BANE Mamadou

F2

DM3 Architecture Logicielle et Qualité

Présentation des patrons Singleton et Factory

Lorsqu'on a plusieurs classes liées entre elles par héritage, en dépit de la puissance de l'héritage, celui-ci atteint sa limite lorsqu'on est amené à modifier la hiérarchie des classes. Ces modifications sont pour la plupart des cas des sources d'erreurs dans le programme ou des absurdités dans son fonctionnement. Pour pallier ce problème, on utilise un design pattern (ou patron de conception en français).

Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet. Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel. Comme problème récurrent on trouve par exemple la conception d'une application où il sera facile d'ajouter des fonctionnalités à une classe sans la modifier. A noter qu'en se plaçant au niveau de la conception les Design Patterns sont indépendants des langages de programmation utilisés.[i]

Nous allons présenter les patrons Singleton et Factory tout en précisant leurs champs d'applications et leurs avantages.

1- Singleton

Le pattern Singleton est utilisé pour garantir qu'une classe n'ait qu'une seule instance et fournit un point d'accès global à cette instance.

Problématique :

Certaines applications possèdent des classes qui doivent être instanciées qu'une seule fois. C'est par exemple le cas d'une classe qui implémenterait la connexion à une base de données. En effet, instancier deux fois une classe servant la connexion à la base de données provoquerait une surcharge inutile du programme du système ou des comportements incohérents.

On peut alors se demander comment créer une classe, utilisée plusieurs fois au sein de la même application, qui ne pourra être instancié qu'une seule fois ?

Pour répondre à cette problématique, une première idée qui nous vient en tête est d'instancier la classe dès le lancement de l'application dans une variable globale, accessible depuis n'importe quel emplacement du programme. Cependant cette solution doit être évitée car en plus d'enfreindre le principe d'encapsulation elle présente de nombreux inconvénients. En effet, rien ne garantit qu'un développeur n'instanciera pas une deuxième fois la classe à la place d'utiliser la variable globale définie. De plus, on est obligé d'instancier les variables globales dès le lancement de l'application et non à la demande (ce qui peut avoir un impact non négligeable sur la performance de l'application). Enfin, lorsqu'on arrive à plusieurs

centaines de variables globales le développement devient rapidement ingérable surtout si plusieurs programmeurs travails simultanément.[ii]

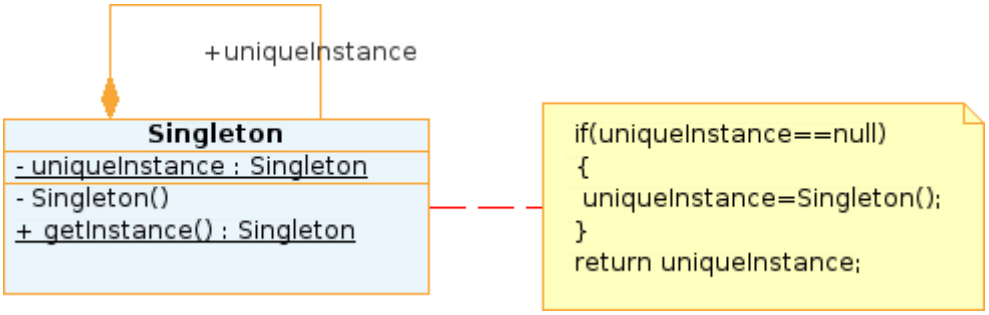
Solution :

Pour résoudre ce problème d'instanciation unique, on utilise le design pattern Singleton.

L'objectif est d'avoir qu'une seule instance sur laquelle on applique ce patron. Pour cela on applique un contrôle sur le nombre d'instance de la classe. Ainsi pour restreindre l'instanciation de la classe, on empêche les utilisateurs de la classe d'utiliser le ou les constructeurs de la classe pour l'instancier, donc les constructeurs seront déclarés privés. Pour instancier la classe, on passera par une méthode statique qui nous fournira une instance de la classe.

Nous allons construire un pseudo constructeur. Pour cela il faut déclarer une méthode statique qui retournera un objet correspondant au type de la classe. L'avantage de cette méthode par rapport à un constructeur, est que l'on peut contrôler la valeur que l'on va retourner. Le fait que cette méthode soit déclarée statique permet de l'appeler sans posséder d'instance de cette classe. A noter que, par convention, ce pseudo constructeur est nommé getInstance.[iii]

La figure suivante représente le diagramme de classe UML d'une classe qui implémente le patron Singleton.



2- Factory

Le design pattern Fabrique (Factory) définit une interface pour la création d'un objet en déléguant à ses sous-classes le choix des classes à instancier.

Problématique :

Ce pattern est utilisé pour permettre à une classe d'instancier différents types d'objets en fonction du paramètre fourni.

Par exemple une usine va fabriquer des produits en fonction du modèle qu'on lui indique. L'idée la plus simple pour répondre à ce besoin est d'écrire une succession de conditions qui suivant le modèle demandé, instancie et retourne l'objet correspondant.

Le problème avec cette implémentation, c'est que la classe correspondant à l'usine va être fortement couplée à tous les produits qu'elle peut instancier car elle fait appel à leur type concret.

Or ce code va être amené à évoluer régulièrement lors de l'ajout de nouveaux produits à fabriquer ou de la suppression de certains produits obsolètes.

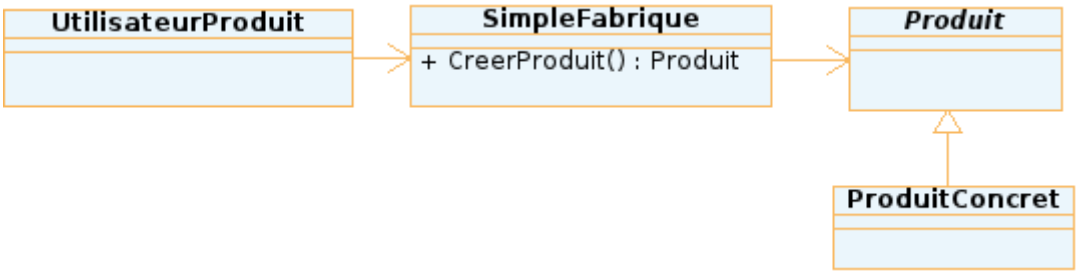
De plus, il est fort probable que l'instanciation des différents produits soit également réalisée dans d'autres classes par exemple pour présenter un catalogue des produits fabriqués.

On se retrouve alors avec du code fortement couplé, qui risque d'être dupliqué à plusieurs endroits de l'application.[iv]

Solution :

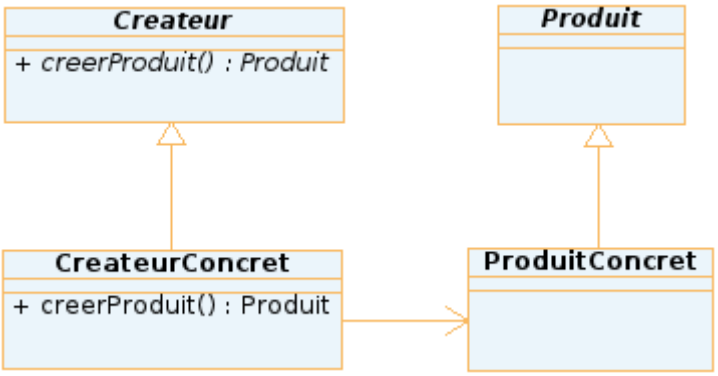
Pour résoudre ce problème, il faut déléguer une classe qui regroupera l'instanciation de tous les produits. Cela permettra d'éviter la duplication du code.

Le diagramme de classe ci-dessous représente l'utilisation de la Fabrique Simple.



L'utilisateur du produit fait appel à la Fabrique Simple pour obtenir un Produit. C'est la Fabrique Simple qui est chargée d'instancier et de retourner le Produit Concret attendu (par exemple grâce à un paramètre passé en argument de la fonction). L'utilisateur du produit est donc fortement couplé uniquement à la Fabrique Simple et non à tous les produits qu'il prend en charge.

Si par la suite l'entreprise évolue et a besoin de plusieurs usines, chacune spécialisée dans la fabrication de certains produits, Fabrique Simple ne va plus suffire. Dans ce cas il faut prévoir l'utilisation du design pattern Fabrique dont le diagramme UML est présenté ci-dessous.



[i] http://design-patterns.fr/introduction-aux-design-patterns consulté le 03/02/2019
[ii] http://design-patterns.fr/singleton consulté le 03/02/2019
[iii] http://design-patterns.fr/singleton consulté le 03/02/2019
[iv] http://design-patterns.fr/fabrique consulté le 03/02/2019