COURS 3
Design Patterns

Cours 3 :

Les Design Patterns

Définition

- Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet.
- Un patron de conception décrit la solution éprouvée pour résoudre ce problème d'architecture de logiciel.

  - Exemple: Comme problème récurrent on trouve la conception d'une application où il sera facile d'ajouter des fonctionnalités à une classe sans la modifier.

NB: la conception des Design Patterns est indépendante des langages de programmation utilisés.

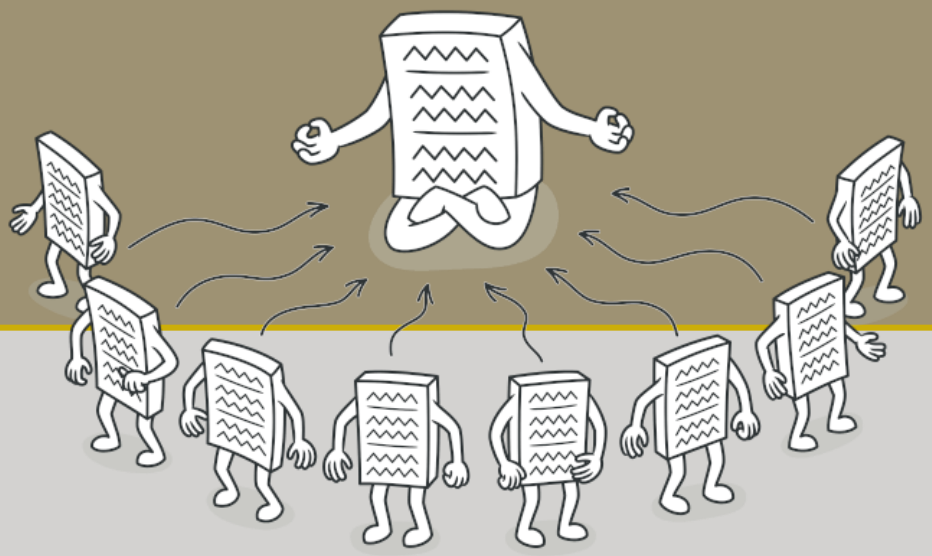
Avantages

- Permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts.
  - on gagne en rapidité et en qualité de conception ce qui diminue également les coûts.
- les Design Patterns sont réutilisables et permettent de mettre en avant les bonnes pratiques de conception.
- Les Design Patterns sont largement documentés et connus d'un grand nombre de développeurs.
  - Ils permettent également de faciliter la communication.
  - Si un développeur annonce que sur ce point du projet il va utiliser un Design Pattern, il est compris des informaticiens sans pour autant rentrer dans les détails de la conception (diagramme UML, objectif visé...).

Types des DP

- Les patrons de conception peuvent être classés en 3 types:
  - DP de création (creational): Ces modèles sont conçus pour atteindre certains objectifs ou critère concernant l'instanciation de classe.
  - DP de structuration (structural): Ces modèles sont conçus en ce qui concerne la structure et la composition d'une classe. Ces patterns concernent la manière d'organiser les classes et objets pour former des structures complexes tout en maintenant leur flexibilité et efficacité.
  - DP Comportementaux (Behavioral): Ces patterns se concentrent sur les interactions entre objets et la manière dont ils se comportent.

DP 1 : Singleton



Les Design Patterns

Singleton

- **La problématique**

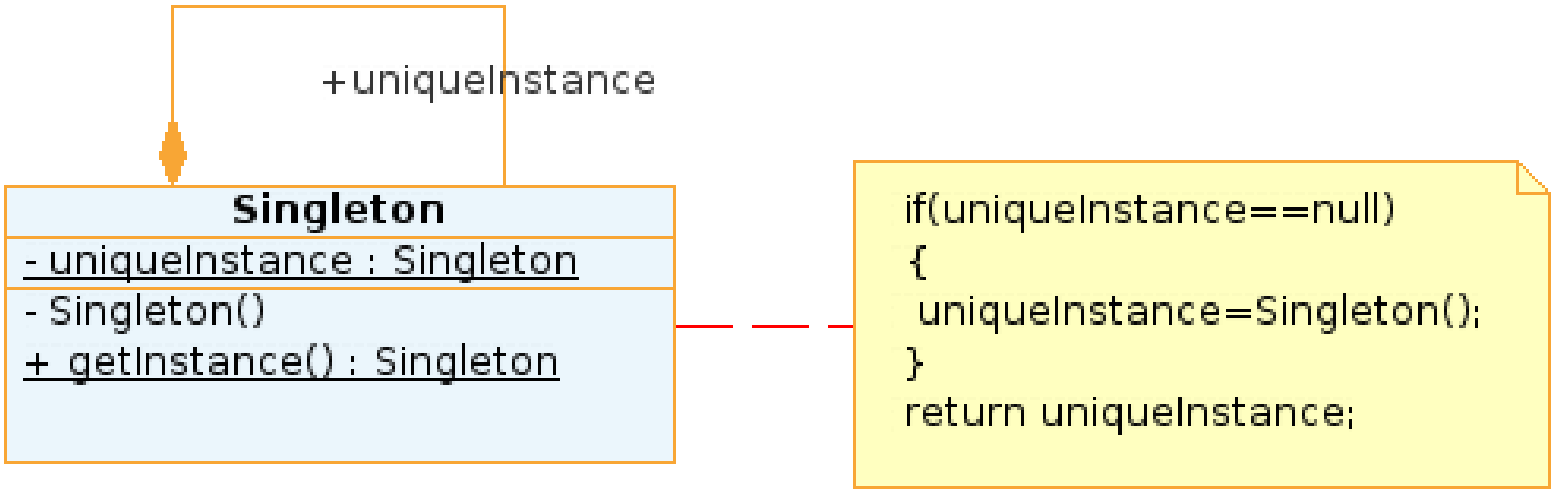
S'assurer que seule et une seule instance d'une classe soit utilisée.

- **La solution**

Rendre le constructeur de la classe privé puis créer une méthode statique dont la charge est de toujours fournir le même objet.

Singleton

Diagramme UML:



Singleton (Exemple)

```php
class singleton
{

    private static $_instance = null;

    private function __construct() {
    }

    /**
     * Méthode qui crée l'unique instance de la classe
     * si elle n'existe pas encore puis la retourne.
     *
     * @param void
     * @return Singleton
     */
    public static function getInstance() {

        if(is_null(self::$_instance)) {
            self::$_instance = new Singleton();
        }

        return self::$_instance;
    }
}
```

Singleton (Exemple)

```php
require(dirname(__FILE__).'/singleton.php');

// Tentative d'instanciation de la classe
$oSingletonA = Singleton::getInstance();
$oSingletonB = Singleton::getInstance();

var_dump($oSingletonA);
var_dump($oSingletonB);
```

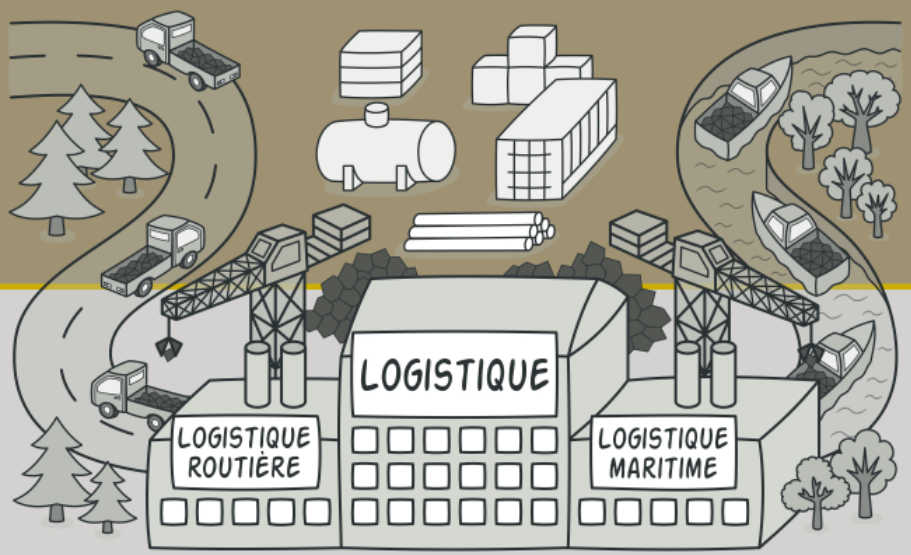
NB => Ajout de la méthode __clone () privée pour éviter le clonage

Singleton

Exercice:

1. Créer une classe config de type singleton permettant de charger la configuration de la BD à partir d'un fichier et de retourner une instance de PDO prête à utiliser.
2. Vérifier si la méthode getInstance() retourne toujours la même instance.
3. Comparer le résultat avec le cas sans singleton.
4. Cloner l'instance de la classe config et vérifier si ça retourne la même instance.
5. rendre la méthode clone privée pour empêcher le clonage.
6. vérifier si le clonage est possible.

DP 2 : Factory Pattern



Les Design Patterns

Factory Pattern

- **problématique:**

L'objectif du modèle de conception *facrtory ou Fabrique* est de fournir un objet prêt à l'emploi, configuré correctement, en libérant le code client de toute responsabilité (choix de l'implémentation, configuration, instanciation, …).

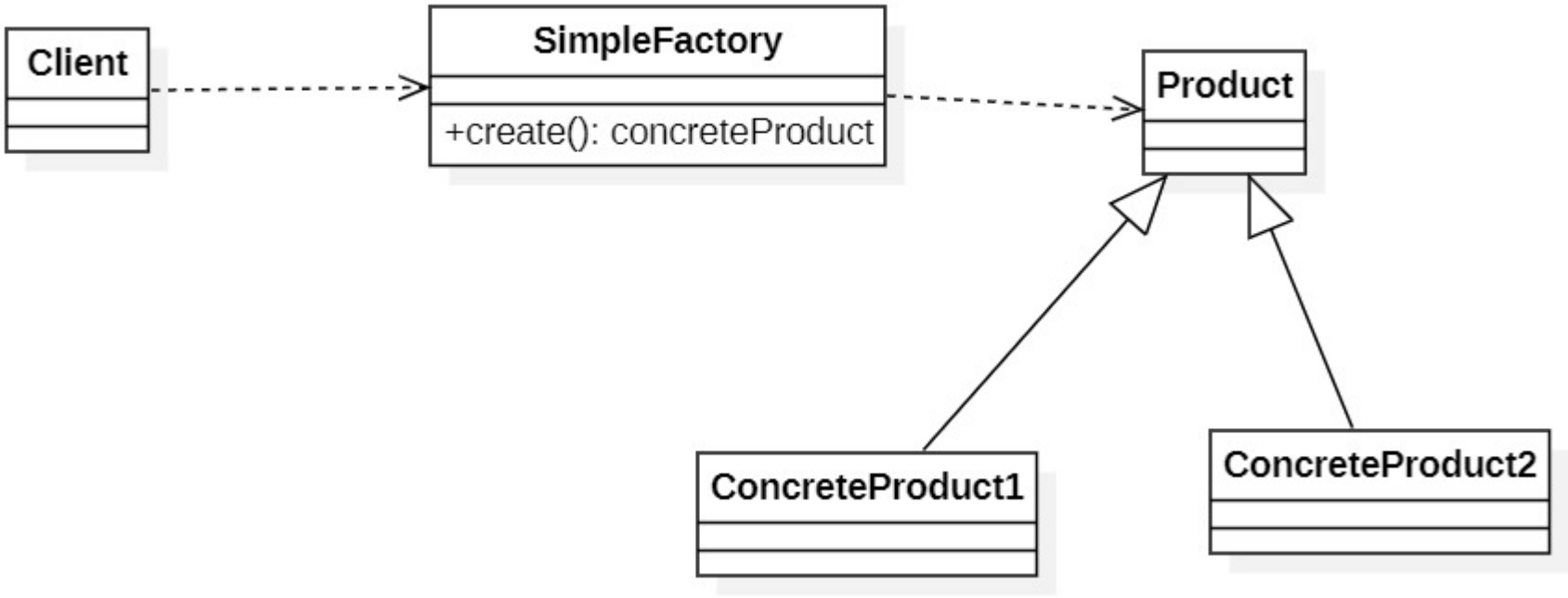
- **Dans quels cas ?**

Le modèle de conception *Fabrique* est intéressant dans les cas suivants :

- Vous avez plusieurs objets respectant la même *interface* ou héritant de la même *classe abstraite*, mais chacun adapté à un contexte différent (par exemple une interface IDB avec une classe par base de données DBMySql, DBOci, DBPgSql, …)
- Vous voulez centraliser le code en charge du choix de l'objet à créer.

Factory Pattern

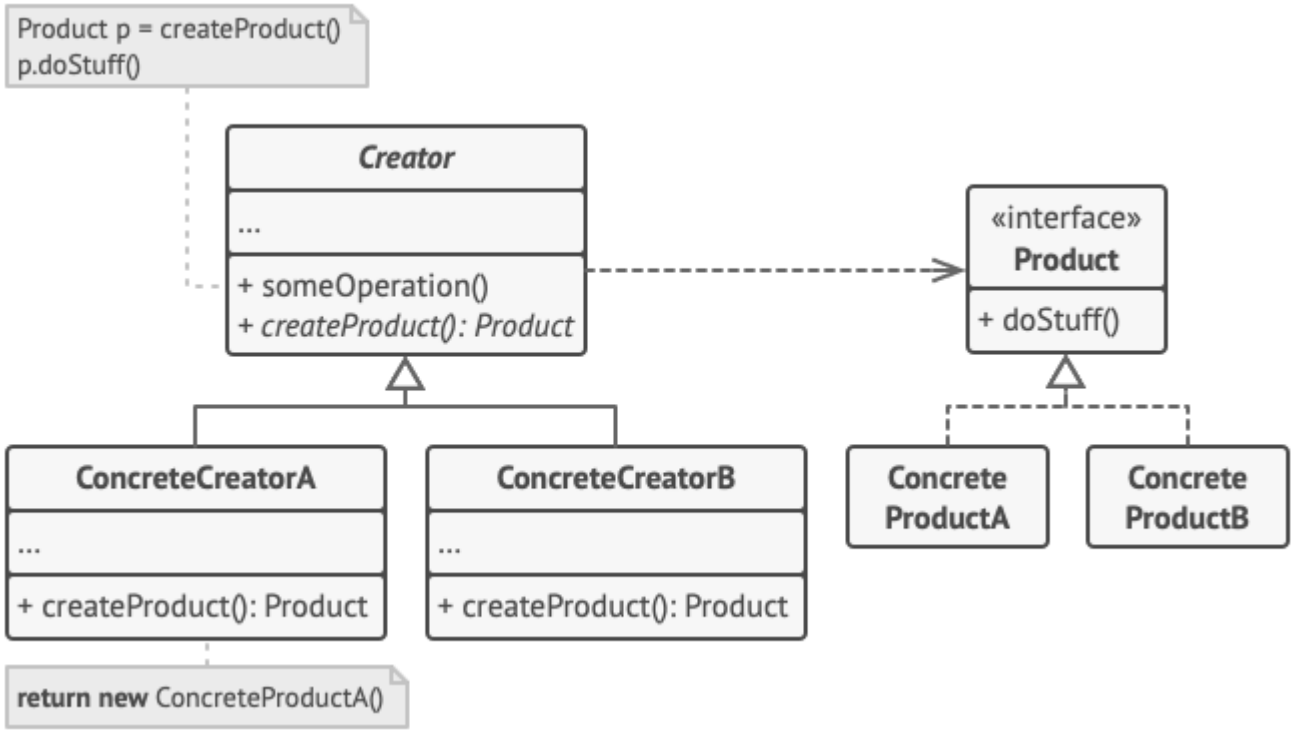
- **Diagramme UML:**



Factory Pattern

- **Diagramme UML (une autre version):**

Propose une famille de créateurs des produits:



Factory Pattern

Supposons que vous avez créé l'interface IDB suivante :

```php
interface IDB {
/**
* Connecte l'objet à la base de données
*/
public function connect ($connectionString);
/**
* Exécute une requête à la base de données
*/
public function doQuery ($queryString );
//.... suivent d'autres méthodes inutiles dans l'exemple
}
```

Dans votre application, vous utilisez plusieurs types de base de données, des bases MySql et des bases Oracles. De ce fait, vous développez deux classes spécifiques pour ces bases, respectant l'interface IDB:

```php
class DBMysql implements IDB {
//Implémentation spécifique à MySql
}
class DBOracle implements IDB {
//Implémentation spécifique à Oracle
}
```

Factory Pattern

Utilisation des objets sans Factory Pattern

Si vous n'utilisez pas de FP, chaque fois que vous allez vouloir instancier une connexion, vous allez devoir choisir vous même l'objet à instancier.

```
1  $db = new DBMysql ('mysql://user:password@localhost');
2  //plus loin
3  $db2 = new DBOracle ('oracle://user:password@localhost')
```

L'inconvénient est clair : si on change de type de base pour une partie de votre code, vous devrez convertir les instanciations de DBOracle vers DBMysql ou vice versa.

Un autre inconvénient très important est si le paramètre pris par l'instance est variable, ça sera compliqué de savoir quelle classe doit servir.

Factory Pattern

Pour éviter cela, on peut aussi procéder de la sorte :

```php
//quelquepart dans le code, on définit le paramètre de connexion
$connectionString1 = 'mysql://user:password@localhost';

//plus loin (en supposant que la chaine de connexion est valide)
switch (substr ($connectionString1, 0, strpos ($connectionString1, ':'))){
   case 'mysql':
      $db = new DBMySql ($connectionString1);
      break;
   case 'oracle':
      $db = new DBOracle ($connectionString1);
      break;
   default:
      throw new Exception ('Type de base inconnu');
}
```

Si l'avantage est de permettre d'appréhender le possible changement de base de données, il reste plusieurs inconvénients :

- Il faut copier/coller le code concerné de partout ou on souhaite créer une connexion.
- Si demain on veut intégrer un nouveau type de base de données, on doit modifier ce code partout ou il est dupliqué.

Factory Pattern

La solution avec le modèle de conception Factory

```php
class DBFactory {
    public static function create ($connectionString){
        if (($driverEndPos = strpos ($connectionString, ':')) === false){
            throw new Exception ('Mauvaise chaine de connexion');
        }

        switch (substr ($connectionString, 0, $driverEndPos)){
            case 'mysql':
                $db = new DBMySql ($connectionString1);
                break;
            case 'oracle':
                $db = new DBOracle ($connectionString1);
                break;
            default:
                throw new Exception ('Type de base inconnu');
        }
        return $db;
    }
}
```

Factory Pattern

le code client précédent est grandement allégé, avec par exemple :

```
$db1 = DBFactory::create ('mysql:dbname=blog;host=localhost', 'root', '');
$db2 = DBFactory::create ('oci:dbname=//host/service.com', 'user', 'password');

$query = $db1->doQuery('select * from article');
```

De plus, si un nouveau type de base de données est ajouté, seule la fabrique sera à adapter.

Factory Pattern

Exercice:

On vise à proposer une classe Faker qui permet de construire des objets avec des valeurs aléatoires. Ces objets peuvent avoir des instances soit en français ou en arabe. On a comme objectif de permettre la création de ces objets prêts à utiliser dans une classe client sans avoir besoin de manipuler les classes spécialisées dans le code client.

1. Montrer comment Factory pattern permettra un code plus optimisé.
2. Construire les classes selon la nouvelle conception.