Design Pattern – TD n°2

Factory et Factory Builder

Dans ce second TD, nous allons voir comment fonctionne le pattern Factory. Nous reviendrons aussi sur le Video Club de l'exercice 1, et tenterons de construire une partie du Video Club à l'aide du pattern Factory.

LE PATTERN FACTORY

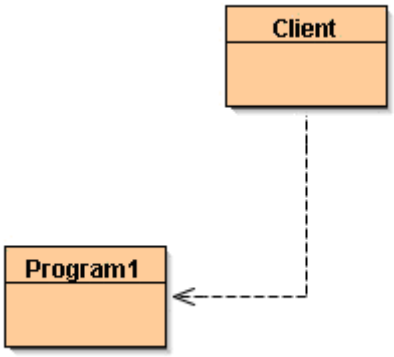
Le but ici est d'implémenter le modèle *Factory* avec un exemple simple, afin d'être sûr d'avoir compris ce Pattern.

Dans ce projet, on trouve deux classes : une classe *Program1* qui se contente, lorsqu'on appelle go, d'afficher un message (dans la réalité, un traitement particulier aurait lieu), et une classe *Client* qui appelle *Program1*.

```
public class Client
{
    public static void main1()
    {
        Program1 p = new Program1();
        System.out.println("Je suis le main1");
        p.go();
    }

    public static void main2()
    {
        Program1 p = new Program1();
        System.out.println("Je suis le main2");
        p.go();
    }

    public static void main3()
    {
        Program1 p = new Program1();
        System.out.println("Je suis le main3");
        p.go();
    }
}
```

```
public class Program1
{
    public Program1()
    {
       // Le constructeur ne fait rien
    }

    public void go()
    {
       System.out.println("Je suis le traitement 1");
    }
}
```

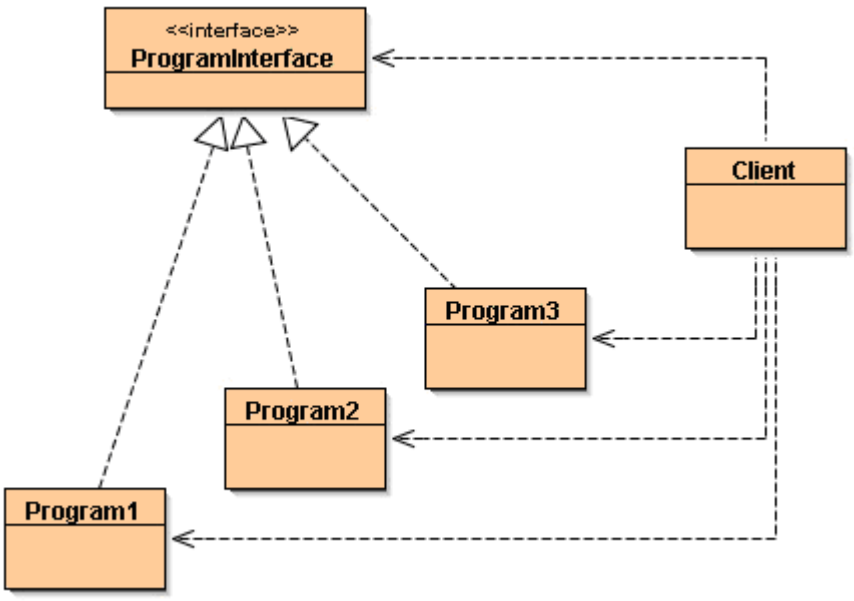
Le diagramme de classe ressemble à ceci :



1. On souhaite rajouter deux classes *Program2* et *Program3* à notre projet. Ces classes afficheront à l'écran le même message que *Program1*, excepté qu'elles y feront apparaître leur numéro de programme (2 ou 3). Ajoutez ces classes (en dupliquant et modifiant le code de *Program1*).

Dans le client, on souhaite modifier le code des fonctions main pour que, selon le paramètre entier qui leur est passé en argument (1, 2, ou 3), ces dernières lancent le traitement du *Program1*, *Program2* ou *Program3*. Pouvez-vous réaliser cette fonctionnalité ?

Voici, en codant de la bonne manière, le diagramme de classe que vous devriez obtenir:



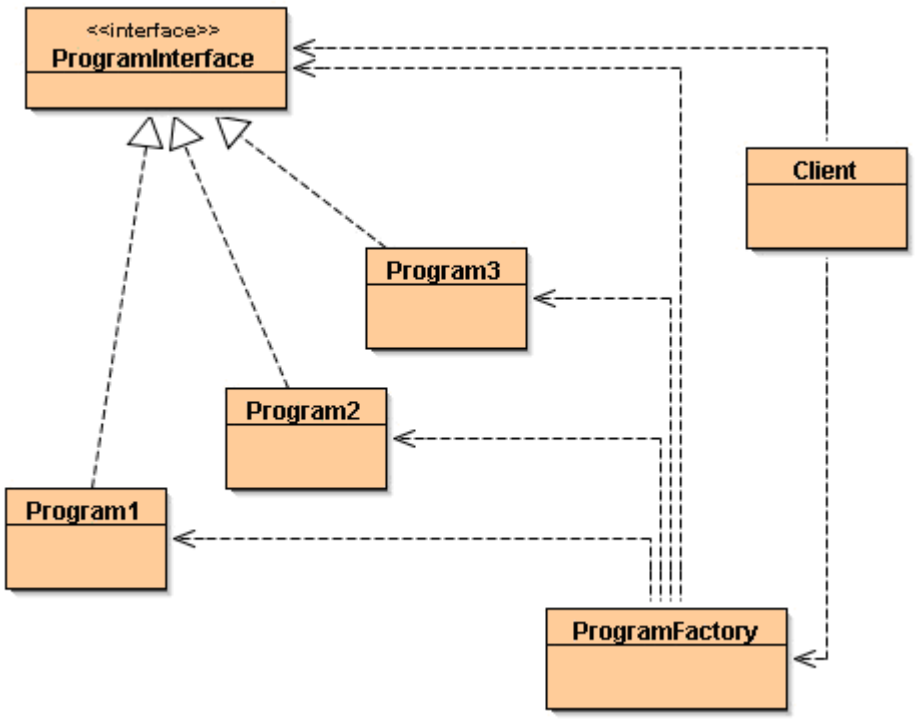
2. En Java, la duplication de code n'est pas souhaitable. Avez-vous, dans les trois fonctions main de client, dupliqué le code de création des objets *Program* ? Si c'est le cas, changez votre classe et créez-y une fonction, appelée par les trois main, qui s'occupera de construire les objets *Program*.

3. Jusque là, rien de bien nouveau dans ce que l'on a fait. Le problème ici est que les objets *Program1*, *Program2* et *Program3* sont construits dans la fonction main de *Client*, et ceci n'est pas souhaitable. On préfèrera déléguer les détails de la construction à une classe spécifique : la Factory.

En effet, s'il y avait plusieurs classes *Client*, chacune aurait une fonction de création des objets *Program*, et si l'on voulait modifier la façon dont les *Program* sont générés, il faudrait modifier chacune des fonctions de création des *Program* dans les classes *Client*.

Pour éviter ça, on va déléguer la création des objets de type *Program* à une classe dont le nom sera *ProgramFactory*. Les clients devront construire une instance de cette classe afin de pouvoir y utiliser la fonction de création des *Program* qui y sera stockée.

Voici le diagramme de classe que vous devriez obtenir :



On voit bien sur le diagramme de classe que le *Client* n'a plus connaissance de l'existence de *Program1*, *Program2* ou *Program3*. Il se contente de voir l'interface *ProgramInterface* et le *ProgramFactory*.

4. Pouvez-vous ajouter un *Program4* ? Cela a-t-il été compliqué à mettre en place ? Avez-vous dû modifier le code du *Client* ?

Vous venez d'implémenter le **Pattern Factory**. Il permet au client de ne pas avoir connaissance du produit (*Program1*, *2*, *3* ou *4*), et de déléguer les détails de la production à une autre classe. Si on souhaite rajouter un produit (un *Program*), on doit juste éditer la factory. Grâce à la factory, on évite à la classe *Client* d'avoir connaissance des différentes instances de *ProgramInterface*.

LE PATTERN ABSTRACT FACTORY

Ici, on va utiliser le Pattern Factory dans un projet avec un véritable but, puis passer au Pattern Abstract Factory afin d'étendre les possibilités de votre programme sans avoir à modifier trop de code.

Vous venez d'arriver dans votre nouvelle entreprise, et vous récupérez un programme qui avait été mise en place par votre prédécesseur. Ce programme permet, étant donné un chemin complet vers un fichier Windows, d'afficher juste le nom du fichier.

Voici le code de la seule classe ce projet :

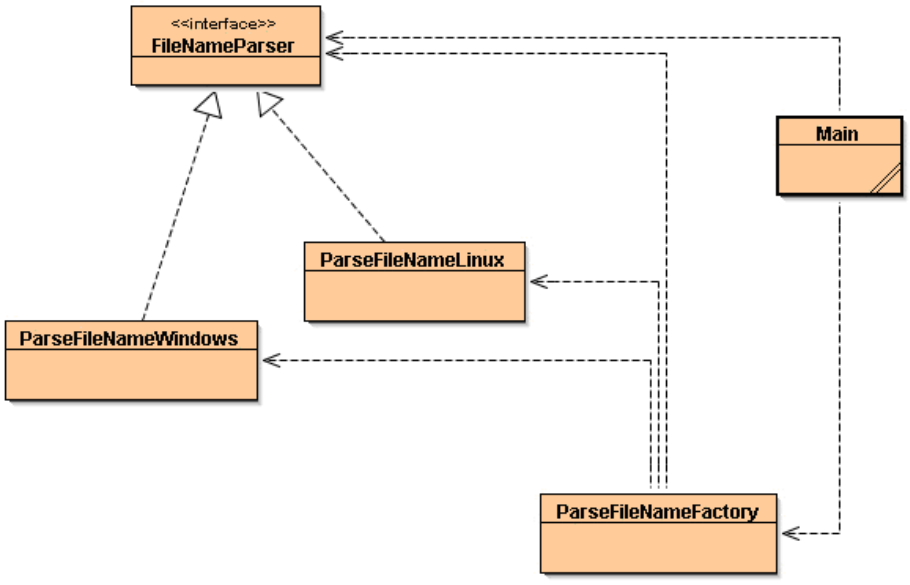
```
public class Main
{
    public static void main_parse_filename(String path)
    {
       //index est l'endroit où se situe, dans la String path, la dernière
       //apparition du caractère \
       int index = path.lastIndexOf("\\");
       //On construit une String qui ne contient que la partie située à droite
       //du dernier caractère \
       String r = path.substring(index+1);

       System.out.println(r);
    }
}
```

Si vous testez ce programme en lui passant en paramètre « C:\\Windows\\hello.dll », il devrait vous renvoyer « hello.dll ».

1. Avec le temps, on souhaite faire évoluer les possibilités de ce programme. On va ajouter la possibilité de trouver, étant donné un chemin vers un fichier Linux (de type « /user/share/hello.rc »), le nom du fichier en question.

Pour ce faire, on procède comme à la question précédente : on va tout d'abord créer une interface *FileNameParser*, qu'interfaceront deux sous classes *ParseFileNameWindows* et *ParseFileNameLinux*. Ensuite, on créera une Factory qui permettra de créer des *FileParser* dans notre main.

Votre diagramme de classe devra ressembler à ceci :



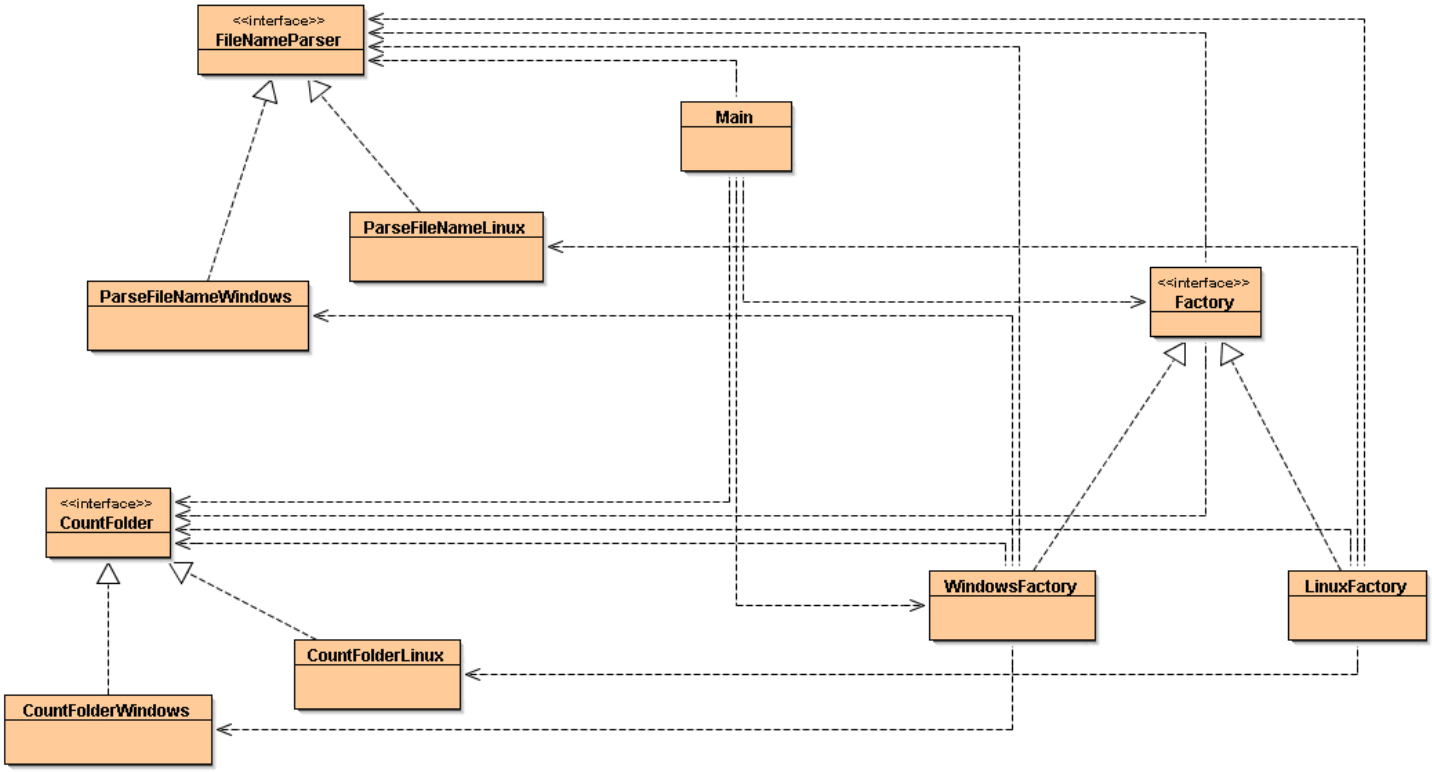
2. On souhaite maintenant ajouter des fonctionnalités, et on veut maintenant compter le nombre de dossiers qu'il faut parcourir depuis la racine pour trouver notre fichier. On va ajouter une interface *CountFolders*, et deux sous classes qui l'implémenteront (une windows, et une linux) (jetez un œil à ce que count peut faire).

Ensuite, il va falloir dire à la factory de créer à chaque fois deux types d'objets (un *FileNameParser* et un *CountFolders*) qui seront rangés dans des variables de classe de la factory.

3. Le problème de cette implémentation est que, si on veut encore ajouter une fonctionnalité, il faudra aller changer le code de la factory (où le code va vite grandir). On voudrait en fait, si quelqu'un veut ajouter des fonctionnalités, ne pas le forcer à ouvrir notre factory et avoir à comprendre tout ce qui s'y passe pour savoir où ranger son code.

On va donc faire de notre factory une interface et créer deux sous factory qui l'implémentent (une Linux et une Windows). Il faudra cependant, dans la classe *Main*, choisir explicitement quelle factory on veut utiliser.

Votre diagramme de classe devrait ressembler à ceci :



4. Pour éviter que, dans la classe *Main*, on ait à choisir explicitement quelle factory implémenter, et pour éviter que le *Main* « ait connaissance » de toutes nos différents factory, que va-t-on faire ? Et oui, une factory de factory !

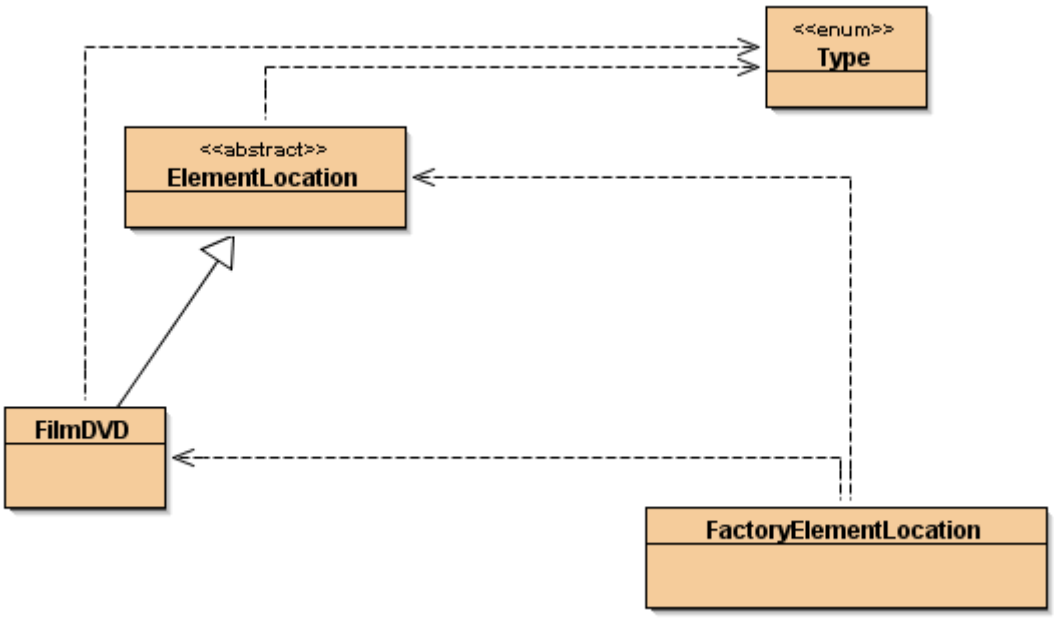
5. On veut ajouter le support des chemins Macintosh (reconnaître les « / » ou les « : »). Faîtes les modifications nécessaires.

6. On souhaite ajouter la fonctionnalité de donner le nom du répertoire source qui contient notre fichier. Faîtes les modifications nécessaires.

Vous venez d'implémenter le Pattern Abstract Factory. Il fonctionne à peu près comme Factory, sauf qu'ici, on souhaite créer non pas un seul produit, mais des groupes de produits (ici, les *FIleNameParser* et *CountFolder*). La Factory est abstraite et différentes sous Factory permettent de produire différents groupes de produits.

FACTORY ET LE VIDÉO CLUB

Tentez d'écrire une partie du code du Vidéo Club (voir TP n°1) en vous servant du Pattern Factory. Ici, on souhaite gérer, avec une factory, la création des éléments que l'on peut louer. Le diagramme de classe à obtenir à la fin est celui-ci :



Dans *ElementLocation*, vous aurez le type de l'objet, son prix, son titre, le nombre de copies totales, un type (qui provient d'un enum), le nombre de copies disponibles, une liste de numéros d'identité permettant d'identifier toutes les copies de l'élément en question) et une liste de numéros d'identité permettant d'identifier toutes les copies disponibles. l'enum permettra d'éviter, dans des classes extérieures au processus de création, d'avoir besoin de connaître les sous classes de ElementLocation : tout se fera à travers l'enum.

Dans *FilmsDVD*, vous ajouterez d'autres informations, comme la durée du film, les langues disponibles, et un résumé du film.

Vous ajouterez ensuite *FilmBluray* (avec un champ pour savoir si le film est en 3d), *JeuxVideo* (avec un champ pour connaître le nombre de joueurs et un champ pour connaître la console de jeux). Vous pouvez ajouter, pour les langues ainsi que pour les consoles, des enum. Faîtes évoluer votre factory en conséquent.