Rapport - Qualité de Développement

Réalisé par

Ferhani Hicham - Haye Marc – Moreira Pereira Paulo - Nunes Evan - Rostaing Damien

SOMMAIRE

Exemple 1: Pattern Singleton, et Responsabilité Unique

Lorsque l'on utilise le pattern Singleton, on souhaite garantir qu'il n'existe qu'une seule instance d'une classe dans le programme et fournir un point d'accès global à cette instance.

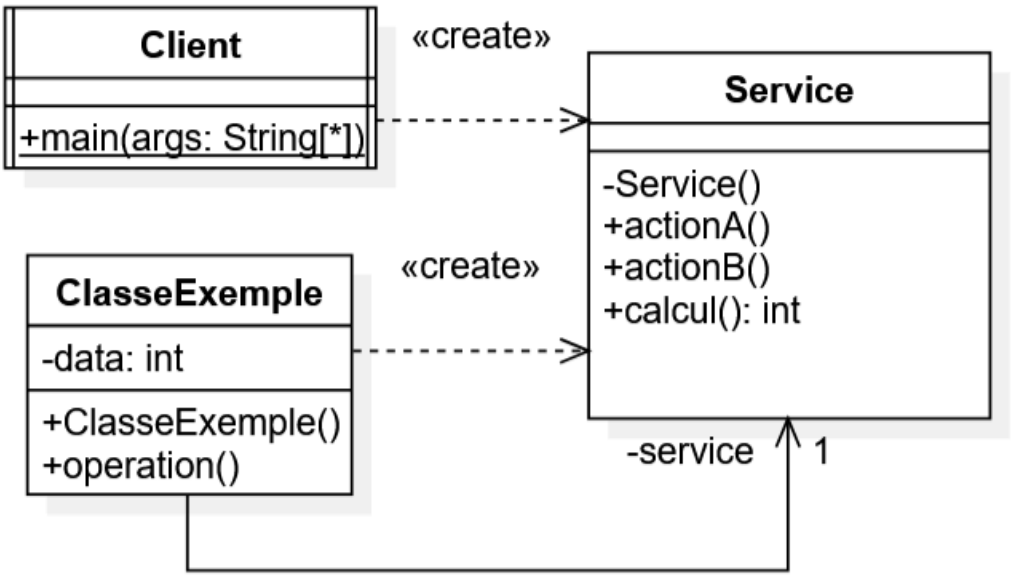
Les autres objets/classes de l'application utilisent l'instance "globale" et ne doivent pas pouvoir l'instancier directement (l'utilisation de new doit être bloquée en dehors de la classe). L'instance globale doit être accessible à travers une méthode qui l'expose et qui a une visibilité appropriée.

Ce type de bonne pratique est implicite et décidée de manière assez instinctive, car il n'y a que deux cas où il était nécessaire de n'avoir qu'une seule instance de classe à la fois, accessible globalement par toutes les classes de l'architecture.

1.1 - Connexion à la base de données

Tout d'abord, nous avons utilisé le pattern Singleton sur la connexion afin d'éviter d'en créer une nouvelle à chaque fois qu'une requête est faite. Cela permet de ne pas avoir plusieurs connexions inutiles en simultanées qui pourraient ralentir la base de données et accumuler inutilement sa mémoire.

Nous avons également utilisé le principe de Single Responsibility pour centraliser la méthode de connexion, ce qui facilite le refactoring en cas de modification de celle-ci.



L’application de cet exemple se trouve dans la classe ConnexionBaseDeDonnees :

```php
10      private static ?ConnexionBaseDeDonnees $instance = null;
```

```php
        25 usages  march
35      public function getPdo(): PDO{
36          return ConnexionBaseDeDonnees::getInstance()->pdo;
37      }
38
        26 usages  march
39      public static function getInstance(): ConnexionBaseDeDonnees{
40          if (is_null(ConnexionBaseDeDonnees::$instance))
41              ConnexionBaseDeDonnees::$instance = new ConnexionBaseDeDonnees();
42          return ConnexionBaseDeDonnees::$instance;
43      }
```

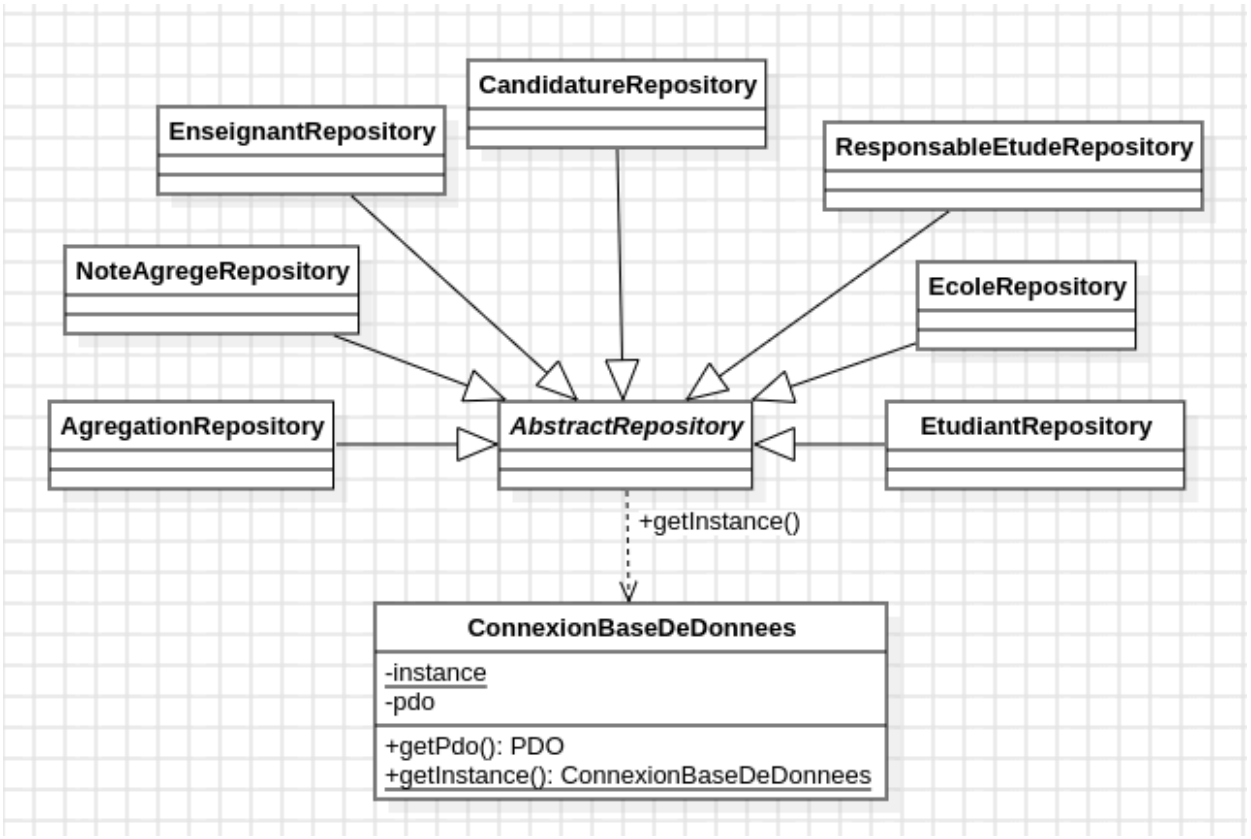
Il suffit donc aux autres classes d’appeler cette méthode pour assurer que l’on utilise toujours la même instance de la connexion.

Exemple d’utilisation dans EcoleRepository, où l’on effectue une requête pour obtenir des ecoles dans la base de donnée :

```php
27          $pdoStatement = ConnexionBaseDeDonnees::getInstance()->getPdo()->prepare($sql);
28
29          $values = array(
30              "etudid" => $etudid
31          );
32
33          $pdoStatement->execute($values);
```

Voici l’architecture actuelle (simplifiée) des classes qui se connectent à la base de données:

Diagramme de classe des repository au sprint 4



Commit associé

1.2 - Session Utilisateur

Dans la même idée, la session d'un utilisateur sur l'application nécessite elle aussi d'être instanciée une seule fois pour éviter les conflits de sessions simultanés pour une même fenêtre:

```php
4 usages
private static ?self $instance = null;
```

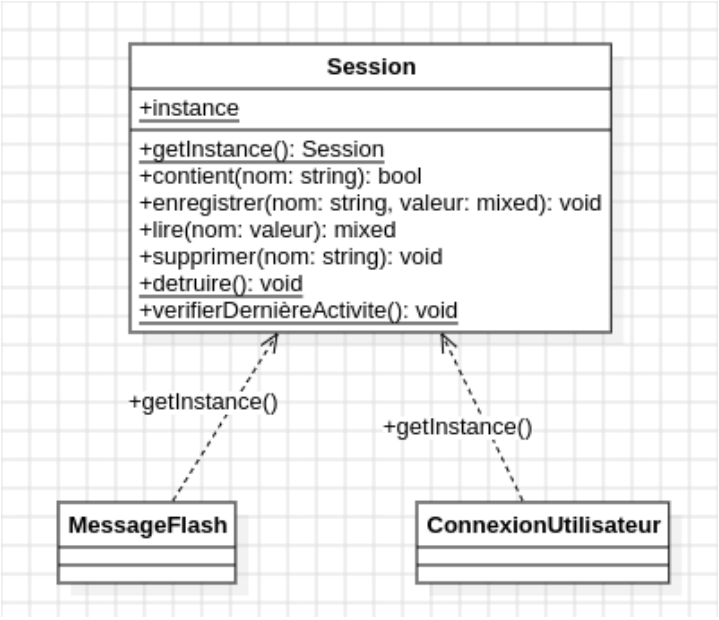
```php
8 usages  march
public static function getInstance(): self {
    if (is_null(self::$instance)) {
        self::$instance = new self();
    }

    self::verifierDerniereActivite();
    return self::$instance;
}
```

Exemple d'utilisation dans ConnexionUtilisateur:

```php
Haye Marc +1
public static function connecter(string $loginUtilisateur): void{
    Session::getInstance()->enregistrer(self::$cleConnexion, $loginUtilisateur);
}
```

Diagramme UML:



Commit associé

Exemple 2: Substitution de Liskov, et Ségrégation des Interfaces

2.1 - Contrôleur Administratif, et Contrôleur Utilisateur

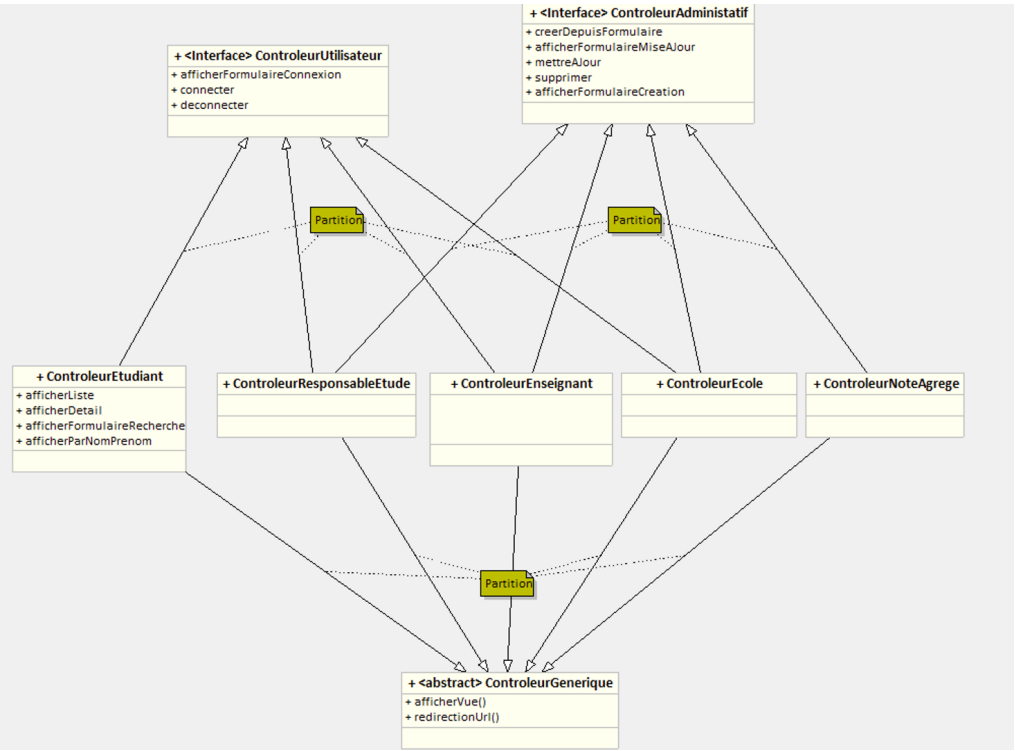
Durant la conception des contrôleurs de l'architecture MVC, l'application est pensée pour prendre en compte différents types d'utilisateurs : étudiant, école, enseignant et responsable de poursuite d'étude. Chacune de ces classes sont conçues comme sous-classe d'une classe mère abstraite Utilisateur.

Ces derniers ayant des informations stockées dans la base de données (login, mot de passe), la méthode CRUD est appliquée sur ces utilisateurs. (par exemple, modifier le mot de passe), et afin de respecter le principe d'inversion des dépendances : les différentes classes ne doivent pas dépendre d'implémentations concrètes, mais d'abstractions, ce qui est appliqué avec la classe abstraite Utilisateur..

Cependant une contrainte imposée par le client est de ne pas pouvoir modifier ni supprimer d'étudiants via l'application. La classe étudiant n'appliquant plus CRUD, elle ne respecte plus le principe de ségrégation des interfaces, qui dit qu'un objet ne doit pas être forcé de dépendre de méthodes qu'il n'utilise pas. Globalement, il ne faut pas qu'une interface définisse dans son contrat des méthodes qui ne pourront pas être implémentées par la classe implémentant l'interface. Un refactor est nécessaire.

Le choix a été de créer deux interfaces : ControleurUtilisateur, qui applique les tâches liées à la connexion utilisateur, et ControleurAdministratif, lié aux actions CRUD. Cela limite les dépendances, rend plus adaptable le code, et permet de mieux respecter le principe de ségrégation des interfaces.

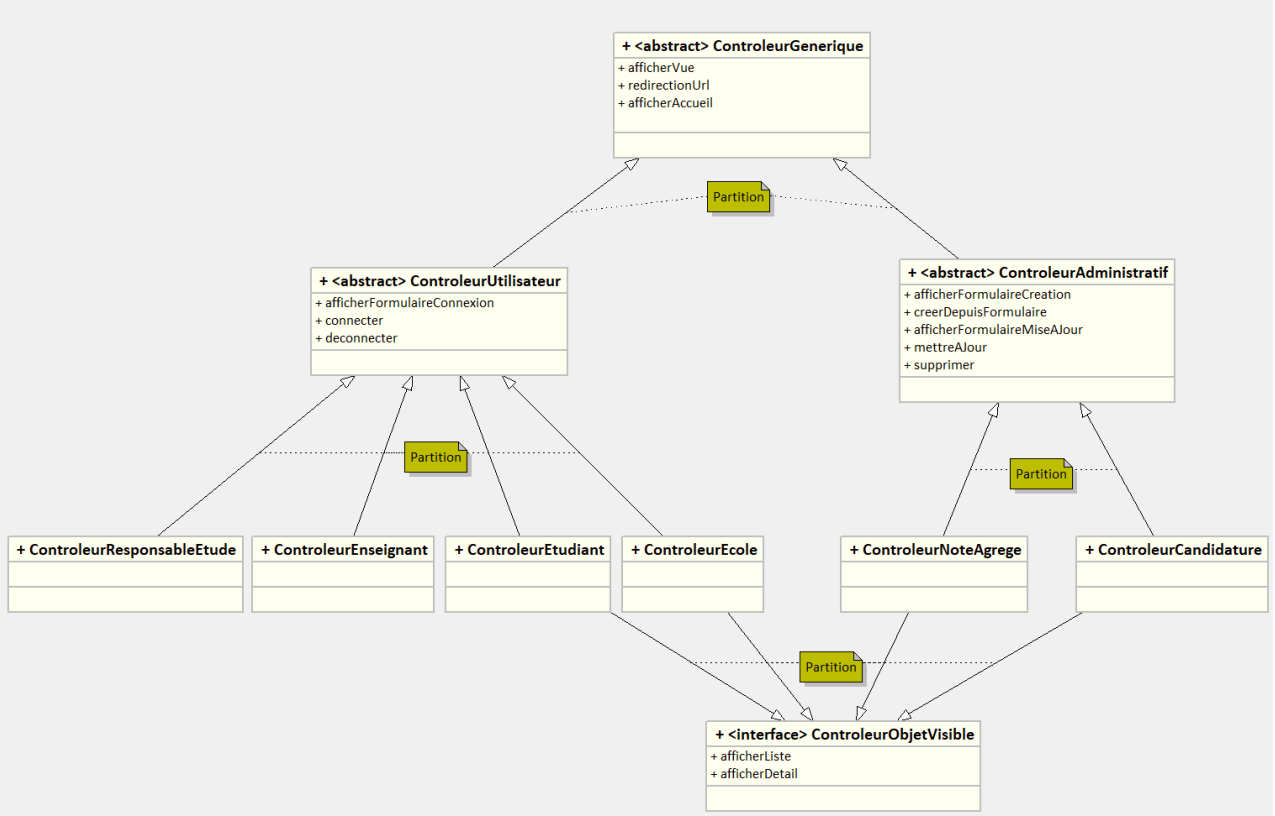
Diagramme de classe des contrôleurs au sprint 2 :



Commit associé

Durant le développement, d'autres contraintes sont intervenues, la structure du code devait par la suite prendre en compte la connexion via LDAP, et le fait que le principe CRUD ne devait désormais plus être utilisé sur aucun utilisateur.

Diagramme de classe des contrôleurs au sprint 4 :



Cette nouvelle structure permet de mieux respecter le principe Open/Close, de plus facilement s'étendre pour de nouvelles fonctionnalités. Comme la connexion via LDAP ne nécessite qu'un seul type de connexion pour tout type d'utilisateur et qu'un seul formulaire, il y a avec cette structure moins de redondance de code et plus d'abstractions.

Commit associé