Design Pattern – TD n°5

LE PATTERN PROTOTYPE

Vous venez d'être recruté en tant que responsable des effets spéciaux d'une grande boîte. Vous récupérez ici les effets spéciaux réalisés par votre prédécesseur, qui avait travaillé sur le film Le Seigneur des Anneaux. En exécutant le main, vous voyez une grandiose scène de bataille apparaître.

1. George Lucas vient de vous contacter pour s'occuper des effets spéciaux du prochain Star Wars: l'histoire de Jar Jar Binks. Vous devrez vous occuper aussi d'une grande bataille épique. Récupérez le code ici (réalisé par l'équipe design), et tentez de modifier la fonction *main* afin de faire apparaître des vaisseaux de la guerre des étoiles.

2. On va tenter de modifier le code afin de rendre plus simple les modifications du *Main* si on décide, à l'avenir, de rajouter un projet. De plus, on se rend compte d'un problème: René a fait une bêtise la dernière fois, et a glissé dans la scène du Seigneur des Anneaux (que votre équipe retravaillait pour l'édition en BluRay) un Xwing de Star Wars. Peter Jackson et George Lucas sont furieux, il faut donc faire en sorte, qu'à l'avenir, le *Main* ne voit pas les différents *Soldiers* d'une scène.

Tout d'abord, on va implémenter le patter Prototype en proposant, dans l'interface *Soldier*, une fonction *clone* qui prend en paramètre un unique *Soldier*.

3. Ensuite, construisez une factory qui propose une fonction

```
Soldier getSoldier(Soldier s, int x, int y)
```

dont le but sera de cloner s et positionner le clone aux coordonnées x et y.

4. Modifiez ensuite *Scene* afin que chaque scène possède deux soldats (un de chacun des deux camps qui s'affrontent).

5. Modifiez ensuite le *Main* afin que celui-ci ne voie plus les détails d'implémentation de *Soldiers*.

6. Testez votre code en ajoutant ces classes et en tentant de générer une image où des marios et des goombas s'affrontent (téléchargez les classes de base, à modifier, ici).

Le pattern prototype permet de créer des objets en se basant sur une instance déjà existante de l'objet, sans connaître les détails de l'objet (est-ce un xwing, un mario, un knight ?). Tout ceci est rendu possible grâce à une fonction de clonage qu'il faut définir.

LE PATTERN COMPOSITE : UN « LS » FAIT MAISON

Le pattern Composite permet de considérer un objet ou l'une de ses parties comme étant du même type. Typiquement, ce pattern permet d'effectuer des opérations récursives sur un objet, afin de propager une action.

Rappelez-vous la bonne façon de mettre en place un analyseur d'expressions mathématiques. Comment aviez-vous, par le passé, implémenté le programme de calculatrice permettant de calculer, par exemple, « (2+4)*5-6 » ?

Vous aviez utilisé un pattern composite qui considérait les feuilles et les nœuds de l'arbre comme étant des objets du même type. Ici, le but sera de mettre en place, grâce à la classe *Files* de Java, une commande affichant le contenu d'un répertoire (fichiers et sous-dossiers).

Par exemple, voici ce que l'on souhaite voir à l'écran si l'on considère un dossier « test » qui contient « Dossier 1 » et « Dossier 2 », « Dossier 1 » contient « Sous-Dossier A », « Sous-Dossier B », et le fichier « file1 », tandis que « Dossier 2 » contient « Sous-Dossier C » qui contient le fichier « file 2 » :

```
-test
 |-Dossier 1
 | |>file1(26 bytes)
 | |-Sous-Dossier B
 | |-Sous-Dossier A
 |-Dossier 2
 | |-Sous-Dossier C
 | | |>file2(6 bytes)
```

(remarquez que l'on affiche la taille des fichiers)

Vous devrez mettre en place une classe abstraite *FileComposite*, dont héritent *Dossier* et *Fichier*. A la création du premier *Dossier* (correspondant au dossier en cours), l'arbre des sous-dossiers doit être automatiquement créé par appels récursifs. Puis, une fonction *print* devra être implémentée dans chacune des classes afin de permettre l'affichage indiqué ci-dessus.

TROUVER LE BON PATTERN POUR RÉSOUDRE UN PROBLÈME

Dans cet exercice, on veut modifier un peu le comportement de la méthode write de FileOutputStream afin de pouvoir effectuer certaines opération lorsqu'on écrit des caractères.

1. Considérez ce code :

```
import java.io.*;

public class Test
{
    public static void main(String[] argv)
    {
        String text1="Premiere Ligne";
        String text2="Seconde ligne 123";
        String text3="Troisième 635 ligne";

        File fic = new File("monfichier.txt");

        try
        {
```

```
            OutputStream ostream = new FileOutputStream(fic);
            ostream.write(text1.getBytes());
            ostream.write(text2.getBytes());
            ostream.write(text3.getBytes());
        }
        catch(Exception e)
        {
            System.out.println("Problème pour ouvrir le fichier.");
        }
    }
}
```

Il permet d'écrire du texte dans un fichier. Testez-le.

2. On voudrait rajouter une fonctionnalité : cacher, dans le flux de sortie, les chiffres et les remplacer par des étoiles. Le problème, c'est que le client ne veut pas que l'on touche à *ostream.write* : quoique l'on fasse, on doit continuer d'écrire dans le fichier avec *ostream.write*. De plus, il souhaiterait peut-être plus tard rajouter d'autres fonctionnalités assez simplement.

Voyez-vous une façon d'ajouter des fonctionnalités à *ostream.write* sans modifier *ostream.write* ?

Indices : .La solution est un pattern que vous avez déjà vu.

.*FileOutputStream* hérite de *OutputStream*.

.Imaginons que votre "filtre" s'appelle *HideDigitFilter*. Pour l'utiliser, vous feriez dans le client :

```
OutputStream ostream = new HideDigitFilter(new FileOutputStream(fic));
```

.Jetez un coup d'œil sur la classe *FilterOutputStream*.

.Vous pouvez, si vous le souhaitez, appeler au secours à l'aide si vous ne comprenez vraiment pas.

3. Rajoutez, tout aussi facilement, une méthode permettant de changer toutes les lettres en majuscules, et une méthode permettant de compter les caractères écrits en sortie.

Combinez vos méthodes, est-ce que cela fonctionne ?