Design Pattern – TD n°6

LE PATTERN PROXY : CHANGER UN COMPORTEMENT SANS CHANGER L'OBJET

Vous êtes de retour au club Jamba (pas de chance...). Et maintenant, après le bon travail réalisé sur René la Taupe, vous êtes placé sur un projet prioritaire : le Jamba Album Cover Generator. Ce petit programme permet de télécharger, sur son mobile, les pochettes des CDs les plus cools du marché (moyennant finance) !

1. Récupérez le code de l'application ici. Lancez le main, vous devriez voir apparaître une fenêtre avec un menu, permettant de sélectionner l'album à afficher. Essayez d'afficher un album.

Vous l'avez compris, Henri (votre prédécesseur) n'a pas très bien codé l'application. En effet, les ~~petits adolescents boutonneux~~ clients du club Jamba ne sont pas toujours patients. Or, du temps est nécessaire pour télécharger la couverture de l'album choisi. Si l'on clique plusieurs fois sur le bouton d'affichage, beaucoup de fenêtres apparaissent.

*Pour éviter ça, il faudrait modifier un peu de code MAIS problème : les classes **Main**, **FenetrePrincipale** et **RecupererAlbum** sont considérées comme déjà téléchargées par le client (on ne peut pas y toucher, à moins de demander au client de faire une mise à jour du programme, chose que l'on évitera de lui proposer) et la classe **AfficheurAlbum** est sur un serveur extérieur, que le club Jamba loue, et sur lequel il faut payer pour modifier quoi que ce soit.*

2. Vous n'avez le droit que de rajouter des classes, ou modifier **FactoryRecuperationAlbum**. L'idée serait d'afficher au client une image d'attente (une image est présente dans le dossier img) en attendant qu'il récupère son image.

Pour ce faire, voici quelque indices :

. Vous devrez créer une classe qui sera notre proxy : elle implémentera **RecuperationAlbum**, se fera passer auprès du client pour un **RecuperationAlbum**. La fonction **afficheralbum()**, appelée par le client pour afficher la pochette, pourra ainsi être modifiée.

. Le proxy devra tout de même appeler certaines fonctions d'**AfficheurAlbum** : il faut simplement ajouter des fonctionnalités (affichage d'une image d'attente) à cette fonction.

. Dans **AfficheurAlbum**, vous avez des fonctions pour récupérer l'image, et non l'afficher.

. Vous devrez peut être modifier aussi **FactoryRecuperationAlbum**.

. Dans **MyImage**, la fonction **replacewith()** permet de modifier une image déjà affichée à l'écran.

3. Testez votre application : il NE FAUT PAS que, lorsqu'on clique sur le bouton de sélection de pochette d'albums, ce dernier reste bloqué en position cliquée. Si c'est le cas, inspirez vous de **AfficheurAlbum** pour résoudre ce problème (utilisez des threads).

4. Parfait, le programme fonctionne bien mais il y a encore un problème : que se passe-t-il si la personne appuie plusieurs fois sur le bouton pour afficher les albums ? Et oui, il va toujours s'afficher des dizaines de couvertures d'album. Modifiez votre proxy afin qu'il bloque la création en cascade de couvertures d'albums.

5. Question subsidiaire : grâce à quel pattern a-t-on évité la modification des classes « côté client » ?

LE PATTERN CHAÎNE DE RESPONSABILITÉ : GESTION DE PROCESSEURS

Le pattern chaîne de responsabilités permet d'exécuter, à la chaîne, des fonctions comme le Décorateur. Cependant, ici, le but n'est pas d'ajouter des fonctionnalités mais de déléguer des responsabilités : si on ne peut pas traiter la demande, on la passe à son voisin, sinon, on traite la commande et c'est fini (tandis que Décorateur consiste à traiter une commande ET passer au voisin). Nous allons implémenter ce pattern pour gérer une liste de processeurs.

1. Récupérez le code de départ ici. Il vous faut compléter le code de **MathCompute** (la fonction **run**) afin que ce dernier calcule la décomposition en facteurs premiers de la variable de classe **num** : il faut ajouter chaque facteur dans la liste **res**.

*Le reste du projet consiste en des processeurs lents et des processeurs rapides qui, au travers de la fonction **runTask** peuvent lancer le calcul de la décomposition en facteurs premiers d'un nombre. Si un calcul était déjà en cours sur le processeur, ce dernier attendra la fin du calcul avant de commencer un nouveau calcul. On peut tester si un processeur est occupé avec la fonction **isBusy**. Un objet de type **ResultPool** doit être créé dans le main et passé aux processeurs pour collecter les résultats de tous les calculs.*

*Lors du calcul de la décomposition, on utilise un **Thread.sleep** pour émuler le traitement, plus ou moins long, de la tâche par le processeur.*

2. Le but de ce TP sera de créer plusieurs processeur (trois rapides, deux lents) et de faire une boucle qui tire vingt nombres au hasard entre 1 et 100 000 afin de calculer leur décomposition en facteurs premiers. Évidemment, si on fait faire tout le travail par un seul et même processeur, on perdra du temps. Le but sera de gérer, de façon quasi transparente pour le main, la file des processeurs.

Vous devrez créer ces classes :

.La classe **QuickProcessorHandler** permettra de gérer les processeurs rapides (et contiendra donc un processeur p de ce type). Elle aura une variable de classe **nextprocessor** (et un setter sur cette variable) permettant de connaître quel **ProcessorHandler** (voir plus bas) appeler si le processeur p est occupé. La fonction **handlerequest(int n)** devra soit calculer la décomposition en facteur premier de n (grâce à p), ou bien passer le relai à **nextprocessor**.

.La classe **SlowProcessorHandler** fera de même.

.La classe **ProcessorHandler**, qui sera abstraite, sera une classe dont héritent les deux précédentes. Elle contiendra la fonction (abstraite ?) **handlerequest**. De plus, ce sera elle en vérité qui contiendra la variable **nextprocessor** et son setter.

Le but est que le main créé plusieurs **ProcessorHandler**, les lie entre eux, et lance la requête de calcul au premier élément de la file qui transmettra la requête s'il est occupé. Si tous les processeurs sont occupés, on abandonne la requête (en affichant un message). Pensez à

afficher un message à l'écran pour afficher, pour chaque nombre, quel processeur va le traiter. Votre fonction **handlerequest** de **ProcessorHandler** ne devrait pas être abstraite afin de justement implémenter un comportement par défaut (que faire par défaut si le processeur est occupé ?)...

3. Comment faire pour que, si tous les processeurs sont occupés, l'on boucle de nouveau sur le premier processeur de la file pour tenter de faire traiter la requête. Pensez, quand vous passez la requête d'un processeur à l'autre, à faire un petit **Thread.sleep** pour éviter que le main ne passe sont temps à boucler sur les processeurs.

4. Connaissez-vous un mécanisme connu dans Java qui ressemble à Chaîne de responsabilités ?

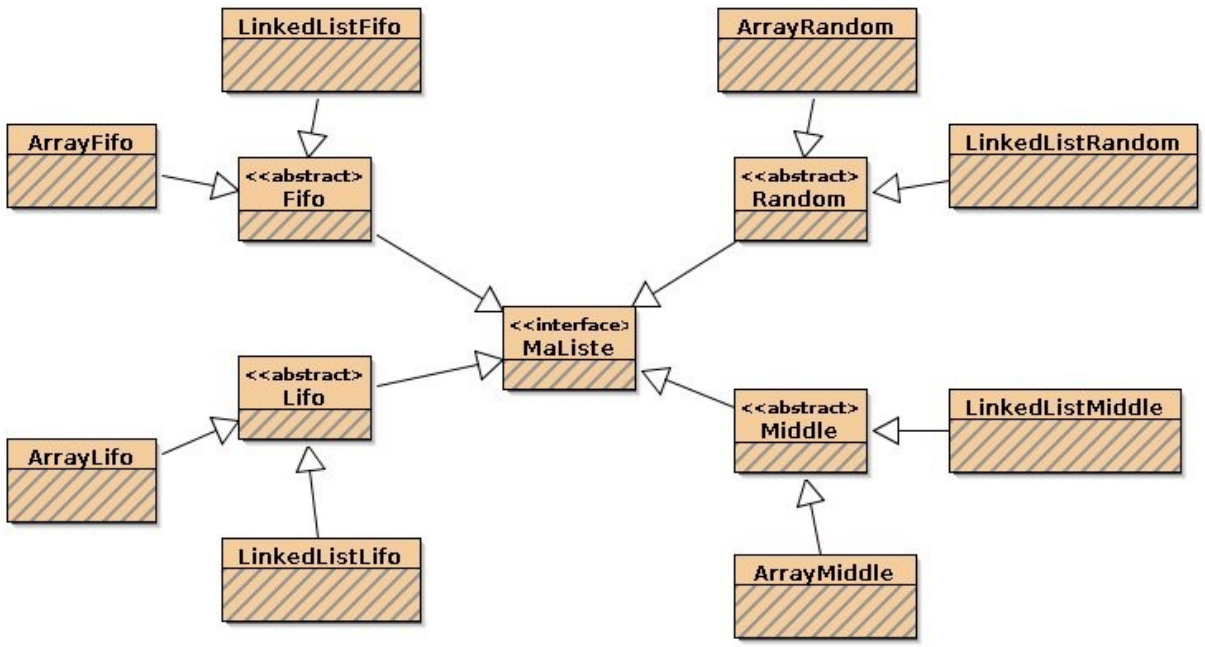
LE PATTERN BRIDGE : ÉVITER UNE EXPLOSION DU NOMBRE DE CLASSES

Rappelez-vous le TP n°2, avec l'Abstract Factory. On créait des CountFolderWindows, des ParseFileNameLinux, etc... On avait d'un côté des OS (Windows, Linux, …) et de l'autre côté, des fonctionnalités (compter le nombre de dossier, récupérer le nom de fichier). Si l'on souhaitait rajouter une seule fonctionnalité, on devait rajouter un nombre de classe égal au nombre d'OS à gérer, et si l'on souhaitait rajouter un OS, on devait rajouter un nombre de classe égal au nombre de fonctionnalités à implémenter. Bref, pas super.

On souhaite réaliser le projet suivant : créer une interface **MaListe** qui gère des ensemble d'entiers et qui contient trois fonctions : push, pop, et isEmpty.

Cette interface pourra être déclinée par des classes abstraites Lifo, Fifo, Random (qui affiche un nombre choisi au hasard dans l'ensemble quand on appelle pop) et Middle (qui affiche le nombre rangé au centre de la liste quand on appelle pop). De plus, chacun des ces listes pourra êtres implémentée à l'aide d'un tableau ou d'une système de liste chaînée.

Voici le schéma de classe du projet :



Après un séjour au club Jamba, vous avez appris à en faire le moins possible. Or, vous vous

rendez compte que, le jour où vous voudrez ajouter la gestion des ensembles avec des arbres binaires, il vous faudra rajouter 4 classes, et le jour ou vous souhaiterez ajouter un système de gestion d'ensemble qui affiche le sort à chaque fois le plus petit élément de la liste, il vous faudra rajouter deux classes. En résumé, si vous avez N abstractions et M implémentations, le jour où vous souhaitez rajouter une seule implémentation, vous devrez écrire N classes, et le jour où vous souhaitez rajouter une seule abstraction, vous devrez rajouter M classes.

Tentons de trouver mieux.

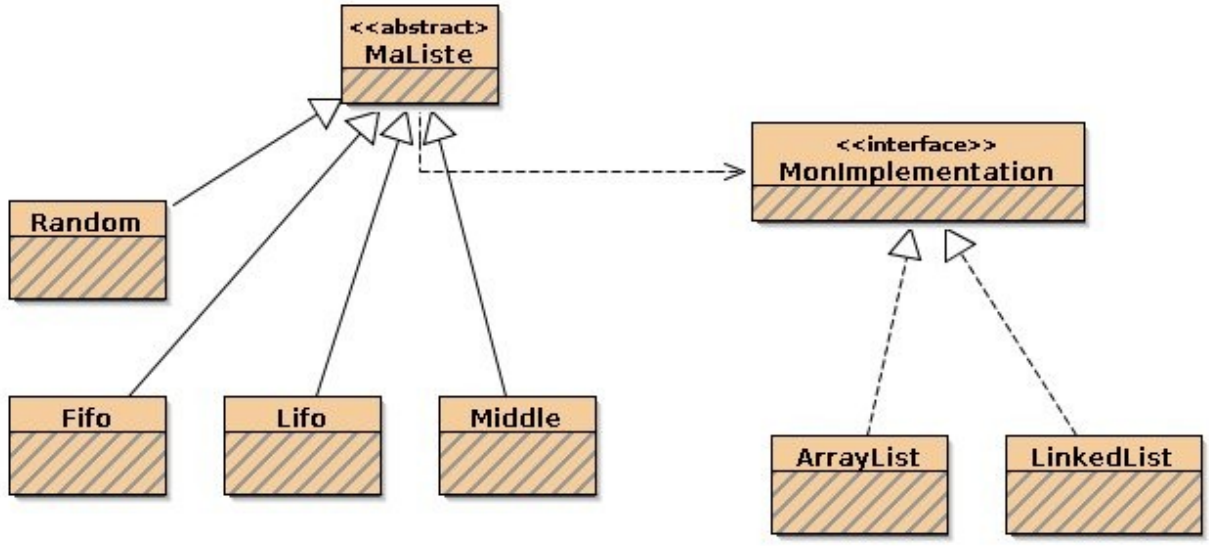
Dans la suite, vous devrez implémenter vos classes sans utiliser les classes List de Java (pas d'ArrayList, LinkedList, …).

Vous aurez besoin d'une interface MonImplementation, qui contiendra les méthodes que vous choisirez (du genre getsize, removeElementAt, addElementAt) et dont hériteront les classes ArrayList et LinkedList.

Vous aurez aussi besoin d'une classe abtsraite MaListe, qui contiendra un MonImplementation, des fonctions abstraites pop, push et isEmpty, et dont hériteront les classes Fifo, Lifo, Random et Middle.

En séparant les systèmes d'accès aux éléments (Lifo, Fifo, …) et les structures internes des listes (Array, LinkedList), on évite l'explosion de classe lorsque l'on souhaitera, plus tard, rajouter un élément.

Votre schéma de classe devra ressembler à ceci :



OBSERVATEUR ET OBSERVÉ

Téléchargez le code ici. Le but est de mettre en place une horloge qui se mette à jour toutes les secondes. On a d'un côté l'horloge (l'observé), et de l'autre côté, une ou plusieurs fenêtres (les observateurs). On souhaite faire communiquer ces différentes entités.

Pour ce faire, on va implémenter le pattern Observateur/Observé : l'horloge est un observé qui connait ses observateurs. A chaque fois que l'horloge change de valeur, elle ira sonner chez chacun de ses observateurs pour les mettre au courant de la nouvelle heure.

L'horloge devra implémenter l'interface suivante :

```
public interface Observable {
      public void addObservateur(Observateur obs);
      public void updateObservateur();
```

```
      public void delObservateur();
}
```

et la fenêtre devra implémenter l'interface suivante :

```
public interface Observateur {
      public void update(String hour);
}
```

Complétez les classes afin que l'affichage de l'horloge s'effectue correctement.

Cet exercice a été tiré du site www.siteduzero.com

ENCORE DU BUILDER...

1. Récupérez votre code d'exercice 2 du contrôle. Vous pouvez soit recommencer depuis le début le code de cet exercice, ou bien récupérer votre code. Le but est de voir si votre code s'adapte bien aux changements.

2. Petit changement : la fonction load3denvironments doit changer : pour Iphone, la fonction doit faire

```
operationsdone[5]=true;
```

et pour PC, elle doit faire

```
operationsdone[0]=false;
```

Récupérez la nouvelle fonction display de GraphicMultiPlatform ici :

```
public void display()
    {
        if( !operationsdone[0] && !operationsdone[1] && !operationsdone[2] &&
operationsdone[3] &&
            !operationsdone[4] && !operationsdone[5] && operationsdone[6] && !
operationsdone[7] &&
            !operationsdone[8] && operationsdone[9] && operationsdone[10] &&
operationsdone[11])
        {
            c.getImage().display();
            System.out.println("Load PC graphics ok");
        }
        else if(    !operationsdone[0] && operationsdone[1] && !
operationsdone[2] && !operationsdone[3] &&
                    operationsdone[4] && operationsdone[5] && operationsdone[6]
&& !operationsdone[7] &&
                    operationsdone[8] && !operationsdone[9] &&
operationsdone[10] && operationsdone[11])
        {
            c.getImage().display();
            System.out.println("Load IPhone graphics ok");
        }
        else if(    !operationsdone[0] && !operationsdone[1] &&
operationsdone[2] && !operationsdone[3] &&
                    !operationsdone[4] && operationsdone[5] && !
operationsdone[6] && operationsdone[7] &&
                    operationsdone[8] && !operationsdone[9] && !
operationsdone[10] && operationsdone[11])
        {
            c.getImage().display();
```

```
            System.out.println("Load Gameboy graphics ok");
        }
        else
        {
            MyImage err = new
MyImage(SystemValues.project_path+"img/error.jpg");
            err.display();
            System.out.println("Error loading graphics");
        }
    }
```

Adaptez votre code (normalement, si tout est bien fait, cela doit être très facile) à ces changements. Si vous rencontrez des problèmes, c'est que vous n'avez pas très bien implémenté Builder...