

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

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

Date de publication : 04/02/2008

Dernière mise à jour : 11/10/2009

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.....	3
II - Exemple : un gestionnaire d'erreur PHP.....	4
II-A - Problème : gestion du changement.....	4
II-B - Solution : L'observateur.....	5
II-C - Allons plus loin : d'autres observateurs.....	8
II-D - Plus de potentiel : la SPL entre en jeu.....	9
II-E - Encore plus d'encapsulation : les designs patterns arrivent en masse !.....	11
III - Conclusion.....	14

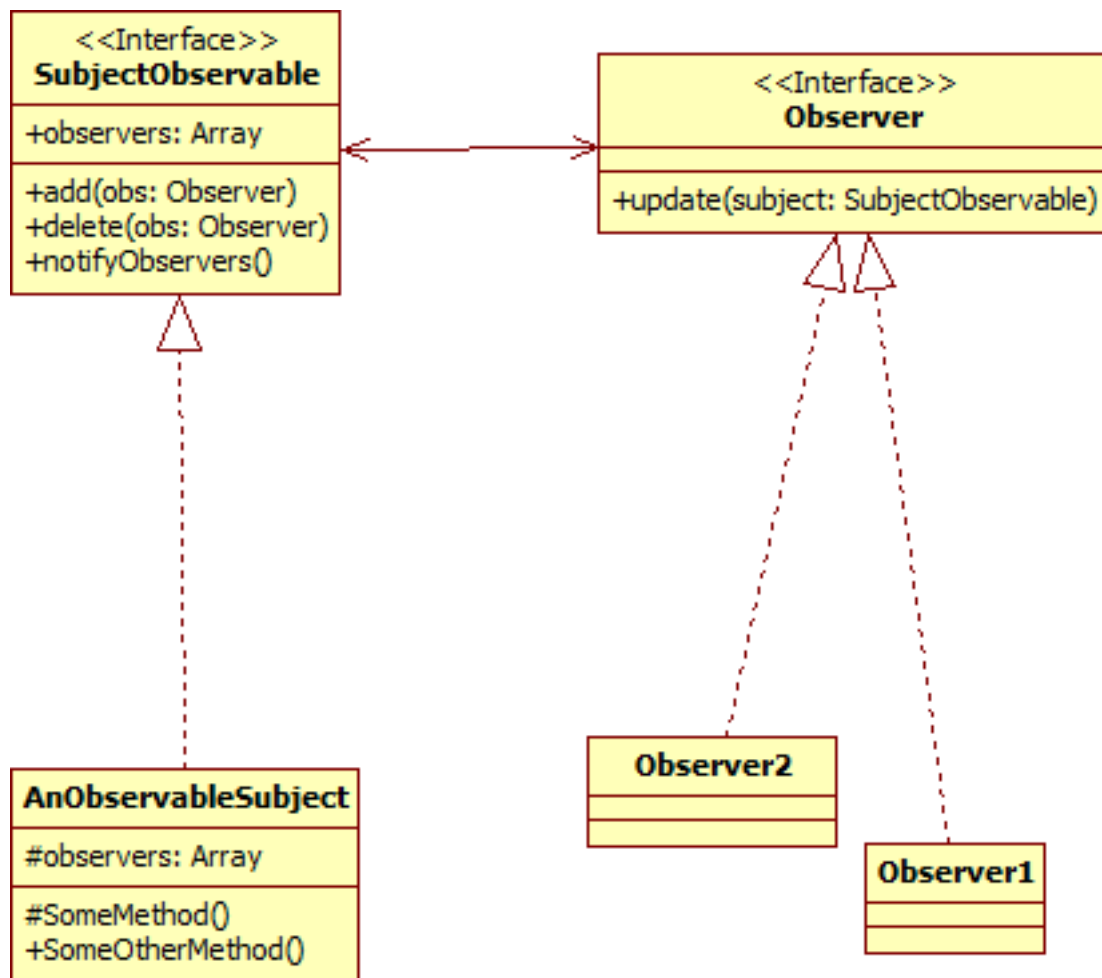
## I - Introduction au design pattern observateur

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 événements. 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'erreurs de PHP  
Grâce 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 true;
    }
}
```


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

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

Devant `fopen()`, et `fputs()`, nous avons mis un arobase afin de masquer les erreurs; En effet, la moindre erreur générée dans le gestionnaire d'erreurs lui-même est interceptée par le gestionnaire d'erreurs classique par défaut de PHP, ceci afin d'éviter une boucle infinie.

Aussi, nous avons décidé que notre gestionnaire ne gèrera pas les erreurs (externes à lui-même donc) précédées d'un arobase : nous vérifions si `error_reporting` est sur 0 ce qui signifie qu'un arobase a été utilisé pour masquer une erreur.

De plus, nous retournons `true` à la fin de notre fonction. Ceci signifie que notre gestionnaire prend totalement la main sur celui de PHP. Si nous avons retourné `false`, le gestionnaire d'erreurs classique de PHP aurait été interrogé après le notre, et le message d'erreur serait probablement apparu à l'écran, ainsi que le remplissage éventuel de la variable `$php_errormsg`.

 **Rappel :** Si une erreur PHP est déclenchée au sein même de la fonction de gestion des erreurs, alors le gestionnaire d'erreurs classique de PHP interviendra et la traitera dans tous les cas. Ceci évite les boucles infinies évidentes.

### 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 true;
    }
}
```

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 ! Tout changement se traduira obligatoirement par la cassure de l'existant : ce code est donc pauvre et pas du tout orienté patterns.


## 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 découplée, plus cohésive (les responsabilités mieux mises en évidence), mieux testable et elle résistera beaucoup mieux le changement.

Pour ceci nous allons faire appel à 2 interfaces, qui sont déjà présentes dans PHP via la  **SPL** : **SplObserver**, et **SplSubject**. 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.

 *Dans un langage à héritage simple, pensez bien à un héritage : si vous héritez, vous ne pourrez plus hériter dans le futur.*

*Souvent il est préférable d'implémenter une interface et de laisser la classe libre d'hériter de ce qu'elle voudra plus tard (éventuellement). A moins d'avoir vraiment beaucoup de code à hériter et un lien "est un / est une sorte de" : l'interface est souvent recommandée.*

notre nouveau fichier index

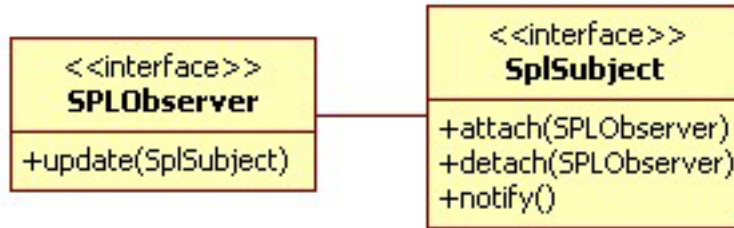
```
<?php
$errorHandler = new ErrorHandler;

$errorHandler->attach(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



*Interfaces de la SPL*

notre nouvelle classe ErrorHandler, observable

```

<?php

class ErrorHandler implements SplSubject
{
    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->notify();
        return true;
    }

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

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

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

    public function notifyObservers()
    {
        foreach ($this->_observers AS $observer) {
            try{
                $observer->update($this); // 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 **notify()**, qui va boucler sur tous les observateurs, et leur passer une instance du gestionnaire d'erreur.

A eux d'en faire ce qu'ils veulent, ça n'est plus du ressort de **ErrorHandler**. En réalité ils utiliseront sa méthode publique **getError()** afin de faire leur travail.

Le couplage est passé sur les interfaces et la cohésion est fortement améliorée. Couplage : ErrorHandler ne connaît pas ses observateurs réels, il ne connaît que leur interface, cohésion : à 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 **attach()** retourne \$this va nous permettre de chaîner les méthodes, regardez plutôt :

#### observateur fichiers

```
<?php
class FileWriter implements SPLObserver
{
    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(SplSubject $errorHandler)
    {
        @fputs($this->_fp,$errorHandler->getError() . PHP_EOL);
    }
}
```

#### observateur base de données

```
<?php
class BDDWriter implements SplObserver
{
    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(SplSubject $errorHandler)
    {
        $this->_pdo->exec("INSERT INTO $this->_table (`$this->_col`) VALUES ('{$errorHandler->getError()}')");
    }
}
```

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'));
```

## injection d'observateurs configurés

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

**i** *Et oui, nous utilisons en plus de l'injection de dépendances, nous respectons donc un bon paquet des principes objets fondamentaux **SOLID**.  
Single Responsibility, Open/Close principe, Liskov Substitution, Interface dependencies et Dependencies injection. Voilà une bonne conception !*

## 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 SplObserver
{
    private $_to;
    const SUBJECT = 'erreur signalée';

    public function __construct($to)
    {
        $this->_to = (string)$to;
        if(filter_var($this->_to,FILTER_VALIDATE_EMAIL) === false) {
            throw new Exception('Adresse email non conforme');
        }
    }

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

### observateur mock

```
<?php
class MockWriter implements SplObserver
{
    private $_messages = array();

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

    public function show()
    {
        return print_r($this->_messages, true);
    }
}
```

L'implémentation complète est très intuitive :

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

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

```

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

echo $arr[0]; // générons une erreur

echo $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, une fois de plus, **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 **SplObjectStorage**, qui contient déjà tout ce qu'il faut !

<b>SplObjectStorage</b>
- \$index
- \$storage
+ attach()
+ contains()
+ count()
+ current()
+ detach()
+ key()
+ next()
+ rewind()
+ valid()

*SplObjectStorage*

### ErrorHandler complet, itératif

```

<?php
class ErrorHandler implements SplSubject, IteratorAggregate, Countable
{
    private $_errno;
    private $_errstr;
    private $_errline;
    private $_errfile;
    private $_observers;

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

    public function error($errno, $errstr, $errfile, $errline)
    {
        // identique
    }

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

    public function attach(SplObserver $obs)
  
```

### ErrorHandler complet, itératif

```

{
    $this->_observers->attach($obs);
    return $this;
}

public function detach(SplObserver $obs)
{
    $this->_observers->detach($obs);
    return $this;
}

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

public function getIterator()
{
    return $this->_observers; // SplObjectStorage est itératif :)
}
}

```

Remarquez comment **ErrorHandler** profite de l'itérateur lorsqu'il s'agit d'itérer sur tous les observateurs. Ceci peut se remarquer à l'extérieur de la classe :

### index qui itère sur les observateurs de ErrorHandler

```

<?php
$errorHandler = new ErrorHandler;

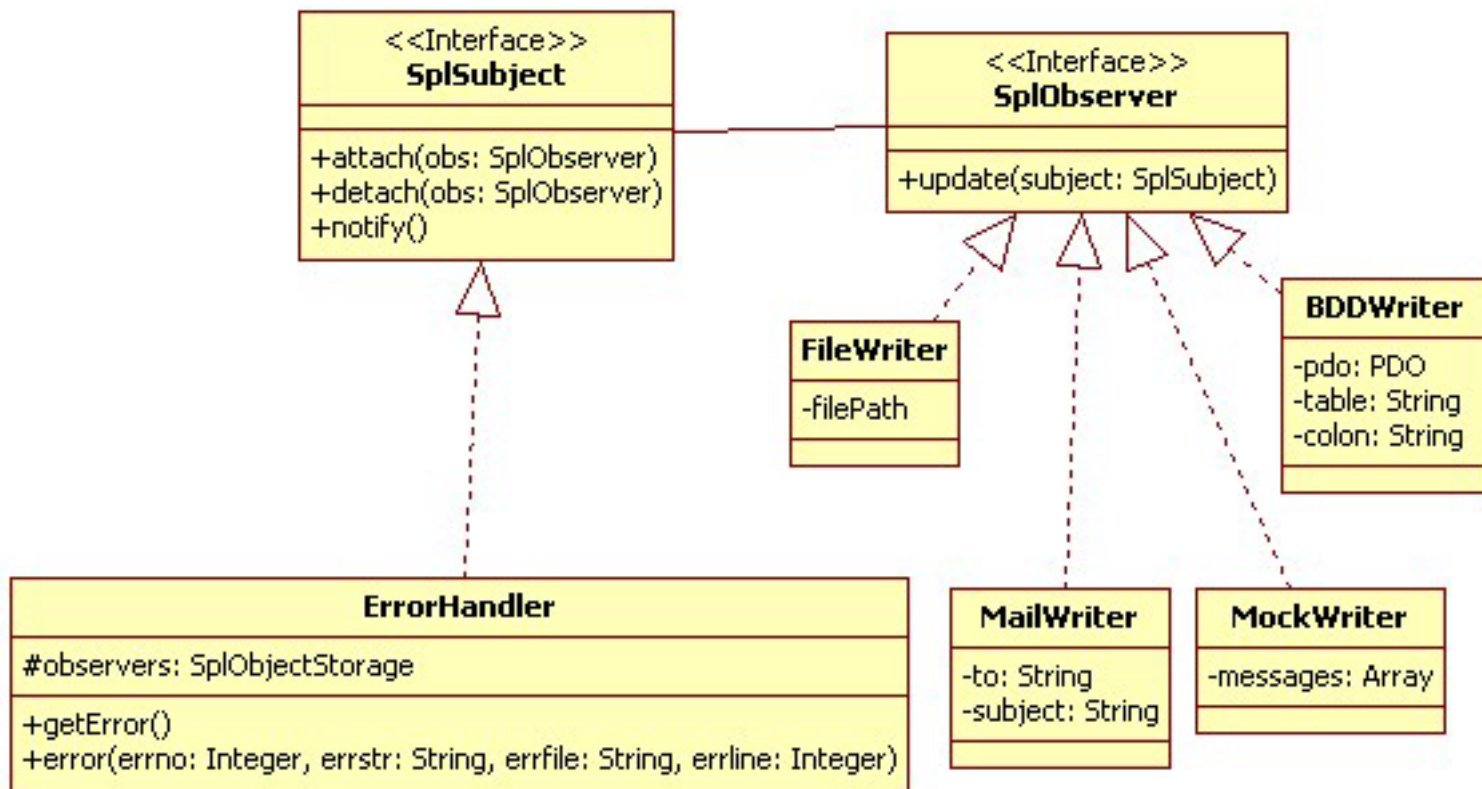
$errorHandler->attach(new FileWriter(dirname(__FILE__).'/log.txt'))
    ->attach(new BDDWriter('localhost','root','','test','test','nom'))
    ->attach(new FileWriter('php://output'))
    ->attach(new MailWriter('julien@anaska.com'))
    ->attach($mock = new mockWriter());

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 ;-)



## II-E - Encore plus d'encapsulation : les designs patterns arrivent en masse !

Halte là, on peut faire encore mieux. Les designs patterns arrivent à commencer par le singleton : notre ErrorHandler est "le" gestionnaire d'erreur de l'application, il est donc unique :

### ErrorHandler devient singleton

```

<?php
class ErrorHandler implements SplSubject, IteratorAggregate, Countable
{
    // ...
    protected static $_instance;

    protected function __construct()
    {
        $this->_observers = new SplObjectStorage();
    }

    public static function getInstance()
    {
        if(self::$_instance == null) {
            self::$_instance = new self();
        }
        return self::$_instance;
    }

    public static function resetInstance()
    {
        self::$_instance = null;
        return self::getInstance();
    }

    // suite du code ici
}

```

Voilà pour notre singleton. On pourrait d'ailleurs lui donner plus de responsabilités : c'est à notre classe ErrorHandler de savoir démarrer et arrêter la gestion des erreurs, voyez plutôt :

### Encore des responsabilités pour ErrorHandler

```
<?php
class ErrorHandler implements SplSubject, IteratorAggregate, Countable
{
    public static function start()
    {
        set_error_handler(array(self::getInstance(), 'error'));
        return self::getInstance();
    }

    public static function stop()
    {
        restore_error_handler();
        return self::getInstance();
    }

    // suite du code ici
}
```

On retourne l'instance à chaque fois, ceci va permettre de chainer les méthodes, voici un exemple de l'utilisation de la classe une fois enrichie :

### utilisation du ErrorHandler enrichi en fonctionnalités

```
<?php
ErrorHandler::start()->attach(new FileWriter(...));
```

Mais au fait : ErrorHandler utilise des observateurs. On crée ErrorHandler, on crée les observateurs et on encapsule tout cela. ErrorHandler ne serait-il pas capable de savoir fabriquer des observateurs pour nous ? Mais si ! Et hop, un pattern fabrique qui arrive :)

### ErrorHandler devient fabrique d'observateurs

```
<?php
class ErrorHandler implements SplSubject, IteratorAggregate, Countable
{
    public function factory($listener, array $args = array())
    {
        $reflect = new ReflectionClass($class);
        return $reflect->newInstanceArgs($args);
    }

    // suite du code de ErrorHandler ici
}
```

Ce code simple utilise la réflexion pour dans un premier temps nous envoyer une exception (RunTimeException) s'il ne trouve pas la classe (il faudra la charger avant au moyen d'un require ou jouer avec l'autoload) et dans un deuxième temps nous permettre de passer un tableau d'options à la méthode, qui sera dispatché sur le constructeur de la classe en question. Et oui, la reflexion regorge de petites astuces qu'un maitre en PHP doit maitriser ;-)

### Exemple d'utilisation de la fabrique précédemment écrite

```
<?php
ErrorHandler::start()->attach(ErrorHandler::factory('FileWriter', array('path/to/foo.log')));
```

Evidemment le cas ici est bateau, une fabrique plus complexe peut être mise en place.

Dernier point, utilisons la flexibilité des méthodes magiques de PHP pour réécrire **attach()** et **detach()** autrement :

### Un peu de magie PHP

```
<?php
class ErrorHandler implements SplSubject, IteratorAggregate, Countable
{
    public function __call($funcnt, $args)
    {
        if (preg_match('#(att|det)ach(\w+)#', $funcnt, $matches)) {
```

### Un peu de magie PHP

```
$meth      = $matches[1].'.ach';  
$listener = strtolower($matches[2]);  
return $this->$meth(self::factory($listener, $args));  
}  
throw new BadMethodCallException("methode $funct inexistante");  
}  
  
// suite du code de ErrorHandler ici  
}
```

Et voilà! On peut désormais utiliser une API plus sympa, dans ce style là :

### Utilisation de la méthode magique \_\_call

```
<?php  
$instance = ErrorHandler::start();  
$instance->attachFiles('path/to/log.log')  
->attachMail('foo@bar.baz');
```

### III - Conclusion

Nous venons de voir comment à partir d'un projet simple en apparence, la conception à base de patterns et de théorie générale de l'objet peut s'appliquer. Un design pattern Observateurs/Sujet au centre du débat, aidé de ses amis qui viennent naturellement se greffer dessus.

Observateur/Sujet participe à la séparation des rôles et au découplage applicatif.

Nous remarquons aussi l'utilisation somme toute intensive de la SPL : les 2 interfaces du patterns y sont déjà présentes. De plus, l'objet **SplObjectStorage** est extrêmement pratique pour stocker des objets (comme son nom l'indique), car il va se charger de toute la logique de recherche de ceux-ci, et en plus, il est itératif.

Une fois de plus, on voit nettement que PHP5 permet une conception objet poussée mais néanmoins souple : il est très dommageable de faire l'impasse sur cette étape indispensable lorsqu'on souhaite faire perdurer un projet.

Pour aller plus loin : **Pattern Theory of Observer**

**Motif observateur en PHP5**

**Introduction aux GRASP patterns Introduction à la SPL de PHP5**