplication de code anciennement présente. Cependant, les traits ne se résument pas qu'à ça ! Regardons par exemple comment utiliser plusieurs traits dans une classe.

Utiliser plusieurs traits

Syntaxe

Pour utiliser plusieurs traits, rien de plus simple : il vous suffit de lister tous les traits à utiliser séparés par des virgules, comme ceci :

Code : PHP

```
<?php
trait HTMLFormater
{
  public function formatHTML($text)
  {
    return '<p>Date : '.date('d/m/Y').'</p>'."\n".
           '<p>'.nl2br($text).'</p>';
  }
}

trait TextFormater
{
  public function formatText($text)
  {
```

```
        return 'Date : '.date('d/m/Y')."\n".$text;
    }
}

class Writer
{
    use HTMLFormater, TextFormater;

    public function write($text)
    {
        file_put_contents('fichier.txt', $this->formatHTML($text));
    }
}
```

Je vous laisse essayer ce code, je suis sûr que vous y arriverez seuls !

Résolution des conflits

Le code donné plus haut est bien beau, mais que se passerait-il si nos traits avaient tous les deux une méthode nommée `format()` ? Je vous laisse essayer... Et oui, une erreur fatale avec le message *« Trait method format has not been applied, because there are collisions with other trait methods »* est levée. Pour pallier ce problème, nous pouvons donner une priorité à une méthode d'un trait afin de lui permettre d'écraser la méthode de l'autre trait si il y en une identique.

Par exemple, si dans notre classe `Writer` nous voulions formater notre message en HTML, nous pourrions faire :

Code : PHP

```
<?php
class Writer
{
    use HTMLFormater, TextFormater
    {
        HTMLFormater::format insteadof TextFormater;
    }

    public function write($text)
    {
        file_put_contents('fichier.txt', $this->format($text));
    }
}
```

Regardons de plus près cette ligne n°6. Pour commencer, notez qu'elle est définie dans une paire d'accolades suivant les noms des traits à utiliser. À l'intérieur de cette paire d'accolades se trouve la liste des « méthodes prioritaires ». Chaque déclaration de priorité se fait en se terminant par un point-virgule. Cette ligne signifie donc *: « La méthode `format()` du trait `HTMLFormater` écrasera la méthode du même nom du trait `TextFormater` (si elle y est définie). »*

Méthodes de traits vs. méthodes de classes

La classe plus forte que le trait

Terminons cette introduction aux traits en nous penchant sur les conflits entre méthodes de traits et méthodes de classes. Si une classe déclare une méthode et qu'elle utilise un trait possédant cette même méthode, alors la méthode déclarée dans la classe l'emportera sur la méthode déclarée dans le trait. Exemple :

Code : PHP

```
<?php
trait MonTrait
{
    public function sayHello()
    {
```

```
    echo 'Hello !';
  }
}

class MaClasse
{
  use MonTrait;

  public function sayHello()
  {
    echo 'Bonjour !';
  }
}

$objet = new MaClasse;
$objet->sayHello(); // Affiche « Bonjour ! ».
```

Le trait plus fort que la mère

À l'inverse, si une classe utilise un trait possédant une méthode déjà implémentée dans la classe mère de la classe utilisant le trait, alors ce sera la méthode du trait qui sera utilisée (la méthode du trait écrasera celle de la méthode de la classe mère). Exemple :

Code : PHP

```
<?php
trait MonTrait
{
  public function speak()
  {
    echo 'Je suis un trait !';
  }
}

class Mere
{
  public function speak()
  {
    echo 'Je suis une classe mère !';
  }
}

class Fille extends Mere
{
  use MonTrait;
}

$fille = new Fille;
$fille->speak();
```

Plus loin avec les traits

Définition d'attributs

Syntaxe

Nous avons vu que les traits servaient à isoler des méthodes afin de pouvoir les utiliser dans deux classes totalement indépendantes. Si le besoin s'en fait sentir, sachez que vous pouvez aussi définir des attributs dans votre trait. Ils seront alors à leur tour **importés** dans la classe qui utilisera ce trait. Exemple :

Code : PHP

```
<?php
trait MonTrait
```

```
{
  protected $attr = 'Hello !';

  public function showAttr()
  {
    echo $this->attr;
  }
}

class MaClasse
{
  use MonTrait;
}

$fille = new MaClasse;
$fille->showAttr();
```

À l'inverse d'une méthode, un attribut ne peut pas être `static` !

Conflit entre attributs

Si un attribut est défini dans un trait, alors la classe utilisant le trait ne peut pas définir d'attribut possédant le même nom. Suivant la déclaration de l'attribut, deux cas peuvent se présenter :

- Si l'attribut déclaré dans la classe a le même nom mais pas la même valeur initiale ou pas la même visibilité, une erreur fatale est levée.
- Si l'attribut déclaré dans la classe a le même nom, une valeur initiale identique et la même visibilité, une erreur stricte est levée (il est possible, suivant votre configuration, que PHP n'affiche pas ce genre d'erreur).

Malheureusement, il est impossible, comme nous l'avons fait avec les méthodes, de définir des attributs prioritaires. Veillez donc bien à ne pas utiliser ce genre de code :

Code : PHP

```
<?php
trait MonTrait
{
  protected $attr = 'Hello !';
}

class MaClasse
{
  use MonTrait;

  protected $attr = 'Hello !'; // Lèvera une erreur stricte.
  protected $attr = 'Bonjour !'; // Lèvera une erreur fatale.
  private $attr = 'Hello !'; // Lèvera une erreur fatale.
}
```

Traits composés d'autres traits

Au même titre que les classes, les traits peuvent eux aussi utiliser des traits. La façon de procéder est la même qu'avec les classes, tout comme la gestion des conflits entre méthodes. Voici un exemple :

Code : PHP

```
<?php
trait A
{
```

```
  public function saySomething()
  {
    echo 'Je suis le trait A !';
  }
}

trait B
{
  use A;

  public function saySomethingElse()
  {
    echo 'Je suis le trait B !';
  }
}

class MaClasse
{
  use B;
}

$o = new MaClasse;
$o->saySomething();
$o->saySomethingElse();
```

Changer la visibilité et le nom des méthodes

Si un trait implémente une méthode, toute classe utilisant ce trait a la capacité de changer sa visibilité, c'est-à-dire la passer en privé, protégé ou public. Pour cela, nous allons à nouveau se servir des accolades qui ont suivi la déclaration de `use` pour y glisser une instruction. Cette instruction fait appel à l'opérateur `as`, que vous avez déjà peut-être rencontré dans l'utilisation de *namespaces* (si ce n'est pas le cas, ce n'est pas grave du tout). Le rôle est ici le même : créer un alias. En effet, vous pouvez aussi **changer** le nom des méthodes. Dans ce dernier cas, la méthode ne sera pas renommée, mais copiée sous un autre nom, ce qui signifie que vous pourrez toujours y accéder sous son ancien nom.

Quelques petits exemples pour que vous compreniez :

Code : PHP

```
<?php
trait A
{
  public function saySomething()
  {
    echo 'Je suis le trait A !';
  }
}

class MaClasse
{
  use A
  {
    saySomething as protected;
  }
}

$o = new MaClasse;
$o->saySomething(); // Lèvera une erreur fatale car on tente
d'accéder à une méthode protégée.
```

Code : PHP

```
<?php
trait A
```

```
{
  public function saySomething()
  {
    echo 'Je suis le trait A !';
  }
}

class MaClasse
{
  use A
  {
    saySomething as sayWhoYouAre;
  }
}

$o = new MaClasse;
$o->sayWhoYouAre(); // Affichera « Je suis le trait A ! »
$o->saySomething(); // Affichera « Je suis le trait A ! »
```

Code : PHP

```
<?php
trait A
{
  public function saySomething()
  {
    echo 'Je suis le trait A !';
  }
}

class MaClasse
{
  use A
  {
    saySomething as protected sayWhoYouAre;
  }
}

$o = new MaClasse;
$o->saySomething(); // Affichera « Je suis le trait A ! ».
$o->sayWhoYouAre(); // Lèvera une erreur fatale, car l'alias créé
est une méthode protégée.
```

Méthodes abstraites dans les traits

Enfin, sachez que l'on peut forcer la classe utilisant le trait à implémenter certaines méthodes au moyen de méthodes abstraites. Ainsi, ce code lèvera une erreur fatale :

Code : PHP

```
<?php
trait A
{
  abstract public function saySomething();
}

class MaClasse
{
  use A;
}
```

Cependant, si la classe utilisant le trait déclarant une méthode abstraite est elle aussi abstraite, alors ce sera à ses classes filles d'implémenter les méthodes abstraites du trait (elle peut le faire, mais elle n'est pas obligée). Exemple :

Code : PHP

```php
<?php
trait A
{
  abstract public function saySomething();
}

abstract class Mere
{
  use A;
}

// Jusque-là, aucune erreur n'est levée.

class Fille extends Mere
{
  // Par contre, une erreur fatale est ici levée, car la méthode saySomething() n'a pas été implémentée.
}
```

En résumé

- Les traits sont un moyen pour éviter la duplication de méthodes.
- Un trait s'utilise grâce au mot-clé `use`.
- Il est possible d'utiliser une infinité de traits dans une classe en résolvant les conflits éventuels avec `insteadof`.
- Un trait peut lui-même utiliser un autre trait.
- Il est possible de changer la visibilité d'une méthode ainsi que son nom grâce au mot-clé `as`.

L'API de réflexivité

En tant que développeur, vous savez que telle classe hérite de telle autre ou que tel attribut est protégé par exemple. Cependant, votre script PHP, lui ne le sait pas : comment feriez-vous pour afficher à l'écran par exemple que telle classe hérite de telle autre (dynamiquement bien entendu) ? Cela est pour vous impossible. Je vais cependant vous dévoiler la solution : pour y parvenir, on se sert de l'API de réflexivité qui va justement nous permettre d'obtenir des informations sur nos classes, attributs et méthodes.

Une fois que nos aurons vu tout cela, nous allons nous pencher sur un exemple d'utilisation. Pour cela, nous utiliserons une bibliothèque qui se sert de l'API de réflexivité pour exploiter les **annotations**, terme sans doute inconnu en programmation pour la plupart d'entre vous.

Obtenir des informations sur ses classes

Qui dit « Réflexivité » dit « Instanciation de classe ». Nous allons donc instancier une classe qui nous fournira des informations sur telle classe. Dans cette section, il s'agit de la classe `ReflectionClass`. Quand nous l'instancierons, nous devrons spécifier le nom de la classe sur laquelle on veut obtenir des informations. Nous allons prendre pour exemple la classe `Magicien` de notre précédent TP.

Pour obtenir des informations la concernant, nous allons procéder comme suit :

Code : PHP

```php
<?php
$classeMagicien = new ReflectionClass('Magicien'); // Le nom de la classe doit être entre apostrophes ou guillemets.
?>
```

La classe `ReflectionClass` possède beaucoup de méthodes et je ne pourrai, par conséquent, toutes vous les présenter.

Il est également possible d'obtenir des informations sur une classe grâce à un objet. Nous allons pour cela instancier la classe `ReflectionObject` en fournissant l'instance en guise d'argument. Cette classe hérite de toutes les méthodes de `ReflectionClass` : elle ne réécrit que deux méthodes (dont le constructeur). La seconde méthode réécrite ne nous intéresse pas. Cette classe n'implémente pas de nouvelles méthodes.

Exemple d'utilisation très simple :

Code : PHP

```php
<?php
$magicien = new Magicien(array('nom' => 'vyk12', 'type' => 'magicien'));
$classeMagicien = new ReflectionObject($magicien);
?>
```

Informations propres à la classe

Les attributs

Pour savoir si la classe possède tel attribut, tournons-nous vers `ReflectionClass::hasProperty($atrributeName)`. Cette méthode retourne vrai si l'attribut passé en paramètre existe et faux s'il n'existe pas. Exemple :

Code : PHP

```php
<?php
if ($classeMagicien->hasProperty('magie'))
```

```
{
  echo 'La classe Magicien possède un attribut $magie';
}
else
{
  echo 'La classe Magicien ne possède pas d\'attribut $magie';
}
?>
```

Vous pouvez aussi récupérer cet attribut afin d'obtenir des informations le concernant comme nous venons de faire jusqu'à maintenant avec notre classe. Nous verrons cela plus tard, une sous-partie entière sera dédiée à ce sujet.

Les méthodes

Si vous voulez savoir si la classe implémente telle méthode, alors il va falloir regarder du côté de `ReflectionClass::hasMethod($methodName)`. Celle-ci retourne vrai si la méthode est implémentée ou faux si elle ne l'est pas. Exemple :

Code : PHP

```
<?php
if ($classeMagicien->hasMethod('lancerUnSort'))
{
  echo 'La classe Magicien implémente une méthode lancerUnSort()';
}
else
{
  echo 'La classe Magicien n\'implémente pas de méthode
lancerUnSort()';
}
?>
```

Vous pouvez, comme pour les attributs, récupérer une méthode, et une sous-partie sera consacrée à la classe permettant d'obtenir des informations sur telle méthode.

Les constantes

Dans ce cas également, il est possible de savoir si telle classe possède telle constante. Ceci grâce à la méthode `ReflectionClass::hasConstant($constName)`. Exemple :

Code : PHP

```
<?php
if ($classePersonnage->hasConstant('NOUVEAU'))
{
  echo 'La classe Personnage possède une constante NOUVEAU';
}
else
{
  echo 'La classe Personnage ne possède pas de constante NOUVEAU';
}
?>
```

Vous pouvez aussi récupérer la valeur de la constante grâce à `ReflectionClass::getConstant($constName)` :

Code : PHP

```
<?php
```

```
if ($classePersonnage->hasConstant('NOUVEAU'))
{
  echo 'La classe Personnage possède une constante NOUVEAU (celle ci
vaut ', $classePersonnage->getConstant('NOUVEAU'), ')';
}
else
{
  echo 'La classe Personnage ne possède pas de constante NOUVEAU';
}
?>
```

Nous pouvons également retrouver la liste complète des constantes d'une classe sous forme de tableau grâce à `ReflectionClass::getConstants()` :

Code : PHP

```
<?php
echo '<pre>', print_r($classePersonnage->getConstants(), true),
'</pre>';
?>
```

Les relations entre classes

L'héritage

Pour récupérer la classe parente de notre classe, nous allons regarder du côté de `ReflectionClass::getParentClass()`. Cette méthode nous renvoie la classe parente s'il y en a une : la valeur de retour sera une instance de la classe `ReflectionClass` qui représentera la classe parente ! Si la classe ne possède pas de parent, alors la valeur de retour sera `false`.

Exemple :

Code : PHP

```
<?php
$classeMagicien = new ReflectionClass('Magicien');

if ($parent = $classeMagicien->getParentClass())
{
  echo 'La classe Magicien a un parent : il s\'agit de la classe ',
$parent->getName();
}
else
{
  echo 'La classe Magicien n\'a pas de parent';
}
?>
```

Voici une belle occasion de vous présenter la méthode `ReflectionClass::getName()`. Cette méthode se contente de renvoyer le nom de la classe. Pour l'exemple avec le magicien, cela aurait été inutile puisque nous connaissions déjà le nom de la classe, mais ici nous ne le connaissons pas (quand je dis que nous ne le connaissons pas, cela signifie qu'il n'est pas déclaré explicitement dans les dernières lignes de code).

Dans le domaine de l'héritage, nous pouvons également citer `ReflectionClass::isSubclassOf($className)`. Cette méthode nous renvoie vrai si la classe spécifiée en paramètre est le parent de notre classe. Exemple :

Code : PHP

```php
<?php
if ($classeMagicien->isSubclassOf('Personnage'))
{
  echo 'La classe Magicien a pour parent la classe Personnage';
}
else
{
  echo 'La classe Magicien n\'a la classe Personnage pour parent';
}
?>
```

Les deux prochaines méthodes que je vais vous présenter ne sont pas en rapport direct avec l'héritage mais sont cependant utilisées lorsque cette relation existe : il s'agit de savoir si la classe est abstraite ou finale. Nous avons pour cela les méthodes `ReflectionClass::isAbstract()` et `ReflectionClass::isFinal()`. Notre classe `Personnage` est abstraite, vérifions donc cela :

Code : PHP

```php
<?php
$classePersonnage = new ReflectionClass('Personnage');

// Est-elle abstraite ?
if ($classePersonnage->isAbstract())
{
  echo 'La classe Personnage est abstraite';
}
else
{
  echo 'La classe Personnage n\'est pas abstraite';
}

// Est-elle finale ?
if ($classePersonnage->isFinal())
{
  echo 'La classe Personnage est finale';
}
else
{
  echo 'La classe Personnage n\'est pas finale';
}
?>
```

Dans le même genre, `ReflectionClass::isInstantiable()` permet également de savoir si notre classe est instanciable. Comme la classe `Personnage` est abstraite, elle ne peut pas l'être. Vérifions cela :

Code : PHP

```php
<?php
if ($classePersonnage->isInstantiable())
{
  echo 'La classe Personnage est instanciable';
}
else
{
  echo 'La classe personnage n\'est pas instanciable';
}
?>
```

Bref, pas de grosse surprise. 

Les interfaces

Voyons maintenant les méthodes en rapport avec les interfaces. Comme je vous l'ai déjà dit, une interface n'est autre qu'une classe entièrement abstraite : nous pouvons donc instancier la classe `ReflectionClass` en spécifiant une interface en paramètre et vérifier si celle-ci est bien une interface grâce à la méthode `ReflectionClass::isInterface()`.

Dans les exemples qui vont suivre, nous allons admettre que la classe `Magicien` implémente une interface `iMagicien`.

Code : PHP

```
<?php
$classeIMagicien = new ReflectionClass('iMagicien');

if ($classeIMagicien->isInterface())
{
   echo 'La classe iMagicien est une interface';
}
else
{
  echo 'La classe iMagicien n\'est pas une interface';
}
?>
```

Vous pouvez aussi savoir si telle classe implémente telle interface grâce à la méthode `ReflectionClass::implementsInterface($interfaceName)`. Exemple :

Code : PHP

```
<?php
if ($classeMagicien->implementsInterface('iMagicien'))
{
  echo 'La classe Magicien implémente l\'interface iMagicien';
}
else
{
  echo 'La classe Magicien n\'implémente pas l\'interface
iMagicien';
}
?>
```

Il est aussi possible de récupérer toutes les interfaces implémentées, interfaces contenues dans un tableau. Pour cela, deux méthodes sont à votre disposition : `ReflectionClass::getInterfaces()` et `ReflectionClass::getInterfaceNames()`. La première renvoie autant d'instances de la classe `ReflectionClass` qu'il y a d'interfaces, chacune représentant une interface. La seconde méthode se contente uniquement de renvoyer un tableau contenant le nom de toutes les interfaces implémentées. Je pense qu'il est inutile de donner un exemple sur ce point-là. 

Obtenir des informations sur les attributs de ses classes

Nous avons assez parlé de la classe, intéressons-nous maintenant à ses attributs. La classe qui va nous permettre d'en savoir plus à leur sujet est `ReflectionProperty`. Il y a deux moyens d'utiliser cette classe : l'instancier directement ou utiliser une méthode de `ReflectionClass` qui nous renverra une instance de `ReflectionProperty`.

Instanciation directe

L'appel du constructeur se fait en lui passant deux arguments. Le premier est le nom de la classe, et le second est le nom de l'attribut. Exemple :

Code : PHP

```php
<?php
$attributMagie = new ReflectionProperty('Magicien', 'magie');
?>
```

Tout simplement.

Récupération d'attribut d'une classe

Récupérer un attribut

Pour récupérer un attribut d'une classe, nous aurons besoin de la méthode `ReflectionClass::getProperty($attrName)` :

Code : PHP

```php
<?php
$classeMagicien = new ReflectionClass('Magicien');
$attributMagie = $classeMagicien->getProperty('magie');
?>
```

Récupérer tous les attributs

Si vous souhaitez récupérer tous les attributs d'une classe, il va falloir se servir de `ReflectionClass::getProperties()`. Le résultat retourné est un tableau contenant autant d'instances de `ReflectionProperty` que d'attributs.

Code : PHP

```php
<?php
$classePersonnage = new ReflectionClass('Personnage');
$attributsPersonnage = $classeMagicien->getProperties();
?>
```

Après avoir vu comment récupérer un attribut, nous allons voir ce que l'on peut faire avec.

Le nom et la valeur des attributs

Afin de récupérer le nom de l'attribut, nous avons toujours la méthode `ReflectionProperty::getName()`. Pour obtenir la valeur de celui-ci, nous utiliserons la méthode `ReflectionProperty::getValue($object)`. Nous devrons spécifier à cette dernière méthode l'instance dans laquelle nous voulons obtenir la valeur de l'attribut : chaque attribut est propre à chaque instance, ça n'aurait pas de sens de demander la valeur de l'attribut d'une classe.

Pour nous exercer, nous allons lister tous les attributs de la classe `Magicien`.

Code : PHP

```php
<?php
$classeMagicien = new ReflectionClass('Magicien');
$magicien = new Magicien(array('nom' => 'vyk12', 'type' =>
'magicien'));

foreach ($classeMagicien->getProperties() as $attribut)
{
  echo $attribut->getName(), ' => ', $attribut->getValue($magicien);
}
```

```
?>
```

Il fonctionne pas ton code, j'ai une vieille erreur fatale qui me bloque tout...

En effet. Cette erreur fatale est levée car vous avez appelé la méthode `ReflectionProperty::getValue()` sur un attribut **non public**. Il faut donc rendre l'attribut **accessible** grâce à `ReflectionProperty::setAccessible($accessible)`, où `$accessible` vaut vrai ou faux selon si vous voulez rendre l'attribut accessible ou non.

Code : PHP

```php
<?php
$classeMagicien = new ReflectionClass('Magicien');
$magicien = new Magicien(array('nom' => 'vyk12', 'type' =>
'magicien'));

foreach ($classeMagicien->getProperties() as $attribut)
{
  $attribut->setAccessible(true);
  echo $attribut->getName(), ' => ', $attribut->getValue($magicien);
}
?>
```

Quand vous rendez un attribut accessible, vous pouvez modifier sa valeur grâce à `ReflectionProperty::setValue($objet, $valeur)`, ce qui est contre le principe d'encapsulation. Pensez donc bien à rendre l'attribut inaccessible après sa lecture en faisant `$attribut->setAccessible(false)`.

Portée de l'attribut

Il est tout à fait possible de savoir si un attribut est privé, protégé ou public grâce aux méthodes `ReflectionProperty::isPrivate()`, `ReflectionProperty::isProtected()` et `ReflectionProperty::isPublic()`.

Code : PHP

```php
<?php
$uneClasse = new ReflectionClass('MaClasse');

foreach ($uneClasse->getProperties() as $attribut)
{
  echo $attribut->getName(), ' => attribut ';

  if ($attribut->isPublic())
  {
    echo 'public';
  }
  elseif ($attribut->isProtected())
  {
    echo 'protégé';
  }
  else
  {
    echo 'privé';
  }
}
?>
```

Il existe aussi une méthode permettant de savoir si l'attribut est statique ou non grâce à `ReflectionProperty::isStatic()`.

Code : PHP

```php
<?php
$uneClasse = new ReflectionClass('MaClasse');

foreach ($uneClasse->getProperties() as $attribut)
{
  echo $attribut->getName(), ' => attribut ';

  if ($attribut->isPublic())
  {
    echo 'public';
  }
  elseif ($attribut->isProtected())
  {
    echo 'protégé';
  }
  else
  {
    echo 'privé';
  }

  if ($attribut->isStatic())
  {
    echo ' (attribut statique)';
  }
}
?>
```

Les attributs statiques

Le traitement d'attributs statiques diffère un peu dans le sens où ce n'est pas un attribut d'une **instance** mais un attribut de la **classe**. Ainsi, vous n'êtes pas obligés de spécifier d'instance lors de l'appel de `ReflectionProperty::getValue()` car un attribut statique n'appartient à aucune instance.

Code : PHP

```php
<?php
class A
{
  public static $attr = 'Hello world !';
}

$classeA = new ReflectionClass('A');
echo $classeA->getProperty('attr')->getValue();
?>
```

Au lieu d'utiliser cette façon de faire, vous pouvez directement appeler `ReflectionClass::getStaticPropertyValue($attr)`, où `$attr` est le nom de l'attribut. Dans le même genre, on peut citer `ReflectionClass::setStaticPropertyValue($attr, $value)` où `$value` est la nouvelle valeur de l'attribut.

Code : PHP

```php
<?php
```

```php
class A
{
  public static $attr = 'Hello world !';
}

$classeA = new ReflectionClass('A');
echo $classeA->getStaticPropertyValue('attr'); // Affiche Hello
world !

$classeA->setStaticPropertyValue('attr', 'Bonjour le monde !');
echo $classeA->getStaticPropertyValue('attr'); // Affiche Bonjour le
monde !
?>
```

Vous avez aussi la possibilité d'obtenir tous les attributs statiques grâce à `ReflectionClass::getStaticProperties()`. Le tableau retourné ne contient pas des instances de `ReflectionProperty` mais uniquement les valeurs de chaque attribut.

Code : PHP

```php
<?php
class A
{
  public static $attr1 = 'Hello world !';
  public static $attr2 = 'Bonjour le monde !';
}

$classeA = new ReflectionClass('A');

foreach ($classeA->getStaticProperties() as $attr)
{
  echo $attr;
}

// À l'écran s'affichera Hello world ! Bonjour le monde !
?>
```

Obtenir des informations sur les méthodes de ses classes

Voici la dernière classe faisant partie de l'API de réflexivité que je vais vous présenter : il s'agit de `ReflectionMethod`. Comme vous l'aurez deviné, c'est grâce à celle-ci que l'on pourra obtenir des informations concernant telle ou telle méthode. Nous pourrons connaître la portée de la méthode (publique, protégée ou privée), si elle est statique ou non, abstraite ou finale, s'il s'agit du constructeur ou du destructeur et on pourra même l'appeler sur un objet. 

Création d'une instance de `ReflectionMethod`

Instanciation directe

Le constructeur de `ReflectionMethod` demande deux arguments : le nom de la classe et le nom de la méthode. Exemple :

Code : PHP

```php
<?php
class A
{
  public function hello($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello
world !')
  {
    echo 'Hello world !';
  }
}

$methode = new ReflectionMethod('A', 'hello');
?>
```

Récupération d'une méthode d'une classe

La seconde façon de procéder est de récupérer la méthode de la classe grâce à `ReflectionClass::getMethod($name)`. Celle-ci renvoie une instance de `ReflectionClass` représentant la méthode.

Code : PHP

```php
<?php
class A
{
  public function hello($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello world !')
  {
    echo 'Hello world !';
  }
}

$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');
?>
```

Publique, protégée ou privée ?

Comme pour les attributs, nous avons des méthodes pour le savoir : j'ai nommé `ReflectionMethod::isPublic()`, `ReflectionMethod::isProtected()` et `ReflectionMethod::isPrivate()`. Je ne vais pas m'étendre sur le sujet, vous savez déjà vous en servir.

Code : PHP

```php
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

echo 'La méthode ', $methode->getName(), ' est ';

if ($methode->isPublic())
{
  echo 'publique';
}
elseif ($methode->isProtected())
{
  echo 'protégée';
}
else
{
  echo 'privée';
}
?>
```

Je suis sûr que vous savez quelle méthode permet de savoir si elle est statique ou non.

Code : PHP

```php
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');
```

```
echo 'La méthode ', $methode->getName(), ' est ';

if ($methode->isPublic())
{
  echo 'publique';
}
elseif ($methode->isProtected())
{
  echo 'protégée';
}
else
{
  echo 'privée';
}

if ($methode->isStatic())
{
  echo ' (en plus elle est statique)';
}
?>
```

Abstraite ? Finale ?

Les méthodes permettant de savoir si une méthode est abstraite ou finale sont très simples à retenir : il s'agit de `ReflectionMethod::isAbstract()` et `ReflectionMethod::isFinal()`.

Code : PHP

```
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

echo 'La méthode ', $methode->getName(), ' est ';

if ($methode->isAbstract())
{
  echo 'abstraite';
}
elseif ($methode->isFinal())
{
  echo 'finale';
}
else
{
  echo '« normale »';
}
?>
```

Constructeur ? Destructeur ?

Dans le même genre, `ReflectionMethod::isConstructor()` et `ReflectionMethod::isDestructor()` permettent de savoir si la méthode est le constructeur ou le destructeur de la classe.

Code : PHP

```
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

if ($methode->isConstructor())
{
```

```
  echo 'La méthode ', $methode->getName(), ' est le constructeur';
}
elseif ($methode->isDestructor())
{
  echo 'La méthode ', $methode->getName(), ' est le destructeur';
}
?>
```

Pour que la première condition renvoie vrai, il ne faut pas obligatoirement que la méthode soit nommée `__construct`. En effet, si la méthode a le même nom que la classe, celle-ci est considérée comme le constructeur de la classe car, sous PHP 4, c'était de cette façon que l'on implémentait le constructeur : il n'y avait jamais de `__construct`. Pour que les scripts développés sous PHP 4 soient aussi compatibles sous PHP 5, le constructeur peut également être implémenté de cette manière, mais il est clairement préférable d'utiliser la méthode magique créée pour cet effet.

Appeler la méthode sur un objet

Pour réaliser ce genre de chose, nous allons avoir besoin de `ReflectionMethod::invoke($object, $args)`. Le premier argument est l'objet sur lequel on veut appeler la méthode. Viennent ensuite tous les arguments que vous voulez passer à la méthode : vous devrez donc passer autant d'arguments que la méthode appelée en exige. Prenons un exemple tout simple, vous comprendrez mieux :

Code : PHP

```
<?php
class A
{
  public function hello($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello
world !')
  {
    var_dump($arg1, $arg2, $arg3, $arg4);
  }
}

$a = new A;
$hello = new ReflectionMethod('A', 'hello');

$hello->invoke($a, 'test', 'autre test'); // On ne va passer que
deux arguments à notre méthode.

// A l'écran s'affichera donc :
// string(4) "test" string(10) "autre test" int(1) string(13) "Hello
world !"
?>
```

Une méthode semblable à `ReflectionMethod::invoke($object, $args)` existe : il s'agit de `ReflectionMethod::invokeArgs($object, $args)`. La différence entre ces deux méthodes est que la seconde demandera les arguments listés dans un tableau au lieu de les lister en paramètres. L'équivalent du code précédent avec `Reflection::invokeArgs()` serait donc le suivant :

Code : PHP

```
<?php
class A
{
  public function hello($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello
world !')
  {
    var_dump($arg1, $arg2, $arg3, $arg4);
```

```
    }
}

$a = new A;
$hello = new ReflectionMethod('A', 'hello');

$hello->invokeArgs($a, array('test', 'autre test')); // Les deux
arguments sont cette fois-ci contenus dans un tableau.

// Le résultat affiché est exactement le même.
?>
```

Si vous n'avez pas accès à la méthode à cause de sa portée restreinte, vous pouvez la rendre accessible comme on l'a fait avec les attributs, grâce à la méthode `ReflectionMethod::setAccessible($bool)`. Si `$bool` vaut `true`, alors la méthode sera accessible, sinon elle ne le sera pas. Je me passe d'exemple, je suis sûr que vous trouverez tous seuls.

Utiliser des annotations

À présent, je vais vous montrer quelque chose d'intéressant qu'il est possible de faire grâce à la réflexivité : utiliser des annotations pour vos classes, méthodes et attributs, mais surtout y accéder durant l'exécution du script. Que sont les annotations ? Les annotations sont des méta-données relatives à la classe, méthode ou attribut, qui apportent des informations sur l'entité souhaitée. Elles sont insérées dans des commentaires utilisant le syntaxe **doc block**, comme ceci :

Code : PHP

```
<?php
/**
* @version 2.0
*/
class Personnage
{
  // ...
}
```

Les annotations s'insèrent à peu près de la même façon, mais la syntaxe est un peu différente. En effet, la syntaxe doit être précise pour qu'elle puisse être parsée par la bibliothèque que nous allons utiliser pour récupérer les données souhaitées.

Présentation d'addendum

Cette section aura pour but de présenter les annotations par le biais de la bibliothèque **addendum** qui parsera les codes pour en extraire les informations. Pour cela, commencez par télécharger addendum, et décompressez l'archive dans le dossier contenant votre projet.

Commençons par créer une classe sur laquelle nous allons travailler tout au long de cette partie, comme `Personnage` (à tout hasard). Avec addendum, toutes les annotations sont des classes héritant d'une classe de base : `Annotation`. Si nous voulons ajouter une annotation, `Table` par exemple, à notre classe pour spécifier à quelle table un objet `Personnage` correspond, alors il faudra au préalable créer une classe `Table`.

Pour travailler simplement, créez sur votre ordinateur un dossier **annotations** contenant un fichier **index.php**, un fichier **Personnage.class.php** qui contiendra notre classe, un fichier **MyAnnotations.php** qui contiendra nos annotations, et enfin le dossier **addendum.**

Code : PHP

```
<?php
class Table extends Annotation {}
```

À toute annotation correspond une **valeur**, valeur à spécifier lors de la déclaration de l'annotation :

Code : PHP

```php
<?php
/**
* @Table("personnages")
*/
class Personnage
{

}
```

Nous venons donc de créer une annotation basique, mais concrètement, nous n'avons pas fait grand-chose. Nous allons maintenant voir comment récupérer cette annotation, et plus précisément la valeur qui lui est assignée, grâce à addendum.

À quoi servent-elles ces annotations ?

Les annotations sont surtout utilisées par les frameworks, comme PHPUnit (framework de tests unitaires) ou Zend Framework par exemple, ou bien les ORM tel que Doctrine, qui apportent ici des informations pour le mapping des classes. Vous n'aurez donc peut-être pas à utiliser les annotations dans vos scripts, mais il est important d'en avoir entendu parler si vous décidez d'utiliser des frameworks ou bibliothèques les utilisant.

Récupérer une annotation

Pour récupérer une annotation, il va d'abord falloir récupérer la classe *via* la bibliothèque en créant une instance de `ReflectionAnnotatedClass`, comme nous l'avions fait en début de chapitre avec `ReflectionClass` :

Code : PHP

```php
<?php
// On commence par inclure les fichiers nécessaires.
require 'addendum/annotations.php';
require 'MyAnnotations.php';
require 'Personnage.class.php';

$reflectedClass = new ReflectionAnnotatedClass('Personnage');
```

Maintenant que c'est fait, nous allons pouvoir récupérer l'annotation grâce à la méthode `getAnnotation`. De manière générale, cette méthode retourne une instance de `Annotation`. Dans notre cas, puisque nous voulons l'annotation `Table`, ce sera une instance de `Table` qui sera retournée. La valeur de l'annotation est contenue dans l'attribut `value`, attribut public disponible dans toutes les classes filles de `Annotation` :

Code : PHP

```php
<?php
// On commence par inclure les fichiers nécessaires.
require 'addendum/annotations.php';
require 'MyAnnotations.php';
require 'Personnage.class.php';

$reflectedClass = new ReflectionAnnotatedClass('Personnage');

echo 'La valeur de l\'annotation <strong>Table</strong> est
<strong>', $reflectedClass->getAnnotation('Table')->value,
'</strong>';
```

Il est aussi possible, pour une annotation, d'avoir un tableau comme valeur. Pour réaliser ceci, il faut mettre la valeur de l'annotation entre accolades et séparer les valeurs du tableau par des virgules :

Code : PHP

```php
<?php
/**
* @Type({'brute', 'guerrier', 'magicien'})
*/
class Personnage
{

}
```

Si vous récupérez l'annotation, vous obtiendrez un tableau classique :

Code : PHP

```php
<?php
print_r($reflectedClass->getAnnotation('Type')->value); // Affiche
le détail du tableau.
```

Vous pouvez aussi spécifier des clés pour les valeurs comme ceci :

Code : PHP

```php
<?php
/**
* @Type({meilleur = 'magicien', 'moins bon' = 'brute', neutre =
'guerrier'})
*/
class Personnage
{

}
```

Notez la mise entre quotes de **moins bon** : elles sont utiles ici car un espace est présent. Cependant, comme vous le voyez avec **meilleur** et **neutre**, elles ne sont pas obligatoires. 

Enfin, pour finir avec les tableaux, je précise que vous pouvez en emboîter tant que vous voulez. Pour placer un tableau dans un autre, il suffit d'ouvrir une nouvelle paire d'accolades :

Code : PHP

```php
<?php
/**
* @UneAnnotation({uneCle = 1337, {uneCle2 = true, uneCle3 = 'une
valeur'}})
*/
```

Savoir si une classe possède telle annotation

Il est possible de savoir si une classe possède telle annotation grâce à la méthode `hasAnnotation` :

Code : PHP

```php
<?php
require 'addendum/annotations.php';
require 'MyAnnotations.php';
require 'Personnage.class.php';

$reflectedClass = new ReflectionAnnotatedClass('Personnage');

$ann = 'Table';
if ($reflectedClass->hasAnnotation($ann))
{
  echo 'La classe possède une annotation <strong>', $ann, '</strong>
dont la valeur est <strong>', $reflectedClass->getAnnotation($ann)-
>value, '</strong><br />';
}
```

Une annotation à multiples valeurs

Il est possible pour une annotation de posséder plusieurs valeurs. Chacune de ces valeurs est stockée dans un attribut de la classe représentant l'annotation. Par défaut, une annotation ne contient qu'un attribut (`$value`) qui est la valeur de l'annotation.

Pour pouvoir assigner plusieurs valeurs à une annotation, il va donc falloir ajouter des attributs à notre classe. Commençons par ça :

Code : PHP

```php
<?php
class ClassInfos extends Annotation
{
  public $author;
  public $version;
}
```

Il est important ici que les attributs soient publics pour que le code extérieur à la classe puisse modifier leur valeur.

Maintenant, tout se joue lors de la création de l'annotation. Pour assigner les valeurs souhaitées aux attributs, il suffit d'écrire ces valeurs précédées du nom de l'attribut. Exemple :

Code : PHP

```php
<?php
/**
* @ClassInfos(author = "vyk12", version = "1.0")
*/
class Personnage
{

}
```

Pour accéder aux valeurs des attributs, il faut récupérer l'annotation, comme nous l'avons fait précédemment, et récupérer l'attribut.

Code : PHP

```php
<?php
```

```
$classInfos = $reflectedClass->getAnnotation('ClassInfos');

echo $classInfos->author;
echo $classInfos->version;
```

Le fait que les attributs soient publics peut poser quelques problèmes. En effet, de la sorte, nous ne pouvons pas être sûrs que les valeurs assignées soient correctes. Heureusement, la bibliothèque nous permet de pallier ce problème en réécrivant la méthode `checkConstraints()` (déclarée dans sa classe mère `Annotation`) dans notre classe représentant l'annotation, appelée à chaque assignation de valeur, dans l'ordre dans lequel sont assignées les valeurs. Vous pouvez ainsi vérifier l'intégrité des données, et lancer une erreur si il y a un problème. Cette méthode prend un argument : la cible d'où provient l'annotation. Dans notre cas, l'annotation vient de notre classe `Personnage`, donc le paramètre sera une instance de `ReflectionAnnotatedClass` représentant `Personnage`. Vous verrez ensuite que cela peut être une méthode ou un attribut.

Code : PHP

```
<?php
class ClassInfos extends Annotation
{
  public $author;
  public $version;

  public function checkConstraints($target)
  {
    if (!is_string($this->author))
    {
      throw new Exception('L\'auteur doit être une chaîne de
caractères');
    }

    if (!is_numeric($this->version))
    {
      throw new Exception('Le numéro de version doit être un nombre
valide');
    }
  }
}
```

Des annotations pour les attributs et méthodes

Jusqu'ici nous avons ajouté des annotations à une classe. Il est cependant possible d'en ajouter à des méthodes et attributs comme nous l'avons fait pour la classe :

Code : PHP

```
<?php
/**
* @Table("Personnages")
* @ClassInfos(author = "vyk12", version = "1.0")
*/
class Personnage
{
  /**
* @AttrInfos(description = 'Contient la force du personnage, de 0 à
100', type = 'int')
*/
  protected $force_perso;

  /**
* @ParamInfo(name = 'destination', description = 'La destination du
personnage')
* @ParamInfo(name = 'vitesse', description = 'La vitesse à laquelle
```

```
se déplace le personnage')
* @MethodInfos(description = 'Déplace le personnage à un autre
endroit', return = true, returnDescription = 'Retourne true si le
personnage peut se déplacer')
*/
  public function deplacer($destination, $vitesse)
  {
    // ...
  }
}
```

Pour récupérer une de ces annotations, il faut d'abord récupérer l'attribut ou la méthode. Nous allons pour cela se tourner vers `ReflectionAnnotatedProperty` et `ReflectionAnnotatedMethod`. Le constructeur de ces classes attend en premier paramètre le nom de la classe contenant l'élément et, en second, le nom de l'attribut ou de la méthode. Exemple :

Code : PHP

```
<?php
$reflectedAttr = new ReflectionAnnotatedProperty('Personnage',
'force_perso');
$reflectedMethod = new ReflectionAnnotatedMethod('Personnage',
'deplacer');

echo 'Infos concernant l\'attribut :';
var_dump($reflectedAttr->getAnnotation('AttrInfos'));

echo 'Infos concernant les paramètres de la méthode :';
var_dump($reflectedMethod->getAllAnnotations('ParamInfo'));

echo 'Infos concernant la méthode :';
var_dump($reflectedMethod->getAnnotation('MethodInfos'));
```

Notez ici l'utilisation de `ReflectionAnnotatedMethod::getAllAnnotations()`. Cette méthode permet de récupérer toutes les annotations d'une entité correspondant au nom donné en argument. Si aucun nom n'est donné, alors toutes les annotations de l'entité seront retournées.

Contraindre une annotation à une cible précise

Grâce à une annotation un peu spéciale, vous avez la possibilité d'imposer un type de cible pour une annotation. En effet, jusqu'à maintenant, nos annotations pouvaient être utilisées aussi bien par des classes que par des attributs ou des méthodes. Dans le cas des annotations `ClassInfos`, `AttrInfos`, `MethodInfos` et `ParamInfos`, cela présenterait un non-sens qu'elles puissent être utilisées par n'importe quel type d'élément.

Pour pallier ce problème, retournons à notre classe `ClassInfos`. Pour dire à cette annotation qu'elle ne peut être utilisée que sur des classes, il faut utiliser l'annotation spéciale `@Target` :

Code : PHP

```
<?php
/** @Target("class") */
class ClassInfos extends Annotation
{
  public $author;
  public $version;
}
```

À présent, essayez d'utiliser l'annotation `@ClassInfos` sur un attribut ou une méthode et vous verrez qu'une erreur sera levée.

Cette annotation peut aussi prendre pour valeur `property`, `method` ou `nesty`. Ce dernier type est un peu particulier et cette partie commençant déjà à devenir un peu imposante, j'ai décidé de ne pas en parler. Si cela vous intéresse, je ne peux que vous conseiller d'aller faire un tour sur la documentation d'addendum. 

Je vous ai aussi volontairement caché une classe : la classe `ReflectionParameter` qui vous permet d'obtenir des informations sur les paramètres de vos méthodes. Je vous laisse vous documenter à ce sujet, son utilisation est très simple. 

En résumé

- Il est possible d'obtenir des informations sur ses classes, attributs et méthodes, respectivement grâce à `ReflectionClass`, `ReflectionProperty` et `ReflectionMethod`.
- Utiliser des annotations sur ses classes permet, grâce à une bibliothèque telle qu'addendum, de récupérer dynamiquement leurs contenus.
- L'utilisation d'annotations dans un but de configuration dynamique est utilisée par certains frameworks ou ORM tel que Doctrine par exemple, ce qui permet d'économiser un fichier de configuration en plaçant la description des tables et des colonnes directement dans des annotations.

UML : présentation (1/2)

Programmer en orienté objet c'est bien beau, mais au début, c'est un peu difficile. On a du mal à s'y retrouver, on ne sait pas trop comment lier nos classes ni comment **penser** cet ensemble. L'UML est justement l'un des moyens pour y parvenir. L'UML (pour *Unified Modeling Language*, ou "langage de modélisation unifié" en français) est un langage permettant de modéliser nos classes et leurs interactions. Concrètement, cela s'effectue par le biais d'un diagramme : vous dessinerez vos classes et les lierez suivant des conventions bien précises. Cela vous permettra ainsi de mieux visualiser votre application et de mieux la penser.

La présentation de l'UML ne sera volontairement pas très approfondie, le but n'est pas de faire de vous des pros de la modélisation, mais juste de vous enseigner les bases afin que vous puissiez être capables de modéliser vous-mêmes.

UML, kézako ?

L'UML est un langage de modélisation objet. Il permet donc de modéliser vos objets et ainsi représenter votre application sous forme de diagramme.
Avant d'aller plus loin, attardons nous un peu sur la signification d'UML : *Unified Modeling Language*, « langage de modélisation unifié ». UML n'est pas un langage à proprement parler, plutôt une sorte de *méthodologie*.

Mais qu'est-ce que c'est concrètement ? À quoi ça pourra bien me servir ?

Grâce à UML, vous pourrez modéliser toute votre application. Quand je dis *toute*, j'entends par là la plupart de votre application car PHP n'est pas un langage orienté objet : le modèle objet lui a été implémenté au cours des versions (dès la version 4). Grâce à ces diagrammes, vous pourrez donc représenter votre application : son fonctionnement, sa mise en route, les actions susceptibles d'être effectuées par l'application, etc. Ces diagrammes ont été conçus pour que quelqu'un n'ayant aucune connaissance en informatique puisse comprendre le fonctionnement de votre application. Certes, certains diagrammes sont réservés aux développeurs car assez difficiles à expliquer si on ne sait pas programmer : ce sont ces diagrammes que nous allons étudier.

D'accord, je peux modéliser mon application mais en quoi cela va-t-il m'aider ?

Si vous avez un gros projet (et là je ne parle pas forcément de PHP mais de tout langage implémentant la POO), il peut être utile de le modéliser afin d'y voir plus clair. Si vous vous focalisez sur la question *« Par où commencer ? »* au début du projet, c'est qu'il vous faut un regard plus objectif sur le projet. Les diagrammes vous apporteront ce regard différent sur votre application.

L'UML peut aussi être utile quand on commence à programmer OO mais qu'on ne sait pas trop comment s'y prendre. Par exemple, vous ne savez pas trop encore comment programmer orienté objet de façon concrète (vous avez vu comment créer un mini-jeu, c'est cool, mais un livre d'or ou un système de news, vous savez ?). Sachez donc que l'UML va vous être d'une grande aide au début. Ça vous aidera à mieux penser objet car les diagrammes représentent vos classes, vous vous rendrez ainsi mieux compte de ce qu'il est intelligent de faire ou pas. Cependant, l'UML n'est pas un remède miracle et vous ne saurez toujours pas comment vous y prendre pour réaliser votre toute première application, mais une fois que je vous aurai montré et que vous aurez acquis un minimum de pratique, l'UML pourra vous aider.

Enfin, l'UML a aussi un rôle de documentation. En effet, quand vous représenterez votre projet sous forme de diagramme, quiconque sachant le déchiffrer pourra (du moins, si vous l'avez bien construit) comprendre le déroulement de votre application et éventuellement reprendre votre projet (c'est toujours mieux que la phpdoc que nous utilisions jusqu'à présent pour commenter nos classes (revenez au dernier TP si vous avez un trou de mémoire)).

Bref, que ce soient pour les gros projets personnels ou professionnels, l'UML vous suivra partout. Cependant, tout le monde ne trouve pas l'UML très utile et certains préfèrent modéliser le projet « dans leur tête ». Vous verrez avec le temps si vous avez réellement besoin de l'UML ou si vous pouvez vous en passer, mais pour l'instant, je vous conseille de modéliser (enfin, dès le prochain chapitre). 

Il existe plusieurs types de diagrammes. Nous, nous allons étudier les *diagrammes de classe*. Ce sont des diagrammes modélisant vos classes et montrant les différentes interactions entre elles. Un exemple ? Regardez plutôt la figure suivante.

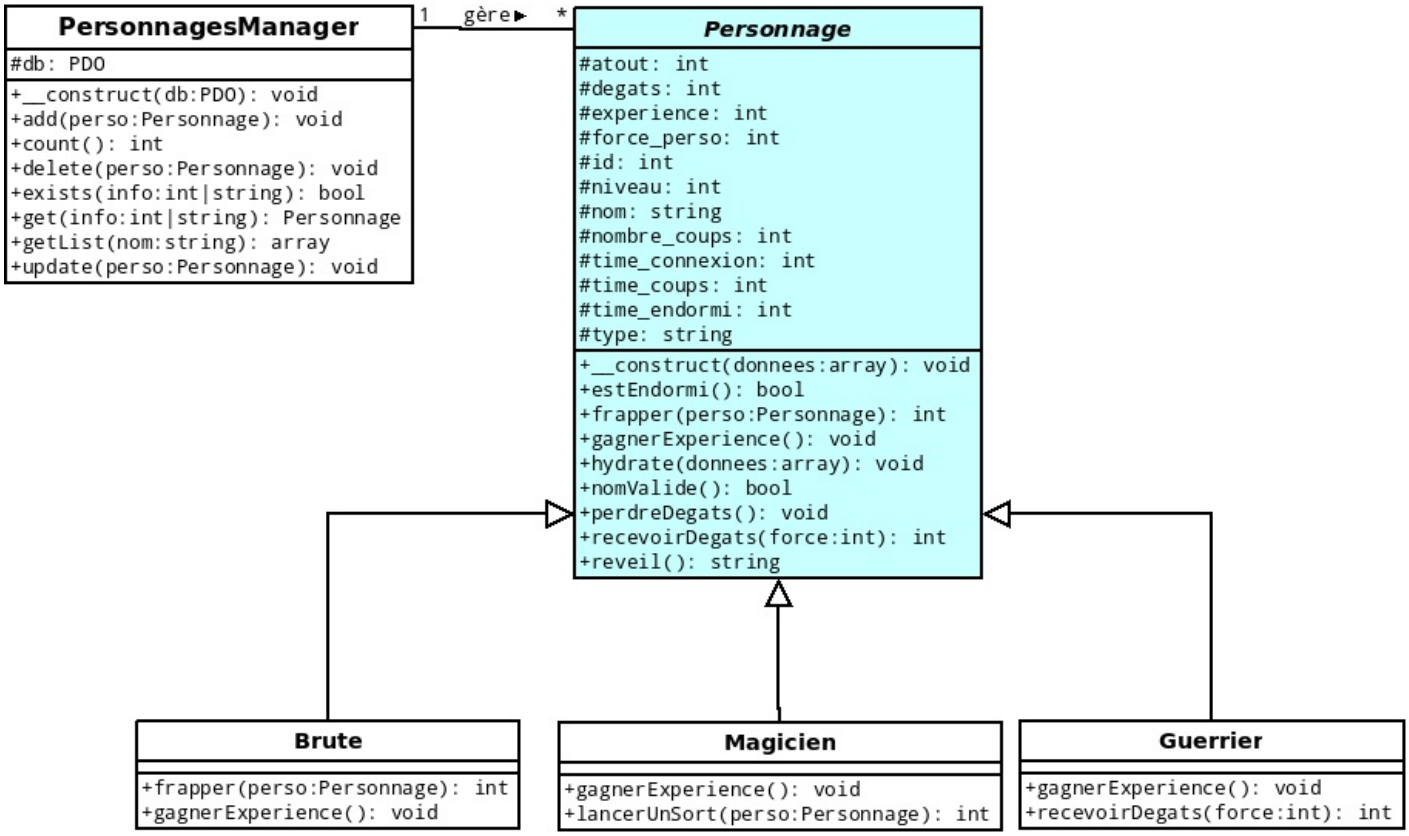


Diagramme UML modélisant notre dernier TP

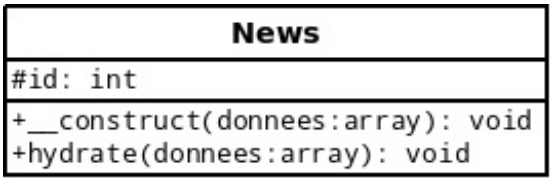
Vous aurez sans doutes reconnu notre dernier TP représenté sur un diagramme.

Ce diagramme a l'avantage d'être très simple à étudier. Cependant, il y a des cas complexes et des conventions à connaître car, par exemple, des méthodes abstraites n'ont pas le même style d'écriture qu'une méthode finale. Nous allons étudier les bases, petit à petit, puis, à la fin de ce chapitre, vous serez capable d'analyser complètement un diagramme de classe.

Modéliser une classe

Première approche

Nous allons commencer en douceur par analyser une simple classe modélisée. Nous allons prendre notre classe `News` simplifiée (voir la figure suivante).



Notre classe News modélisée

Nous avons donc ici une simple classe. Commençons par analyser celle-ci.

- En haut, en gras et gros, nous avons le nom de la classe (ici, `News`).
- Ensuite, séparé du nom de la classe, vient un attribut de cette classe : il s'agit de `id` précédé d'un #, nous verrons ce que ça signifie juste après.
- Enfin, séparée de la liste d'attributs, vient la liste des méthodes de la classe. Toutes sont précédées d'un +, qui a, comme le #, une signification.

Nous allons commencer par analyser la signification du # et des +. Je ne vous fais pas attendre : ces signes symbolisent la portée de l'attribut ou de la méthode. Voici la liste des trois symboles :

- Le signe + : l'élément suivi de ce signe est public.
- Le signe # : l'élément suivi de ce signe est protégé.

- Le signe - : l'élément suivi de ce signe est privé.

Maintenant que vous savez à quoi correspondent ces signes, regardez à droite de l'attribut : suivi de deux points, on peut lire `int`. Cela veut dire que cet attribut est de type `int`, tout simplement. `int` est le diminutif de `integer` qui veut dire `entier` : notre attribut est donc un nombre entier.

À droite des méthodes, nous pouvons apercevoir la même chose. Ceci indique le type de la valeur de retour de celle-ci. Si elle ne renvoie rien, alors on dit qu'elle est vide (= `void`). Si elle peut renvoyer plusieurs types de résultats différents, alors on dit qu'elle est mixte (= `mixed`). Par exemple, une méthode `__get` peut renvoyer une chaîne de caractères ou un entier (regardez la classe `Personnage` sur le premier diagramme).

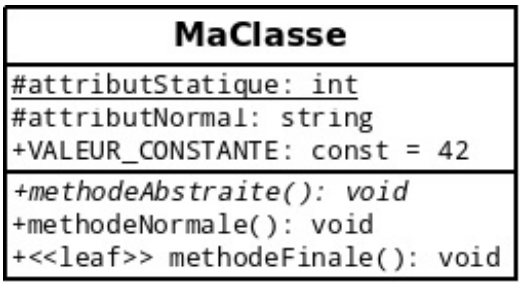
Encore une dernière chose à expliquer sur ce diagramme : ce qu'il y a entre les parenthèses suivant le nom des méthodes. Comme vous vous en doutez, elles contiennent tous les attributs ainsi que leur type (entier, tableau, chaîne de caractères, etc.). Si un paramètre peut être de plusieurs types, alors, comme pour les valeurs de retour des méthodes, on dit qu'il est *mixte*.

Si un attribut, une méthode ou un paramètre est une instance ou en renvoie une, il ne faut pas dire que son type est `object`. Son type est le nom de la classe, à savoir `Personnage`, `Brute`, `Magicien`, `Guerrier`, etc.

Ensuite, il y a des conventions concernant le style d'écriture des attributs, des constantes et des méthodes.

- Sans style d'écriture particulier, la méthode est « normale », c'est-à-dire ni abstraite, ni finale (comme ici). Si la méthode est en italique, cela signifie qu'elle est abstraite. Pour montrer qu'une méthode est finale, on le spécifie en *stéréotype* (on place le mot `leaf` entre chevrons à côté de la méthode).
- Si l'attribut ou la méthode est souligné, cela signifie que l'élément est statique.
- Une constante est représentée comme étant un attribut public, statique, de type `const` et est écrite en majuscules.

Exemple récapitulant tout ça (voir la figure suivante).



Exemple de classe modélisée

Tous ces « codes » (textes en gras, soulignés ou en italiques) sont une norme qu'il faut respecter. Tout le monde sait qu'une méthode en italique signifie qu'elle est abstraite par exemple : ne faites donc pas comme bon vous semble !

Exercices

Maintenant que vous savez tout cela, je vais vous donner quelques attributs et méthodes et vous allez essayer de deviner quels sont ses particularités (portée, statique ou non, abstraite, finale, ou ni l'un ni l'autre, les arguments et leur type...).

Exercice 1 : *-maMethode (param1:int) : void*

Correction : nous avons ici une méthode `maMethode` qui est abstraite (car en italique) et statique (car soulignée). Elle ne renvoie rien et accepte un argument : `$param1`, qui est un entier. Sans oublier sa visibilitée, symbolisée par le signe -, qui est privée.

Exercice 2 : #maMethode (param1:mixed) : array

Correction : nous avons ici une méthode `maMethode` ni abstraite, ni finale. Elle renvoie un tableau et accepte un argument : `$param1`, qui peut être de plusieurs types. Sans oublier sa visibilitée, symbolisée par le signe #, qui est protégée.

Exercice 3 : +<<leaf>> maMethode() : array

Correction : cette fois-ci, la méthode est finale (signalée par le mot `leaf`) et statique (car soulignée). Elle ne prend aucun argument et renvoie un tableau. Quant à sa visibilité, le signe + nous informe qu'elle est publique.

Les différents codes (italique, soulignements, etc.) rentreront dans votre tête au fil du temps, je ne vous impose pas de tous les apprendre sur-le-champ. Référez-vous autant que nécessaire à cette partie en cas de petits trous de mémoire.

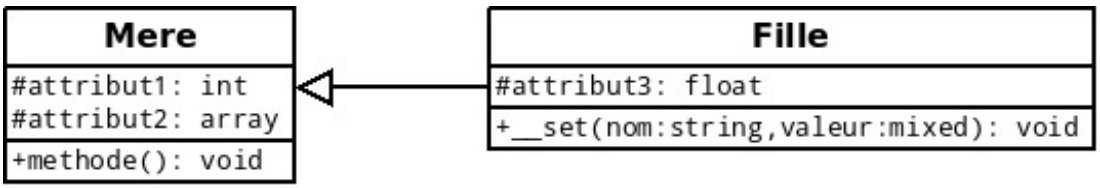
Maintenant que vous savez analyser un objet, il serait bien de savoir analyser les interactions entre ceux-ci, non ?

Modéliser les interactions

Nous attaquons la dernière partie de ce chapitre. Nous allons voir comment modéliser les interactions entre les objets que l'on a modélisés. Jusqu'à présent, nous savions comment les reconnaître, mais l'avantage de la POO est de créer un ensemble d'objets qui interagissent entre eux afin de créer une véritable application.

L'héritage

Parmi les interactions, on peut citer l'héritage. Comme montré dans le premier diagramme que vous avez vu, l'héritage est symbolisé par une simple flèche, comme indiqué à la figure suivante.



Exemple de modélisation d'une classe fille et d'une classe mère

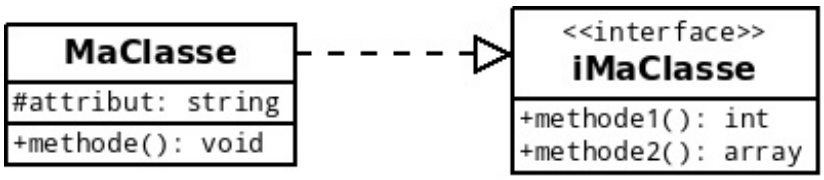
Ce diagramme équivaut au code suivant :

Code : PHP

```php
<?php
class Mere
{
  protected $attribut1;
  protected $attribut2;

  public function methode()
  {

  }
}

class Fille extends Mere
{
  protected $attribut3;

  public function __set($nom, $valeur)
  {

  }
}
?>
```

Les interfaces

Vous connaissez également les interactions avec les interfaces. En effet, comme je l'ai déjà dit, une interface n'est rien d'autre qu'une classe entièrement abstraite, elle est donc considérée comme tel. Si une classe doit implémenter une interface, alors on utilisera la flèche en pointillés, comme à la figure suivante.



Exemple de modélisation d'une classe implémentant une interface

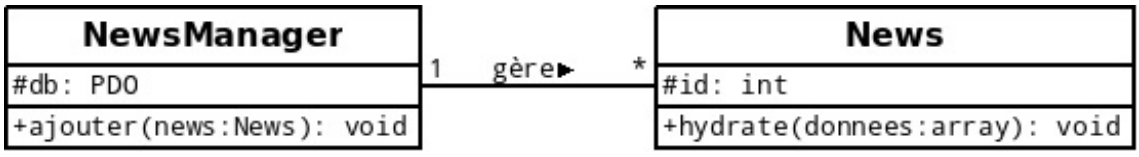
Traduit en PHP, ça donne :

Code : PHP

```php
<?php
interface iMaClasse
{
  public function methode1();
  public function methode2();
}

class MaClasse implements iMaClasse
{
  protected $attribut;

  public function methode()
  {

  }

  // Ne pas oublier d'implémenter les méthodes de l'interface !

  public function methode1()
  {

  }

  public function methode2()
  {

  }
}
?>
```

L'association

Nous allons voir encore trois interactions qui se ressemblent. La première est **l'association**. On dit que deux classes sont associées lorsqu'une instance des deux classes est amenée à interagir avec l'autre instance. L'association entre deux classes est modélisée comme à la figure suivante.



Exemple de modélisation d'une association

L'association est ici caractérisée par le fait qu'une méthode de la classe `NewsManager` entre en relation avec une instance de la classe `News`.

Attends deux secondes... Je vois des choses écrites sur la ligne ainsi qu'aux extrémités, qu'est-ce que c'est ?

Le mot écrit au centre, au-dessus de la ligne est la **définition** de la relation. Il est suivi d'un petit symbole indiquant le sens de

l'association. Ainsi, on peut lire facilement « `NewsManager` gère `News` ».

Ensuite, vous voyez le chiffre 1 écrit à gauche et une astérisque à droite. Ce sont les **cardinalités**. Ces cardinalités présentent le nombre d'instances qui participent à l'interaction. Nous voyons donc qu'il y a **1** manager pour une infinité de news. On peut désormais lire facilement « 1 `NewsManager` gère une infinité de `News` ». Les cardinalités peuvent être écrites sous différentes formes :

- **x** (nombre entier) : tout simplement la valeur exacte de **x**.
- **x..y** : de **x** à **y** (exemple : **1..5**).
- ***** : une infinité.
- **x..*** : **x** ou plus (exemple : **5..***).

L'agrégation

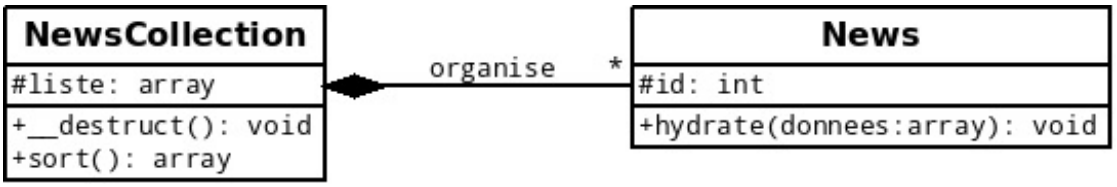
La deuxième flèche que je vais vous présenter est **l'agrégation**. Il s'agit d'une forme d'association un peu particulière : on parlera d'agrégation entre deux classes lorsque l'une d'entre elles contiendra au moins une instance de l'autre classe. Une agrégation est caractérisée de la sorte (voir figure suivante).

Exemple de modélisation d'une agrégation

Vous pouvez remarquer qu'il n'y a pas de cardinalité du côté du losange. En effet, le côté ayant le losange signifie qu'il y a obligatoirement une et une seule instance de la classe par relation (ici la classe est `NewsCollection`).

La composition

La dernière interaction que je vais vous présenter est la **composition**. La composition est une agrégation particulière. Imaginons que nous avons une classe A qui contient une ou plusieurs instance(s) de B. On parlera de composition si, lorsque l'instance de A sera supprimée, toutes les instances de B contenues dans l'instance de A sont elles aussi supprimées (ce qui n'était pas le cas avec l'agrégation). Une composition est représentée de la sorte (voir la figure suivante).



Exemple de modélisation d'une composition

Notez la présence de la méthode *__destruct()* qui sera chargée de détruire les instances de la classe `News`. Cependant, celle-ci n'est pas obligatoire : vous pouvez très bien, dans l'une de vos méthodes, placer un `$this->liste[] = new News;`, et l'instance créée puis stockée dans l'attribut `$liste` sera donc automatiquement détruit. 

Ce chapitre ne se veut pas complet pour la simple et bonne raison qu'il serait beaucoup trop long de faire le tour de l'UML. En effet, un cours complet sur l'UML s'étalerait sur beaucoup de chapitres, et ce n'est clairement pas le but du tutoriel. Si ce sujet vous intéresse, je peux vous renvoyer sur le site uml.free.fr.

En résumé

- L'UML est un moyen parmi d'autres de modéliser son application afin de mieux s'y retrouver.
- La modélisation d'une classe et des interactions se réalise en respectant certaines conventions.
- Il y a association lorsqu'une classe A se sert d'une classe B.
- Une agrégation est une association particulière : il y a agrégation entre A et B si l'objet A possède une ou plusieurs instances de B.
- Une composition est une agrégation particulière : il y a composition entre A et B si toutes les instances de B contenues dans A sont supprimées lorsque A est supprimée.

UML : modélisons nos classes (2/2)

Voici la suite directe du précédent chapitre ayant pour objectif de vous introduire à l'UML. Nous savons actuellement analyser des diagrammes de classes et les interactions entre elles, mais que diriez-vous de créer vous-même ces diagrammes ?

Pour ce faire, nous allons avoir besoin d'un programme : je vais vous présenter Dia.

Ayons les bons outils

Avant de se lancer tête baissée dans la modélisation, il serait bien d'avoir les logiciels qui le permettront (ça pourrait aider, sait-on jamais). Pour cela, nous allons avoir besoin d'un logiciel de modélisation de diagrammes UML. Il en existe plusieurs, pour ma part, j'ai choisi Dia.

La raison de ce choix est simple : une extension qui permet de convertir les diagrammes UML en code PHP a été développée pour ce logiciel ! Ainsi, en quelques clics, toutes vos classes contenant les attributs et méthodes ainsi que les interactions seront générées, le tout commenté avec une phpdoc. Bref, cela nous simplifiera grandement la tâche !

Installation

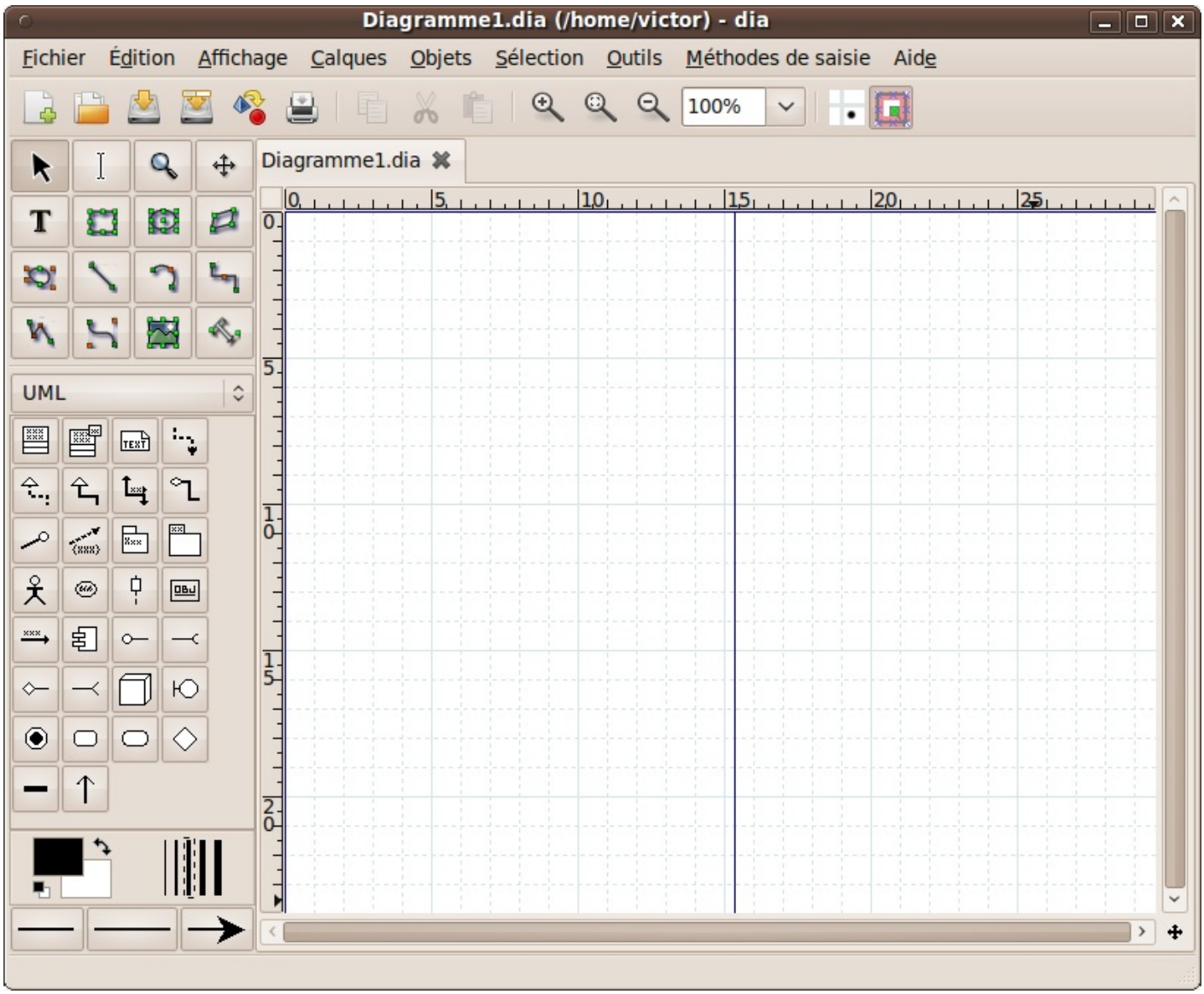
Des installateurs sont disponibles sur le site dia-installer.de. Le lien de téléchargement pour Windows est sur la page d'accueil. Deux petits liens situés en-dessous de ce lien de téléchargement mènent aux pages proposant les liens pour Linux et Mac OS X.

Installation de l'extension uml2php5

Vous avez correctement installé Dia sur votre ordinateur. Il faut maintenant installer l'extension uml2php5 qui nous permettra de générer automatiquement le code de nos classes à partir de nos diagrammes. Pour cela, il vous faut télécharger cinq fichiers. Ces fichiers sont disponibles sur le site de l'extension, en ayant cliqué sur **Download** à gauche. Téléchargez l'archive .zip ou .tar.gz suivant vos préférences. Décompressez cette archive et copiez / collez les cinq fichiers dans le dossier **xslt** présent dans le répertoire d'installation de Dia.

Lancer Dia

Histoire que nous soyons tous au point, je vais vous demander de lancer votre logiciel afin que nous soyons bien sûrs d'avoir tous le même. Si l'interface est différente, veillez bien à avoir téléchargé la version 0.97 du logiciel (actuellement la dernière). Si une version plus récente est sortie, vous pouvez me contacter.



Fenêtre principale de Dia

Deux zones principales

L'interface étant assez intuitive, je ne vais pas m'étendre sur sa présentation. Je vais me contenter de vous donner le rôle des deux parties de cette interface.

La première partie, à gauche, contient tous les outils (créer une classe, une interaction, ou bien un trait, un cercle, etc.). La seconde, à droite, est beaucoup plus grande : elle contiendra notre diagramme. C'est à l'intérieur d'elle que l'on effectuera nos opérations pour tout mettre en œuvre.

Unir les fenêtres

Il se peut que vous ayez deux fenêtres séparées. Si c'est le cas, c'est que vous n'avez pas passé l'argument **--integrated** au programme. Vous pouvez passer cet argument en lançant le logiciel en ligne de commande. Il y a cependant plus pratique, voici donc des petites astuces suivant votre système d'exploitation.

Sous Windows

Sous Windows, vous pouvez créer un raccourci. Pour cela, allez dans le dossier d'installation de Dia puis dans le dossier **bin**. Faites un clic droit sur le fichier **diaw.exe** et cliquez sur **Copier**. Allez dans le répertoire où vous voulez créer le raccourci, faites un clic droit dans ce répertoire puis choisissez l'option **Coller le raccourci**. Faites un clic droit sur le fichier créé, puis cliquez sur **Propriétés**. Dans le champ texte **Cible**, rajoutez à la fin (en dehors des guillemets) **--integrated**. Ainsi, quand vous lancerez Dia depuis ce raccourci, les deux fenêtres seront fusionnées.

Par défaut, le raccourci présent dans le menu **démarrer** lance Dia avec l'option **--intergrated**. Si ce n'est pas le cas, modifiez la cible de celui-ci comme on vient de le faire (Clic droit > Propriétés > ...).

Sous Linux

Je vais ici vous donner une manipulation simple sous Ubuntu afin de modifier la cible du lien pointant sur le logiciel dans le menu **Applications**. Dans le menu **Système**, allez dans **Préférences** puis **Menu principal**. À gauche de la fenêtre qui est apparue, sous le menu **Applications**, cliquez sur **Graphisme**. Au milieu de la fenêtre, cliquez sur **Éditeur de diagrammes Dia**. À droite, cliquez sur **Propriétés**. Dans le champ texte **Commande**, placez-y **dia --integrated % F**. Fermez le tout, relancez votre programme via le menu et vous avez vos deux fenêtres fusionnées.

Sous les autres distributions, l'idée est la même : il faut vous arranger pour modifier le raccourci afin d'ajouter l'option **--integrated**.

Sous Mac OS

Hélas sous Mac OS, vous ne pouvez créer de raccourci. La solution consiste donc à lancer le logiciel en ligne de commande. Heureusement, celle-ci est assez courte et simple à retenir. En effet, un simple `dia --integrated` suffit.

Cependant, vous pouvez télécharger un lanceur qui vous facilitera la tâche pour lancer l'application.

Modéliser une classe

Créer une classe

Lançons-nous maintenant dans la création d'un diagramme. Pour commencer, modélisons notre première classe. Pour cela, rien de plus simple. Je vais vous demander de cliquer sur cette icône (voir figure suivante).

Modéliser une classe

Cette icône, comme l'infobulle nous l'indique, représente l'outil permettant de créer une classe. Ainsi, dans la zone contenant notre diagramme, il suffira de cliquer n'importe où pour qu'une classe soit créée à l'endroit où l'on a cliqué. Essayez donc.

Normalement, vous devriez avoir obtenu quelque chose ressemblant à ceci (voir figure suivante).

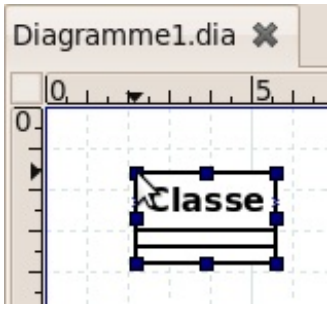
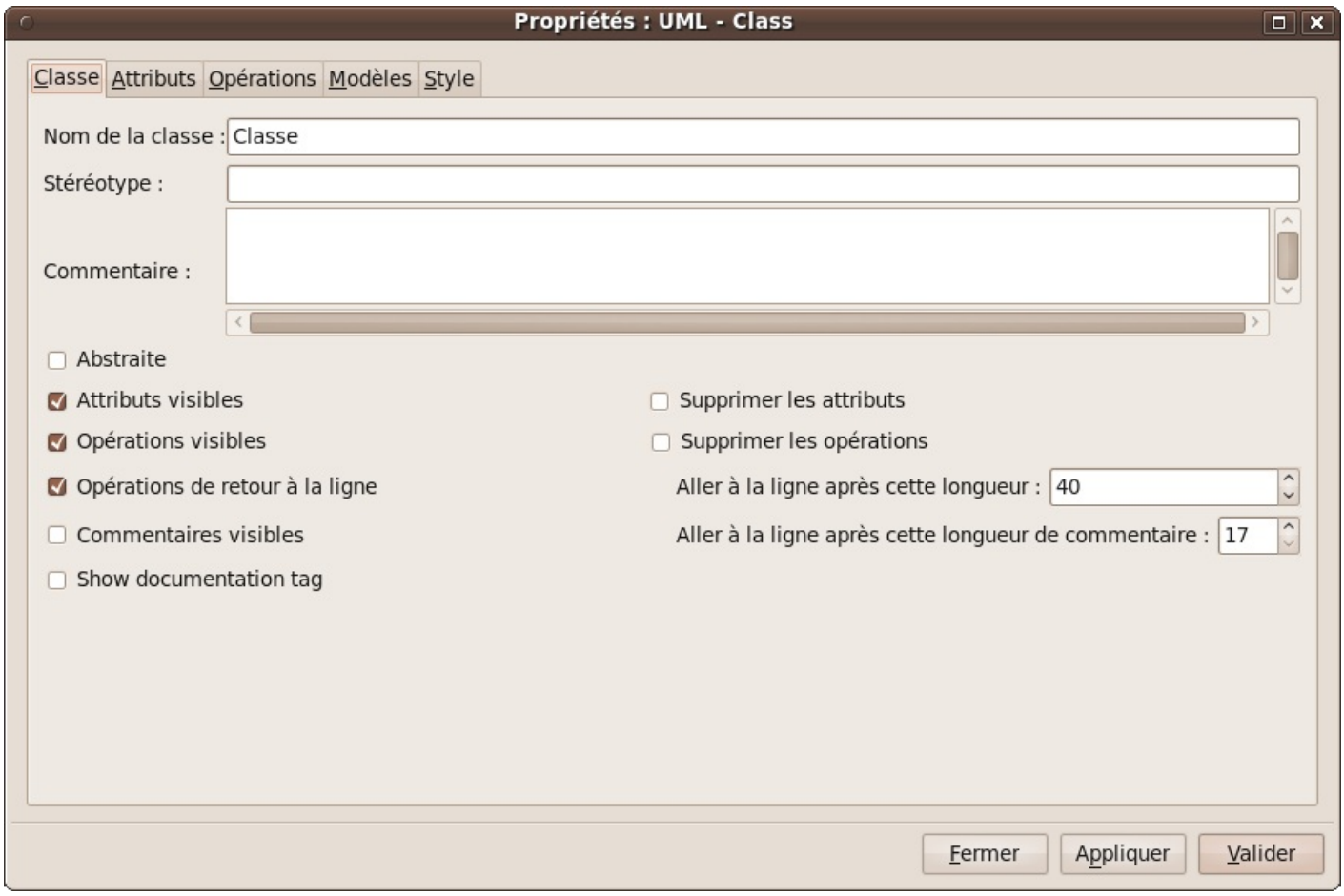


Diagramme de classe vierge

Vous voyez huit carrés ainsi que deux croix bleues autour de la classe. Si les carrés sont présents, cela veut dire que la classe est sélectionnée (par défaut sélectionnée quand vous en créez une). Si vous voulez enlever cette sélection, il vous suffit de cliquer à côté. Vous verrez ainsi les huit carrés remplacés par de nouvelles petites croix bleues qui ont une signification bien particulière. Je vous expliquerai ceci plus tard.

Modifier notre classe

Nous allons accéder aux propriétés de notre classe afin de pouvoir la modifier. Pour cela, je vais vous demander de double-cliquer dessus afin d'obtenir cette fenêtre (voir figure suivante).



Fenêtre permettant de modifier la classe

Décortiquons un peu cette fenêtre. Tout ceci est en fait très simple, il suffit juste de quelques explications.

Tout en haut, vous voyez une liste de cinq onglets :

- **Classe** : permet de gérer les options concernant cette classe.
- **Attributs** : permet de gérer la liste des attributs de cette classe.
- **Opérations** : permet de gérer la liste des méthodes de cette classe.
- **Modèles** : inutile en PHP étant donné qu'il n'y a pas de templates. Si ça vous intrigue, allez voir en C++ par exemple ;).
- **Style** : permet de modifier le style de la classe (couleurs et effets de texte).

Gestion des options de la classe

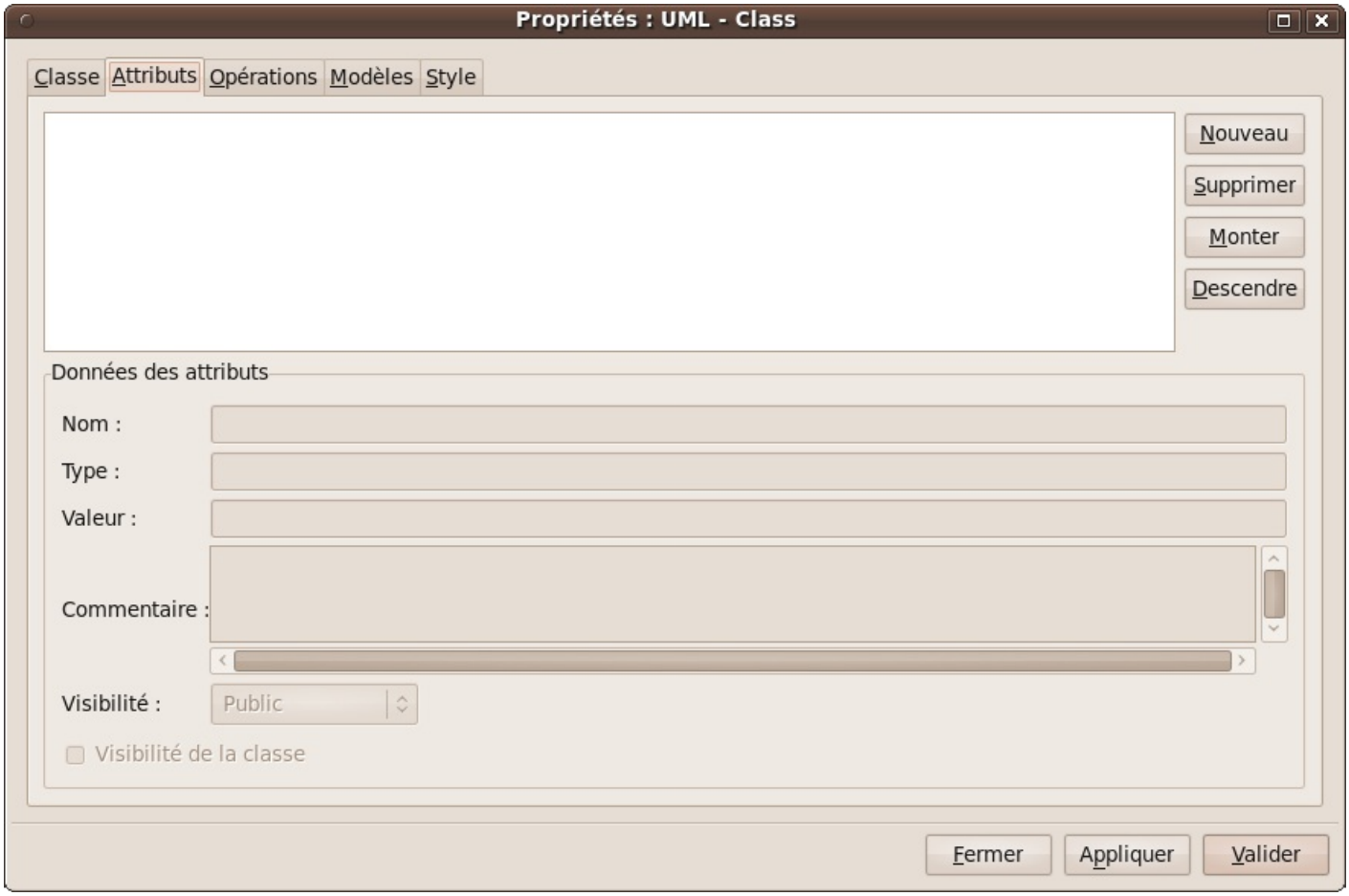
En haut, vous pouvez apercevoir trois champs texte. Le premier est le nom de la classe. Vous pouvez par exemple y indiquer le nom `Personnage`. Ensuite vient le stéréotype. Ce champ est à utiliser pour spécifier que la classe est particulière. Par exemple, s'il s'agit d'une interface, on y écrira simplement « interface ». Enfin, dans le dernier champ texte, nous placerons des commentaires relatifs à la classe.

Ensuite viennent une série de cases à cocher. La première signifie (comme vous vous en doutez) que la classe est abstraite ou non. Par la suite, nous pouvons apercevoir trois cases à cocher permettant d'afficher ou non des éléments (afficher ou masquer les attributs, méthodes ou commentaires). Si vous décidez de masquer les attributs ou méthodes, leur bloc disparaîtra de la classe (les bordures entourant le bloc comprises), tandis que si vous décidez de supprimer les attributs ou méthodes (via les cases à cocher de droite), le bloc restera visible mais ne contiendra plus leur liste d'attributs ou méthodes. Le masquage des attributs est utile dans le cas où la classe est une interface par exemple.

Pour que vous vous rendiez compte des modifications que vous effectuez en temps réel, cliquez sur **Appliquer**. Vous verrez ainsi votre classe se modifier avec les nouvelles options. Si ça ne vous plait pas et que vous voulez annuler tout ce que vous avez fait, il vous suffit de cliquer sur **Fermer**. Par contre, si vous voulez enregistrer vos informations, cliquez sur **Valider**.

Gestion des attributs

Commençons par ajouter des attributs à notre classe. Pour cela, cliquez sur l'onglet **Attributs** (voir figure suivante).



Fenêtre permettant de gérer les attributs

Analysons le contenu de cette fenêtre. La partie la plus évidente que vous voyez est le gros carré blanc en haut à gauche : c'est à l'intérieur de ce bloc que seront listés tous les attributs. Vous pourrez, à partir de là, en sélectionner afin d'en modifier, d'en supprimer, d'en monter ou descendre d'un cran dans la liste. Afin d'effectuer ces actions, il vous suffira d'utiliser les boutons situés à droite de ce bloc :

- Le bouton **Nouveau** créera un nouvel attribut.
- Le bouton **Supprimer** supprimera l'attribut sélectionné.
- Le bouton **Monter** montera l'attribut d'un cran dans la liste (s'il n'est pas déjà tout en haut).
- Le bouton **Descendre** descendra l'attribut d'un cran dans la liste (s'il n'est pas déjà tout en bas).

Créez donc un nouvel attribut en cliquant sur le bouton adapté. Vous voyez maintenant tous les champs du bas qui se dégrisent. Regardons de plus près ce dont il s'agit.

- Un premier champ texte est présent afin d'y spécifier le nom de l'attribut (par exemple, vous pouvez y inscrire `force`).
- En dessous est présent un champ texte demandant le type de l'attribut (s'il s'agit d'une chaîne de caractères, d'un nombre entier, d'un tableau, d'un objet, etc.). Par exemple, pour l'attribut créé, vous pouvez entrer `int`.
- Ensuite vient un champ texte demandant la valeur de l'attribut. Ce n'est pas obligatoire, il faut juste y spécifier quelque chose quand on souhaite que notre attribut ait une valeur par défaut (par exemple, vous pouvez entrer **0**).
- Le dernier champ texte est utile si vous souhaitez laisser un commentaire sur l'attribut (ce que je vous conseille de faire).
- Vient ensuite une liste déroulante permettant d'indiquer la visibilité de l'attribut. Il y a quatre options. Vous reconnaîtrez parmi elles les trois types de visibilités. En plus de ces trois types, vous pouvez apercevoir un quatrième type nommé **Implémentation**. Ne vous en préoccupez pas, vous n'en aurez pas besoin. 
- La dernière option est une case à cocher, suivie de **Visibilité de la classe**. Si vous cochez cette case, cela voudra dire que votre attribut sera statique.

Entraînez-vous à créer quelques attributs bidons (ou bien ceux de la classe `Personnage` si vous êtes à cours d'idée). Vous pourrez ainsi tester les quatre boutons situés à droite du bloc listant les attributs afin de bien comprendre leur utilisation (bien que ceci ne doit pas être bien difficile, mais sait-on jamais).

Gestion des constantes

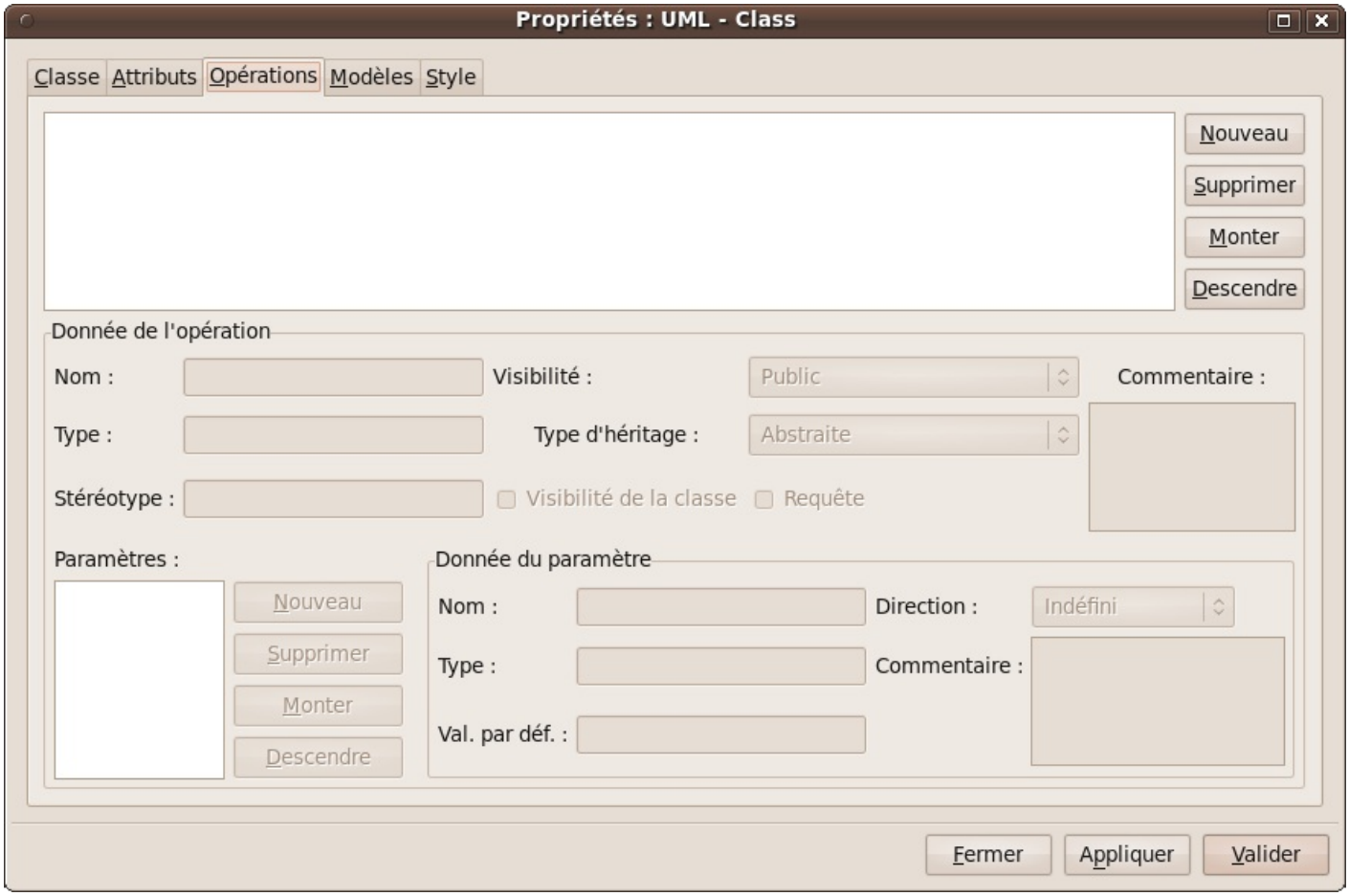
Comme je l'avais brièvement évoqué au chapitre précédent, une constante se reconnaît grâce à plusieurs signes. En effet, une constante se déclare comme un attribut sur un diagramme, mais respecte plusieurs contraintes :

- Elle doit être écrite en majuscules et les espaces sont remplacées par des *underscores* (comme dans votre script PHP).
- Elle doit avoir une visibilité **publique**.
- Elle doit être de type `const`.
- Elle doit être **statique**.
- Elle doit posséder une **valeur**.

Je vous ai appris à créer des attributs, il serait donc redondant que je vous apprenne à créer des constantes qui ne sont, comme je viens de le dire, que des attributs particuliers.

Gestion des méthodes

Après avoir ajouté des attributs à notre classe, voyons comment lui ajouter des méthodes. Pour cela, cliquez sur l'onglet **Opérations** (voir la figure suivante).

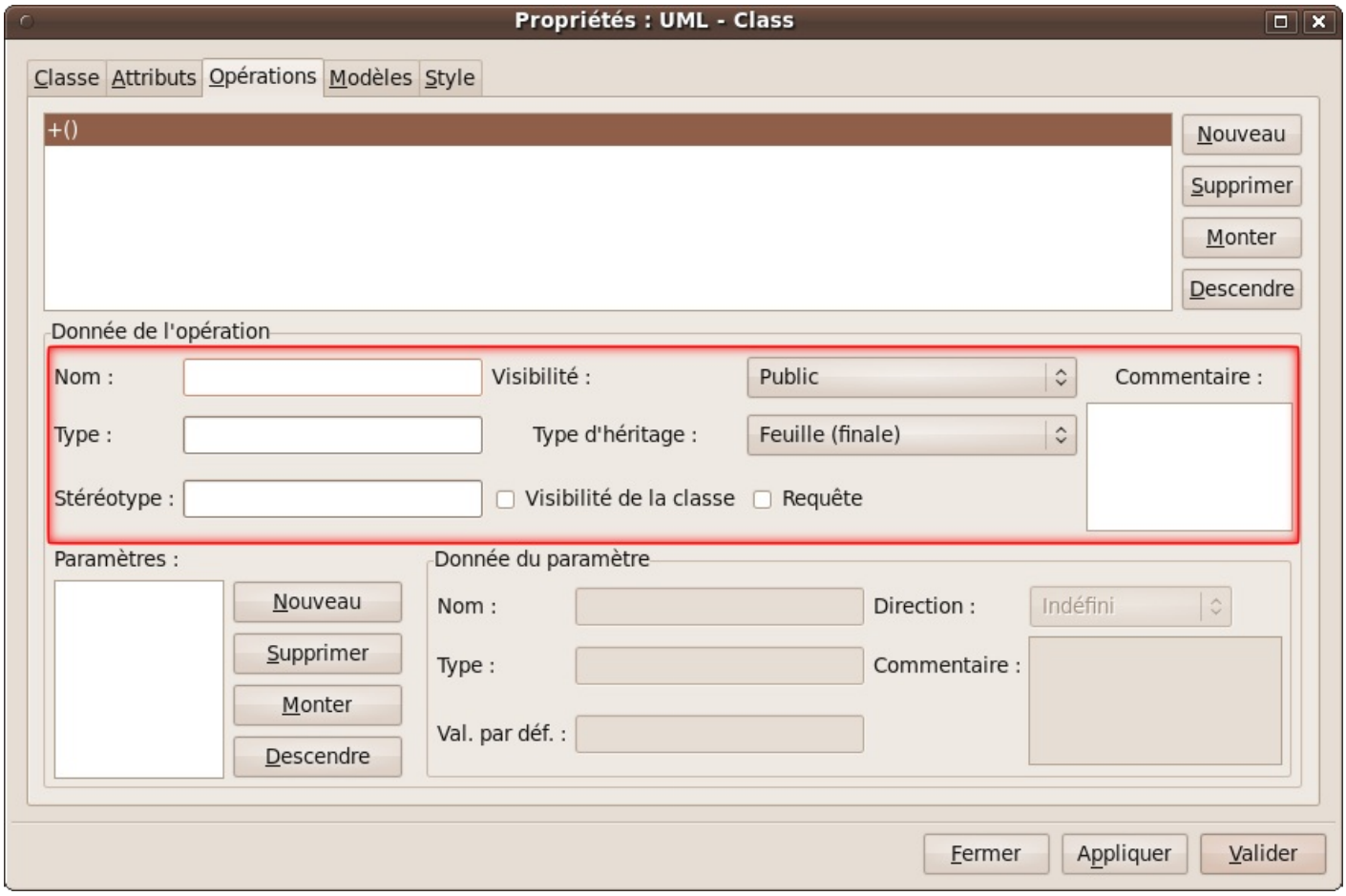


Fenêtre permettant de gérer les méthodes

Vous constatez déjà un peu plus de bazar.

Cependant, si vous observez bien et que vous avez bien suivi la précédente partie sur la gestion des attributs, vous ne devriez pas être entièrement dépaysés. En effet, en haut, nous avons toujours notre bloc blanc qui listera, cette fois-ci, nos méthodes, ainsi que les quatre boutons à droite effectuant les mêmes opérations. Créez donc une nouvelle méthode en cliquant simplement sur le bouton **Nouveau**, comme pour les attributs, et bidouillons-la un peu.

Pour commencer, je vais parler de cette partie de la fenêtre (voir la figure suivante).



Zone permettant de modifier la méthode

La partie encadrée de rouge concerne la méthode en elle-même. Les champs qui restent concernent ses paramètres, mais on verra ça juste après.

Les deux premiers champs textes (**Nom** et **Type**) ont le même rôle que pour les attributs (je vous rappelle que le type de la méthode est le type de la valeur renvoyée par celle-ci). Ensuite est présent un champ texte **Stéréotype**. Cela est utile pour afficher une caractéristique de la méthode. Par exemple, si une méthode est finale, on mettra *leaf* dans ce champ. Nous avons, comme pour les attributs, deux autres options qui refont leur apparition : la visibilité et la case à cocher **Visibilité de la classe**. Inutile de vous rappeler ce que c'est je pense (si toutefois vous avez un trou de mémoire, il vous suffit de remonter un tout petit peu dans ce cours). Et pour en finir sur les points communs avec les attributs, vous pouvez apercevoir tout à droite le champ texte **Commentaire** qui a aussi pour rôle de spécifier les commentaires relatifs à la méthode.

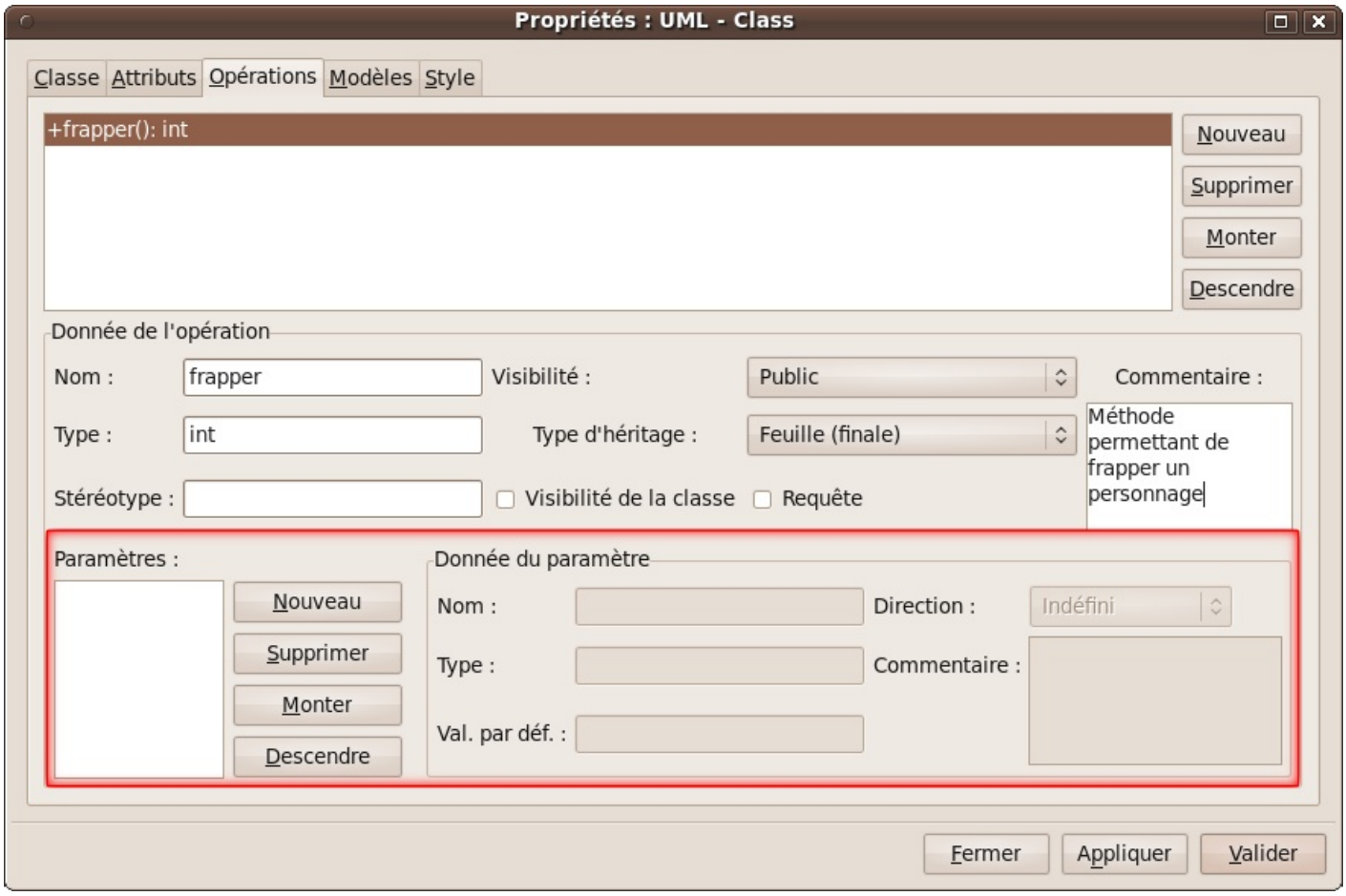
Par contre, contrairement aux attributs, nous avons ici deux nouvelles options. La première est **Type d'héritage** qui est une liste déroulante présentant trois options :

- **Abstraite** : à sélectionner si la méthode est abstraite.
- **Polymorphe (virtuelle)** : cela ne nous concerne pas (nous travaillons avec PHP qui n'implémente pas le concept de méthode virtuelle).
- **Feuille (finale)** : à sélectionner si la méthode n'est ni abstraite, ni finale (contrairement à ce qui est indiqué, cela ne veut pas dire que la méthode est finale !).

La deuxième option est une case à cocher. À côté de celle-ci est écrit **Requête**. Vous vous demandez sans doute ce qu'une requête vient faire ici, non ? En fait, si vous cochez cette case, cela veut dire que la méthode ne modifiera aucun attribut de la classe. Par exemple, vos accesseurs (les méthodes étant chargées de renvoyer la valeur d'attributs privés ou protégés) sont considérés comme de simples requêtes afin d'accéder à ces valeurs. Elles pourront donc avoir cette case de cochée.

Entraînez-vous à créer des méthodes, il est toujours utile de pratiquer.

Maintenant que vous êtes prêts (enfin j'espère), nous allons leur ajouter des arguments. Pour cela, nous allons nous intéresser à cette partie de la fenêtre (voir la figure suivante).



Zone permettant de modifier les paramètres de la méthode

Encore une fois, nous remarquons des ressemblances avec tout ce que nous avons vu. En effet, nous retrouvons notre bloc blanc ainsi que ses quatre boutons, toujours fidèles au rendez-vous. Inutile de vous expliquer le rôle de chacun hein je crois  . Au cas où vous n'ayez pas deviné, la gestion des paramètres de la méthode s'effectue à cet endroit. 

Créez donc un nouveau paramètre. Les champs de droite, comme à leur habitude, se dégrisent. Je fais une brève description des fonctionnalités des champs, étant donné que c'est du déjà vu.

- Le champ **Nom** indique le nom de l'argument.
- Le champ **Type** spécifie le type de l'argument (entier, chaîne de caractères, tableau, etc.).
- Le champ **Val. par déf.** révèle la valeur par défaut de l'argument.
- Le champ **Commentaire** permet de laisser un petit commentaire concernant l'argument.
- L'option **Direction** est inutile.

Gestion du style de la classe

Si vous êtes sous Windows, vous avez peut-être remarqué que tous vos attributs et méthodes ont le même style, quels que soient leurs spécificités. Pour remédier à ce problème, il faut modifier le style de la classe en cliquant sur l'onglet **Style** (voir la figure suivante).


Fenêtre permettant de gérer le style de la classe

Je ne pense pas que de plus amples explications s'imposent, je ne ferais que répéter ce que la fenêtre vous affiche.

Pour modifier le style de toutes les classes, sélectionnez-les toutes (en cliquant sur chacune tout en maintenant la touche Shift) puis double-cliquez sur l'une d'elle. Modifiez le style dans la fenêtre ouverte, validez et admirez.

Je crois avoir fait le tour de toutes les fonctionnalités. Pour vous entraîner, vous pouvez essayer de reconstituer la classe `Personnage`.

Modéliser les interactions

Vous avez réussi à créer une belle classe avec plein d'attributs et de méthodes. Je vais vous demander d'en créer une autre afin de les faire interagir entre elles. Nous allons voir les quatre interactions abordées dans le précédent chapitre.

Création des liaisons

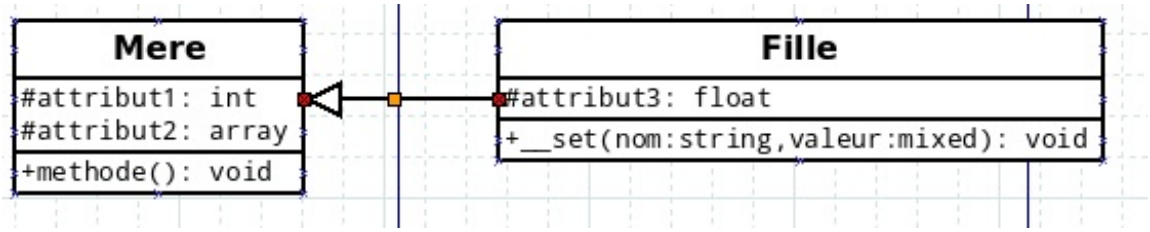
L'héritage

Nous allons créer notre première interaction : l'héritage entre deux classes. Pour cela, il faut cliquer sur l'icône représentant l'interaction de l'héritage (voir la figure suivante).

Modéliser un héritage

Ensuite, dirigez-vous sur la partie de droite contenant vos deux classes. Cliquez sur une croix bleue de la classe mère et, tout en maintenant le clic de la souris enfoncé, glissez jusqu'à une croix bleue de votre classe fille. Lorsque vous déplacez votre flèche, le curseur prend la forme d'une croix. Une fois qu'il survole une croix bleue, il est transformé en une autre croix représentant trois chemins qui se croisent, et la classe reliée se voit entourée d'un épais trait rouge : c'est à ce moment-là que vous pouvez lâcher le clic.

Normalement, vous devriez avoir obtenu ceci (voir la figure suivante).



Modélisation d'un héritage simple

Notez les deux points rouges sur la flèche ainsi que le point orange au milieu. Ceux-ci indiquent que la flèche est sélectionnée. Il se peut que l'un des points des extrémités de la flèche soit vert, auquel cas cela signifie que ce point n'est pas situé sur une croix bleue ! Si elle n'est pas sur une croix bleue, alors elle n'est reliée à **aucune classe**.
Le point orange du milieu ne bougera jamais (sauf si vous le déplacez manuellement). La flèche passera toujours par ce point. Ainsi, si vous voulez déplacer vos deux classes sans toucher à ce point, votre flèche fera une trajectoire assez étrange.

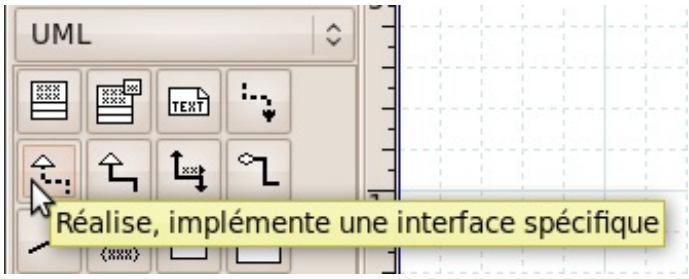
Alors, premières impressions ? Si vous êtes parvenus sans encombre à réaliser ce qu'on vient de faire, les quatre prochaines interactions seront toutes aussi simples !

Les interfaces

Passons maintenant à l'implémentation d'interfaces. Créez une classe banale et, histoire que les classes coordonnent avec

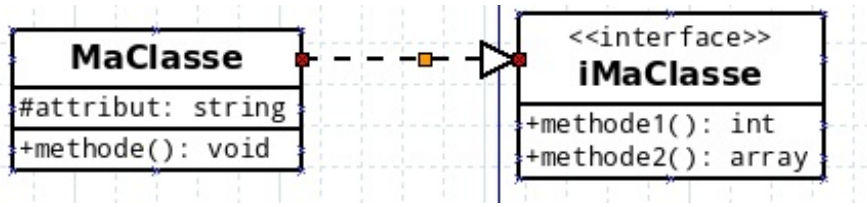
l'interaction, écrivez `interface` dans le champ texte **Stéréotype** de la classe. Pensez aussi à décocher la case **Attributs visibles**.

L'icône représentant l'interaction de l'implémentation d'interface est située juste à gauche de celui de l'héritage (voir la figure suivante).



Modéliser une implémentation d'interface

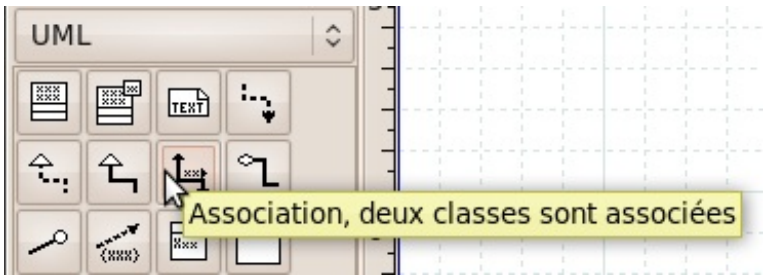
Comme pour la précédente interaction, cliquez sur une des croix bleues de l'interface, puis glissez la souris sur une croix bleue de la classe l'implémentant et relâchez la souris. Si tout s'est bien déroulé, vous devriez avoir obtenu ceci (voir figure suivante).



Exemple d'implémentation d'interface

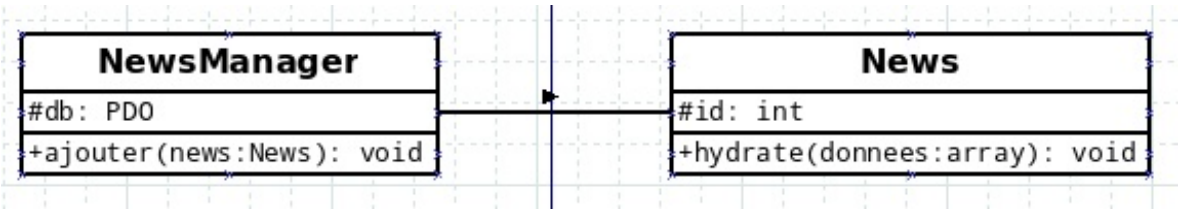
L'association

Voyons maintenant comment modéliser l'association. Cette icône est placée juste à droite de l'icône représentant l'héritage (voir la figure suivante).



Modéliser une association

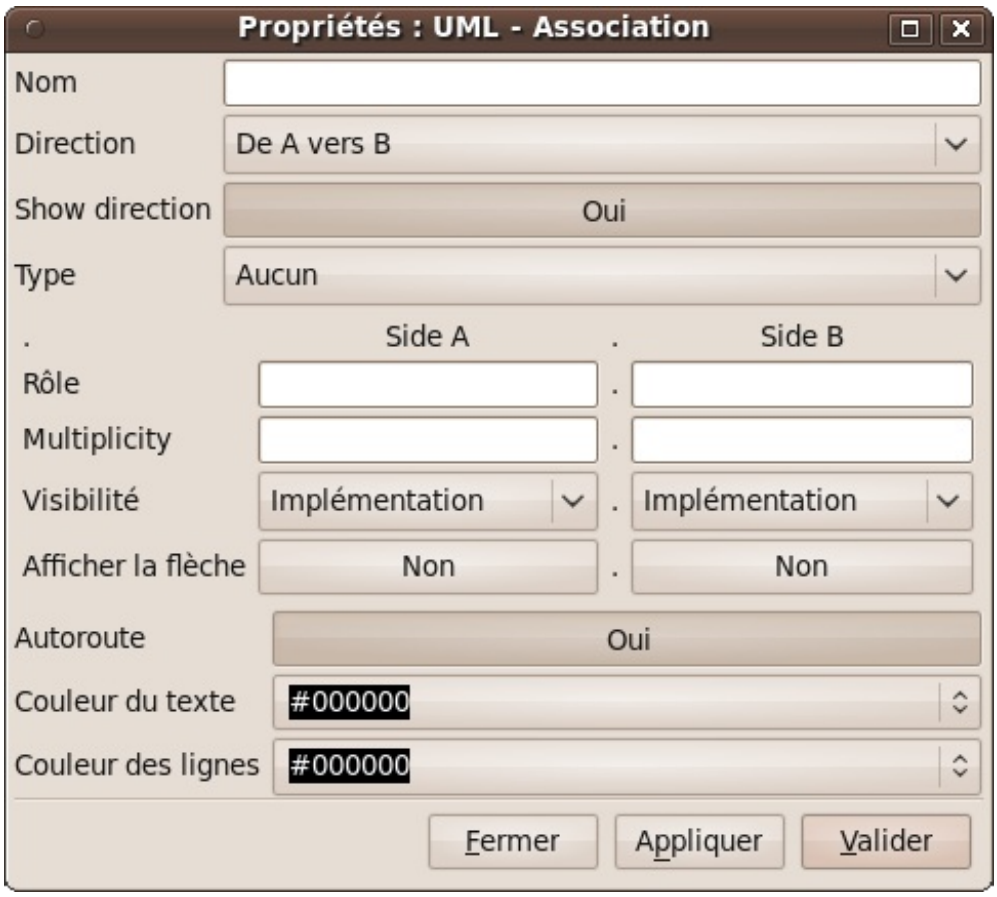
Cliquez donc sur la croix bleue de la classe ayant un attribut stockant une instance de l'autre classe, puis glissez sur cette autre classe. Vous devriez avoir ceci sous vos yeux (voir la figure suivante).



Première étape pour créer une association

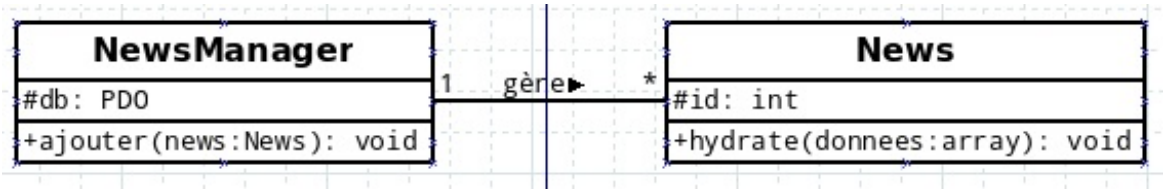
Cette fois-ci, les points rouges et le point orange n'apparaissent pas. Cela signifie simplement que la flèche n'est pas sélectionnée.

Voyons maintenant comment définir l'association et placer les cardinalités. En fait, ce sont des options de l'association que l'on peut modifier en accédant à la configuration de cette liaison. Pour cela, double-cliquez sur la ligne. Vous devriez avoir obtenu cette fenêtre (voir la figure suivante).



Paramétrer l'association

La définition de l'association se place dans le premier champ texte correspondant au nom. Mettez-y par exemple « gère ». Les cardinalités se placent dans les champs texte **Multiplicity**. La première colonne « Side A » correspond à la classe qui lie l'autre classe. Dans notre exemple, « Side A » correspond à `NewsManager` et « Side B » à `News`. Si vous vous êtes bien débrouillés, vous devriez avoir obtenu quelque chose dans ce genre (voir la figure suivante).

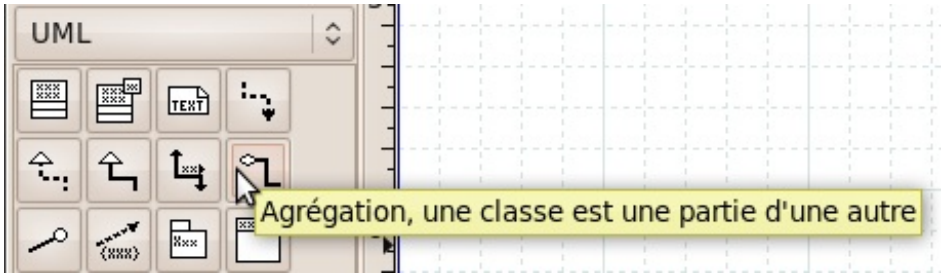


Exemple d'une association complète

L'agrégation

Intéressons-nous maintenant à la flèche représentant l'agrégation. On peut le faire de deux façons différentes. La première est la même que la précédente (j'entends par là que l'icône sur laquelle cliquer est la même). Cette fois-ci, il ne faut pas afficher le sens de la liaison et afficher un losange. Pour ce faire, il suffit de sélectionner **Aggregation** dans la liste déroulante **Type** et cliquer sur le gros bouton **Oui** de la ligne **Show direction**. La seconde solution consiste à utiliser une autre icône qui construira directement la flèche avec le losange. En réalité, il s'agit simplement d'un raccourci de la première solution.

Cette icône se trouve juste à droite de la précédente (voir la figure suivante).



Modéliser une agrégation

Je pense qu'il est inutile de vous dire comment lier vos deux classes étant donné que le principe reste le même.

La composition

Enfin, terminons par la composition. La composition n'a pas d'icône pour la représenter. Vous pouvez soit cliquer sur l'icône de l'association, soit cliquer sur l'icône de l'agrégation (cette dernière est à préférer dans le sens où une composition se rapproche beaucoup plus d'une agrégation que d'une association). Reliez vos classes comme nous l'avons fait jusqu'à maintenant puis double-cliquez sur celles-ci. Nous allons regarder la même liste déroulante que celle utilisée dans l'agrégation : je parle de la liste déroulante **Type**. À l'intérieur de celle-ci, choisissez l'option **Composition** puis validez.

Exercice

Vous avez désormais accumulé toutes les connaissances pour faire un diagramme complet. Je vais donc vous demander de me reconstituer le diagramme que je vous ai donné au début (voir la figure suivante).

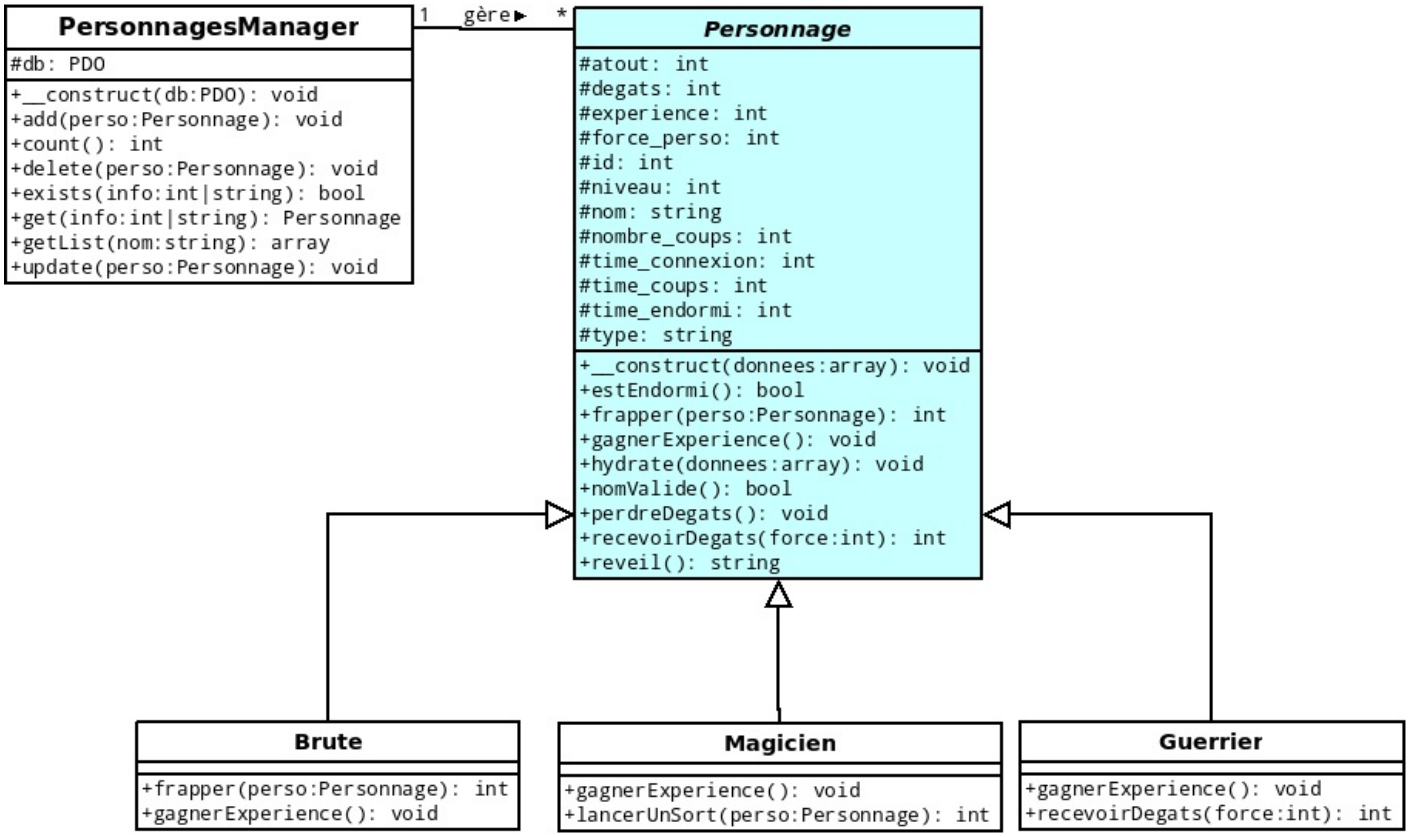


Diagramme modélisant le dernier TP

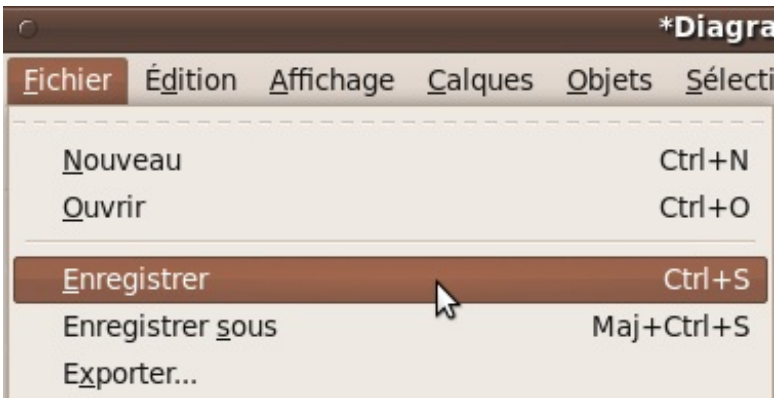
Exploiter son diagramme

Vous avez maintenant un diagramme conséquent avec un grand nombre d'interactions, mais qu'allez vous en faire ? Je vais ici vous expliquer comment exporter votre diagramme dans deux formats différents bien pratiques. L'un sera sous forme d'image, l'autre sous forme de code PHP.

Enregistrer son diagramme

Le chemin menant au dossier dans lequel vous allez enregistrer votre diagramme ne doit pas contenir d'espace ou de caractères spéciaux (lettres accentuées, signes spéciaux comme le ©, etc.). Je vous conseille donc (pour les utilisateurs de Windows), de créer un dossier à la racine de votre disque dur. S'il comporte des caractères spéciaux, vous ne pourrez pas double-cliquer dessus afin de le lancer (comme vous faites avec tout autre document), et s'il contient des espaces ou caractères spéciaux vous ne pourrez pas exporter par la suite votre diagramme sous forme de code PHP (et tout autre langage que propose le logiciel).

Avant toute exportation, il faut sauvegarder son diagramme, sinon cela ne fonctionnera pas. Pour ce faire, cliquez sur le menu **Fichier** puis cliquez sur **Enregistrer** (voir la figure suivante).

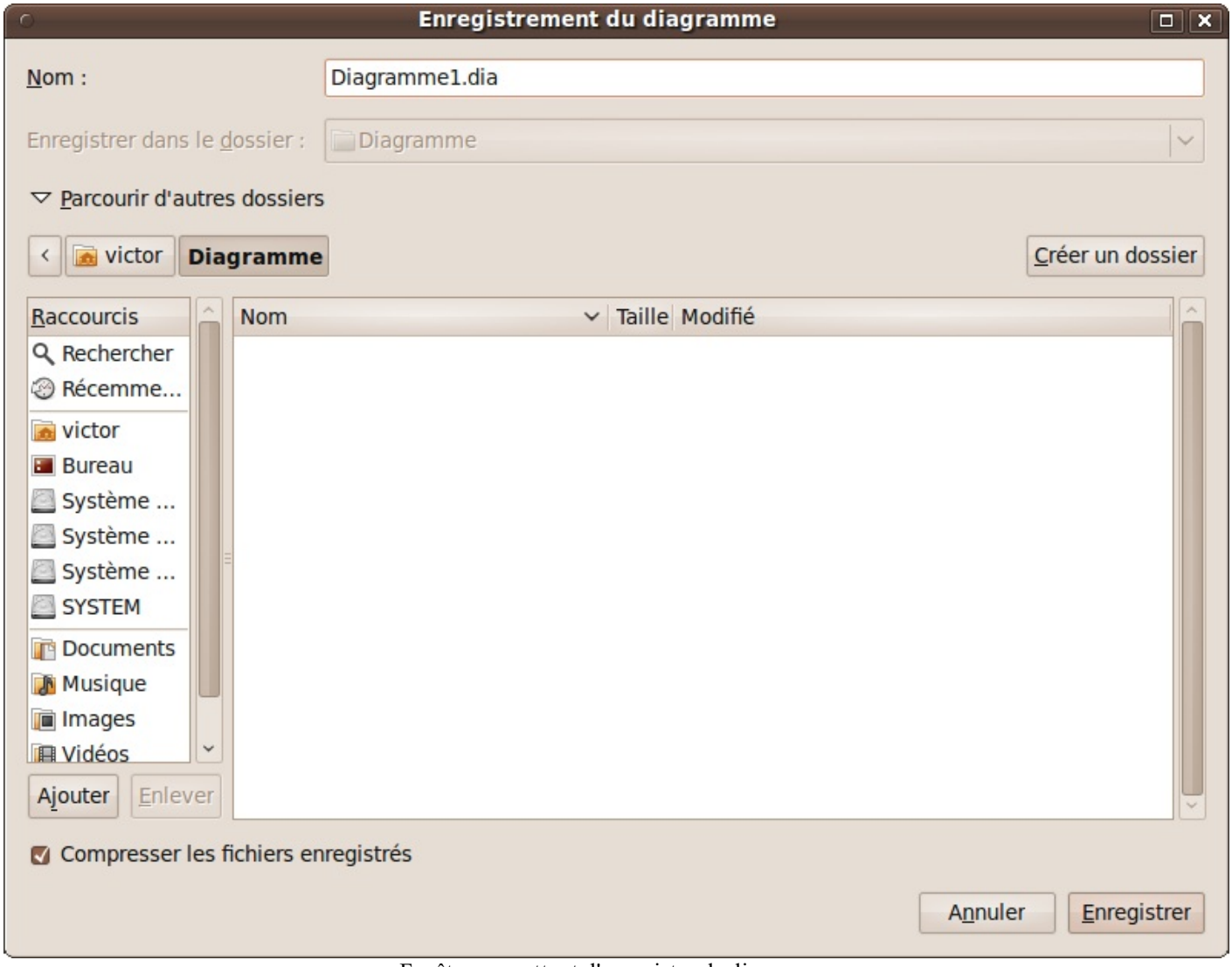


Enregistrer le diagramme par le menu

Ou cliquez sur cette icône (voir la figure suivante).

Enregistrer le diagramme par la barre d'outils

Cette fenêtre s'ouvre à vous (voir la figure suivante).



Fenêtre permettant d'enregistrer le diagramme

Tout est écrit en français, je ne pense pas avoir besoin d'expliquer en détails. Cette fenêtre contient un explorateur. À gauche sont listés les raccourcis menant aux dossiers les plus courants (le bureau, les documents, ainsi que les disques), tandis qu'à droite sont listés tous les fichiers et dossiers présents dans celui où vous vous trouvez (dans mon cas, le dossier est vierge).

Le champ texte situé tout en haut contient le nom du fichier sous lequel doit être enregistré le diagramme. Une fois que vous avez bien tout paramétré, cliquez sur **Enregistrer**. Votre diagramme est maintenant enregistré au format **.dia**.

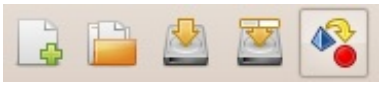
Exporter son diagramme

Sous forme d'image

Nous avons enregistré notre diagramme, nous sommes maintenant fin prêts à l'exporter. Pour obtenir une image de ce diagramme, nous allons cliquer sur **Exporter** dans le menu **Fichier** (voir la figure suivante).

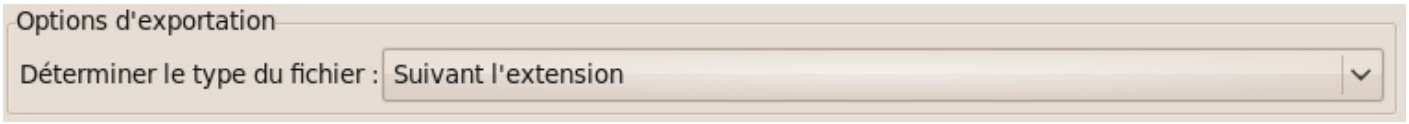
Exporter le diagramme par le menu

Ou cliquez sur cette icône (voir la figure suivante).



Exporter le diagramme par la barre d'outils

Une fenêtre semblable à celle de l'enregistrement du diagramme apparaît. Cependant, il y a une liste déroulante qui a fait son apparition (voir la figure suivante).



Déterminer le type du fichier

Pour exporter votre diagramme sous forme d'image, vous avez deux possibilités. Soit vous placez l'extension **.png** à la fin du nom de votre image en haut, soit, dans la liste déroulante, vous sélectionnez l'option **Pixbuf[png] (*.png)**. Avec cette deuxième solution, le champ texte du haut contiendra le nom de votre image avec l'extension **.png**. Cliquez sur **Enregistrer**, puis contemplez votre image générée. Pas mal, n'est-ce pas ?

Sous forme de code PHP

Au début, la marche à suivre reste la même, c'est-à-dire qu'il faut aller dans le menu **Fichier** puis cliquer sur **Exporter**. Cette fois-ci, nous allons choisir une autre option (logique). Vous pouvez là aussi réaliser l'opération de deux façons. Soit vous placez à la fin du nom du diagramme l'extension **.code**, soit vous sélectionnez **Filtre de transformation XSL (*.code)** dans la liste déroulante. Cliquez sur **Enregistrer**. Cette nouvelle fenêtre apparaît (voir la figure suivante).

Déterminer le langage dans lequel sera exporté le diagramme

Dans la première liste déroulante, choisissez **UML-CLASSES-EXTENDED**. Dans la seconde liste, choisissez **PHP5** puis cliquez sur **Valider**. Regardez les nouveaux fichiers générés dans votre dossier. Alors, ça en jette non ?

Si vous n'avez pas l'option **UML-CLASSES-EXTENDED** c'est que vous n'avez pas installé comme il faut l'extension **uml2php5**. Relisez donc bien la première partie du cours.

En résumé

- Dia est un logiciel permettant de créer facilement des diagrammes UML.
- Ce logiciel a l'avantage de pouvoir exporter vos diagrammes en images **et** en codes PHP.
- La modélisation grâce à UML est très appréciée dans le milieu professionnel, n'en ayez pas peur !

Les design patterns

Depuis les débuts de la programmation, tout un tas de développeurs ont rencontré différents problèmes de conception. La plupart de ces problèmes étaient récurrents. Pour éviter aux autres développeurs de buter sur le même souci, certains groupes de développeurs ont développé ce qu'on appelle des *design patterns* (ou *masques de conceptions* en français). Chaque design pattern répond à un problème précis. Comme nous le verrons dans ce chapitre, certains problèmes reviennent de façon récurrente et nous allons utiliser les moyens de conception déjà inventés pour les résoudre.

Ce chapitre est donc divisé en plusieurs sous-parties où chacune répond à un problème précis. Nous procèderons ainsi par étude de cas en posant le problème puis en le résolvant grâce aux moyens de conception connus.

Laisser une classe créant les objets : le pattern Factory

Le problème

Admettons que vous venez de créer une application relativement importante. Vous avez construit cette application en associant plus ou moins la plupart de vos classes entre elles. À présent, vous voudriez modifier un petit morceau de code afin d'ajouter une fonctionnalité à l'application. Problème : étant donné que la plupart de vos classes sont plus ou moins liées, il va falloir modifier un tas de chose ! Le pattern Factory pourra sûrement vous aider.

Ce motif est très simple à construire. En fait, si vous implémentez ce pattern, vous n'aurez plus de `new` à placer dans la partie globale du script afin d'instancier une classe. En effet, ce ne sera pas à vous de le faire mais à une **classe usine**. Cette classe aura pour rôle de charger les classes que vous lui passez en argument. Ainsi, quand vous modifierez votre code, vous n'aurez qu'à modifier le masque d'usine pour que la plupart des modifications prennent effet. En gros, vous ne vous soucierez plus de l'instanciation de vos classes, ce sera à l'usine de le faire !

Voici comment se présente une classe implémentant le pattern Factory :

Code : PHP

```php
<?php
class DBFactory
{
  public static function load($sgbdr)
  {
    $classe = 'SGBDR_' . $sgbdr;

    if (file_exists($chemin = $classe . '.class.php'))
    {
      require $chemin;
      return new $classe;
    }
    else
    {
      throw new RuntimeException('La classe <strong>' . $classe .
'</strong> n\'a pu être trouvée !');
    }
  }
}
?>
```

Dans votre script, vous pourrez donc faire quelque chose de ce genre :

Code : PHP

```php
<?php
try
{
  $mysql = DBFactory::load('MySQL');
}
catch (RuntimeException $e)
{
  echo $e->getMessage();
}
```

```
?>
```

Exemple concret

Le but est de créer une classe qui nous distribuera les objets PDO plus facilement. Nous allons partir du principe que vous avez plusieurs SGBDR, ou plusieurs BDD qui utilisent des identifiants différents. Pour résumer, nous allons tout centraliser dans une classe.

Code : PHP

```php
<?php
class PDOFactory
{
  public static function getMysqlConnexion()
  {
    $db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
    $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    return $db;
  }

  public static function getPgsqlConnexion()
  {
    $db = new PDO('pgsql:host=localhost;dbname=tests', 'root', '');
    $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    return $db;
  }
}
?>
```

Ceci vous simplifiera énormément la tâche. Si vous avez besoin de modifier vos identifiants de connexion, vous n'aurez pas à aller chercher dans tous vos scripts : tout sera placé dans notre factory.

Écouter ses objets : le pattern Observer

Le problème

Dans votre script est présente une classe qui s'occupe de la gestion d'un module. Lors d'une action précise, vous exécutez une ou plusieurs instructions. Celles-ci n'ont qu'une seule chose en commun : le fait qu'elles soient appelées car telle action s'est produite. Elles ont été placées dans la méthode *« parce qu'il faut bien les appeler et qu'on sait pas où les mettre »*. Il est intéressant dans ce cas-là de séparer les différentes actions effectuées lorsque telle action survient. Pour cela, nous allons regarder du côté du pattern Observer.

Le principe est simple : vous avez une classe observée et une ou plusieurs autre(s) classe(s) qui l'observe(nt). Lorsque telle action survient, vous allez prévenir toutes les classes qui l'observent. Nous allons, pour une raison d'homogénéité, utiliser les interfaces prédéfinies de la SPL. Il s'agit d'une librairie standard qui est fournie d'office avec PHP. Elle contient différentes classes, fonctions, interfaces, etc. Vous vous en êtes déjà servi en utilisant `spl_autoload_register()`.

Attardons nous plutôt sur ce qui nous intéresse, à savoir deux interfaces : SplSubject et SplObserver.

La première interface, `SplSubject`, est l'interface implémentée par l'objet observé. Elle contient trois méthodes :

- `attach (SplObserver $observer)` : méthode appelée pour ajouter une classe observatrice à notre classe observée.
- `detach (SplObserver $observer)` : méthode appelée pour supprimer une classe observatrice.
- `notify()` : méthode appelée lorsque l'on aura besoin de prévenir toutes les classes observatrices que quelque chose s'est produit.

L'interface `SplObserver` est l'interface implémentée par les différents observateurs. Elle ne contient qu'une seule méthode qui est celle appelée par la classe observée dans la méthode `notify()` : il s'agit de `update ( (SplSubject $subject))`.

Voici un diagramme mettant en œuvre ce design pattern (voir la figure suivante).

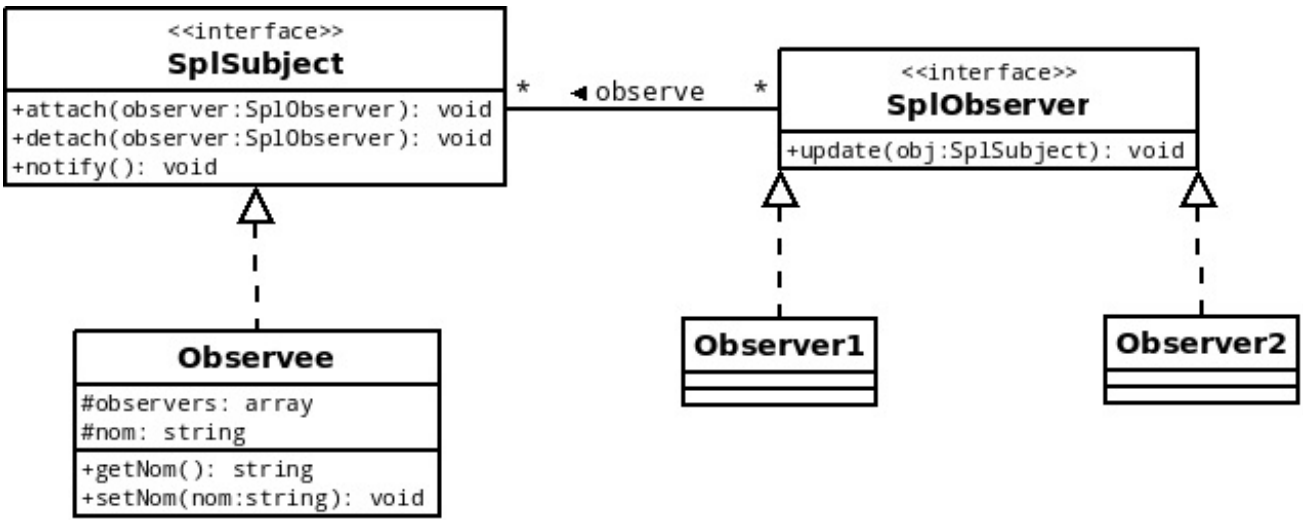


Diagramme modélisant une mise en place du design pattern Observer

Nous allons maintenant imaginer le code correspondant au diagramme. Commençons par la classe observée :

Code : PHP - Classe observée

```php
<?php
class Observee implements SplSubject
{
  // Ceci est le tableau qui va contenir tous les objets qui nous observent.
  protected $observers = array();

  // Dès que cet attribut changera on notifiera les classes observatrices.
  protected $nom;

  public function attach(SplObserver $observer)
  {
    $this->observers[] = $observer;
  }

  public function detach(SplObserver $observer)
  {
    if (is_int($key = array_search($observer, $this->observers, true)))
    {
      unset($this->observers[$key]);
    }
  }

  public function notify()
  {
    foreach ($this->observers as $observer)
    {
      $observer->update($this);
    }
  }

  public function getNom()
  {
    return $this->nom;
  }

  public function setNom($nom)
```

```
    {
      $this->nom = $nom;
      $this->notify();
    }
}
?>
```

Vous pouvez constater la présence du nom des interfaces en guise d'argument. Cela signifie que cet argument doit implémenter l'interface spécifiée.

Voici les deux classes observatrices :

Code : PHP - Classes observatrices

```
<?php
class Observer1 implements SplObserver
{
  public function update(SplSubject $obj)
  {
    echo __CLASS__, ' a été notifié ! Nouvelle valeur de l\'attribut
<strong>nom</strong> : ', $obj->getNom();
  }
}

class Observer2 implements SplObserver
{
  public function update(SplSubject $obj)
  {
    echo __CLASS__, ' a été notifié ! Nouvelle valeur de l\'attribut
<strong>nom</strong> : ', $obj->getNom();
  }
}
?>
```

Ces deux classes font exactement la même chose, ce n'était qu'à titre d'exemple basique que je vous ai donné ces exemples, afin que vous puissiez vous rendre compte de la syntaxe de base lors de l'utilisation du pattern Observer.

Pour tester nos classes, vous pouvez utiliser ce bout de code :

Code : PHP

```
<?php
$o = new Observee;
$o->attach(new Observer1); // Ajout d'un observateur.
$o->attach(new Observer2); // Ajout d'un autre observateur.
$o->setNom('Victor'); // On modifie le nom pour voir si les classes
observatrices ont bien été notifiées.
?>
```

Vous pouvez constater qu'ajouter des classes observatrices de cette façon peut être assez long si on en a cinq ou six. Il y a une petite technique qui consiste à pouvoir obtenir ce genre de code :

Code : PHP

```
<?php
$o = new Observee;
```

```
$o->attach(new Observer1)
  ->attach(new Observer2)
  ->attach(new Observer3)
  ->attach(new Observer4)
  ->attach(new Observer5);

$o->setNom('Victor'); // On modifie le nom pour voir si les classes
observatrices ont bien été notifiées.
?>
```

Pour effectuer ce genre de manœuvres, la méthode `attach()` doit retourner l'instance qui l'a appelé (en d'autres termes, elle doit retourner `$this`).

Exemple concret

Regardons un exemple concret à présent. Nous allons imaginer que vous avez, dans votre script, une classe gérant les erreurs générées par PHP. Lorsqu'une erreur est générée, vous aimeriez qu'il se passe deux choses :

- Que l'erreur soit enregistrée en BDD.
- Que l'erreur vous soit envoyée par mail.

Pour cela, vous pensez donc coder une classe comportant une méthode attrapant l'erreur et effectuant les deux opérations ci-dessus. Grave erreur ! Ceci est surtout à éviter : votre classe est chargée **d'intercepter** les erreurs, et non de les **gérer** ! Ce sera à d'autres classes de s'en occuper : ces classes vont observer la classe gérant l'erreur et une fois notifiée, elles vont effectuer l'action pour laquelle elles ont été conçues. Vous voyez un peu la tête qu'aura le script ?

Rappel : pour intercepter les erreurs, il vous faut utiliser `set_error_handler()`. Pour faire en sorte que la fonction de callback appelée lors de la génération d'une erreur soit une méthode d'une classe, passez un tableau à deux entrées en premier argument. La première entrée est l'objet sur lequel vous allez appeler la méthode, et la seconde est le nom de la méthode.

Vous êtes capables de le faire tout seul. Voici la correction.

ErrorHandler : classe gérant les erreurs

Code : PHP

```
<?php
class ErrorHandler implements SplSubject
{
  // Ceci est le tableau qui va contenir tous les objets qui nous
observent.
  protected $observers = array();

  // Attribut qui va contenir notre erreur formatée.
  protected $formatedError;

  public function attach(SplObserver $observer)
  {
    $this->observers[] = $observer;
    return $this;
  }

  public function detach(SplObserver $observer)
  {
    if (is_int($key = array_search($observer, $this->observers,
true)))
    {
```

```
      {
        unset($this->observers[$key]);
      }
    }

    public function getFormatedError()
    {
      return $this->formatedError;
    }

    public function notify()
    {
      foreach ($this->observers as $observer)
      {
        $observer->update($this);
      }
    }

    public function error($errno, $errstr, $errfile, $errline)
    {
      $this->formatedError = '[' . $errno . '] ' . $errstr . "\n" .
'Fichier : ' . $errfile . ' (ligne ' . $errline . ')';
      $this->notify();
    }
}
?>
```

MailSender : classe s'occupant d'envoyer les mails

Code : PHP

```
<?php
class MailSender implements SplObserver
{
  protected $mail;

  public function __construct($mail)
  {
    if (preg_match('`^[a-z0-9._-]+@[a-z0-9._-]{2,}\.[a-z]{2,4}$`',
$mail))
    {
      $this->mail = $mail;
    }
  }

  public function update(SplSubject $obj)
  {
    mail($this->mail, 'Erreur détectée !', 'Une erreur a été
détectée sur le site. Voici les informations de celle-ci : ' . "\n"
. $obj->getFormatedError());
  }
}
?>
```

BDDWriter : classe s'occupant de l'enregistrement en BDD

Code : PHP

```
<?php
```

```php
class BDDWriter implements SplObserver
{
  protected $db;

  public function __construct(PDO $db)
  {
    $this->db = $db;
  }

  public function update(SplSubject $obj)
  {
    $q = $this->db->prepare('INSERT INTO erreurs SET erreur =
:erreur');
    $q->bindValue(':erreur', $obj->getFormatedError());
    $q->execute();
  }
}
?>
```

Testons notre code !

Code : PHP

```php
<?php
$o = new ErrorHandler; // Nous créons un nouveau gestionnaire
d'erreur.
$db = PDOFactory::getMysqlConnexion();

$o->attach(new MailSender('login@fai.tld'))
  ->attach(new BDDWriter($db));

set_error_handler(array($o, 'error')); // Ce sera par la méthode
error() de la classe ErrorHandler que les erreurs doivent être
traitées.

5 / 0; // Générons une erreur
?>
```

D'accord, cela en fait du code ! Je ne sais pas si vous vous en rendez compte, mais ce que nous venons de créer là est une **excellente** manière de coder. Nous venons de séparer notre code comme il se doit et nous pourrons le modifier aisément car les différentes actions ont été séparées avec logique.

Séparer ses algorithmes : le pattern Strategy

Le problème

Vous avez une classe dédiée à une tâche spécifique. Dans un premier temps, celle-ci effectue une opération suivant un algorithme bien précis. Cependant, avec le temps, cette classe sera amenée à évoluer, et elle suivra plusieurs algorithmes, tout en effectuant la même tâche de base. Par exemple, vous avez une classe `FileWriter` qui a pour rôle d'écrire dans un fichier ainsi qu'une classe `DBWriter`. Dans un premier temps, ces classes ne contiennent qu'une méthode `write()` qui n'écrira que le texte passé en paramètre dans le fichier ou dans la BDD.
Au fil du temps, vous vous rendez compte que c'est dommage qu'elles ne fassent que ça et vous aimeriez bien qu'elles puissent écrire en différents formats (HTML, XML, etc.) : les classes doivent donc **formater** puis **écrire**. C'est à ce moment qu'il est intéressant de se tourner vers le pattern Strategy. En effet, sans ce design pattern, vous seriez obligés de créer deux classes différentes pour écrire au format HTML par exemple : `HTMLFileWriter` et `HTMLDBWriter`. Pourtant, ces deux classes devront formater le texte de la même façon : nous assisterons à une duplication du code, la pire chose à faire dans un script ! Imaginez que vous voulez modifier l'algorithme dupliqué une dizaine de fois... Pas très pratique n'est-ce pas ?

Exemple concret

Passons directement à l'exemple concret. Nous allons suivre l'idée que nous avons évoquée à l'instant : l'action d'écrire dans un fichier ou dans une BDD. Il y aura pas mal de classes à créer donc au lieu de vous faire un grand discours, je vais détailler le diagramme représentant l'application (voir la figure suivante).

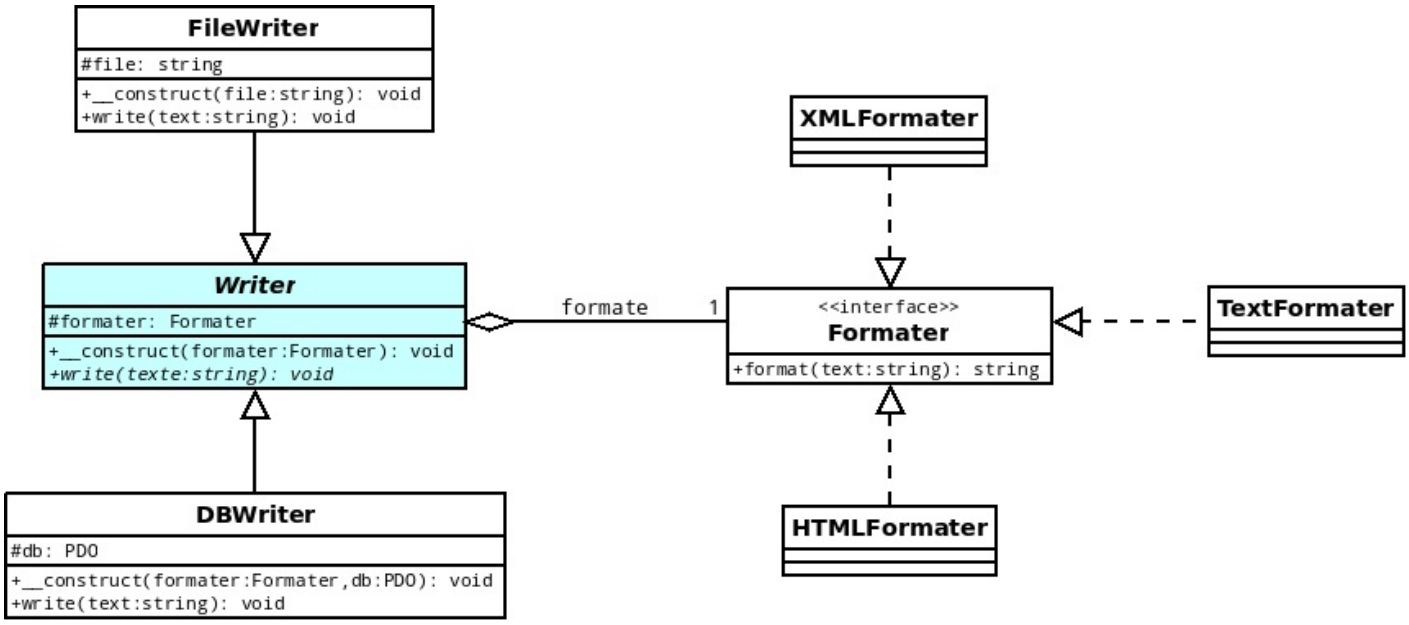


Diagramme modélisant une mise en place du design pattern Strategy

Ça en fait des classes ! Pourtant (je vous assure) le principe est très simple à comprendre. La classe `Writer` est abstraite (ça n'aurait aucun sens de l'instancier : on veut écrire, d'accord, mais sur quel support ?) et implémente un constructeur qui acceptera un argument : il s'agit du formateur que l'on souhaite utiliser. Nous allons aussi placer une méthode abstraite `write()`, ce qui forcera toutes les classes filles de `Writer` à implémenter cette méthode qui appellera la méthode `format()` du formateur associé (instance contenue dans l'attribut `$formater`) afin de récupérer le texte formaté. Allez, au boulot !

Commençons par l'interface. Rien de bien compliqué, elle ne contient qu'une seule méthode :

Code : PHP - Formater.interface.php

```php
<?php
interface Formater
{
  public function format($text);
}
?>
```

Ensuite vient la classe abstraite `Writer` que voici :

Code : PHP - Writer.class.php

```php
<?php
abstract class Writer
{
  // Attribut contenant l'instance du formateur que l'on veut
utiliser.
  protected $formater;

  abstract public function write($text);

  // Nous voulons une instance d'une classe implémentant Formater
en paramètre.
  public function __construct(Formater $formater)
  {
```

```
        $this->formater = $formater;
    }
}
?>
```

Nous allons maintenant créer deux classes héritant de `Writer` : `FileWriter` et `DBWriter`.

Code : PHP - DBWriter.class.php

```
<?php
class DBWriter extends Writer
{
  protected $db;

  public function __construct(Formater $formater, PDO $db)
  {
    parent::__construct($formater);
    $this->db = $db;
  }

  public function write ($text)
  {
    $q = $this->db->prepare('INSERT INTO lorem_ipsum SET text =
:text');
    $q->bindValue(':text', $this->formater->format($text));
    $q->execute();
  }
}
?>
```

Code : PHP - FileWriter.class.php

```
<?php
class FileWriter extends Writer
{
  // Attribut stockant le chemin du fichier.
  protected $file;

  public function __construct(Formater $formater, $file)
  {
    parent::__construct($formater);
    $this->file = $file;
  }

  public function write($text)
  {
    $f = fopen($this->file, 'w');
    fwrite($f, $this->formater->format($text));
    fclose($f);
  }
}
?>
```

Enfin, nous avons nos trois formateurs. L'un ne fait rien de particulier (`TextFormater`), et les deux autres formatent le texte en deux langages différents (`HTMLFormater` et `XMLFormater`). J'ai décidé d'ajouter le timestamp dans le formatage du texte histoire que le code ne soit pas complètement inutile (surtout pour la classe qui ne fait pas de formatage particulier). 

Code : PHP - TextFormater.class.php

```
<?php
```

```php
class TextFormater implements Formater
{
  public function format($text)
  {
    return 'Date : ' . time() . "\n" . 'Texte : ' . $text;
  }
}
?>
```

Code : PHP - HTMLFormater.class.php

```php
<?php
class HTMLFormater implements Formater
{
  public function format($text)
  {
    return '<p>Date : ' . time() . '<br />' ."\n". 'Texte : ' .
$text . '</p>';
  }
}
?>
```

Code : PHP - XMLFormater.class.php

```php
<?php
class XMLFormater implements Formater
{
  public function format($text)
  {
    return '<?xml version="1.0" encoding="ISO-8859-1"?>' ."\n".
           '<message>' ."\n".
           "\t". '<date>' . time() . '</date>' ."\n".
           "\t". '<texte>' . $text . '</texte>' ."\n".
           '</message>';
  }
}
?>
```

Et testons enfin notre code :

Code : PHP - index.php

```php
<?php
function autoload($class)
{
  if (file_exists($path = $class . '.class.php') ||
file_exists($path = $class . '.interface.php'))
  {
    require $path;
  }
}

spl_autoload_register('autoload');

$writer = new FileWriter(new HTMLFormater, 'file.html');
$writer->write('Hello world !');
?>
```

Ce code de base a l'avantage d'être très flexible. Il peut paraître un peu imposant pour notre utilisation, mais si l'application est

amenée à obtenir beaucoup de fonctionnalités supplémentaires, nous aurons déjà préparé le terrain !

Une classe, une instance : le pattern Singleton

Nous allons terminer par un pattern qui est en général le premier qu'on vous présente. Si je ne vous l'ai pas présenté au début c'est parce que je veux que vous fassiez attention en l'employant car il peut être très mal utilisé et se transformer en mauvaise pratique. On considérera alors le pattern comme un « anti-pattern ». Cependant, il est très connu et par conséquent il est essentiel de le connaître et de savoir pourquoi il ne faut pas l'utiliser dans certains contextes.

Le problème

Nous avons une classe qui ne doit être instanciée qu'une seule fois. À première vue, ça vous semble impossible et c'est normal. Jusqu'à présent, nous pouvions faire de multiples `$obj = new Classe;` jusqu'à l'infini, et nous nous retrouvions avec une infinité d'instances de `Classe`. Il va donc falloir empêcher ceci.

Pour empêcher la création d'une instance de cette façon, c'est très simple : il suffit de mettre le constructeur de la classe en privé ou en protégé !

T'es marrant toi, on ne pourra jamais créer d'instance avec cette technique !

Bien sûr que si ! Nous allons créer une instance de notre classe **à l'intérieur d'elle-même** ! De cette façon nous aurons accès au constructeur.

Oui mais voilà, il ne va falloir créer qu'une seule instance... Nous allons donc créer un attribut statique dans notre classe qui contiendra... l'instance de cette classe ! Nous aurons aussi une méthode statique qui aura pour rôle de renvoyer cette instance. Si on l'appelle pour la première fois, alors il faut instancier la classe puis retourner l'objet, sinon on se contente de le retourner.

Il reste un petit détail à régler. Nous voulons vraiment une seule instance, et là, il est encore possible d'en avoir plusieurs. En effet, rien n'empêche l'utilisateur de cloner l'instance ! Il faut donc bien penser à interdire l'accès à la méthode `__clone()`.

Ainsi, une classe implémentant le pattern Singleton ressemblerait à ceci :

Code : PHP

```php
<?php
class MonSingleton
{
  protected static $instance; // Contiendra l'instance de notre classe.

  protected function __construct() { } // Constructeur en privé.
  protected function __clone() { } // Méthode de clonage en privé aussi.

  public static function getInstance()
  {
    if (!isset(self::$instance)) // Si on n'a pas encore instancié notre classe.
    {
      self::$instance = new self; // On s'instancie nous-mêmes. :)
    }

    return self::$instance;
  }
}
?>
```

Ceci est le strict minimum. À vous d'implémenter de nouvelles méthodes comme vous l'auriez fait dans votre classe normale.

Voici donc une utilisation de la classe :

Code : PHP

```php
<?php
$obj = MonSingleton::getInstance(); // Premier appel : instance créée.
$obj->methode1();
?>
```

Exemple concret

Un exemple concret pour le pattern Singleton ? Non, désolé, nous allons devoir nous en passer. 

 Quoi ? Tu te moques de nous ? Alors il sert à rien ce design pattern ? 

Selon moi, non. Je n'ai encore jamais eu besoin de l'utiliser. Ce pattern doit être employé uniquement si plusieurs instanciations de la classe provoquaient un dysfonctionnement. Si le script peut continuer normalement alors que plusieurs instances sont créées, le pattern Singleton ne doit pas être utilisé.

 Donc ce que nous avons appris là, ça ne sert à rien ?

Non. Il est important de connaître ce design pattern, non pas pour l'utiliser, mais au contraire pour ne pas y avoir recours, et surtout savoir pourquoi. Cependant, avant de vous répondre, je vais vous présenter un autre pattern très important : **l'injection de dépendances**.

L'injection de dépendances

Comme tout pattern, celui-ci est né à cause d'un problème souvent rencontré par les développeurs : le fait d'avoir de nombreuses classes dépendantes les unes des autres. L'injection de dépendances consiste à découpler nos classes. Le pattern singleton que nous venons de voir favorise les dépendances, et l'injection de dépendances palliant ce problème, il est intéressant d'étudier ce nouveau pattern avec celui que nous venons de voir.

Soit le code suivant :

Code : PHP

```php
<?php
class NewsManager
{
  public function get($id)
  {
    // On admet que MyPDO étend PDO et qu'il implémente un singleton.
    $q = MyPDO::getInstance()->query('SELECT id, auteur, titre, contenu FROM news WHERE id = '.(int)$id);

    return $q->fetch(PDO::FETCH_ASSOC);
  }
}
```

Vous vous apercevez qu'ici, le singleton a introduit une dépendance entre deux classes n'appartenant pas au même module. Deux modules ne doivent jamais être liés de cette façon, ce qui est le cas dans cet exemple. Deux modules doivent être indépendants l'un de l'autre. D'ailleurs, en y regardant de plus près, cela ressemble fortement à une variable globale. En effet, un singleton n'est rien d'autre qu'une variable globale déguisée (il y a juste une étape en plus pour accéder à la variable) :

Code : PHP

```php
<?php
```

```php
class NewsManager
{
  public function get($id)
  {
    global $db;
    // Revient EXACTEMENT au même que :
    $db = MyPDO::getInstance();

    // Suite des opérations.
  }
}
```

Vous ne voyez pas où est le problème ? Souvenez-vous de l'un des points forts de la POO : le fait de pouvoir redistribuer sa classe ou la réutiliser. Dans le cas présent, c'est impossible, car notre classe `NewsManager` *dépend* de `MyPDO`. Qu'est-ce qui vous dit que la personne qui utilisera `NewsManager` aura cette dernière ? Rien du tout, et c'est normal. Nous sommes ici face à une dépendance créée par le singleton. De plus, la classe dépend aussi de PDO : il y avait donc déjà une dépendance au début, et le pattern Singleton en a créé une autre. Il faut donc supprimer ces deux dépendances.

Comment faire alors ?

Ce qu'il faut, c'est passer notre DAO au constructeur, sauf que notre classe ne doit pas être dépendante d'une quelconque bibliothèque. Ainsi, notre objet peut très bien utiliser PDO, MySQLi ou que sais-je encore, la classe se servant de lui doit fonctionner de la même manière. Alors comment procéder ? Il faut imposer un **comportement spécifique à notre objet** en l'obligeant à implémenter certaines méthodes. Je ne vous fais pas attendre : les interfaces sont là pour ça. Nous allons donc créer une interface `iDB` contenant (pour faire simple) qu'une seule méthode : `query()`.

Code : PHP

```php
<?php
interface iDB
{
  public function query($query);
}
```

Pour que l'exemple soit parlant, nous allons créer deux classes utilisant cette structure, l'une utilisant PDO et l'autre MySQLi. Cependant, un problème se pose : le résultat retourné par la méthode `query()` des classes PDO et MySQLi sont des instances de deux classes différentes, les méthodes disponibles ne sont, par conséquent, pas les mêmes. Il faut donc créer d'autres classes pour gérer les résultats qui suivent eux aussi une structure définie par une interface (admettons `iResult`).

Code : PHP

```php
<?php
interface iResult
{
  public function fetchAssoc();
}
```

Nous pouvons donc à présent écrire nos quatre classes : `MyPDO`, `MyMySQLi`, `MyPDOStatement` et `MyMySQLiResult`.

Code : PHP - MyPDO

```php
<?php
class MyPDO extends PDO implements iDB
{
  public function query($query)
  {
    return new MyPDOStatement(parent::query($query));
  }
```

```
}
```

Code : PHP - MyPDOStatement

```php
<?php
class MyPDOStatement implements iResult
{
  protected $st;

  public function __construct(PDOStatement $st)
  {
    $this->st = $st;
  }

  public function fetchAssoc()
  {
    return $this->st->fetch(PDO::FETCH_ASSOC);
  }
}
```

Code : PHP - MyMySQLi

```php
<?php
class MyMySQLi extends MySQLi implements iDB
{
  public function query($query)
  {
    return new MyMySQLiResult(parent::query($query));
  }
}
```

Code : PHP - MyMySQLiResult

```php
<?php
class MyMySQLiResult implements iResult
{
  protected $st;

  public function __construct(MySQLi_Result $st)
  {
    $this->st = $st;
  }

  public function fetchAssoc()
  {
    return $this->st->fetch_assoc();
  }
}
```

Nous pouvons donc maintenant écrire notre classe `NewsManager`. N'oubliez pas de vérifier que les objets sont bien des instances de classes implémentant les interfaces désirées. 

Code : PHP

```php
<?php
class NewsManager
{
  protected $dao;

  // On souhaite un objet instanciant une classe qui implémente
```

```
iDB.
  public function __construct(iDB $dao)
  {
    $this->dao = $dao;
  }

  public function get($id)
  {
    $q = $this->dao->query('SELECT id, auteur, titre, contenu FROM
news WHERE id = '.(int)$id);

    // On vérifie que le résultat implémente bien iResult.
    if (!$q instanceof iResult)
    {
      throw new Exception('Le résultat d\'une requête doit être un
objet implémentant iResult');
    }

    return $q->fetchAssoc();
  }
}
```

Testons maintenant notre code.

Code : PHP

```
<?php
$dao = new MyPDO('mysql:host=localhost;dbname=news', 'root', '');
// $dao = new MyMySQLi('localhost', 'root', '', 'news');

$manager = new NewsManager($dao);
print_r($manager->get(2));
```

Je vous laisse commenter les deux premières lignes pour vérifier que les deux fonctionnent. Après quelques tests, vous vous rendrez compte que nous avons bel et bien découplé nos classes ! Il n'y a ainsi plus aucune dépendance entre notre classe `NewsManager` et une quelconque autre classe.

Le problème, dans notre cas, c'est qu'il est difficile de faire de l'injection de dépendances pour qu'une classe supporte toutes les bibliothèques d'accès aux BDD (PDO, MySQLi, etc.) à cause des résultats des requêtes. De son côté, PDO a la classe `PDOStatement`, tandis que MySQLi a MySQLi_STMT pour les requêtes préparées et MySQLi_Result pour les résultats de requêtes classiques. Cela est donc difficile de les conformer au même modèle. Nous allons donc, dans le TP à venir, utiliser une autre technique pour découpler nos classes.

Pour conclure

Le principal problème du singleton est de favoriser les dépendances entre deux classes. Il faut donc être très méfiant de ce côté-là, car votre application deviendra difficilement modifiable et l'on perd alors les avantages de la POO. Je vous recommande donc d'utiliser le singleton en dernier recours : si vous décidez d'implémenter ce pattern, c'est pour garantir que cette classe ne doit être instanciée qu'une seule fois. Si vous vous rendez compte que deux instances ou plus ne causent pas de problème à l'application, alors n'implémentez pas le singleton. Et par pitié : **n'implémentez pas un singleton pour l'utiliser comme une variable globale** ! C'est la pire des choses à faire car cela favorise les dépendances entre classes comme nous l'avons vu.

Si vous voulez en savoir plus sur l'injection de dépendances (notamment sur l'utilisation de **conteneurs**), je vous invite à lire cet excellent tutoriel de vincent1870.

En résumé

- Un *design pattern* est un moyen de conception répondant à un problème récurrent.
- Le pattern *factory* a pour but de laisser des classes usine créer les instances à votre place.

- Le pattern *observer* permet de lier certains objets à des « écouteurs » eux-mêmes chargés de notifier les objets auxquels ils sont rattachés.
- Le pattern *strategy* sert à délocaliser la partie algorithmique d'une méthode afin de le permettre réutilisable, évitant ainsi la duplication de cet algorithme.
- Le pattern *singleton* permet de pouvoir instancier une classe une seule et unique fois, ce qui présente quelques soucis au niveau des dépendances entre classes.
- Le pattern *injection de dépendances* a pour but de rendre le plus indépendantes possible les classes.

TP : un système de news

La plupart des sites web dynamiques proposent un système d'actualités. Je sais par conséquent que c'est l'exemple auquel la plupart d'entre vous s'attend : nous allons donc réaliser ici un système de news ! Cela sera le dernier petit TP permettant de clarifier vos acquis avant celui plus important qui vous attend. Ce TP est aussi l'occasion pour vous de voir un exemple concret qui vous sera utile pour votre site !

Ce que nous allons faire

Cahier des charges

Commençons par définir ce que nous allons faire et surtout, ce dont nous allons avoir besoin.

Ce que nous allons réaliser est très simple, à savoir un système de news basique avec les fonctionnalités suivantes :

- Affichage des cinq premières news à l'accueil du site avec texte réduit à 200 caractères.
- Possibilité de cliquer sur le titre de la news pour la lire entièrement. L'auteur et la date d'ajout apparaîtront, ainsi que la date de modification si la news a été modifiée.
- Un espace d'administration qui permettra d'ajouter / modifier / supprimer des news. Cet espace tient sur une page : il y a un formulaire et un tableau en-dessous listant les news avec des liens modifier / supprimer. Quand on clique sur « Modifier », le formulaire se pré-remplit.

Pour réaliser cela, nous allons avoir besoin de créer une table **news** dont la structure est la suivante :

Code : SQL

```sql
CREATE TABLE `news` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `auteur` varchar(30) NOT NULL,
  `titre` varchar(100) NOT NULL,
  `contenu` text NOT NULL,
  `dateAjout` datetime NOT NULL,
  `dateModif` datetime NOT NULL,
  PRIMARY KEY (`id`)
);
```

Concernant l'organisation des classes, nous allons suivre la même structure que pour les personnages, à savoir :

- Une classe `News` qui contiendra les champs sous forme d'attributs. Son rôle sera de représenter une news.
- Une classe `NewsManager` qui gérera les news. C'est elle qui interagira avec la BDD.

Cependant, je voudrais vous enseigner quelque chose de nouveau. Je voudrais que la classe `NewsManager` ne soit pas dépendante de PDO. Nous avons vu dans le chapitre précédent que l'injection de dépendances pouvait être intéressante mais nous compliquait trop la tâche concernant l'adaptation des DAO. Au lieu d'injecter la dépendance, nous allons plutôt créer des classes **spécialisées**, c'est-à-dire qu'à chaque API correspondra une classe. Par exemple, si nous voulons assurer la compatibilité de notre système de news avec l'API PDO et MySQLi, alors nous aurons deux managers différents, l'un effectuant les requêtes avec PDO, l'autre avec MySQLi. Néanmoins, étant donné que ces classes ont une nature en commun (celle d'être toutes les deux des managers de news), alors elles devront hériter d'une classe représentant cette nature.

Voir le résultat que vous devez obtenir

Retour sur le traitement des résultats

Je vais vous demander d'essayer de vous rappeler le dernier TP, celui sur les personnages, et, plus précisément, la manière dont nous récupérions les résultats. Nous obtenions quelque chose comme ça :

Code : PHP

```php
<?php
// On admet que $db est une instance de PDO

$q = $db->prepare('SELECT id, nom, degats FROM personnages WHERE nom
<> :nom ORDER BY nom');
$q->execute(array(':nom' => $nom));

while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
{
  $persos[] = new Personnage($donnees);
}
```

Comme vous pouvez le constater, on récupère la liste des personnages sous forme de tableau grâce à la constante `PDO::FETCH_ASSOC`, puis on instancie notre classe `Personnage` en passant ce tableau au constructeur. Je vais vous dévoiler un moyen plus simple et davantage optimisé pour effectuer cette opération. je ne l'ai pas abordé précédemment car je trouvais qu'il était un peu tôt pour vous en parler dès la première partie.

Avec PDO (comme avec MySQLi, mais nous le verrons plus tard), il existe une constante qui signifie *« retourne-moi le résultat dans un objet »* (au même titre que la constante `PDO::FETCH_ASSOC` signifie *« retourne-moi le résultat dans un tableau »*). Cette constante est tout simplement `PDO::FETCH_CLASS`. Comment s'utilise-t-elle ? Comme vous vous en doutez peut-être, si nous voulons que PDO nous retourne les résultats sous forme d'instances de notre classe `News`, il va falloir lui dire ! Or, nous ne pouvons pas lui dire le nom de notre classe directement dans la méthode `fetch()` comme nous le faisons avec `PDO::FETCH_ASSOC`. Nous allons nous servir d'une autre méthode, `setFetchMode()`. Si vous avez lu les toutes premières lignes de la page pointée par le lien que j'ai donné, vous devriez savoir utiliser cette méthode :

Code : PHP

```php
<?php
// On admet que $db est une instance de PDO

$q = $db->prepare('SELECT id, nom, degats FROM personnages WHERE nom
<> :nom ORDER BY nom');
$q->execute(array(':nom' => $nom));

$q->setFetchMode(PDO::FETCH_CLASS, 'Personnage');

$persos = $q->fetchAll();
```

Notez qu'il n'est plus utile de parcourir chaque résultat pour les stocker dans un tableau puisque nous n'avons aucune opération à effectuer. Un simple appel à `fetchAll()` suffit donc amplement.

Comme vous le voyez, puisque nous avons dit à PDO à la ligne 7 que nous voulions une instance de `Personnage` en guise de résultat, il n'est pas utile de spécifier de nouveau quoi que ce soit lorsqu'on appelle la méthode `fetchAll()`.

Vous pouvez essayer ce code et vous verrez que ça fonctionne. Maintenant, je vais vous demander d'effectuer un petit test. Affichez la valeur des attributs dans le constructeur de `News` (avec des simples `echo`, n'allez pas chercher bien loin). Lancez de nouveau le script. Sur votre écran vont alors s'afficher les valeurs que PDO a assignées à notre objet. Rien ne vous titille ? Dois-je vous rappeler que, normalement, le constructeur d'une classe est appelé **en premier** et qu'il a pour rôle principal d'initialiser l'objet ? Pensez-vous que ce rôle est accompli alors que les valeurs ont été assignées **avant même que le constructeur soit appelé** ? Si vous aviez un constructeur qui devait assigner des valeurs par défaut aux attributs, celui-ci aurait écrasé les valeurs assignées par PDO.

Pour résumer, mémorisez qu'avec cette technique, l'objet n'est pas instancié comme il se doit. L'objet est créé (**sans que le constructeur soit appelé**), puis les valeurs sont assignées aux attributs par PDO, enfin le constructeur est appelé.

Pour remettre les choses dans l'ordre, une autre constante est disponible : `PDO::FETCH_PROPS_LATE`. Elle va de paire avec `PDO::FETCH_CLASS`. Essayez dès à présent cette modification :

Code : PHP

```php
<?php
// On admet que $db est une instance de PDO

$q = $db->prepare('SELECT id, nom, degats FROM personnages WHERE nom
<> :nom ORDER BY nom');
$q->execute(array(':nom' => $nom));

$q->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
'Personnage');

$persos = $q->fetchAll();
```

Affichez avec des `echo` la valeur de quelques attributs dans le constructeur et vous verrez qu'ils auront tous une valeur nulle, pour la simple et bonne raison que cette fois-ci, le constructeur a été appelé **avant** que les valeurs soient assignées aux attributs par PDO.

Maintenant, puisque le cahier des charges vous demande de rendre ce script compatible avec l'API MySQLi, je vais vous dire comment procéder. C'est beaucoup plus simple : au lieu d'appeler la méthode `fetch_assoc()` sur le résultat, appelez tout simplement la méthode `fetch_object('NomDeLaClasse')`.

Mes attributs sont privés ou protégés, pourquoi PDO et MySQLi peuvent-ils modifier leur valeur sans appeler les *setters* ?

Procéder de cette façon, c'est de la "bidouille" de bas niveau. PDO et MySQLi sont des API compilées avec PHP, elles ne fonctionnent donc pas comme les classes que vous développez. En plus de vous dire des bêtises, cela serait un hors sujet de vous expliquer le pourquoi du comment, mais sachez juste que les API compilées ont quelques super pouvoirs de ce genre.

Correction

Diagramme UML

Avant de donner une correction du code, je vais corriger la construction des classes en vous donnant le diagramme UML représentant le module (voir la figure suivante).

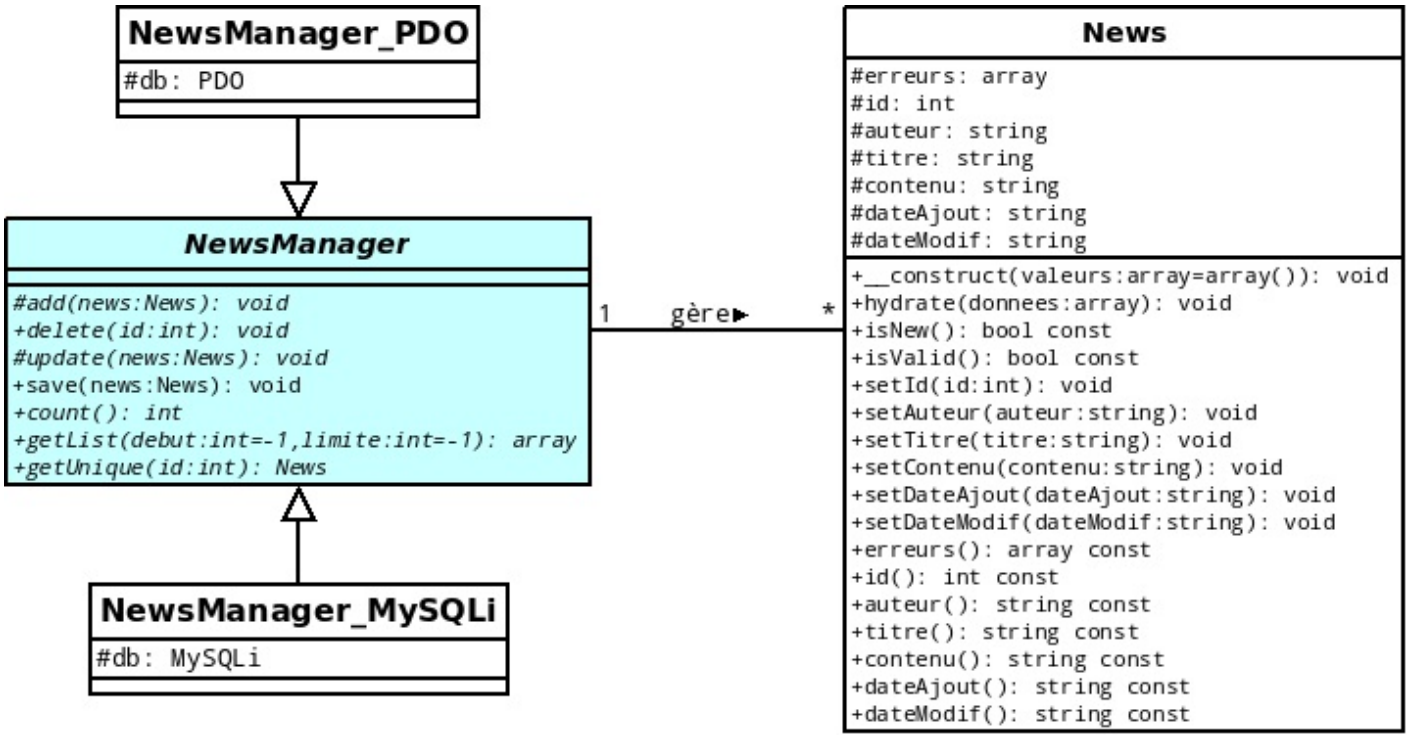


Diagramme modélisant le système de news

Le code du système

Pour des raisons d'organisation, j'ai décidé de placer les quatre classes dans un dossier **lib**.

Code : PHP - News.class.php

```php
<?php
/**
* Classe représentant une news, créée à l'occasion d'un TP du
tutoriel « La programmation orientée objet en PHP » disponible sur
http://www.siteduzero.com/
* @author Victor T.
* @version 2.0
*/
class News
{
  protected $erreurs = array(),
            $id,
            $auteur,
            $titre,
            $contenu,
            $dateAjout,
            $dateModif;

  /**
* Constantes relatives aux erreurs possibles rencontrées lors de
l'exécution de la méthode.
*/
  const AUTEUR_INVALIDE = 1;
  const TITRE_INVALIDE = 2;
  const CONTENU_INVALIDE = 3;


  /**
* Constructeur de la classe qui assigne les données spécifiées en
paramètre aux attributs correspondants.
* @param $valeurs array Les valeurs à assigner
* @return void
*/
  public function __construct($valeurs = array())
  {
    if (!empty($valeurs)) // Si on a spécifié des valeurs, alors on
hydrate l'objet.
    {
      $this->hydrate($valeurs);
    }
  }

  /**
* Méthode assignant les valeurs spécifiées aux attributs
correspondant.
* @param $donnees array Les données à assigner
* @return void
*/
  public function hydrate($donnees)
  {
    foreach ($donnees as $attribut => $valeur)
    {
      $methode = 'set'.ucfirst($attribut);

      if (is_callable(array($this, $methode)))
      {
        $this->$methode($valeur);
      }
    }
  }

  /**
* Méthode permettant de savoir si la news est nouvelle.
* @return bool
*/
```

```
  public function isNew()
  {
    return empty($this->id);
  }

  /**
* Méthode permettant de savoir si la news est valide.
* @return bool
*/
  public function isValid()
  {
    return !(empty($this->auteur) || empty($this->titre) ||
empty($this->contenu));
  }


  // SETTERS //

  public function setId($id)
  {
    $this->id = (int) $id;
  }

  public function setAuteur($auteur)
  {
    if (!is_string($auteur) || empty($auteur))
    {
      $this->erreurs[] = self::AUTEUR_INVALIDE;
    }
    else
    {
      $this->auteur = $auteur;
    }
  }

  public function setTitre($titre)
  {
    if (!is_string($titre) || empty($titre))
    {
      $this->erreurs[] = self::TITRE_INVALIDE;
    }
    else
    {
      $this->titre = $titre;
    }
  }

  public function setContenu($contenu)
  {
    if (!is_string($contenu) || empty($contenu))
    {
      $this->erreurs[] = self::CONTENU_INVALIDE;
    }
    else
    {
      $this->contenu = $contenu;
    }
  }

  public function setDateAjout($dateAjout)
  {
    if (is_string($dateAjout) && preg_match('`le [0-9]{2}/[0-
9]{2}/[0-9]{4} à [0-9]{2}h[0-9]{2}`', $dateAjout))
    {
      $this->dateAjout = $dateAjout;
    }
  }

  public function setDateModif($dateModif)
  {
```

```
      if (is_string($dateModif) && preg_match('`le [0-9]{2}/[0-
9]{2}/[0-9]{4} à [0-9]{2}h[0-9]{2}`', $dateModif))
      {
        $this->dateModif = $dateModif;
      }
    }

    // GETTERS //

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

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

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

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

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

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

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

Code : PHP - NewsManager.class.php

```
<?php
abstract class NewsManager
{
  /**
* Méthode permettant d'ajouter une news.
* @param $news News La news à ajouter
* @return void
*/
  abstract protected function add(News $news);

  /**
* Méthode renvoyant le nombre de news total.
* @return int
*/
  abstract public function count();

  /**
* Méthode permettant de supprimer une news.
* @param $id int L'identifiant de la news à supprimer
* @return void
*/
```

```
  abstract public function delete($id);

  /**
* Méthode retournant une liste de news demandée.
* @param $debut int La première news à sélectionner
* @param $limite int Le nombre de news à sélectionner
* @return array La liste des news. Chaque entrée est une instance
de News.
*/
  abstract public function getList($debut = -1, $limite = -1);

  /**
* Méthode retournant une news précise.
* @param $id int L'identifiant de la news à récupérer
* @return News La news demandée
*/
  abstract public function getUnique($id);

  /**
* Méthode permettant d'enregistrer une news.
* @param $news News la news à enregistrer
* @see self::add()
* @see self::modify()
* @return void
*/
  public function save(News $news)
  {
    if ($news->isValid())
    {
      $news->isNew() ? $this->add($news) : $this->update($news);
    }
    else
    {
      throw new RuntimeException('La news doit être valide pour être
enregistrée');
    }
  }

  /**
* Méthode permettant de modifier une news.
* @param $news news la news à modifier
* @return void
*/
  abstract protected function update(News $news);
}
```

Code : PHP - NewsManager_PDO.class.php

```
<?php
class NewsManager_PDO extends NewsManager
{
  /**
* Attribut contenant l'instance représentant la BDD.
* @type PDO
*/
  protected $db;

  /**
* Constructeur étant chargé d'enregistrer l'instance de PDO dans
l'attribut $db.
* @param $db PDO Le DAO
* @return void
*/
  public function __construct(PDO $db)
  {
    $this->db = $db;
  }
```

```
  /**
* @see NewsManager::add()
*/
  protected function add(News $news)
  {
    $requete = $this->db->prepare('INSERT INTO news SET auteur =
:auteur, titre = :titre, contenu = :contenu, dateAjout = NOW(),
dateModif = NOW()');

    $requete->bindValue(':titre', $news->titre());
    $requete->bindValue(':auteur', $news->auteur());
    $requete->bindValue(':contenu', $news->contenu());

    $requete->execute();
  }

  /**
* @see NewsManager::count()
*/
  public function count()
  {
    return $this->db->query('SELECT COUNT(*) FROM news')-
>fetchColumn();
  }

  /**
* @see NewsManager::delete()
*/
  public function delete($id)
  {
    $this->db->exec('DELETE FROM news WHERE id = '.(int) $id);
  }

  /**
* @see NewsManager::getList()
*/
  public function getList($debut = -1, $limite = -1)
  {
    $sql = 'SELECT id, auteur, titre, contenu, DATE_FORMAT
(dateAjout, \'le %d/%m/%Y à %Hh%i\') AS dateAjout, DATE_FORMAT
(dateModif, \'le %d/%m/%Y à %Hh%i\') AS dateModif FROM news ORDER BY
id DESC';

    // On vérifie l'intégrité des paramètres fournis.
    if ($debut != -1 || $limite != -1)
    {
      $sql .= ' LIMIT '.(int) $limite.' OFFSET '.(int) $debut;
    }

    $requete = $this->db->query($sql);
    $requete->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
'News');

    $listeNews = $requete->fetchAll();

    $requete->closeCursor();

    return $listeNews;
  }

  /**
* @see NewsManager::getUnique()
*/
  public function getUnique($id)
  {
    $requete = $this->db->prepare('SELECT id, auteur, titre,
contenu, DATE_FORMAT (dateAjout, \'le %d/%m/%Y à %Hh%i\') AS
dateAjout, DATE_FORMAT (dateModif, \'le %d/%m/%Y à %Hh%i\') AS
dateModif FROM news WHERE id = :id');
    $requete->bindValue(':id', (int) $id, PDO::PARAM_INT);
```

```
    $requete->execute();

    $requete->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
'News');

    return $requete->fetch();
  }

  /**
* @see NewsManager::update()
*/
  protected function update(News $news)
  {
    $requete = $this->db->prepare('UPDATE news SET auteur = :auteur,
titre = :titre, contenu = :contenu, dateModif = NOW() WHERE id =
:id');

    $requete->bindValue(':titre', $news->titre());
    $requete->bindValue(':auteur', $news->auteur());
    $requete->bindValue(':contenu', $news->contenu());
    $requete->bindValue(':id', $news->id(), PDO::PARAM_INT);

    $requete->execute();
  }
}
```

Code : PHP - NewsManager_MySQLi.class.php

```
<?php
class NewsManager_MySQLi extends NewsManager
{
  /**
* Attribut contenant l'instance représentant la BDD.
* @type MySQLi
*/
  protected $db;

  /**
* Constructeur étant chargé d'enregistrer l'instance de MySQLi dans
l'attribut $db.
* @param $db MySQLi Le DAO
* @return void
*/
  public function __construct(MySQLi $db)
  {
    $this->db = $db;
  }

  /**
* @see NewsManager::add()
*/
  protected function add(News $news)
  {
    $requete = $this->db->prepare('INSERT INTO news SET auteur = ?,
titre = ?, contenu = ?, dateAjout = NOW(), dateModif = NOW()');

    $requete->bind_param('sss', $news->auteur(), $news->titre(),
$news->contenu());

    $requete->execute();
  }

  /**
* @see NewsManager::count()
*/
  public function count()
  {
    return $this->db->query('SELECT id FROM news')->num_rows;
```

```
  }

  /**
* @see NewsManager::delete()
*/
  public function delete($id)
  {
    $id = (int) $id;

    $requete = $this->db->prepare('DELETE FROM news WHERE id = ?');

    $requete->bind_param('i', $id);

    $requete->execute();
  }

  /**
* @see NewsManager::getList()
*/
  public function getList($debut = -1, $limite = -1)
  {
    $listeNews = array();

    $sql = 'SELECT id, auteur, titre, contenu, DATE_FORMAT
(dateAjout, \'le %d/%m/%Y à %Hh%i\') AS dateAjout, DATE_FORMAT
(dateModif, \'le %d/%m/%Y à %Hh%i\') AS dateModif FROM news ORDER BY
id DESC';

    // On vérifie l'intégrité des paramètres fournis.
    if ($debut != -1 || $limite != -1)
    {
      $sql .= ' LIMIT '.(int) $limite.' OFFSET '.(int) $debut;
    }

    $requete = $this->db->query($sql);

    while ($news = $requete->fetch_object('News'))
    {
      $listeNews[] = $news;
    }

    return $listeNews;
  }

  /**
* @see NewsManager::getUnique()
*/
  public function getUnique($id)
  {
    $id = (int) $id;

    $requete = $this->db->prepare('SELECT id, auteur, titre,
contenu, DATE_FORMAT (dateAjout, \'le %d/%m/%Y à %Hh%i\') AS
dateAjout, DATE_FORMAT (dateModif, \'le %d/%m/%Y à %Hh%i\') AS
dateModif FROM news WHERE id = ?');
    $requete->bind_param('i', $id);
    $requete->execute();

    $requete->bind_result($id, $auteur, $titre, $contenu,
$dateAjout, $dateModif);

    $requete->fetch();

    return new News(array(
      'id' => $id,
      'auteur' => $auteur,
      'titre' => $titre,
      'contenu' => $contenu,
      'dateAjout' => $dateAjout,
      'dateModif' => $dateModif
```

```
        ));
    }

    /**
  * @see NewsManager::update()
  */
    protected function update(News $news)
    {
      $requete = $this->db->prepare('UPDATE news SET auteur = ?, titre
= ?, contenu = ?, dateModif = NOW() WHERE id = ?');

      $requete->bind_param('sssi', $news->auteur(), $news->titre(),
$news->contenu(), $news->id());

      $requete->execute();
    }
  }
```

Pour accéder aux instances de PDO et MySQLi, nous allons nous aider du design pattern factory. Veuillez donc créer une simple classe `DBFactory`.

Code : PHP - DBFactory.class.php

```
<?php
class DBFactory
{
  public static function getMysqlConnexionWithPDO()
  {
    $db = new PDO('mysql:host=localhost;dbname=news', 'root', '');
    $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    return $db;
  }

  public static function getMysqlConnexionWithMySQLi()
  {
    return new MySQLi('localhost', 'root', '', 'news');
  }
}
```

Nous allons créer deux pages : **index.php** qui sera accessible au grand public et listera les news, ainsi que **admin.php** qui nous permettra de gérer les news. Dans ces deux pages, nous aurons besoin d'un autoload. Nous allons donc créer un fichier **autoload.inc.php** dans le dossier **lib** qui contiendra notre autoload. Il s'agit d'un simple fichier, voyez par vous-même :

Code : PHP - autoload.inc.php

```
<?php
function autoload($classname)
{
  if (file_exists($file = dirname (__FILE__) . '/' . $classname .
'.class.php'))
  {
    require $file;
  }
}

spl_autoload_register('autoload');
```

Maintenant que nous avons créé la partie interne, nous allons nous occuper des pages qui s'afficheront devant vos yeux. Il s'agit bien entendu de la partie la plus facile, le pire est derrière nous.

Commençons par la page d'administration :

Code : PHP - admin.php

```php
<?php
require 'lib/autoload.inc.php';

$db = DBFactory::getMysqlConnexionWithMySQLi();
$manager = new NewsManager_MySQLi($db);

if (isset($_GET['modifier']))
{
  $news = $manager->getUnique ((int) $_GET['modifier']);
}

if (isset($_GET['supprimer']))
{
  $manager->delete((int) $_GET['supprimer']);
  $message = 'La news a bien été supprimée !';
}

if (isset($_POST['auteur']))
{
  $news = new News(
    array(
      'auteur' => $_POST['auteur'],
      'titre' => $_POST['titre'],
      'contenu' => $_POST['contenu']
    )
  );

  if (isset($_POST['id']))
  {
    $news->setId($_POST['id']);
  }

  if ($news->isValid())
  {
    $manager->save($news);

    $message = $news->isNew() ? 'La news a bien été ajoutée !' : 'La
news a bien été modifiée !';
  }
  else
  {
    $erreurs = $news->erreurs();
  }
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>Administration</title>
    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />

    <style type="text/css">
      table, td {
        border: 1px solid black;
      }

      table {
        margin:auto;
        text-align: center;
        border-collapse: collapse;
      }
```

```
        td {
          padding: 3px;
        }
      </style>
    </head>

    <body>
      <p><a href=".">Accéder à l'accueil du site</a></p>

      <form action="admin.php" method="post">
        <p style="text-align: center">
<?php
if (isset($message))
{
  echo $message, '<br />';
}
?>
          <?php if (isset($erreurs) && in_array(News::AUTEUR_INVALIDE,
$erreurs)) echo 'L\'auteur est invalide.<br />'; ?>
          Auteur : <input type="text" name="auteur" value="<?php if
(isset($news)) echo $news->auteur(); ?>" /><br />

          <?php if (isset($erreurs) && in_array(News::TITRE_INVALIDE,
$erreurs)) echo 'Le titre est invalide.<br />'; ?>
          Titre : <input type="text" name="titre" value="<?php if
(isset($news)) echo $news->titre(); ?>" /><br />

          <?php if (isset($erreurs) &&
in_array(News::CONTENU_INVALIDE, $erreurs)) echo 'Le contenu est
invalide.<br />'; ?>
          Contenu :<br /><textarea rows="8" cols="60"
name="contenu"><?php if (isset($news)) echo $news->contenu(); ?
></textarea><br />
<?php
if(isset($news) && !$news->isNew())
{
?>
          <input type="hidden" name="id" value="<?php echo $news-
>id(); ?>" />
          <input type="submit" value="Modifier" name="modifier" />
<?php
}
else
{
?>
          <input type="submit" value="Ajouter" />
<?php
}
?>
        </p>
      </form>

      <p style="text-align: center">Il y a actuellement <?php echo
$manager->count(); ?> news. En voici la liste :</p>

      <table>
        <tr><th>Auteur</th><th>Titre</th><th>Date
d'ajout</th><th>Dernière modification</th><th>Action</th></tr>
<?php
foreach ($manager->getList() as $news)
{
  echo '<tr><td>', $news->auteur(), '</td><td>', $news->titre(),
'</td><td>', $news->dateAjout(), '</td><td>', ($news->dateAjout() ==
$news->dateModif() ? '-' : $news->dateModif()), '</td><td><a href="?
modifier=', $news->id(), '">Modifier</a> | <a href="?supprimer=',
$news->id(), '">Supprimer</a></td></tr>', "\n";
}
?>
      </table>
    </body>
```

```
</html>
```

Et enfin, la partie visible à tous vos visiteurs :

Code : PHP - index.php

```
<?php
require 'lib/autoload.inc.php';

$db = DBFactory::getMysqlConnexionWithMySQLi();
$manager = new NewsManager_MySQLi($db);
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>Accueil du site</title>
    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
  </head>

  <body>
    <p><a href="admin.php">Accéder à l'espace
d'administration</a></p>
<?php
if (isset($_GET['id']))
{
  $news = $manager->getUnique((int) $_GET['id']);

  echo '<p>Par <em>', $news->auteur(), '</em>, ', $news-
>dateAjout(), '</p>', "\n",
       '<h2>', $news->titre(), '</h2>', "\n",
       '<p>', nl2br($news->contenu()), '</p>', "\n";

  if ($news->dateAjout() != $news->dateModif())
  {
    echo '<p style="text-align: right;"><small><em>Modifiée ',
$news->dateModif(), '</em></small></p>';
  }
}

else
{
  echo '<h2 style="text-align:center">Liste des 5 dernières
news</h2>';

  foreach ($manager->getList(0, 5) as $news)
  {
    if (strlen($news->contenu()) <= 200)
    {
      $contenu = $news->contenu();
    }

    else
    {
      $debut = substr($news->contenu(), 0, 200);
      $debut = substr($debut, 0, strrpos($debut, ' ')) . '...';

      $contenu = $debut;
    }

    echo '<h4><a href="?id=', $news->id(), '">', $news->titre(),
'</a></h4>', "\n",
         '<p>', nl2br($contenu), '</p>';
  }
}
```

```
    ?>
      </body>
    </html>
```

Partie 3 : [Pratique] Réalisation d'un site web

Description de l'application

Il serait temps de faire un TP conséquent pour mettre en pratique tout ce que nous avons vu : nous allons réaliser un site web (souvent comparé à une **application**). En effet, vous avez maintenant toutes les connaissances nécessaires pour réaliser une tel projet. Ce sera donc l'occasion de faire un point sur vos connaissances en **orienté objet**. Si vous pensez que vous êtes à l'aise avec l'orienté objet, ce TP sera l'occasion de vérifier cela. Si vous avez toujours du mal, alors ce TP vous aidera de façon très concrète.

Pour y arriver, nous allons créer une **bibliothèque** (terme que nous allons définir dès ce premier chapitre), un **module** de news avec commentaires ainsi qu'un **espace d'administration** complet. Le plus difficile consiste à développer notre bibliothèque. Au lieu de nous lancer tête baissée dans sa création, nous allons réfléchir sur ce que c'est, à quoi elle doit nous servir, etc. Nous ne développerons cette bibliothèque que dans le prochain chapitre.

Ceci est une étape importante qu'il ne faut surtout pas négliger. Ne lisez donc pas ce chapitre *vite fait, bien fait*, sinon vous risquez d'être perdus par la suite. Par ailleurs, ne vous découragez pas si vous ne comprenez pas tout d'un coup : ce chapitre est d'un niveau plus élevé que les précédents, il est donc normal de ne pas tout comprendre dès la première lecture. Ne vous découragez pas, vous allez y arriver ! 

Une application, qu'est ce que c'est ?
Le déroulement d'une application

Avant de commencer à créer une application, encore faudrait-il savoir ce que c'est et en quoi cela consiste. En fait, il faut décomposer le déroulement des actions effectuées du début à la fin (le début étant la requête envoyée par le client et la fin étant la réponse renvoyée à ce client). De manière très schématique, voici le déroulement d'une application (voir la figure suivante).

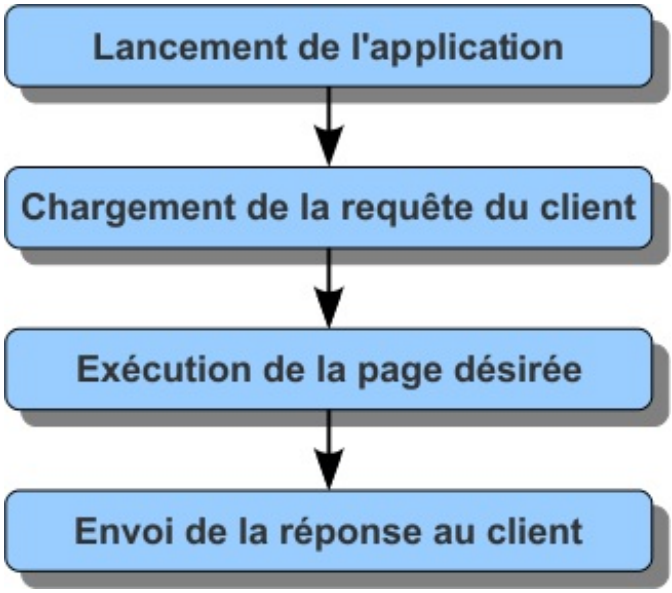


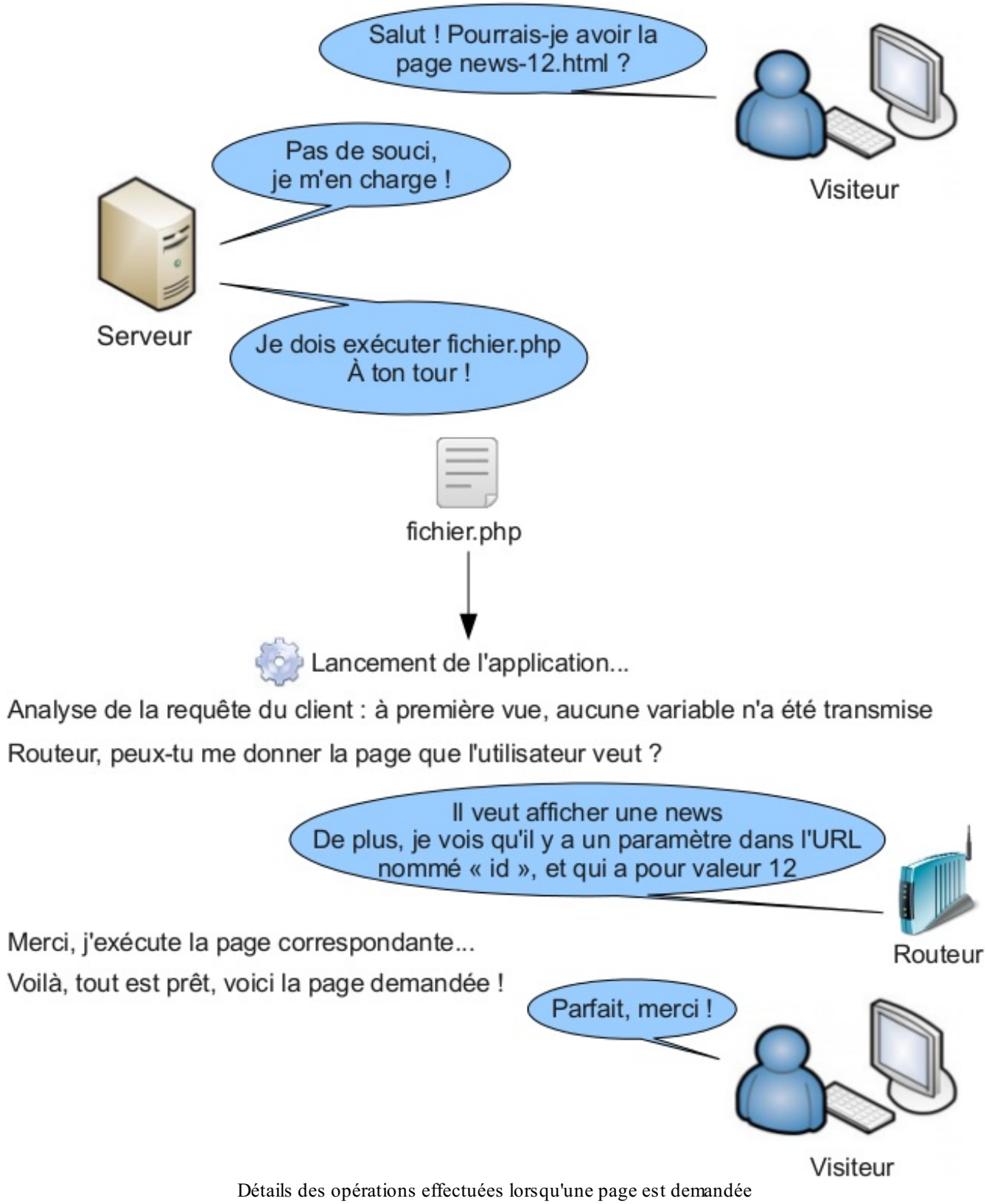
Schéma simplifié du déroulement d'une application

Détaillons ce schéma, étape par étape.

- **Lancement de l'application** : lorsque l'internaute accèdera à votre site, un fichier PHP est exécuté sur le serveur. Dans notre cas, ce fichier sera exécuté **à chaque fois que le visiteur voudra accéder à une page**, quelle que soit cette dernière. Que le visiteur veuille afficher une news, ouvrir un forum ou poster un commentaire, c'est ce fichier qui sera exécuté (nous verrons comment plus tard). Ce fichier sera très réduit : il ne fera que **lancer l'application** (nous verrons plus tard qu'il se contente d'instancier une classe et d'invoquer une méthode).
- **Chargement de la requête du client** : cette étape consiste à analyser la requête envoyée par le client. C'est lors de cette étape que l'application ira chercher les variables transmises par formulaire ou par URL (les fameuses variables GET et POST).
- **Exécution de la page désirée** : c'est ici le cœur de l'exécution de l'application. Mais comment l'application connaît-elle la page à exécuter ? Quelle action le visiteur veut-il exécuter ? Veut-il afficher une news, visiter un forum, poster un commentaire ? Cette action est déterminée par ce qu'on appelle un **routeur**. En analysant l'URL, le routeur est capable de savoir ce que le visiteur veut. Par exemple, si l'URL entrée par le visiteur est **http://www.monsupersite.com/news-12.html**, alors le routeur saura que le visiteur veut afficher la news ayant pour identifiant 12 dans la base de données. Le routeur

va donc retourner cette action à l'application qui l'exécutera (nous verrons plus tard ce dont il s'agit vraiment).
- **Envoi de la réponse au client** : après avoir exécuté l'action désirée (par exemple, si le visiteur veut afficher une news, l'action correspondante est de récupérer cette news), l'application va afficher le tout, et l'exécution du script sera terminée. C'est à ce moment-là que la page sera envoyée au visiteur.

Pour bien cerner le principe, prenons un exemple. Si le visiteur veut voir la news n°12, alors il demandera la page **news-12.html**. Schématiquement, voici ce qui va se passer (voir la figure suivante).



Détails des opérations effectuées lorsqu'une page est demandée

Il est très important de comprendre ce principe. Si vous ne saisissez pas tout, n'hésitez pas à relire les explications, sinon vous risquez de ne pas suivre la suite du cours.

Vous êtes maintenant au courant qu'une application s'exécute et qu'elle a pour rôle d'orchestrer l'exécution du script afin de donner la page au visiteur. Dans un tutoriel sur la POO en PHP, je suis sûr que vous savez comment sera représenté cette application dans votre script... Oui, notre application sera un objet ! Qui dit objet dit classe, et qui dit classe dit fonctionnalités. Reprenez le schéma et énoncez-moi les fonctionnalités que possède notre classe `Application`...

Pour l'instant, elle ne possède qu'une fonctionnalité : celle de s'exécuter (nous verrons par la suite qu'elle en possèdera plusieurs autres, mais vous ne pouviez pas les deviner). Cette fonctionnalité est obligatoire quelle que soit l'application : quel serait l'intérêt d'une application si elle ne pouvait pas s'exécuter ?

En général, dans un site web, il y a deux applications : le *frontend* et le *backend*. La première est l'application accessible par tout le monde : c'est à travers cette application qu'un visiteur pourra afficher une news, lire un sujet d'un forum, poster un commentaire, etc. La seconde application est l'espace d'administration : l'accès est bloqué aux visiteurs. Pour y accéder, une paire identifiant-mot de passe est requise. Comme vous l'aurez peut-être compris, ces deux applications seront représentées par deux classes héritant de notre classe de base `Application`.

Un peu d'organisation

Nous avons vu suffisamment de notions pour se pencher sur l'architecture de notre projet. En effet, nous savons que notre site web sera composé de deux applications (*frontend* et *backend*) ainsi que d'une classe de base `Application`.

La classe `Application` fait partie de notre **bibliothèque** (ou **library** en anglais), comme de nombreuses autres classes dont nous parlerons plus tard. Toutes ces classes vont, par conséquent, être placées dans un dossier **/Library**.

Pour les deux applications, c'est un peu plus compliqué. En effet, une application ne tient pas dans un seul fichier : elle est divisée en plusieurs parties. Parmi ces parties, nous trouvons la plus grosse : la partie contenant les modules.

Qu'est-ce qu'un module ?

Un module est un ensemble d'actions et de données concernant une partie du site. Par exemple, les actions « afficher une news » et « commenter une news » font partie du même module de news, tandis que les actions « afficher ce sujet » et « poster dans ce sujet » font partie du module du forum.

Ainsi, je vous propose l'architecture suivante :

- Le dossier **/Applications** contiendra nos applications (*frontend* et *backend*).
- Le sous-dossier **/Applications/Nomdelapplication/Modules** contiendra les modules de l'application (par exemple, si l'application *frontend* possède un module de news, alors il y aura un dossier **/Applications/Frontend/Modules/News**).

Ne vous inquiétez pas, nous reviendrons en détail sur ces fameux modules. J'ai introduit la notion ici pour parler de l'architecture (et surtout de ce qui va suivre). En effet, vous devriez vous posez une question : quand l'utilisateur veut afficher une page, quel fichier PHP sera exécuté en premier ? Ou pourrai-je placer mes feuilles de style et mes images ?

Tous les fichiers accessibles au public devront être placés dans un dossier **/Web**. Pour être plus précis, votre serveur HTTP ne pointera pas vers la racine du projet, mais vers le dossier **/Web**. Je m'explique.

Sur votre serveur local, si vous tapez **localhost** dans la barre d'adresse, alors votre serveur renverra le contenu du dossier C:\Wamp\www si vous êtes sous WampServer, ou du dossier /var/www si vous êtes sous LAMP. Vous êtes-vous déjà demandé comment est-ce que cela se faisait ? Qui a décrété cela ? En fait, c'est écrit quelque part, dans un fichier de configuration. Il y en a un qui dit que si on tape **localhost** dans la barre d'adresse, alors on se connectera sur l'ordinateur. Ce fichier de configuration est le fichier **hosts**. Le chemin menant à ce fichier est **C:\windows\system32\drivers\etc\hosts** sous Windows et **/etc/hosts** sous Linux et Mac OS. Vous pouvez l'éditer (à condition d'avoir les droits). Faites le test : ajoutez la ligne suivante à la fin du fichier.

Code : autre

```
127.0.0.1  monsupersite
```

Sauvegardez le fichier et fermez-le. Ouvrez votre navigateur, tapez **monsupersite** et... vous atterrissez sur la même page que quand vous tapiez **localhost** !

Maintenant, nous allons utiliser un autre fichier. La manipulation va consister à dire à l'ordinateur que lorsqu'on tape **monsupersite**, on ne voudra pas le contenu de **C:\Wamp\www** ou de **/var/www**, mais de **C:\Wamp\www\monsupersite\Web** si vous êtes sous Windows ou de **/var/www/monsupersite/Web** si vous êtes sous Linux ou Mac OS.

Tout d'abord, je vous annonce que nous allons utiliser le module **mod_vhost_alias** d'Apache. Assurez-vous donc qu'il soit bien activé. Le fichier que nous allons manipuler est le fichier de configuration d'Apache, à savoir **httpd.conf** sous WampServer. Rajoutez à la fin du fichier :

Code : Apache

```
<VirtualHost *:80>
  ServerAdmin webmaster@localhost

  # Mettez ici le nom de domaine que vous avez utilisé dans le
fichier hosts.
  ServerName monsupersite

  # Mettez ici le chemin vers lequel doit pointer le domaine.
  # Je suis sous Linux. Si vous êtes sous Windows, le chemin sera
de la forme C:\Wamp\www\monsupersite\Web
  DocumentRoot /home/victor/www/monsupersite/Web
  <Directory /home/victor/www/monsupersite/Web>
    Options Indexes FollowSymLinks MultiViews

    # Cette directive permet d'activer les .htaccess.
    AllowOverride All

    # Si le serveur est accessible via l'Internet mais que vous
n'en faites qu'une utilisation personnelle
    # pensez à interdire l'accès à tout le monde
    # sauf au localhost, sinon vous ne pourrez pas y accéder !
    deny from all
    allow from localhost
  </Directory>
</VirtualHost>
```

Si vous avez un serveur LAMP, n'essayez pas de trouver le fichier **httpd.conf**, il n'existe pas. 

En fait, la création d'hôtes virtuels s'effectue en créant un nouveau fichier contenant la configuration de ce dernier. Le fichier à créer est **/etc/apache2/sites-available/monsupersite** (remplacez **monsupersite** par le domaine choisi). Placez-y à l'intérieur le contenu que je vous ai donné. Ensuite, il n'y a plus qu'à activer cette nouvelle configuration grâce à la commande `sudo a2ensite monsupersite` (remplacez **monsupersite** par le nom du fichier contenant la configuration de l'hôte virtuel).

Dans tous les cas, que vous soyez sous Windows, Linux, Mac OS ou quoi que ce soit d'autre, **redémarrez Apache** pour que la nouvelle configuration soit prise en compte.

Essayez d'entrer **monsupersite** dans la barre d'adresse, et vous verrez que le navigateur vous affiche le dossier spécifié dans la configuration d'Apache !

Les entrailles de l'application

Avant d'aller plus loin, il est indispensable (voire obligatoire) de savoir utiliser le pattern MVC. Pour cela, je vous conseille d'aller lire le tutoriel *Adopter un style de programmation clair avec le modèle MVC*. En effet, ce cours explique la théorie du pattern MVC et je ne ferais que créer un doublon si je vous expliquais à mon tour ce motif.

Nous pouvons donc passer directement aux choses sérieuses.

Retour sur les modules

Comme je vous l'avais brièvement indiqué, un module englobe un ensemble d'actions agissant sur une même partie du site. C'est donc à l'intérieur de ce module que nous allons créer le contrôleur, les vues et les modèles.

Le contrôleur

Nous allons donc créer pour chaque module un contrôleur qui contiendra au moins autant de méthodes que d'actions. Par exemple, si dans le module de news je veux pouvoir avoir la possibilité d'afficher l'index du module (qui nous dévoilera la liste des cinq dernières news par exemple) et afficher une news, j'aurais alors deux méthodes dans mon contrôleur : `executeIndex` et `executeShow`. Ce fichier aura pour nom **NomDuModuleController.class.php**, ce qui nous donne, pour le module de news, un fichier du nom de **NewsController.class.php**. Celui-ci est directement situé dans le dossier du module.

Les vues

Chacune de ces actions correspond, comme vous le savez, à une vue. Nous aurons donc pour chaque action une vue du même nom. Par exemple, pour l'action `show`, nous aurons un fichier **show.php**. Toutes les vues sont à placer dans le dossier **Views** du module.

Les modèles

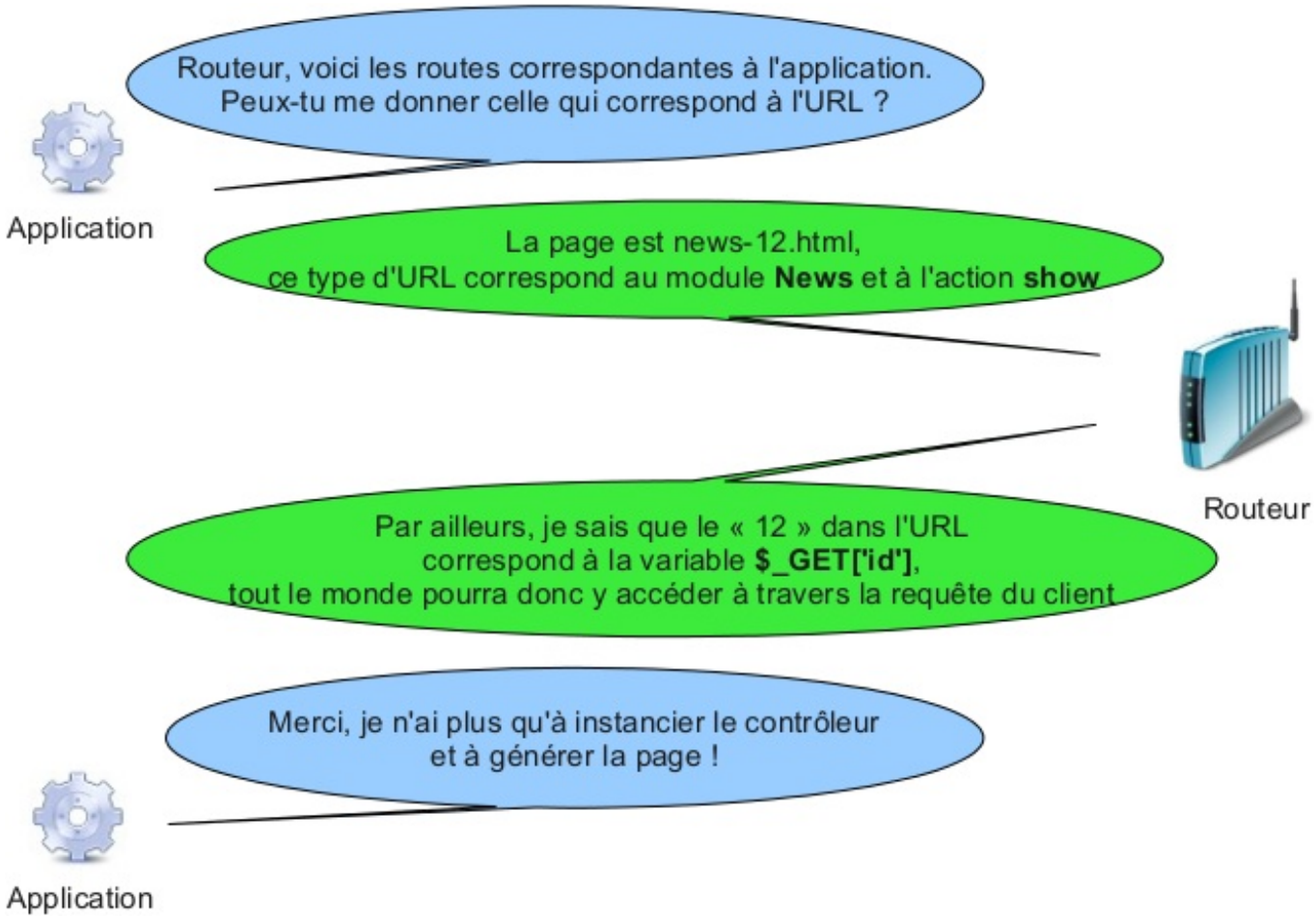
En fait, les modèles, vous les connaissez déjà : il s'agit des **managers**. Ce sont eux qui feront office de modèles. Les modèles ne sont rien d'autre que des fichiers permettant l'interaction avec les données. Pour chaque module, nous aurons donc au moins deux fichiers constituant le modèle : le manager abstrait de base (**NewsManager.class.php**) et au moins une classe exploitant ce manager (par exemple **NewsManager_PDO.class.php**). Tous les modèles devront être placés dans le dossier **/Library/Models** afin qu'ils puissent être utilisés facilement par deux applications différentes (ce qui est souvent le cas avec les applications *backend* et *frontend*). Cependant, ces modèles ont besoin des classes représentant les **entités** qu'ils gèrent. La classe représentant un enregistrement (comme `News` dans notre cas) sera, elle, placée dans le dossier **/Library/Entities**. Nous aurons l'occasion de faire un petit rappel sur cette organisation lors du prochain chapitre.

Le *back controller* de base

Tous ces contrôleurs sont chacun des *back controller*. Et que met-on en place quand on peut dire qu'une entité B est une entité A ? Un lien de parenté, bien évidemment ! Ainsi, nous aurons au sein de notre bibliothèque une classe abstraite `BackController` dont héritera chaque *back controller*. L'éternelle question que l'on peut se poser : mais que permet de faire cette classe ? Pour l'instant, il n'y en a qu'une : celle d'exécuter une action (donc une méthode).

Je voudrais faire un petit retour sur le fonctionnement du routeur. Souvenez-vous : je vous avais dit que le routeur savait à quoi correspondait l'URL et retournait en conséquence l'action à exécuter à l'application, qui sera donc apte à l'exécuter. Cela prendra plus de sens maintenant que nous avons vu la notion de *back controller*. Le routeur aura pour rôle de **récupérer la route correspondant à l'URL**. L'application, qui exploitera le routeur, instanciera donc le contrôleur correspondant à la route que le routeur lui aura renvoyée. Par exemple, si l'utilisateur veut afficher une news, le routeur retournera la route correspondante, et l'application créera une instance de `NewsController` en lui ayant spécifié qu'il devra effectuer l'action `show`. L'application n'aura donc plus qu'à exécuter le *back controller*.

Souvenez-vous du schéma sur le déroulement de l'application. Si nous faisions un petit zoom sur le routeur, nous obtiendrions ceci (voir la figure suivante).



Zoom sur le routeur

Vous vous posez peut-être des questions sur le contenu de la deuxième bulle associée au routeur. En fait, le routeur analyse toute l'URL et décrypte ce qui s'y cache (nous verrons plus tard comment). Quand je dis que « tout le monde pourra y accéder à travers la requête du client », c'est que nous pourrons accéder à cette valeur à travers la classe qui représentera la requête du client (que nous construirons d'ailleurs durant le prochain chapitre).

La page

Nous avons parlé jusqu'à présent du déroulement de l'application et nous avons commencé à creuser un petit peu en parlant de l'organisation interne avec l'utilisation du pattern MVC. Cependant, comment est générée la page que le visiteur aura devant les yeux ? Cette page, vous l'aurez peut-être deviné, sera représentée par une classe. Comme d'habitude, qui dit classe dit fonctionnalités. En effet, une page en possède plusieurs :

- Celle d'ajouter une variable à la page (le contrôleur aura besoin de passer des données à la vue).
- Celle d'assigner une vue à la page (il faut qu'on puisse dire à notre page quelle vue elle doit utiliser).
- De générer la page avec le *layout* de l'application.

Des explications s'imposent, notamment sur ce que sont précisément ces vues et ce fameux *layout*. Le *layout* est le fichier contenant « l'enveloppe » du site (déclaration du doctype, inclusion des feuilles de style, déclaration des balises meta, etc.). Voici un exemple très simple :

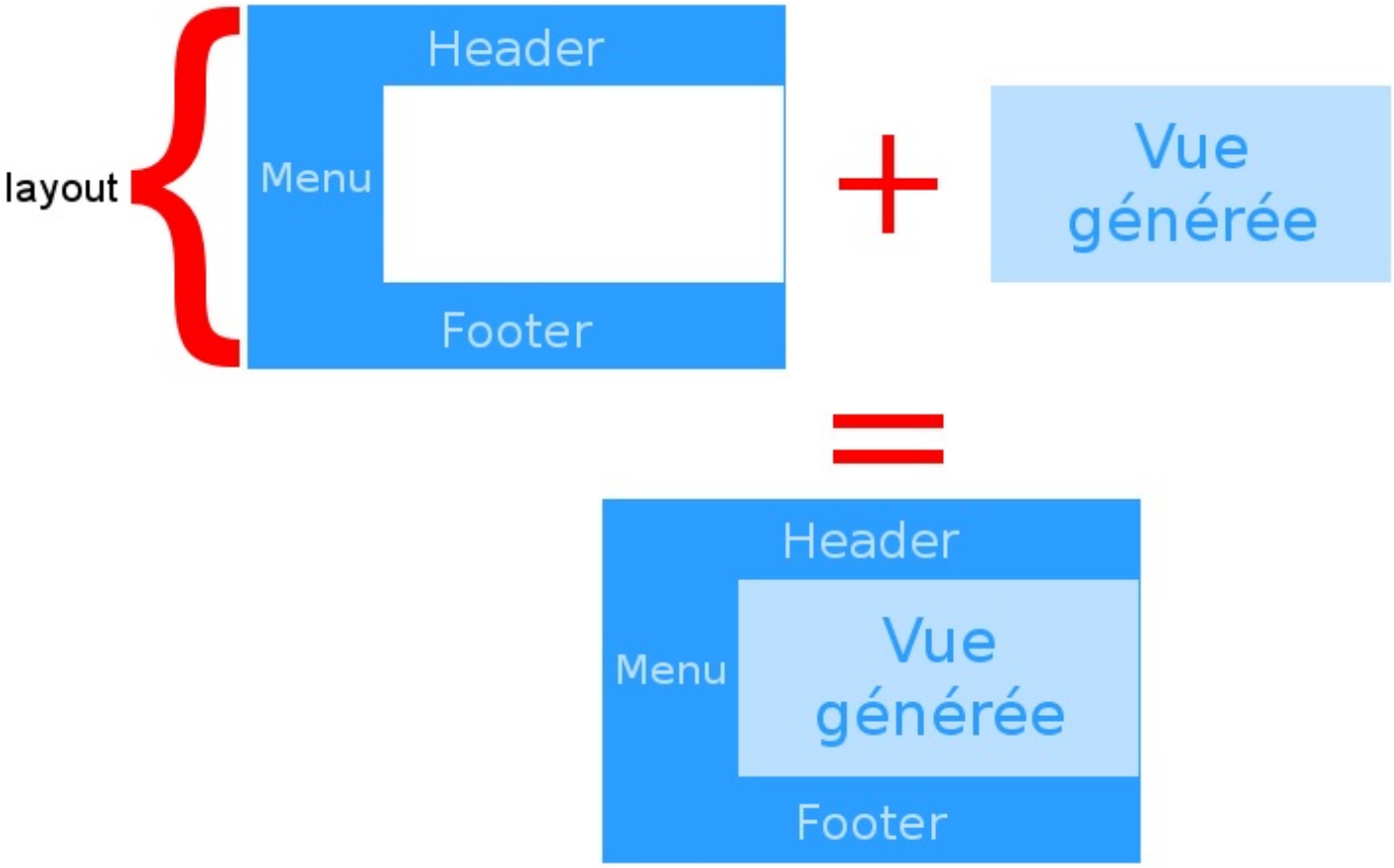
Code : PHP

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>Mon super site</title>
    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />
  </head>
```

```
    <body>
      <?php echo $content; ?>
    </body>
  </html>
```

Le *layout*, spécifique à chaque application, doit se placer dans **/Applications/Nomdelapplication/Templates**, sous le nom de **layout.php**. Comme vous pouvez vous en apercevoir, il y a une variable `$content` qui traîne. Vous aurez sans doute deviné que cette variable sera le contenu de la page : ce sera donc notre classe `Page` qui se chargera de générer la vue, de stocker ce contenu dans `$content` puis d'inclure le layout correspondant à l'application.

Schématiquement, voici comment la page se construit (voir la figure suivante).



Représentation schématique de la construction d'une page

L'autoload

L'autoload sera très simple. En effet, comme nous le verrons plus tard, chaque classe se situe dans un *namespace*. Ces *namespaces* correspondent aux dossiers dans lesquels sont placées les classes.

Si vous n'avez jamais entendu parler des *namespaces*, je vous invite à lire le tutoriel *Les espaces de noms en PHP*.

Prenons l'exemple de la classe `Application`. Comme nous l'avons vu, cette classe fait partie de notre bibliothèque, donc est placée dans le dossier **/Library**, stocké dans le fichier **Application.class.php**. Ce fichier, puisqu'il est placé dans le dossier **/Library**, contiendra donc la classe `Application` dans le **namespace** `Library` :

Code : PHP - /Library/Application.class.php

```
<?php
namespace Library;
```

```php
class Application
{
  // ...
}
```

Maintenant, regardons du côté de l'autoload. Souvenez-vous : notre fonction, appelée par PHP lorsqu'une classe non déclarée est invoquée, possède un paramètre, le nom de la classe. Cependant, si la classe si situe dans un *namespace*, alors il n'y aura pas que le nom de la classe qui sera passé en argument à notre fonction, mais **le nom de la classe précédé du namespace qui la contient**. Si l'on prend l'exemple de notre classe `Application`, et que l'on l'instancie pour que l'autoload soit appelé, alors l'argument vaudra `Library\Application`. Vous voyez à quoi ressemblera notre autoload ? Il se contentera de remplacer les antislashes (\) par des slashes (/) de l'argument et d'inclure le fichier correspondant.

Notre autoload, situé dans **/Library**, ressemblera donc à ça :

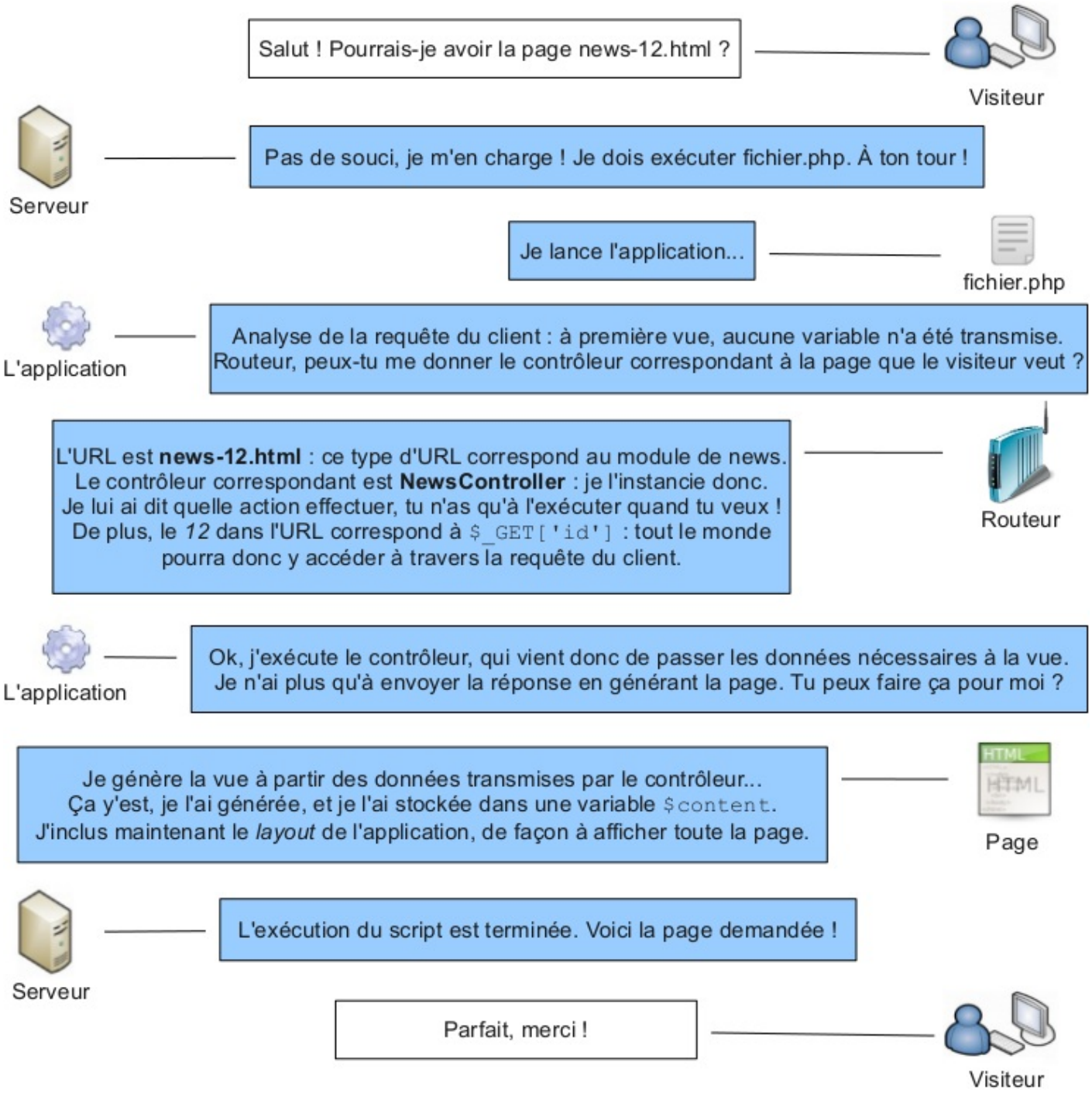
Code : PHP - /Library/autoload.php

```php
<?php
function autoload($class)
{
  require '../'.str_replace('\\', '/', $class).'.class.php';
}

spl_autoload_register('autoload');
```

Gardez-le dans un coin de votre ordinateur, vous le comprendrez sans doute mieux au fil de la création de l'application.

Résumé du déroulement de l'application

Nous allons ici résumer tout ce que nous avons vu à travers un gros schéma. Je vous conseille de l'imprimer sur une feuille qui sera toujours à côté de votre écran quand vous lirez le cours : vous comprendrez mieux ce que nous serons en train de faire (voir la figure suivante).



Déroulement détaillé de l'application

Il est **indispensable** que vous compreniez ce schéma. Si vous ne voulez pas l'apprendre par cœur, **imprimez-le** afin de toujours l'avoir sous les yeux quand vous travaillerez sur ce projet !

Développement de la bibliothèque

Le plus gros a été fait : nous savons comment fonctionnera notre application, nous savons où nous voulons aller. Maintenant, il ne reste plus qu'à nous servir de tout cela pour construire les diagrammes UML qui vont lier toutes nos classes, nous permettant ainsi de les écrire plus facilement.

Accrochez-vous à vos claviers, ça ne va pas être de tout repos !

L'application
L'application

Commençons par construire notre classe `Application`. Souvenez-vous : nous avions dit que cette classe possédait une fonctionnalité, celle de s'exécuter. Or je ne vous ai pas encore parlé des **caractéristiques** de l'application. La première ne vous vient peut-être pas à l'esprit, mais je l'ai pourtant déjà évoqué : il s'agit du **nom** de l'application.

Il en existe deux autres. Nous en avons brièvement parlé, et il est fort probable que vous les ayez oubliées : il s'agit de la **requête** ainsi que la **réponse** envoyée au client. Ainsi, avant de créer notre classe `Application`, nous allons nous intéresser à ces deux entités.

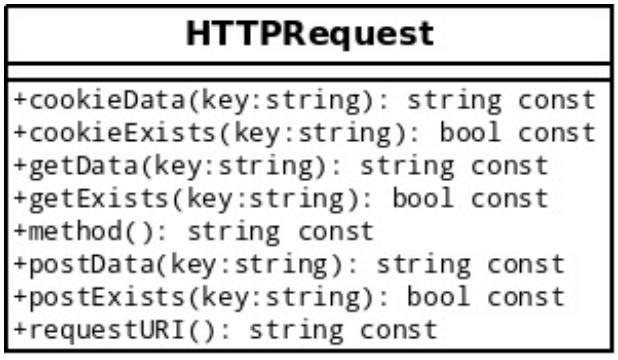
La requête du client

Schématisons

Comme vous vous en doutez, nous allons représenter la requête du client au travers d'une instance de classe. Comme pour toute classe, intéressons-nous aux fonctionnalités attendues. Qu'est-ce qui nous intéresse dans la requête du client ? Quelles fonctionnalités seraient intéressantes ? À partir de cette instance, il serait pratique de pouvoir :

- Obtenir une variable `POST`.
- Obtenir une variable `GET`.
- Obtenir un cookie.
- Obtenir la méthode employée pour envoyer la requête (méthode `GET` ou `POST`).
- Obtenir l'URL entrée (utile pour que le routeur connaisse la page souhaitée).

Et pour la route, voici un petit diagramme (j'en ai profité pour ajouter des méthodes permettant de vérifier l'existence de tel cookie / variable `GET` / variable `POST`) - voir la figure suivante.



Modélisation de la classe HTTPRequest

Codons

Le contenu est assez simple, les méthodes effectuent des opérations basiques. Voici donc le résultat auquel vous étiez censé arriver :

Code : PHP

```php
<?php
namespace Library;

class HTTPRequest
{
```

```
        public function cookieData($key)
        {
          return isset($_COOKIE[$key]) ? $_COOKIE[$key] : null;
        }

        public function cookieExists($key)
        {
          return isset($_COOKIE[$key]);
        }

        public function getData($key)
        {
          return isset($_GET[$key]) ? $_GET[$key] : null;
        }

        public function getExists($key)
        {
          return isset($_GET[$key]);
        }

        public function method()
        {
          return $_SERVER['REQUEST_METHOD'];
        }

        public function postData($key)
        {
          return isset($_POST[$key]) ? $_POST[$key] : null;
        }

        public function postExists($key)
        {
          return isset($_POST[$key]);
        }

        public function requestURI()
        {
          return $_SERVER['REQUEST_URI'];
        }
      }
```

Un petit mot sur la toute première ligne, celle qui contient la déclaration du *namespace*. J'en avais déjà parlé lors de l'écriture de l'autoload, mais je me permets de faire une petite piqûre de rappel. Toutes les classes de notre projet sont déclarées dans des *namespaces*. Cela permet d'une part de structurer son projet et, d'autre part, d'écrire un autoload simple qui sait directement, grâce au *namespace* contenant la classe, le chemin du fichier contenant ladite classe.

Par exemple, si j'ai un contrôleur du module news. Celui-ci sera placé dans le dossier **/Applications/Frontend/Modules/News** (si vous avez oublié ce chemin, ne vous inquiétez pas, nous y reviendrons : je l'utilise juste pour l'exemple). La classe représentant ce contrôleur (`NewsController`) sera donc dans le *namespace* `Applications\Frontend\Modules\News` !

La réponse envoyée au client

Schématisons

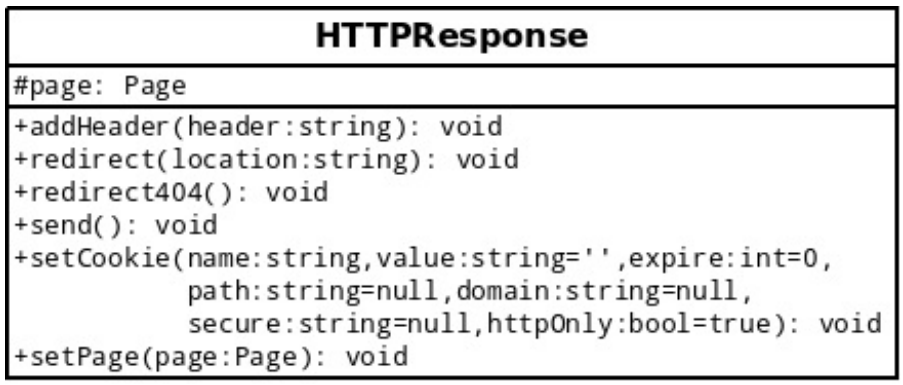
Là aussi, nous allons représenter la réponse envoyée au client au travers d'une entité. Cette entité, vous l'aurez compris, n'est autre qu'une instance d'une classe. Quelles fonctionnalités attendons-nous de cette classe ? Que voulons-nous envoyer au visiteur ? La réponse la plus évidente est la **page**. Nous voulons pouvoir assigner une **page** à la réponse. Cependant, il est bien beau d'assigner une page, encore faudrait-il pouvoir l'envoyer ! Voici une deuxième fonctionnalité : celle d'**envoyer** la réponse en **générant** la page.

Il existe de nombreuses autres fonctionnalités « accessoires ». Pour ma part, je trouvais intéressant de pouvoir rediriger le visiteur vers une erreur 404, lui écrire un *cookie* et d'ajouter un *header* spécifique. Pour résumer, notre classe nous permettra :

- D'assigner une page à la réponse.

- D'envoyer la réponse en générant la page.
- De rediriger l'utilisateur.
- De le rediriger vers une erreur 404.
- D'ajouter un cookie.
- D'ajouter un header spécifique.

Et à la figure suivante, le schéma tant attendu !



Modélisation de la classe HHTPResponse

Notez la valeur par défaut du dernier paramètre de la méthode `setCookie()` : elle est à `true`, alors qu'elle est à `false` sur la fonction `setcookie()` de la librairie standard de PHP. Il s'agit d'une sécurité qu'il est toujours préférable d'activer.

Concernant la redirection vers la page 404, laissez-là vide pour l'instant, nous nous chargerons de son implémentation par la suite.

Codons

Voici le code que vous devriez avoir obtenu :

Code : PHP

```php
<?php
namespace Library;

class HTTPResponse
{
  protected $page;

  public function addHeader($header)
  {
    header($header);
  }

  public function redirect($location)
  {
    header('Location: '.$location);
    exit;
  }

  public function redirect404()
  {
  }

  public function send()
  {
    // Actuellement, cette ligne a peu de sens dans votre esprit.
    // Promis, vous saurez vraiment ce qu'elle fait d'ici la fin du chapitre
    // (bien que je suis sûr que les noms choisis sont assez explicites !).
    exit($this->page->getGeneratedPage());
```

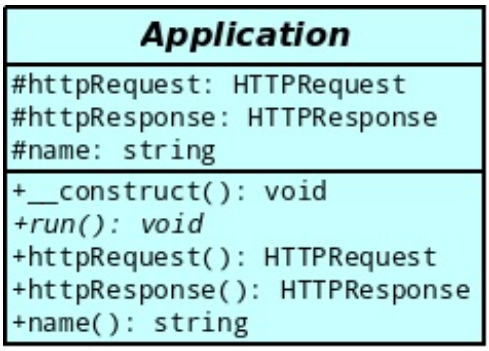
```
  }

  public function setPage(Page $page)
  {
    $this->page = $page;
  }

  // Changement par rapport à la fonction setcookie() : le dernier
argument est par défaut à true.
  public function setCookie($name, $value = '', $expire = 0, $path =
null, $domain = null, $secure = false, $httpOnly = true)
  {
    setcookie($name, $value, $expire, $path, $domain, $secure,
$httpOnly);
  }
}
```

Retour sur notre application

Schématisons

Maintenant que nous avons vu comment sont représentées la requête du client et la réponse que nous allons lui envoyer, nous pouvons réfléchir pleinement à ce qui compose notre classe. Elle possède une fonctionnalité (celle de s'exécuter) et trois caractéristiques : son nom, la requête du client et la réponse que nous allons lui envoyer.

Je vous rappelle que nous construisons une classe `Application` dont héritera chaque classe représentant une application. Par conséquent, cela n'a aucun sens d'instancier cette classe. Vous savez ce que cela signifie ? Oui, notre classe `Application` est abstraite ! La représentation graphique de notre classe sera donc telle que vous pouvez la voir sur la figure suivante.



Modélisation de la classe Application

Codons

Le code est très basique. Le constructeur se charge uniquement d'instancier les classes `HTTPRequest` et `HTTPResponse`. Quant aux autres méthodes, ce sont des accesseurs, nous avons donc vite fait le tour. 

Code : PHP

```
<?php
namespace Library;

abstract class Application
{
  protected $httpRequest;
  protected $httpResponse;
  protected $name;

  public function __construct()
  {
    $this->httpRequest = new HTTPRequest;
    $this->httpResponse = new HTTPResponse;
    $this->name = '';
```

```
  }

  abstract public function run();

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

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

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

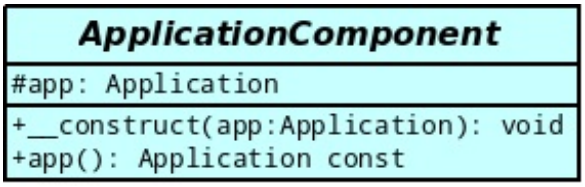
Dans le constructeur, vous voyez qu'on assigne une valeur nulle à l'attribut `name`. En fait, chaque application (qui héritera donc de cette classe) sera chargée de spécifier son nom en initialisant cet attribut (par exemple, l'application *frontend* assignera la valeur `Frontend` à cet attribut).

Les composants de l'application

Les deux dernières classes (comme la plupart des classes que nous allons créer) sont des **composants de l'application**. Toutes ces classes ont donc une nature en commun et doivent hériter d'une même classe représentant cette nature : j'ai nommé `ApplicationComponent`.

Que permet de faire cette classe ?

D'obtenir l'application à laquelle l'objet appartient. C'est tout !
Cette classe se chargera juste de stocker, pendant la construction de l'objet, l'instance de l'application exécutée. Nous avons donc une simple classe ressemblant à celle-ci (voir la figure suivante).



Modélisation de la classe ApplicationComponent

Niveau code, je pense qu'on peut difficilement faire plus simple :

Code : PHP

```
<?php
namespace Library;

abstract class ApplicationComponent
{
  protected $app;

  public function __construct(Application $app)
  {
    $this->app = $app;
  }

  public function app()
```

```
    {
      return $this->app;
    }
  }
```

Pensez donc à ajouter le lien de parenté aux classes `HTTPRequest` et `HTTPResponse`.
Et n'oubliez donc pas de passer l'instance de l'application lors de l'instanciation de ces deux classes dans le constructeur de `Application`.

Le routeur
Réfléchissons, schématisons

Comme nous l'avons vu, le routeur est l'objet qui va nous permettre de savoir quelle page nous devons exécuter. Pour en être capable, le routeur aura à sa disposition des **routes** pointant chacune vers un module et une action.

Rappel : une route, c'est une URL associée à un module et une action. Créer une route signifie donc assigner à une URL un module et une action.

La question que l'on se pose alors est : **où seront écrites les routes ?** Certains d'entre vous seraient tentés de les écrire directement à l'intérieur de la classe et faire une sorte de `switch / case` sur les routes pour trouver laquelle correspond à l'URL. Cette façon de faire présente un énorme inconvénient : votre classe représentant le routeur sera **dépendante** du projet que vous développez. Par conséquent, vous ne pourrez plus l'utiliser sur un autre site ! Il va donc falloir **externaliser** ces définitions de routes.

Comment pourrions-nous faire alors ? Puisque je vous ai dit que nous n'allons pas toucher à la classe pour chaque nouvelle route à ajouter, nous allons placer ces routes dans un autre fichier. Ce fichier doit être placé dans le dossier de l'application, et puisque ça touche à la configuration de celle-ci, nous le placerons dans un sous-dossier `Config`. Il y a aussi un détail à régler : dans quel format allons-nous écrire le fichier ? Je vous propose le format XML car ce langage est intuitif et simple à parser, notamment grâce à la bibliothèque native DOMDocument de PHP. Si ce format ne vous plaît pas, vous êtes libre d'en choisir un autre, cela n'a pas d'importance, le but étant que vous ayez un fichier que vous arriviez à parser. Le chemin complet vers ce fichier devient donc **/Applications/Nomdelapplication/Config/routes.xml**.

Comme pour tout fichier XML qui se respecte, celui-ci doit suivre une structure précise. Essayez de deviner la fonctionnalité de la ligne 3 :

Code : XML

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<routes>
  <route url="/news.html" module="News" action="index" />
</routes>
```

Alors, avez-vous une idée du rôle de la troisième ligne ? Lorsque nous allons aller sur la page **news.html**, le routeur dira donc à l'application : *« le client veut accéder au module `News` et exécuter l'action `index` »*.

Un autre problème se pose. Par exemple, si je veux afficher une news spécifique en fonction de son identifiant, comment faire ? Ou, plus généralement, comment passer des variables GET ? L'idéal serait d'utiliser des expressions régulières (ou regex) en guise d'URL. Chaque paire de parenthèses représentera une variable GET. Nous spécifierons leur nom dans un quatrième attribut `vars`.

Comme un exemple vaut mieux qu'un long discours, voyons donc un cas concret :

Code : XML

```
<route url="/news-(.+)-([0-9]+)\.html" module="News" action="show"
vars="slug,id" />
```

Ainsi, toute URL vérifiant cette expression pointera vers le module `News` et exécutera l'action `show`. Les variables `$_GET['slug']` et `$_GET['id']` seront créées et auront pour valeur le contenu des parenthèses capturantes.

Puisqu'il s'agit d'une expression régulière et qu'elle sera vérifiée par `preg_match()`, il est important d'échapper le point précédent **html**. En effet, dans une expression régulière, un point signifie « tout caractère ». Il faut donc l'échapper pour qu'il perde cette fonctionnalité.

On sait désormais que notre routeur a besoin de routes pour nous renvoyer celle qui correspond à l'URL. Cependant, s'il a besoin de routes, il va falloir les lui donner !

Pourquoi ne peut-il pas aller chercher lui-même les routes ?

S'il allait les chercher lui-même, notre classe serait dépendante de l'architecture de l'application. Si vous voulez utiliser votre classe dans un projet complètement différent, vous ne pourrez pas, car le fichier contenant les routes (**/Applications/Nomdelapplication/Config/routes.xml**) n'existera tout simplement pas. De plus, dans ce projet, les routes ne seront peut-être pas stockées dans un fichier XML, donc le parsage ne se fera pas de la même façon. Or, je vous l'avais déjà dit dans les premiers chapitres, mais l'un des points forts de la POO est son caractère **réutilisable**. Ainsi, votre classe représentant le routeur ne dépendra ni d'une architecture, ni du format du fichier stockant les routes.

De cette façon, notre classe présente déjà deux fonctionnalités :

- Celle d'ajouter une route.
- Celle de renvoyer la route correspondant à l'URL.

Avec, bien entendu, une caractéristique : la liste des routes attachée au routeur. Cependant, une autre question se pose : nous disons qu'on *« passe une route »* au routeur. Mais comment est matérialisée une route ? Puisque vous pensez orienté objet, vous devriez automatiquement me dire *« une route, c'est un objet ! »*.

Intéressons-nous donc maintenant à notre objet représentant une route. Cet objet, qu'est-ce qui le caractérise ?

- Une URL.
- Un module.
- Une action.
- Un tableau comportant les noms des variables.
- Un tableau clé/valeur comportant les noms/valeurs des variables.

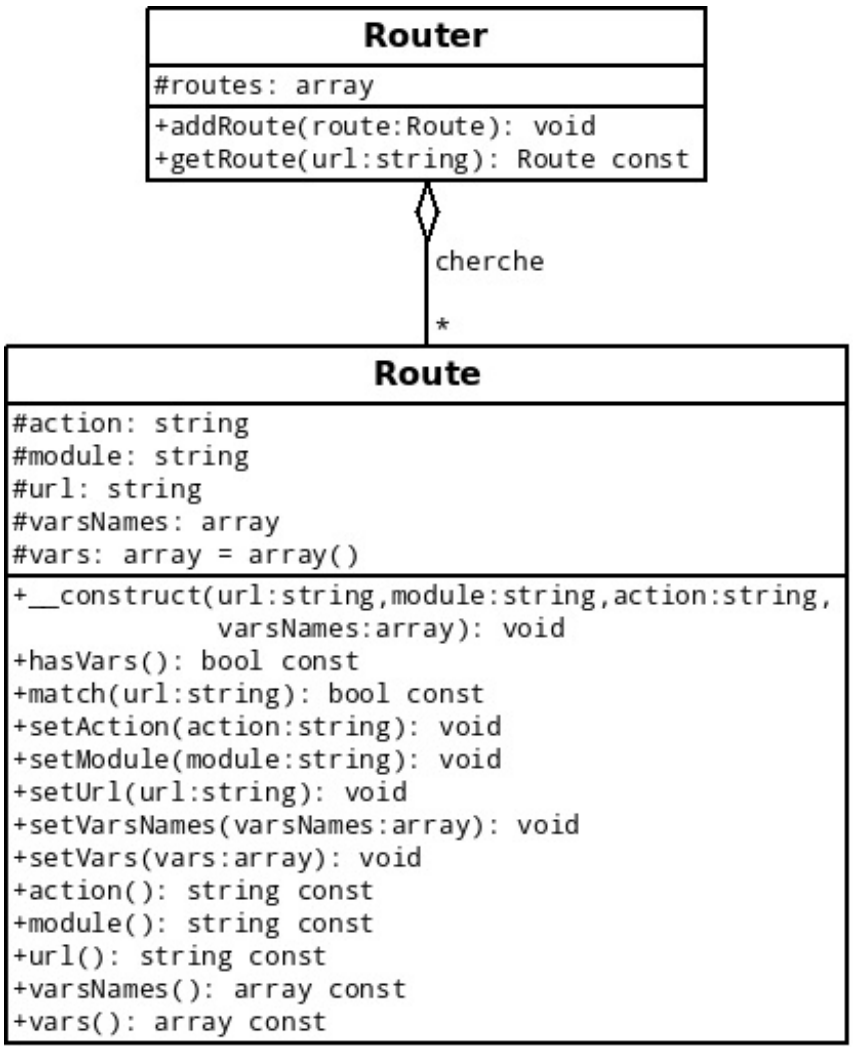
Quelle différence entre les deux dernières caractéristiques ?

En fait, lorsque nous créerons les routes, nous allons assigner les quatre premières caractéristiques (souvenez-vous du fichier XML : nous définissons une URL, un module, une action, et la liste des noms des variables). C'est donc cette dernière liste de variables que nous allons assigner à notre route. Ensuite, notre routeur ira parcourir ces routes et c'est lui qui assignera les valeurs des variables. C'est donc à ce moment-là que le tableau comportant les noms/valeurs des variables sera créé et assigné à l'attribut correspondant.

Nous pouvons maintenant dresser la liste des fonctionnalités de notre objet représentant une route :

- Celle de savoir si la route correspond à l'URL.
- Celle de savoir si la route possède des variables (utile, nous le verrons, dans notre routeur).

Pour résumer, voici le diagramme UML représentant nos classes (voir la figure suivante).



Modélisation des classes Router et Route

Codons

Commençons par nos deux classes `Router` et `Route` :

Code : PHP - /Library/Router.class.php

```php
<?php
namespace Library;

class Router
{
  protected $routes = array();

  const NO_ROUTE = 1;

  public function addRoute(Route $route)
  {
    if (!in_array($route, $this->routes))
    {
      $this->routes[] = $route;
    }
  }

  public function getRoute($url)
  {
    foreach ($this->routes as $route)
    {
      // Si la route correspond à l'URL.
      if (($varsValues = $route->match($url)) !== false)
      {
```

```
            // Si elle a des variables.
            if ($route->hasVars())
            {
              $varsNames = $route->varsNames();
              $listVars = array();

              // On créé un nouveau tableau clé/valeur.
              // (Clé = nom de la variable, valeur = sa valeur.)
              foreach ($varsValues as $key => $match)
              {
                // La première valeur contient entièrement la chaine
capturée (voir la doc sur preg_match).
                if ($key !== 0)
                {
                  $listVars[$varsNames[$key - 1]] = $match;
                }
              }

              // On assigne ce tableau de variables à la route.
              $route->setVars($listVars);
            }

            return $route;
          }
        }

        throw new \RuntimeException('Aucune route ne correspond à
l\'URL', self::NO_ROUTE);
      }
    }
```

Code : PHP - /Library/Route.class.php

```
<?php
namespace Library;

class Route
{
  protected $action;
  protected $module;
  protected $url;
  protected $varsNames;
  protected $vars = array();

  public function __construct($url, $module, $action, array
$varsNames)
  {
    $this->setUrl($url);
    $this->setModule($module);
    $this->setAction($action);
    $this->setVarsNames($varsNames);
  }

  public function hasVars()
  {
    return !empty($this->varsNames);
  }

  public function match($url)
  {
    if (preg_match('`^'.$this->url.'$`', $url, $matches))
    {
      return $matches;
    }
    else
    {
      return false;
    }
```

```
    }

    public function setAction($action)
    {
      if (is_string($action))
      {
        $this->action = $action;
      }
    }

    public function setModule($module)
    {
      if (is_string($module))
      {
        $this->module = $module;
      }
    }

    public function setUrl($url)
    {
      if (is_string($url))
      {
        $this->url = $url;
      }
    }

    public function setVarsNames(array $varsNames)
    {
      $this->varsNames = $varsNames;
    }

    public function setVars(array $vars)
    {
      $this->vars = $vars;
    }

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

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

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

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

Tout cela est bien beau, mais il serait tout de même intéressant d'exploiter notre routeur afin de l'intégrer dans notre application. Pour cela, nous allons implémenter une méthode dans notre classe `Application` qui sera chargée de nous donner le contrôleur correspondant à l'URL. Pour cela, cette méthode va parcourir le fichier XML pour ajouter les routes au routeur. Ensuite, elle va récupérer la route correspondante à l'URL (si une exception a été levée, on lèvera une erreur 404). Enfin, la méthode instanciera le contrôleur correspondant à la route et le renverra (il est possible que vous ne sachiez pas comment faire car nous n'avons pas encore vu le contrôleur en détail, donc laissez ça de côté et regardez la correction, vous comprendrez plus tard).

Voici notre nouvelle classe `Application` :

Code : PHP - /Library/Application.class.php

```php
<?php
namespace Library;

abstract class Application
{
  protected $httpRequest;
  protected $httpResponse;
  protected $name;

  public function __construct()
  {
    $this->httpRequest = new HTTPRequest($this);
    $this->httpResponse = new HTTPResponse($this);

    $this->name = '';
  }

  public function getController()
  {
    $router = new \Library\Router;

    $xml = new \DOMDocument;
    $xml->load(__DIR__.'/../Applications/'.$this->name.'/Config/routes.xml');

    $routes = $xml->getElementsByTagName('route');

    // On parcourt les routes du fichier XML.
    foreach ($routes as $route)
    {
      $vars = array();

      // On regarde si des variables sont présentes dans l'URL.
      if ($route->hasAttribute('vars'))
      {
        $vars = explode(',', $route->getAttribute('vars'));
      }

      // On ajoute la route au routeur.
      $router->addRoute(new Route($route->getAttribute('url'), $route->getAttribute('module'), $route->getAttribute('action'), $vars));
    }

    try
    {
      // On récupère la route correspondante à l'URL.
      $matchedRoute = $router->getRoute($this->httpRequest->requestURI());
    }
    catch (\RuntimeException $e)
    {
      if ($e->getCode() == \Library\Router::NO_ROUTE)
      {
        // Si aucune route ne correspond, c'est que la page demandée n'existe pas.
        $this->httpResponse->redirect404();
      }
    }

    // On ajoute les variables de l'URL au tableau $_GET.
    $_GET = array_merge($_GET, $matchedRoute->vars());

    // On instancie le contrôleur.
    $controllerClass = 'Applications\\'.$this->name.'\\Modules\\'.$matchedRoute->module().'\\'.$matchedRoute->module().'Controller';
```

```
            return new $controllerClass($this, $matchedRoute->module(),
        $matchedRoute->action());
          }

          abstract public function run();

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

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

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

Le back controller

Nous venons de construire notre routeur qui donne à l'application le contrôleur associé. Afin de suivre la logique du déroulement de l'application, construisons maintenant notre *back controller* de base.

Réfléchissons, schématisons

Remémorons-nous ce que permet de faire un objet `BackController`. Nous avions vu qu'un tel objet n'offrait qu'une seule fonctionnalité : celle d'exécuter une action. Comme pour l'objet `Application`, je ne vous avais pas parlé des caractéristiques d'un *back controller*. Qu'est-ce qui caractérise un *back controller* ? Si vous connaissez l'architecture MVC, vous savez qu'une vue est associée au *back controller* : c'est donc l'une de ses caractéristiques.

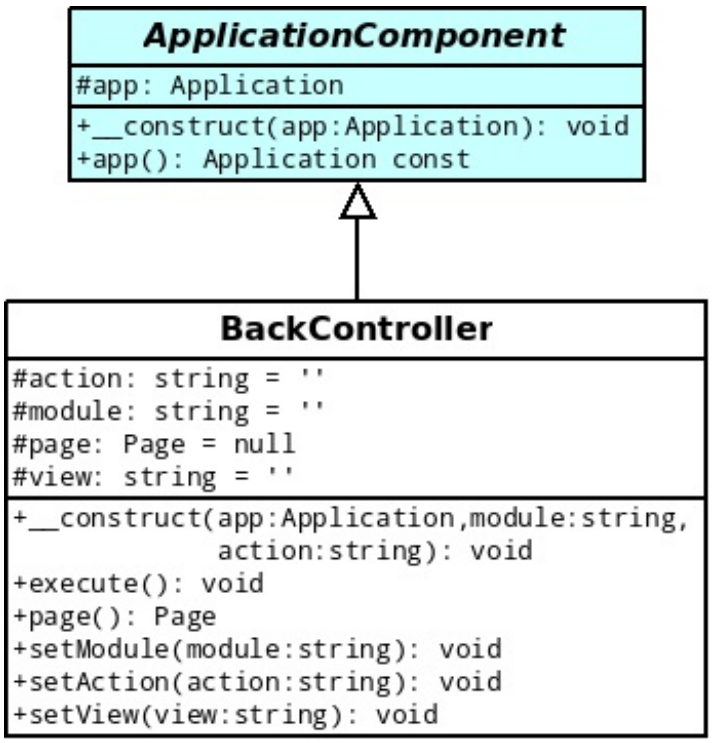
Maintenant, pensons à la nature d'un *back controller*. Celui-ci est propre à un **module**, et si on l'a instancié c'est que nous voulons qu'il exécute une **action**. Cela fait donc deux autres caractéristiques : le module et l'action.

Enfin, il y en a une dernière que vous ne pouvez pas deviner : la page associée au contrôleur. Comme nous le verrons bientôt, c'est à travers cette instance représentant la page que nous allons envoyer au visiteur que le contrôleur transmettra des données à la vue. Pour l'instant, mémorisez juste l'idée que le contrôleur est associé à une page stockée en tant qu'instance dans un attribut de la classe `BackController`.

Une instance de `BackController` nous permettra donc :

- D'exécuter une action (donc une méthode).
- D'obtenir la page associée au contrôleur.
- De modifier le module, l'action et la vue associés au contrôleur.

Cette classe est une classe de base dont héritera chaque contrôleur. Par conséquent, elle se doit d'être abstraite. Aussi, il s'agit d'un **composant** de l'application, donc un lien de parenté avec `ApplicationComponent` est à créer. Nous arrivons donc à une classe ressemblant à ça (voir la figure suivante).



Modélisation de la classe Backcontroller

Notre constructeur se chargera dans un premier temps d'appeler le constructeur de son parent. Dans un second temps, il créera une instance de la classe `Page` qu'il stockera dans l'attribut correspondant. Enfin, il assignera les valeurs au module, à l'action et à la vue (par défaut la vue a la même valeur que l'action).

Concernant la méthode `execute()`, comment fonctionnera-t-elle ? Son rôle est d'invoquer la méthode correspondant à l'action assignée à notre objet. Le nom de la méthode suit une logique qui est de se nommer `executeNomdelaction()`. Par exemple, si nous avons une action `show` sur notre module, nous devrons implémenter la méthode `executeShow()` dans notre contrôleur. Aussi, pour une question de simplicité, nous passerons la requête du client à la méthode. En effet, dans la plupart des cas, les méthodes auront besoin de la requête du client pour obtenir une donnée (que ce soit une variable GET, POST, ou un cookie).

Codons

Voici le résultat qui était à obtenir :

Code : PHP

```php
<?php
namespace Library;

abstract class BackController extends ApplicationComponent
{
  protected $action = '';
  protected $module = '';
  protected $page = null;
  protected $view = '';

  public function __construct(Application $app, $module, $action)
  {
    parent::__construct($app);

    $this->page = new Page($app);

    $this->setModule($module);
    $this->setAction($action);
    $this->setView($action);
  }

  public function execute()
  {
```

```
        $method = 'execute'.ucfirst($this->action);

        if (!is_callable(array($this, $method)))
        {
          throw new \RuntimeException('L\'action "'.$this->action.'"
n\'est pas définie sur ce module');
        }

        $this->$method($this->app->httpRequest());
      }

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

      public function setModule($module)
      {
        if (!is_string($module) || empty($module))
        {
          throw new \InvalidArgumentException('Le module doit être une
chaine de caractères valide');
        }

        $this->module = $module;
      }

      public function setAction($action)
      {
        if (!is_string($action) || empty($action))
        {
          throw new \InvalidArgumentException('L\'action doit être une
chaine de caractères valide');
        }

        $this->action = $action;
      }

      public function setView($view)
      {
        if (!is_string($view) || empty($view))
        {
          throw new \InvalidArgumentException('La vue doit être une
chaine de caractères valide');
        }

        $this->view = $view;
      }
    }
```

Accéder aux managers depuis le contrôleur

Un petit souci se pose : comment le contrôleur accèdera aux managers ? On pourrait les instancier directement dans la méthode, mais les managers exigent le DAO lors de la construction de l'objet et ce DAO n'est pas accessible depuis le contrôleur. Nous allons donc créer une classe qui gérera les managers : j'ai nommé `Managers`. Nous instancierons donc cette classe au sein de notre contrôleur en lui passant le DAO. Les méthodes filles auront accès à cet objet et pourront accéder aux managers facilement.

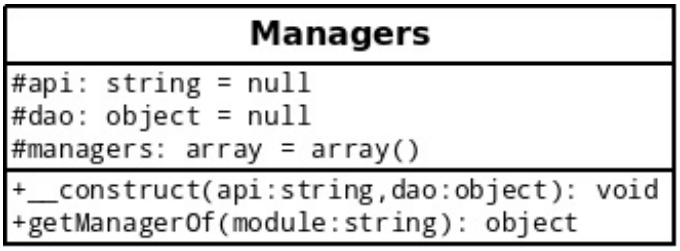
Petit rappel sur la structure d'un manager

Je vais faire un bref rappel concernant la structure des managers. Un manager, comme nous l'avons vu durant le TP des news, est divisé en deux parties. La première partie est une classe abstraite listant toutes les méthodes que le manager doit implémenter. La seconde partie est constituée des classes qui vont implémenter ces méthodes, **spécifiques à chaque DAO**. Pour reprendre l'exemple des news, la première partie était constituée de la classe abstraite `NewsManager` et la seconde partie de `NewsManager_PDO` et `NewsManager_MySQLi`.

En plus du DAO, il faudra donc spécifier à notre classe gérant ces managers l'API que l'on souhaite utiliser. Suivant ce qu'on lui demande, notre classe nous retournera une instance de `NewsManager_PDO` ou `NewsManager_MySQLi` par exemple.

La classe Managers

Schématiquement, voici à quoi ressemble la classe `Managers` (voir la figure suivante).



Modélisation de la classe Managers

Cette instance de `Managers` sera stockée dans un attribut de l'objet `BackController` comme `$managers` par exemple. L'attribution d'une instance de `Managers` à cet attribut se fait dans le constructeur de la manière suivante :

Code : PHP

```php
<?php
namespace Library;

abstract class BackController extends ApplicationComponent
{
  // ...
  protected $managers = null;

  public function __construct(Application $app, $module, $action)
  {
    parent::__construct($app);

$this->managers = new Managers('PDO',
PDOFactory::getMysqlConnexion());
    $this->page = new Page($app);

    $this->setModule($module);
    $this->setAction($action);
    $this->setView($action);
  }

  // ...
}
```

Niveau code, voici à quoi ressemble la classe `Managers` :

Code : PHP

```php
<?php
namespace Library;

class Managers
{
  protected $api = null;
  protected $dao = null;
  protected $managers = array();

  public function __construct($api, $dao)
  {
    $this->api = $api;
    $this->dao = $dao;
```

```
    }

    public function getManagerOf($module)
    {
      if (!is_string($module) || empty($module))
      {
        throw new \InvalidArgumentException('Le module spécifié est
invalide');
      }

      if (!isset($this->managers[$module]))
      {
        $manager = '\\Library\\Models\\'.$module.'Manager_'.$this-
>api;
        $this->managers[$module] = new $manager($this->dao);
      }

      return $this->managers[$module];
    }
  }
```

Et maintenant, comment je passe l'instance de PDO au constructeur de `Managers` ? Je l'instancie directement ?

Non car cela vous obligerait à modifier la classe `BackController` à chaque modification, ce qui n'est pas très flexible. Je vous conseille plutôt d'utiliser le pattern factory que nous avons vu durant la précédente partie avec la classe `PDOFactory` :

Code : PHP

```
<?php
namespace Library;

class PDOFactory
{
  public static function getMysqlConnexion()
  {
    $db = new \PDO('mysql:host=localhost;dbname=news', 'root', '');
    $db->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);

    return $db;
  }
}
```

À propos des managers

Nous trouvons ici des natures en commun : les managers sont tous des managers ! Et les entités, et bien ce sont toutes des entités !

Cela peut vous sembler bête (oui, ça l'est), mais si je vous dis ceci c'est pour mettre en évidence le lien de parenté qui saute forcément aux yeux après cette phrase. Tous les managers devront donc hériter de `Manager` et chaque entité de `Entity`. La classe `Manager` se chargera d'implémenter un constructeur qui demandera le DAO par le biais d'un paramètre, comme ceci :

Code : PHP

```
<?php
namespace Library;

abstract class Manager
{
  protected $dao;
```

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

Par contre, la classe `Entity` est légèrement plus complexe. En effet, celle-ci offre quelques fonctionnalités :

- Implémentation d'un constructeur qui hydratera l'objet si un tableau de valeurs lui est fourni.
- Implémentation d'une méthode qui permet de vérifier si l'enregistrement est nouveau ou pas. Pour cela, on vérifie si l'attribut `$id` est vide ou non (ce qui inclut le fait que toutes les tables devront posséder un champ nommé `id`).
- Implémentation des getters / setters.
- Implémentation de l'interface `ArrayAccess` (ce n'est pas obligatoire, c'est juste que je préfère utiliser l'objet comme un tableau dans les vues).

Le code obtenu devrait s'apparenter à celui-ci :

Code : PHP

```
<?php
namespace Library;

abstract class Entity implements \ArrayAccess
{
  protected $erreurs = array(),
        $id;

  public function __construct(array $donnees = array())
  {
    if (!empty($donnees))
    {
      $this->hydrate($donnees);
    }
  }

  public function isNew()
  {
    return empty($this->id);
  }

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

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

  public function setId($id)
  {
    $this->id = (int) $id;
  }

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $attribut => $valeur)
    {
      $methode = 'set'.ucfirst($attribut);

      if (is_callable(array($this, $methode)))
      {
        $this->$methode($valeur);
      }
    }
```

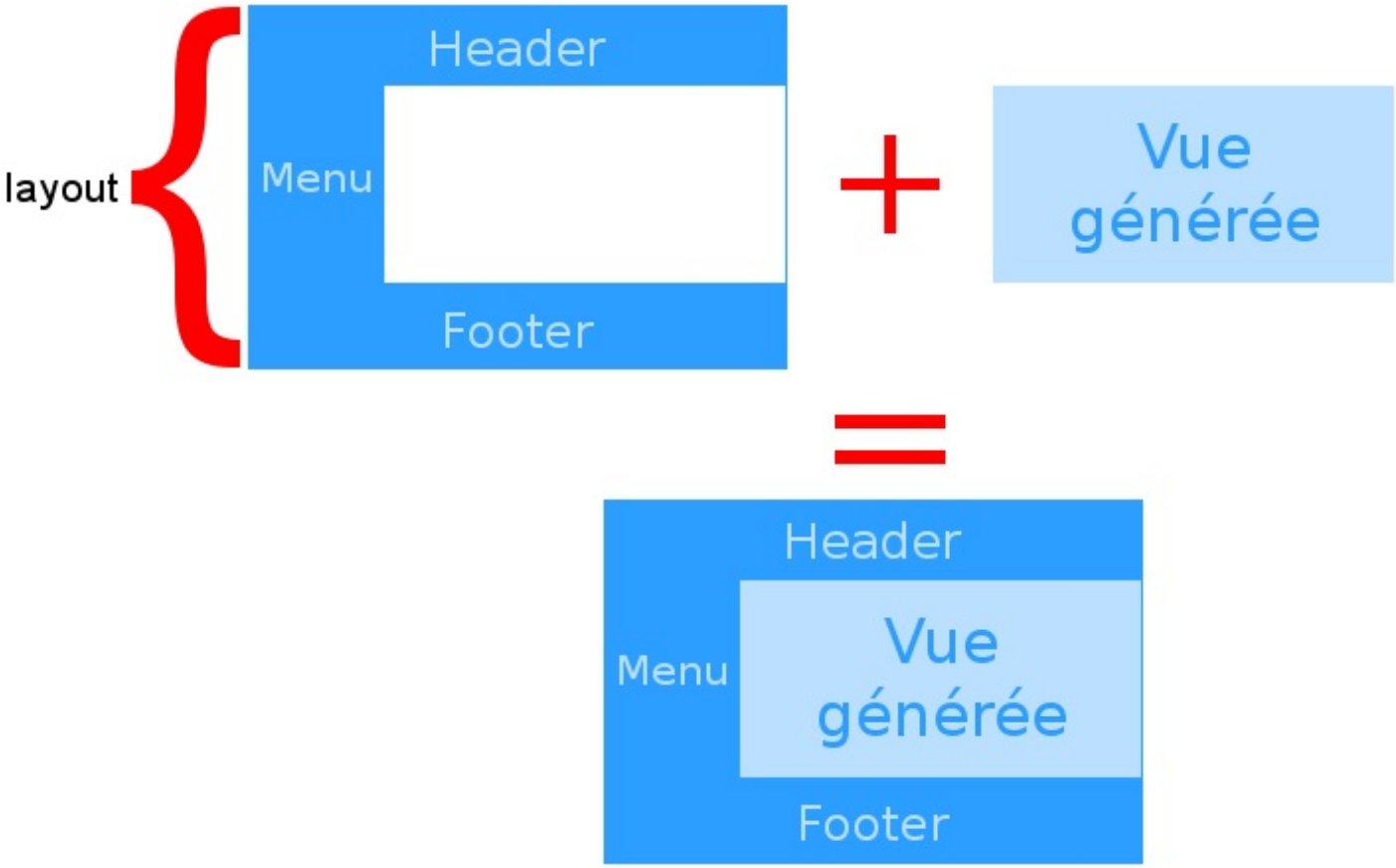
```
    }

    public function offsetGet($var)
    {
      if (isset($this->$var) && is_callable(array($this, $var)))
      {
        return $this->$var();
      }
    }

    public function offsetSet($var, $value)
    {
      $method = 'set'.ucfirst($var);

      if (isset($this->$var) && is_callable(array($this, $method)))
      {
        $this->$method($value);
      }
    }

    public function offsetExists($var)
    {
      return isset($this->$var) && is_callable(array($this, $var));
    }

    public function offsetUnset($var)
    {
      throw new \Exception('Impossible de supprimer une quelconque
valeur');
    }
}
```

La page

Toujours dans la continuité du déroulement de l'application, nous allons nous intéresser maintenant à la page qui, nous venons de le voir, était attachée à notre contrôleur.

Réfléchissons, schématisons

Commençons par nous intéresser aux fonctionnalités de notre classe `Page`. Vous le savez, une page est composée de la vue et du *layout* afin de générer le tout. Ainsi, nous avons une première fonctionnalité : celle de **générer** une page. De plus, le contrôleur doit pouvoir transmettre des variables à la vue, stockée dans cette page : **ajouter une variable à la page** est donc une autre fonctionnalité. Enfin, la page doit savoir quelle vue elle doit générer et ajouter au *layout* : il est donc possible d'**assigner une vue à la page** (voir la figure suivante).

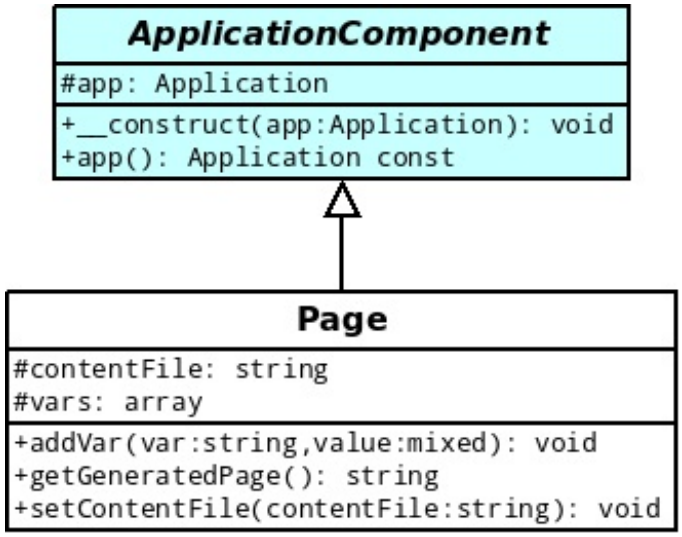


Composition d'une page

Pour résumer, une instance de notre classe `Page` nous permet :

- D'ajouter une variable à la page (le contrôleur aura besoin de passer des données à la vue).
- D'assigner une vue à la page.
- De générer la page avec le layout de l'application.

Avant de commencer à coder cette classe, voici le diagramme la représentant (voir la figure suivante).



Modélisation de la classe Page

Codons

La classe est, me semble-t-il, plutôt facile à écrire. Cependant, il se peut que vous vous demandiez comment écrire la méthode `getGeneratedPage()`. En fait, il faut **inclure** les pages pour générer leur contenu, et stocker ce contenu dans une variable

grâce aux fonctions de tamporisation de sortie pour pouvoir s'en servir plus tard. Pour la transformation du tableau stocké dans l'attribut `$vars` en variables, regardez du côté de la fonction `extract`.

Code : PHP

```php
<?php
namespace Library;

class Page extends ApplicationComponent
{
  protected $contentFile;
  protected $vars = array();

  public function addVar($var, $value)
  {
    if (!is_string($var) || is_numeric($var) || empty($var))
    {
      throw new \InvalidArgumentException('Le nom de la variable
doit être une chaine de caractère non nulle');
    }

    $this->vars[$var] = $value;
  }

  public function getGeneratedPage()
  {
    if (!file_exists($this->contentFile))
    {
      throw new \RuntimeException('La vue spécifiée n\'existe pas');
    }

    extract($this->vars);

    ob_start();
      require $this->contentFile;
    $content = ob_get_clean();

    ob_start();
      require __DIR__.'/../Applications/'.$this->app-
>name().'/Templates/layout.php';
    return ob_get_clean();
  }

  public function setContentFile($contentFile)
  {
    if (!is_string($contentFile) || empty($contentFile))
    {
      throw new \InvalidArgumentException('La vue spécifiée est
invalide');
    }

    $this->contentFile = $contentFile;
  }
}
```

Retour sur la classe BackController

Maintenant que nous avons écrit notre classe `Page`, je peux vous faire écrire une instruction dans la classe `BackController` et, plus particulièrement, la méthode `setView($view)`. En effet, lorsque l'on change de vue, il faut en informer la page concernée grâce à la méthode `setContentFile()` de notre classe `Page` :

Code : PHP

```php
<?php
namespace Library;
```

```
abstract class BackController extends ApplicationComponent
{
  // ...

  public function setView($view)
  {
    if (!is_string($view) || empty($view))
    {
      throw new \InvalidArgumentException('La vue doit être une
chaine de caractères valide');
    }

    $this->view = $view;

$this->page->setContentFile(__DIR__.'/../Applications/'.$this->app-
>name().'/Modules/'.$this->module.'/Views/'.$this->view.'.php');
  }
}
```

Retour sur la méthode HTTPResponse::redirect404()

Étant donné que vous avez compris comment fonctionne un objet `Page`, vous êtes capables d'écrire cette méthode laissée vide jusqu'à présent. Comment procéder ?

- On commence d'abord par créer une instance de la classe `Page` que l'on stocke dans l'attribut correspondant.
- On assigne ensuite le fichier qui fait office de vue à générer à la page. Ce fichier contient le message d'erreur formaté. Vous pouvez placer tous ces fichiers dans le dossier **/Errors** par exemple, sous le nom ***code*.html**. Le chemin menant au fichier contenant l'erreur 404 sera donc **/Errors/404.html**.
- On ajoute un header disant que le document est non trouvé (**HTTP/1.0 404 Not Found**).
- On envoie la réponse.

Et voici ce que l'on obtient :

Code : PHP

```
<?php
namespace Library;

class HTTPResponse extends ApplicationComponent
{
  // ...

  public function redirect404()
  {
    $this->page = new Page($this->app);
    $this->page->setContentFile(__DIR__.'/../Errors/404.html');

    $this->addHeader('HTTP/1.0 404 Not Found');

    $this->send();
  }

  // ...
}
```

Bonus : l'utilisateur

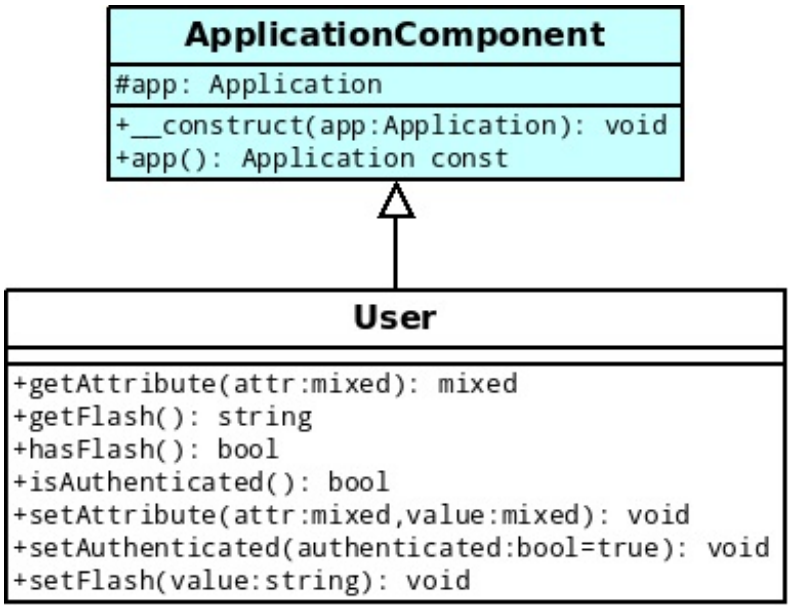
Cette classe est un « bonus », c'est-à-dire qu'elle n'est pas indispensable à l'application. Cependant, nous allons nous en servir plus tard donc ne sautez pas cette partie ! Mais rassurez-vous, nous aurons vite fait de la créer.

Réfléchissons, schématisons

L'utilisateur, qu'est-ce que c'est ? L'utilisateur est celui qui visite votre site. Comme tout site web qui se respecte, nous avons besoin d'enregistrer temporairement l'utilisateur dans la mémoire du serveur afin de stocker des informations le concernant. Nous créons donc une **session** pour l'utilisateur. Vous connaissez sans doute ce système de sessions avec le tableau `$_SESSION` et les fonctions à ce sujet que propose l'API. Notre classe, que nous nommerons `User`, devra nous permettre de gérer facilement la session de l'utilisateur. Nous pourrons donc, par le biais d'un objet `User` :

- Assigner un attribut à l'utilisateur.
- Obtenir la valeur d'un attribut.
- Authentifier l'utilisateur (cela nous sera utile lorsque nous ferons un formulaire de connexion pour l'espace d'administration).
- Savoir si l'utilisateur est authentifié.
- Assigner un message informatif à l'utilisateur que l'on affichera sur la page.
- Savoir si l'utilisateur a un tel message.
- Et enfin, récupérer ce message.

Cela donne naissance à une classe de ce genre (voir la figure suivante).



Modélisation de la classe User

Codons

Avant de commencer à coder la classe, il faut que vous ajoutiez l'instruction invoquant `session_start()` au début du fichier, en dehors de la classe. Ainsi, dès l'inclusion du fichier par l'autoload, la session démarrera et l'objet créé sera fonctionnel.

Ceci étant, voici le code que je vous propose :

Code : PHP

```php
<?php
namespace Library;

session_start();

class User extends ApplicationComponent
{
  public function getAttribute($attr)
  {
    return isset($_SESSION[$attr]) ? $_SESSION[$attr] : null;
  }

  public function getFlash()
```

```
    public function getFlash()
    {
      $flash = $_SESSION['flash'];
      unset($_SESSION['flash']);

      return $flash;
    }

    public function hasFlash()
    {
      return isset($_SESSION['flash']);
    }

    public function isAuthenticated()
    {
      return isset($_SESSION['auth']) && $_SESSION['auth'] === true;
    }

    public function setAttribute($attr, $value)
    {
      $_SESSION[$attr] = $value;
    }

    public function setAuthenticated($authenticated = true)
    {
      if (!is_bool($authenticated))
      {
        throw new \InvalidArgumentException('La valeur spécifiée à la
méthode User::setAuthenticated() doit être un boolean');
      }

      $_SESSION['auth'] = $authenticated;
    }

    public function setFlash($value)
    {
      $_SESSION['flash'] = $value;
    }
}
```

Comme promis, ce fut court, et tout ce qui compose cette classe est, il me semble, facilement compréhensible.

Pensez à modifier votre classe `Application` afin d'ajouter un attribut `$user` et à créer l'objet `User` dans le constructeur que vous stockerez dans l'attribut créé.

Bonus 2 : la configuration

Cette classe est également un bonus dans la mesure où elle n'est pas essentielle pour que l'application fonctionne. Je vous encourage à vous entraîner à créer cette classe : tout comme la classe `User`, celle-ci n'est pas compliquée (si tant est que vous sachiez parser du XML avec une bibliothèque telle que DOMDocument).

Réfléchissons, schématisons

Tout site web bien conçu se doit d'être configurable à souhait. Par conséquent, il faut que chaque application possède un **fichier de configuration** déclarant des paramètres propres à ladite application. Par exemple, si nous voulons afficher un nombre de news précis sur l'accueil, il serait préférable de spécifier un paramètre `nombre_de_news` à l'application que nous mettrons par exemple à cinq plutôt que d'insérer ce nombre en dur dans le code. De cette façon, nous aurons à modifier uniquement ce nombre dans le fichier de configuration pour faire varier le nombre de news sur la page d'accueil.

Un format pour le fichier

Le format du fichier sera le même que le fichier contenant les routes, à savoir le format XML. La base du fichier sera celle-ci :

Code : XML

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<definitions>
</definitions>
```

Chaque paramètre se déclarera avec une balise `define` comme ceci :

Code : XML

```xml
<define var="nombre_news" value="5" />
```

Emplacement du fichier

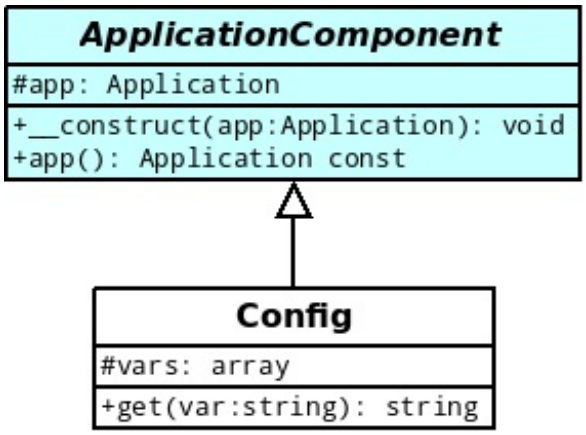
Le fichier de configuration est propre à chaque application. Par conséquent, il devra être placé aux côtés du fichier **routes.xml** sous le doux nom de **app.xml**. Son chemin complet sera donc **/Applications/Nomdelapplication/Config/app.xml**.

Fonctionnement de la classe

Nous aurons donc une classe s'occupant de gérer la configuration. Pour faire simple, nous n'allons lui implémenter qu'une seule fonctionnalité : celle de récupérer un paramètre. Il faut également garder à l'esprit qu'il s'agit d'un **composant de l'application**, donc il faut un lien de parenté avec... Je suis sûr que vous savez !

La méthode `get($var)` (qui sera chargée de récupérer la valeur d'un paramètre) ne devra pas parcourir à chaque fois le fichier de configuration, cela serait bien trop lourd. S'il s'agit du premier appel de la méthode, il faudra ouvrir le fichier XML en instanciant la classe `DOMDocument` et stocker tous les paramètres dans un attribut (admettons `$vars`). Ainsi, à chaque fois que la méthode `get()` sera invoquée, nous n'aurons qu'à retourner le paramètre précédemment enregistré.

Notre classe, plutôt simple, ressemble donc à ceci (voir la figure suivante).



Modélisation de la classe Config

Codons

Voici le résultat qui vous deviez obtenir :

Code : PHP

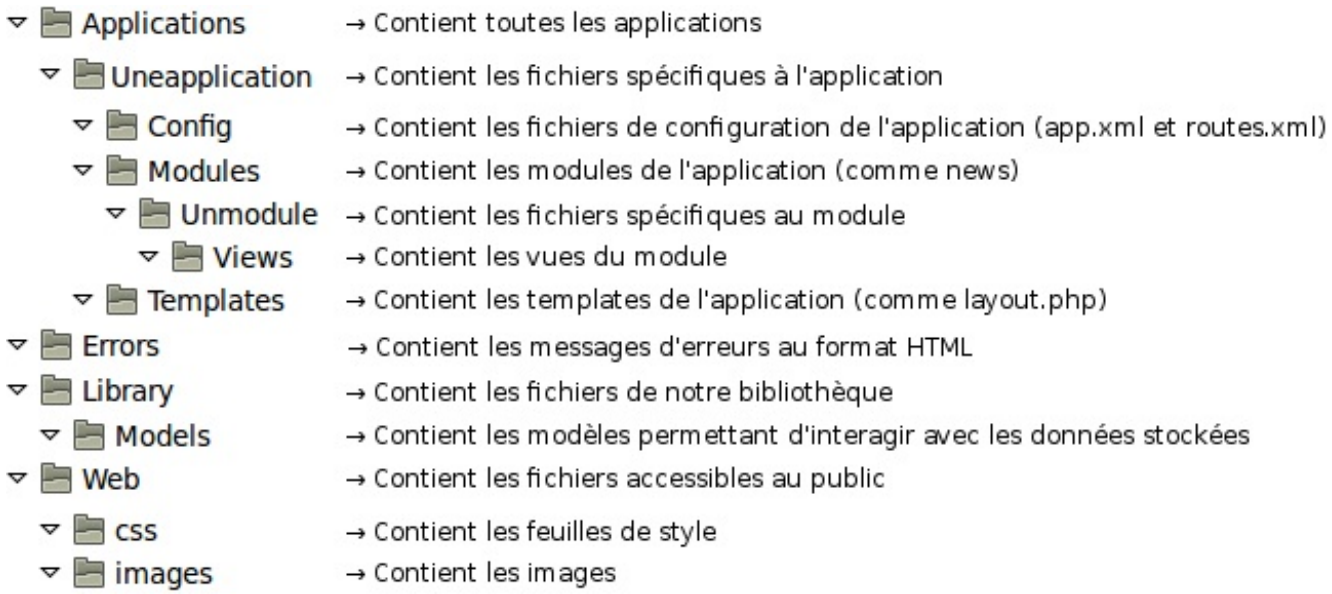
```php
<?php
namespace Library;

class Config extends ApplicationComponent
{
  protected $vars = array();

  public function get($var)
  {
    if (!$this->vars)
```

```
        {
          $xml = new \DOMDocument;
          $xml->load(__DIR__.'/../Applications/'.$this->app-
>name().'/Config/app.xml');

          $elements = $xml->getElementsByTagName('define');

          foreach ($elements as $element)
          {
            $this->vars[$element->getAttribute('var')] = $element-
>getAttribute('value');
          }
        }

        if (isset($this->vars[$var]))
        {
          return $this->vars[$var];
        }

        return null;
      }
    }
```

Il faut, comme nous l'avons fait en créant la classe `User`, ajouter un nouvel attribut à notre classe `Application` qui stockera l'instance de `Config`. Appelez-le par exemple `$config` (pour être original ).

Notre bibliothèque est écrite, voilà une bonne chose de faite ! Il ne reste plus qu'à écrire les applications et à développer les modules.

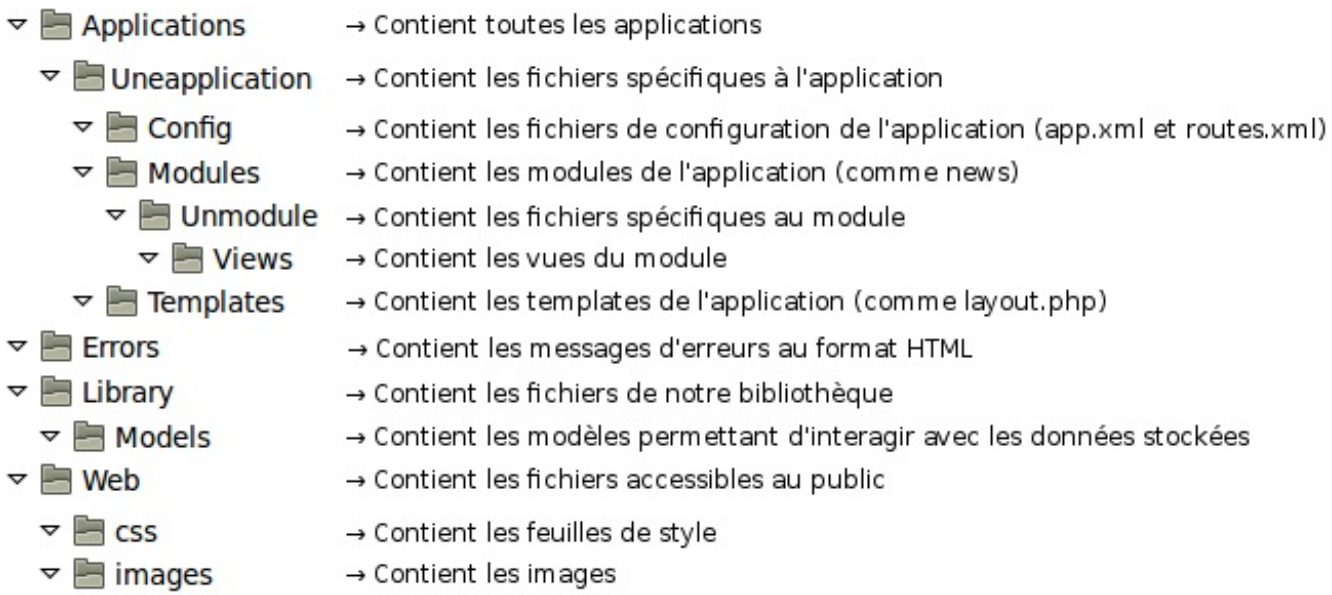
Cependant, avant de continuer, je vais m'assurer que vous me suiviez toujours. Voici l'arborescence de l'architecture, avec des explications sur chaque dossier (voir la figure suivante).



Il est possible que des dossiers vous paraissent sortir de nulle part. Cependant, je vous assure que j'en ai parlé au moins une fois lors de la création de certaines classes. N'hésitez surtout pas à relire le chapitre en vous appuyant sur cette arborescence, vous comprendrez sans doute mieux. 

Notre bibliothèque est écrite, voilà une bonne chose de faite ! Il ne reste plus qu'à écrire les applications et développer les modules.

Cependant, avant de continuer, je vais m'assurer que vous me suiviez toujours. Voici l'arborescence de l'architecture, avec des explications sur chaque dossier :



Il est possible que des dossiers vous paraissent sortir de nulle part. Cependant, je vous assure que j'en ai parlé au moins une fois lors de la création de certaines classes. N'hésitez surtout pas à relire le chapitre en vous appuyant sur cette arborescence, vous comprendrez sans doute mieux.

Le frontend

Nous allons enfin aborder quelque chose de concret en construisant notre première application : le *frontend*. Cette application est la partie visible par tout le monde. Nous allons construire un module de news avec commentaires, il y a de quoi faire, donc ne perdons pas une seconde de plus ! 

Il est indispensable de connaître la classe `DateTime`. Cependant, ne vous inquiétez pas, cette classe est très simple d'utilisation, surtout que nous n'utiliserons que deux méthodes (le constructeur et la méthode `format()`).

L'application

Nous allons commencer par créer les fichiers de base dont nous aurons besoin. Vous en connaissez quelques uns déjà : la classe représentant l'application, le layout, et les deux fichiers de configuration. Cependant, il nous en faut deux autres : un fichier qui instanciera notre classe et qui invoquera la méthode `run()`, et un .htaccess qui redirigera toutes les pages sur ce fichier. Nous verrons cela après avoir créé les quatre fichiers précédemment cités.

La classe `FrontendApplication`

Commençons par créer notre classe `FrontendApplication`. Avant de commencer, assurez-vous que vous avez bien créé le dossier **/Applications/Frontend** qui contiendra notre application. Créez à l'intérieur le fichier contenant notre classe, à savoir **FrontendApplication.class.php**.

Bien. Commencez par écrire le minimum de la classe avec le *namespace* correspondant (je le rappelle : le *namespace* est identique au chemin venant vers le fichier contenant la classe) en implémentant les deux méthodes à écrire, à savoir `__construct()` (qui aura pour simple contenu d'appeler le constructeur parent puis de spécifier le nom de l'application), et `run()`. Cette dernière méthode écrira cette suite d'instruction :

- Obtention du contrôleur grâce à la méthode parente `getController()`.
- Exécution du contrôleur.
- Assignation de la page créée par le contrôleur à la réponse.
- Envoi de la réponse.

Votre classe devrait ressembler à ceci :

Code : PHP - /Applications/Frontend/FrontendApplication.class.php

```php
<?php
namespace Applications\Frontend;

class FrontendApplication extends \Library\Application
{
  public function __construct()
  {
    parent::__construct();

    $this->name = 'Frontend';
  }

  public function run()
  {
    $controller = $this->getController();
    $controller->execute();

    $this->httpResponse->setPage($controller->page());
    $this->httpResponse->send();
  }
}
```

Finalement, le déroulement est assez simple quand on y regarde de plus près.

Le layout

Tout site web qui se respecte se doit d'avoir un design. Nous n'allons pas nous étaler sur ce type de création, ce n'est pas le sujet qui nous occupe. Nous allons donc nous servir d'un pack libre anciennement disponible sur un site de designs : j'ai nommé envision. C'est un design très simple et facilement intégrable, idéal pour ce que nous avons à faire.

Je vous laisse télécharger le pack et découvrir les fichiers qu'il contient. Pour rappel, les fichiers sont à placer dans **/Web**. Vous devriez ainsi vous retrouver avec deux dossiers dans **/Web** : **/Web/css** et **/Web/images**.

Revenons-en au layout. Celui-ci est assez simple et respecte les contraintes imposées par le design.

Code : PHP - /Applications/Frontend/Templates/layout.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
  <head>
    <title>
      <?php if (!isset($title)) { ?>
        Mon super site
      <?php } else { echo $title; } ?>
    </title>

    <meta http-equiv="Content-type" content="text/html; charset=iso-
8859-1" />

    <link rel="stylesheet" href="/css/Envision.css" type="text/css"
/>
  </head>

  <body>
    <div id="wrap">
      <div id="header">
        <h1 id="logo-text"><a href="/">Mon super site</a></h1>
        <p id="slogan">Comment ça « il n'y a presque rien » ?</p>
      </div>

      <div  id="menu">
        <ul>
          <li><a href="/">Accueil</a></li>
          <?php if ($user->isAuthenticated()) { ?>
          <li><a href="/admin/">Admin</a></li>
          <li><a href="/admin/news-insert.html">Ajouter une
news</a></li>
          <?php } ?>
        </ul>
      </div>

      <div id="content-wrap">
        <div id="main">
          <?php if ($user->hasFlash()) echo '<p style="text-align:
center;">', $user->getFlash(), '</p>'; ?>

          <?php echo $content; ?>
        </div>
      </div>

      <div id="footer"></div>
    </div>
  </body>
</html>
```

Qu'est-ce que c'est que ces variables `$user` ?

La variable `$user` fait référence à l'instance de `User`. Elle doit être initialisée dans la méthode `getGeneratedPage()` de la classe `Page` :

Code : PHP - Extrait de /Library/Page.class.php

```
<?php
namespace Library;

class Page extends ApplicationComponent
{
  // ...

  public function getGeneratedPage()
  {
    if (!file_exists($this->contentFile))
    {
      throw new \InvalidArgumentException('La vue spécifiée
n\'existe pas');
    }

$user = $this->app->user();

    extract($this->vars);

    ob_start();
      require $this->contentFile;
    $content = ob_get_clean();

    ob_start();
      require dirname(__FILE__).'/../Applications/'.$this->app-
>name().'/Templates/layout.php';
    return ob_get_clean();
  }

  // ...
}
```

Notez que si vous utilisez la variable `$this`, elle fera référence à l'objet `Page` car le layout est inclut dans la méthode `Page::getGeneratedPage()`.

Les deux fichiers de configuration

Nous allons préparer le terrain en créant les deux fichiers de configuration dont nous avons besoin : les fichiers **app.xml** et **routes.xml**, pour l'instant quasi-vierges :

Code : XML - /Applications/Frontend/Config/app.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<definitions>
</definitions>
```

Code : XML - /Applications/Frontend/Config/routes.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<routes>
</routes>
```

L'instanciation de `FrontendApplication`

Pour instancier `FrontendApplication`, il va falloir créer un fichier **frontend.php**. Ce fichier devra d'abord inclure l'autoload. Celui-ci aura donc pour simple contenu :

Code : PHP - /Web/frontend.php

```php
<?php
require '../Library/autoload.php';

$app = new Applications\Frontend\FrontendApplication;
$app->run();
```

Réécrire toutes les URL

Il faut que toutes les URL pointent vers ce fichier. Pour cela, nous allons nous pencher vers l'URL rewriting. Voici le contenu du .htaccess :

Code : Apache - /web/.htaccess

```
RewriteEngine On

# Si le fichier auquel on tente d'accéder existe (si on veut
accéder à une image par exemple).
# Alors on ne réécrit pas l'URL.
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ frontend.php [QSA,L]
```

Le module de news

Nous allons commencer en douceur par un système de news. Pourquoi en douceur ? Car vous avez déjà fait cet exercice lors du précédent TP ! Ainsi, nous allons voir comment **l'intégrer** au sein de l'application, et vous verrez ainsi plus clair sur la manière dont elle fonctionne. Pour ne perdre personne, nous allons refaire le TP petit à petit pour mieux l'intégrer dans l'application. Ainsi, je vais commencer par vous rappeler ce que j'attends du système de news.

Fonctionnalités

Il doit être possible d'exécuter deux actions différentes sur le module de news :

- Afficher l'index du module. Cela aura pour effet de dévoiler les cinq dernières news avec le titre et l'extrait du contenu (seuls les 200 premiers caractères seront affichés).
- Afficher une news spécifique en cliquant sur son titre. L'auteur apparaîtra, ainsi que la date de modification si la news a été modifiée.

Comme pour tout module, commencez par créer les dossiers et fichiers de base, à savoir :

- Le dossier **/Applications/Frontend/Modules/News** qui contiendra notre module.
- Le fichier **/Applications/Frontend/Modules/News/NewsController.class.php** qui contiendra notre contrôleur.
- Le dossier **/Applications/Frontend/Modules/News/Views** qui contiendra les vues.
- Le fichier **/Library/Models/NewsManager.class.php** qui contiendra notre manager de base;
- Le fichier **/Library/Models/NewsManager_PDO.class.php** qui contiendra notre manager utilisant PDO.
- Le fichier **/Library/Entities/News.class.php** qui contiendra la classe représentant un enregistrement.

Structure de la table *news*

Une news est constituée d'un titre, d'un auteur et d'un contenu. Aussi, il faut stocker la date d'ajout de la news ainsi que sa date de modification. Cela nous donne une table **news** ressemblant à ceci :

Code : SQL

```
CREATE TABLE IF NOT EXISTS `news` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `auteur` varchar(30) COLLATE latin1_general_ci NOT NULL,
  `titre` varchar(100) COLLATE latin1_general_ci NOT NULL,
  `contenu` text COLLATE latin1_general_ci NOT NULL,
  `dateAjout` datetime NOT NULL,
  `dateModif` datetime NOT NULL,
  PRIMARY KEY (`id`)
) DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci ;
```

Nous pouvons désormais écrire la classe représentant cette entité : `News`.

Code : PHP - /Library/Entities/News.class.php

```
<?php
namespace Library\Entities;

class News extends \Library\Entity
{
  protected $auteur,
            $titre,
            $contenu,
            $dateAjout,
            $dateModif;

  const AUTEUR_INVALIDE = 1;
  const TITRE_INVALIDE = 2;
  const CONTENU_INVALIDE = 3;

  public function isValid()
  {
    return !(empty($this->auteur) || empty($this->titre) ||
empty($this->contenu));
  }


  // SETTERS //

  public function setAuteur($auteur)
  {
    if (!is_string($auteur) || empty($auteur))
    {
      $this->erreurs[] = self::AUTEUR_INVALIDE;
    }
    else
    {
      $this->auteur = $auteur;
    }
  }

  public function setTitre($titre)
  {
    if (!is_string($titre) || empty($titre))
    {
      $this->erreurs[] = self::TITRE_INVALIDE;
    }
    else
    {
      $this->titre = $titre;
    }
  }

  public function setContenu($contenu)
  {
    if (!is_string($contenu) || empty($contenu))
    {
```

```
            $this->erreurs[] = self::CONTENU_INVALIDE;
          }
          else
          {
            $this->contenu = $contenu;
          }
        }

        public function setDateAjout(\DateTime $dateAjout)
        {
          $this->dateAjout = $dateAjout;
        }

        public function setDateModif(\DateTime $dateModif)
        {
          $this->dateModif = $dateModif;
        }

        // GETTERS //

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

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

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

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

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

L'action *index*

La route

Commençons par implémenter cette action. La première chose à faire est de créer une nouvelle route : quelle URL pointera vers cette action ? Je vous propose simplement que ce soit la racine du site web, donc ce sera l'URL /. Pour créer cette route, il va falloir modifier notre fichier de configuration et y ajouter cette ligne :

Code : XML - Extrait de /Applications/Frontend/Config/routes.xml

```
    <route url="/" module="News" action="index" />
```

Vient ensuite l'implémentation de l'action dans le contrôleur.

Le contrôleur

Qui dit nouvelle action dit nouvelle méthode, et cette méthode c'est `executeIndex()`. Cette méthode devra récupérer les cinq dernières news (le nombre cinq devra être stocké dans le fichier de configuration de l'application, à savoir **/Applications/Frontend/Config/app.xml**). Il faudra parcourir cette liste de news afin de n'assigner aux news qu'un contenu de 200 caractères au maximum. Ensuite, il faut passer la liste des news à la vue :

Code : PHP - /Applications/Frontend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Frontend\Modules\News;

class NewsController extends \Library\BackController
{
  public function executeIndex(\Library\HTTPRequest $request)
  {
    $nombreNews = $this->app->config()->get('nombre_news');
    $nombreCaracteres = $this->app->config()->get('nombre_caracteres');

    // On ajoute une définition pour le titre.
    $this->page->addVar('title', 'Liste des '.$nombreNews.' dernières news');

    // On récupère le manager des news.
    $manager = $this->managers->getManagerOf('News');

    // Cette ligne, vous ne pouviez pas la deviner sachant qu'on n'a pas encore touché au modèle.
    // Contentez-vous donc d'écrire cette instruction, nous implémenterons la méthode ensuite.
    $listeNews = $manager->getList(0, $nombreNews);

    foreach ($listeNews as $news)
    {
      if (strlen($news->contenu()) > $nombreCaracteres)
      {
        $debut = substr($news->contenu(), 0, $nombreCaracteres);
        $debut = substr($debut, 0, strrpos($debut, ' ')) . '...';

        $news->setContenu($debut);
      }
    }

    // On ajoute la variable $listeNews à la vue.
    $this->page->addVar('listeNews', $listeNews);
  }
}
```

Comme vous le voyez, j'utilise le fichier de configuration pour récupérer le nombre de news à afficher et le nombre maximum de caractères. Voici le fichier de configuration :

Code : XML - /Applications/Frontend/Config/app.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions>
  <define var="nombre_news" value="5" />
  <define var="nombre_caracteres" value="200" />
</definitions>
```

La vue

À toute action correspond une vue du même nom. Ici, la vue à créer sera **/Applications/Frontend/Modules/News/Views/index.php**. Voici un exemple très simple pour cette vue :

Code : PHP - /Applications/Frontend/Modules/News/Views/index.php

```
<?php
foreach ($listeNews as $news)
{
?>
  <h2><a href="news-<?php echo $news['id']; ?>.html"><?php echo
$news['titre']; ?></a></h2>
  <p><?php echo nl2br($news['contenu']); ?></p>
<?php
}
```

Notez l'utilisation des news comme des tableaux : cela est possible du fait que l'objet est une instance d'une classe qui implémente `ArrayAccess`.

Le modèle

Nous allons modifier deux classes faisant partie du modèle, à savoir `NewsManager` et `NewsManager_PDO`. Nous allons implémenter à cette dernière classe une méthode : `getList()`. Sa classe parente doit donc aussi être modifiée pour déclarer cette méthode.

Code : PHP - /Library/Models/NewsManager.class.php

```
<?php
namespace Library\Models;

abstract class NewsManager extends \Library\Manager
{
  /**
* Méthode retournant une liste de news demandée
* @param $debut int La première news à sélectionner
* @param $limite int Le nombre de news à sélectionner
* @return array La liste des news. Chaque entrée est une instance
de News.
*/
  abstract public function getList($debut = -1, $limite = -1);
}
```

Code : PHP - /Library/Models/NewsManager_PDO.class.php

```
<?php
namespace Library\Models;

use \Library\Entities\News;

class NewsManager_PDO extends NewsManager
{
  public function getList($debut = -1, $limite = -1)
  {
    $sql = 'SELECT id, auteur, titre, contenu, dateAjout, dateModif
FROM news ORDER BY id DESC';

    if ($debut != -1 || $limite != -1)
    {
      $sql .= ' LIMIT '.(int) $limite.' OFFSET '.(int) $debut;
    }

    $requete = $this->dao->query($sql);
    $requete->setFetchMode(\PDO::FETCH_CLASS |
\PDO::FETCH_PROPS_LATE, '\Library\Entities\News');

    $listeNews = $requete->fetchAll();
```

```
        foreach ($listeNews as $news)
        {
          $news->setDateAjout(new \DateTime($news->dateAjout()));
          $news->setDateModif(new \DateTime($news->dateModif()));
        }

        $requete->closeCursor();

        return $listeNews;
      }
    }
```

Vous pouvez faire le test : accédez à la racine de votre site et vous découvrirez... un gros blanc, car aucune news n'est présente en BDD. Vous pouvez en ajouter manuellement *via* phpMyAdmin en attendant que nous ayons construit l'espace d'administration.

Si la constante `PDO::FETCH_CLASS` vous est inconnue, je vous invite à relire la première partie du TP sur la réalisation d'un système de news.

L'action *show*

La route

Je vous propose que les URL du type **news-id.html** pointent vers cette action. Modifiez donc le fichier de configuration des routes pour y ajouter celle-ci :

Code : XML - /Applications/Frontend/Config/routes.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/" module="News" action="index" />
<route url="/news-([0-9]+)\.html" module="News" action="show"
vars="id"/>
</routes>
```

Le contrôleur

Le contrôleur implémentera la méthode `executeShow()`. Son contenu est simple : le contrôleur ira demander au manager la news correspondant à l'identifiant puis, il passera cette news à la vue, en ayant pris soin de remplacer les sauts de lignes par des balises **
** dans le contenu de la news.

Si la news n'existe pas, il faudra rediriger l'utilisateur vers une erreur 404.

Code : PHP - /Applications/Frontend/Modules/News

```
<?php
namespace Applications\Frontend\Modules\News;

class NewsController extends \Library\BackController
{
  public function executeIndex(\Library\HTTPRequest $request)
  {
    $nombreNews = $this->app->config()->get('nombre_news');
    $nombreCaracteres = $this->app->config()-
>get('nombre_caracteres');
```

```
        // On ajoute une définition pour le titre.
        $this->page->addVar('title', 'Liste des '.$nombreNews.'
    dernières news');

        // On récupère le manager des news.
        $manager = $this->managers->getManagerOf('News');

        $listeNews = $manager->getList(0, $nombreNews);

        foreach ($listeNews as $news)
        {
          if (strlen($news->contenu()) > $nombreCaracteres)
          {
            $debut = substr($news->contenu(), 0, $nombreCaracteres);
            $debut = substr($debut, 0, strrpos($debut, ' ')) . '...';

            $news->setContenu($debut);
          }
        }

        // On ajoute la variable $listeNews à la vue.
        $this->page->addVar('listeNews', $listeNews);
      }

      public function executeShow(\Library\HTTPRequest $request)
      {
        $news = $this->managers->getManagerOf('News')-
    >getUnique($request->getData('id'));

        if (empty($news))
        {
          $this->app->httpResponse()->redirect404();
        }

        $this->page->addVar('title', $news->titre());
        $this->page->addVar('news', $news);
      }
    }
```

La vue

La vue se contente de gérer l'affichage de la news. Faites comme bon vous semble, cela n'a pas trop d'importance. Voici la version que je vous propose :

Code : PHP - /Applications/Frontend/Modules/News/Views/show.php

```
<p>Par <em><?php echo $news['auteur']; ?></em>, le <?php echo
$news['dateAjout']->format('d/m/Y à H\hi'); ?></p>
<h2><?php echo $news['titre']; ?></h2>
<p><?php echo nl2br($news['contenu']); ?></p>

<?php if ($news['dateAjout'] != $news['dateModif']) { ?>
  <p style="text-align: right;"><small><em>Modifiée le <?php echo
$news['dateModif']->format('d/m/Y à H\hi'); ?></em></small></p>
<?php } ?>
```

Le modèle

Nous allons là aussi toucher à nos classes `NewsManager` et `NewsManager_PDO` en ajoutant la méthode `getUnique()`.

Code : PHP - /Library/Models/NewsManager.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\News;

abstract class NewsManager extends \Library\Manager
{
  /**
* Méthode retournant une liste de news demandée.
* @param $debut int La première news à sélectionner
* @param $limite int Le nombre de news à sélectionner
* @return array La liste des news. Chaque entrée est une instance
de News.
*/
  abstract public function getList($debut = -1, $limite = -1);

  /**
* Méthode retournant une news précise.
* @param $id int L'identifiant de la news à récupérer
* @return News La news demandée
*/
  abstract public function getUnique($id);
}
```

Code : PHP - /Library/Models/NewsManager_PDO.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\News;

class NewsManager_PDO extends NewsManager
{
  public function getList($debut = -1, $limite = -1)
  {
    $sql = 'SELECT id, auteur, titre, contenu, dateAjout, dateModif
FROM news ORDER BY id DESC';

    if ($debut != -1 || $limite != -1)
    {
      $sql .= ' LIMIT '.(int) $limite.' OFFSET '.(int) $debut;
    }

    $requete = $this->dao->query($sql);
    $requete->setFetchMode(\PDO::FETCH_CLASS |
\PDO::FETCH_PROPS_LATE, '\Library\Entities\News');

    $listeNews = $requete->fetchAll();

    foreach ($listeNews as $news)
    {
      $news->setDateAjout(new \DateTime($news->dateAjout()));
      $news->setDateModif(new \DateTime($news->dateModif()));
    }

    $requete->closeCursor();

    return $listeNews;
  }

  public function getUnique($id)
  {
    $requete = $this->dao->prepare('SELECT id, auteur, titre,
contenu, dateAjout, dateModif FROM news WHERE id = :id');
    $requete->bindValue(':id', (int) $id, \PDO::PARAM_INT);
    $requete->execute();

    $requete->setFetchMode(\PDO::FETCH_CLASS |
```

```
\PDO::FETCH_PROPS_LATE, '\Library\Entities\News');

    if ($news = $requete->fetch())
    {
      $news->setDateAjout(new \DateTime($news->dateAjout()));
      $news->setDateModif(new \DateTime($news->dateModif()));

      return $news;
    }

    return null;
  }
}
```

Ajoutons des commentaires

Cahier des charges

Nous allons ajouter une action à notre module de news : l'ajout d'un commentaire. Il ne faudra pas oublier de modifier notre module de news, et plus spécialement l'action **show** pour laisser apparaître la liste des commentaires ainsi qu'un lien menant au formulaire d'ajout de commentaire.

Avant toute chose, il va falloir créer les modèles nous permettant d'interagir avec la BDD pour accéder aux commentaires :

- Le fichier **/Library/Models/CommentsManager.class.php** qui contiendra notre manager de base.
- Le fichier **/Library/Models/CommentsManager_PDO.class.php** qui inclura notre manager utilisant PDO.
- Le fichier **/Library/Entities/Comment.class.php** qui comportera la classe représentant un enregistrement.

Structure de la table *comments*

Un commentaire est assigné à une news. Il est constitué d'un auteur et d'un contenu, ainsi que de sa date d'enregistrement. Notre table **comments** doit donc être constituée de la sorte :

Code : SQL

```
CREATE TABLE IF NOT EXISTS `comments` (
  `id` mediumint(9) NOT NULL AUTO_INCREMENT,
  `news` smallint(6) NOT NULL,
  `auteur` varchar(50) COLLATE latin1_general_ci NOT NULL,
  `contenu` text COLLATE latin1_general_ci NOT NULL,
  `date` datetime NOT NULL,
  PRIMARY KEY (`id`)
) DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci ;
```

Puisque nous connaissons la structure d'un commentaire, nous pouvons écrire une partie du modèle, à savoir la classe `Comment` :

Code : PHP - /Library/Entities/Comment.class.php

```
<?php
namespace Library\Entities;

class Comment extends \Library\Entity
{
  protected $news,
            $auteur,
            $contenu,
            $date;

  const AUTEUR_INVALIDE = 1;
  const CONTENU_INVALIDE = 2;

  public function isValid()
```

```
    {
      return !(empty($this->auteur) || empty($this->contenu));
    }

    // SETTERS

    public function setNews($news)
    {
      $this->news = (int) $news;
    }

    public function setAuteur($auteur)
    {
      if (!is_string($auteur) || empty($auteur))
      {
        $this->erreurs[] = self::AUTEUR_INVALIDE;
      }
      else
      {
        $this->auteur = $auteur;
      }
    }

    public function setContenu($contenu)
    {
      if (!is_string($contenu) || empty($contenu))
      {
        $this->erreurs[] = self::CONTENU_INVALIDE;
      }
      else
      {
        $this->contenu = $contenu;
      }
    }

    public function setDate(\DateTime $date)
    {
      $this->date = $date;
    }

    // GETTERS

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

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

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

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

L'action *insertComment*

La route

Nous n'allons pas faire dans la fantaisie pour cette action, nous allons prendre une URL basique : **commenter-idnews.html**. Rajoutez donc cette nouvelle route dans le fichier de configuration des routes :

Code : XML - /Applications/Frontend/Config/routes.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/" module="News" action="index" />
  <route url="/news-([0-9]+)\.html" module="News" action="show"
vars="id"/>
<route url="/commenter-([0-9]+)\.html" module="News"
action="insertComment" vars="news" />
</routes>
```

La vue

Dans un premier temps, nous allons nous attarder sur la vue car c'est à l'intérieur de celle-ci que nous allons construire le formulaire. Cela nous permettra donc de savoir quels champs seront à traiter par le contrôleur. Je vous propose un formulaire très simple demandant le pseudo et le contenu à l'utilisateur :

Code : PHP - /Applications/Frontend/Modules/News/Views/insertComment.php

```
<h2>Ajouter un commentaire</h2>
<form action="" method="post">
  <p>
    <?php if (isset($erreurs) &&
in_array(\Library\Entities\Comment::AUTEUR_INVALIDE, $erreurs)) echo
'L\'auteur est invalide.<br />'; ?>
    <label>Pseudo</label>
    <input type="text" name="pseudo" value="<?php if
(isset($comment)) echo htmlspecialchars($comment['auteur']); ?>"
/><br />

    <?php if (isset($erreurs) &&
in_array(\Library\Entities\Comment::CONTENU_INVALIDE, $erreurs))
echo 'Le contenu est invalide.<br />'; ?>
    <label>Contenu</label>
    <textarea name="contenu" rows="7" cols="50"><?php if
(isset($comment)) echo htmlspecialchars($comment['contenu']); ?
></textarea><br />

    <input type="submit" value="Commenter" />
  </p>
</form>
```

L'identifiant de la news est stocké dans l'URL. Puisque nous allons envoyer le formulaire sur cette même page, l'identifiant de la news sera toujours présent dans l'URL et donc accessible *via* le contrôleur.

Le contrôleur

Notre méthode `executeInsertComment()` se chargera dans un premier temps de vérifier si le formulaire a été envoyé en vérifiant si la variable POST `pseudo` existe. Ensuite, elle procédera à la vérification des données et insérera le commentaire en BDD si toutes les données sont valides.

Code : PHP - /Applications/Frontend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Frontend\Modules\News;

class NewsController extends \Library\BackController
```

```
{
  // ...

  public function executeInsertComment(\Library\HTTPRequest
$request)
  {
    $this->page->addVar('title', 'Ajout d\'un commentaire');

    if ($request->postExists('pseudo'))
    {
      $comment = new \Library\Entities\Comment(array(
        'news' => $request->getData('news'),
        'auteur' => $request->postData('pseudo'),
        'contenu' => $request->postData('contenu')
      ));

      if ($comment->isValid())
      {
        $this->managers->getManagerOf('Comments')->save($comment);

        $this->app->user()->setFlash('Le commentaire a bien été
ajouté, merci !');

        $this->app->httpResponse()->redirect('news-'.$request-
>getData('news').'.html');
      }
      else
      {
        $this->page->addVar('erreurs', $comment->erreurs());
      }

      $this->page->addVar('comment', $comment);
    }
  }

  // ...
}
```

Le modèle

Nous aurons besoin d'implémenter une méthode dans notre classe `CommentsManager` : `save()`. En fait, il s'agit d'un « raccourci », cette méthode appelle elle-même une autre méthode : `add()` ou `modify()` selon si le commentaire est déjà présent en BDD. Notre manager peut savoir si l'enregistrement est déjà enregistré ou pas grâce à la méthode `isNew()`.

Code : PHP - /Library/Models/CommentsManager.class.php

```
<?php
namespace Library\Models;

use \Library\Entities\Comment;

abstract class CommentsManager extends \Library\Manager
{
  /**
* Méthode permettant d'ajouter un commentaire
* @param $comment Le commentaire à ajouter
* @return void
*/
  abstract protected function add(Comment $comment);

  /**
* Méthode permettant d'enregistrer un commentaire.
* @param $comment Le commentaire à enregistrer
* @return void
*/
  public function save(Comment $comment)
```

```
    {
      if ($comment->isValid())
      {
        $comment->isNew() ? $this->add($comment) : $this->modify($comment);
      }
      else
      {
        throw new \RuntimeException('Le commentaire doit être validé pour être enregistré');
      }
    }
  }
```

Code : PHP - /Library/Models/CommentsManager_PDO.class.php

```
<?php
namespace Library\Models;

use \Library\Entities\Comment;

class CommentsManager_PDO extends CommentsManager
{
  protected function add(Comment $comment)
  {
    $q = $this->dao->prepare('INSERT INTO comments SET news = :news, auteur = :auteur, contenu = :contenu, date = NOW()');

    $q->bindValue(':news', $comment->news(), \PDO::PARAM_INT);
    $q->bindValue(':auteur', $comment->auteur());
    $q->bindValue(':contenu', $comment->contenu());

    $q->execute();

    $comment->setId($this->dao->lastInsertId());
  }
}
```

L'implémentation de la méthode `modify()` se fera lors de la construction de l'espace d'administration.

Affichage des commentaires

Modification du contrôleur

Il suffit simplement de passer la liste des commentaires à la vue. Une seule instruction suffit donc :

Code : PHP - /Applications/Frontend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Frontend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeShow(\Library\HTTPRequest $request)
  {
    $news = $this->managers->getManagerOf('News')->getUnique($request->getData('id'));

    if (empty($news))
    {
      $this->app->httpResponse()->redirect404();
```

```
        }

        $this->page->addVar('title', $news->titre());
        $this->page->addVar('news', $news);
$this->page->addVar('comments', $this->managers->getManagerOf('Comments')->getListOf($news->id()));
      }
}
```

Modification de la vue affichant une news

La vue devra parcourir la liste des commentaires passés pour les afficher. Les liens pointant vers l'ajout d'un commentaire devront aussi figurer sur la page.

Code : PHP - /Applications/Frontend/Modules/News/Views/show.php

```
<p>Par <em><?php echo $news['auteur']; ?></em>, le <?php echo $news['dateAjout']->format('d/m/Y à H\hi'); ?></p>
<h2><?php echo $news['titre']; ?></h2>
<p><?php echo nl2br($news['contenu']); ?></p>

<?php if ($news['dateAjout'] != $news['dateModif']) { ?>
  <p style="text-align: right;"><small><em>Modifiée le <?php echo $news['dateModif']->format('d/m/Y à H\hi'); ?></em></small></p>
<?php } ?>

<p><a href="commenter-<?php echo $news['id']; ?>.html">Ajouter un commentaire</a></p>

<?php
if (empty($comments))
{
?>
<p>Aucun commentaire n'a encore été posté. Soyez le premier à en laisser un !</p>
<?php
}

foreach ($comments as $comment)
{
?>
  <fieldset>
    <legend>
      Posté par <strong><?php echo htmlspecialchars($comment['auteur']); ?></strong> le <?php echo $comment['date']->format('d/m/Y à H\hi'); ?>
    </legend>
    <p><?php echo nl2br(htmlspecialchars($comment['contenu'])); ?></p>
  </fieldset>
<?php
}
?>

<p><a href="commenter-<?php echo $news['id']; ?>.html">Ajouter un commentaire</a></p>
```

Modification du manager des commentaires

Le manager des commentaires devra implémenter la méthode `getListOf()` dont a besoin notre contrôleur pour bien fonctionner. Voici la version que je vous propose :

Code : PHP - /Library/Models/CommentsManager.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\Comment;

abstract class CommentsManager extends \Library\Manager
{
  /**
* Méthode permettant d'ajouter un commentaire.
* @param $comment Le commentaire à ajouter
* @return void
*/
  abstract protected function add(Comment $comment);

  /**
* Méthode permettant d'enregistrer un commentaire.
* @param $comment Le commentaire à enregistrer
* @return void
*/
  public function save(Comment $comment)
  {
    if ($comment->isValid())
    {
      $comment->isNew() ? $this->add($comment) : $this-
>modify($comment);
    }
    else
    {
      throw new \RuntimeException('Le commentaire doit être validé
pour être enregistré');
    }
  }

  /**
* Méthode permettant de récupérer une liste de commentaires.
* @param $news La news sur laquelle on veut récupérer les
commentaires
* @return array
*/
  abstract public function getListOf($news);
}
```

Code : PHP - /Library/Models/CommentsManager_PDO.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\Comment;

class CommentsManager_PDO extends CommentsManager
{
  protected function add(Comment $comment)
  {
    $q = $this->dao->prepare('INSERT INTO comments SET news = :news,
auteur = :auteur, contenu = :contenu, date = NOW()');

    $q->bindValue(':news', $comment->news(), \PDO::PARAM_INT);
    $q->bindValue(':auteur', $comment->auteur());
    $q->bindValue(':contenu', $comment->contenu());

    $q->execute();

    $comment->setId($this->dao->lastInsertId());
  }

  public function getListOf($news)
```

```
    {
      if (!ctype_digit($news))
      {
        throw new \InvalidArgumentException('L\'identifiant de la news
passé doit être un nombre entier valide');
      }

      $q = $this->dao->prepare('SELECT id, news, auteur, contenu, date
FROM comments WHERE news = :news');
      $q->bindValue(':news', $news, \PDO::PARAM_INT);
      $q->execute();

      $q->setFetchMode(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
'\Library\Entities\Comment');

      $comments = $q->fetchAll();

      foreach ($comments as $comment)
      {
        $comment->setDate(new \DateTime($comment->date()));
      }

      return $comments;
    }
  }
```

Le backend

Notre application est composée d'un système de news avec commentaires. Or, nous ne pouvons actuellement pas ajouter de news, ni modérer les commentaires. Pour ce faire, nous allons créer un espace d'administration, qui n'est autre ici que le **backend**. Cet espace d'administration sera ainsi composé d'un système de gestion de news (ajout, modification et suppression) ainsi que d'un système de gestion de commentaires (modification et suppression).

Ayant déjà créé le *frontend*, ce chapitre devrait être plus facile pour vous. Néanmoins, une nouveauté fait son apparition : celle d'interdire le contenu de l'application aux visiteurs. Ne traînons donc pas, nous avons pas mal de travail qui nous attend !

L'application

Comme pour l'application *Frontend*, nous aurons besoin de créer les fichiers de base : la classe représentant l'application, le layout, les deux fichiers de configuration et le fichier qui instanciera notre classe. Assurez-vous également d'avoir créé le dossier **/Applications/Backend**.

La classe `BackendApplication`

Cette classe ne sera pas strictement identique à la classe `FrontendApplication`. En effet, nous devons **sécuriser** l'application afin que seuls les utilisateurs authentifiés y aient accès.

Pour rappel, voici le fonctionnement de la méthode `run()` de la classe **FrontendApplication** :

- Obtention du contrôleur grâce à la méthode parente `getController()`.
- Exécution du contrôleur.
- Assignation de la page créée par le contrôleur à la réponse.
- Envoi de la réponse.

La classe `BackendApplication` fonctionnera de la même façon, à la différence près que la première instruction ne sera exécutée que si l'utilisateur est authentifié. Sinon, nous allons récupérer le contrôleur du module de connexion que nous allons créer dans ce chapitre. Voici donc le fonctionnement de la méthode `run()` de la classe `BackendApplication` :

- Si l'utilisateur est authentifié :
  - obtention du contrôleur grâce à la méthode parente `getController()`.
- Sinon :
  - instanciation du contrôleur du module de connexion.
- Exécution du contrôleur.
- Assignation de la page créée par le contrôleur à la réponse.
- Envoi de la réponse.

Aide : nous avons un attribut `$user` dans notre classe qui représente l'utilisateur. Regardez les méthodes que nous lui avons implémentées si vous ne savez pas comment on peut savoir s'il est authentifié ou non.

Rappel : n'oubliez pas d'inclure l'autoload au début du fichier !

Vous devriez avoir une classe de ce type :

Code : PHP - /Applications/Backend/BackendApplication.class.php

```php
<?php
namespace Applications\Backend;

class BackendApplication extends \Library\Application
{
  public function __construct()
  {
    parent::__construct();

    $this->name = 'Backend';
  }
```

```
  public function run()
  {
    if ($this->user->isAuthenticated())
    {
      $controller = $this->getController();
    }
    else
    {
      $controller = new Modules\Connexion\ConnexionController($this,
'Connexion', 'index');
    }

    $controller->execute();

    $this->httpResponse->setPage($controller->page());
    $this->httpResponse->send();
  }
}
```

Le layout

Le layout est le même que celui du *frontend*. Sachez qu'en pratique, cela est rare et vous aurez généralement deux layouts différents (chaque application a ses spécificités). Cependant, ici il n'est pas nécessaire de faire deux fichiers différents. Vous pouvez donc soit copier/coller le layout du frontend dans le dossier **/Applications/Backend/Templates**, soit créer le layout et inclure celui du frontend :

Code : PHP - /Applications/Backend/Templates/layout.php

```
<?php require
dirname(__FILE__).'/../../Frontend/Templates/layout.php';
```

Les deux fichiers de configuration

Là aussi il faut créer les deux fichiers de configuration. Mettez-y, comme nous l'avons fait au précédent chapitre, les structures de base.

Code : XML - /Applications/Backend/Config/app.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<definitions>
</definitions>
```

Code : XML - /Applications/Backend/Config/routes.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<routes>
</routes>
```

L'instanciation de `BackendApplication`

L'instanciation de `BackendApplication` se fera dans le fichier **/Web/backend.php**, comme nous avons procédé pour l'instanciation de `FrontendApplication`.

Rappel : pensez à inclure l'autoload avant d'instancier la classe `BackendApplication` !

Code : PHP - /Web/backend.php

```php
<?php
require '../Library/autoload.php';

$app = new Applications\Backend\BackendApplication;
$app->run();
```

Réécrire les URL

Nous allons maintenant modifier le fichier .htaccess. Actuellement, toutes les URL sont redirigées vers **frontend.php**. Nous garderons toujours cette règle, mais nous allons d'abord en ajouter une autre : nous allons rediriger toutes les URL commençant par **admin/** vers **backend.php**. Vous êtes libres de choisir un autre préfixe, mais il faut bien choisir quelque chose pour sélectionner l'application concernée par l'URL.

Le .htaccess ressemble donc à ceci :

Code : Apache - /Web/.htaccess

```
RewriteEngine On

RewriteRule ^admin/ backend.php [QSA,L]

RewriteCond %{REQUEST_FILENAME} !-f

RewriteRule ^(.*)$ frontend.php [QSA,L]
```

Le module de connexion

Ce module est un peu particulier. En effet, aucune route ne sera définie pour pointer vers ce module. De plus, ce module ne nécessite aucun stockage de données, nous n'aurons donc pas de modèle. La seule fonctionnalité attendue du module est d'afficher son index. Cet index aura pour rôle d'afficher le formulaire de connexion et de traiter les données de ce formulaire.

Avant de commencer à construire le module, nous allons préparer le terrain. Où stocker l'identifiant et le mot de passe permettant d'accéder à l'application ? Je vous propose tout simplement d'ajouter deux définitions dans le fichier de configuration de l'application :

Code : XML - /Applications/Backend/Config/app.xml

```xml
<?xml version="1.0" encoding="utf-8" ?>
<definitions>
  <define var="login" value="admin" />
  <define var="pass" value="mdp" />
</definitions>
```

Vous pouvez maintenant, si ce n'est pas déjà fait, créer le dossier **/Applications/Backend/Modules/Connexion**.

La vue

Nous allons débuter par créer la vue correspondant à l'index du module. Ce sera un simple formulaire demandant le nom d'utilisateur et le mot de passe à l'internaute :

Code : PHP - /Applications/Backend/Modules/Connexion/Views/index.php

```php
<h2>Connexion</h2>

<form action="" method="post">
```

```
    <label>Pseudo</label>
    <input type="text" name="login" /><br />

    <label>Mot de passe</label>
    <input type="password" name="password" /><br /><br />

    <input type="submit" value="Connexion" />
  </form>
```

Le contrôleur

Procédons maintenant à l'élaboration du contrôleur. Ce contrôleur implémentera une seule méthode : `executeIndex()`. Cette méthode devra, si le formulaire a été envoyé, vérifier si le pseudo et le mot de passe entrés sont corrects. Si c'est le cas, l'utilisateur est authentifié, sinon un message d'erreur s'affiche.

Code : PHP - /Applications/Backend/Modules/Connexion/ConnexionController.class.php

```
<?php
namespace Applications\Backend\Modules\Connexion;

class ConnexionController extends \Library\BackController
{
  public function executeIndex(\Library\HTTPRequest $request)
  {
    $this->page->addVar('title', 'Connexion');

    if ($request->postExists('login'))
    {
      $login = $request->postData('login');
      $password = $request->postData('password');

      if ($login == $this->app->config()->get('login') && $password
== $this->app->config()->get('pass'))
      {
        $this->app->user()->setAuthenticated(true);
        $this->app->httpResponse()->redirect('.');
      }
      else
      {
        $this->app->user()->setFlash('Le pseudo ou le mot de passe
est incorrect.');
      }
    }
  }
}
```

Ce fut court, mais essentiel. Nous venons de sécuriser en un rien de temps l'application toute entière. De plus, ce module est réutilisable dans d'autres projets ! En effet, rien ne le lie à cette application. À partir du moment où l'application aura un fichier de configuration adapté (c'est-à-dire qu'il a déclaré les variables `login` et `pass`), alors elle pourra s'en servir. 

Le module de news

Nous allons maintenant attaquer le module de news sur notre application. Comme d'habitude, nous commençons par la liste des fonctionnalités attendues.

Fonctionnalités

Ce module doit nous permettre de gérer le contenu de la base de données. Par conséquent, nous devons avoir quatre actions :

- L'action **index** qui nous affiche la liste des news avec des liens pour les modifier ou supprimer.
- L'action **insert** pour ajouter une news.
- L'action **update** pour modifier une news.
- L'action **delete** pour supprimer une news.

L'action *index*

La route

Tout d'abord, définissons l'URL qui pointera vers cette action. Je vous propose tout simplement que ce soit l'accueil de l'espace d'administration :

Code : XML - /Applications/Backend/Config/routes.xml

```xml
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/admin/" module="News" action="index" />
</routes>
```

Le contrôleur

Le contrôleur se chargera uniquement de passer la liste des news à la vue ainsi que le nombre de news présent. Le contenu de la méthode est donc assez simple :

Code : PHP - /Applications/Backend/Modules/News/NewsController.class.php

```php
<?php
namespace Applications\Backend\Modules\News;

class NewsController extends \Library\BackController
{
  public function executeIndex(\Library\HTTPRequest $request)
  {
    $this->page->addVar('title', 'Gestion des news');

    $manager = $this->managers->getManagerOf('News');

    $this->page->addVar('listeNews', $manager->getList());
    $this->page->addVar('nombreNews', $manager->count());
  }
}
```

Comme vous le voyez, nous nous resservons de la méthode `getList()` que nous avions implémentée au cours du précédent chapitre au cours de la construction du *frontend*. Cependant, il nous reste à implémenter une méthode dans notre manager : `count()`.

Le modèle

La méthode `count()` est très simple : elle ne fait qu'exécuter une requête pour renvoyer le résultat.

Code : PHP - /Library/Models/NewsManager.class.php

```php
<?php
namespace Library\Models;

abstract class NewsManager extends \Library\Manager
{
  /**
* Méthode renvoyant le nombre de news total.
* @return int
*/
  abstract public function count();
```

```
    // ...
  }
```

Code : PHP - /Library/Models/NewsManager_PDO.class.php

```
<?php
namespace Library\Models;

class NewsManager_PDO extends NewsManager
{
  public function count()
  {
    return $this->dao->query('SELECT COUNT(*) FROM news')->fetchColumn();
  }

  // ...
}
```

La vue

La vue se contente de parcourir le tableau de news pour en afficher les données. Faites dans la simplicité. 

Code : PHP - /Applications/Backend/Modules/News/Views/index.php

```
<p style="text-align: center">Il y a actuellement <?php echo $nombreNews; ?> news. En voici la liste :</p>

<table>
  <tr><th>Auteur</th><th>Titre</th><th>Date d'ajout</th><th>Dernière modification</th><th>Action</th></tr>
<?php
foreach ($listeNews as $news)
{
  echo '<tr><td>', $news['auteur'], '</td><td>', $news['titre'], '</td><td>le ', $news['dateAjout']->format('d/m/Y à H\hi'), '</td><td>', ($news['dateAjout'] == $news['dateModif'] ? '-' : 'le '.$news['dateModif']->format('d/m/Y à H\hi')), '</td><td><a href="news-update-', $news['id'], '.html"><img src="/images/update.png" alt="Modifier" /></a> <a href="news-delete-', $news['id'], '.html"><img src="/images/delete.png" alt="Supprimer" /></a></td></tr>', "\n";
}
?>
</table>
```

L'action *insert*

La route

Je vous propose que l'URL qui pointera vers cette action soit **/admin/news-insert.html**. Je pense que maintenant vous savez comment définir une route, mais je remets le fichier pour que tout le monde suive bien :

Code : XML - /Applications/Backend/Config/routes.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/admin/" module="News" action="index" />
  <route url="/admin/news-insert\.html" module="News"
```

```
    action="insert" />
    </routes>
```

Le contrôleur

Le contrôleur vérifie si le formulaire a été envoyé. Si c'est le cas, alors il procédera à la vérification des données et insérera la news en BDD si tout est valide. Cependant, il y a un petit problème : lorsque nous implémenterons l'action *update*, nous allons devoir réécrire la partie « traitement du formulaire » car la validation des données suit la même logique. Nous allons donc créer une autre méthode au sein du contrôleur, nommée `processForm()`, qui se chargera de traiter le formulaire et d'enregistrer la news en BDD.

Rappel : le manager contient une méthode `save()` qui se chargera soit d'ajouter la news si elle est nouvelle, soit de la mettre à jour si elle est déjà enregistrée. C'est cette méthode que vous devez invoquer.

Code : PHP - /Applications/Backend/Modules/News/NewsController.class.php

```php
<?php
namespace Applications\Backend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeInsert(\Library\HTTPRequest $request)
  {
    if ($request->postExists('auteur'))
    {
      $this->processForm($request);
    }

    $this->page->addVar('title', 'Ajout d\'une news');
  }

  public function processForm(\Library\HTTPRequest $request)
  {
    $news = new \Library\Entities\News(
      array(
        'auteur' => $request->postData('auteur'),
        'titre' => $request->postData('titre'),
        'contenu' => $request->postData('contenu')
      )
    );

    // L'identifiant de la news est transmis si on veut la
modifier.
    if ($request->postExists('id'))
    {
      $news->setId($request->postData('id'));
    }

    if ($news->isValid())
    {
      $this->managers->getManagerOf('News')->save($news);

      $this->app->user()->setFlash($news->isNew() ? 'La news a bien
été ajoutée !' : 'La news a bien été modifiée !');
    }
    else
    {
      $this->page->addVar('erreurs', $news->erreurs());
    }

    $this->page->addVar('news', $news);
  }
```

```
}
```

Le modèle

Nous allons implémenter les méthodes `save()` et `add()` dans notre manager afin que notre contrôleur puisse être fonctionnel.

Rappel : la méthode `save()` s'implémente directement dans `NewsManager` puisqu'elle ne dépend pas du DAO.

Code : PHP - /Library/Models/NewsManager.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\News;

abstract class NewsManager extends \Library\Manager
{
  /**
* Méthode permettant d'ajouter une news.
* @param $news News La news à ajouter
* @return void
*/
  abstract protected function add(News $news);

  /**
* Méthode permettant d'enregistrer une news.
* @param $news News la news à enregistrer
* @see self::add()
* @see self::modify()
* @return void
*/
  public function save(News $news)
  {
    if ($news->isValid())
    {
      $news->isNew() ? $this->add($news) : $this->modify($news);
    }
    else
    {
      throw new \RuntimeException('La news doit être validée pour
être enregistrée');
    }
  }

  // ...
}
```

Code : PHP - /Library/Models/NewsManager_PDO.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\News;

class NewsManager_PDO extends NewsManager
{
  protected function add(News $news)
  {
    $requete = $this->dao->prepare('INSERT INTO news SET auteur =
:auteur, titre = :titre, contenu = :contenu, dateAjout = NOW(),
dateModif = NOW()');
```

```
      $requete->bindValue(':titre', $news->titre());
      $requete->bindValue(':auteur', $news->auteur());
      $requete->bindValue(':contenu', $news->contenu());

      $requete->execute();
    }

    // ...
}
```

La vue

Là aussi, nous utiliserons de la duplication de code pour afficher le formulaire. En effet, la vue correspondant à l'action *update* devra également afficher ce formulaire. Nous allons donc créer un fichier qui contiendra ce formulaire et qui sera inclus au sein des vues. Je vous propose de l'appeler **_form.php** (le _ est ici utilisé pour bien indiquer qu'il ne s'agit pas d'une vue mais d'un élément à inclure).

Code : PHP - /Applications/Backend/Modules/News/Views/insert.php

```
<h2>Ajouter une news</h2>
<?php require '_form.php';
```

Code : PHP - /Applications/Backend/Modules/News/Views/_form.php

```
<form action="" method="post">
  <p>
    <?php if (isset($erreurs) &&
in_array(\Library\Entities\News::AUTEUR_INVALIDE, $erreurs)) echo
'L\'auteur est invalide.<br />'; ?>
    <label>Auteur</label>
    <input type="text" name="auteur" value="<?php if (isset($news))
echo $news['auteur']; ?>" /><br />

    <?php if (isset($erreurs) &&
in_array(\Library\Entities\News::TITRE_INVALIDE, $erreurs)) echo 'Le
titre est invalide.<br />'; ?>
    <label>Titre</label><input type="text" name="titre" value="<?php
if (isset($news)) echo $news['titre']; ?>" /><br />

    <?php if (isset($erreurs) &&
in_array(\Library\Entities\News::CONTENU_INVALIDE, $erreurs)) echo
'Le contenu est invalide.<br />'; ?>
    <label>Contenu</label><textarea rows="8" cols="60"
name="contenu"><?php if (isset($news)) echo $news['contenu']; ?
></textarea><br />
<?php
if(isset($news) && !$news->isNew())
{
?>
    <input type="hidden" name="id" value="<?php echo $news['id']; ?
>" />
    <input type="submit" value="Modifier" name="modifier" />
<?php
}
else
{
?>
    <input type="submit" value="Ajouter" />
<?php
}
?>
  </p>
</form>
```

L'action *update*

La route

Pas d'originalité ici, nous allons choisir une URL basique : **/admin/news-update-*id*.html**.

Code : XML - /Applications/Backend/Config/routes.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/admin/" module="News" action="index" />
  <route url="/admin/news-insert\.html" module="News"
action="insert" />
  <route url="/admin/news-update-([0-9]+)\.html" module="News"
action="update" vars="id" />
</routes>
```

Le contrôleur

La méthode `executeUpdate()` est quasiment identique à `executeInsert()`. La seule différence est qu'il faut passer la news à la vue si le formulaire n'a pas été envoyé.

Code : PHP - /Applications/Backend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Backend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeUpdate(\Library\HTTPRequest $request)
  {
    if ($request->postExists('auteur'))
    {
      $this->processForm($request);
    }
    else
    {
      $this->page->addVar('news', $this->managers-
>getManagerOf('News')->getUnique($request->getData('id')));
    }

    $this->page->addVar('title', 'Modification d\'une news');
  }

  // ...
}
```

Le modèle

Ce code fait appel à deux méthodes : `getUnique()` et `modify()`. La première a déjà été implémentée au cours du précédent chapitre et la seconde avait volontairement été laissée de côté. Il est maintenant temps de l'implémenter.

Code : PHP - /Library/Models/NewsManager.class.php

```
<?php
```

```php
namespace Library\Models;

use \Library\Entities\News;

abstract class NewsManager extends \Library\Manager
{
  // ...

  /**
* Méthode permettant de modifier une news.
* @param $news news la news à modifier
* @return void
*/
  abstract protected function modify(News $news);

  // ...
}
```

Code : PHP - /Library/Models/NewsManager_PDO.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\News;

class NewsManager_PDO extends NewsManager
{
  // ...

  protected function modify(News $news)
  {
    $requete = $this->dao->prepare('UPDATE news SET auteur =
:auteur, titre = :titre, contenu = :contenu, dateModif = NOW() WHERE
id = :id');

    $requete->bindValue(':titre', $news->titre());
    $requete->bindValue(':auteur', $news->auteur());
    $requete->bindValue(':contenu', $news->contenu());
    $requete->bindValue(':id', $news->id(), \PDO::PARAM_INT);

    $requete->execute();
  }

  // ...
}
```

La vue

De la même façon que pour l'action *insert*, ce procédé tient en deux lignes : il s'agit seulement d'inclure le formulaire, c'est tout !

Code : PHP - /Applications/Backend/Modules/News/Views/update.php

```php
<h2>Modifier une news</h2>
<?php require '_form.php';
```

L'action *delete*

La route

Pour continuer dans l'originalité, l'URL qui pointera vers cette action sera du type **/admin/news-delete-*id*.html**.

Code : XML - /Applications/Backend/Config/routes.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/admin/" module="News" action="index" />
  <route url="/admin/news-insert\.html" module="News"
action="insert" />
  <route url="/admin/news-update-([0-9]+)\.html" module="News"
action="update" vars="id" />
  <route url="/admin/news-delete-([0-9]+)\.html" module="News"
action="delete" vars="id" />
</routes>
```

Le contrôleur

Le contrôleur se chargera d'invoquer la méthode du manager qui supprimera la news. Ensuite, il redirigera l'utilisateur à l'accueil de l'espace d'administration en ayant pris soin de spécifier un message qui s'affichera au prochain chargement de page. Ainsi, cette action ne possèdera aucune vue.

Code : PHP - /Applications/Backend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Backend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeDelete(\Library\HTTPRequest $request)
  {
    $this->managers->getManagerOf('News')->delete($request-
>getData('id'));

    $this->app->user()->setFlash('La news a bien été supprimée !');

    $this->app->httpResponse()->redirect('.');
  }

  // ...
}
```

Le modèle

Ici, une simple requête de type **DELETE** suffit.

Code : PHP - /Library/Models/NewsManager.class.php

```
<?php
namespace Library\Models;

use \Library\Entities\News;

abstract class NewsManager extends \Library\Manager
{
  // ...

  /**
* Méthode permettant de supprimer une news.
* @param $id int L'identifiant de la news à supprimer
* @return void
```

```
  */
    abstract public function delete($id);

    // ...
}
```

Code : PHP - /Library/Model/NewsManager_PDO.class.php

```
<?php
namespace Library\Models;

use \Library\Entities\News;

class NewsManager_PDO extends NewsManager
{
  // ...

  public function delete($id)
  {
    $this->dao->exec('DELETE FROM news WHERE id = '.(int) $id);
  }

  // ...
}
```

N'oublions pas les commentaires !

Finissons de construire notre application en implémentant les dernières fonctionnalités permettant de gérer les commentaires.

Fonctionnalités

Nous allons faire simple et implémenter deux fonctionnalités :

- La **modification** de commentaires.
- La **suppression** de commentaires.

L'action *updateComment*

La route

Commençons, comme d'habitude, par définir l'URL qui pointera vers ce module. Je vous propose quelque chose de simple comme **/admin/comments-update-*id*.html**.

Code : XML - /Applications/Backend/Config/routes.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/admin/" module="News" action="index" />
  <route url="/admin/news-insert\.html" module="News"
action="insert" />
  <route url="/admin/news-update-([0-9]+)\.html" module="News"
action="update" vars="id" />
  <route url="/admin/news-delete-([0-9]+)\.html" module="News"
action="delete" vars="id" />
<route url="/admin/comment-update-([0-9]+)\.html" module="News"
action="updateComment" vars="id" />
</routes>
```

Le contrôleur

La méthode que l'on implémentera aura pour rôle de contrôler les valeurs du formulaire et de modifier le commentaire en BDD si

tout est valide. Vous devriez avoir quelque chose de semblable à ce que nous avons dans l'application *Frontend*. Il faudra ensuite rediriger l'utilisateur sur la news qu'il lisait.

Aide : pour rediriger l'utilisateur sur la news, il va falloir obtenir l'identifiant de cette dernière. Il faudra donc ajouter un champ caché dans le formulaire pour transmettre ce paramètre.

Code : PHP - /Applications/Backend/Modules/News/NewsController.class.php

```php
<?php
namespace Applications\Backend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeUpdateComment(\Library\HTTPRequest
$request)
  {
    $this->page->addVar('title', 'Modification d\'un commentaire');

    if ($request->postExists('pseudo'))
    {
      $comment = new \Library\Entities\Comment(array(
        'id' => $request->getData('id'),
        'auteur' => $request->postData('pseudo'),
        'contenu' => $request->postData('contenu')
      ));

      if ($comment->isValid())
      {
        $this->managers->getManagerOf('Comments')->save($comment);

        $this->app->user()->setFlash('Le commentaire a bien été
modifié !');

        $this->app->httpResponse()->redirect('/news-'.$request-
>postData('news').'.html');
      }
      else
      {
        $this->page->addVar('erreurs', $comment->erreurs());
      }

      $this->page->addVar('comment', $comment);
    }
    else
    {
      $this->page->addVar('comment', $this->managers-
>getManagerOf('Comments')->get($request->getData('id')));
    }
  }

  // ...
}
```

Le modèle

Nous avons ici besoin d'implémenter deux méthodes : `modify()` et `get()`. La première se contente d'exécuter une requête de type `UPDATE` et la seconde une requête de type `SELECT`.

Code : PHP - /Library/Models/CommentsManager.class.php

```php
<?php
```

```php
namespace Library\Models;

use \Library\Entities\Comment;

abstract class CommentsManager extends \Library\Manager
{
  // ...

  /**
* Méthode permettant de modifier un commentaire.
* @param $comment Le commentaire à modifier
* @return void
*/
  abstract protected function modify(Comment $comment);

  /**
* Méthode permettant d'obtenir un commentaire spécifique.
* @param $id L'identifiant du commentaire
* @return Comment
*/
  abstract public function get($id);
}
```

Code : PHP - /Library/Models/CommentsManager_PDO.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\Comment;

class CommentsManager_PDO extends CommentsManager
{
  // ...

  protected function modify(Comment $comment)
  {
    $q = $this->dao->prepare('UPDATE comments SET auteur = :auteur,
contenu = :contenu WHERE id = :id');

    $q->bindValue(':auteur', $comment->auteur());
    $q->bindValue(':contenu', $comment->contenu());
    $q->bindValue(':id', $comment->id(), \PDO::PARAM_INT);

    $q->execute();
  }

  public function get($id)
  {
    $q = $this->dao->prepare('SELECT id, news, auteur, contenu FROM
comments WHERE id = :id');
    $q->bindValue(':id', (int) $id, \PDO::PARAM_INT);
    $q->execute();

    $q->setFetchMode(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE,
'\Library\Entities\Comment');

    return $q->fetch();
  }
}
```

La vue

La vue ne fera que contenir le formulaire et afficher les erreurs s'il y en a.

Code : PHP - /Applications/Backend/Modules/News/Views/updateComment.php

```
<form action="" method="post">
  <p>
    <?php if (isset($erreurs) &&
in_array(\Library\Entities\Comment::AUTEUR_INVALIDE, $erreurs)) echo
'L\'auteur est invalide.<br />'; ?>
    <label>Pseudo</label><input type="text" name="pseudo" value="<?
php echo htmlspecialchars($comment['auteur']); ?>" /><br />

    <?php if (isset($erreurs) &&
in_array(\Library\Entities\Comment::CONTENU_INVALIDE, $erreurs))
echo 'Le contenu est invalide.<br />'; ?>
    <label>Contenu</label><textarea name="contenu" rows="7"
cols="50"><?php echo htmlspecialchars($comment['contenu']); ?
></textarea><br />

    <input type="hidden" name="news" value="<?php echo
$comment['news']; ?>" />
    <input type="submit" value="Modifier" />
  </p>
</form>
```

Modification de la vue de l'affichage des commentaires

Pour des raisons pratiques, il serait préférable de modifier l'affichage des commentaires afin d'ajouter un lien à chacun menant vers la modification du commentaire. Pour cela, il faudra modifier la vue correspondant à l'action **show** du module **news** de l'application **Frontend**.

Code : PHP - /Applications/Frontend/Modules/News/Views/show.php

```
<p>Par <em><?php echo $news['auteur']; ?></em>, le <?php echo
$news['dateAjout']->format('d/m/Y à H\hi'); ?></p>
<h2><?php echo $news['titre']; ?></h2>
<p><?php echo nl2br($news['contenu']); ?></p>

<?php if ($news['dateAjout'] != $news['dateModif']) { ?>
  <p style="text-align: right;"><small><em>Modifiée le <?php echo
$news['dateModif']->format('d/m/Y à H\hi'); ?></em></small></p>
<?php } ?>

<p><a href="commenter-<?php echo $news['id']; ?>.html">Ajouter un
commentaire</a></p>

<?php
if (empty($comments))
{
?>
  <p>Aucun commentaire n'a encore été posté. Soyez le premier à en
laisser un !</p>
<?php
}

foreach ($comments as $comment)
{
?>
  <fieldset>
    <legend>
      Posté par <strong><?php echo
htmlspecialchars($comment['auteur']); ?></strong> le <?php echo
$comment['date']->format('d/m/Y à H\hi'); ?>
<?php if ($user->isAuthenticated()) { ?> -
<a href="admin/comment-update-<?php echo $comment['id']; ?
>.html">Modifier</a>
<?php } ?>
```

```
      </legend>
      <p><?php echo nl2br(htmlspecialchars($comment['contenu'])); ?
></p>
    </fieldset>
<?php
}
?>

<p><a href="commenter-<?php echo $news['id']; ?>.html">Ajouter un
commentaire</a></p>
```

L'action *deleteComment*

La route

Faisons là aussi très simple : nous n'avons qu'à prendre une URL du type **/admin/comments-delete-*id*.html**.

Code : XML - /Applications/Backend/Config/routes.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <route url="/admin/" module="News" action="index" />
  <route url="/admin/news-insert\.html" module="News"
action="insert" />
  <route url="/admin/news-update-([0-9]+)\.html" module="News"
action="update" vars="id" />
  <route url="/admin/news-delete-([0-9]+)\.html" module="News"
action="delete" vars="id" />
  <route url="/admin/comment-update-([0-9]+)\.html" module="News"
action="updateComment" vars="id" />
<route url="/admin/comment-delete-([0-9]+)\.html" module="News"
action="deleteComment" vars="id" />
</routes>
```

Le contrôleur

Il faut dans un premier temps invoquer la méthode du manager permettant de supprimer un commentaire. Redirigez ensuite l'utilisateur sur l'accueil de l'espace d'administration.

Code : PHP - /Applications/Backend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Backend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeDeleteComment(\Library\HTTPRequest
$request)
  {
    $this->managers->getManagerOf('Comments')->delete($request-
>getData('id'));

    $this->app->user()->setFlash('Le commentaire a bien été supprimé
!');

    $this->app->httpResponse()->redirect('.');
  }
}
```

Aucune vue n'est donc nécessaire ici.

Le modèle

Il suffit ici d'implémenter la méthode `delete()` exécutant une simple requête `DELETE`.

Code : PHP - /Library/Models/CommentsManager.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\Comment;

abstract class CommentsManager extends \Library\Manager
{
  // ...

  /**
* Méthode permettant de supprimer un commentaire.
* @param $id L'identifiant du commentaire à supprimer
* @return void
*/
  abstract public function delete($id);

  // ...
}
```

Code : PHP - /Library/Models/CommentsManager_PDO.class.php

```php
<?php
namespace Library\Models;

use \Library\Entities\Comment;

class CommentsManager_PDO extends CommentsManager
{
  // ...

  public function delete($id)
  {
    $this->dao->exec('DELETE FROM comments WHERE id = '.(int) $id);
  }

  // ...
}
```

Modification de l'affichage des commentaires

Nous allons là aussi insérer le lien de suppression de chaque commentaire afin de nous faciliter la tâche. Modifiez donc la vue de l'action **show** du module **news** de l'application **frontend** :

Code : PHP - /Applications/Frontend/Modules/News/Views/show.php

```php
<p>Par <em><?php echo $news['auteur']; ?></em>, le <?php echo
$news['dateAjout']->format('d/m/Y à H\hi'); ?></p>
<h2><?php echo $news['titre']; ?></h2>
<p><?php echo nl2br($news['contenu']); ?></p>

<?php if ($news['dateAjout'] != $news['dateModif']) { ?>
  <p style="text-align: right;"><small><em>Modifiée le <?php echo
```

```
        $news['dateModif']->format('d/m/Y à H\hi'); ?></em></small></p>
        <?php } ?>

        <p><a href="commenter-<?php echo $news['id']; ?>.html">Ajouter un
        commentaire</a></p>

        <?php
        if (empty($comments))
        {
        ?>
          <p>Aucun commentaire n'a encore été posté. Soyez le premier à en
        laisser un !</p>
        <?php
        }

        foreach ($comments as $comment)
        {
        ?>
          <fieldset>
            <legend>
              Posté par <strong><?php echo
        htmlspecialchars($comment['auteur']); ?></strong> le <?php echo
        $comment['date']->format('d/m/Y à H\hi'); ?>
              <?php if ($user->isAuthenticated()) { ?> -
                <a href="admin/comment-update-<?php echo $comment['id']; ?
        >.html">Modifier</a> |
        <a href="admin/comment-delete-<?php echo $comment['id']; ?
        >.html">Supprimer</a>
              <?php } ?>
            </legend>
            <p><?php echo nl2br(htmlspecialchars($comment['contenu'])); ?
        ></p>
          </fieldset>
        <?php
        }
        ?>

        <p><a href="commenter-<?php echo $news['id']; ?>.html">Ajouter un
        commentaire</a></p>
```

Gérer les formulaires

Il est possible que quelque chose vous chiffonne un petit peu. En effet, dans le *frontend*, nous avons créé un formulaire pour ajouter un commentaire. Dans le *backend*, nous avons recréé quasiment le même : nous avons fait de la **duplication de code**. Or, puisque vous êtes un excellent programmeur, cela devrait vous piquer les yeux !

Pour pallier ce problème courant de duplication de formulaires, nous allons **externaliser** nos formulaires à l'aide d'une API, c'est-à-dire que le code créant le formulaire sera accessible, à un autre endroit, par n'importe quel module de n'importe quelle application. Cette technique fera d'une pierre deux coups : non seulement nos formulaires seront décentralisés (donc réutilisables une infinité de fois), mais la création se fera de manière beaucoup plus aisée ! Bien sûr, comme pour la conception de l'application, cela deviendra rapide une fois l'API développée.

Le formulaire

Conception du formulaire

Commençons dans ce chapitre par créer un premier formulaire. Un formulaire, vous le savez, n'est autre qu'un ensemble de champs permettant d'interagir avec le contenu du site. Par exemple, voici notre formulaire d'ajout de commentaire :

Code : PHP

```
<form action="" method="post">
  <p>
    <?php if (isset($erreurs) &&
in_array(\Library\Entities\Comment::AUTEUR_INVALIDE, $erreurs)) echo
'L\'auteur est invalide.<br />'; ?>
    <label>Pseudo</label>
    <input type="text" name="pseudo" value="<?php if
(isset($comment)) echo htmlspecialchars($comment['auteur']); ?>"
/><br />

    <?php if (isset($erreurs) &&
in_array(\Library\Entities\Comment::CONTENU_INVALIDE, $erreurs))
echo 'Le contenu est invalide.<br />'; ?>
    <label>Contenu</label>
    <textarea name="contenu" rows="7" cols="50"><?php if
(isset($comment)) echo htmlspecialchars($comment['contenu']); ?
></textarea><br />

    <input type="submit" value="Commenter" />
  </p>
</form>
```

Cependant, vous conviendrez qu'il est long et fastidieux de créer ce formulaire. De plus, si nous voulons éditer un commentaire, il va falloir le dupliquer dans l'application *backend*. Dans un premier temps, nous allons nous occuper de l'aspect long et fastidieux : laissons un objet générer tous ces champs à notre place !

L'objet Form

Comme nous venons de le voir, **un formulaire n'est autre qu'une liste de champs**. Vous connaissez donc déjà le rôle de cet objet : il sera chargé de représenter le formulaire en possédant une liste de champs.

Commençons alors la liste des fonctionnalités de notre formulaire. Notre formulaire contient divers champs. Nous devons donc pouvoir **ajouter des champs** à notre formulaire. Ensuite, que serait un formulaire si on ne pouvait pas l'afficher ? Dans notre cas, le formulaire ne doit pas être capable de s'afficher mais de **générer** tous les champs qui lui sont attachés afin que le contrôleur puisse récupérer le corps du formulaire pour le passer à la vue. Enfin, notre formulaire doit posséder une dernière fonctionnalité : le capacité de déclarer si le formulaire est valide ou non en vérifiant que chaque champ l'est.

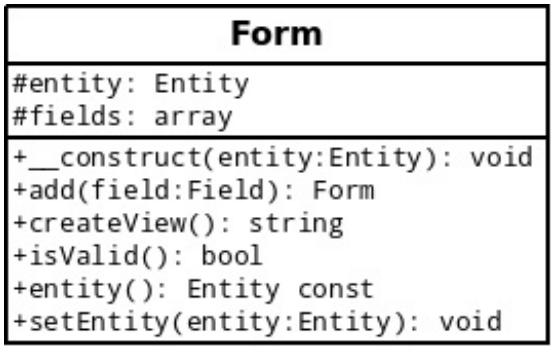
Pour résumer, nous avons donc trois fonctionnalités. Un objet `Form` doit être capable :

- D'ajouter des champs à sa liste de champs.
- De générer le corps du formulaire.
- De vérifier si tous les champs sont valides.

Ainsi, au niveau des caractéristiques de l'objet, nous en avons qui saute aux yeux : la **liste des champs** !

Cependant, un formulaire est également caractérisé par autre chose. En effet, si je vous demande de me dire comment vous allez vérifier si tous les champs sont valides, vous sauriez comment faire ? À aucun moment nous n'avons passé des valeurs à notre formulaire, donc aucune vérification n'est à effectuer. Il faudrait donc, dans le constructeur de notre objet `Form`, passer un objet contenant toutes ces valeurs. Ainsi, lors de l'ajout d'un champ, la méthode irait chercher la valeur correspondante dans cet objet et l'assignerait au champ (nous verrons plus tard comment la méthode sait à quel attribut de l'entité correspond le champ). À votre avis, à quoi vont ressembler ces objets ? En fait, vous les avez déjà créés ces objets : ce sont toutes les classes filles de `Entity` ! Par exemple, si vous voulez modifier un commentaire, vous allez créer un objet `Comment` que vous allez hydrater, puis vous créerez un objet `Form` en passant l'objet `Comment` au constructeur.

Ainsi, voici notre classe `Form` schématisée (voir la figure suivante).



Modélisation de la classe Form

Vous pouvez remarquer que la méthode `add()` renvoie un objet `Form`. En fait, il s'agit du formulaire auquel on a ajouté le champ : cela permet d'enchaîner facilement les appels à la méthode `add()` comme nous le verrons juste après.

L'objet Field

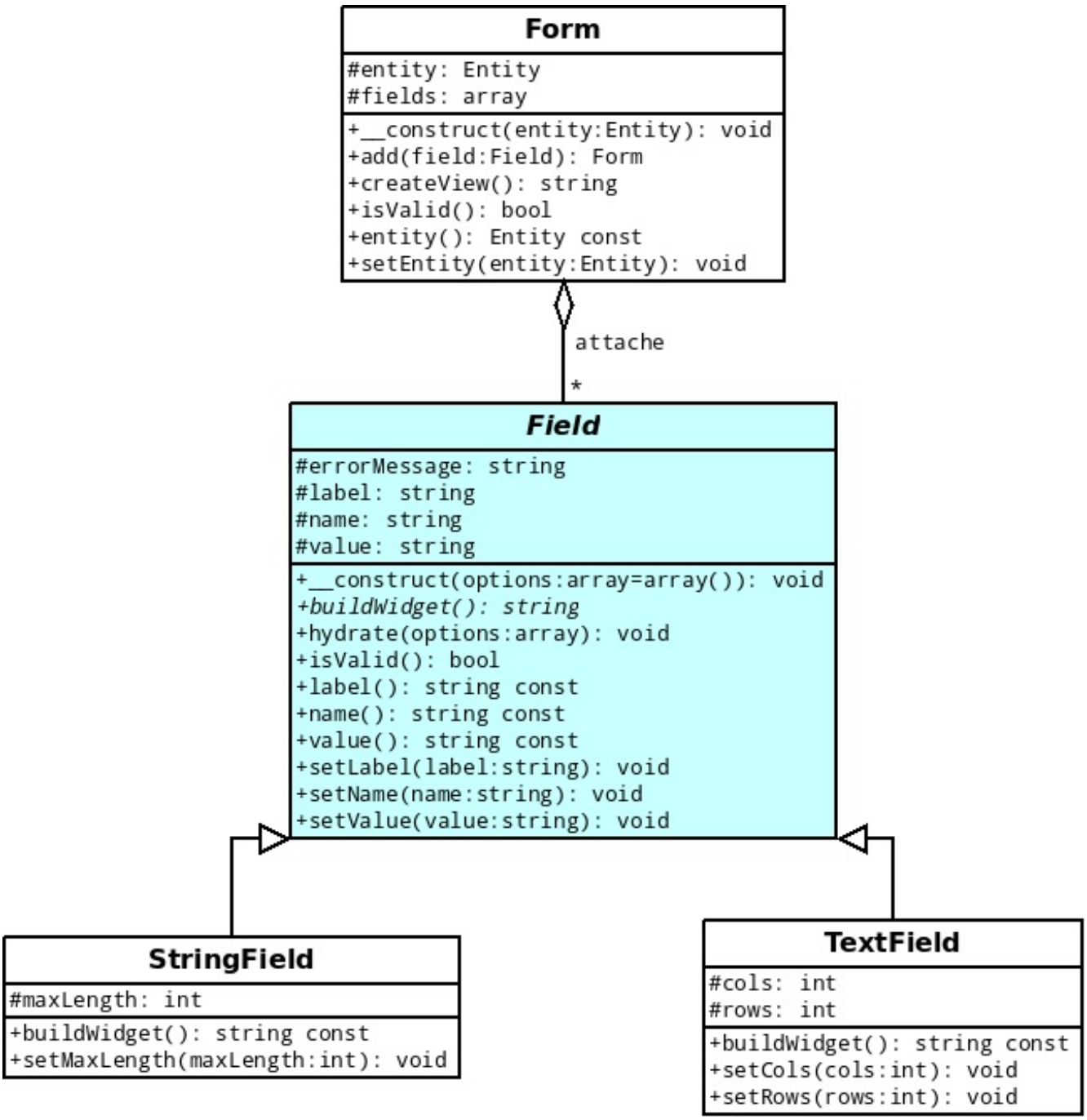
Puisque l'objet `Form` est intimement lié à ses champs, intéressons-nous à la conception de ces champs (ou *fields* en anglais). Vous l'aurez peut-être deviné : tous nos champs seront des objets, chacun représentant un champ différent (une classe représentera un champ texte, une autre classe représentera une zone de texte, etc.). À ce stade, un *tilt* devrait s'être produit dans votre tête : ce sont tous des champs, ils doivent donc hériter d'une même classe représentant leur nature en commun, à savoir une classe `Field` !

Commençons par cette classe `Field`. Quelles fonctionnalités attendons-nous de cette classe ? Un objet `Field` doit être capable :

- De renvoyer le code HTML représentant le champ.
- De vérifier si la valeur du champ est valide.

Je pense que vous aviez ces fonctionnalités plus ou moins en tête. Cependant, il y a encore une autre fonctionnalité que nous devons implémenter. En effet, pensez aux classes qui hériteront de `Field` et qui représenteront chacune un type de champ. Chaque champ a des attributs spécifiques. Par exemple, un champ texte (sur une ligne) possède un attribut `maxlength`, tandis qu'une zone de texte (un *textarea*) possède des attributs `rows` et `cols`. Chaque classe fille aura donc des attributs à elles seules. Il serait pratique, dès la construction de l'objet, de passer ces valeurs à notre champ (par exemple, assigner **50** à l'attribut `maxlength`). Pour résoudre ce genre de cas, nous allons procéder d'une façon qui ne vous est pas inconnue : nous allons créer une méthode permettant à l'objet de s'hydrater ! Ainsi, notre classe `Field` possédera une méthode `hydrate()`, comme les entités.

Voici à la figure suivante le schéma représentant notre classe `Field` liée à la classe `Form`, avec deux classes filles en exemple (`StringField` représentant un champ texte sur une ligne et la classe `TextField` représentant un *textarea*).



Modélisation de la classe Field

Développement de l'API

La classe `Form`

Pour rappel, voici de quoi la classe `Form` est composée :

- D'un attribut stockant la **liste des champs**.
- D'un attribut stockant l'**entité** correspondant au formulaire.
- D'un constructeur récupérant l'entité et invoquant le *setter* correspondant.
- D'une méthode permettant d'ajouter un champ à la liste des champs.
- D'une méthode permettant de générer le formulaire.
- D'une méthode permettant de vérifier si le formulaire est valide.

Voici la classe `Form` que vous auriez du obtenir :

Code : PHP - /Library/Form.class.php

```php
<?php
namespace Library;

class Form
{
  protected $entity;
  protected $fields;

  public function __construct(Entity $entity)
  {
    $this->setEntity($entity);
  }

  public function add(Field $field)
  {
    $attr = $field->name(); // On récupère le nom du champ.
    $field->setValue($this->entity->$attr()); // On assigne la
valeur correspondante au champ.

    $this->fields[] = $field; // On ajoute le champ passé en
argument à la liste des champs.
    return $this;
  }

  public function createView()
  {
    $view = '';

    // On génère un par un les champs du formulaire.
    foreach ($this->fields as $field)
    {
      $view .= $field->buildWidget().'<br />';
    }

    return $view;
  }

  public function isValid()
  {
    $valid = true;

    // On vérifie que tous les champs sont valides.
    foreach ($this->fields as $field)
    {
      if (!$field->isValid())
      {
        $valid = false;
      }
    }

    return $valid;
  }

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

  public function setEntity(Entity $entity)
  {
    $this->entity = $entity;
  }
}
```

La classe Field et ses filles

Voici un petit rappel sur la composition de la classe `Field`. Cette classe doit être composée :

- D'un attribut stockant le **message d'erreur** associé au champ.
- D'un attribut stockant le *label* du champ.
- D'un attribut stockant le **nom** du champ.
- D'un attribut stockant la **valeur** du champ.
- D'un constructeur demandant la liste des attributs avec leur valeur afin d'hydrater l'objet.
- D'une méthode (abstraite) chargée de renvoyer le code HTML du champ.
- D'une méthode permettant de savoir si le champ est valide ou non.

Les classes filles, quant à elles, n'implémenteront que la méthode abstraite. Si elles possèdent des attributs spécifiques (comme l'attribut `maxlength` pour la classe `StringField`), alors elles devront implémenter les mutateurs correspondant (comme vous le verrez plus tard, ce n'est pas nécessaire d'implémenter les accesseurs).

Voici les trois classes que vous auriez du obtenir (la classe `Field` avec deux classes filles en exemple, `StringField` et `TextField`) :

Code : PHP - /Library/Field.class.php

```php
<?php
namespace Library;

abstract class Field
{
  protected $errorMessage;
  protected $label;
  protected $name;
  protected $value;

  public function __construct(array $options = array())
  {
    if (!empty($options))
    {
      $this->hydrate($options);
    }
  }

  abstract public function buildWidget();

  public function hydrate($options)
  {
    foreach ($options as $type => $value)
    {
      $method = 'set'.ucfirst($type);

      if (is_callable(array($this, $method)))
      {
        $this->$method($value);
      }
    }
  }

  public function isValid()
  {
    // On écrira cette méthode plus tard.
  }

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

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

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

  public function setLabel($label)
  {
    if (is_string($label))
    {
      $this->label = $label;
    }
  }

  public function setName($name)
  {
    if (is_string($name))
    {
      $this->name = $name;
    }
  }

  public function setValue($value)
  {
    if (is_string($value))
    {
      $this->value = $value;
    }
  }
}
```

Code : PHP - /Library/StringField.class.php

```
<?php
namespace Library;

class StringField extends Field
{
  protected $maxLength;

  public function buildWidget()
  {
    $widget = '';

    if (!empty($this->errorMessage))
    {
      $widget .= $this->errorMessage.'<br />';
    }

    $widget .= '<label>'.$this->label.'</label><input type="text"
name="'.$this->name.'"';

    if (!empty($this->value))
    {
      $widget .= ' value="'.htmlspecialchars($this->value).'"';
    }

    if (!empty($this->maxLength))
    {
      $widget .= ' maxlength="'.$this->maxLength.'"';
    }

    return $widget .= ' />';
  }

  public function setMaxLength($maxLength)
  {
    $maxLength = (int) $maxLength;
```

```
      if ($maxLength > 0)
      {
        $this->maxLength = $maxLength;
      }
      else
      {
        throw new \RuntimeException('La longueur maximale doit être un
nombre supérieur à 0');
      }
    }
}
```

Code : PHP - /Library/TextField.class.php

```
<?php
namespace Library;

class TextField extends Field
{
  protected $cols;
  protected $rows;

  public function buildWidget()
  {
    $widget = '';

    if (!empty($this->errorMessage))
    {
      $widget .= $this->errorMessage.'<br />';
    }

    $widget .= '<label>'.$this->label.'</label><textarea
name="'.$this->name.'"';

    if (!empty($this->cols))
    {
      $widget .= ' cols="'.$this->cols.'"';
    }

    if (!empty($this->rows))
    {
      $widget .= ' rows="'.$this->rows.'"';
    }

    $widget .= '>';

    if (!empty($this->value))
    {
      $widget .= htmlspecialchars($this->value);
    }

    return $widget.'</textarea>';
  }

  public function setCols($cols)
  {
    $cols = (int) $cols;

    if ($cols > 0)
    {
      $this->cols = $cols;
    }
  }

  public function setRows($rows)
  {
    $rows = (int) $rows;
```

```
      if ($rows > 0)
      {
        $this->rows = $rows;
      }
    }
  }
```

Testons nos nouvelles classes

Testons dès maintenant nos classes. Dans notre contrôleur de news du *frontend*, nous allons modifier l'action chargée d'ajouter un commentaire. Créons notre formulaire avec nos nouvelles classes :

Code : PHP - /Applications/Frontend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Frontend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeInsertComment(\Library\HTTPRequest
$request)
  {
    // Si le formulaire a été envoyé, on crée le commentaire avec
les valeurs du formulaire.
    if ($request->method() == 'POST')
    {
      $comment = new \Library\Entities\Comment(array(
        'news' => $request->getData('news'),
        'auteur' => $request->postData('auteur'),
        'contenu' => $request->postData('contenu')
      ));
    }
    else
    {
      $comment = new \Library\Entities\Comment;
    }

    $form = new Form($comment);

    $form->add(new \Library\StringField(array(
        'label' => 'Auteur',
        'name' => 'auteur',
        'maxLength' => 50),
       )))
       ->add(new \Library\TextField(array(
        'label' => 'Contenu',
        'name' => 'contenu',
        'rows' => 7,
        'cols' => 50,
       )));

    if ($form->isValid())
    {
      // On enregistre le commentaire
    }

    $this->page->addVar('comment', $comment);
    $this->page->addVar('form', $form->createView()); // On passe le
formulaire généré à la vue.
    $this->page->addVar('title', 'Ajout d\'un commentaire');
  }

  // ...
}
```

Code : PHP - /Applications/Frontend/Modules/News/Views/insertComment.php

```
<h2>Ajouter un commentaire</h2>
<form action="" method="post">
  <p>
    <?php echo $form; ?>

    <input type="submit" value="Commenter" />
  </p>
</form>
```

Cependant, avouez que ce n'est pas pratique d'avoir ceci en plein milieu de notre contrôleur. De plus, si nous avons besoin de créer ce formulaire à un autre endroit, nous devrons copier/coller tous ces appels à la méthode `add()` et recréer tous les champs. Niveau duplication de code, nous sommes servi ! Nous résoudrons ce problème dans la suite du chapitre. Mais avant cela, intéressons-nous à la validation du formulaire. En effet, le contenu de la méthode `isValid()` est resté vide : faisons appel aux **validateurs** !

Les validateurs

Un validateur, comme son nom l'indique, est chargé de valider une donnée. Mais attention : un validateur ne peut valider **qu'une contrainte**. Par exemple, si vous voulez vérifier que votre valeur n'est pas nulle **et** qu'elle ne dépasse pas les cinquante caractères, alors vous aurez besoin de deux validateurs : le premier vérifiera que la valeur n'est pas nulle, et le second vérifiera que la chaine de caractères ne dépassera pas les cinquante caractères.

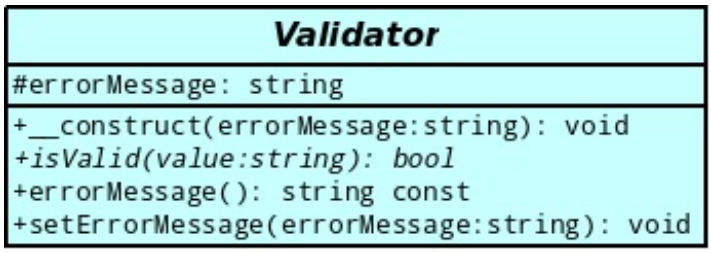
Là aussi, vous devriez savoir ce qui vous attend au niveau des classes : nous aurons une classe de base (`Validator`) et une infinité de classes filles (dans le cas précédent, on peut imaginer les classes `NotNullValidator` et `MaxLengthValidator`). Attaquons-les dès maintenant !

Conception des classes

La classe Validator

Notre classe de base, `Validator`, sera chargée, comme nous l'avons dit, de **valider** une donnée. Et c'est tout : un validateur ne sert à rien d'autre que valider une donnée. Au niveau des caractéristiques, il n'y en a dans ce cas également, une seule : le message d'erreur que le validateur doit pouvoir renvoyer si la valeur passée n'est pas valide.

Voici donc notre classe schématisée (voir la figure suivante).

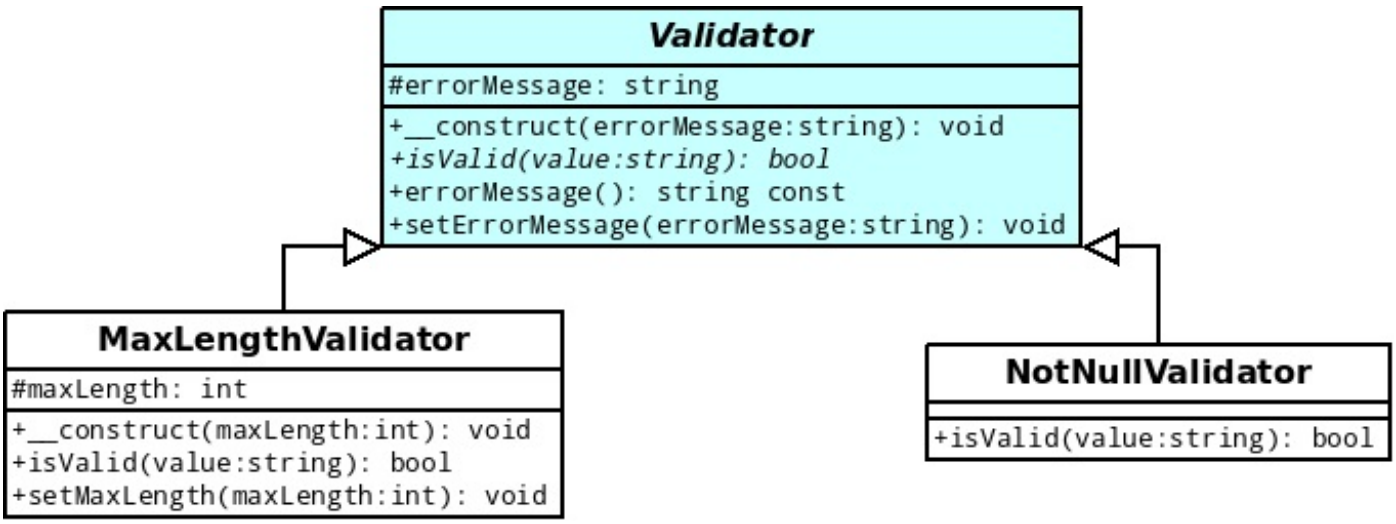


Modélisation de notre classe Validator

Les classes filles

Les classes filles sont elles aussi très simples. Commençons par la plus facile : `NotNullValidator`. Celle-ci, comme toute classe fille, sera chargée d'implémenter la méthode `isValid($value)`. Et c'est tout ! La seconde classe, `MaxLengthValidator`, implémente elle aussi cette méthode. Cependant, il faut qu'elle connaisse le nombre de caractères maximal que la chaîne doit avoir ! Pour cela, cette classe implémentera un constructeur demandant ce nombre en paramètre, et assignera cette valeur à l'attribut correspondant.

Ainsi, voici nos deux classes filles héritant de `Validator` (voir la figure suivante).



Modélisation des classes MaxLengthValidator et NotNullValidator

Développement des classes

La classe Validator

Cette classe (comme les classes filles) est assez simple à développer. En effet, il n'y a que l'accesseur et le mutateur du message d'erreur à implémenter, avec un constructeur demandant ledit message d'erreur. La méthode `isValid()`, quant à elle, est abstraite, donc rien à écrire de ce côté-là !

Code : PHP - /Library/Validator.class.php

```php
<?php
namespace Library;

abstract class Validator
{
  protected $errorMessage;

  public function __construct($errorMessage)
  {
    $this->setErrorMessage($errorMessage);
  }

  abstract public function isValid($value);

  public function setErrorMessage($errorMessage)
  {
    if (is_string($errorMessage))
    {
      $this->errorMessage = $errorMessage;
    }
  }

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

Les classes filles

Comme la précédente, les classes filles sont très simples à concevoir. J'espère que vous y êtes parvenus !

Code : PHP - /Library/NotNullValidator.class.php

```php
<?php
namespace Library;

class NotNullValidator extends Validator
{
  public function isValid($value)
  {
    return $value != '';
  }
}
```

Code : PHP - /Library/MaxLengthValidator.class.php

```php
<?php
namespace Library;

class MaxLengthValidator extends Validator
{
  protected $maxLength;

  public function __construct($errorMessage, $maxLength)
  {
    parent::__construct($errorMessage);

    $this->setMaxLength($maxLength);
  }

  public function isValid($value)
  {
    return strlen($value) <= $this->maxLength;
  }

  public function setMaxLength($maxLength)
  {
    $maxLength = (int) $maxLength;

    if ($maxLength > 0)
    {
      $this->maxLength = $maxLength;
    }
    else
    {
      throw new \RuntimeException('La longueur maximale doit être un
nombre supérieur à 0');
    }
  }
}
```

Modification de la classe Field

Comme nous l'avions vu, pour savoir si un champ est valide, il lui faut des **validateurs**. Il va donc falloir passer, dans le constructeur de l'objet `Field` créé, la liste des validateurs que l'on veut imposer au champ. Dans le cas du champ **auteur** par exemple, nous lui passerons les deux validateurs : nous voulons à la fois que le champ ne soit pas vide et que la valeur ne dépasse pas les cinquante caractères. La création du formulaire ressemblerait donc à ceci :

Code : PHP

```php
<?php
// $form représente le formulaire que l'on souhaite créer.
// Ici, on souhaite lui ajouter le champ « auteur ».
$form->add(new \Library\StringField(array(
  'label' => 'Auteur',
  'name' => 'auteur',
```

```
    'maxLength' => 50,
    'validators' => array(
      new \Library\MaxLengthValidator('L\'auteur spécifié est trop
long (50 caractères maximum)', 50),
      new \Library\NotNullValidator('Merci de spécifier l\'auteur du
commentaire'),
    )
)));
```

De cette façon, quelques modifications au niveau de notre classe `Field` s'imposent. En effet, il va falloir créer un attribut `$validators`, ainsi que l'accesseur et le mutateur correspondant. De la sorte, notre méthode `hydrate()` assignera automatiquement les validateurs passés au constructeur à l'attribut `$validators`. Je vous laisse faire cela.

Vient maintenant l'implémentation de la méthode `isValid()`. Cette méthode doit parcourir tous les validateurs et invoquer la méthode `isValid($value)` sur ces validateurs afin de voir si la valeur passe au travers du filet de **tous** les validateurs. De cette façon, nous sommes sûrs que toutes les contraintes ont été respectées ! Si un validateur renvoie une réponse négative lorsqu'on lui demande si la valeur est valide, alors on devra lui demander le message d'erreur qui lui a été assigné et l'assigner à notre tour à l'attribut correspondant. Ainsi, voici la nouvelle classe `Field` :

Code : PHP - /Library/Field.class.php

```
<?php
namespace Library;

abstract class Field
{
  protected $errorMessage;
  protected $label;
  protected $name;
  protected $validators = array();
  protected $value;

  public function __construct(array $options = array())
  {
    if (!empty($options))
    {
      $this->hydrate($options);
    }
  }

  abstract public function buildWidget();

  public function hydrate($options)
  {
    foreach ($options as $type => $value)
    {
      $method = 'set'.ucfirst($type);

      if (is_callable(array($this, $method)))
      {
        $this->$method($value);
      }
    }
  }

  public function isValid()
  {
    foreach ($this->validators as $validator)
    {
      if (!$validator->isValid($this->value))
      {
        $this->errorMessage = $validator->errorMessage();
        return false;
      }
    }
```

```
        return true;
      }

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

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

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

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

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

      public function setLabel($label)
      {
        if (is_string($label))
        {
          $this->label = $label;
        }
      }

      public function setLength($length)
      {
        $length = (int) $length;

        if ($length > 0)
        {
          $this->length = $length;
        }
      }

      public function setName($name)
      {
        if (is_string($name))
        {
          $this->name = $name;
        }
      }

      public function setValidators(array $validators)
      {
        foreach ($validators as $validator)
        {
          if ($validator instanceof Validator && !in_array($validator,
$this->validators))
          {
            $this->validators[] = $validator;
          }
        }
      }

      public function setValue($value)
      {
        if (is_string($value))
        {
```

```
            $this->value = $value;
        }
    }
}
```

Vous pouvez apercevoir l'utilisation de l'opérateur `instanceof` dans le code. Pour en savoir plus à ce sujet, je vous invite à aller lire le chapitre dédié à cet opérateur.

Le constructeur de formulaires

Comme nous l'avons vu, créer le formulaire au sein du contrôleur présente deux inconvénients. Premièrement, cela encombre le contrôleur. Imaginez que vous ayez une dizaine de champs, cela deviendrait énorme ! Le contrôleur doit être clair, et la création du formulaire devrait donc se faire autre part. Deuxièmement, il y a le problème de duplication de code : si vous voulez utiliser ce formulaire dans un autre contrôleur, vous devrez copier/coller tout le code responsable de la création du formulaire. Pas très flexible vous en conviendrez ! Pour cela, nous allons donc créer des **constructeurs de formulaire**. Il y aura par conséquent autant de constructeurs que de formulaires différents.

Conception des classes

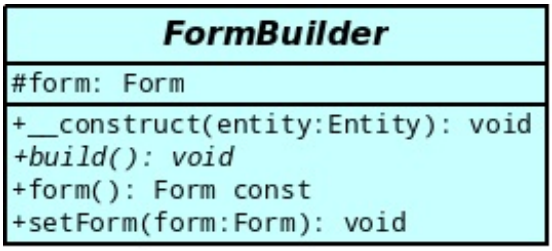
La classe FormBuilder

La classe `FormBuilder` a un rôle bien précis : elle est chargée de **construire un formulaire**. Ainsi, il n'y a qu'une seule fonctionnalité à implémenter... celle de construire le formulaire ! Mais, pour ce faire, encore faudrait-il avoir un objet `Form`. Nous le créerons donc dans le constructeur et nous l'assignerons à l'attribut correspondant.

Nous avons donc :

- Une méthode abstraite chargée de construire le formulaire.
- Un attribut stockant le formulaire.
- L'accesseur et le mutateur correspondant.

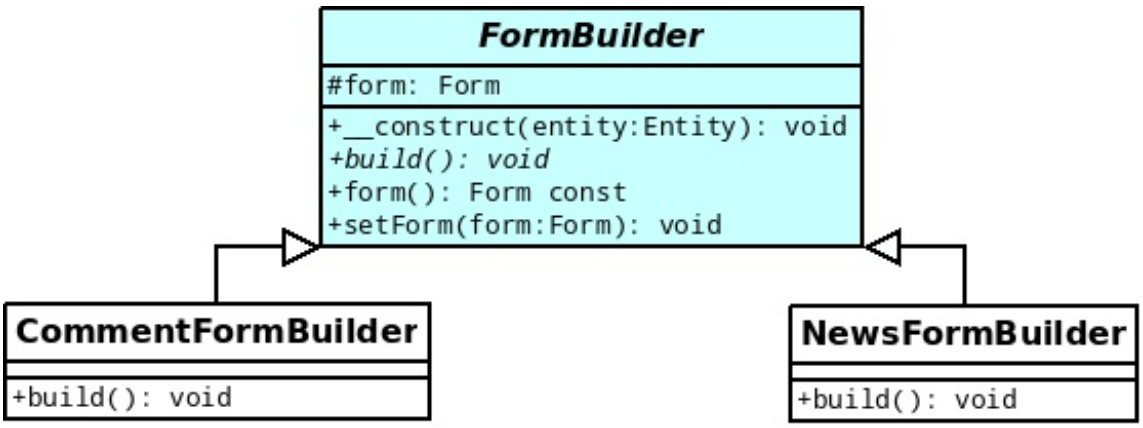
Voici notre classe schématisée (voir la figure suivante).



Modélisation de la classe FormBuilder

Les classes filles

Un constructeur de base c'est bien beau, mais sans classe fille, difficile de construire grand-chose. Je vous propose donc de créer deux constructeurs de formulaire : un constructeur de formulaire de commentaires, et un constructeur de formulaire de news. Nous aurons donc notre classe `FormBuilder` dont hériteront deux classes, `CommentFormBuilder` et `NewsFormBuilder` (voir la figure suivante).



Modélisation des classes CommentFormBuilder et NewsFormBuilder

Développement des classes

La classe FormBuilder

Cette classe est assez simple à créer, j'espère que vous y êtes parvenus !

Code : PHP - /Library/FormBuilder.class.php

```php
<?php
namespace Library;

abstract class FormBuilder
{
  protected $form;

  public function __construct(Entity $entity)
  {
    $this->setForm(new Form($entity));
  }

  abstract public function build();

  public function setForm(Form $form)
  {
    $this->form = $form;
  }

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

Les classes filles

Les classes filles sont simples à créer. En effet, il n'y a que la méthode `build()` à implémenter, en ayant pour simple contenu d'appeler successivement les méthodes `add()` sur notre formulaire. Pour l'emplacement des fichiers stockant les classes, je vous propose de les placer dans le dossier **/Library/FormBuilder**.

Code : PHP - /Library/FormBuilder/CommentFormBuilder.class.php

```php
<?php
namespace Library\FormBuilder;

class CommentFormBuilder extends \Library\FormBuilder
{
```

```
  public function build()
  {
    $this->form->add(new \Library\StringField(array(
        'label' => 'Auteur',
        'name' => 'auteur',
        'maxLength' => 50,
        'validators' => array(
          new \Library\MaxLengthValidator('L\'auteur spécifié est
trop long (50 caractères maximum)', 50),
          new \Library\NotNullValidator('Merci de spécifier
l\'auteur du commentaire'),
        ),
       )))
       ->add(new \Library\TextField(array(
        'label' => 'Contenu',
        'name' => 'contenu',
        'rows' => 7,
        'cols' => 50,
        'validators' => array(
          new \Library\NotNullValidator('Merci de spécifier votre
commentaire'),
        ),
       )));
  }
}
```

Code : PHP - /Library/NewsFormBuilder.class.php

```
<?php
namespace Library\FormBuilder;

class NewsFormBuilder extends \Library\FormBuilder
{
  public function build()
  {
    $this->form->add(new \Library\StringField(array(
        'label' => 'Auteur',
        'name' => 'auteur',
        'maxLength' => 20,
        'validators' => array(
          new \Library\MaxLengthValidator('L\'auteur spécifié est
trop long (20 caractères maximum)', 20),
          new \Library\NotNullValidator('Merci de spécifier
l\'auteur de la news'),
        ),
       )))
       ->add(new \Library\StringField(array(
        'label' => 'Titre',
        'name' => 'titre',
        'maxLength' => 100,
        'validators' => array(
          new \Library\MaxLengthValidator('Le titre spécifié est
trop long (100 caractères maximum)', 100),
          new \Library\NotNullValidator('Merci de spécifier le titre
de la news'),
        ),
       )))
       ->add(new \Library\TextField(array(
        'label' => 'Contenu',
        'name' => 'contenu',
        'rows' => 8,
        'cols' => 60,
        'validators' => array(
          new \Library\NotNullValidator('Merci de spécifier le
contenu de la news'),
        ),
       )));
  }
```

```
  }
```

Modification des contrôleurs

L'ajout de commentaire (frontend)

Effectuons des premières modifications, en commençant par le formulaire d'ajout de commentaire dans le *frontend*. En utilisant nos classes, voici les instructions que nous devons exécuter :

- Si la requête est de type POST (formulaire soumis), il faut créer un nouveau commentaire en le remplissant avec les données envoyées, sinon on crée un nouveau commentaire.
- On instancie notre constructeur de formulaire en lui passant le commentaire en argument.
- On invoque la méthode de construction du formulaire.
- Si le formulaire est valide, on enregistre le commentaire en BDD.
- On passe le formulaire généré à la vue.

Voilà ce que ça donne :

Code : PHP - /Applications/Frontend/Modules/News/NewsController.class.php

```php
<?php
namespace Applications\Frontend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeInsertComment(\Library\HTTPRequest
$request)
  {
    // Si le formulaire a été envoyé.
    if ($request->method() == 'POST')
    {
      $comment = new \Library\Entities\Comment(array(
        'news' => $request->getData('news'),
        'auteur' => $request->postData('auteur'),
        'contenu' => $request->postData('contenu')
      ));
    }
    else
    {
      $comment = new \Library\Entities\Comment;
    }

    $formBuilder = new
\Library\FormBuilder\CommentFormBuilder($comment);
    $formBuilder->build();

    $form = $formBuilder->form();

    if($request->method() == 'POST' && $form->isValid())
    {
      $this->managers->getManagerOf('Comments')->save($comment);
      $this->app->user()->setFlash('Le commentaire a bien été
ajouté, merci !');
      $this->app->httpResponse()->redirect('news-'.$request-
>getData('news').'.html');
    }

    $this->page->addVar('comment', $comment);
    $this->page->addVar('form', $form->createView());
    $this->page->addVar('title', 'Ajout d\'un commentaire');
  }
```

```
    // ...
  }
```

La modification de commentaire, l'ajout et la modification de news (backend)

Normalement, vous devriez être capables, grâce à l'exemple précédent, de parvenir à créer ces trois autres formulaires. Voici le nouveau contrôleur :

Code : PHP - /Applications/Backend/Modules/News/NewsController.class.php

```
<?php
namespace Applications\Backend\Modules\News;

class NewsController extends \Library\BackController
{
  // ...

  public function executeInsert(\Library\HTTPRequest $request)
  {
    $this->processForm($request);

    $this->page->addVar('title', 'Ajout d\'une news');
  }

  public function executeUpdate(\Library\HTTPRequest $request)
  {
    $this->processForm($request);

    $this->page->addVar('title', 'Modification d\'une news');
  }

  public function executeUpdateComment(\Library\HTTPRequest
$request)
  {
    $this->page->addVar('title', 'Modification d\'un commentaire');

    if ($request->method() == 'POST')
    {
      $comment = new \Library\Entities\Comment(array(
        'id' => $request->getData('id'),
        'auteur' => $request->postData('auteur'),
        'contenu' => $request->postData('contenu')
      ));
    }
    else
    {
      $comment = $this->managers->getManagerOf('Comments')-
>get($request->getData('id'));
    }

    $formBuilder = new
\Library\FormBuilder\CommentFormBuilder($comment);
    $formBuilder->build();

    $form = $formBuilder->form();

    if ($request->method() == 'POST' && $form->isValid())
    {
      $this->managers->getManagerOf('Comments')->save($comment);
      $this->app->user()->setFlash('Le commentaire a bien été
modifié');
      $this->app->httpResponse()->redirect('/admin/');
    }

    $this->page->addVar('form', $form->createView());
  }
```

```
  public function processForm(\Library\HTTPRequest $request)
  {
    if ($request->method() == 'POST')
    {
      $news = new \Library\Entities\News(
        array(
          'auteur' => $request->postData('auteur'),
          'titre' => $request->postData('titre'),
          'contenu' => $request->postData('contenu')
        )
      );

      if ($request->getExists('id'))
      {
        $news->setId($request->getData('id'));
      }
    }
    else
    {
      // L'identifiant de la news est transmis si on veut la
modifier.
      if ($request->getExists('id'))
      {
        $news = $this->managers->getManagerOf('News')-
>getUnique($request->getData('id'));
      }
      else
      {
        $news = new \Library\Entities\News;
      }
    }

    $formBuilder = new \Library\FormBuilder\NewsFormBuilder($news);
    $formBuilder->build();

    $form = $formBuilder->form();

    if ($request->method() == 'POST' && $form->isValid())
    {
      $this->managers->getManagerOf('News')->save($news);
      $this->app->user()->setFlash($news->isNew() ? 'La news a bien
été ajoutée !' : 'La news a bien été modifiée !');
      $this->app->httpResponse()->redirect('/admin/');
    }

    $this->page->addVar('form', $form->createView());
  }
}
```

Le gestionnaire de formulaires

Terminons ce chapitre en améliorant encore notre API permettant la création de formulaire. Je voudrais attirer votre attention sur ce petit passage, que l'on retrouve à chaque fois (que ce soit pour ajouter ou modifier une news ou un commentaire) :

Code : PHP

```
<?php
// Nous sommes ici au sein d'un contrôleur
if($request->method() == 'POST' && $form->isValid())
{
  $this->managers->getManagerOf('Manager')->save($comment);
  // ...
}
```

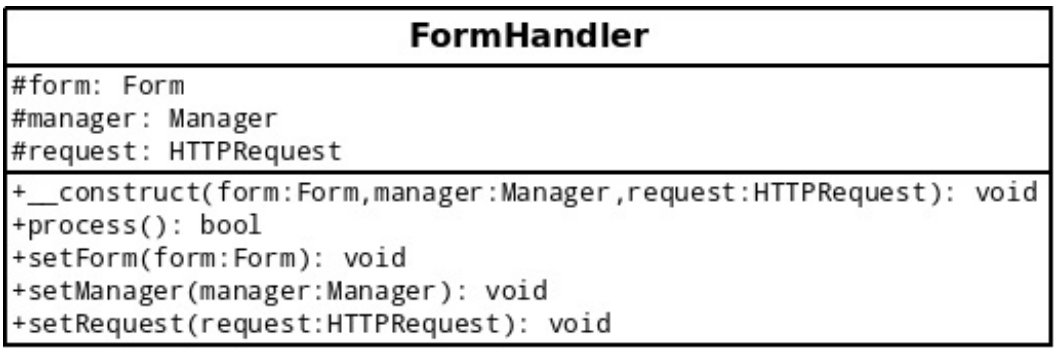
Bien que réduit, ce bout de code est lui aussi dupliqué. De plus, si l'on veut vraiment externaliser la gestion du formulaire, alors il

va falloir le sortir du contrôleur. Ainsi, il ne restera plus d'opération de traitement dans le contrôleur. On séparera donc bien les rôles : le contrôleur n'aura plus à réfléchir sur le formulaire qu'il traite. En effet, il ne fera que demander au constructeur de formulaire de construire le formulaire qu'il veut, puis demandera au gestionnaire de formulaire de s'occuper de lui s'il a été envoyé. On ne se souciera donc plus de l'aspect interne du formulaire !

Conception du gestionnaire de formulaire

Comme nous venons de le voir, le gestionnaire de formulaire est chargé de traiter le formulaire une fois qu'il a été envoyé. Nous avons donc d'ores et déjà une fonctionnalité de notre classe : celle de traiter le formulaire. Du côté des caractéristiques, penchons-nous du côté des éléments dont notre gestionnaire a besoin pour fonctionner. Le premier élément me paraît évident : comment s'occuper d'un formulaire si on n'y a pas accès ? Ce premier élément est donc bien entendu le formulaire dont il est question. Le deuxième élément, lui, est aussi évident : comment enregistrer l'entité correspondant au formulaire si on n'a pas le *manager* correspondant ? Le deuxième élément est donc le *manager* correspondant à l'entité. Enfin, le troisième élément est un peu plus subtil, et il faut réfléchir au contenu de la méthode qui va traiter le formulaire. Cette méthode devra savoir si le formulaire a été envoyé pour pouvoir le traiter (si rien n'a été envoyé, il n'y a aucune raison de traiter quoi que ce soit). Ainsi, pour savoir si le formulaire a été envoyé, il faut que notre gestionnaire de formulaire ait accès à la requête du client afin de connaitre le type de la requête (GET ou POST). Ces trois éléments devront être passés au constructeur de notre objet.

Schématiquement, voici notre gestionnaire de formulaire (voir la figure suivante).



Modélisation de la classe FormHandler

Développement du gestionnaire de formulaire

Voici le résultat que vous auriez du obtenir :

Code : PHP - /Library/FormHandler.class.php

```php
<?php
namespace Library;

class FormHandler
{
  protected $form;
  protected $manager;
  protected $request;

  public function __construct(\Library\Form $form, \Library\Manager $manager, \Library\HTTPRequest $request)
  {
    $this->setForm($form);
    $this->setManager($manager);
    $this->setRequest($request);
  }

  public function process()
  {
    if($this->request->method() == 'POST' && $this->form->isValid())
    {
      $this->manager->save($this->form->entity());

      return true;
    }
```

```
      return false;
    }

    public function setForm(\Library\Form $form)
    {
      $this->form = $form;
    }

    public function setManager(\Library\Manager $manager)
    {
      $this->manager = $manager;
    }

    public function setRequest(\Library\HTTPRequest $request)
    {
      $this->request = $request;
    }
}
```

Modification des contrôleurs

Ici, la modification est très simple. En effet, nous avons juste décentralisé ce bout de code :

Code : PHP

```
<?php
if($request->method() == 'POST' && $form->isValid())
{
  $this->managers->getManagerOf('Manager')->save($comment);
  // Autres opérations (affichage d'un message informatif,
redirection, etc.).
}
```

Il suffit donc de remplacer ce code par la simple invocation de la méthode `process()` sur notre objet `FormHandler` :

Code : PHP

```
<?php
// On récupère le gestionnaire de formulaire (le paramètre de
getManagerOf() est bien entendu à remplacer).
$formHandler = new \Library\FormHandler($form, $this->managers-
>getManagerOf('Comments'), $request);

if ($formHandler->process())
{
  // Ici ne résident plus que les opérations à effectuer une fois
l'entité du formulaire enregistrée
  // (affichage d'un message informatif, redirection, etc.).
}
```

Je vous fais confiance pour mettre à jour vos contrôleurs comme il se doit !

Partie 4 : Annexes

Voici ici quelques pages vous présentant certaines notions que je n'ai pas pu glisser dans le cours au risque de compliquer celui-ci inutilement. Au début de chaque chapitre seront précisés les pré-requis afin de pouvoir suivre sans difficulté.

L'opérateur instanceof

Vous êtes-vous déjà demandé s'il était possible de savoir si un objet était une instance de telle classe ? Si vous vous êtes déjà posé cette question, vous n'avez normalement pas trouvé de réponse vraiment claire. Un moyen simple de vérifier une telle chose est d'utiliser l'opérateur `instanceof`. Ce sera un court chapitre car cette notion n'est pas bien difficile. Il faut juste posséder quelques pré-réquis.

En voici la liste :

- Bien maîtriser les notions de **classe**, d'**objet** et d'**instance** .
- Bien maîtriser le concept de l'**héritage** (si vous ne maîtrisez pas bien la résolution statique à la volée ce n'est pas bien important).
- Savoir ce qu'est une **interface** et savoir s'en servir.

Présentation de l'opérateur

L'opérateur `instanceof` permet de vérifier si tel objet est une *instance de* telle classe. C'est un opérateur qui s'utilise dans une condition. Ainsi, on pourra créer des conditions comme *« si* `$monObjet` *est une instance de* `MaClasse`*, alors... »*.

Maintenant voyons comment construire notre condition. À gauche de notre opérateur,nous allons y placer notre objet. À droite de notre opérateur, nous allons placer, comme vous vous en doutez sûrement, le nom de la classe.

Exemple :

Code : PHP

```php
<?php
class A { }
class B { }

$monObjet = new A;

if ($monObjet instanceof A) // Si $monObjet est une instance de A.
{
  echo '$monObjet est une instance de A';
}
else
{
  echo '$monObjet n\'est pas une instance de A';
}

if ($monObjet instanceof B) // Si $monObjet est une instance de B.
{
  echo '$monObjet est une instance de B';
}
else
{
  echo '$monObjet n\'est pas une instance de B';
}
?>
```

Bref, je pense que vous avez compris le principe.

Si votre version de PHP est ultérieure à la version 5.1, alors aucune erreur fatale ne sera générée si vous utilisez l'opérateur `instanceof` en spécifiant une classe qui n'a pas été déclarée. La condition renverra tout simplement `false`.

Il y a cependant plusieurs façons de procéder et quelques astuces (c'est d'ailleurs pour toutes les présenter que j'ai créé ce chapitre).

Parmi ces méthodes, il y en a une qui consiste à placer le nom de la classe pour laquelle on veut vérifier que tel objet est une instance dans une variable sous forme de chaîne de caractères. Exemple :

Code : PHP

```php
<?php
class A { }
class B { }

$monObjet = new A;

$classeA = 'A';
$classeB = 'B';

if ($monObjet instanceof $classeA)
{
  echo '$monObjet est une instance de ', $classeA;
}
else
{
  echo '$monObjet n\'est pas une instance de ', $classeA;
}

if ($monObjet instanceof $classeB)
{
  echo '$monObjet est une instance de ', $classeB;
}
else
{
  echo '$monObjet n\'est pas une instance de ', $classeB;
}
?>
```

Attention ! Vous ne pouvez spécifier le nom de la classe entre apostrophes ou guillemets directement dans la condition ! Vous devez obligatoirement passer par une variable. Si vous le faites directement, vous obtiendrez une belle erreur d'analyse.

Une autre façon d'utiliser cet opérateur est de spécifier un autre objet à la place du nom de la classe. La condition renverra `true` si les deux objets sont des instances de la même classe. Exemple :

Code : PHP

```php
<?php
class A { }
class B { }

$a = new A;
$b = new A;
$c = new B;

if ($a instanceof $b)
{
  echo '$a et $b sont des instances de la même classe';
}
else
{
  echo '$a et $b ne sont pas des instances de la même classe';
}
```

```php
if ($a instanceof $c)
{
  echo '$a et $c sont des instances de la même classe';
}
else
{
  echo '$a et $c ne sont pas des instances de la même classe';
}
?>
```

Et voilà. Vous connaissez les trois méthodes possibles pour utiliser cet opérateur. Pourtant, il existe encore quelques effets que peut produire `instanceof`. Poursuivons donc ce chapitre tranquillement.

instanceof et l'héritage

L'héritage est de retour ! En effet, `instanceof` a un comportement bien particulier avec les classes qui héritent entre elles. Voici ces effets.

Vous vous souvenez sans doute (enfin j'espère) de la première façon d'utiliser l'opérateur. Voici une révélation : la condition renvoie `true` si la classe spécifiée est une classe **parente** de la classe instanciée par l'objet spécifié. Exemple :

Code : PHP

```php
<?php
class A { }
class B extends A { }
class C extends B { }

$b = new B;

if ($b instanceof A)
{
  echo '$b est une instance de A ou $b instancie une classe qui est
une fille de A';
}
else
{
  echo '$b n\'est pas une instance de A et $b instancie une classe
qui n\'est pas une fille de A';
}

if ($b instanceof C)
{
  echo '$b est une instance de C ou $b instancie une classe qui est
une fille de C';
}
else
{
  echo '$b n\'est pas une instance de C et $b instancie une classe
qui n\'est pas une fille de C';
}
?>
```

Voilà, j'espère que vous avez compris le principe car celui-ci est le même avec les deuxième et troisième méthodes.

Nous allons donc maintenant terminer ce chapitre avec une dernière partie concernant les réactions de l'opérateur avec les interfaces. Ce sera un mélange des deux premières parties, donc si vous êtes perdus, relisez bien tout (eh oui, j'espère que vous n'avez pas oublié l'héritage entre interfaces).

instanceof et les interfaces

Voyons maintenant les effets produits par l'opérateur avec les interfaces.

Hein ? Comment ça ? Je comprends pas... Comment peut-on vérifier qu'un objet soit une instance d'une interface sachant que c'est impossible ?

Comme vous le dites si bien, il est impossible de créer une instance d'une interface (au même titre que de créer une instance d'une classe abstraite, ce qu'est à peu près une interface). L'opérateur va donc renvoyer `true` si tel objet instancie une classe implémentant telle interface.

Voici un exemple :

Code : PHP

```php
<?php
interface iA { }
class A implements iA { }
class B { }

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

if ($a instanceof iA)
{
  echo 'Si iA est une classe, alors $a est une instance de iA ou $a instancie une classe qui est une fille de iA. Sinon, $a instancie une classe qui implémente iA.';
}
else
{
  echo 'Si iA est une classe, alors $a n\'est pas une instance de iA et $a n\'instancie aucune classe qui est une fille de iA. Sinon, $a instancie une classe qui n\'implémente pas iA.';
}

if ($b instanceof iA)
{
  echo 'Si iA est une classe, alors $b est une instance de iA ou $b instancie une classe qui est une fille de iA. Sinon, $b instancie une classe qui implémente iA.';
}
else
{
  echo 'Si iA est une classe, alors $b n\'est pas une instance de iA et $b n\'instancie aucune classe qui est une fille de iA. Sinon, $b instancie une classe qui n\'implémente pas iA.';
}
?>
```

Ce code se passe de commentaires, les valeurs affichées détaillant assez bien je pense.

Après avoir vu l'utilisation de l'opérateur avec les interfaces, nous allons voir comment il réagit lorsqu'on lui passe en paramètre une interface qui est héritée par une autre interface qui est implémentée par une classe qui est instanciée. Vous voyez à peu près à quoi je fais sréférence ? Je vais procéder en PHP au cas où vous n'ayez pas tout suivi.

Code : PHP

```php
<?php
interface iParent { }
interface iFille extends iParent { }
class A implements iFille { }

$a = new A;

if ($a instanceof iParent)
{
```

```
  echo 'Si iParent est une classe, alors $a est une instance de
iParent ou $a instancie une classe qui est une fille de iParent.
Sinon, $a instancie une classe qui implémente iParent ou une fille
de iParent.';
}
else
{
  echo 'Si iParent est une classe, alors $a n\'est pas une instance
de iParent et $a n\'instancie aucune classe qui est une fille de
iParent. Sinon, $a instancie une classe qui n\'implémente ni
iParent, ni une de ses filles.';
}
?>
```

Vous savez maintenant tous les comportements que peut adopter cet opérateur et tous les effets qu'il peut produire (tout est écrit dans le précédent code).

En résumé

- L'opérateur `instanceof` permet de vérifier la nature de la classe dont l'objet testé est une instance.
- Cet opérateur permet de vérifier qu'un certain objet est bien une instance d'une classe fille de telle classe.
- Cet opérateur permet de vérifier qu'un certain objet est une instance d'une classe implémentant telle interface, ou que l'une de ses classes mère l'implémente.

Ce tutoriel est maintenant terminé. J'espère qu'il vous aura plu et apporté beaucoup de connaissances. 

Pour aller plus loin

Dès à présent, vous êtes prêts à être lâchés dans la nature. Cependant, il est possible d'aller encore plus loin dans le monde de l'orienté objet. En effet, la meilleure technique pour progresser est de **pratiquer** afin d'être le plus à l'aise possible avec ce paradigme. Cette pratique s'acquiert en utilisant des **frameworks** orientés objets. Il en existe un bon nombre, dont voici les principaux.

- Symfony. Framework assez complet et puissant, il est beaucoup utilisé dans le monde professionnel (le Site du Zéro l'utilise par exemple). Si ce framework vous intéresse, je vous conseille de lire le tutoriel sur Symfony disponible sur ce site. Si vous avez lu et compris la partie *« Réalisation d'un site web »*, l'apprentissage des bases de ce *framework* sera quasi instantanée (cette partie du tutoriel s'inspire beaucoup de Symfony).
- Zend Framework. Framework encore plus complet, souvent comparé à une usine à gaz. Certains diront que c'est une qualité (car le framework est par conséquent très souple et s'adapte à votre façon de coder), d'autres un défaut. Ce framework est aussi très utilisé dans le milieu professionnel.
- CodeIgniter. Framework bien plus léger mais moins complet. C'est à vous de compléter sa bibliothèque, soit en téléchargeant des scripts que la communauté a développés, soit en les créant vous-mêmes. Lire le tutoriel sur CodeIgniter disponible sur ce site.
- CakePHP. Ce framework a cherché du même côté que Symfony pour trouver son inspiration. Par conséquent, là aussi, l'apprentissage des bases de ce framework sera très rapide. Si vous voulez vous pencher sur CakePHP, je vous conseille le *CakePHP Cookbook*.

Merci à Talus et Lpu8er concernant l'UML, à christophetd pour la correction orthographique de quelques chapitres, à prs513rosewood pour l'installation de Dia sous Mac OS, et à tous les autres qui commentent pour m'aider ou m'encourager !