

POO PHP5 : Créer un agrégateur à base de réflexion et de SPL

par Julien Pauli ([Tutoriels](#), [article](#) et [conférences PHP et développement web](#)) ([Blog](#))

Date de publication : 04/05/2008

Dernière mise à jour :

L'icône **agrégation** est une icône **association** UML qui lie deux classes. Alors que l'association indique qu'un objet utilise d'autres objets pour son fonctionnement, l'agrégation indique l'utilisation et le pilotage d'un ou plusieurs objets d'un même type.

La composition, elle, est une agrégation particulière dans laquelle un objet agrégé ne peut exister sans son objet conteneur. La relation d'agrégation permet de répondre efficacement à l'étude des variations et des communalités d'une classe, en cherchant à déléguer des responsabilités à certains objets, agrégés dans d'autres (en général tout ce qui est susceptible de varier).

Dans cet article, nous allons voir comment créer une classe mère qui permettra l'agrégation de ses filles, selon certaines règles.

- I - Introduction au lien UML agrégation
- II - Agregator : notre classe mère
 - II-A - Le UseCase (cas d'utilisation)
 - II-B - Design de la classe mère
 - II-C - SPL en action : itérateurs
- III - Conclusion

I - Introduction au lien UML agrégation

Un objet agrège un autre lorsqu'il en contient une ou plusieurs instances, et que ces instances ne l'aident pas particulièrement dans une tâche précise (chacun sa responsabilité). Bien que ces 2 objets soient alors couplés, il peuvent très bien vivre aussi l'un sans l'autre.

Exemple : un ascenseur agrège des personnes : il peut agréger plusieurs personnes, il peut exister sans personnes (il est alors vide), et les personnes peuvent exister sans être dans l'ascenseur.

Au contraire, la composition est une agrégation particulière : elle est non partageable et non isolable.

Exemple : un hôtel se compose de chambre. Si l'hôtel est détruit, toutes les chambres le sont. Une chambre ne fait partie que d'un seul hôtel à la fois, elle n'est pas partageable, et n'est pas isolable : une chambre fait obligatoirement partie d'un hôtel et ne peut exister sans.

En UML, une agrégation se matérialise par un petit losange blanc, une composition par un losange noir.

En PHP, il n'y a pas moyen de différencier, dans le code source, une agrégation d'une composition.



Agrégation




Composition

II - Agregator : notre classe mère

II-A - Le UseCase (cas d'utilisation)

Habituellement, lorsque vous voulez agréger un objet B dans un objet A, vous procédez comme suit :

- 1 Créer dans la classe A une propriété, non publique en général, qui stockera des instances de B
- 2 Créer un setter setB() accueillant des objets B, et un getter getB(), récupérant un objet B
- 3 Vous avez fait une agrégation :-)

 *Remarquons que les liens "association", et "agrégation", sont difficiles à cerner, les grands penseurs UML et conception définissent d'ailleurs vaguement l'agrégation. Cependant le principe exposé ici se retrouve dans de nombreuses références bibliques, nous l'accepterons donc.*

Nous allons tenter d'automatiser ce processus. Si notre objet A doit agréger des B, mais aussi des C, des D, des E ...

Il va devenir pénible d'écrire toutes les méthodes et les propriétés. Dans la réalité, un tel cas ne se rencontre pas, ou très rarement. Le jeu n'en vaut donc pas la chandelle, cet article n'aura donc pas de réelle application, si ce n'est de vous démontrer la grande flexibilité du modèle objet de PHP.

Ainsi notre classe A va hériter d'**Agregator**, dans laquelle il y aura toute la logique de contrôle des agrégations.

Les spécifications sont les suivantes :

- 1 Tout objet héritant d'Agregator doit pouvoir utiliser une méthode **setSomething()**.
- 2 Something doit représenter une propriété non publique de la classe.
- 3 Il ne peut être affecté à la propriété something qu'un objet instance de la classe **Something** (qu'il faudra évidemment définir)

Cas d'utilisation

```
<?php
class Agregateur extends Agregator
{
    protected $foo;
    protected $bar;
    public $anaska;
}

class Foo { }
class Bar { }

$agregateur = new Agregateur();

$agregateur->setFoo(new Foo); // OK
$agregateur->setBar(new Bar); // OK
$agregateur->setAnaska(new Bar); // KO : $anaska est publique
$agregateur->setAnything(new Anything) // KO : la propriété (et la classe) Anything n'existe pas
$agregateur->setBar('hello world') // KO 'hello world' n'est pas une instance de Bar.
```

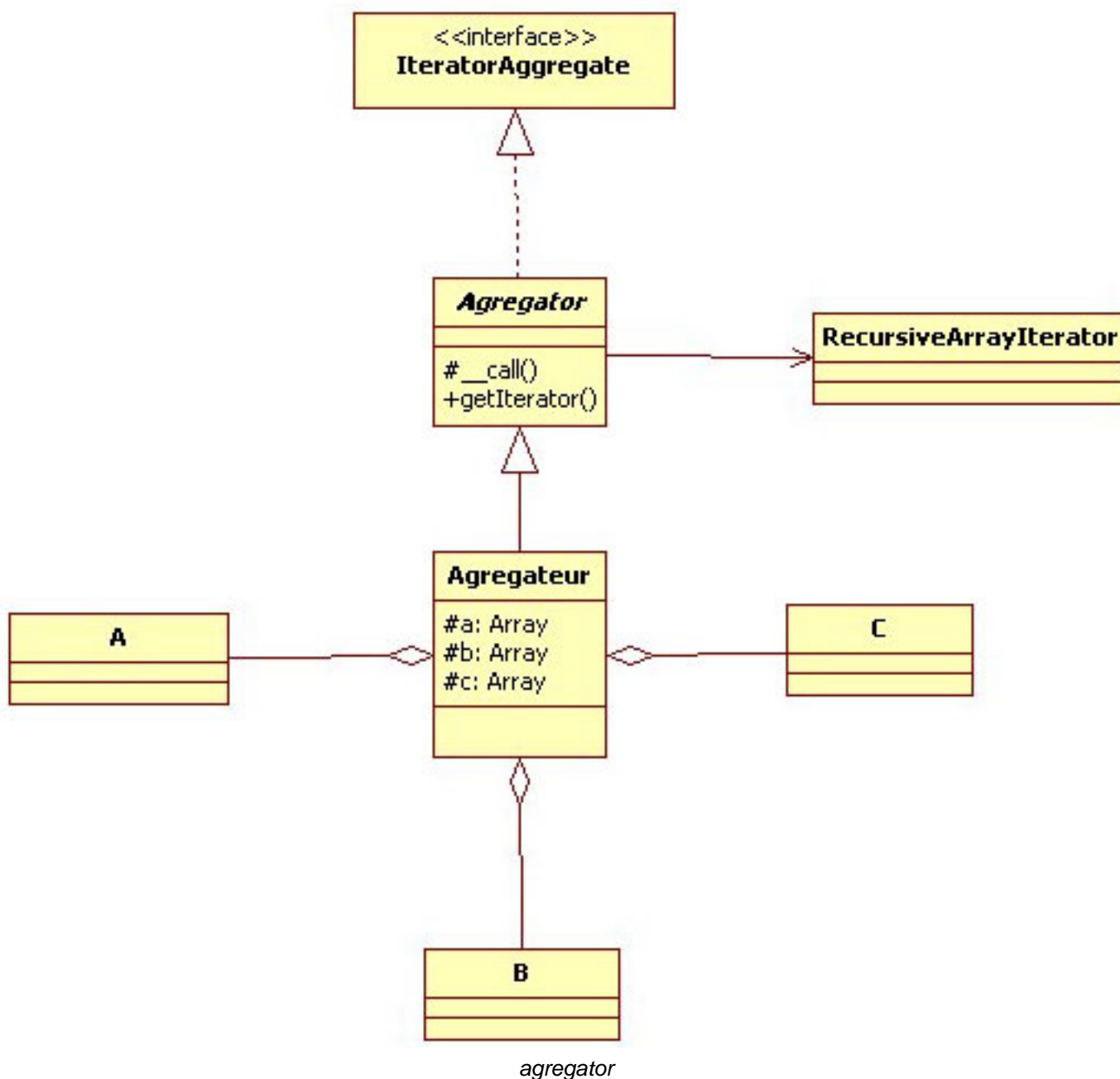
Nous ajouterons ensuite une particularité, qui est en fait assez commune : rendre l'agregateur itérable, afin qu'il nous "rende" tous ce qu'il contient :

itération de l'agregateur



```
<?php
```

itération de l'agrégateur

```
foreach ($agregateur as $objetagrege) {
    var_dump($objetagrege);
}
/* doit afficher quelque chose du style :
object(Foo)#2 (0) { }
object(Bar)#4 (0) { }
object(Bar)#5 (0) { }
tous nos objets agrégés, quels qu'ils soient
*/
```



II-B - Design de la classe mère

Pour arriver à un tel résultat, vous vous doutez bien que la classe mère, **Agregator**, comporte un bout de code. Ce code est intéressant pour tous ceux qui veulent apprendre un peu plus sur la  réflexion, et sur la  SPL

Agregator.php

```
<?php
abstract class Agregator
{
    private $props = array();

    final private function __call($fun,$args) // méthode "set{Paramètre}"
    {
        if (substr($fun, 0, 3) == "set" && // si la méthode commence par "set"
            array_key_exists(0,$args) // et qu'elle possède au moins un argument
        ) {
            $class = new ReflectionClass($this);
            try { // le paramètre existe-t-il dans cette classe?
                $param = new ReflectionProperty($this,$paramName = strtolower(substr($fun,3)));
            } catch(ReflectionException $e) { // non ? gardons le comportement de PHP : pas
d'exceptions
                trigger_error("Call to undefined method
".get_class($this)."::$fun()",E_USER_ERROR); // erreur : la méthode "set{Paramètre}" n'existe pas
            }
            if (!$param->isPublic() && // le paramètre est-il non public ?
                $args[0] instanceof $paramName) { // l'argument passé en est-il une instance ?
                if (!is_array($this->$paramName)) {
un tableau forcé
                    $this->$paramName = array(); // quelque soit le type du paramètre, c'est
                    }
                array_push($this->$paramName,$args[0]); // auquel on rajoute l'argument
"objet"
            }
            if(!in_array($paramName,$this->props)) {
                $this->props[] = $paramName; // nous le mémorisons pour l'itérer plus
tard
            }
        }
    }
} else{
    trigger_error("Call to undefined method ".get_class($this)."::$fun()",E_USER_ERROR);
}
return $this;
}
```

Nous définissons une méthode `__call()` qui va intercepter les méthodes *setSomething()*. Nous la déclarons *final* de manière à ce qu'elle ne puisse être redéfinie.

En effet l'enfant peut changer tout le comportement s'il redéfinit `__call()`, en oubliant d'appeler le parent par exemple.

Puis l'API de réflexion entre en jeu. Cette API introduite en même temps que le modèle objet de PHP5, permet d'inspecter dans des classes et des objets pour fouiller.

Ici, nous analysons `$this`, qui sera résolu en l'objet qui va hériter d'**Agregator**, puis nous vérifions qu'il possède une propriété non publique du même nom que celui de la méthode, privée de la partie 'set', et passée en minuscule.

Dans tous les cas, nous ne préférons pas lever d'exception, afin de garantir le comportement par défaut de PHP (via les erreurs).

Remarquez que nous gérons des collections d'instances. Nous pouvons appeler plusieurs fois le même 'setter', celui-ci va stocker toutes les instances des objets agrégés dans un tableau.

Notre méthode `__call()` retourne `$this`, donc nous pouvons créer une interface fluide et ainsi écrire `$agregateur->setA(new A)->setB(new B)->setC(new C).....`

II-C - SPL en action : itérateurs

Très souvent lorsqu'il y a agrégation, il y a itération. Pourquoi donc ne pas rajouter un itérateur à notre classe mère `agregator` ? C'est très simple en plus : il suffit d'itérer sur toutes les propriétés non publiques de l'objet, et si celles-ci ont été récemment affectées, il faut retourner les instances des objets agrégés.

Heureusement, la SPL met à disposition `RecursiveArrayIterator`, et l'interface `IteratorAggregate` nous sera bien utile :

suite de `agregator.php`

```
<?php
abstract class Agregator implements IteratorAggregate
{
    // ... suite de la classe précédente ...
    final public function getIterator()
    {
        $array = array();
        $self = new ReflectionObject($this);
        foreach ($self->getProperties() AS $property) {
            $array[] = $property->getName();
        }
        $props = array_intersect($this->props, $array);
        $array = array();
        foreach ($props as $prop) {
            $array[] = $this->$prop;
        }
        return new RecursiveIteratorIterator(new RecursiveArrayIterator($array));
    }
}
```

Suite à cela, il est possible d'itérer sur notre classe principale, et elle nous donnera directement tous les objets y étant agrégés :

Utilisation de l'itérateur

```
<?php
class Agregateur extends Agregator
{
    protected $foo;
    protected $bar;
    protected $foobar;
}

class Foo { }
class Bar { }
class Foobar { }
$ag = new Agregateur();

$ag->setFoo(new Foo);
$ag->setBar(new Bar);
$ag->setBar(new Bar);
$ag->setFoobar(new Foobar);

foreach ($ag as $object) {
    var_dump($object);
}
// affiche object(Foo)#1 (0) { } object(Bar)#2 (0) { } object(Bar)#3 (0) { } object(Foobar)#4 (0)
{ }
```

 Un  **bug** dans `RecursiveArrayIterator` empêche l'itérateur de retourner des objets.

Cette partie du code (l'itération des objets agrégés) ne fonctionne qu'à partir de PHP5.3

III - Conclusion

A la question "à quoi cet exemple va-t-il me servir concrètement ?", je réponds : "à rien". En PHP, l'héritage multiple n'existe pas, ainsi si vous héritez d'**Agregator**, vous ne pourrez plus hériter d'autre chose. De la même manière : vous ne pourrez plus définir de méthode `__call()` dans vos enfants car elle est déclarée *final*. C'est fait exprès pour obliger l'enfant à implémenter ce processus.

Quoiqu'il en soit, le but était de montrer que l'API de réflexion est très pratique : elle permet d'introspecter dans des classes, des objets, des fonctions, des paramètres afin de les analyser et de changer un comportement à la volée.

Aussi, très souvent lorsqu'un objet en agrège un autre (ou plusieurs), on a recours à un itérateur. Dans  **ZendFramework** par exemple, c'est le cas.

