

DÉVELOPPEMENT PHP NIVEAU 2

SQLI INSTITUT

EXPERIENCED
BY



Présenté par
Olivier LONZI

Chapitres

1

Environnements et outils de travail

2

Rappel sur la Programmation Orientée Objet en PHP

3

Réflexion avancée sur les bonnes pratiques

4

Maîtrise des tests avec PHPUnit

5

Introduction à la qualité et aux métriques de code

6

Collaboration des équipes

7

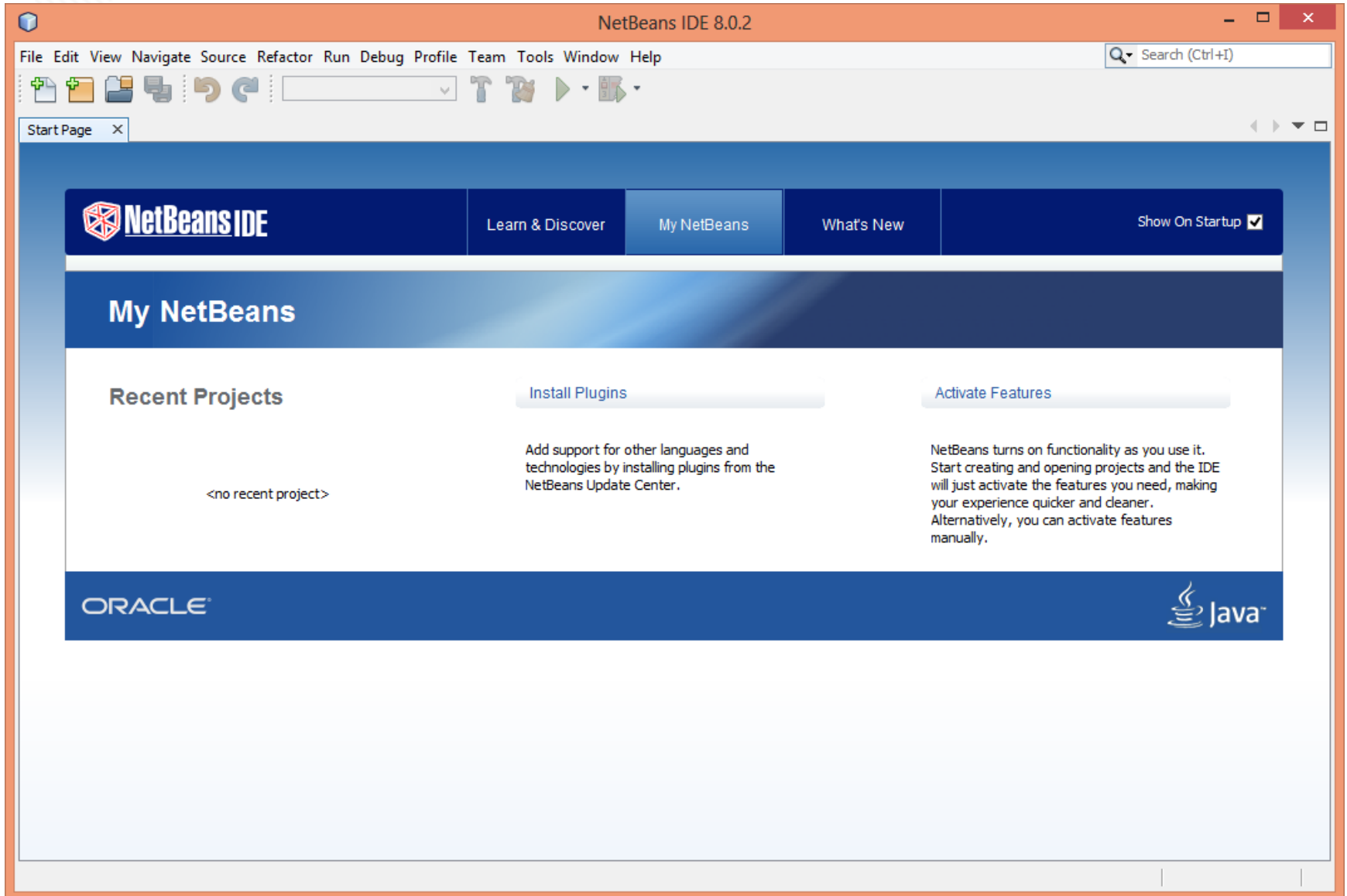
Les Frameworks, comment ne pas ré-inventer la roue ?

Environnements et outils de travail

- NetBeans
-
- Netbeans est un EDI (Environnement de Développement Intégré)
 - Souvent considéré comme le concurrent direct d'Eclipse.
 - Il est Opensource depuis l'an 2000
 - Originellement prévu pour Java, il permet de développer avec PHP/HTML via des plugins officiels.
 - Multiplateforme et Multilingue
 - Téléchargement de Netbeans
 - <https://netbeans.org/downloads/index.html>
 - Dans le tableau, choisir la version « HTML5 & PHP »

Environnements et outils de travail

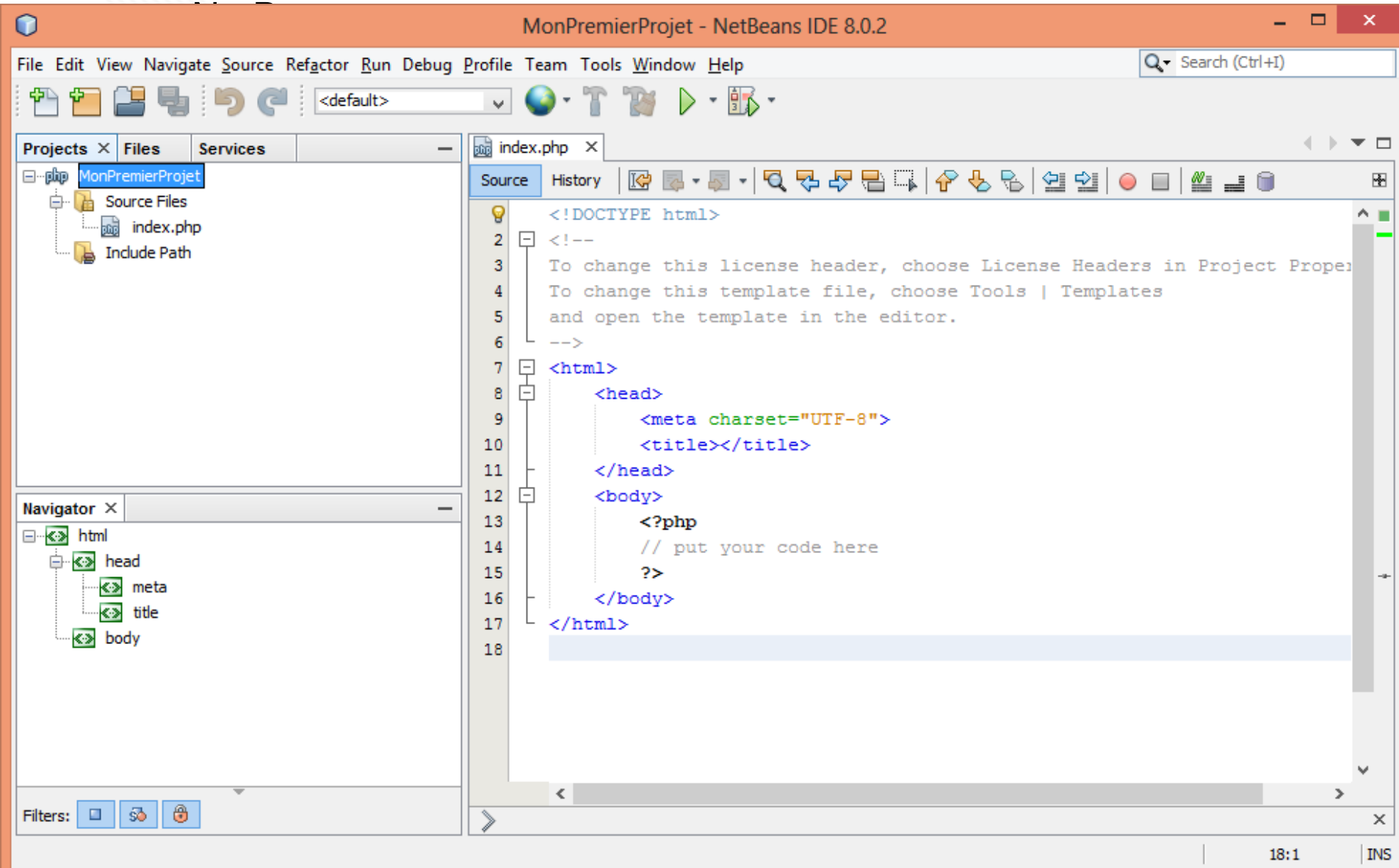
- NetBeans



Environnements et outils de travail

- NetBeans
-
- Premier Projet
 - File > New Project ...
 - PHP > PHP Application
 - **Project Name** : MonPremierProjet
 - **Source folder** : C:\formation\MonPremierProjet\
 - **PHP Version** : <Version utilisée en production>
 - **Encoding** : UTF-8
 - Valider avec « **Next** > »
 - **Run As** : Local Web Site
 - **Project URL** : <Url Http Locale>
 - Valider avec « **Next** > »
 - Choisir un framework au besoin
 - Valider avec « **Finish** »

Environnements et outils de travail



Environnements et outils de travail

- NetBeans
-
- Configurer **xDebug**
 - Installer la dernière version
 - <http://xdebug.org/download.php>
 - Configurer NetBeans
 - Tools > Options > PHP > Debugging
 - **Cocher** « Show Debugger Console »
 - Configurer PHP dans le php.ini
 - Utiliser « <http://xdebug.org/wizard.php> » pour configurer l'accès à la DLL
 - !! Utiliser les guillemets**
 - **output_buffering** = off
 - **xdebug.idekey** = "netbeans-xdebug"
 - **xdebug.remote_enable** = On
 - **xdebug.remote_host** = localhost
 - **xdebug.remote_port** = 9000
 - **xdebug.remote_handler** = dbgp

Environnements et outils de travail

- Configurer **PHPUnit**

- Télécharger dans « C:\Applications\phar\ »:

<https://phar.phpunit.de/phpunit.phar>

<https://phar.phpunit.de/phpunit-skelgen.phar>

- Configurer dans « C:\Applications\bin\ »

phpunit.bat : @php.exe "%~dp0../phar/**phpunit.phar**" %*

phpunit-skelgen.bat : @php.exe "%~dp0../phar/**phpunit-skelgen.phar**" %*

- Configurer NetBeans

- **Tools > Options > PHP**
- Onglet « **Frameworks & Tools** »
- Sélectionner PHPUnit dans la liste

- **PHPUnit Script** : C:\Applications\phar\phpunit.phar

- **Skelton Generator** : C:\Applications\phar\phpunit-skelgen.phar

Environnements et outils de travail

– Télécharger dans « C:\Applications\phar\ »:

- **Code_Sniffer**

https://squizlabs.github.io/PHP_CodeSniffer/phpcs.phar

https://squizlabs.github.io/PHP_CodeSniffer/phpcbf.phar

- **PHPMD**

<http://static.phpmd.org/php/latest/phpmd.phar>

- **PDepend**

<http://static.pdepend.org/php/latest/pdepend.phar>

- **PHPCPD**

<https://phar.phpunit.de/phpcpd.phar>

– Configurer dans « C:\Applications\bin\ »

- **phpcs.bat** : @php.exe "%~dp0../phar/**phpcs**.phar" %*

- **phpcbf.bat** : @php.exe "%~dp0../phar/**phpcbf**.phar" %*

- **phpmd.bat** : @php.exe "%~dp0../phar/**phpmd**.phar" %*

- **pdepend.bat** : @php.exe "%~dp0../phar/**pdepend**.phar" %*

- **phpcpd.bat** : @php.exe "%~dp0../phar/**phpcpd**.phar" %*

Environnements et outils de travail

- Configurer Netbeans
 - **Tools > Plugins**
 - Rechercher « phpcsmd » dans l'onglet « Available Plugins », puis l'installer.
 - Si non présent le télécharger :
« <http://plugins.netbeans.org/plugin/42434/phpcsmd> »
- Au besoin redémarrer NetBeans
 - **Tools > Options > PHP**
 - Onglet « Code Analysis »
 - » **Code Sniffer** : C:\Applications\bin\phpcs.bat
 - » **Mess Detector** : C:\Applications\bin\phpmd.bat
 - Onglet « PHPCSMD » :
 - » **PHPCS** : C:\Applications\bin\phpcs.bat
 - » **PHPMD** : C:\Applications\bin\phpmd.bat
 - » **PHPCPD** : C:\Applications\bin\phpcpd.bat
 - » **PDepend** : C:\Applications\bin\pdepend.bat

Environnements et outils de travail

- Configurer les PATH vers « php.exe »

- Trouver le programme

- Wamp :

C:\wamp\bin\php\php<version>\php.exe

- Easy PHP :

C:\Program Files (x86)\EasyPHP-DevServer-<version> \binaries\php\php_runningversion\php.exe

- Netbeans

- **Tools > Options > PHP > General**

- PHP 5 Interpreter : *<chemin vers php.exe>*

- Configurer les fichiers Batch (.bat)

- Modifier l'environnement système pour lui donner connaissance de php.exe

- Modifier chaque scripts (.bat), vus précédemment en remplaçant « @php » par « @*<chemin vers php.exe>* »

PHPDoc : Documenter son code

- + La documentation de code peut sembler ennuyeuse, souvent mise de côté par manque de temps.
- + Pourtant, documenter son code peut être source de gain de temps, même dans le cas d'un unique développeur :
 - Les IDE, comme Netbeans, reconnaissent la syntaxe PHP Doc, et adapte l'auto-complétion en fonction de ce que vous avez écrits.
 - Projet : <http://phpdoc.org/docs/latest/index.html>
- + Exemple
 - ```
/**
 * @return string|null Retourne le label, ou la valeur nulle
 */
function getLabel() {
 // Le traitement
}
```

# Chapitres

Environnements et outils de travail

2

**Rappel sur la Programmation Orientée Objet en PHP**

Réflexion avancée sur les bonnes pratiques

Maîtrise des tests avec PHPUnit

5

Introduction à la qualité et aux métriques de code

Collaboration des équipes

Les Frameworks, comment ne pas ré-inventer la roue ?

# Programmation Orientée Objet

- Rappel

+ *Classe « Personnes »*

+ *Attributs protégés*

- *prenom*
- *nom*

+ *Méthodes publiques*

- *modifier()*
- *afficher()*

Personnes

# \$prenom  
# \$nom

+ modifier()  
+ afficher()

# Programmation Orientée Objet

- Rappel

```
+ class Personnes {
 protected $prenom = null;
 protected $nom = null;

 function modifier($prenom, $nom) {
 $this->prenom = $prenom;
 $this->nom = $nom;
 }
 function afficher() {
 return $this->prenom.' '.$this->nom;
 }
}
```

## Personnes

# \$prenom

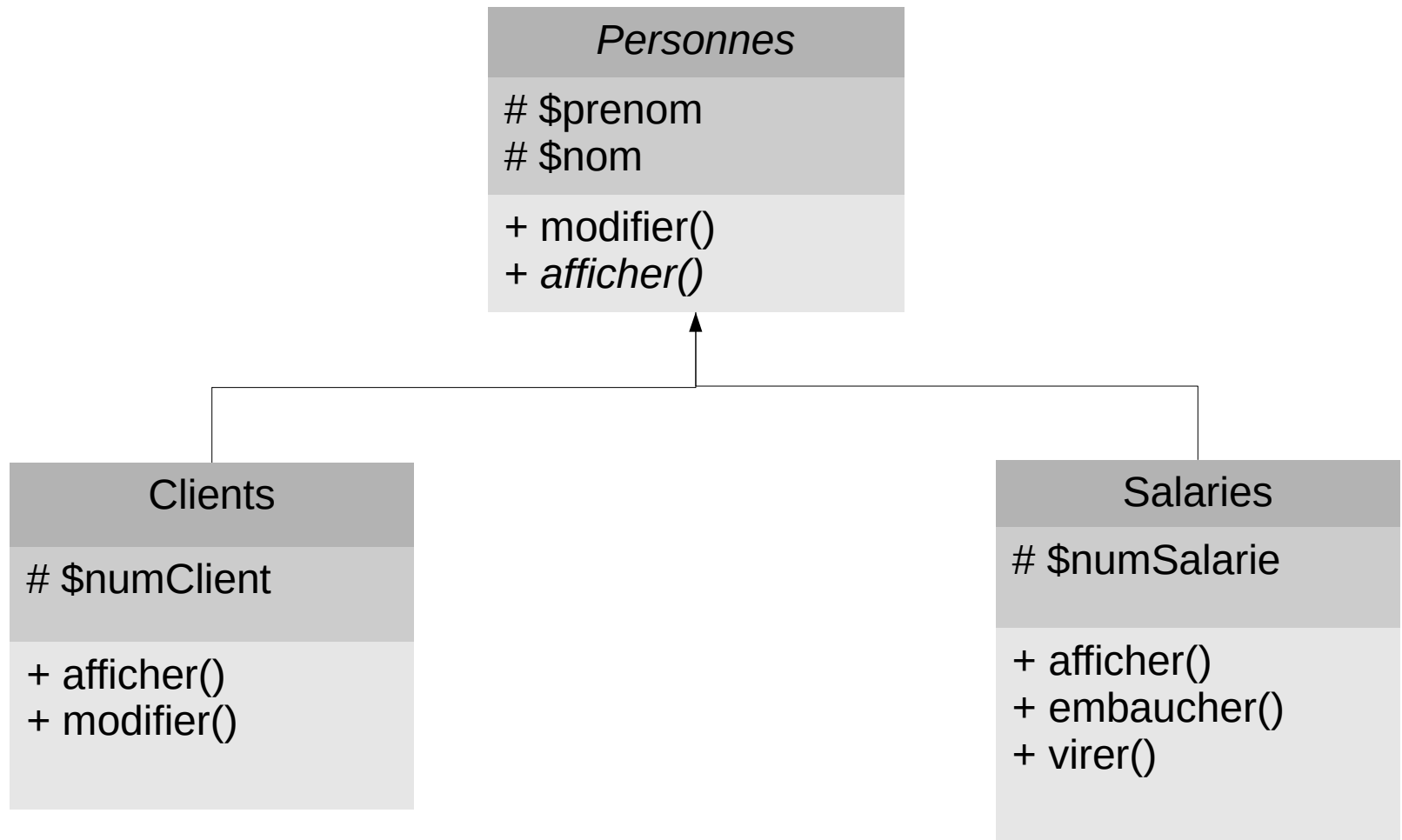
# \$nom

+ modifier()

+ afficher()

# Programmation Orientée Objet

- L'Héritage





# Programmation Orientée Objet

- L'Héritage
- 

+ *abstract class* **Personnes** {

...

}

+ *class* **Clients** *extends* **Personnes** {

...

}

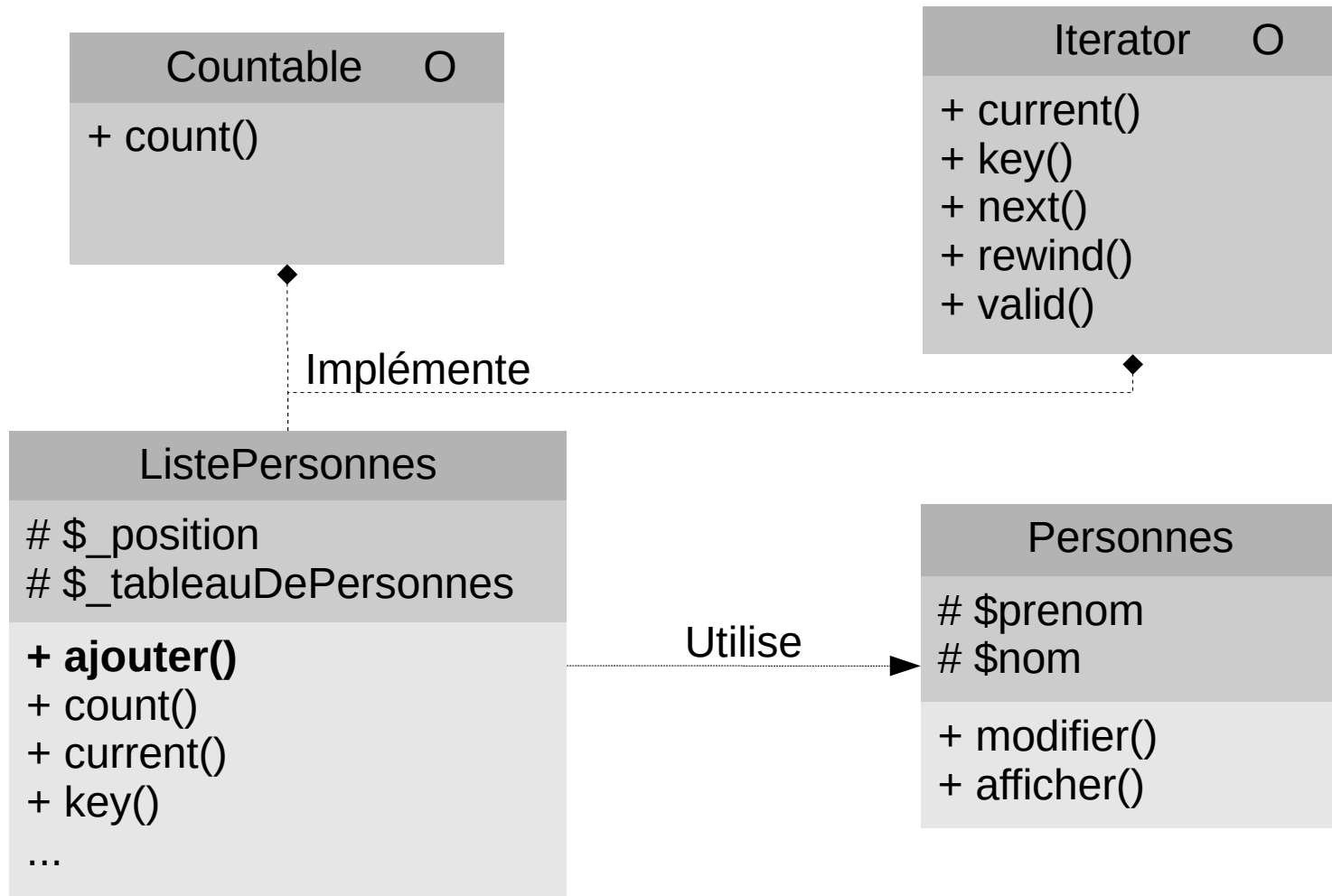
+ *class* **Salaries** *extends* **Personnes** {

...

}

# Programmation Orientée Objet

- L'implémentation d'interfaces



# Programmation Orientée Objet

- L'implémentation d'interfaces

```
+ interface Countable {
 function count();
}

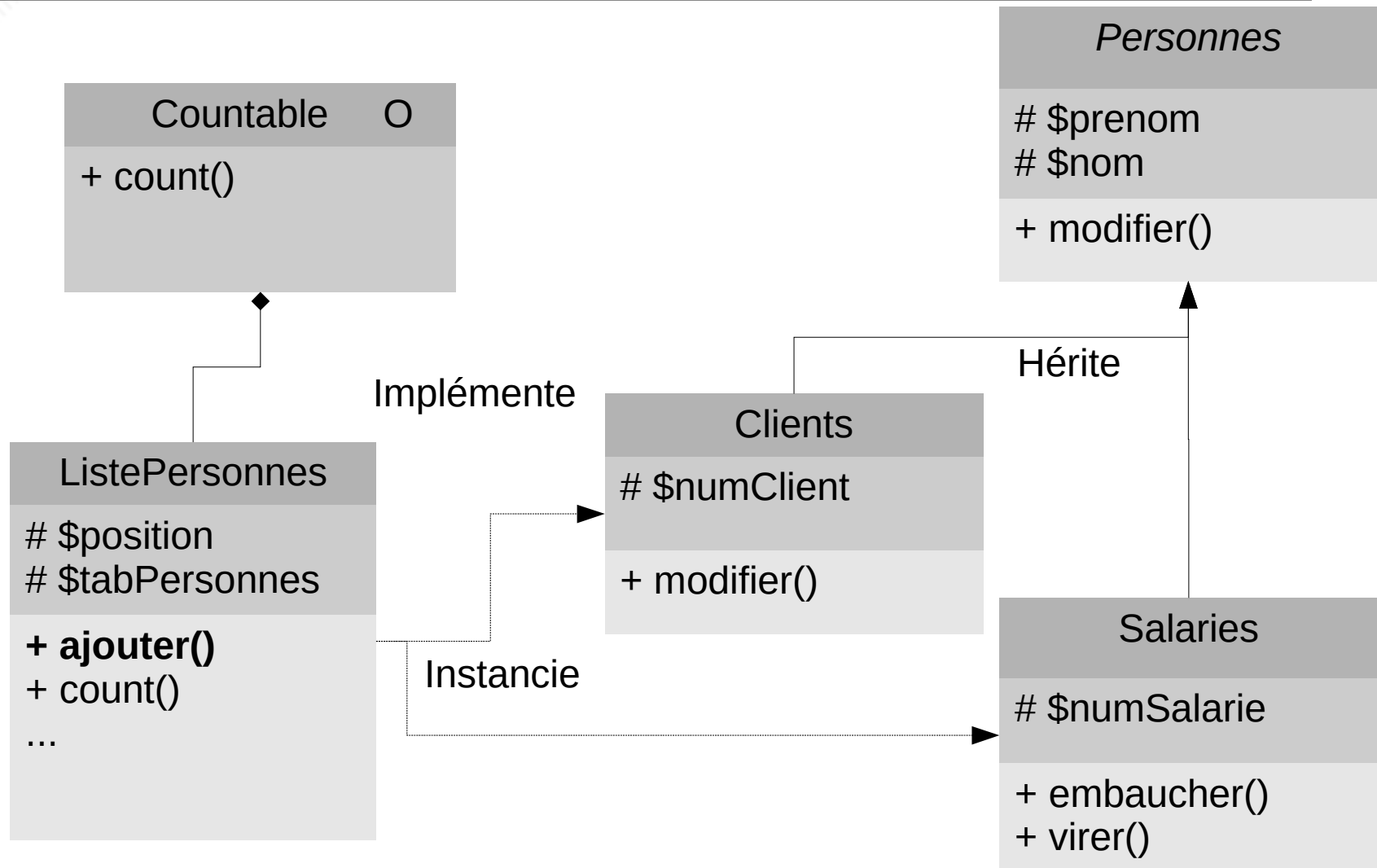
+ class ListePersonnes implements Countable, Iterator {
 protected $tabP = array();

 public function ajouter(Personnes $p) {
 $this->_tabP[] = $p;
 }
 public function count() {
 return count($this->_tabP);
 }
 ... // Implémenter les méthodes de l'interface Iterator
}

+ $liste = new ListePersonnes(); // Créer l'objet de liste
 $liste->ajouter(new Personnes()); // Ajouter une personne
 var_dump($liste->count()); // Affiche 1
 var_dump($liste instanceof Countable); // Retourne Vrai
```

# Programmation Orientée Objet

- L'implémentation d'interfaces complexes



# Programmation Orientée Objet

- L'implémentation d'interfaces complexes

```
+ interface Countable ...;
 abstract class Personnes ...;

+ class ListePersonnes implements Countable {
 protected $tabP = array();

 public function ajouter(Personnes $p) {
 $this->tabP[] = $p;
 }
 ... // Implémenter les méthodes de l'interface Countable
}

+ $liste = new ListePersonnes(); // Créer l'objet de liste
 $liste->ajouter(new Salaries()); // Ajouter un salarié
 $liste->ajouter(new Salaries()); // Ajouter un salarié
 $liste->ajouter(new Clients()); // Ajouter un client
 $liste->ajouter(new Clients()); // Ajouter un client
 var_dump($liste->count()); // Affiche 4
```

# Chapitres

Environnements et outils de travail

Rappel sur la Programmation Orientée Objet en PHP

3

**Réflexion avancée sur les bonnes pratiques**

Maîtrise des tests avec PHPUnit

Introduction à la qualité et aux métriques de code

Collaboration des équipes

Les Frameworks, comment ne pas ré-inventer la roue ?

# Les bonnes pratiques

---

- + *Aujourd'hui, une majorité des développeurs programme avec des langages orientés objet.*
- + *Après avoir passé de nombreuses années à maintenir et développer du code, on se rend malheureusement compte que les **principes du développement objet** sont souvent :*
  - *Ignorés,*
  - *Mal compris,*
  - *Mal utilisés.*
- + *Cela rend assez souvent la **maintenance** des logiciels au mieux **malaisée**, au pire **impossible**.*

# Les bonnes pratiques

- SOLID
- 
- SOLID est l'acronyme de cinq principes de base
    - Single Responsibility Principle
    - Open/Closed Principle
    - Liskov Substitution Principle
    - Interface Segregation Principle
    - Dependency Inversion Principle
  - On verra dans ce chapitre que ce sont avant tout des principes de bon sens.
  - Aucun d'eux ne nécessite une connaissance approfondie d'un langage donné.
  - Ces principes, lorsqu'ils sont compris et suivis, permettent d'améliorer la **cohésion**, de diminuer le **couplage**, et de favoriser l'**encapsulation** d'un programme orienté objet.



# Les bonnes pratiques

- SOLID
- 

- **Cohésion**

- La cohésion traduit à quel point les **pièces d'un seul composant** sont en **relation** les unes avec les autres.
- Un module est cohésif, lorsqu'au **haut niveau** d'abstraction il ne fait qu'une seule **tâche précise**.
- Plus un module est centré sur **un seul but**, plus il est **cohésif**.

# Les bonnes pratiques

- SOLID
- 

- **Couplage**

- Le couplage est une métrique qui mesure l'**interconnexion** des modules.
- Deux modules sont dit **couplés** si une **modification d'un** de ces modules demande une **modification dans l'autre**.

# Les bonnes pratiques

- SOLID

---

- **Encapsulation**

- L'idée derrière l'encapsulation est d'intégrer à un objet **tous les éléments nécessaires à son fonctionnement**, que ce soit des fonctions ou des données.
- Le corollaire est qu'un objet devrait **masquer** la **cuisine interne** de la classe :
  - pour exposer une **interface propre**,
  - pour que les développeurs puissent **manipuler** l'objet **sans** avoir à **connaître** le **fonctionnement interne** de l'objet.

# Les bonnes pratiques

- SOLID – Single Responsibility Principle
- 
- Responsabilité unique – **Single Responsibility Principle**
    - Le principe de responsabilité unique, réduit à sa plus simple expression, est qu'une classe donnée ne doit avoir qu'une seule responsabilité, et, par conséquent, qu'elle ne doit avoir qu'**une seule raison de changer**.
    - On peut aussi parler de **responsabilité humaine**, une seule personne devrait être responsable d'une classe, celle-ci devant connaître tous les tenants et aboutissants de sa classe lorsqu'il la modifie.
    - Les avantages de cette approche sont les suivants :
      - Diminution de la complexité du code,
      - Augmentation de la **lisibilité de la classe**,
      - **Meilleure encapsulation**, et **meilleure cohésion**, les responsabilités étant regroupées.

# Les bonnes pratiques

- SOLID – **O**pen/**C**losed Principle
- 

- Principe Ouvert/Fermé – Open/Closed Principle
  - Les entités logicielles (classes, modules, fonctions, etc...) devraient être **ouvertes pour l'extension**, mais **fermées à la modification**.
  - Elles sont « **ouvertes pour l'extension** » :
    - le **comportement** du module **peut être étendu**,
    - l'on peut **faire se comporter** ce module de façon nouvelle et **différente** si les exigences de l'application sont modifiées.
  - Elles sont « **Fermées à la modification** » :
    - le code source d'un tel module ne doit pas être modifié.  
**Personne** n'est **autorisé** à y apporter des **modifications**.
  - Les avantages de cette approche sont les suivants :
    - Plus de **flexibilité** pour les évolutions,
    - **Diminution du couplage**.

# Les bonnes pratiques

- SOLID – Liskov Substitution Principle
- 
- Substitution de Liskov – Liskov Substitution Principle
    - En cas d'héritage, les **sous-classes** doivent être **remplaçables** par leur **classe de base**.
    - Les sous-classes doivent pouvoir être substituées à leur classe de base sans altérer le comportement de cette dernière :
      - Autrement dit : une **méthode utilisant** la *classe de base* **doit** pouvoir **continuer de fonctionner correctement** si une *classe héritant* de la *classe de base* **lui est fournie**.
    - Les avantages de cette approche sont les suivants :
      - **Augmentation de l'encapsulation,**
      - **Diminution du couplage.**

# Les bonnes pratiques

- SOLID – Interface Segregation Principle
- 

- Séparation des Interfaces – Interface Segregation Principle
  - Les classes ne doivent pas avoir à dépendre d'une interface qu'ils n'utilisent pas.
  - Utiliser deux (ou plus) interfaces distinctes, plutôt qu'une seule interface qui oblige l'implémentation de fonctionnalités non utiles pour les classes qui en découlent.
  - Les avantages de cette approche sont les suivants :
    - **Diminution du couplage,**
    - Augmentation de la robustesse.

# Les bonnes pratiques

- SOLID – **D**ependency Inversion Principle
- 
- Inversion des dépendances – Dependency Inversion Principle
    - Les classes de **haut niveau ne doivent pas dépendre** des classes de **bas niveau**. *Les deux doivent dépendre d'abstractions.*
    - Les abstractions ne doivent pas dépendre des détails. *Les détails doivent dépendre des abstractions.*
    - Si on **change** le mode de **fonctionnement** de la base (*passage de Oracle à MySQL*), les **classes métiers** ne doivent **pas être impactées**.
    - Les avantages de cette approche sont les suivants :
      - **Meilleure encapsulation,**
      - **Diminution drastique du couplage.**



# Les bonnes pratiques

- SOLID

- 
- SOLID ou non ?
    - Nous l'avons vu, la conception objet et le respect de principes simples permettent de rendre un logiciel plus souple, plus évolutif, moins dépendant de son environnement et de ses évolutions.
    - Néanmoins, cela nécessite un niveau d'abstraction élevé qui n'est pas toujours utile.
  - YAGNI – You Ain't Gonna Need It
    - En effet, un autre principe tend à s'opposer à l'application systématique des principes SOLID « Vous n'en aurez probablement pas besoin ».
    - Pourquoi mettre en place tout un système de tolérance au changement dans un système statique ?
    - Notre logiciel sera effectivement extrêmement robuste envers des événements qui n'auront pas lieu, ou avec un impact minime.
    - L'investissement effectué s'avère peu productif, avec un retour sur investissement négatif.

# Les bonnes pratiques

- SOLID
- 

- KISS Principle
  - Keep it Simple, Stupid :
    - Laisse-le simple, stupide »
  - Keep it Stupidly Simple :
    - Laisse-le stupidement simple
  - Keep it Simple & Stupid :
    - Laisse-le simple et stupide
  - [http://fr.wikipedia.org/wiki/Principe\\_KISS](http://fr.wikipedia.org/wiki/Principe_KISS)
  - Il est utilisé comme principe de développement de logiciels, pour rappeler aux développeurs qu'un programme simple est plus facile à maintenir et à comprendre.
  - « *Pourquoi faire compliqué quand on peut faire simple ?* »

# Les Design patterns

---

- Patrons de conception (Design Patterns)
  - Les « Design Patterns » furent introduits en 1995 dans le livre « *Design Patterns Elements of Reusable Object Oriented Software* » du Gang of Four (La bande des quatre auteurs).
  - Les patterns répondent à des problèmes de conception de logiciels dans le cadre de la programmation par objets.
  - Ce sont des solutions connues et éprouvées dont la conception provient de l'expérience de programmeurs.

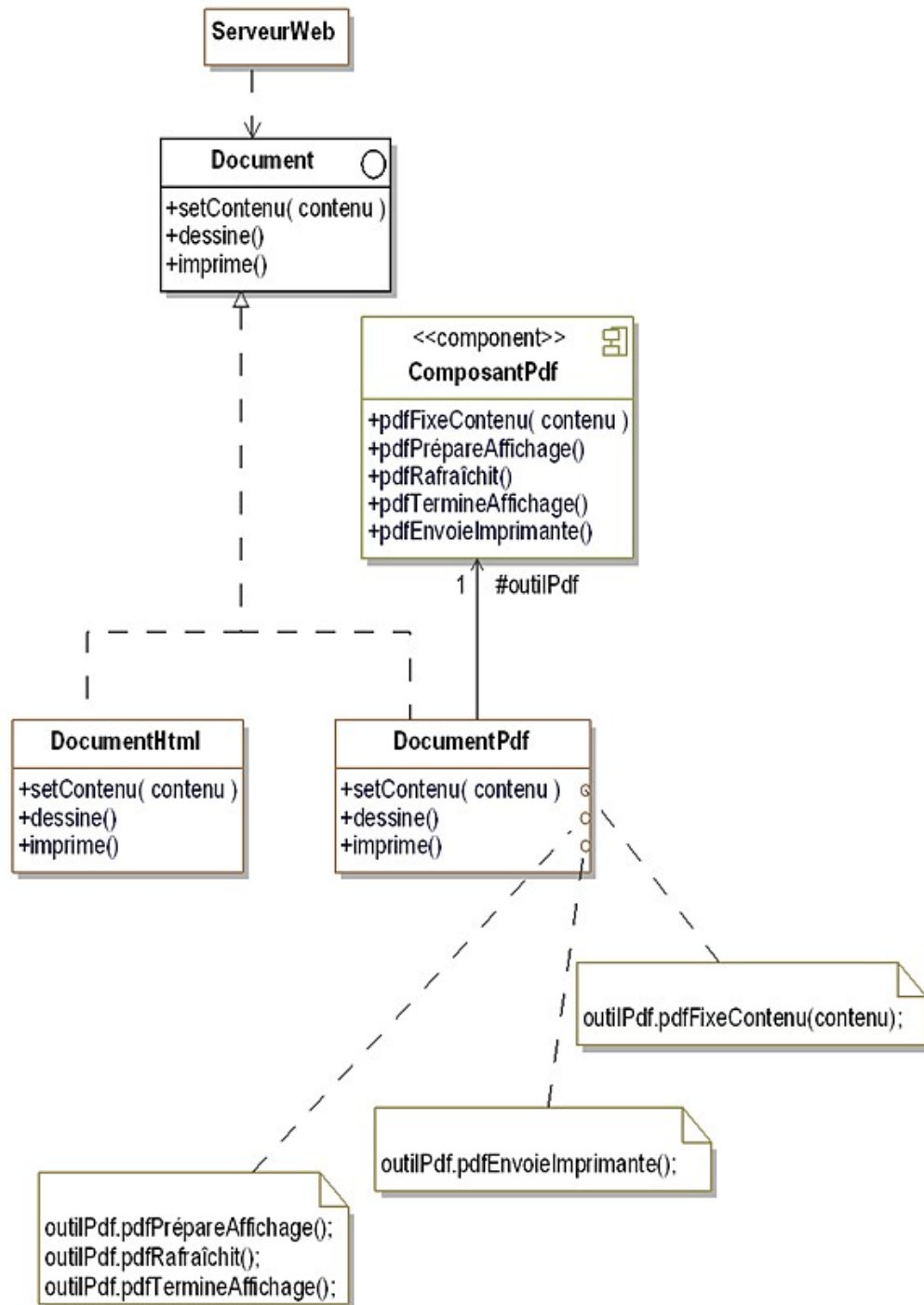
# Les Design patterns

- Patrons de structuration – Adapter
- 

- Adapter
  - Le but du pattern Adapter est de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent travailler ensemble.
  - Il s'agit de conférer à une classe existante une nouvelle interface pour répondre aux besoins de clients.

# Les Design patterns

- Patrons de structure



# Les Design patterns

- Patrons de construction – Adapter
- 

- **Quand ?**

- Quand vous récupérez une librairie externe dont les méthodes ne sont pas celles prévues
- Quand vous souhaitez laisser le choix d'une librairie externe parmi plusieurs, mais voulez proposer une classe commune d'utilisation

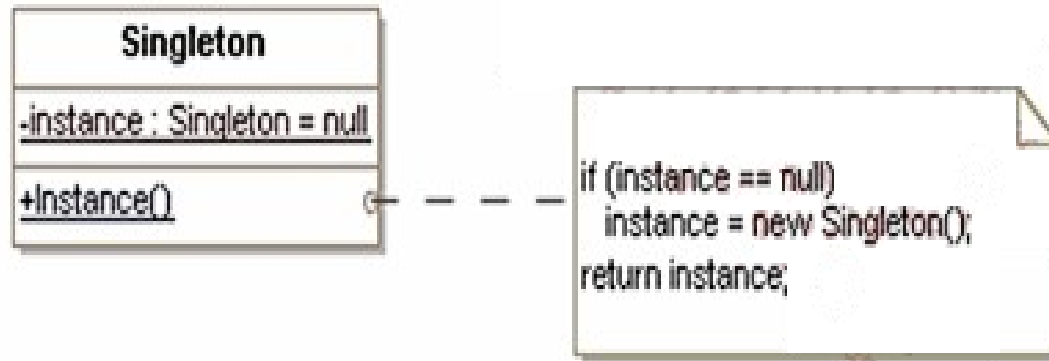
- **Pourquoi ?**

- Pour intégrer dans un système un objet dont l'interface ne correspond pas à l'interface requise au sein de ce système
- Pour fournir des interfaces multiples à un objet lors de sa conception

# Les Design patterns

- Patrons de construction – Singleton

- Singleton
  - Le pattern Singleton a pour but d'assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode de classe unique retournant cette instance.
- Structure



# Les Design patterns

- Patrons de construction – Singleton
- 

- **Quand ?**

- Besoin d'un objet accessible à l'identique de n'importe où et n'importe quand
- Accéder à des informations communes aux projets

- **Pourquoi ?**

- Pour s'affranchir des variables globales
- Garantir la maîtrise des informations communes



# Exercice chapitre 3

- Exercices – Préambule
- 

- + Les exercices suivants ont pour objectifs :
  - Recréer le même projet que nous avons à la fin de la formation « PHP Niveau 1/chapitre 7 »
  - Mettre en place, au maximum, les bonnes pratiques de ce chapitre.
- + **Ressources :**
  - Dossier « *Exercices/chapitre03/0-init* »
    - Projet complet tel qu'à la fin du niveau 1
  - Dossier « *Exercices/chapitre03/1-application* »
    - Version élaguée pour débiter la série d'exercices

# Exercice chapitre 3

- Exercice 1 – Application

---

- + **Comprendre** le Singleton « Application »

- Ce sera le point d'entrée de toutes nos pages
  - Il initialise – entre autres choses – la session PHP
    - *function* \_\_construct()
  - Il va analyser l'url saisie, pour afficher la bonne page
    - *function* dispatch()

– **Exemple 1** : index.php?page=index → **index.phtml**

– **Exemple 2** : index.php?page=login → **login.phtml**

- + *Le fichier « index.php » utilisera simplement l'instance de « Application »*

- *\$app = Application::getInstance();*
- *\$app->dispatch();*

- + **Schématiser** et **créer** la classe « Application » avec UML

- + **Utiliser** les pages :

- *index* : Mot de bienvenue
- *login* : Formulaire d'identification (Non fonctionnel)

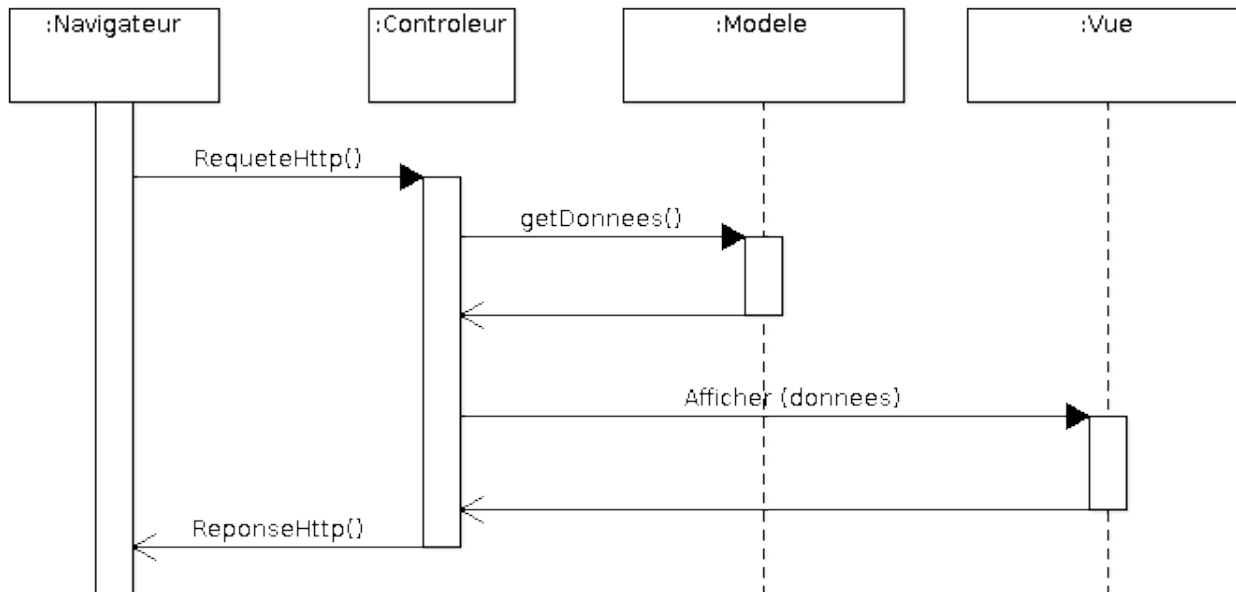
# Les Design patterns

- Patrons de conception – MVC
- 

- Le pattern composite MVC
  - La solution « MVC » (Model-View-Controller) prône la séparation suivante des composants d'une application :
    - **Model (modèle)** : il s'agit du noyau fonctionnel qui gère les données manipulées par l'application.
    - **View (vue)** : il s'agit des composants destinés à afficher les informations à l'utilisateur. Chaque vue est liée à un modèle. Un modèle peut être lié à plusieurs vues.
    - **Controller (contrôleur)** : un composant de type contrôleur reçoit les événements en provenance de l'utilisateur et les traduit en requêtes pour le modèle ou pour la vue. Chaque vue est associée à un contrôleur.

# Les Design patterns

- Patrons de conception – MVC



# Les Design patterns

- Patrons de conception – MVC
- 

- **Quand ?**

- Quand l'intégration de code PHP au milieu des balises HTML devient trop complexe
- Quand vous souhaitez travailler à plusieurs sur un projet, et faire en sorte que les intégrateurs graphistes ne touchent pas aux codes PHP

- **Pourquoi ?**

- Pour séparer :
  - L'accès aux données
  - Le traitement des données
  - L'affichage des données

# Exercice chapitre 3

- Exercice 2 – Vue

---

- + **Problématique & Solution :**

- Afin d'alléger le travail de notre classe Application, nous allons créer une classe « Vue », celle-ci aura pour utilité la gestion de nos vues.
- Le dispatcher pourra s'écrire plus simplement :  

```
$vue = new Vue($page);
$vue->afficher();
```

- + **Schématiser :**

- Avec un diagramme des classes
- Avec des diagrammes de séquence pour expliciter les interactions entre les classes.

- + **Créer** la classe et modifier le code.

- + Pour les plus rapides : Notre application devra pouvoir gérer la notion de layout « structure.phtml », un fichier HTML avec une structure commune à toutes les pages.

# Exercice chapitre 3

- Exercice 3 – Contrôleur

---

+ **Problématique :**

– Comment gérer le formulaire de login ?

1. dans « index.php » ?

2. dans le dispatcher ?

3. dans la vue « login.phtml » ?

4. dans un script PHP tiers ?

+ **Solution :**

– Les 3 premières propositions ne sont pas des solutions viables, en effet, la 1 et 2 sont trop génériques, et la vue (n°3) ne doit faire qu'afficher.

– Nous allons avoir besoin d'un « contrôleur » : *class **Login**Controller { ... }*

– Celui-ci sera appelé par le dispatcher, uniquement si il existe.

+ **Schématiser :**

– La/les nouvelle(s) classe(s)

– Des séquences représentatives de l'interaction entre les classes.

+ **Modifier** le code pour permettre l'identification de l'utilisateur « admin », qui devra être redirigé sur une page sécurisée, avec bouton de déconnexion.

# Exercice chapitre 3

- Exercice 4 : Organisation & Namespaces

---

- + **Problématique :**

- Le projet commence à disposer de plusieurs fichiers, et a vocation à en avoir de plus en plus, en particulier les contrôleurs.

- + **Organiser** les dossiers sous cette forme :

- public/
  - index.php
- src/
  - App/
    - Application.php
    - Vue.php
    - Controllers/
      - » LoginController.php
- templates/\*

- + **Utiliser** les espaces de noms (namespaces), pour « packager » les différentes classes, en fonction de leur position dans l'arbre.



# Exercice chapitre 3

- Exercice 5 – Plugins (1/2)

---

+ **Problématique :**

- Pour sécuriser l'accès aux pages, nous devons dans chaque initialisation des contrôleurs ajouter un test de droits d'accès.

+ **Solution 1 :**

- Créer une fonction commune, pour tester les droits, placée dans l'initialisation de tous les contrôleurs :

*\$this->testDroits(\_\_CLASS\_\_);*

- Cette solution implique l'écriture systématique du même morceau de code dans chaque initialisation.

+ **Solution 2 :**

- Mise en place d'une notion de Plugins,
- Ceux-ci pourraient être ajoutés ou supprimés de notre application à tout moment sans modifier les contrôleurs,
- Ils seraient appelés à des moments clés de notre application – entre autres avant que le dispatcher ne fasse son travail (Pour le cas de la gestion des droits).
- Ces Plugins ne seront pas limités à la gestion des droits.

# Exercice chapitre 3

- Exercice 5 – Plugins (2/2)

---

+ **interface** PluginsInterface {  
    public function **init**();  
    public function **beforeDispatch**(string \$page);  
    public function **afterDispatch**(\App\Vue \$vue);  
}

+ **Schématiser :**

- Les nouvelles classes & interfaces
- Les interactions entre les classes

+ **Créer** le Plugin de gestion des droits « PluginAcl.php »

- Si l'utilisateur est identifié, il peut afficher la page Extranet,
- Si l'utilisateur n'est pas reconnu, mais tente d'accéder à une page protégée, il est renvoyé vers une page d'erreur.

+ **Initialiser** le plugin dans l'initialisation de l'Application.

+ Les Plugins, pouvant être utilisés pour autre chose que la gestion des droits, **mettre en place** autant de Plugins que vous jugerez nécessaire.

# Les Design patterns

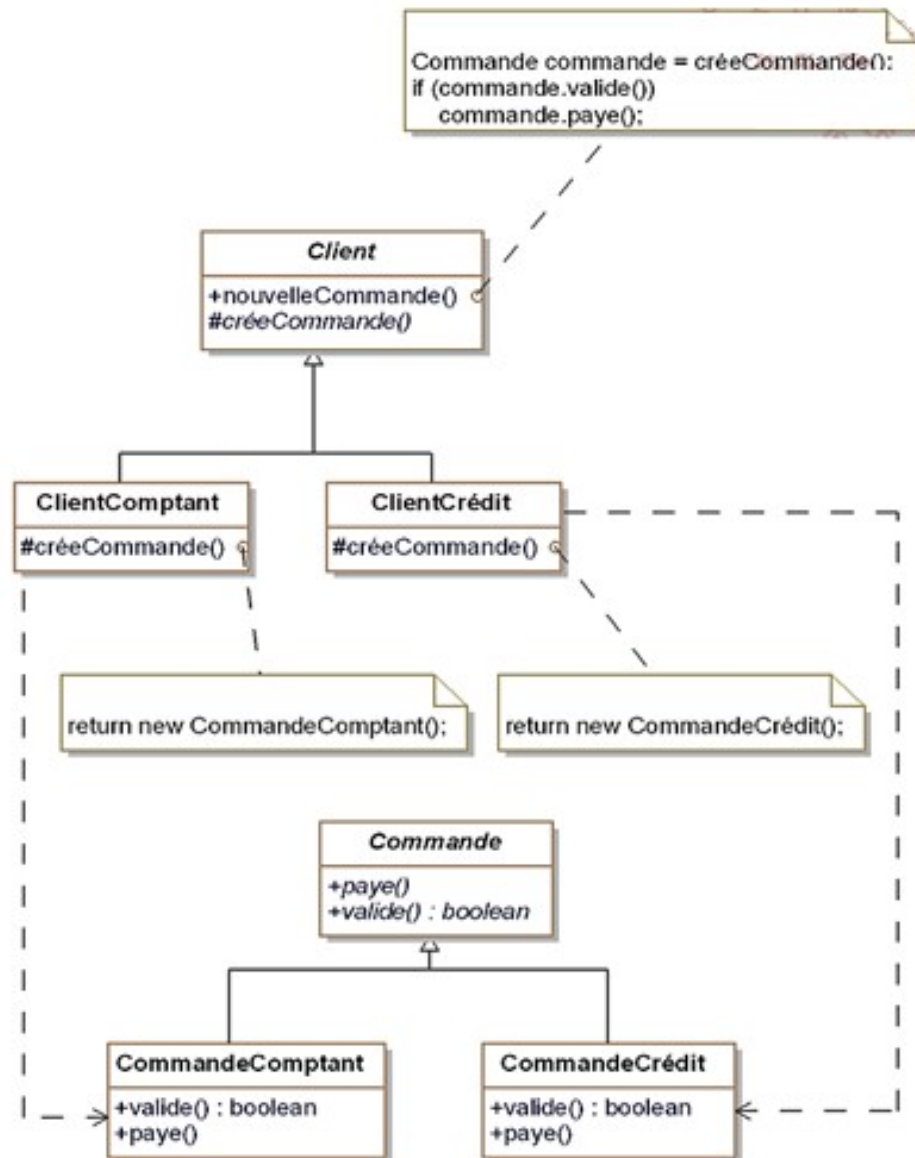
- Patrons de construction – Factory Method
- 

- Factory Method
  - Le but du pattern « Factory Method » est d'introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective.
  - Normalement, il ne faut créer qu'une seule instance de la fabrique, celle-ci pouvant être partagée par plusieurs clients.

# Les Design patterns

- Patrons de construction – Factory Method

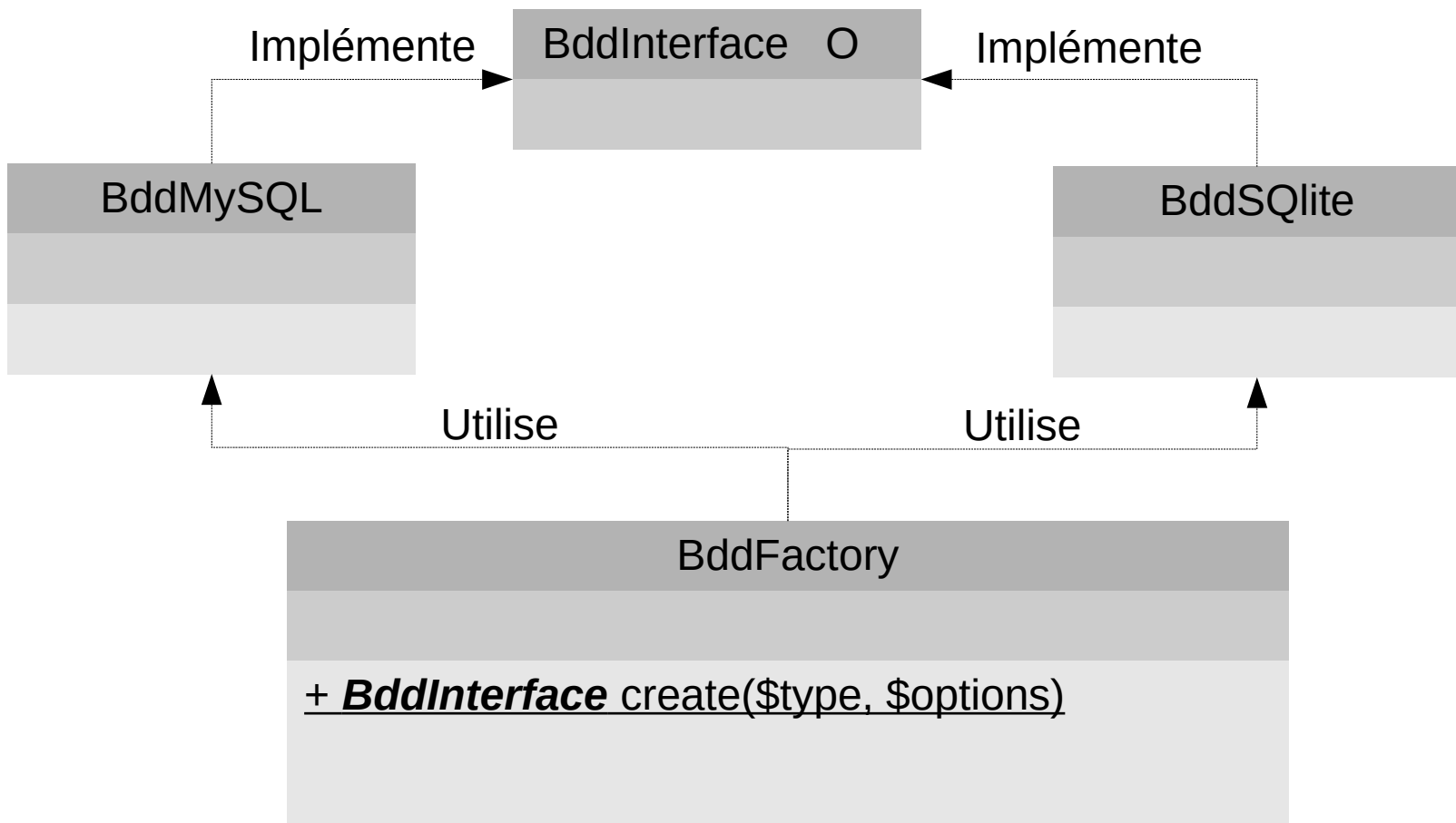
- Exemple du GoF



# Les Design patterns

- Patrons de construction – Factory Method

- Exemple allégé



# Les Design patterns

- Patrons de construction – Factory Method
- 

- **Quand ?**

- Quand vous êtes en train d'écrire du code dans le but de recueillir les informations nécessaires à l'instanciation d'objets.

- **Pourquoi ?**

- Pour contenir la logique de création d'objets similaires à un seul endroit
  - Permet de simplifier l'injection d'objets de tests lors de la phase de tests

# Exercice chapitre 3

- Exercice 6 – Base de données (1/2)
- 

## + Problématique :

- Nous avons besoin de gérer les données provenant d'une source de données
- Aujourd'hui le choix se porte sur MySQL, mais dans un souci de bien faire, nous allons mettre en place une fabrique, qui permettra de minimiser l'impact des modifications si le type de base venait à changer.

## + Solution :

- Créer une interface, pour uniformiser nos différents accès en base

- **interface** *BddInterface* {  
    *function* \_\_construct(...);      *function* connect(...);  
    *function* close(...);          *function* fetchAll(...);  
    ...  
}

- Créer des classes métiers « *BddMysql* », « *BddDbXXX* », ...
- Créer une fabrique « *BddFactory* »

# Exercice chapitre 3

- Exercice 6 – Base de données (2/2)
- 

## + Schématiser :

- Les nouvelles classes « BddMysql », « BddFactory » & Interfaces
- Des diagrammes de séquences entre les classes

## + Mettre en place :

- La connexion à la base de données, afin de confirmer l'identité des clients
  - Nom d'utilisateur = Prénom *en minuscule*
  - Mot de passe = Nom *en minuscule*



# Exercice chapitre 3

- Exercice 7 – Clients & Forfaits

---

**+ Besoins :**

- Adapter les classes Clients & Forfaits, du projet finalisé « 0-init ».
- Affichage de la fiche détails du client connecté.

**+ Informations :**

- Ne pas hésiter à proposer des améliorations, en ajoutant de nouvelles classes/interfaces/...

**+ Schématiser :**

- Les nouvelles classes « BddMysql », « BddFactory » & Interfaces
- Les diagrammes de séquences entre les classes

**+ Mettre en place :**

- Le nécessaire pour répondre au besoin

# Exercice chapitre 3

- Exercice 8 – Back Office Administrateur

---

- + **Problématique :**

- Afin de finaliser le projet comme il était – fonctionnellement parlant – à la fin du cours PHP Niveau 1, cet exercice va porter sur la mise en place des fonctions liées à l'administration.

- + **Besoins :**

- Afficher la liste des clients
- Permettre la suppression d'un client

- + **Pour aller plus loin**, des fonctionnalités peuvent être ajoutées :

- Ajout / Modification d'un client
- Ajout / Modification d'un forfait

- + **Réfléchir** à l'impact de nos besoins sur nos classes schématisées.

- + **Schématiser** les classes nouvelles, avec diagrammes de séquences pour expliciter les nouvelles interactions.

- + **Apporter** les modifications pour la zone d'administration.

# Chapitres

Environnements et outils de travail

Rappel sur la Programmation Orientée Objet en PHP

3

Réflexion avancée sur les bonnes pratiques

4

**Maîtrise des tests avec PHPUnit**

Introduction à la qualité et aux métriques de code

Collaboration des équipes

Les Frameworks, comment ne pas ré-inventer la roue ?

- Pourquoi des tests ?
  - « L'erreur est humaine »
  - Il apparaît évident qu'une application, aussi complexe, ou minime soit-elle, a besoin d'être testée, ne serait-ce que pour vérifier que le bouton mis en place fonctionne correctement.
  - Cependant dans certains cas, le même test doit être ré-itéré plusieurs fois, avant que le code source initial fournisse le bon résultat ; c'est un premier intérêt pour l'automatisation de ceux-ci.
  - Un second cas est le test de non-régression ; tester que dans 1 an, après une modification bénigne, celle-ci n'altère pas votre fonctionnement initial.

- Les impacts d'une anomalie
  - Le premier impact d'une anomalie est l'agacement de l'utilisateur, sa perception du produit. Si l'utilisateur en question a un pouvoir de décision quelconque, et si ses décisions ont un impact financier pour votre projet, alors son insatisfaction et son mécontentement doivent être évités !
  - L'impact d'une anomalie se traduit en coût financier :
    - Un coût direct, si l'anomalie empêche une transaction financière ou entrave fortement l'activité métier qu'elle est sensée permettre.
    - Un coût indirect, car sa non-détection préalable va nécessiter un correctif, parfois en urgence, avec un cycle de tests et de livraison complet.
  - On parlera donc d'anomalie bloquante, non bloquante, grave, mineure ou cosmétique, l'objectif étant de décrire la gravité de l'anomalie.

- On peut citer des anomalies célèbres
  - L'anomalie du Pentium 5
    - *Erreur d'algorithme qui entraînait des fautes dans les décimales des nombres en très haute précision*
    - Gravité technique : Faible
    - Anomalie relayée par la presse en masse
    - Obligation de correction par IBM pour **soigner son image de marque.**

- Bug de l'an 2000
  - Le bug de l'an 2000 a existé, bien que les répercussions ont été très faibles ; majoritairement parce que tous les systèmes pouvant avoir un problème ont été corrigés des mois (ou des années) avant.
  - **Gravité : Forte**
  - Ce bug – là aussi, très médiatisé –, a surtout été l'**occasion de monétiser et vendre** à la fois des correctifs inutiles, voir même des prestation psychologique.

---

– L'anomalie d'Ariane 5 premier lancement

- **Problème :**

*Le vol 501 est le vol inaugural du lanceur européen Ariane 5, qui a eu lieu le 4 juin 1996.*

*Il s'est soldé par un échec, causé par un dysfonctionnement informatique, qui vit la fusée se briser et exploser en vol seulement 36,7 secondes après le décollage.*

- **En cause :** Dépassement d'un Entier.

- **Gravité technique :** Forte



- Conclusion
  - Au travers des exemples précédents, on notera que la notion d'anomalie sort complètement du cadre technique pour prendre une dimension économique : le facteur coût entre immédiatement en ligne de compte et c'est même lui le plus souvent qui détermine le choix d'une solution.

- Une anomalie est :
  - une non-conformité relativement à des spécifications écrites,
  - un dysfonctionnement sans description associée dans les spécifications, correspondant à une règle de gestion implicite,
  - un comportement d'exécution qu'il conviendra de qualifier entre plusieurs parties,
  - un dysfonctionnement en conséquence d'un écart dans des éléments de configuration,
  - un risque à prendre en compte tout au long du processus de production logicielle, de l'idée jusqu'à son utilisation en production... et même au-delà !
- Une anomalie n'est pas :
  - une panne imprévisible,
  - un risque non maîtrisable et non maîtrisé.

- Les conséquences d'une anomalie sont :
  - **Humaines**, en termes de perception du logiciel par l'utilisateur final.
  - **Politiques**, si celui qui voit l'anomalie est un décideur,
  - **Financières**, si le dysfonctionnement entraîne une perte directe.
  - **Coûteuses**, selon la réparation du dysfonctionnement à mettre en œuvre.
  - **Stratégiques**, si l'anomalie peut discréditer toute une entreprise par une communication contre-publicitaire.
- Maîtriser les risques d'anomalie sur un projet, c'est maîtriser les coûts.
- La vérification de l'activité de production est donc une préoccupation constante dans laquelle tous les acteurs du projet doivent être impliqués.

- Les types de tests
  - Tests « Unitaires »
    - Le test unitaire n'est pas un test unique, mais le test effectué par le développeur d'un composant du logiciel, d'une brique. Nous sommes donc nécessairement en phase de réalisation.
    - Une équipe de réalisation qui est soumise à une pression (délais à tenir) aura naturellement tendance à négliger les tests pour répondre à une augmentation de charge de développement : le travailleur se focalise sur ce qu'il sait faire, surtout lorsqu'il y a une urgence.
    - Le test unitaire permet de s'assurer qu'une fonctionnalité de base renvoie toujours le même résultat, avec les mêmes paramètres.

- Tests « d'Intégrations » ou « d'Interfaçages »
  - Il y a deux façons de comprendre cette notion :
    - Il s'agit du test d'interfaçage entre deux composants unitaires.
    - Il s'agit du test d'interfaçage d'un logiciel dans un système global.
  - Ces tests s'articulent dans la phase qui suit les tests unitaires, lors de la réalisation du code, et non lors de la recette de l'application.

- Tests « Fonctionnels » ou « d'Homologations »
  - Les tests fonctionnels ont pour but de valider un logiciel dans sa globalité : ils correspondent à une phase *d'homologation* par une équipe de recette.
  - Cette équipe est constituée de spécialistes du test – les « homologateurs » – qui ne devraient pas avoir participé à la phase de réalisation de l'application.
  - C'est à cette phase de test que l'application est testée dans un nouvel environnement que celui de développement.
  - Ces tests sont beaucoup moins syntaxiques que les précédents, puisqu'ils relèvent d'un comportement global.

- TDD – Tests Driven Development
  - Le développement piloté par les tests est une technique utilisée pendant le développement.
  - L'idée fondamentale derrière un TDD est que vous écrivez un « premier » test, avant même qu'une seule ligne de code dans notre application ne soit écrite.
  - Avec un développement TDD, on écrit le test pour vérifier une fonctionnalité prévue, et ensuite on écrit le code pour correspondre au mieux à notre fonctionnalité.

- Qu'est-ce que PHPUnit ?
  - C'est un ensemble de classes fournies pour automatiser les tests unitaires.
- Installer PHPUnit
  - Dans un premier temps, PHPUnit nécessitait l'installation de PEAR, et de plusieurs dépendances, on retrouve encore aujourd'hui beaucoup de documentation allant dans ce sens.
  - Aujourd'hui, nous utilisons les fichiers PHAR, comme le montre le slide suivant.



# Environnements et outils de travail

---

- Configurer **PHPUnit**

- Télécharger dans « C:\Applications\phar\ »:

- <https://phar.phpunit.de/phpunit.phar>

- <https://phar.phpunit.de/phpunit-skelgen.phar>

- Configurer dans « C:\Applications\bin\ »

- phpunit.bat** : @php.exe "%~dp0../phar/**phpunit.phar**" %\*

- phpunit-skelgen.bat** : @php.exe "%~dp0../phar/**phpunit-skelgen.phar**" %\*

- Configurer NetBeans

- **Tools > Options > PHP**

- Onglet « **Frameworks & Tools** »

- Sélectionner PHPUnit dans la liste

- **PHPUnit Script** : C:\Applications\phar\phpunit.phar

- **Skelton Generator** : C:\Applications\phar\phpunit-skelgen.phar

# PHPUnit

- Premier test

## Code

src/Money.php

```
<?php
class Money
{
 private $amount;

 public function __construct($amount)
 {
 $this->amount = $amount;
 }

 public function getAmount()
 {
 return $this->amount;
 }

 public function negate()
 {
 return new Money(-1 * $this->amount);
 }

 // ...
}
```

## Test Code

tests/MoneyTest.php

```
<?php
class MoneyTest extends PHPUnit_Framework_TestCase
{
 // ...

 public function testCanBeNegated()
 {
 // Arrange
 $a = new Money(1);

 // Act
 $b = $a->negate();

 // Assert
 $this->assertEquals(-1, $b->getAmount());
 }

 // ...
}
```

> phpunit.bat tests/MoneyTest.php

- Configurer le projet (1/4)
- 

## – Configurer le projet

- Créer un dossier « tests » à la racine du projet
- Ouvrir les Propriétés de votre projet
- Dans l'onglet « Testing »
  - Cocher PHPUnit
  - Ajouter le dossier « tests », précédemment créé
- Dans l'onglet « Testing > PHPUnit »
  - Cocher « **Use Bootstrap** »,  
puis cliquer sur le bouton de génération du fichier
  - Cocher « **Use Bootstrap for Creating New Unit Tests** »
  - Cocher « **Use XML Configuration** »,  
puis cliquer sur le bouton de génération du fichier
  - Cocher « **Run All \*Test Files Using PHPUnit** »

# PHPUnit

- Configurer le projet (2/4)
- 
- Dans le dossier « **C:\Applications\src\** » :
    - Cloner le git suivant :  
<https://github.com/sebastianbergmann/phpunit.git>
  - Dans les Propriétés de votre projet
    - Dans l'onglet « Include Path »
      - Appuyer sur « Add Folder... »
      - Ajouter le dossier « **C:\Applications\src\** », précédemment cloné

Cette partie permettra à Netbeans, de connaître les classes spécifiques et ainsi proposer – entre autres – l'auto-complétion

# PHPUnit

- Configurer le projet (3/4)
- 
- Le fichier « bootstrap.php » :
    - Sera appelé systématiquement au début de tous nos tests
    - Doit donc inclure tout le nécessaire, comme le ferait notre Bootstrap applicatif « index.php »
  - ```
<?php  
define("APP_ROOT", realpath("/.."));  
  
chdir(APP_ROOT . '/public');  
  
spl_autoload_register(function ($className) { ... })
```

- Configurer le projet (4/4)
-
- Le fichier XML de configuration créé automatiquement permet de définir certaines configurations communes à tous nos tests.
 - Comme tout fichier XML, il ne peut contenir qu'une balise racine, ici : « <phpunit /> ».
 - L'exemple par défaut, est amplement suffisant pour commencer. Au besoin, il peut être nécessaire d'ajouter l'option « verbose » pour plus de détails lors d'un bogue dans nos tests.
 - ```
<?xml version="1.0"?>
<!-- see http://www.phpunit.de/wiki/Documentation -->
<phpunit
 colors="false"
 verbose="true" />
```

- Créer un fichier « BoiteAOutilsTest.php », pour tester notre classe « BoiteAOutils ».
- La classe hérite de « **PHPUnit\_Framework\_TestCase** ».
- Les jalons principaux sont :
  - **setUp** : Méthode appelée avant chacun des tests unitaires.
  - **tearDown** : Le pendant de « setUp », appelé après chacun des tests.
  - **test\*** : Nos fonctions de tests, qui doivent toutes commencer par **test...**
  - **class BoiteAOutilsTest extends PHPUnit\_Framework\_TestCase** {  
    *protected* function **setUp()** {...}  
    *protected* function **tearDown()** {...}  
    function **testGetGeoZoneFromPhoneNumber()** {...}  
}

- Notre test « **testGetGeoZoneFromPhoneNumber** », aura donc pour but de tester la méthode « **getGeoZoneFromPhoneNumber** » de la classe « *BoiteAOutils* ».
  - Pour cela nous appelons la fonction :  
`$test01=BoiteAOutils::getGeoZoneFromPhoneNumber("01*****");`
  - Et enfin nous utilisons une fonction dite d'assertion, propre à la classe héritée de PHPUnit, pour préciser le retour attendu :
    - `$this->assertEquals("Région parisienne", $test01);`
- Nous pouvons lancer le test :
  - clic droit sur le fichier de test, puis « Run ».
- Si le retour `$test01` est différent de ce que nous avons demandé, alors le test sera en erreur, sinon il sera validé.



- Exercice 1

---

- **Mettre en place :**

- des assertions supplémentaires dans nos tests précédents, pour tester un maximum de cas.
- des tests qui montre un manque ou des erreurs dans notre méthode de base,
  - Par exemple :
    - 00331\*\*\*\*\*
    - 12\*\*\*\*\*
  - Que faire dans ce genre de cas ?

# PHPUnit

- Créer des squelettes automatiquement
- 
- Clic droit sur le fichier PHP à tester
    - Tools
      - Create/Update tests
  - Un nouveau fichier est créé avec le suffixe « Test.php »
- 
- /!\ Parfois l'outil génère un Warning, et ne place pas le fichier généré dans le bon sous-dossier, il faudra le déplacer manuellement.

- **Créer** un squelette pour la classe « Vue »
- **Utiliser** vos connaissances pour tester au maximum cette classe

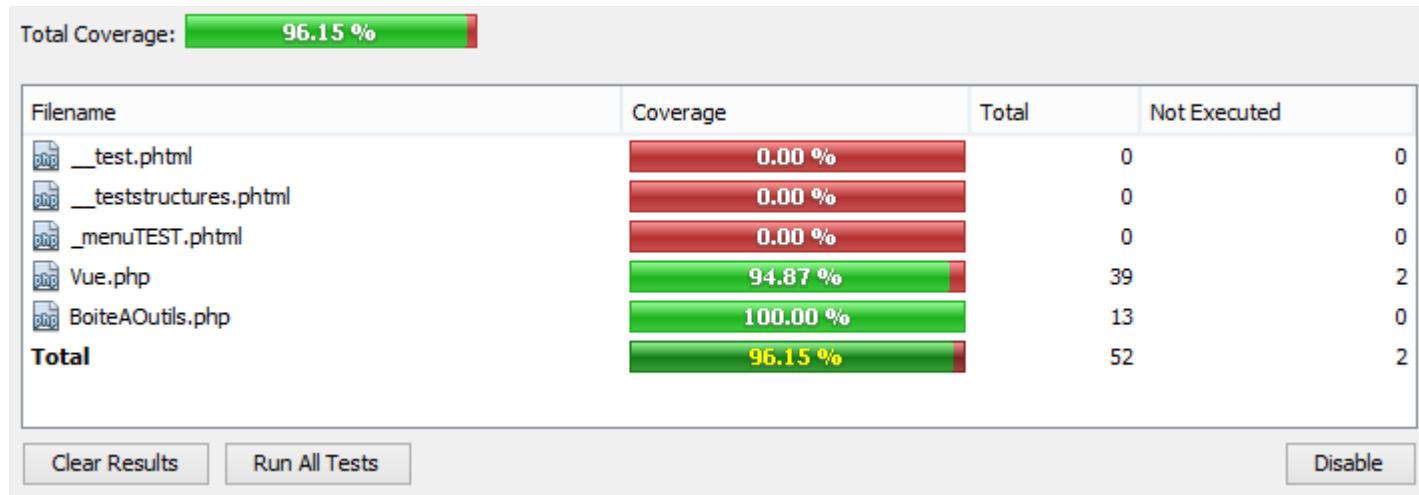
Astuce : Créer des petits templates « .phtml » avec des textes minimums pour ne pas avoir à tester de grosses pages HTML complètes.

- Code Coverage
- 
- La couverture de tests ou « Code Coverage », permet de savoir exactement quelles sont les lignes de code qui ont été traversées lors d'un test.
  - Pour l'activer :
    - Clic droit sur le projet
      - > Code Coverage
        - > Collect and Display Code Coverage
    - puis
      - > Show Report...
  - Et enfin, lancer au moins un test

# PHPUnit

- Code Coverage

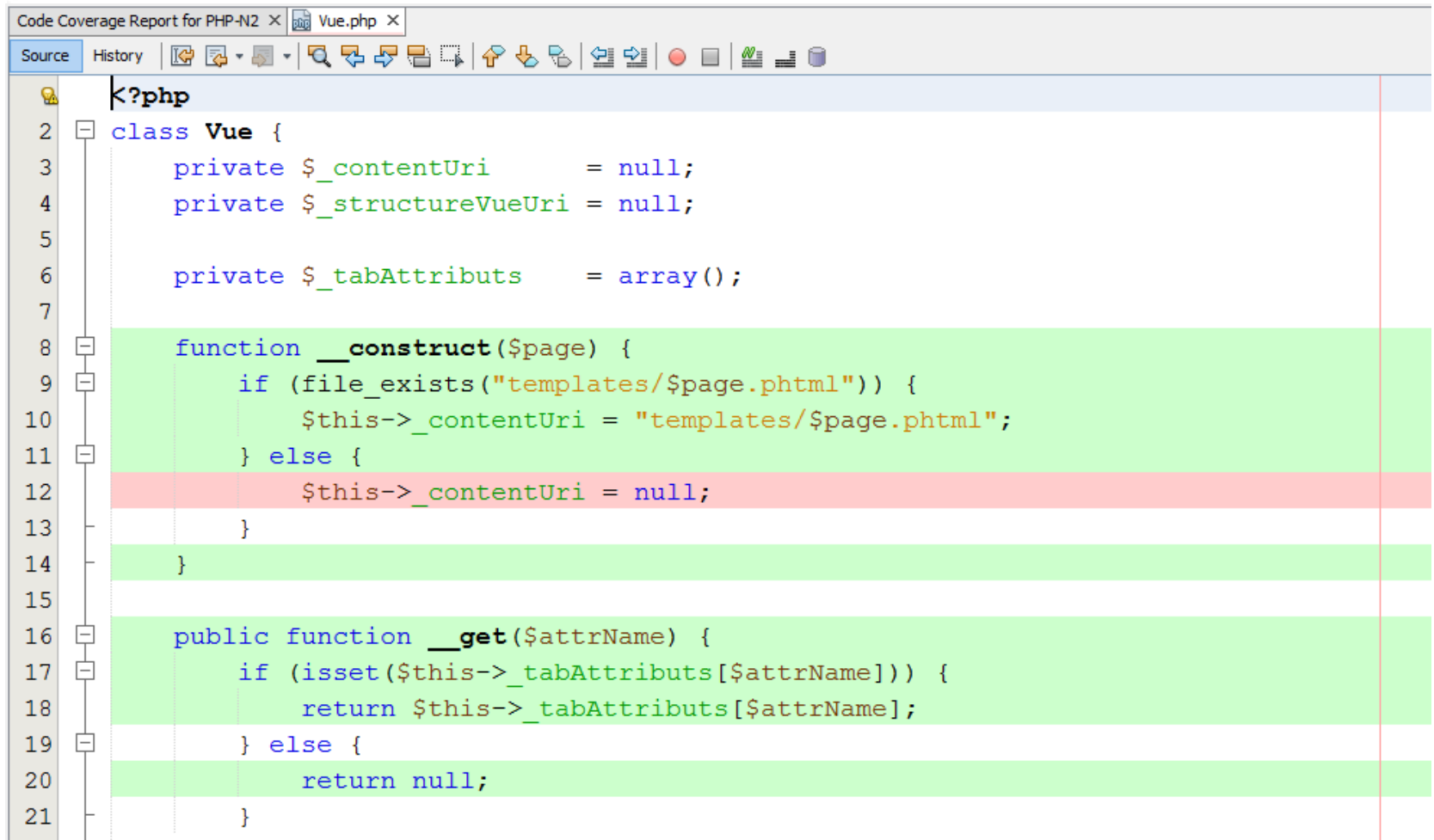
- À ce moment du projet, nous devrions avoir un résultat de couverture comme celui-ci :



- On constate, que les fichiers « phtml » sont traversés, mais ce ne sont pas des classes, donc ils ne comportent aucune ligne de code testable, bien qu'ils resteront à 0 %
- Ce qui est plus important ici, ce sont les 94,87% testés de notre classe « Vue.php » ; quels sont les 5,13% restants ?

# PHPUnit

- Code Coverage



```
Code Coverage Report for PHP-N2 x Vue.php x
Source History

1 k?php
2 class Vue {
3 private $_contentUri = null;
4 private $_structureVueUri = null;
5
6 private $_tabAttributs = array();
7
8 function __construct($page) {
9 if (file_exists("templates/$page.phtml")) {
10 $this->_contentUri = "templates/$page.phtml";
11 } else {
12 $this->_contentUri = null;
13 }
14 }
15
16 public function __get($attrName) {
17 if (isset($this->_tabAttributs[$attrName])) {
18 return $this->_tabAttributs[$attrName];
19 } else {
20 return null;
21 }
22 }
23 }
```

- Code Coverage
- 
- En couleur, les lignes de codes traversables
    - En vert, les lignes traversées
    - En rouge, les lignes non traversées
  - Il est important de bien comprendre qu'en vert, il s'agit de lignes « traversées », et non que le code traversé est fonctionnel à 100% ; ce sont les lignes qui ont été réellement exécutées par PHP lors du test.
  - Si vous n'avez pas autant de lignes vertes, mais que pourtant vous êtes certains que PHP doit être passé dans ces lignes, il est possible que la la couverture du code n'a pas été correctement configurée ou qu'une fermeture/ré-ouverture du fichier incriminé résolve le problème.

# PHPUnit

- Code Coverage

```
/**
 * @covers Vue::__get
 * @covers Vue::__set
 */
public function testSetterGetter()
{
```

```
16 public function __get($a
17 if (isset($this->_ta
18 return $this->_t
19 } else {
20 return null;
21 }
```

```
/**
 * @covers Vue::__get
 * @covers Vue::__set
 * @coversNothing
 */
public function testSetterGetter()
{
```

```
16 public function __get($a
17 if (isset($this->_ta
18 return $this->_t
19 } else {
20 return null;
21 }
```



- Code Coverage

---

- L'inverse étant possible, si vous avez des lignes en vert (Traversées par PHP) alors que vous n'aviez pas spécifiquement souhaité les tester, c'est que le mode strict n'est pas configuré.
- Pour l'activer, il faut ouvrir le fichier de configuration « configuration.xml », précédemment créé et nommé.
- Ajouter l'attribut « beStrictAboutTestsThatDoNotTestAnything » avec la valeur « true »
- Relancer les tests.
- Parfois Netbeans a besoin d'être relancé aussi.

- Exercice 3

- 
- **Vérifier** vos tests précédents, et assurez-vous d'avoir la meilleure couverture de code possible.
  - **Attention :**
    - Ce n'est pas parce que vous traversez des lignes de code que ceux-ci sont correctement testées.
    - Le principe des tests unitaires étant de faire un test par fonctionnalité :
      - Si une fonction contient des conditions, il faudra tester la dite fonction autant de fois que nécessaire avec divers paramètres afin de tester tous les cas possibles.
      - N'oubliez pas de tester aussi toutes les valeurs de retour.
  - **Effectuer vos tests :**
    - En mode strict/non strict
    - En activant/désactivant les annotations `@covers`
    - En utilisant l'annotation `@coversNothing`

- Exercice 4

---

- Nous allons maintenant tester notre classe « Application »
  - **Créer** un squelette de tests,
  - **Déterminer** les tests à effectuer,
  - **Remanier** ceux proposés par le squelette automatique si vous jugez cela nécessaire.
  - **Constater** que certains des tests effectuent des modifications globales ; pour cela il faudra user des annotations :
    - \* `@runInSeparateProcess`
    - \* `@preserveGlobalState disabled`
  - **Noter** que peut-être vous aurez besoin de modifier certaines classes du projet ; cela doit rester le plus ponctuel possible.

- Exercice 5

- 
- **Tester** l'intégration de notre application au travers de nos contrôleurs ;  
commençons par « **LoginController** » :
    - **Simuler** l'identification de l'administrateur.
      - User : admin
      - Pass : coucou
    - **Simuler** l'identification d'un client présent en base de donnée.
    - **Simuler** une erreur de saisie
  - **Vérifier** que votre couverture de code prend en compte tous les cas possibles

- Exercice 6 (1/2)

---

- **Problématique :**

- Afin de tester des fonctionnalités liées aux bases de données, on s'aperçoit vite que l'on ne peut pas travailler sur une base en production
  - Parfois parce que nous n'y avons pas d'accès direct
  - Ou plus fréquemment parce que nos tests portent sur l'insertion, la suppression ou la mise à jour de données.

- **Solution 1 :**

- Utiliser une base de données spéciale indépendante.

- **Solution 2 :**

- Utiliser une base de données spéciale, et faire une copie de notre base à chaque début de test.

- Exercice 6 (2/2)

---

- **Solution 3 :**

- Utiliser une base de données en mémoire, et exécuter un ensemble de requêtes SQL pour recréer notre base de tests.

- **Autres solutions :**

- Il est possible de mixer ces différentes solutions.

- **Peser** le pour et le contre de chacune des solutions.

- **Créer**, dans le projet principal, la classe « BddMemory », pour mettre en pratique la **solution n°3** :

*new PDO("sqlite::memory:");*

- **Tester** la classe « ClientsManager ».

- **Tester** la classe « Client.php »
  - **Créer** un client depuis des données brutes,
  - **Récupérer** un client depuis les données en base,
  - **Supprimer** un client dans la base de données,
  - Pour les plus rapides :
    - **Ajouter** une fonction d'insertion en base,
    - **Créer** les tests associés.
  - **Assurez-vous** de couvrir tous cas possibles

- Pour les plus rapides :
  - Tester les plugins
  - Tester les contrôleurs
  - Augmenter la cohésion de l'application en diminuant l'utilisation de fonctions natives pouvant apporter des modifications globales, ou en permettant à celles-ci de voir leurs fonctionnements surchargés ; lors des tests par exemple.



- Documentation en ligne de PHPUnit
  - <https://phpunit.de/manual/current/en/index.html>
  - Il s'agit de la documentation la plus récente pour la version stable à l'heure actuelle.
  - La documentation est en Anglais, il n'existe aucune traduction complète en Français.
  - On y trouve des exemples, mais surtout la liste complète des assertions possibles (<https://phpunit.de/manual/current/en/appendixes.assertions.html>)
  - Ou encore un chapitre dédié à la phase de *Fixture*, qui correspond à l'initialisation de l'état pour effectuer les tests  
(<https://phpunit.de/manual/current/en/fixtures.html>)

- Les patterns de test
- 
- Au même titre que nous avons utilisé les Designs Patterns, PHPUnit inclut des patterns de tests
    - **La doublure** – Objet de test très générique.
      - « Comment tester un système qui utilise un objet X très lourd, très lent, très verbeux ou carrément non disponible en environnement de tests ? »
      - Réponse : en remplaçant celui-ci dans les objets testés par une doublure, qui reprend son interface et que l'on injecte dans les objets en ayant besoin.
    - **L'espion** – Sous-ensemble de la doublure.
      - « Comment vérifier que mon système testé appelle bien les bonnes méthodes sur un second système externe sur lequel je n'ai pas la main ? »
      - Réponse : en remplaçant la passerelle vers ce système par un objet qui va mémoriser les méthodes qu'on y appelle et vérifier que ce soit bien les bonnes, dans le bon ordre.

- Les patterns de test

---

- L'objet **Bouchon** (Stub) – Similaire à l'espion
  - « Comment vérifier que mon système testé reçoit bien les bonnes données d'un système externe dont il dépend et sur lequel je n'ai pas la main ? »
  - Réponse : en remplaçant le système externe fournissant des données à mon système testé par un faux objet, dont les données qu'il va fournir à mon système sont connues à l'avance et avec lesquelles je peux travailler en tout sérénité.
- L'objet **Mock** – Mix du bouchon et de l'espion
  - « Comment vérifier que mon système testé appelle les bonnes méthodes et reçoit bien les bonnes données en retour d'un système externe dont il dépend et sur lequel je n'ai pas la main ? »
  - Réponse : en remplaçant le système externe fournissant des données à mon système testé par un faux objet, dont les méthodes qui sont censées y être appelées sont configurées et les données qu'elles sont sensées retourner à mon système sont connues à l'avance et avec lesquelles je peux travailler en tout sérénité.
- <https://phpunit.de/manual/current/en/test-doubles.html>

# Chapitres

---

Environnements et outils de travail

Rappel sur la Programmation Orientée Objet en PHP

Réflexion avancée sur les bonnes pratiques

Maîtrise des tests avec PHPUnit

**5**

**Introduction à la qualité et aux métriques de code**

Collaboration des équipes

Les Frameworks, comment ne pas ré-inventer la roue ?

# Introduction aux métriques de code

---

– Télécharger dans « C:\Applications\phar\ »:

- **Code\_Sniffer**

[https://squizlabs.github.io/PHP\\_CodeSniffer/phpcs.phar](https://squizlabs.github.io/PHP_CodeSniffer/phpcs.phar)

[https://squizlabs.github.io/PHP\\_CodeSniffer/phpcbf.phar](https://squizlabs.github.io/PHP_CodeSniffer/phpcbf.phar)

- **PHPMD**

<http://static.phpmd.org/php/latest/phpmd.phar>

- **PDepend**

<http://static.pdepend.org/php/latest/pdepend.phar>

- **PHPCPD**

<https://phar.phpunit.de/phpcpd.phar>

– Configurer dans « C:\Applications\bin\ »

- **phpcs.bat** : @php.exe "%~dp0../phar/**phpcs**.phar" %\*
- **phpcbf.bat** : @php.exe "%~dp0../phar/**phpcbf**.phar" %\*
- **phpmd.bat** : @php.exe "%~dp0../phar/**phpmd**.phar" %\*
- **pdepend.bat** : @php.exe "%~dp0../phar/**pdepend**.phar" %\*
- **phpcpd.bat** : @php.exe "%~dp0../phar/**phpcpd**.phar" %\*

# Introduction aux métriques de code

---

- Configurer Netbeans
  - **Tools > Plugins**
    - Rechercher « phpcsmd » dans l'onglet « Available Plugins », puis l'installer.
    - Si non présent le télécharger :  
« <http://plugins.netbeans.org/plugin/42434/phpcsmd> »
- Au besoin redémarrer NetBeans
  - **Tools > Options > PHP**
    - Onglet « Code Analysis »
      - » **Code Sniffer** : C:\Applications\bin\phpcs.bat
      - » **Mess Detector** : C:\Applications\bin\phpmd.bat
    - Onglet « PHPCSMD » :
      - » **PHPCS** : C:\Applications\bin\phpcs.bat
      - » **PHPMD** : C:\Applications\bin\phpmd.bat
      - » **PHPCPD** : C:\Applications\bin\phpcpd.bat
      - » **PDepend** : C:\Applications\bin\pdepend.bat

# Introduction aux métriques de code

---

- Configurer les PATH vers « php.exe »

- Trouver le programme

- Wamp :

**C:\wamp64\bin\php\php<version>\php.exe**

- Easy PHP :

**C:\Program Files (x86)\EasyPHP-DevServer-<version> \binaries\php\php\_runningversion\php.exe**

- Netbeans

- **Tools > Options > PHP > General**

- PHP 5 Interpreter : *<chemin vers php.exe>*

- Configurer les fichiers Batch (.bat)

- Modifier l'environnement système pour lui donner connaissance de php.exe

- Modifier chaque scripts (.bat), vus précédemment en remplaçant « @php » par « @*<chemin vers php.exe>* »

# Introduction aux métriques de code

- Clic droit sur le dossier à étudier
  - Clic sur « Scan with pdepend »
    - Lancer l'étude avec le bouton « Scan »

Formation - NetBeans IDE 8.1

File Edit View Navigate Source Refactor Run Debug Team Tools Window Help

Search (Ctrl+I)

Path: C:/wamp/www/Formation-phpunit/src

Pdepend: src

Pdepend Version (pdepend): 2.0.6  
Generated (generated): 2015-11-21T15:48:59

**Codelines**

|                                   |           |
|-----------------------------------|-----------|
| Lines of Code (loc):              | 1118      |
| Non Comment Lines of Code (ndoc): | 1091/1118 |
| Comment Lines of Code (doc):      | 27/1118   |
| Executable Lines of Code (eloc):  | 851/1118  |
| Logical Lines Of Code (lloc):     | 495/1118  |

**Counts**

|                             |     |
|-----------------------------|-----|
| Number of Packages (nop):   | 6   |
| Number of Classes (noc):    | 22  |
| Number of Interfaces (noi): | 2   |
| Number of Methods (nom):    | 101 |
| Number of Functions (nof):  | 0   |

**Classes**

|                                         |       |
|-----------------------------------------|-------|
| Number of Root Classes (roots):         | 2/22  |
| Number of Leaf (final) Classes (leafs): | 20/22 |
| Number of Concrete Classes (dsc):       | 20/22 |
| Number of Abstract Classes (dsa):       | 2/22  |
| Average Hierarchy Height (ahh):         | 0.2   |

Scan



# Introduction aux métriques de code

- Clic droit sur le dossier à étudier
  - Clic sur « Scan for violations »
- Lancer l'étude avec le bouton « Rescan files »

Formation - NetBeans IDE 8.1

File Edit View Navigate Source Refactor Run Debug Team Tools Window Help

Search (Ctrl+I)

Files Favo... Servi...

Formation

- Source Files
  - public
  - ressources
  - src
  - templates
- Test Files
  - src
    - bootstrap.php
    - configuration.xml
  - Include Path

ScanReport: src

Directory: C:/wamp/www/Formation-phpunit/src

| File                                         | phpcs-Errors | phpcs-Warnings | phpmd | phpcpd |
|----------------------------------------------|--------------|----------------|-------|--------|
| /App/Clients/ClientsManager.php              | 3            | 7              | 0     | 0      |
| /App/Clients/Client.php                      | 6            | 2              | 0     | 0      |
| /App/BoiteAOutils.php                        | 1            | 0              | 0     | 0      |
| /App/Forfaits/ForfaitFactory.php             | 0            | 5              | 0     | 0      |
| /App/Forfaits/ForfaitAbstract.php            | 14           | 1              | 0     | 0      |
| /App/Plugins/PluginGPC.php                   | 0            | 2              | 0     | 0      |
| /App/Plugins/PluginAd.php                    | 5            | 4              | 0     | 0      |
| /App/Plugins/PluginLayout.php                | 4            | 2              | 0     | 0      |
| /App/Plugins/PluginsArray.php                | 0            | 1              | 0     | 0      |
| /App/Application.php                         | 6            | 2              | 0     | 0      |
| /App/Bdd/BddMysql.php                        | 7            | 3              | 0     | 0      |
| /App/Bdd/BddMemory.php                       | 3            | 8              | 0     | 0      |
| /App/Bdd/BddFactory.php                      | 0            | 2              | 0     | 0      |
| /App/Vue.php                                 | 4            | 3              | 0     | 0      |
| /App/Controllers/ControllerAbstract.php      | 1            | 0              | 0     | 0      |
| /App/Controllers/LoginController.php         | 0            | 8              | 0     | 0      |
| /App/Controllers/DetailsclientController.php | 0            | 1              | 0     | 0      |

Overview phpcs phpmc phpcpd

/App/Controllers/\_TestController.php

24/24

Cancel

# Chapitres

4

Environnements et outils de travail

Rappel sur la Programmation Orientée Objet en PHP

Réflexion avancée sur les bonnes pratiques

5

Maîtrise des tests avec PHPUnit

Introduction à la qualité et aux métriques de code

6

**Collaboration des équipes**

Les Frameworks, comment ne pas ré-inventer la roue ?

# Collaboration des équipes

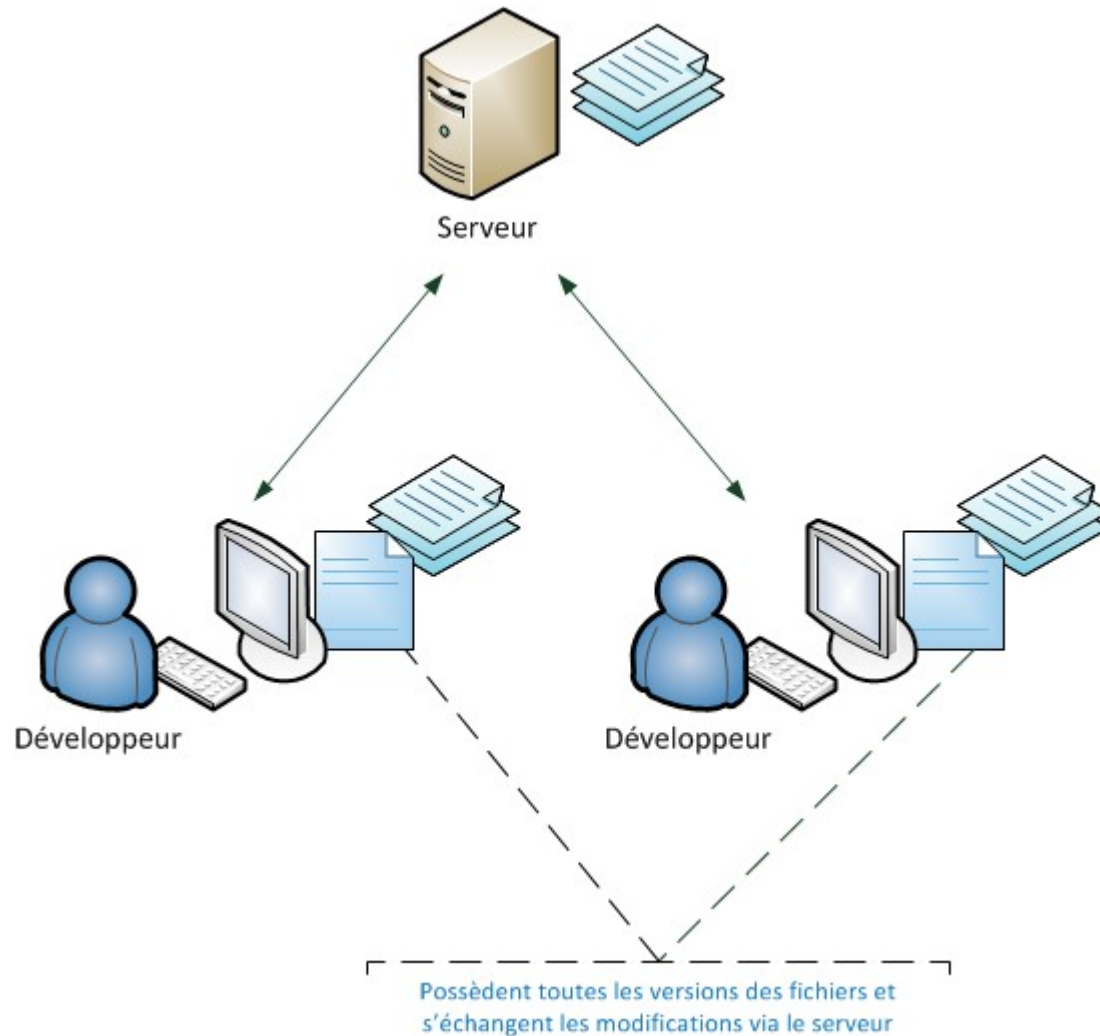
- Outils de versioning
- 
- Lorsque l'on travaille à plusieurs sur un projet (voire même tout seul dans certains cas), on se retrouve confronté à des problématiques de collaboration au sein d'un même projet.
    - » Qui a modifié mon code ?
    - » Pourquoi il y a un dossier « test », est-il encore utile ? Qui l'a créé ? Quand ?
    - » Tu peux m'aider sur le fichier X, par contre attention, je travaille sur Y !
    - » J'ai besoin de retrouver un bout de code, rapport à un bug déjà survenu, mais je ne sais plus ni où ni quand !

# Collaboration des équipes

- Outils de versioning
- 
- C'est exactement dans ce genre de cas qu'un système de gestion de version est indispensable.
    - Il en existe plusieurs, entre autres :
      - GIT
      - SVN
    - Ces systèmes sont capables de gérer correctement, entre autres choses :
      - Versions des fichiers
      - Auteur et « Modifieur » des fichiers
      - Regrouper les modifications de plusieurs fichiers
      - Capacité de retour en arrière par lot ou par fichier
      - Possibilité de travailler sur des branches différentes.
      - Capable de gérer les conflits de fichier et proposer une fusion quand deux personnes ont modifié le même fichier.

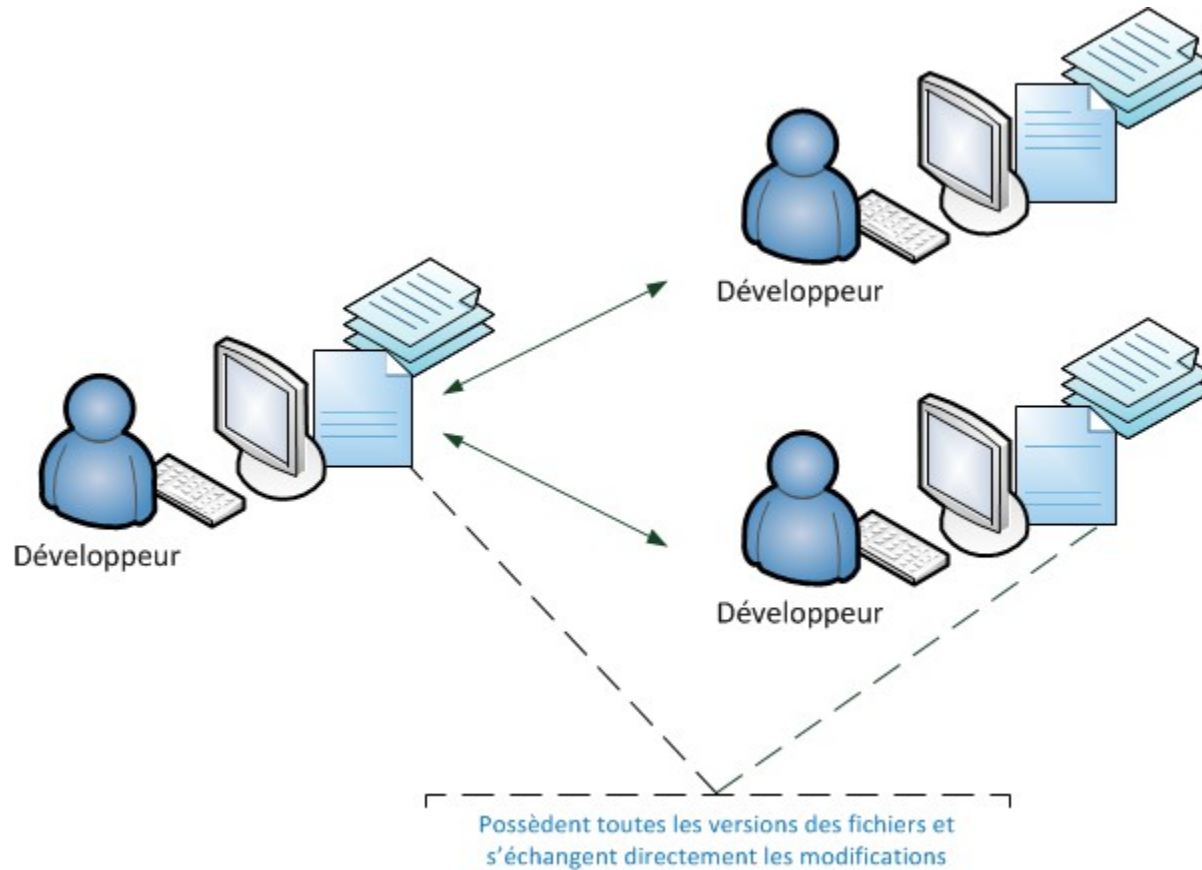
# Collaboration des équipes

- Outils de versioning



# Collaboration des équipes

- Outils de versioning
- Travail en Distribué



# Collaboration des équipes

- SVN – GIT
- 

- **SVN** (Subversion)

- L'un des plus connus, probablement encore le plus utilisé.
- SVN nécessite obligatoirement un accès au serveur.

- **GIT**

- Relativement récent, il a été développé par le créateur du noyau linux.
- Il est plus autonome que SVN, et considéré plus rapide.
- Ne nécessite pas de serveur centralisé, pratique pour des petits projets avec un développeur.

# Collaboration des équipes

- Bonnes pratiques
- 
- En équipe la bonne pratique est de définir des conventions avant de commencer le projet :
    - Convention de nommage des variables/fonctions/classes/...
    - Convention de syntaxe
      - Par exemple ne pas utiliser l'opérateur ternaire
      - Aligner les signes « = », et « () », si plusieurs lignes sont quasiment identiques.
    - ...



# Collaboration des équipes

- Outils de gestion de bogues / projets
- 
- Si des outils de gestion de bogues et/ou gestion de tickets sont mis en place, il est une bonne chose de s'y référer pour proposer une amélioration ou une découverte d'une incohérence, après avoir vérifié que personne d'autre ne l'a déjà fait.
  - Si un outil de gestion de projet est mis en place, il est probablement indispensable de l'utiliser
  - On peut citer dans ces outils :
    - Mantis
    - Redmine
    - Orange Forge
    - ...

# Chapitres

4

Environnements et outils de travail

2

Rappel sur la Programmation Orientée Objet en PHP

Réflexion avancée sur les bonnes pratiques

3

Maîtrise des tests avec PHPUnit

Introduction à la qualité et aux métriques de code

Collaboration des équipes

7

**Les Frameworks, comment ne pas ré-inventer la roue ?**

# Les Frameworks

---

- Traduit littéralement, le mot anglais « Framework » signifie « Cadre de travail ».
- Il s'agit d'un ensemble cohérent de composants éprouvés et réutilisables (Bibliothèques, Classes, Helpers, ...),
- Un ensemble de préconisations pour la conception et le développement d'applications.
- En PHP, les frameworks les plus connus actuellement sont :
  - Zend Framework
  - Symfony
- Ils sont donc conçus et utilisés pour modeler l'architecture des logiciels applicatifs, des applications web, des middlewares, des composants logiciels, ...

# Les Frameworks

---

- Concrètement le framework a pour but principal de fournir aux développeurs un ensemble de fonctionnalités, le plus souvent sous forme de classes, afin de ne pas avoir à ré-inventer la roue.
- Si on prend l'exemple de Zend Framework et que l'on étudie son fonctionnement interne, on y retrouve toutes les méthodologies étudiées lors de ce cours, à savoir les designs patterns, le fonctionnement SOLID.

# Les Frameworks

---

- Les avantages d'un Framework
  - Fonctionnel rapidement
  - Adaptable à l'infini (La limite est le langage)
  - Prend en compte les principaux besoins des développeurs
- Les inconvénients d'un Framework
  - Généralement lourd (Pas forcément plus lourd qu'une usine à gaz refaite à la main)
  - Nécessite l'apprentissage des librairies spécifiques au framework.