Solutions pour de l'Héritage multiple en PHP

par Jean-François Lépine (Autres ressources de Jean-François Lépine)

Date de publication : 31 mars 2010

Dernière mise à jour :

PHP ne supporte pas de manière native l'héritage multiple, c'est-à-dire la possibilité pour une classe d'hériter de plusieurs autres classes.

Il est possible, dans une certaine mesure, de pallier en ce problème en faissant appel aux **méthodes magiques**. Je vous propose donc de découvrir ici un modèle de conception qui propose un héritage multiple simple en PHP

I - Introduction

Si PHP ne supporte pas de manière native l'héritage multiple, il est possible, dans une certaine mesure, de pallier en ce problème en faissant appel aux méthodes magiques.

Par exemple, nous voulons créer une classe "child1" qui hérite de trois autres classes, nommées "parent1", "parent2" et "parent3":

```
class parent1{
    public function method1() {
        echo 'Cette fonction est héritée de la classe parent1';
    }
}
class parent2 {
    public $attribute2 = 'demo';
    public function method2() {
        echo 'Cette fonction est héritée de la classe parent2';
    }
}
class parent3 {
    public function method3() {
        echo 'Cette fonction est héritée de la classe parent3';
    }
}
```

Si PHP permet d'utiliser plusieurs interfaces, la conception d'un héritage multiple est plus complexe :

```
    class Child1 implements Interface1, Interface 2 {
        // implémenter plusieurs interface est possible nativement
    }

    class Child1 extends Parent1, Parent2, Parent3 {
        // Ce code génère une erreur
    }
```

II - Solution proposée : utilisation des méthodes magiques

Nous allons commencer par créer notre classe Child1, en lui donnant un tableau (nommé $_tExtends) qui contiendra la liste des classes dont on souhaite qu'elle hérite. Bien évidement, pour la première classe un simple extends suffit.

```
class Child extends Parent1 {
    private $_tExtends          = array('Parent2', 'Parent3');
}
```

Maintenant, nous allons automatiquement instancier ces classes lors de la construction de l'objet. Ces instances seront stockées, nous nous en serviront pour reporter les actions effectuées sur la classe vers les classes mères.

```
class Child extends Parent1 {
    private $_tExtends          = array('Parent2', 'Parent3');
    private $_tExendInstances   = array();
  // ce tableau contient toutes les instances créées par le constructeur

    /**
     * Constructeur
     * création des instances de chaque classe mère
     */
    public function __construct() {
        // ::::: build instance for each parent class :::::
        foreach($this->_tExtends as $className) $this->_tExendInstances[] = new $className;
```

```
    }
}
```

Ensuite, nous allons reporter chaque appel de méthode, si elle n'existe pas dans Child1, vers l'un de ses parents, si cette méthode existe.

```
/**
 * Méthode magique __call()
 * On va reporter chaque appel sur une des instances des classes mères
 * @param string $funcName
 * @param array $tArgs
 * @return mixed
 */
public function __call($funcName, $tArgs) {
    foreach($this->_tExendInstances as &$object) {

  if(method_exists($object, $funcName)) return call_user_func_array(array($object, $funcName), $tArgs);
    }
    throw new Exception("The $funcName method doesn't exist");
}
```

Désormais, tout appel de méthode de Child1 est reporté, si elle n'existe pas dans Child1 même, vers une de ses classes mères. Il est donc possible, comme à l'habitude, de surcharger une méthode de ces classes mères simplement en la déclarant dans Child1.

Enfin, nous allons reporter toutes les lectures d'attributes (accesseurs) vers les attributs des instances des classes mères:

```
/**
 * Méthode magique __get()
 * On va reporter chaque lecture d'attribut (accesseur) sur une des instances des classes mères
 * @param string $varName
 * @return mixed
 */
public function __get($varName) {
    foreach($this->_tExendInstances as &$object) {
        $tDefinedVars   = get_defined_vars($object);
        if(property_exists($object, $funcName)) return $object->{$varName};
    }
    throw new Exception("The $varName attribute doesn't exist");
}
```

Pour au final avoir:

```
class Child extends Parent1 {
    private $_tExtends          = array('Parent2', 'Parent3');
    private $_tExendInstances   = array();

    /**
     * Constructeur
     * création des instances de chaque classe mère
     */
    public function __construct() {
        // ::::: build instance for each parent class :::::
        foreach($this->_tExtends as $className) $this->_tExendInstances[] = new $className;
    }

    /**
     * Méthode magique __call()
     * On va reporter chaque appel sur une des instances des classes mères
     * @param string $funcName
     * @param array $tArgs
     * @return mixed
```

```
     */
    public function __call($funcName, $tArgs) {
        foreach($this->_tExendInstances as &$object) {

  if(method_exists($object, $funcName)) return call_user_func_array(array($object, $funcName), $tArgs);
        }
        throw new Exception("The $funcName method doesn't exist");
    }

    /**
     * Méthode magique __get()
     * On va reporter chaque lecture d'attribut (accesseur) sur une des instances des classes mères
     * @param string $varName
     * @return mixed
     */
    public function __get($varName) {
        foreach($this->_tExendInstances as &$object) {
            $tDefinedVars   = get_defined_vars($object);
            if(property_exists($object, $funcName)) return $object->{$varName};
        }
        throw new Exception("The $varName attribute doesn't exist");
    }
}
```

Nous avons tout ce qu'il nous faut pour pouvoir utiliser notre classe :

```
$oChild = new Child;

// appel d'une méthode parente
$oChild->method2(); // affiche "called in parent 2"

// lecture d'une variable
echo $oChild->attribute2; // affiche "demo"
```

III - Limites et évolutions

Bien entendu, ce mode de conception reste limité. Par exemple, s'il est possible de surcharger une méthode, il n'est pas possible d'utiliser la syntaxe **parent::method()**. Ce modèle de conception n'est adapté que dans des situations simples et ponctuelles.

Par ailleurs, la gestion des références est extrêmement limité : si on peut dans notre cas utiliser un passage par référence classique, les passages par référence arrière sont impossibles, pour la simple raison que déclarer une référence arrière sur la méthode magique __call() n'a aucun impact:

```
public function &__call() {
 return $this->attribute;
}

// ce code est, au moment où je rédige cet article, abosulment identique à celui ci-dessous :


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

// la délcaration de la méthode avec un "&" n'est donc pas pris en compte par PHP
```

Il serait bien entendu possible d'améliorer cette classe (gestion des accès sur des variables privées ou protégées dans le __get, implantation de la méthide __callStatic). Néanmoins, le but de cet exemple est de rester simple et compréhensible, libre à vous de le faire évoluer à votre guise.

IV - Autres solutions pour faire de l'héritage multiple

IV-A - Utilisation de l'extension Runkit

L'extension **Runkit** permet de "réviser" certains fonctionnement de PHP.

Il semble alors possible de faire hériter une classe de plusieurs autres en utilisant une **méthode spécifique**.

Toutefois, l'utilisation de Runkit reste, à mon avis, bien qu'utile, hasardeuse et source d'erreurs fréquentes (dépassements de mémoire, etc.).

IV-B - Les "Traits" ou "Horizontal Reuse"

La difficulté dans l'utilisation de l'héritage multiple réside principalement en une question simple : quelle classe mère utiliser pour exécuter une méthode héritée ?

S'il existe des algorithmes pour répondre à cette question, ceux-ci sont trop lourds pour pouvoir être utilisés quand on a à faire à un langage interprété (donc non compilé), comme c'est le cas pour PHP. C'est la raison pour laquelle l'héritage multiple n'existe pas aujourd'hui nativement en PHP.

Toutefois, une tentative existe pour faire évoluer PHP, de telle sorte qu'il puisse intégrer la notion de "Traits".

Un Trait est essentiellement un "ensemble/bloc de comportements", qu'il est possible de réutiliser ailleurs dans le code.

Les traits ne sont pas à proprement parler des héritages, mais établissent des liaisons conceptuelles entre, d'un côté un objet, de l'autre des blocs de comportements d'autres objets que ce premier souhaite réutiliser :

```
class Child1 {
 use Parent1, Parent2, Parent3;
}
```

Comme son nom l'indique ("Reuse"), il devient possible de réutiliser les méthodes de Parent1, Parent2 et Parent3 dans la classe Child1.

Toutefois, si on évoque la possibilité d'intégrer ces traits dans PHP 6 (voire PHP 5.4), rien à ma connaissance ne semble le confirmer officiellement.

Vous pouvez trouver plus d'informations sur ces traits **Wiki de PHP (en)** ou sur cet ensemble de **ressources de unibe.ch**.