

POO PHP5 : Design Pattern observateur aidé de la Standard PHP Library (SPL)

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

Date de publication : 04/02/2008

Dernière mise à jour :

Le design pattern observateur est un classique du GOF, il participe au découplage et à la réduction des dépendances.

En général, 2 interfaces sont utilisées, on peut aussi manipuler des classes abstraites. Nous allons ici montrer un exemple complet de son utilisation et nous allons nous aider de la puissante librairie objet interne de PHP5 : la SPL.

- I - Introduction au design pattern observateur
- II - Exemple : un gestionnaire d'erreur PHP
 - II-A - Problème : gestion du changement
 - II-B - Solution : L'observateur
 - II-C - Allons plus loin : d'autres observateurs
 - II-D - Plus de potentiel : la SPL entre en jeu
- III - Conclusion

I - Introduction au design pattern observateur

Le design pattern observateur autorise un sujet observable, à enregistrer des observateurs. Sur une certaine action, il va notifier ses observateurs, qui vont donc être tenus au courant de ses agissements et changements d'état.

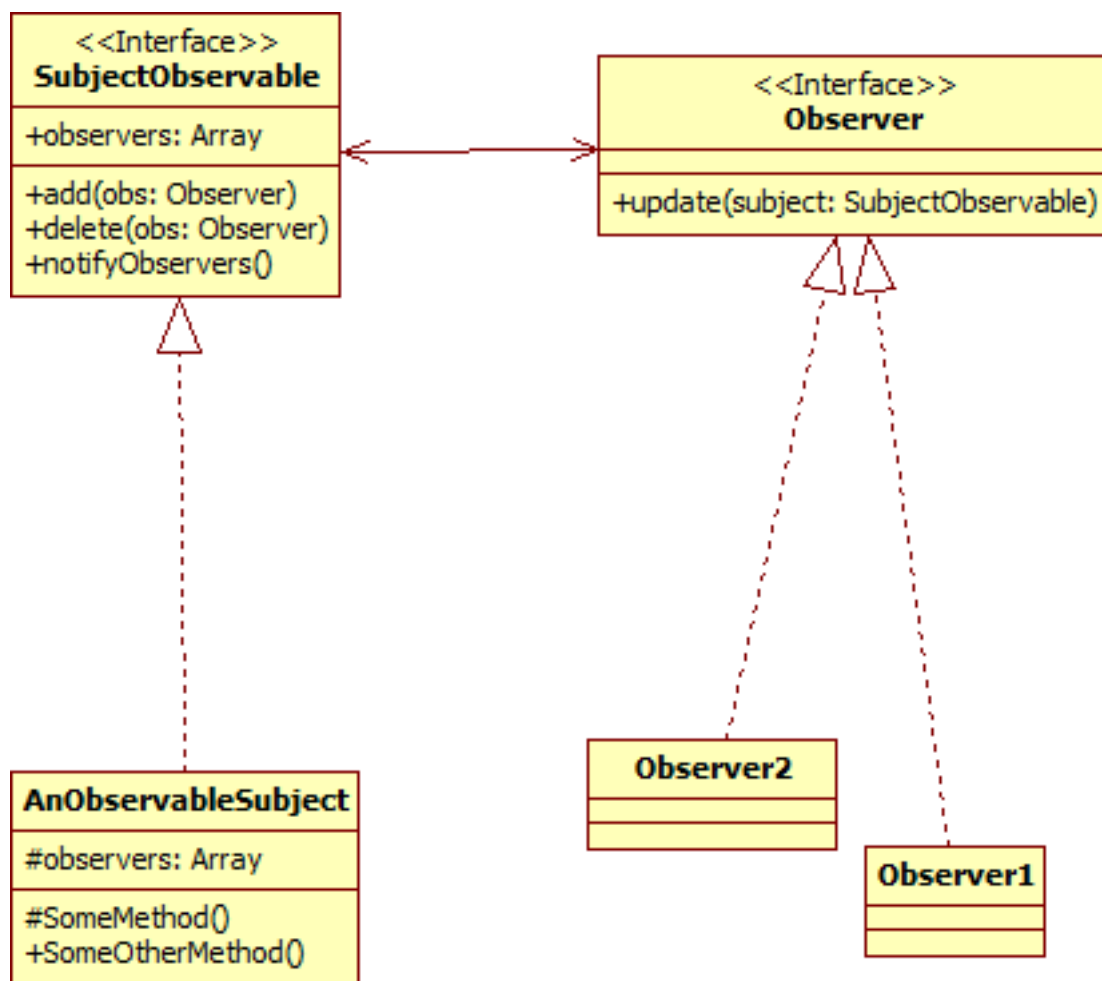
Un sujet pourra donc contenir plusieurs observateurs qui l'écouteront et réagiront à certains de ses évènements.

Le sujet observable ne s'occupe pas de la manière dont ses observateurs vont réagir, l'application gagne donc en découplage et en cohésion : il ne fait que notifier les observateurs, qui vont agir en conséquence, selon leur type et l'état du sujet transmis.

Ce design pattern est donc flexible et extensible, même si on pourra lui trouver des inconvénients.

En général, il fait appel à 2 interfaces, plus rarement à des classes abstraites. Le couplage se situe au niveau des interfaces, et non plus de leur implémentation.

On l'appelle "Observateur-Observable", ou encore "observateurs-sujet" ("Publish/Subscribe").



La théorie la plus généraliste est donc exprimée via le diagramme de classes ci-dessus. Le sujet et les observateurs sont fortement couplés, mais le design pattern représenté par les interfaces permet de centraliser ce couplage au

niveau des interfaces. Je n'ai pas représenté l'implémentation concrète des interfaces (les méthodes dans les classes sujet et observateurs).

Simplement, la méthode ***notifyObservers()*** aura un algorithme ressemblant à "pour tous les observateurs enregistrés : execute update(\$this) dessus".

De cette manière, le sujet observable n'a que faire de la quantité d'observateurs qui le regardent, il ne fait que leur donner un ordre de mise à jour en leur passant lui-même (\$this). Après : chacun des observateurs fait ce qu'il désire du sujet.

II - Exemple : un gestionnaire d'erreur PHP

Notre exemple est lui assez simple : nous voulons redéfinir le gestionnaire d'erreur de PHP

Le couplage est moins fort, le sujet n'aura pas besoin dans notre cas de passer une référence de lui même (\$this) aux observateurs.

Grace à `set_error_handler()`, PHP va passer ses erreurs à notre classe. Celle-ci va alors se charger de les enregistrer pour nous.

Voyons un bref exemple :

index.php, autoload supposé activé

```
<?php
$errorHandler = new ErrorHandler;

set_error_handler(array($errorHandler, 'error'));

echo $arr[0]; // générons une erreur
```

Notre classe ErrorHandler peut ressembler à ceci :

```
<?php
class ErrorHandler
{
    public function error($errno, $errstr, $errfile, $errline)
    {
        if(error_reporting() == 0)
        {
            return;
        }
        if (!$fp = @fopen('my/file', 'a+'))
        {
            throw new Exception('Impossible d\'ouvrir le fichier de log');
        }
        $message = $errstr . ', ' . $errfile . ', ' . $errline;
        @fputs($fp, $message . PHP_EOL);
        return false; // PHP 5.2 : false doit être retourné pour peupler $php_errormsg
    }
}
```

Ce code simple enregistre l'erreur rencontrée dans un fichier. Notez qu'il doit s'agir d'une erreur PHP, les exceptions, elles, sont traitées à part, par un autre gestionnaire.

Si une erreur PHP intervient dans le gestionnaire d'erreur, PHP est capable de s'en sortir tout seul, néanmoins, nous passons ces erreurs sous silence.

Devant `fopen()`, et `fputs()`, nous avons mis un arobase. Celui-ci ne fait qu'une seule chose : pour la commande où il est utilisé, il met (temporairement donc) le rapport d'erreur à off.

L'erreur est donc passée sous silence, mais notre gestionnaire, lui, va quand même l'intercepter. Nous vérifions donc si le rapport d'erreur est à 0. Dans notre cas, cela signifiera qu'une erreur interne à notre classe a été détectée, elle ne sera donc pas traitée par celle-ci.

Evidemment, lorsqu'on fabrique un système qui gère les erreurs, il vaut mieux qu'il en soit exempt lui-même... (il n'y a pas de boucle infinie, PHP sait gérer ce cas là)

De plus, si nous voulons que la variable `$php_errormsg` soit remplie (dans le cas où `track_errors` est à `On` dans le `php.ini`), alors la méthode de traitement des erreurs, **`error()`**, doit retourner `false`. C'est la documentation qui dit ceci hein ;-)

II-A - Problème : gestion du changement

Bien, que se passe-t-il si je veux, en plus d'enregistrer les erreurs dans un fichier, les mémoriser dans une base de données? les envoyer par mail? les afficher à l'écran? les enregistrer dans un tableau php?

Voyons ceci :

```
class ErrorHandler
{
    public function error($errno, $errstr, $errfile, $errline)
    {
        if(error_reporting() == 0)
        {
            return;
        }
        if (!$fp = @fopen('my/file', 'a+'))
        {
            throw new Exception('Impossible d\'ouvrir le fichier de log');
        }
        $pdo = new PDO("mysql:host=localhost;dbname=mydb", 'julien', 'secret');
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        $message = $errstr . ', ' . $errfile . ', ' . $errline;

        $pdo->exec("INSERT INTO matable (`macol`) VALUES ('{$message}')");
        @fputs($fp, $message . PHP_EOL);
        @mail('julien@mail.com', 'erreur applicative', $message);
        return false; // PHP 5.2 : false doit être retourné pour peupler $php_errormsg
    }
}
```

Le problème semble clair, la classe `ErrorHandler` possède trop de responsabilités, le code est peu cohésif : les opérations ne sont pas harmonieuses et partent dans tous les sens. Ainsi, il n'est pas simple de changer un élément, et encore moins de tester ce code !

II-B - Solution : L'observateur

Une solution possible est d'utiliser un design pattern observateur.

La classe `ErrorHandler` gère des événements (ici un seul : la capture d'une erreur PHP), ce n'est pas à elle de les enregistrer, où que ce soit.

Notre classe va agir comme sujet, elle est écoutable, observable.

Des observateurs vont venir se greffer dessus et s'enregistrer sur l'évènement, l'application sera plus dé耦plée, plus cohésive (les responsabilités mieux mises en évidence), mieux testable, et elle gèrera mieux le changement.

Pour ceci nous allons faire appel à 2 interfaces : `Observer`, et `Observable`. Il aurait été possible, voire judicieux d'utiliser des classes abstraites et de les hériter, mais en PHP, on ne peut hériter que d'une seule classe. Ainsi si notre classe `ErrorHandler` devait déjà hériter (ce qui est dans la pratique assez courant), elle ne pourrait plus :

notre nouveau fichier index

```
<?php
$errorHandler = new ErrorHandler;

$errorHandler->add(new FileWriter(dirname(__FILE__).'/log.txt'));

set_error_handler(array($errorHandler,'error'));

echo $arr[0]; // générons une erreur PHP volontaire (notice)
```

ErrorHandler peut donc recevoir des observateurs

observable

```
<?php
interface Observable
{
    function add(Observer $obs);
    function del(Observer $obs);
    function notifyObservers();
}
```

observateur

```
<?php
interface Observer
{
    function update($message);
}
```

notre nouvelle classe ErrorHandler, observable

```
<?php

class ErrorHandler implements Observable
{
    private $_errno;
    private $_errstr;
    private $_errline;
    private $_errfile;
    private $_observers = array();

    public function error($errno, $errstr, $errfile, $errline)
    {
        if(error_reporting() == 0)
        {
            return;
        }
        $this->_errno = $errno;
        $this->_errstr = $errstr;
        $this->_errfile = $errfile;
        $this->_errline = $errline;
        $this->notifyObservers();
        return false; // PHP 5.2 : false doit être retourné pour peupler $php_errormsg
    }

    private function getError()
    {
        return $this->_errstr . ', ' . $this->_errfile . ', ' . $this->_errline;
    }

    public function add(Observer $obs)
    {
        $this->_observers[] = $obs;
        return $this;
    }
}
```

notre nouvelle classe ErrorHandler, observable

```
public function del(Observer $obs)
{
    if (is_int($key = array_search($obs,$this->_observers)))
    {
        unset($this->_observers[$key]);
    }
    return $this;
}

public function notifyObservers()
{
    foreach ($this->_observers AS $observer)
    {
        try{
            $observer->update($this->getError()); // délégation
        }catch(Exception $e){
            die($e->getMessage());
        }
    }
}
```

ErrorHandler doit pouvoir s'ajouter des observateurs (qui vont la "guetter"), s'en supprimer, et les notifier.

A chaque erreur, on utilise **notifyObservers()**, qui va boucler sur tous les observateurs, et leur passer le message d'erreur.

A eux d'en faire ce qu'ils veulent, ça n'est plus du ressort de ErrorHandler.

Le couplage reste le même, mais la cohésion est fortement améliorée. A chaque objet, son rôle, et sa responsabilité.

Le code de la classe ErrorHandler est bien plus facilement testable. De plus, on peut ajouter n'importe quoi comme observateur.

Le fait que **add()** retourne \$this va nous permettre de chaîner les méthodes, regardez plutôt :

observateur fichiers

```
<?php
class FileWriter implements Observer
{
    private $_fp;

    public function __construct($filepath)
    {
        if (FALSE === $this->_fp = @fopen($filepath,'a+'))
        {
            throw new Exception('Impossible d\'ouvrir le fichier de log');
        }
    }

    public function update($message)
    {
        @fputs($this->_fp,$message . PHP_EOL);
    }
}
```

observateur base de données

```
<?php
class BDDWriter implements Observer
```

observateur base de données

```
{
    private $_pdo;
    private $_table;
    private $_col;

    public function __construct($host,$login,$pass,$dbname,$table,$col)
    {
        $this->_pdo = new PDO("mysql:host=$host;dbname=$dbname",$login,$pass);
        $this->_pdo->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION);
        $this->_col = (string)$col;
        $this->_table = (string)$table;
    }

    public function update($message)
    {
        $this->_pdo->exec("INSERT INTO $this->_table (`$this->_col`) VALUES('{ $message}')");
    }
}
```

Chaque observateur est testable de manière unique, il n'a besoin de personne.

Chaque observateur implémente sa propre configuration, une base de données nécessite des identifiants (entres autres), un fichier nécessite une chemin pour le trouver.

Nous injectons après simplement chaque observateur à notre observable ErrorHandler :

injection d'observateurs configurés

```
$errorHandler = new ErrorHandler;

$errorHandler->add(new FileWriter(dirname(__FILE__).'/log.txt'))
    ->add(new BDDWriter('localhost','root','','test','test','nom'));

set_error_handler(array($errorHandler,'error'));

echo $arr[0]; // générons une erreur pour tester
```

II-C - Allons plus loin : d'autres observateurs

Effet immédiat : je peux réutiliser l'observateur fichier (FileWriter) pour lui demander d'écrire vers la sortie standard (php://output), ceci grâce aux contextes de flux de PHP5 (une couche d'abstraction supplémentaire très intéressante).

Autre effet de la séparation des rôles : je peux encore ajouter un observateur, email par exemple

Et même, pour les tests, je peux rajouter un observateur Mock : c'est un observateur qui va simplement stocker dans un tableau PHP, les erreurs déléguées par ErrorHandler. Il me sera possible de les afficher après, regardez plutôt :

un observateur email simple

```
<?php
class MailWriter implements Observer
{
    private $_to;
    const SUBJECT = 'erreur signalée';

    public function __construct($to)
    {
        $this->_to = (string)$to;
        if(filter_var($this->_to,FILTER_VALIDATE_EMAIL) === false)
```

un observateur email simple

```
{
    throw new Exception('Adresse email non conforme');
}

public function update($message)
{
    @mail($this->_to, self::SUBJECT, $message);
}
}
```

observateur mock

```
<?php
class mockWriter implements Observer
{
    private $_messages = array();

    public function update($message)
    {
        $this->_messages[] = $message;
    }

    public function show()
    {
        print_r(new ArrayObject($this->_messages));
    }
}
```

L'implémentation complète est très intuitive :

```
<?php
$errorHandler = new ErrorHandler;

$errorHandler->add(new FileWriter(dirname(__FILE__).'/log.txt'))
    ->add(new BDDWriter('localhost','root','','test','test','nom'))
    ->add(new MailWriter('julien@emailme.com'))
    ->add($mock = new mockWriter());

set_error_handler(array($errorHandler,'error'));

echo $arr[0]; // générons une erreur

$mock->show(); // et affichons là ;)
```

II-D - Plus de potentiel : la SPL entre en jeu

Il est immédiatement remarquable que notre sujet observable (ErrorHandler), agrège des objets observateurs divers.

Pourquoi ne pas utiliser **la SPL, et l'interface Iterator** pour pouvoir facilement lister les observateurs attachés ?

Et même mieux encore, utilisons IteratorAggregate, nous n'aurons pas besoin de définir toutes les méthodes de l'itérateur dans la classe ErrorHandler comme ceci.

Au lieu de cela, il faudra définir une méthode getIterator(), qui va retourner l'objet d'itération. Et encore vraiment mieux : utilisons un ArrayObject, qui contient déjà tout ce qu'il faut !

ErrorHandler complet, itérable

```
<?php
class ErrorHandler implements Observable, IteratorAggregate
```

ErrorHandler complet, itérable

```
{
private $_errno;
private $_errstr;
private $_errline;
private $_errfile;
private $_observers;

public function __construct()
{
    $this->_observers = new ArrayObject();
}

public function error($errno, $errstr, $errfile, $errline)
{
    if(error_reporting() == 0)
    {
        return;
    }
    $this->_errno = $errno;
    $this->_errstr = $errstr;
    $this->_errfile = $errfile;
    $this->_errline = $errline;
    $this->notifyObservers();
    return false; // PHP 5.2 : false doit être retourné pour peupler $php_errormsg
}

private function getError()
{
    return $this->_errstr . ', ' . $this->_errfile . ', ' . $this->_errline;
}

public function add(Observer $obs)
{
    $this->_observers[] = $obs;
    return $this;
}

public function del(Observer $obs)
{
    if (is_int($key = array_search($obs,$this->_observers)))
    {
        unset($this->_observers[$key]);
    }
    return $this;
}

protected function notifyObservers()
{
    // $this est intercepté par l'itérateur
    foreach ($this AS $observer)
    {
        try{
            $observer->update($this->getError());
        }catch(Exception $e){
            die($e->getMessage());
        }
    }
}

public function getIterator()
{
    return $this->_observers;
}
}
```

Remarquez comment ErrorHandler profite de l'itérateur lorsqu'il s'agit d'itérer sur tous les observateurs.

Ceci peut aussi se faire à l'extérieur de la classe :

```

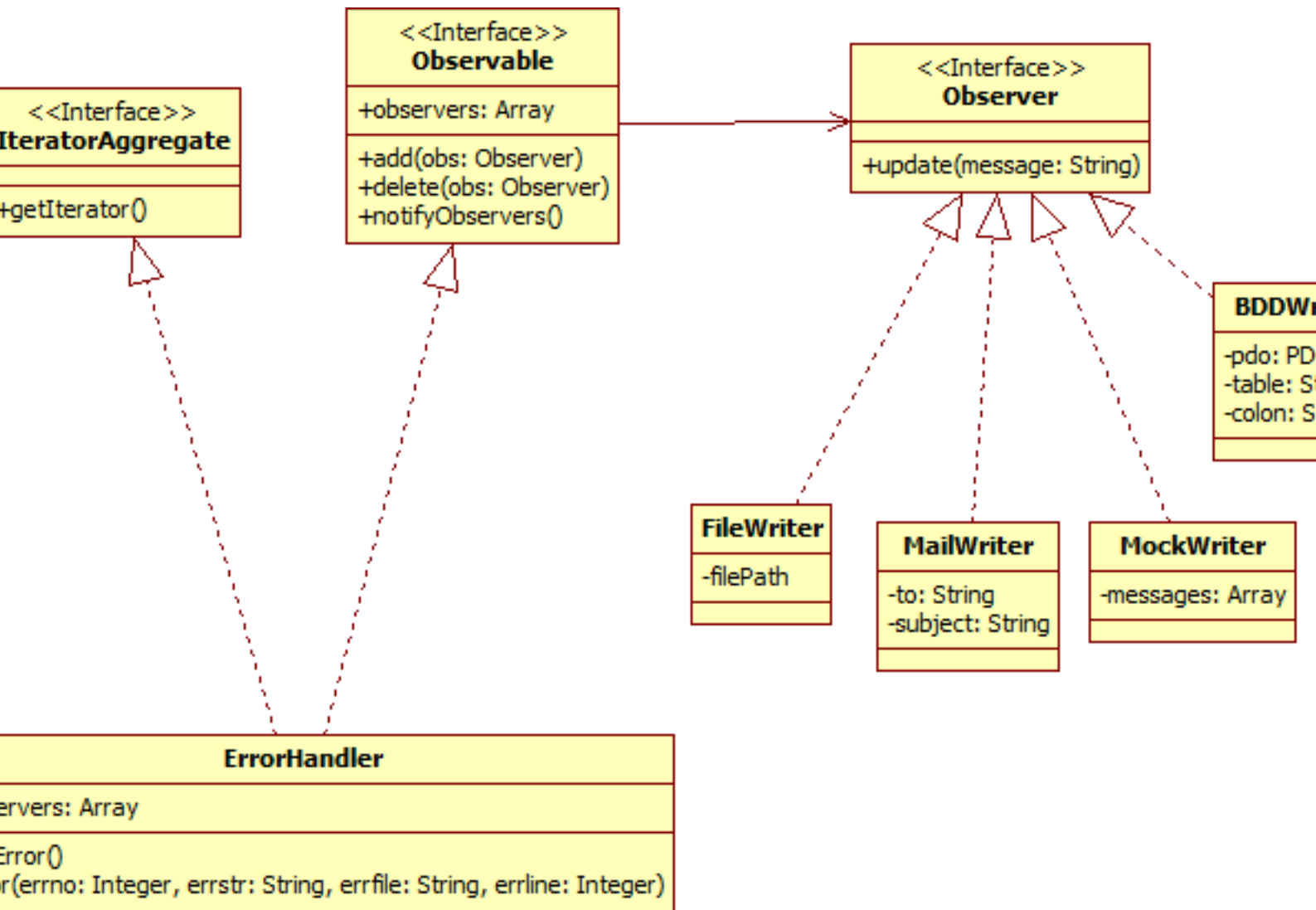
index qui itère sur les observateurs de ErrorHandler
<?php
$errorHandler = new ErrorHandler;

$errorHandler->add(new FileWriter(dirname(__FILE__).'/log.txt'));
$errorHandler->add(new BDDWriter('localhost','root','','test','test','nom'));
$errorHandler->add(new FileWriter('php://output'));
$errorHandler->add(new MailWriter('julien@anaska.com'));
$errorHandler->add($mock = new mockWriter());

set_error_handler(array($errorHandler,'error'));

foreach ($errorHandler as $writer) {
  echo get_class($writer); // facile tout de même non?
}
  
```

Très simple, testable, découplé, efficace, réutilisable; J'espère que ca vous donne des idées ;-)



III - Conclusion

Nous venons de voir comment utiliser un design pattern observateur - observable. Celui-ci est utilisable dès que le couplage (les dépendances) devient trop fort.

Il participe à la séparation des rôles et au découplage applicatif. Ici, le design pattern est relativement lâche, car les objets n'ont qu'un couplage faible.

ErrorHandler nécessite des observateurs, mais pas le contraire. En général, le couplage est très serré, et il y a un lien de dépendance dans les 2 sens, si bien que l'on définit alors l'observateur avec une méthode update(Observable \$obs).

C'est le cas de l'interface SplObserver (car la **SPL** propose déjà un design pattern observateur, regardez [ici](#))

De la même manière, utiliser des classes abstraites aurait permis de passer **notifyObservers()** en private, ce qui est préférable.

Pour aller plus loin : [Pattern Theory of Observer](#)

Motif observateur en PHP5

Introduction aux GRASP patterns

