



PHP & DESIGN PATTERNS

Gérald Croës - gerald@copix.org

http://gcroes.com/conf/php/quebec/2007/design_patterns.pdf

Objectifs de la présentation

- ➊ Comprendre ce qu'est un pattern
- ➋ En connaître les principaux représentants
- ➌ Exemples d'utilisation
- ➍ Des ressources pour aller plus loin

- ➎ Apporter des idées conceptuelles



Définition

❖ **Modèles de conception**

parfois aussi « Motifs de conception » ou « Patrons de conception ».

❖ **Solutions à des problèmes classiques.**

(Règle de 3 utilisations)

❖ **Indépendants du langage.**

Ils sont partout...

- ⚙️ Composition musicale (cadence, apogiatures, tonalité, rubato, ...)
- ⚙️ Communication (intonations dans le discours, métaphores et exemples, ...)
- ⚙️ Graphisme, ergonomie (positionnements, codes couleurs, ...)
- ⚙️ ...

Tout ce qui représente une façon de procéder pour arriver à un résultat

Apparition des patterns

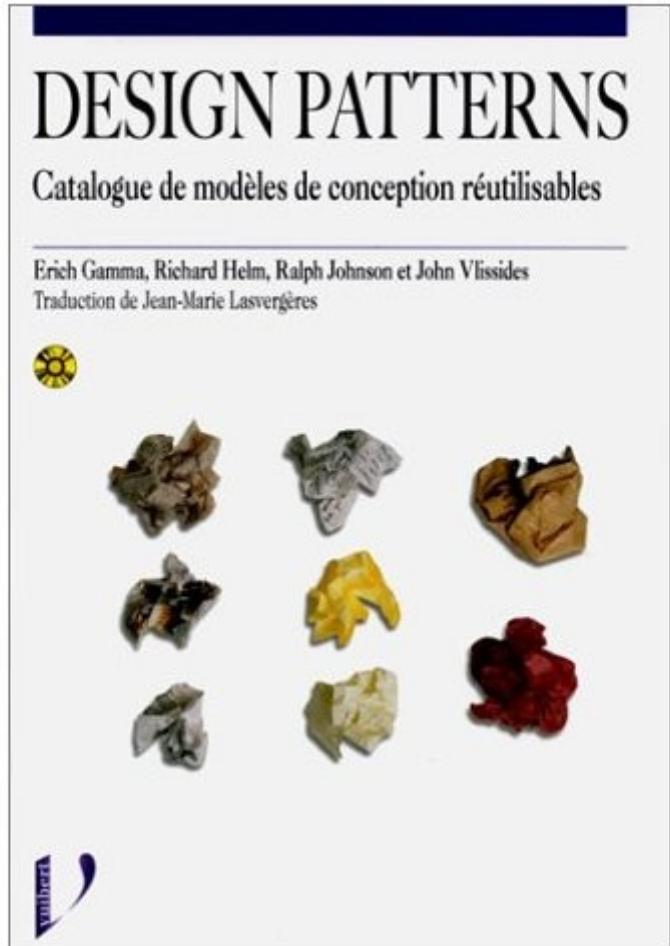


C. Alexander « A Pattern Language: Towns, Buildings, Construction » [1977]

« Les utilisateurs connaissent mieux le bâtiment dont ils ont besoin que les architectes »

« Chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution à ce problème, de telle façon que l'on puisse la réutiliser des millions de fois et ce jamais de la même manière » [AIS+ 77]

L'ouvrage de référence



◆ GoF « Gang of Four »

Design patterns. Elements of reusable Object-Oriented Software [1994]

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Pourquoi les étudier ?

- ✿ Catalogue de solutions.
- ✿ Bénéficier du savoir faire d'experts dans des contextes éprouvés. (fiables, robustes & connus)
- ✿ Facilite la conception.

Pourquoi les utiliser ?

- ➊ Ne pas réinventer la roue.
- ➋ Facilite la communication entre développeurs.
- ➌ Pour résoudre un problème

Fiche d'identité



Nom



Description du problème



Description de la solution

exemples, modèles, liste des éléments et des relations



Conséquences

critiques, impacts sur l'application

« Par convention, 3 utilisations couvertes de succès »

Les 5 patterns de création



Création

Fabrique, Fabrique abstraite, Monteur (Builder), Prototype, Singleton



Abstraction du processus de création.



Encapsulation de la logique de création.



On ne sait pas à l'avance ce qui sera créée ou comment cela sera créé.

Les 7 patterns de structure



Structure

Adaptateur, Pont, Composite, Decorateur, Façade, Poids mouche, Proxy



Comment sont assemblés les objets.



Découpler l'interface de l'implémentation.

Les 11 patterns comportementaux



Comportement

Interprète, Patron, Chaîne de responsabilité, Commande, Itérateur, Médiateur, Memento, Observateur, Etat, Stratégie, Visiteur



Mode de communication entre les objets

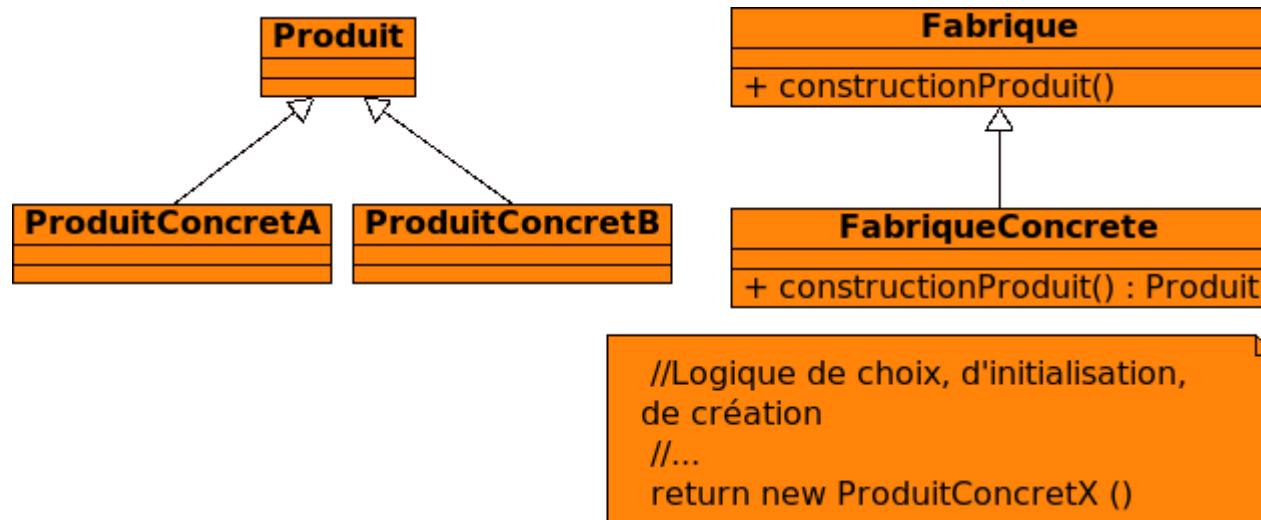
Quelques exemples pour commencer

- ❖ Fabrique
- ❖ Singleton
- ❖ Adaptateur
- ❖ Décorateur
- ❖ Observateur
- ❖ Itérateur

[Création] Fabrique

- ➊ Problématique : Obtenir facilement un objet prêt à l'emploi et qui correspond à nos besoins.
- ➋ Solution : Une classe / Une méthode qui encapsule la logique de création des objets en question.

[Création] Fabrique, diagramme



[Création] Fabrique, exemple

```
require_once ('DBAbstract.class.php');

class DBFactory {

    static function create ($dataSourcId){

        switch (self::__getDriver ($dataSourcId)) {

            case self::__MySQL :

                require_once ('DBMySQL.class.php');

                return new DBMySQL ($dataSourcId);

            case self::__MySQLI :

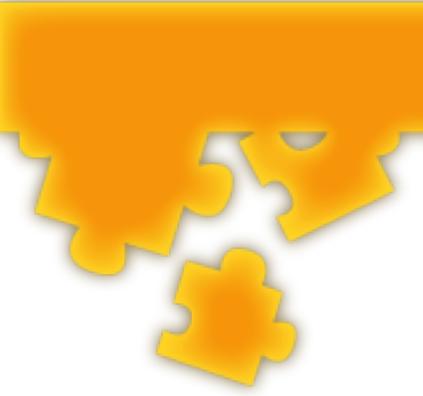
                require_once ('DBMySQLI.class.php');

                return new DBMySQLI ($dataSourcId);

        }

    }

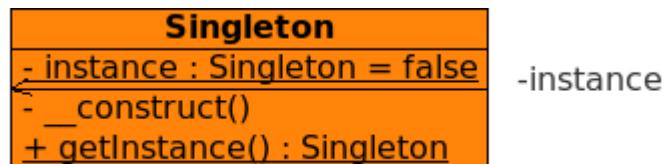
}
```



[Création] Singleton

- ➊ Problématique : S'assurer qu'il existe une seule instance d'un objet donné pour toute l'application.
- ➋ Solution : Une méthode statique pour contrôler l'instanciation. Rendre ce processus d'instanciation l'unique solution possible pour la classe en question.

[Création] Singleton, diagramme



```
if (self::$instance === false){  
    self::$instance = new Singleton ();  
}  
return self::$instance;
```

[Création] Singleton, exemple

```
class ApplicationConfiguration {  
    private static $_instance = false;  
  
    public static function getInstance (){  
        if (self::$_instance === false){  
            self::$_instance = new ApplicationConfiguration ();  
        }  
        return self::$_instance;  
    }  
  
    private function __construct (){  
        //chargement du fichier de configuration  
    }  
}
```

[Création] Singleton, pièges & différences



N'est pas une classe statique

Une instance à manipuler.

Conserve un contexte

Peut être passé en paramètre à une méthode



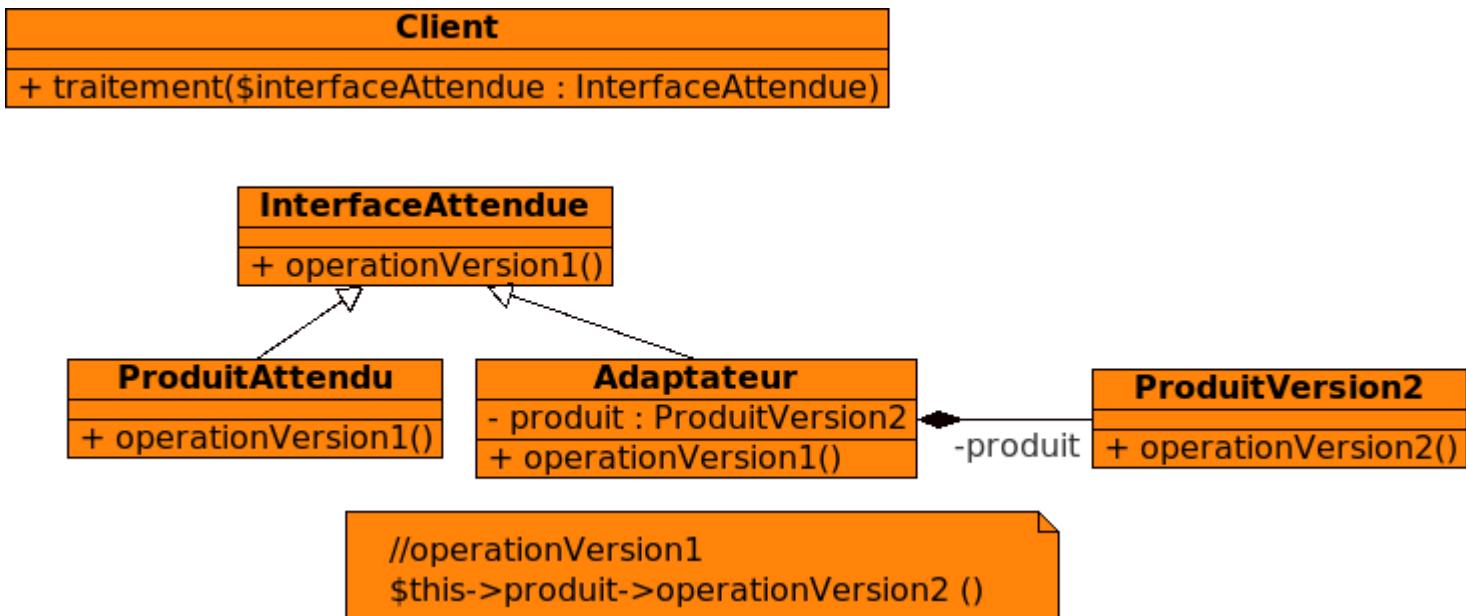
N'est pas une variable globale

Eviter la singletonite

[Structure] Adaptateur

- 💡 Problématique : Ajuster l'interface d'un objet à celle attendue par le code client.
- 💡 Solution : L'adaptateur conserve une instance de la classe adaptée et convertit les appels d'une interface existante vers l'interface implémentée.

[Structure] Adaptateur, diagramme



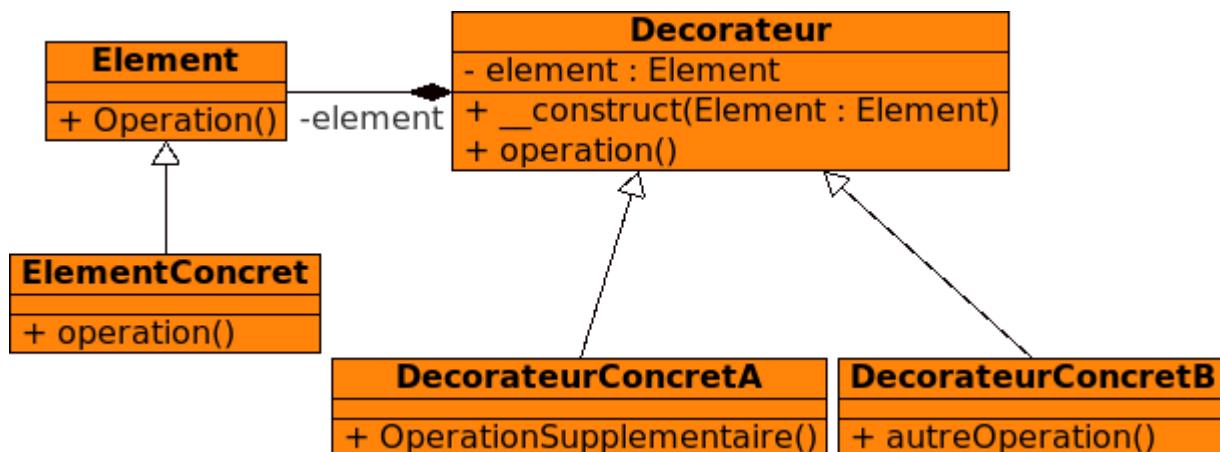
[Structure] Adaptateur, exemple

```
class AuthAutreBibliothèque {  
    public function getLevel (){//calcul le niveau de l'utilisateur}  
}  
  
class AuthAdapter implements Auth {  
    private $_adapted = null;  
    function __construct ($login){  
        $this->_adapted = $login;  
    }  
    function isConnected (){  
        return $this->_adapted->getLevel () >  
            AuthAutreApplication::ANONYMOUS;  
    }  
}
```

[Structure] Décorateur

- ➊ Problématique : Rajouter des fonctionnalités à des composants existants sans utiliser l'héritage.
- ➋ Solution : Encapsuler l'objet existant et y ajouter des comportements nouveaux.

[Structure] Décorateur, diagramme



[Structure] Décorateur, exemple

```
class decorated {}
```

```
abstract class Decorator {
```

```
    private $_decorated;
```

```
    function __construct ($decorated){
```

```
        $this->_decorated = $decorated;
```

```
}
```

```
    function operaton (){
```

```
        return $this->_decorated->operation ();
```

```
}
```

```
}
```

[Structure] Décorateur, exemple (2)

```
class Decorator1 extends Decorator {
```

```
    function operation2 (){
```

```
        //autre traitement
```

```
    }
```

```
}
```

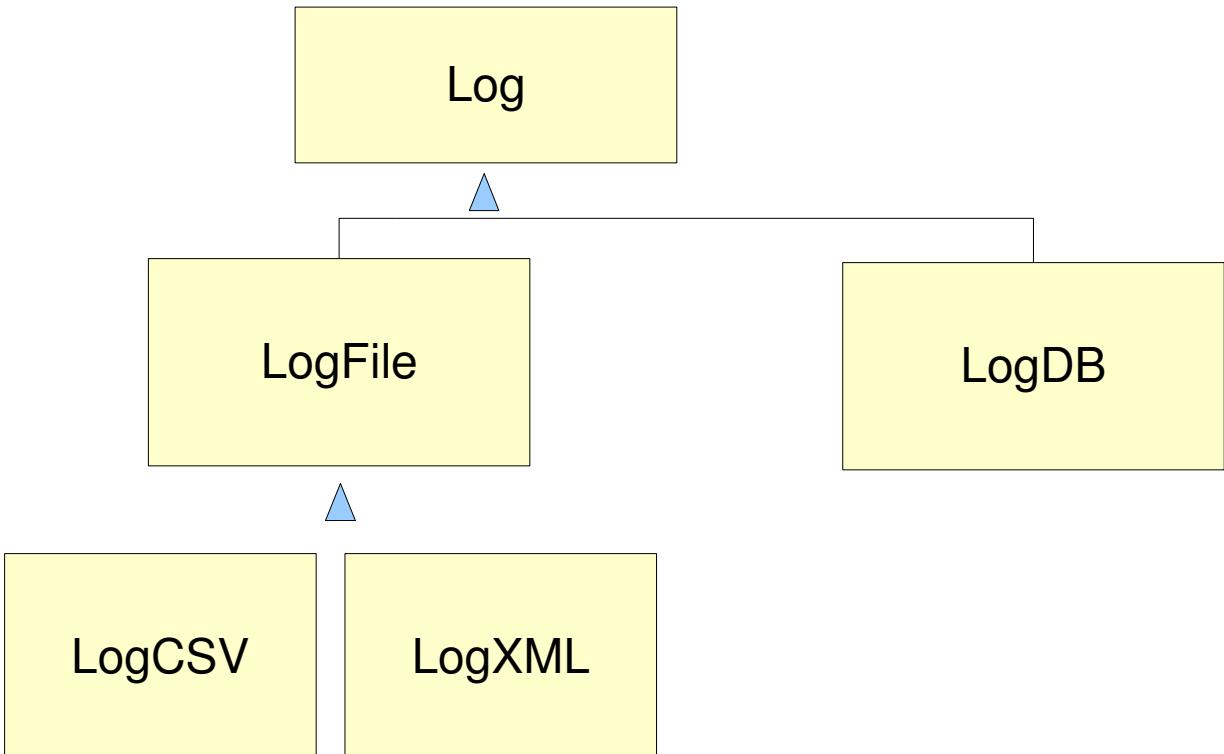
```
class Decorator2 extends Decorator {
```

```
    function operation3 (){
```

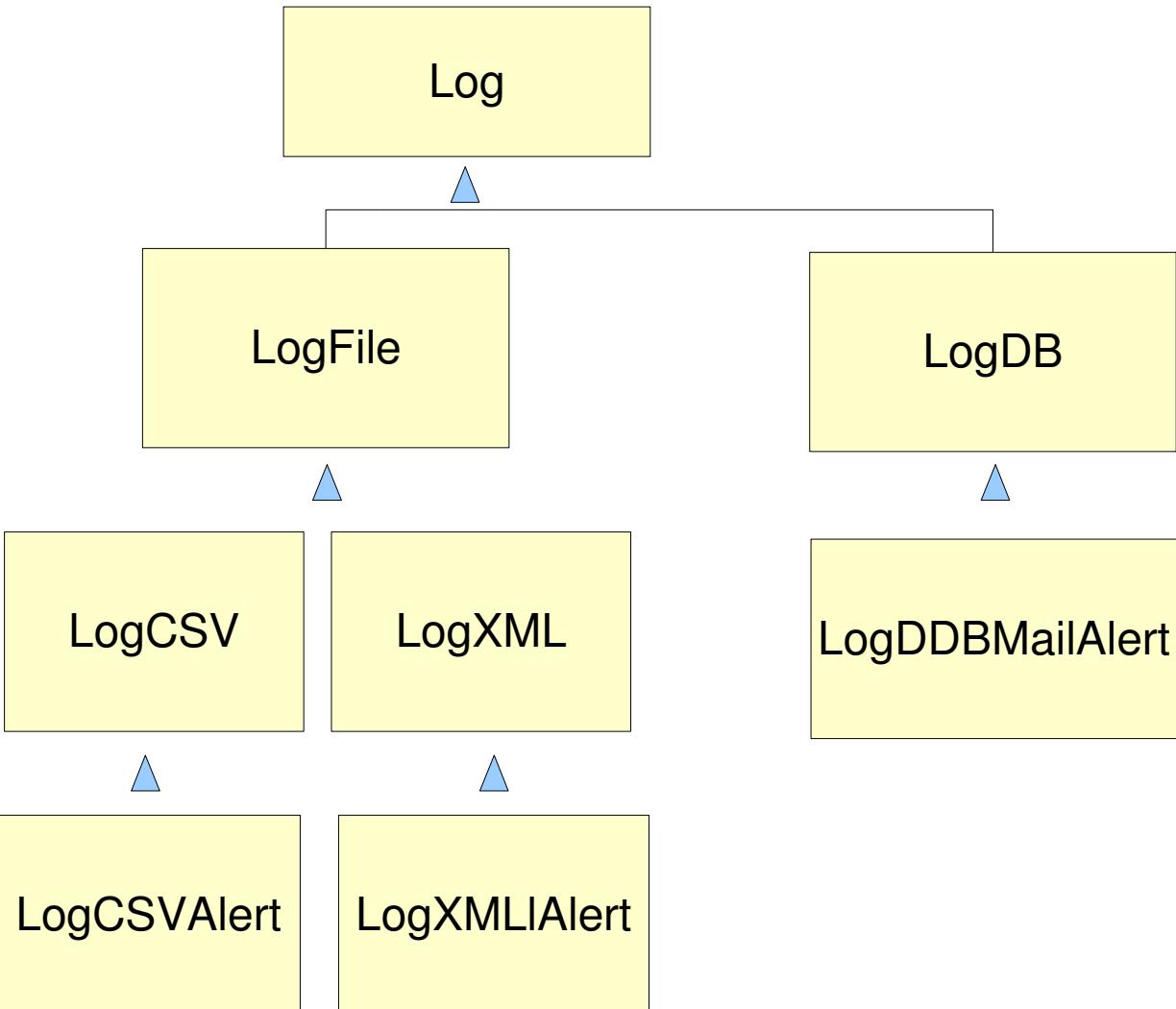
```
        //...
```

```
}
```

[Structure] Decorateur VS Héritage

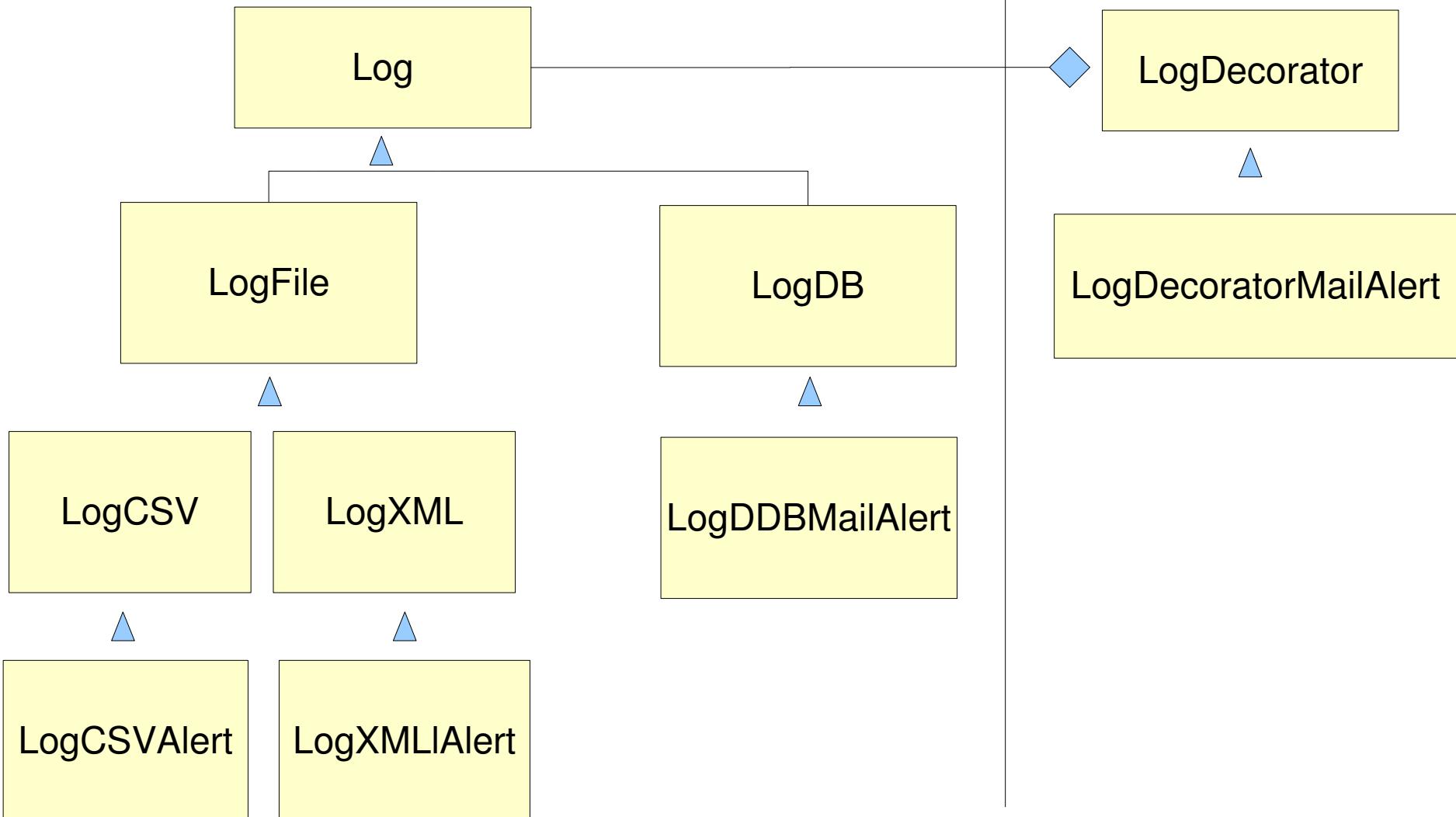


[Structure] Decorateur VS Héritage



Une arborescence qui se complexifie

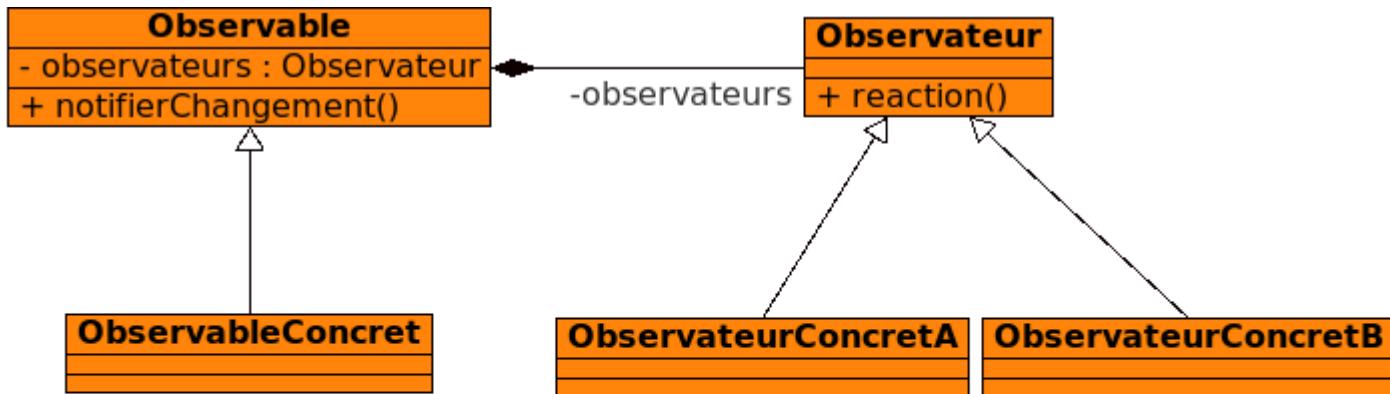
[Structure] Decorateur VS Héritage



[Comportement] Observateur

- ➊ Problématique : Permettre à un objet de réagir aux comportement d'un autre sans pour autant les lier « en dur ».
- ➋ Solution : Définir un objet comme « observable » et donc capable de notifier des changements à des observateurs, quels qu'ils soient.

[Comportement] Observateur, diagramme



```
//notifierChangement
foreach ($this->observateurs as $o){
    $o->reaction ();
}
```

[Comportement] Itérateur

- ➊ Problématique : Parcourir des collections d'objets diverses, éventuellement de différentes façons, sans risque pour le contenu.
- ➋ Solution : Utiliser un objet qui dispose de la connaissance nécessaire à la navigation dans la collection avec une interface unique.

[Comportement] Itérateur, diagramme

Collection
+ getIterator() : Iterator

Iterator
+ next()
+ key()
+ current()
+ rewind()
+ valid()

```
foreach ($collection->getIterator () as $element){  
    //...  
}
```

[Comportement] Itérateur & PHP



SPL & Iterateurs

« Iterator » Utilisable dans les foreach

ArrayIterator, RecursiveArrayIterator,
DirectoryIterator,
RecursiveDirectoryIterator, RegexIterator,
SimpleXMLIterator, ...

[Comportement] Exemple sans Itérateur

```
$hdl = opendir('./');

while ($dirEntry = readdir($hdl)) {

    if (substr($dirEntry, 0, 1) != '.') {

        if (!is_file($dirEntry)) {

            continue;

        }

        echo $dirEntry, '<br />';

    }

}

closedir($hdl);
```

[Comportement] Exemple avec Itérateur

```
try{  
    foreach ( new DirectoryIterator('./') as $item ) {  
        echo $item.'<br />';  
    }  
}  
  
catch(Exception $e){  
    echo 'No files Found!<br />';  
}
```

[Comportement] Etendre un itérateur

```
class imageliterator extends directoryIterator {  
  
    function isImage() {  
        if(!parent::isFile() or !preg_match(`\.(gif|png|jpe?g)$`i,  
        parent::getFilename()) ) {  
            return false;  
        }  
        return true;  
    }  
  
    function valid() {  
        if (parent::valid()) {  
            if (!$this->isImage()) {  
                parent::next();  
                return $this->valid();  
            }  
            return true;  
        }  
        return false;  
    }  
}
```

Et plus encore....

- ❖ Fabrique abstraite
- ❖ Monteur
- ❖ Proxy
- ❖ Façade
- ❖ Visiteur
- ❖ Chaîne de responsabilité

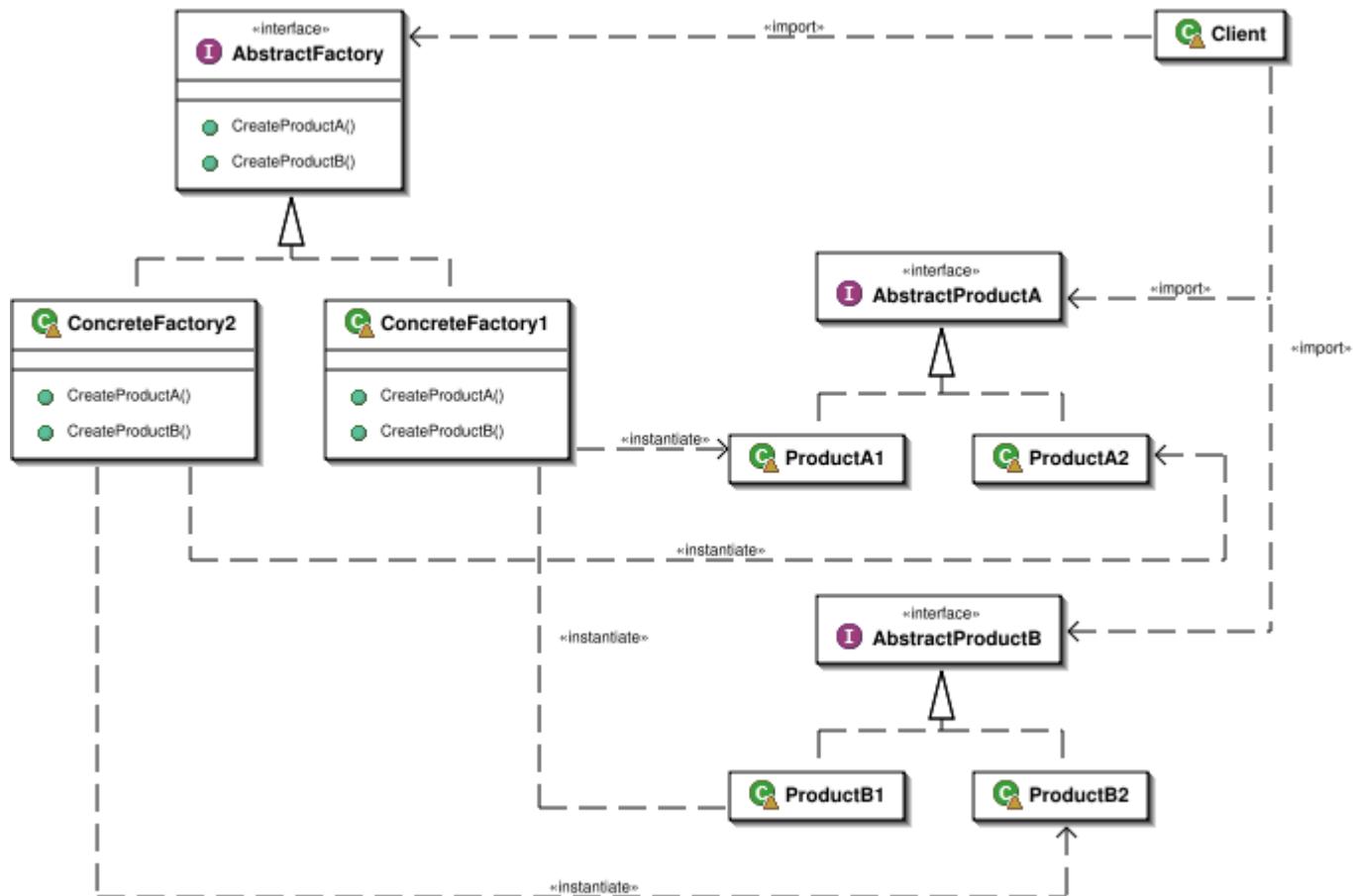
[Créateur] Fabrique abstraite

- ➊ Problématique : Disposer d'une interface pour créer des familles d'objets.
- ➋ Solution : Une fabrique de fabrique.

[Créateur] Fabrique abstraite, diagramme

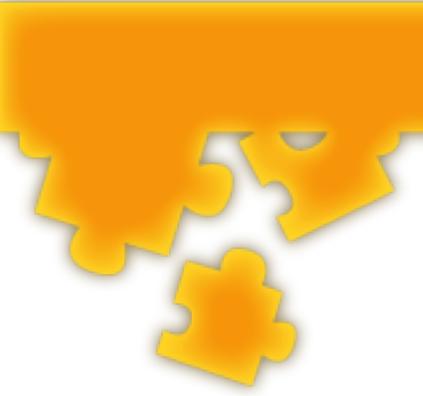


Création d'une famille d'objet



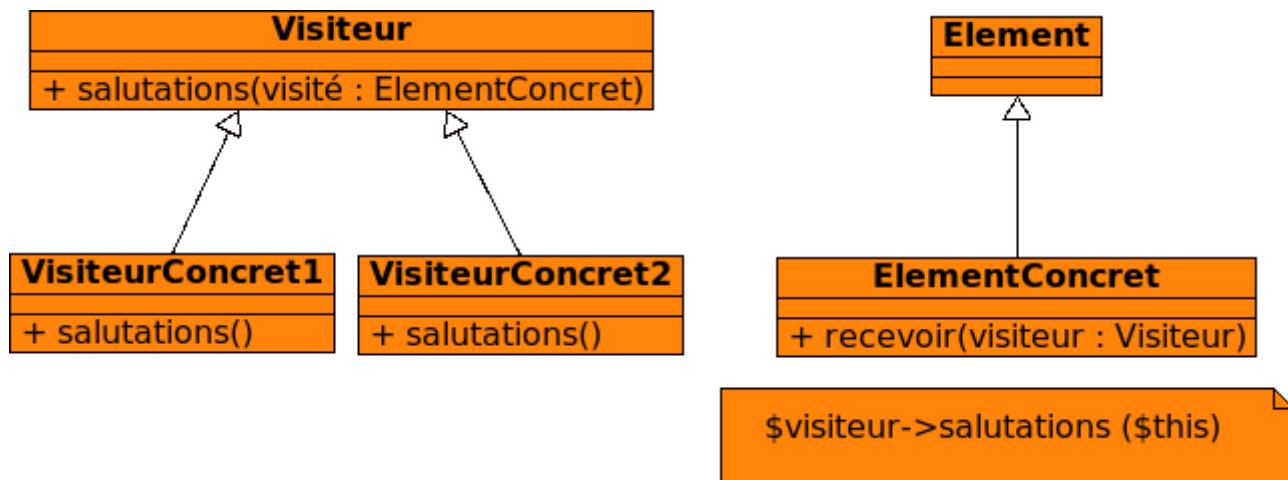
[Créateur] Fabrique abstraite, exemple

```
abstract class AbstractDocumentFactory {  
    abstract public function getDevis ();  
    abstract public function getAdhesion ();  
}  
  
class PdfDocumentFactory {}  
  
class HTMLDocumentFactory {}  
  
abstract class Devis {}  
abstract class Adhesion {}
```

 [Structuraux] Visiteur

- ➊ Problématique : On souhaite réaliser des opérations sur les éléments d'un objet sans pour autant connaître à l'avance le résultat à obtenir.
- ➋ Solution : On utilise un objet tiers (visiteur) qui sera capable d'obtenir le résultat souhaité à partir des données.

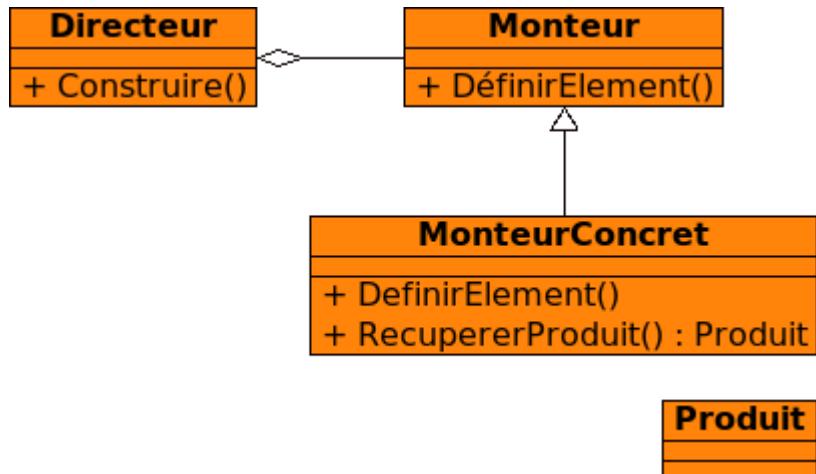
[Structuraux] Visiteur, diagramme



[Créateur] Monteur

- ➊ Problématique : Bien que la façon d'initialiser l'objet soit la même, il doit être possible d'obtenir différents résultats.
- ➋ Solution : Utiliser un Monteur qui implémente les étapes de créations.

[Créateur] Monteur, diagramme



[Créateur] Monteur, exemple

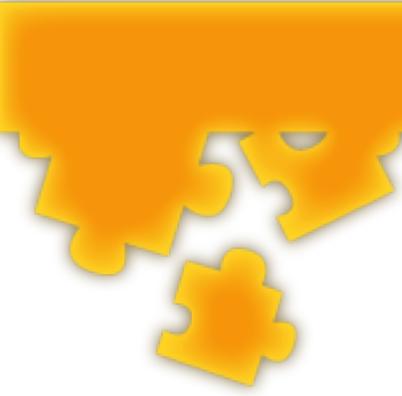
```
class DocumentBuilder {  
    private $_document = false;  
    function getDocument (){  
        if ($this->_document === false){  
            $this->_document = new TextDocument ();  
        }  
        return $this->_document;  
    }  
    abstract public function doTitre ();  
    abstract public function doParagraphe ();  
}
```

```
class DocumentBuilderFactory {  
    function getDocumentBuilder (){  
        switch (Config::getInstance ()->documentType){  
            case 'HTML' : return new HTMLBuilder ();  
            case 'Wiki' : return new WikiBuilder ();  
        }  
    }  
}
```

[Créateur] Monteur, exemple 2

```
class Document {  
    private $_content = " ";  
    function __construct ($baseContent = ""){  
        $this->_content = $baseContent;  
    }  
    function getContent (){  
        return $this->_content;  
    }  
    function add ($text){  
        $this->_content .= $text;  
    }  
}
```

[Créateur] Monteur, exemple 3



```
class HTMLBuilder extends DocumentBuilder {  
    function doTitre ($texte, $niveau){  
        $this->getDocument ()->add ("<h$niveau> $texte </h$niveau>");  
    }  
    function doParagraphe ($texte){  
        $this->getDocument ()->add ("<p>$texte</p>");  
    }  
}  
  
class WikiBuilder extends DocumentBuilder {  
    function doTitre ($texte, $niveau){  
        $this->getDocument ()->add (str_repeat ('!', $niveau).$texte."\n\r");  
    }  
    function doParagraphe ($texte){  
        $this->getDocument ()->add ("\n\r".$texte."\n\r");  
    }  
}
```

[Créateur] Monteur, exemple 4

```
class Devis {  
    private $_client = false;  
    private $_produits = array ();  
  
    //...  
  
    function getDocument ($documentBuilder){  
        $documentBuilder->doTitre ("Client");  
        $documentBuilder->doParagraphe ($this->_client->toString ());  
        foreach ($this->_produits as $produit){  
            documentBuilder->doTitre ($produit->getNom ());  
            documentBuilder->doParagraphe ($produit->getDescription ());  
        }  
    }  
}
```

[Créateur] Monteur, exemple code client

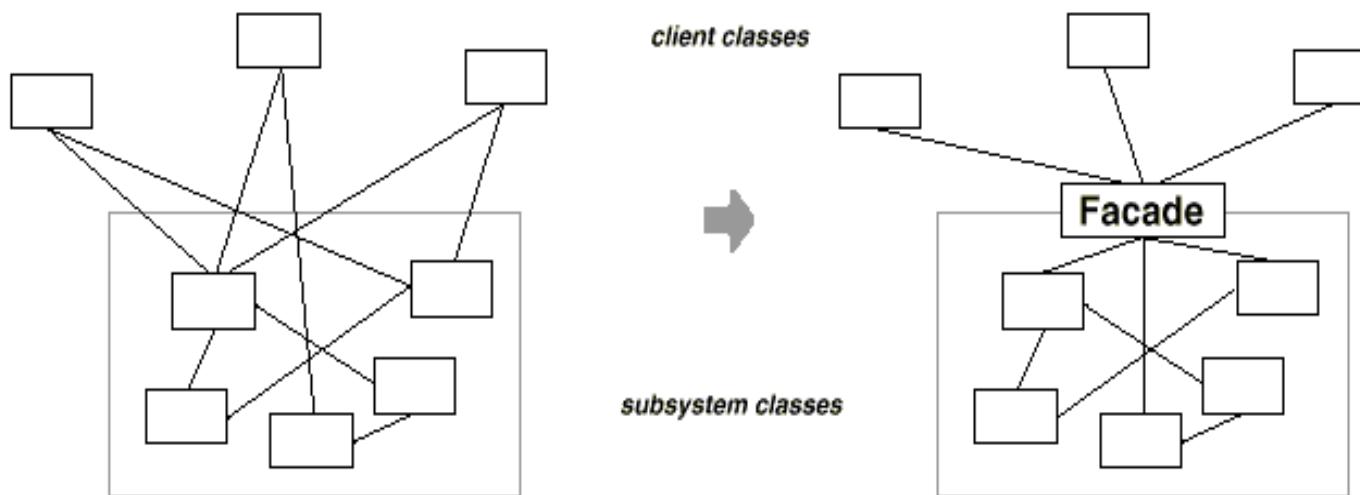
```
//...
function doDevis (){
    $devis = new Devis ();
    $devis->setClient (ClientSession::get ());
    foreach (HttpRequest::getProduit () as $id){
        $devis->addProduit ($id);
    }

    return new DownloadResponse ($devis->getDocument
        (DocumentBuilderFactory::getDocumentBuilder ()));
}
```

[Structuraux] Façade

- ⚙️ Problématique : Comment masquer la complexité d'une API qui réponds à un besoin simple.
- ⚙️ Solution : Un objet qui s'occupe de masquer la complexité des objets sous jacents.

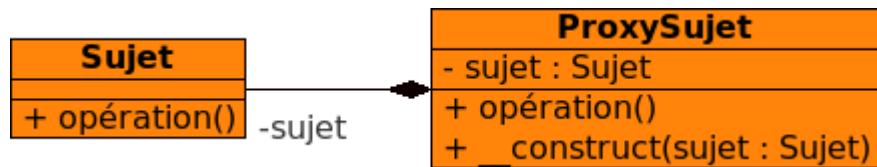
[Structuraux] Façade, Schéma



[Structuraux] Proxy

- ➊ Problématique : Donner l'accès à un objet sans avoir à le transmettre.
- ➋ Solution : Utiliser un Proxy qui va agréger l'objet et utiliser l'objet source pour réaliser les opérations.

[Structuraux] Proxy, diagramme



[Structuraux] Proxy, exemple

```
class DbConnection {  
    abstract function doSql ($sql);  
}  
  
class ProxyDbConnection {  
    private $_connection = false;  
  
    public function __construct ($dbConnection){  
        $this->_connection = $dbConnection;  
    }  
  
    public function doSql ($sql){  
        LogFactory::get ()->log ($sql);  
        $this->_connection->doSql ($sql);  
    }  
}
```

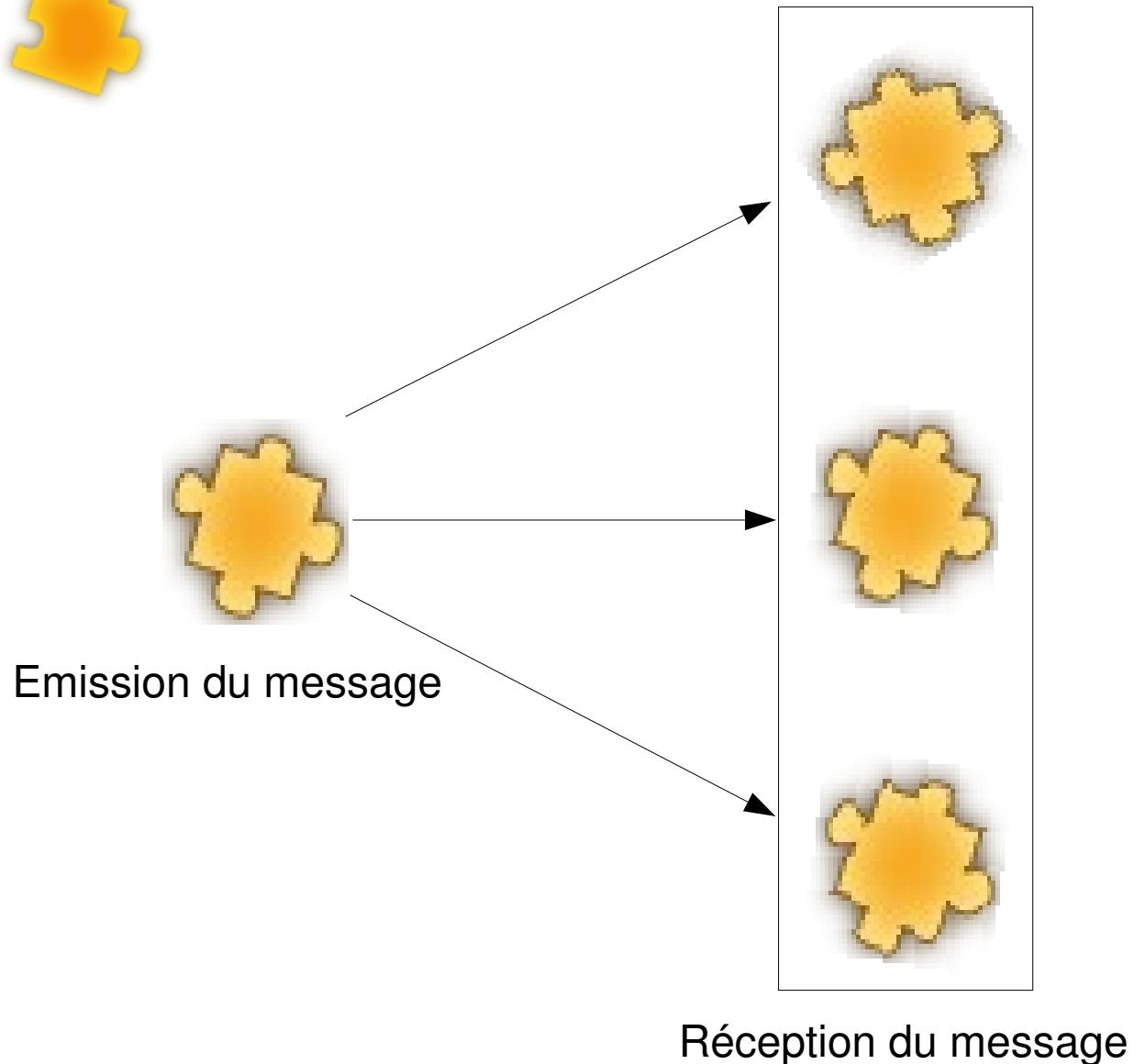
[Structuraux] Proxy, exemple

```
class DBFactory {  
  
    static function create ($dataSourceld){  
  
        switch (self::__getDriver ($dataSourceld)){  
  
            case self::__MySQL :  
  
                require_once ('DBMySQL.class.php');  
  
                $ct = new DBMySQL ($dataSourceld);  
  
                break;  
  
                //...  
  
        }  
  
        if (Config::__instance ()->logSQL){  
  
            return new ProxyDBConnection ($ct);  
  
        }  
  
    }  
  
}
```

[Comportement] Chaîne de responsabilité

- ➊ Problématique : Plusieurs objets sont capables de traiter la demande, mais on ne sait pas lequel.
- ➋ Solution : On va passer successivement l'information à la collection d'objets.

[Comportement] Chaîne de responsabilité



Au delà du GoF

❖ J2EE

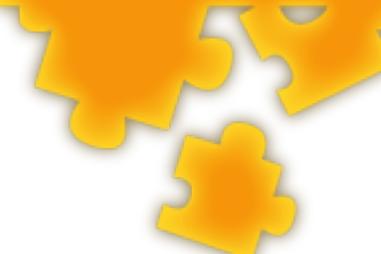
❖ Patterns of Enterprise Application
Architecture (Martin Fowler)

Au delà du GoF

- ❖ Active Record
- ❖ DAO / Data Mapper
- ❖ Lazy Load
- ❖ Front Controller
- ❖ MVC
- ❖ Query Object

Active record

- ➊ Problématique : Le modèle de données s'approche du modèle objet.
- ➋ Solution : Un objet qui contient l'ensemble



Active record, diagramme

```
<?php
```

```
//Exemple basique en utilisant le ZendFramework
```

```
$GoF = new Livre();
```

```
$GoF->setTitre('GoF');
```

```
$GoF->setAuteur('GoF');
```

```
$GoF->setEditeur ('...');
```

```
$GoF->save();
```

```
?>
```

Livre	
-	titre : int
-	auteur : int
-	editeur : int
+	insert()
+	update()
+	delete()
+	autreOperation()

DAO

- ➊ Problématique : Accéder aux données sans connaître la logique de stockage ou de représentation.
- ➋ Solution : Un objet qui encapsule la logique de récupération / mise à jour de données.

DAO, diagramme

Livre
- titre : int
- auteur : int
- editeur : int
+ autreOperation()

DAOLivre
+ insert(\$obj : Livre)
+ update(obj : Livre)
+ delete()

```
//Exemple basique en utilisant le framework Copix
foreach (CopixDaoFactory::create ('Livre')-
    >findAll () as $record){

    echo $record->titre;

}
```

Lazy Load

- ⚙️ Problématique : Ne pas surcharger l'application de traitements pour la récupération de données qui ne sont pas toujours nécessaires
- ⚙️ Solution : Un objet qui ne contient pas l'ensemble des données mais qui sait comment les récupérer.

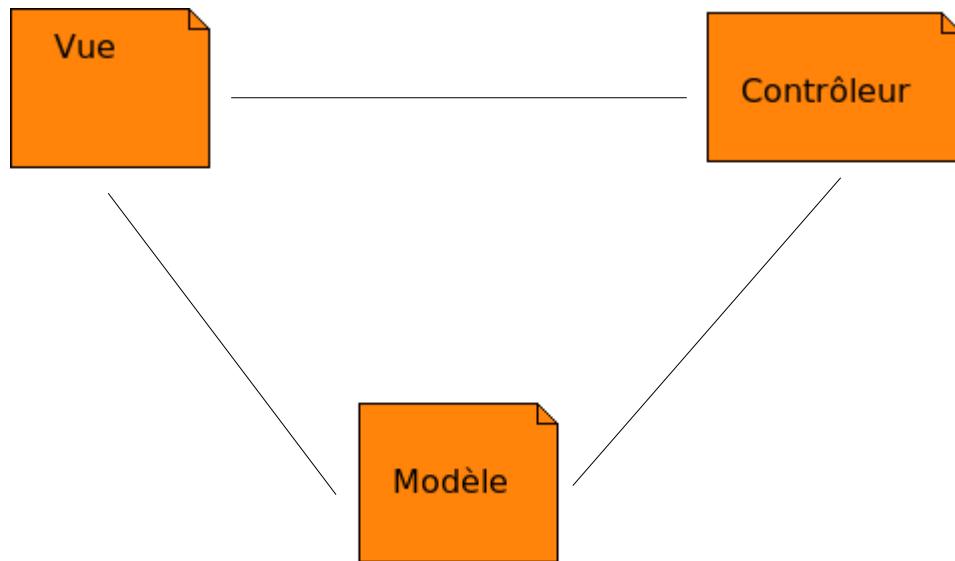
Front controller

- ➊ Problématique : Pouvoir réaliser des opérations diverses relatives à l'ensemble des pages d'un site avant / après leur traitement.
- ➋ Solution : Un objet qui intercepte l'ensemble des requêtes de l'application, objet pouvant être décoré.

MVC

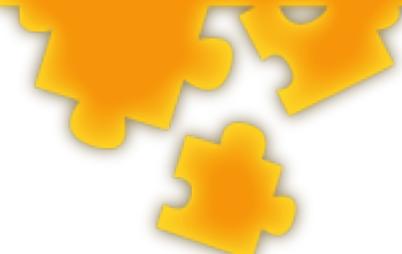
- ➊ Problématique : Comment organiser les interactions utilisateurs dans l'application ?
- ➋ Solution : Définition de 3 rôles : Modèle, Vue, Contrôleur

MVC, Schéma

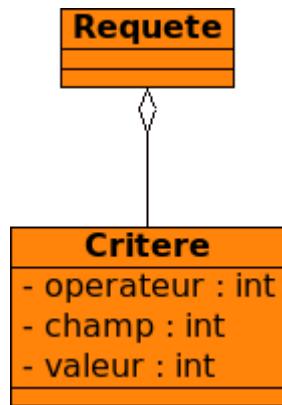


Query Object

- ➊ Problématique : Pouvoir manipuler les données de l'application depuis sa représentation objet.
- ➋ Solution : Un objet qui représente une requête (modèle interprète)



Query Object



```
<?php
//Exemple avec ezComponents
require 'ezc-setup.php';
$db = ezcDbFactory::create( 'mysql://root@localhost/geolocation' );
$sq = $db->createSelectQuery();
$stmt = $sq->select( 'name', 'country', 'lat', 'lon' )
    ->from( 'city' )
    ->where(
        $sq->expr->like(
            'normalized_name', $sq->bindValue( 'sofia%' )
        )
    )
    ->orderBy( 'country', 'name' )
    ->prepare();
$stmt->execute();
foreach ( $stmt as $entry )
{
    list( $name, $country, $lat, $lon ) = $entry;
    printf( '%s, %s is @ %.2f%.2f  
',
        $name, $country,
        abs( $lat ), ( $lat > 0 ? "N" :"S" ),
        abs( $lon ), ( $lon > 0 ? "E" :"W" ) );
}
?>
```

Au dela du GoF 2

- ➊ Page Controller
- ➋ Plugin
- ➌ Registry
- ➍ Template View
- ➎ 2 step View

Page controller

- ❖ Problématique : Ou positionner la cinématique, comment décider à partir d'une URL de l'action à réaliser ?
- ❖ Solution : Un objet qui prend en charge la récupération des données de l'URL, qui décide des éléments du modèle à utiliser ainsi que de la vue à utiliser.

Page controller



Vue



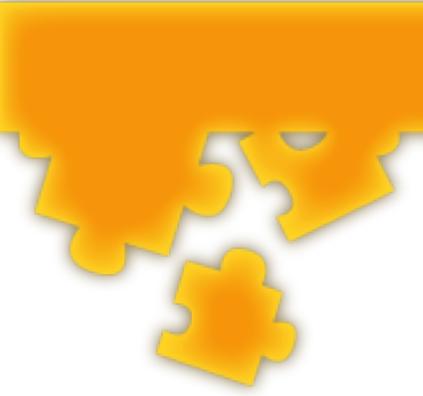
Contrôleur
de Page



Modèle

Plugin

- ❖ Problématique : Comment embarquer des fonctionnalités optionnelles sans alourdir l'application générale ?
- ❖ Solution : Lier les objets au moment de la configuration et non au moment de l'écriture du code.



Registre

- ➊ Problématique : Comment mettre des données à disposition sans devoir transmettre une multitude de paramètres ou sans passer par des variables globales ?
- ➋ Solution : Utiliser un objet connu de tous capable de conserver l'état d'autres objets

Template View

- ➊ Problématique : Comment obtenir une vue HTML avec les données de l'application ?
- ➋ Solution : Utiliser des fichiers intermédiaires qui contiennent la structure de la page ainsi que la logique de présentation.

Critiques

- ✿ Apporte des solutions peu efficientes
- ✿ Standardisation des « best practices ».
- ✿ En pratique, duplication de code inutile là où une solution spécifique bien pensée aurait put être meilleure qu'un pattern « faisant l'affaire ».

Anti Patterns



Catalogue des choses à ne pas faire

- (a) **Copier/Coller**
- (b) **Magic numbers**
- (c) **Singletonite**
- (d) **Code en dur**
- (e) **Masquage d'erreur**
- (f) **Coulée de lave (maintenance d'un mauvais code dans le temps)**
- (g) **Utilisation des types au lieu d'interfaces**
- (h) **Confiance aveugle**
- (i) **Sur-Abstraction**
- (j) « **Super objet** » (trop de responsabilités)
- (k) ...

Références

- ✿ Design Patterns. Elements of Reusable Object-Oriented Software.
- ✿ Design patterns Tête la première
- ✿ Pattern of Enterprise Application Architecture
- ✿ php|architect's Guide to PHP Design Pattern
- ✿ <http://www.phppatterns.com/>
- ✿ Wikipedia

Un design pattern pour résoudre un problème et non pas l'inverse.

Questions ?