BTS SIO
SLAM4
PHP Objet - Avancé

TP Employés

Contexte

Cours Programmation objet avancé
Mise en œuvre des classes, espaces de noms, méthodes statiques et méthodes magiques

Point de départ : solution du TP précédent fournie par votre Professeur

Partie 1 : Restructuration de l'application

1. Dossier classes

Créer 2 sous dossiers :

- ✓ utilitaire

Déplacer dans ce dossier le fichier Bdd.php

- ✓ entity

Déplacer dans ce dossier les fichiers <entity>.php et <entity>Repository.php (Client.php, ClientRepository.php,…)

2. Classes

Les classes situées dans le dossier entity seront définies dans l'espace de noms classes\entity

Les classes définies dans le dossier utilitaire seront définies dans l'espace de noms classes\utilitaire

3. Auto chargement des classes

Mettre en place l'auto chargement des classes par la méthode spl_autoload.
Supprimer la ligne

include_once(INCLUDE_PATH.'autoload.php');

dans le fichier config\conf.php

Supprimer le fichier includes\autoload.php
Créez le fichier spl_autoload.php dans le dossier includes :

```php
<?php

spl_autoload_extensions(".php,.class.php,.int.php");
set_include_path(
        INCLUDE_CLASS . 'entity' . PATH_SEPARATOR .
        INCLUDE_CLASS . 'utilitaire' . PATH_SEPARATOR . '\\' . PATH_SEPARATOR .
        get_include_path() . PATH_SEPARATOR
);
```

Modifier le fichier conf.php qui doit inclure ce fichier en rajoutant la ligne :

```
include_once(INCLUDE_PATH.'spl_autoload.php');
```

Pour que l'inclusion des classes se passe bien, il va falloir rajouter une instruction … à vous de voir

4. Gestion des espaces de noms

Après ces modifications, pour que votre application marche, il va falloir écrire un certain nombre de use dans les différents fichiers.

A titre d'exemple, voici le début de la classe client :

```
namespace classes\entity;
use classes\entity\ClientRepository;
```

Il faudra aussi modifier les appels aux classes Exception et PDO. Elles se trouvent à la racine. Par exemple : \PDO

Il faudra aussi modifier les constantes de classe (<entity>Repository).
Par exemple :

```
class ClientRepository {

    CONST CLASSE = '\classes\entity\Client';

    private $conn;
```

Modifier vos fichiers pour que l'application marche comme dans la solution fournie par le professeur. Toutes les instanciations se feront encore sur le modèle

$obj = new NomClasse(…);

Partie 2 : Une classe ClientCommandeException

On vous demande maintenant de créer une classe ***ClientCommandeException.***
Cette classe sera située dans le dossier classes\utilitaire et sera définie dans l'espace de noms classes\utilitaire. Elle héritera de la classe Exception et comprendra :

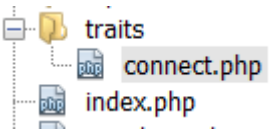
- ✓ l'attribut privé : $nomFichierLog
- ✓ la méthode publique EcritureJournal() qui pour l'instant ne fera rien. Dans la suite, elle ira écrire dans un journal de log au format XML.

Dans les classes <entity>Repository, toutes les exceptions déclenchées par le programmeur seront de la classe ClientCommandeException.

Il faudra aussi modifier le script index.php afin d'intercepter ces exceptions. On prévoira un try… catch général pour intercepter toutes les autres exceptions.

Partie 3 : Un trait

Etant donné qu'on peut avoir besoin du même script de connexion à la BD dans plusiers projets ou classes, on va créer un trait que l'on va intégrer à la classe Bdd.
Créer à la racine du site un dossier traits et créer le fichier connect php qui va contenir le trait :




```php
namespace traits;

use \PDO;
use \Exception;

trait Connect {

    private function connectionBd() {
        try {
            $connect_str = "mysql:host=localhost;dbname=clientsoo";
            $connect_user = "root";
            $connect_pass = "";
            $options[PDO::ATTR_ERRMODE] = PDO::ERRMODE_EXCEPTION;
            return new PDO($connect_str, $connect_user, $connect_pass, $options);
        } catch (PDOException $e) {
            throw new Exception('Erreur à la connexion <br>' . $e->getMessage());
        }
    }
}
```

On remarque que l'on peut appliquer à un trait :

- ✓ Un espace de noms,
- ✓ Des alias sur des espaces de noms

Inclure ce trait dans la classe Bdd.

Partie 4 : Une classe Repository

On remarque que les méthodes findById et findAll vont se retrouver dans toutes les classes de type <entity>Repository. Ce n'est pas une bonne idée car la seule chose qui les diffèrencie l'une de l'autre est le nom de la classe ou le nom de la table concernée.

On va donc créer une classe Repository située dans le dossier classes\utilitaire et qui sera définie dans l'espace de noms classes\utilitaire.

Toutes les classes <entity>Repository hériteront de cette classe.
Extrait de la définition de cette classe :

```php
class Repository {

    protected $_classe;
    protected $conn;
    private $_table;

    public function __construct($classe) {
```

Et constructeur de la classe ClientRepository :

```
public function __construct() {
    parent::__construct(self::CLASSE);
}
```

Modifier votre application afin de ne plus avoir les méthodes findById et findAll dans les classes <entity>Repository mais de les avoir dans la classe Repository.
Testez avec le script index.php

Partie 5 : Find magique

On ne veut plus voir de méthodes findByVille ou findByDate.
On veut intercepter ce type de méthode dans une méthode __call et construire dynamiquement la requête.

Exemple si on lance la méthode findByVille('Bordeaux') depuis le repository client :
Ville correspond au champ de la BD et la requête se construit ainsi :

Select * from client where ville='Bordeaux';

Mieux : on peut aussi lancer la méthode suivante depuis le repository Commande :

findByDatecde_and_idclient($uneDate, 4);

Elle va chercher les commandes les clients numéro 4 qui ont passé une commande le $uneDate.

Pour ceci, prévoir les méthodes dans la classe Repository :

- ✓ Private function findBy : renvoit une collection d'enregistrements
- ✓ Private function findOneBy renvoit un objet d'une classe métier
- ✓ Public function find($id)
- ✓ Public function findAll();

Ne pas dupliquer le code, ce qui amènera à créer des méthodes privées

Bon courage