PROJET : PDC

LES PRINCIPES AVANCEES DE POO ET LES PATRONS GRASP

Réalisé par :

- Bouchafa Lotfi Rafik.
- Bouraba Nazih.

Groupe : SIL 1

Année : 2019-2020

Sommaire

1. Introduction :

Afin de se familiariser avec les patrons de conception, il nous a été demandé d'analyser un système existant en POO (open source) , pour cela nous avons choisi de travailler sur notre projet pluridisciplinaire qu'on l'a fait en 2cpi (application web) .

Après une analyse approfondie du code source de l'application nous avons pu desceller pas mal de problèmes de conception, ainsi que quelques portions de code que nous avons jugé être des failles de programmation impactant négativement la qualité de l'application livré à savoir : des dépendances abondantes entre classes, des méthodes non cohésives et surtout beaucoup de blocs du code répétitif . Ainsi nous essayerons donc dans une première partie de tracer le diagramme de classes du système existant, ensuite nous recenserons et expliquerons les différents problèmes trouvés, nous enchaînerons avec les diverses alternatives de solutions en mettant l'accent sur les patrons de conception vus en cours, et retracerons à la fin le nouveau diagramme de classes , histoire de savoir si le code a été raffiné et s'adapte mieux aux changements .

2.1 PRESETATION DE L'APPLICATION A ANALYSER

L'application à analyser est une application web programmée en « php » destinée a une petite entreprise afin d'automatiser la gestion des ressources humaine de cette dernière ; et ayant pour but principal de gérer le personnel et faire un suivi de leur état de travail, personnel et même pour le recrutement Et pour cela l'application vise les objectifs suivants :

-La gestion des informations relative de l'employé ainsi que son état de travail et l'édition et le suivi de son contrat, attestation et certificat de travail.

-la gestion des informations générales de l'entreprise (logo, cordonnés, thème etc.)

-Faire un suivi de sélection des candidats qui postulent.

-Afficher les cordonnées de l'ensemble des employés.

Le système global de l'application se compose essentiellement de 6 sous systèmes et d'acteurs

1) Les acteurs :

Ce sont les différentes entités qui agissent sur le système et qui possèdent des fonctionnalités d'utilisation et/ou de gestion

Gestionnaires

Etant un site dédié à la gestion du personnel pour une entreprise, l'utilisateur principal du site est le gestionnaire qui est le responsable des missions suivantes : suivi administratif du personnel, suivi de sélection des candidats qui postulent.

Administrateurs

C'est l'entité de pilotage de l'application. L'admin est la personne chargée de la gestion des paramètres relatifs à l'entreprise (logo entreprise, Raison sociale, coordonnées de l'entreprise et coordonnées du gérant, le thème de la page d'accueil) et des comptes des gestionnaires : il peut ajouter des comptes, changer le type des comptes (Admin / gestionnaire), modifier les mots de passe.

2) Les sous-systèmes :

Gestion administrative

La saisie et la consultation des informations personnelles et organisationnelles de l'employé (coordonnées, salaires, augmentations, congés, ...).

Recrutement et sélection

Un suivi de sélection des candidats qui postulent. Contient les informations de l'entretien

Paramétrage

Ce sont les vérifications et les tests effectués lors de la connexion d'un utilisateur (admin ou gestionnaire) au site, ou lors de la réinitialisation du mot de passe d'un utilisateur au cas où ce dernier l'oubli.

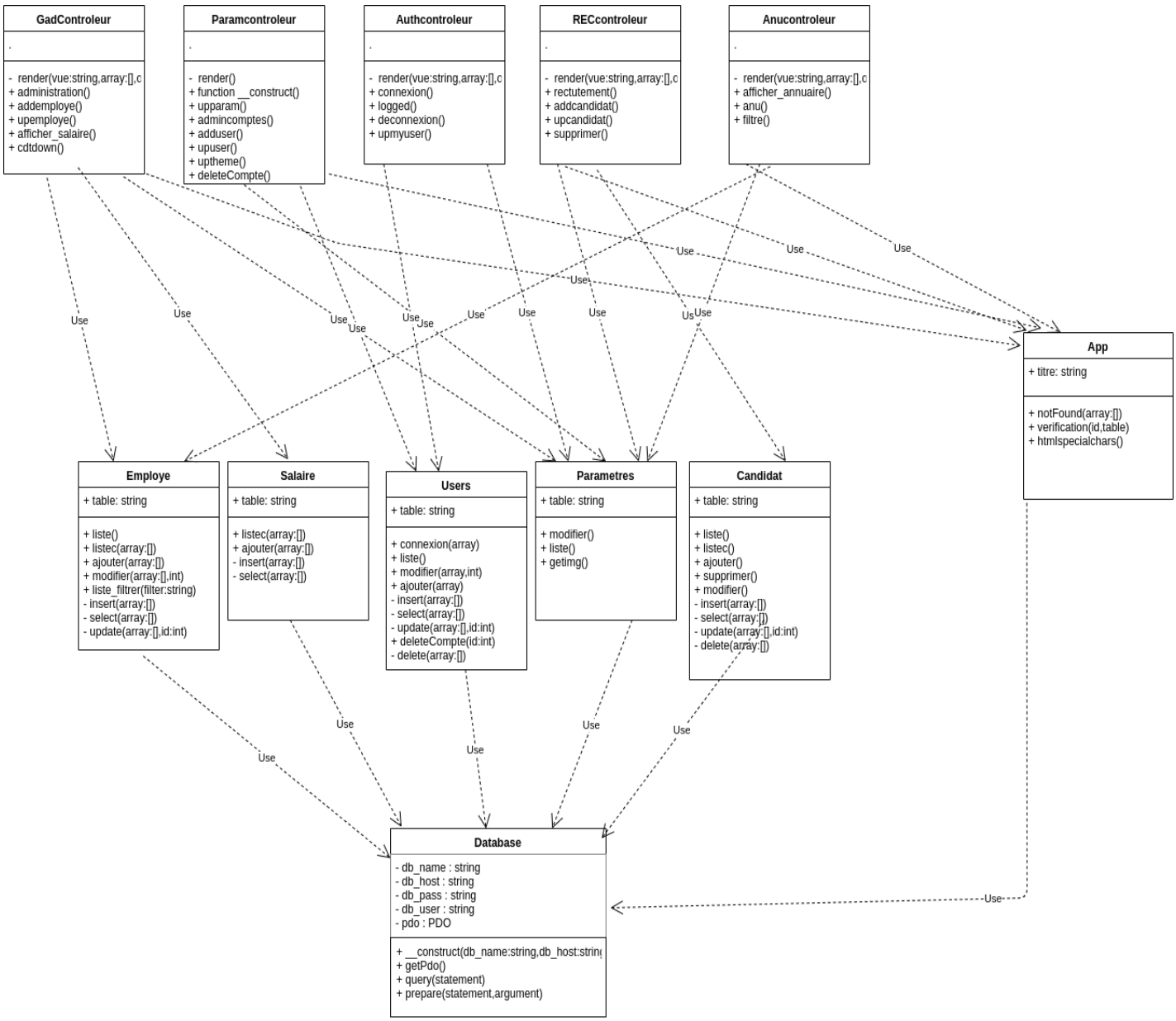
2.2 ANALYSE DU CODE SOURCE :

Cette application est composé de 2 packages , le premier « App » qui est spécifique à l'application elle-même et ne peut pas être réutiliser dans d'autre applications , le package « App » contient 3 sous packages « Controller » « model » « view » , ces sous packages contient des classes entités qui représente les tables de la base de donnée , des contrôleurs traitant l'aspect logique de l'application , et des pages (php,html) pour le graphisme , le 2éme package « Core » contient les classes qui ne sont pas forcément spécifique à notre application mais plutôt des classes génériques qu'on peut les réutiliser dans d'autre application , ce package contient une seule classe « Database » .

Dans ce qui va suivre, on va reconstituer le diagramme de classes à partir du code. Ensuite, on va recenser les problèmes liés au non-respect des principes avancés de POO tels que les principes SOLID ainsi que les problèmes qui contredisent les patrons GRASP. Enfin, nous proposerons une meilleure solution conceptuelle.

3. Reconstitution du diagramme de classes :

L'analyse du code source nous a mené à tracer le diagramme de classe suivant :



2.3 Identification des problèmes :

Dans cette partie, nous allons recenser tous les problèmes de conception que nous avons trouvé en analysant les classes une par une d'une part, et le diagramme de classes d'autre part.

SOLID ET LES PRICIPES AVANCEES DE LA POO:

Dans le tableau suivant, on résume les classes qui n'ont pas respecté les principes SOLID et les autres principes avancés de POO. Ensuite, on détaillera chaque principe ainsi que les problèmes liés.

La classe	SRP	OCP	LSP	ISP	DIP	DRY
AuthControleur	x	x				x
GadControleur		x				x
ParamControleur		x				x
RecControleur		x				x
AuthControleur		x				x
Employe		x				x
Salaire		x				x
Users		x				x
Candidat		x				x
Parametres		x				x
App	x	x				X
Database					x	X

SRP (SINGLE RESPONSIBILITY PRINCIPLE) :

Le principe SRP se résume dans le fait qu'une classe ne devrait avoir qu'un seul objectif fonctionnel.
Les problèmes relatifs à l'absence de ce principe sont :

1) La classe Authcontroleur :

cette classe gère la session courante de l'utilisateur , la méthode upmyuser() change les paramètres de l'application (identifiant , type , mot de passe) de l'utilisateur .

2) La classe App :

cette classe ne respecte pas SRP car elle fait beaucoup de choses non cohésives , elle est responsable de : l'affichage d'une page d'erreur , sécuriser les données introduites par les utilisateurs avant de les manipuler et les sauvegarder , elle vérifier qu'un élément existe dans une table de la base de donnée .

OCP (OPEN CLOSED PRINCIPLE) :

Ce principe indique que les classes doivent être conçues de sorte à pouvoir ajouter de nouvelles fonctionnalités (ouvertes à l'extension) avec un minimum de code changé (fermées à la modification).Ce principe est le plus violé dans la conception de cette application , les concepteurs de n'ont pas prévu quelques future extension. Les problèmes recensés à ce propos sont :

1) Dans tous les classes de cette application on dépend fortement des variables globale comme $_GET[] ou $_POST[] , par exemple dans la classe GADcontroleur dans la méthode afficher_salaire() si on veut dans le future afficher le salaire d'un employé avec la variable $_POST[] ou d'autre , on doit changer le code interne de cette méthode .

2) La classe Paramcontroleur :

2.1) Pour l'instant les types d'utilisateur de cette application sont soit 'gestionnaire' ou 'administrateur', seuls les administrateurs peuvent utiliser la classe Paramcontroleur pour manipuler les paramètres de l'application , mais si on veut créer d'autre type d'utilisateur qui aura par exemple la permission de changer le thème de l'application , on peut pas car le code de la classe Paramcontroleur est fermé à l'extension il accepte que les administrateurs .

2.1) La méthode SHA1() est utilisée pour hacher le mot de passe fourni par l'utilisateur avant de le sauvegarder dans la base de donnée pour des raisons de sécurité , si les spécifications de sécurité change , on doit changer tous les bouts de code ou il y a cette méthode .

2.2) La méthode upuser() est responsable de la modification du type d'utilisateur et de son mot de passe , si dans le futur on permet à l'utilisateur de changer son identifiant on doit changer le code interne de cette méthode .

2.3) Dans la méthode uptheme() : le nombre d'itérations pour la boucle for qui sauvegarde les images d'accueil est fixé à 3 (donc maximum 3 images d'accueil) . La modification de ce nombre nécessite une modification directe du code .

```php
for($i=1;$i<4;$i++)
{
  if (isset($_FILES['imgac'.$i]) AND $_FILES['imgac'.$i]['error']== 0)
  {
    $infosfichier = pathinfo($_FILES['imgac'.$i]['name']);
    $extension = $infosfichier['extension'];
          // Testons si l'extension est autorisée
    if (in_array($extension, array('jpg', 'jpeg', 'gif', 'png')))
    {
      if ($_FILES['imgac'.$i]['size'] <= 2000000)
      {
        $dest = 'public/theme/imgac'.$i.'.'.$extension;
        move_uploaded_file($_FILES['imgac'.$i]['tmp_name'],$dest);
        $img[$i-1] = $dest;
      }
    }
  }
}
```

3) Dans les classes «employe , salaire , users , candidat , parametres » le nom de la base de donnée est fixe 'rh ' , si on veut changer la base de donnée on doit parcourir tous les classes et changer a l'intérieur du code :

```php
public function liste()
{
  $db = new database('rh');

  $data = $db->query('SELECT * FROM '.$this->table);
  //DATA TABLEAU D'OBJETS
  return $data;
}
```

ISP (INTERFACE SEGREGATION PRINCIPLE)

D'après le diagramme de classes, on remarque la notion d'héritage est totalement absente dans la conception de cette application .

DIP(DEPENDANCY INVERTIONPRINCIPLE)

La classe Database utilise la classe PDO qui fournit une interface pour l'accès a plusieurs serveurs de base de données , mais ne fournit pas une abstraction de base de données , les requêtes SQL écrites dans cette application sont spécifique au serveur de base de données « MySQL »

```
$this->pdo=new PDO('mysql:dbname='.$this->db_name.';localhost='.$this->db_host.'','.$this->db_user.'','.$this->db_pass.'');
```

DRY (DON'T REPEAT YOURSELF) :

L'analyse du code de l'application a révélé qu'une grande portion du code était en train de se répéter .

1) La méthode render() se répète dans tous les classes contrôleurs .

2) Les méthodes « crud » pour effectuer des opérations sur la base de données se répète presque dans tous les classes « employe , users , salaire , candidat , parametres » .

3) La logique de la méthode 'upmyuser' dans la classe Authcontroleur est la même que la méthode 'upuser ' dans la classe Paramcontroleur .

3) Les méthodes 'insert , select , update , delete ' on des blocs de code répétitifs

```
        $sql_parts=[];
        $arguments=[];
        $inteog=[];
        $i = 0;
        foreach($array as $k=>$v)
        {
            $sql_parts[$i]=$k;
            $arguments[$i]=$v;
            $inteog[$i]='?';
                $i++;
        }

        $a = implode(",",$sql_parts);
        $b = implode(",",$inteog);

        $statement = 'INSERT INTO '.$this->table. '('.$a.') VALUES ('.$b.')';
        $db = new database('rh');
        $db->prepare($statement,$arguments);}
```

Les patrons GRASP:

Dans le tableau suivant, on résume les classes qui n'ont pas respecté les patrons de conception GRASP. Ensuite, on détaillera chaque patron ainsi que les problèmes liés.

La classe	Expert	Créateur	Couplage faible	Cohésion élevée	Contrôleur	Polymorphisme	Pure Fabrication	Indirection	Variation protégée
AuthControleur						X			x
GadControleur						X			X
ParamControleur						X			x
RecControleur						X			x
AuthControleur						X			x
Employe						X			x
Salaire						x			x
Users						X			x
Condidat						X			x
Parametres						X			x
App	x						x		
Database									

EXPERT :

Le patron Expert consiste en l'affectation d'une responsabilité à l'objet le mieux en mesure d'y satisfaire et qui possède les informations nécessaires. Ce patron a été mal respecté dans la conception de cette application. Notre analyse a révélé ce qui suit :
les classes contrôleurs sont les classes qui reçoivent les messages d'évènements et les données introduites par les utilisateurs de l'application , la méthode Verification() de la classe 'App' sert à vérifier et sécuriser les données introduites par les utilisateurs , donc cette méthode doit être dans les contrôleurs et non pas dans une classe pure fabrication 'App' .

COHESION ELEVE :

Le patron Cohésion élevée sert à assigner les responsabilités à une classe de sorte que sa cohésion reste élevée. Comme le principe SRP n'a pas été respecté, ceci provoque une très faible cohésion à l'intérieur des classes. On rappelle les classes soufrant une faible cohésion :
1) La méthode 'upmyuser()' de la classe 'Authcontroleur' n'est pas cohésive avec les autres méthodes .
2) La classe 'App' ne respecte pas ce principe car chaque méthode n'as aucune relation avec l'autre (l'affichage d'une page d'erreur , sécurisation des données , vérification qu'un élément existe dans la base de donnée) .

CONTROLEUR :

Le patron Contrôleur sert à assigner la responsabilité de gérer les messages d'évènements du système à une classe spécialisée du modèle. Comme mentionné dans l'analyse du code plus haut, la gestion de tous les évènements était assurée par les classes contrôleurs :

```php
public static function verification($id,$table)
    {
        if(isset($id))
        {
                $id = intval($id);
                if(0 < $id)
                $statement = 'SELECT id from ' .$table. '  WHERE id=?';
                $arguments[0] = $id;
                $db = new database('rh');
                if($db->prepare($statement,$arguments))
                {
                    return true;
                }
        }
        header('location:?p=error');
    }
```

Authcontroleur : gère les évènements liés à la session de l'utilisateur (connexion , déconnexion ..ext)
Gadcontroleur : gère les évènements liés à la gestion administrative (la saisie et la consultation des informations personnelles et organisationnelles de l'employé (coordonnées, salaires, augmentations, congés, ...).
Reccontroleur : gère les évènements liés à la sélection des candidats qui postulent.
Paramcontroleur : gère les évènements liés au paramétrage du site (changement du thème , changement des identifiants ..ext)

Ces contrôleurs ont régler plusieurs problèmes liés au patron Créateur et ceux du patron Faible Couplage et du patron Forte Cohésion.

POLYMORPHISM :

Le patron Polymorphisme incite à l'utilisation des interfaces pour traiter des alternatifs qui dépendent du type d'objets. le concepteur n'as pas appliqué ce principe (patron) du tout dans sa conception ,qui permet d'extraire des méthodes communes dans des classes mère , les classes modèles (Employe,Salaire,..ext) ont beaucoup de méthodes en commun (insert,select,update,delete) , même choses pour les classes contrôleurs (render) .

3. Solutions proposés :

3.1) SRP et patron expert :

La méthode de sécurisation des données doit être dans les classes contrôleurs car ce sont les premiers qui reçoivent les messages des utilisateurs .

La méthode verification() dans classe 'App ' , qui permet de vérifier qu'un élément existe dans une table de la base de données est la responsabilité des classes qui représentes les tables employe,salaire...ext car ce sont l'expert en ce qui concerne les tables de la base de données , cette solution permet de réduire le couplage entre (App et Database)

Exemple de la solution :

La methode verification est dans la classe 'App' :

```php
public static function verification($id,$table)
    {
        if(isset($id))
        {
                $id = intval($id);
                if(0 < $id)
                $statement = 'SELECT id from ' .$table. '  WHERE id=?';
                $arguments[0] = $id;
                $db = new database('rh');
                if($db->prepare($statement,$arguments))
                {
                    return true;
                }
        }
        header('location:?p=error');
    }
```

On créer dans chaque classe modèle une méthode exist()

```php
class Employe extends table {

    private $table = "employe";

    public function __construct()
    {…
    }

    public function exist($id)
    {
        if(isset($id))
        {
            $id = intval($id);
            if(0 < $id)
            $statement = 'SELECT id from ' .$this->table. '  WHERE id=?';
            $arguments[0] = $id;
            if($this->db->prepare($statement,$arguments))
            {
                return true;
            }
        }
        header('location:?p=error');
    }
```

3.2) DRY et Polymorphisme:

3.2.1) On remarque que la méthode render() elle se répète dans tous les contrôleurs , pour éliminer cette répétions on créer une classe mère "controleur" contenant cette méthode commune entre tous les contrôleurs , et on hérite de chaque classe controleur la classe "controleur"

```php
class  Controller {

  public function render($vue,$array=[],$other=[])
  {
    $parametres = new Parametres ;
    $param = $parametres->liste();
    ob_start();
    require 'app/views/'.$vue.'.php';
    $content = ob_get_clean();
    require 'app/views/templates/default.php';
  }
}
```

3.2.1) les méthodes "insert,select,update,delete" sont répétitifs dans tous les classes modèles , pour éliminer cette réplétion on créer une classe mère "table" contenant cette méthode commune entre tous les modèles , et on hérite de chaque classe model la classe "table"

```php
class table
{
    protected $table = "";

    public function insert($array=[])
    {}
    public function select($array=[])
    {}
    public function update($array=[],$id)
    {}
    public function delete($array=[])
    {}
}
```

3.3.3) on peut extraire le code qui se répète dans les méthodes "insert,update,select,delete"

et le mettre dans une méthode paramétrée .

3.3) OCP :

3.3.1) on rend tous les méthodes de notre système , qui dépend des variables globales

$_GET[] et $_POST[] des méthodes paramétrés pour qu'ils peuvent recevoir des données autres que GET et POST .

Exemple de cette solution :

Dans la classe gadcontroleur dans la méthode afficher_salaire() on change l'implémentation et on rend la donnée dans cette exemple c'est un entier 'id' dans les paramètres de cette méthode :

Avant :

```php
public function afficher_salaire()
{
    if(!isset($_GET['id']))
    {
        header('location:?p=error');
    }
    App::verification($_GET['id'],'employe');
    $salaire = new Salaire();
    $array['matricule'] = $_GET['id'];
    $historique =  $salaire->listec($array);
    $array2['id'] = $_GET['id'];
  $employe = new Employe();
    $data = $employe->listec($array2);
    $this->render('historique-salaire',$historique,$data);
}
```

Après :

```php
public function afficher_salaire($id)
{
    if(!isset($id))
    {
        header('location:?p=error');
    }
    App::verification($id,'employe');
    $salaire = new Salaire();
    $array['matricule'] = $id;
    $historique =  $salaire->listec($array);
    $array2['id'] = $id;
  $employe = new Employe();
    $data = $employe->listec($array2);
    $this->render('historique-salaire',$historique,$data);
}
```

Appel de la méthode afficher_salaire() :

```php
case 'salaire':
    $controleur  =  new GADcontroleur();
    $controleur->afficher_salaire($_GET['id']);
 break;
```

3.3.2) On extrait la méthode utiliser pour hasher les mots de passe avant de les enregistrer dans la base de donnée dans une méthode appart qu'on l'appel 'hasherMdp()' .

3.3.3) La méthode upuser() fait beaucoup de choses , on l'a divisé en 3 méthodes (upUserType(),upUserMdp()) , comme sa si on veut ajouter une autre méthode par exemple pour changer l'identifiant d'un utilisateur on ajoute directement une autre méthode "upUserId" (ouvert pour l'extension) sans changer le code .

```php
  public function upUserType(){…
  }
  public function upUserPassword(){…
  }
  public function upUserId(){…
  }
```

3.3.4) Dans la classe Paramcontroleur, dans la méthode uptheme() , la boucle qui permet d'ajouter des images d'accueils , on remplace le nombre fixe '3' par un attribut 'nbImageAcc' .

```php
for($i=0;$i<$param->getNbImgAcc();$i++)
{
```

3.3.5) Pour éliminer le couplage fort entre les classes de haut niveau qui représente les tables de la base de données (employe,salaire,..ext) avec la classe de bas niveau Database , on instancier l'objet qui permet d'initialiser une connexion et faire des requêtes a la base de donnée , dans le constructeur de la classe mère 'table' des classes modèles , et on enregistre une instance de cette connexion dans un attribut 'protected db' pour qu'il soit accessible par les classes fils de table.

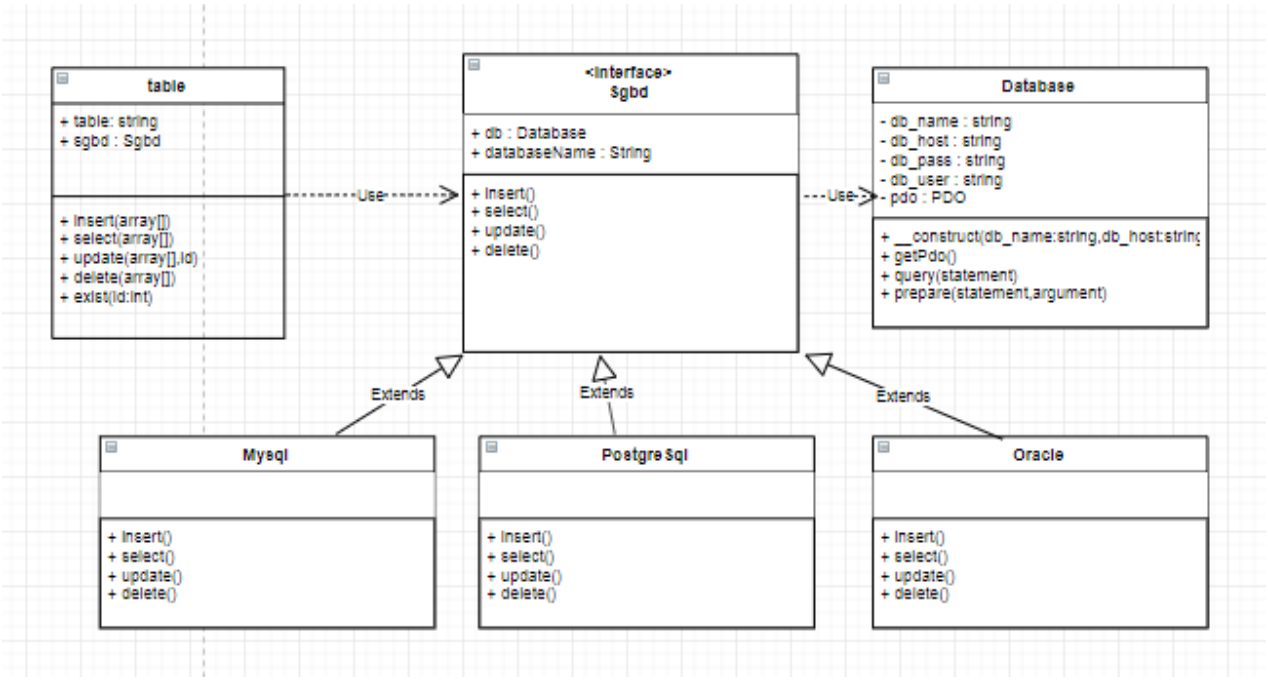
comme sa si on change le base de donnée on change le code dans un seul endroit .

```php
abstract class table
{
    private $databaseName = "rh";
    protected $table = "";
    protected $db = null;

    public function __construct()
    {
        $this->db = new database($databaseName);
    }
```

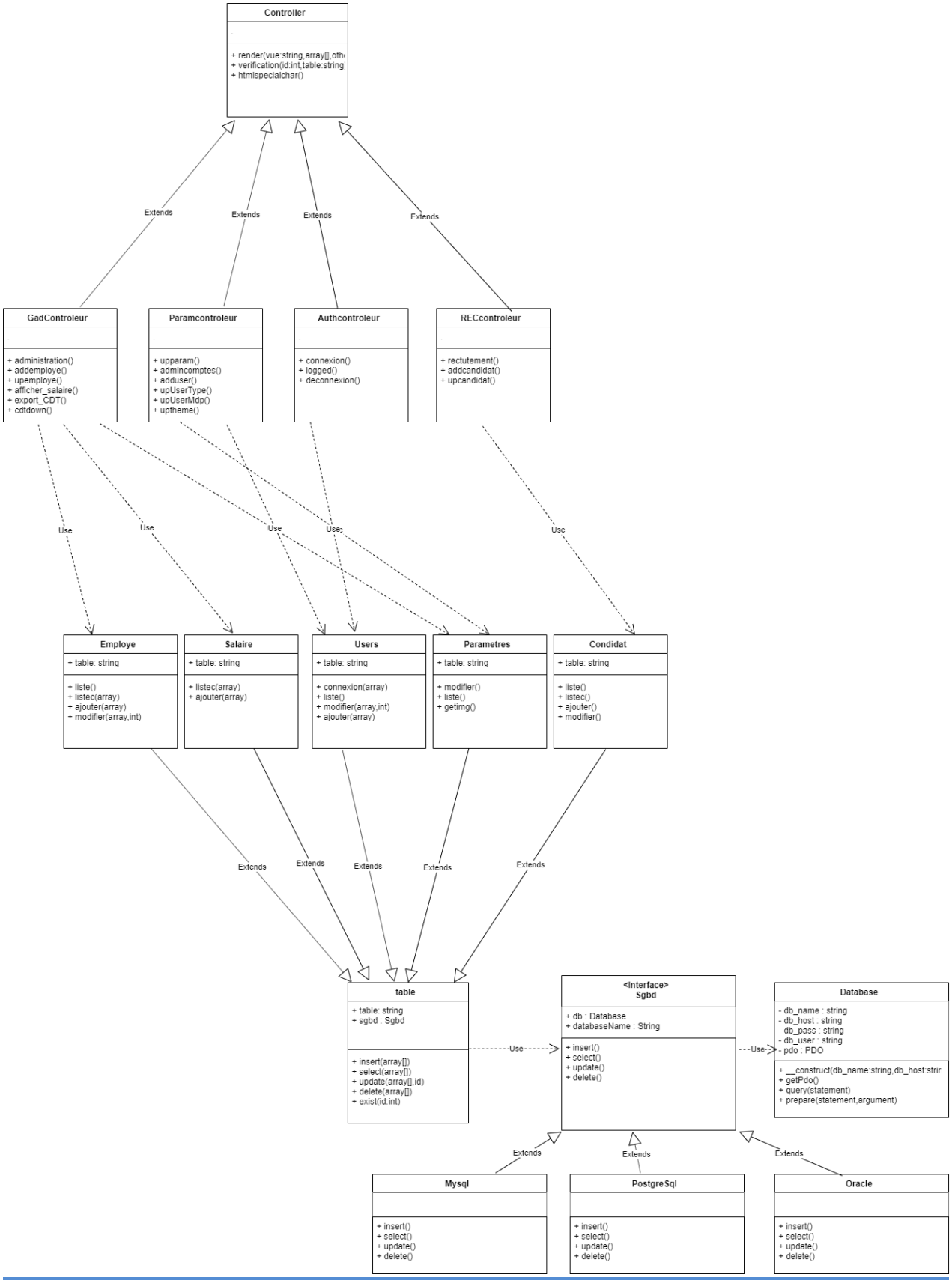
3.4) DIP :

Comme nous le avons mentionnée plus haut la classe 'Database' utilise l'extension PHP Data Objects PDO qui fournit une interface d'abstraction à l'accès de données, ce qui signifie qu'on peut utiliser les mêmes fonctions pour exécuter des requêtes ou récupérer les données quel que soit la base de données utilisée .mais le problème c'est que PDO ne fournit pas une abstraction de base de données : il ne réécrit pas le SQL, n'émule pas des fonctionnalités manquantes. C'est pour cela qu'on devrai utiliser une interface d'abstraction complète pour chaque sgbd (serveur de gestion de base de données)



4. Nouveau diagramme de classe :

Après avoir appliqué les principes poo et grasp , on a constaté que la classe ‘app’ est une classe pure fabrication inutile et elle ne contient aucune méthode après le modifications réalisés c’est pour cela qu’on la supprimer du diagramme de classe .



5. Conclusion :

Le conception initiale de l'application qui a été entre nos mains ne respecte pas quelques principes de base de la POO (polymorphisme) ainsi que ceux avancés n'étaient pas appliqués. Mais en général on peut dire que le système était destiné à l'évolution et à la modification à cause de la séparation des différents modules de l'application les contrôleurs d'une part qui permettent de définir la logique métier de l'application , les modèles qui représente les données utilisées par l'application, et en fin l'interface graphique représenté par un package vues .

L'application des principes SOLID et des patrons GRASP nous a permis de l'améliorer considérablement, notamment le couplage, la cohésion, l'évolution et le changement.