Design Pattern – TD n°3

Dans ce troisième TD, nous nous intéresserons au pattern Builder ainsi qu'au pattern Singleton.

JEUX VIDEO (PATTERN BUILDER)

Vous venez d'être employé dans l'équipe de développement d'un nouveau jeux vidéo, et plus précisément, vous devez vous occuper de la partie de gestion des armes.

1. Récupérez le code qui a déjà été écrit par vos prédécesseurs, à cette adresse. Ce code permet de gérer trois types d'armes (pistolet, tronçonneuse et tricycle). Si l'on souhaite construire un un objet de type *Gun*, il faut le créer (avec un *new*) puis exécuter la liste d'ordres programmée dans le main. Cette liste n'est pas sous votre contrôle, et permet de charger correctement dans la mémoire un objet 3d. Elle dépend de l'architecture utilisée, et peut varier. Si un ordre non programmé dans l'objet est demandé, on ignore l'ordre.

Essayez de faire charger correctement (en exécutant tous les ordres programmés dans le main)un objet de type *Saw*.

2. La semaine dernière, on a vu le pattern Factory. Essayez d'implémenter ce pattern afin de rendre la création et l'exécution des ordres programmés plus transparente au client. Vous devrez certainement construire une interface de *Weapon*.

3. Un premier défaut de cette architecture vient du fait qu'on a mal placé certaines fonctions. Par exemple, dans *Gun*, tout ce qui n'est pas le constructeur devrait être dans une classe permettant de précharger un *Gun* dans la mémoire. Appelons cette classe *GunBuilder*.

Créez *GunBuilder*, qui doit posséder une variable interne de type *Gun*, une fonction *CreateNewWeapon*, et une fonction *GetWeapon* (ainsi que toutes les fonctions de *Gun*).

Créez aussi *SawBuilder* et *BikeBuilder*.

4. Créez une classe *Director* dont le but sera de construire un *Gun* à l'aide de *GunBuilder*, un *Saw* à l'aide de *SawBuilder*, ou un *Bike* à l'aide de *BikeBuilder*.

5. On voudrait maintenant simplifier le *Director* pour que ce dernier n'ait pas du code spécifique pour le *Gun*, le *Saw* ou le *Bike*.

L'idée serait de définir un *AbstractWeaponBuilder*, qui implémenterait l'ensemble de toutes les fonctions des Builder (implémentation simple : rien ne se produit). Chaque Builder spécialisé

hériterait de *AbstractWeaponBuilder* et surchargerait seulement les fonctions où l'on souhaite avoir un comportement spécifique.

6. Enfin, simplifiez au mieux le code du *Director*. Si tout s'est bien passé, vous devriez obtenir le pattern Builder : on délègue la construction complexe d'objets à des sous classes spécialisées, et on appelle un Directeur qui s'occupe de faire l'assemblage.

7. Ajoutez un autre arme : avez-vous dû modifier beaucoup de code existant ? Ajoutez, à cette arme, le fait qu'elle devra, dans son préchargement, posséder une toute nouvelle fonction (du type Preload3dEffects). Etait-ce compliqué ?

LE PATTERN SINGLETON

On va maintenant s'intéresser au pattern Singleton.

1/ On veut créer un *Aeroport* ainsi que des objets de type *Avion*. Voici le code des classes *Aeroport* et *Avion*, ainsi que de la classe test de l'ensemble.

```
public class Aeroport
{
	public Aeroport()
	{
		piste_libre=true;
	}
}

class Avion extends Thread
{
	String nom;
	Aeroport a;

	public Avion(String s)
	{
		nom=s;
	}

	public void run()
	{
		a=new Aeroport();
		System.out.println("Je suis avion "+nom+" sur aeroport "+a);
	}
}

class testaeroport
{
	public static void main(String[] args)
	{
		Avion v1 = new Avion("Avion 1");
		Avion v2 = new Avion("Avion 2");
		Avion v3 = new Avion("Avion 3");
		Avion v4 = new Avion("Avion 4");

		v1.start();
		v2.start();
		v3.start();
		v4.start();
```

```
        }
}
```

2. Que fait la méthode *start* lorsqu'elle est appelée sur les avions ?

On souhaite que les clients, les objets de type *Avion*, ne puisse pas créer plus d'un *Aeroport*, afin qu'ils se situent tous dans un même *Aeroport*. Comment empêcher la possibilité à un *Avion* de créer un *Aeroport* s'il en existe déjà un ?

Mettez-en place votre solution et compilez.

3. Vous venez de mettre en place le pattern Singleton : une variable de classe de type *Aeroport*, un constructeur privé et une fonction statique qui créé un *Aeroport* s'il n'en existe pas, et qui renvoie l'*Aeroport* existant s'il existe.

Maintenant, vérifions si *Aeroport* est robuste à la création de threads. On va imaginer que la création d'un *Aeroport* prend du temps... Avant de créer un nouvel *Aeroport*, ajoutez ce code :

```
try{
      Thread.sleep(500);
}
catch(Exception e){}
```

Cela endormira le thread *Avion* qui tente de créer un *Aeroport* pendant 500ms. Testez votre code : que se passe-t-il ?

4. Il faut faire en sorte que, si plusieurs threads Avion tentent d'obtenir un *Aeroport*, un seul puisse créer l'*Aeroport* à chaque fois. Comment mettre en place cette solution (regardez le mot clef synchronized en Java).

Testez votre nouveau code. Y a-t-il de meilleurs moyens pour résoudre le problème ?

Vous venez d'améliorer votre pattern Singleton afin qu'il prenne en charge une exécution multi-thread. Le pattern Singleton est souvent utilisé dans le cas de threads afin de créer un seul objet centralisé sur lesquels certains threads se connectent, d'où l'importance de la dernière partie de l'exercice que vous venez de mettre en place.

DONJONS & PATTERNS (OU PATTERNS & DRAGONS)

On souhaite créer un labyrinthe dans lequel des joueurs (manipulés par des threads se déplaçant au hasard) évoluent.

.Le **Labyrinthe** ne doit pouvoir être instancié qu'une seule fois. Il possède une matrice de Salles qui représente le labyrinthe. Les **Salles** peuvent être des **SalleAuTresor**, **SalleAvecMonstre**, **SalleAvecPiege**, **SalleVide** et une seule **SalleSortie**.

Lorsqu'un joueur atteint la **SalleSortie**, il gagne. Au début, l'emplacement de la **SalleSortie** est choisi.

.Une salle possède des **Sorties** (Est, Ouest, Nord, Sud) qui sont ou non activées (c'est un labyrinthe, donc toutes les salles ne se ressemblent pas). On souhaite, lors de la création d'une salle, créer un objet **Sortie** sur lequel la méthode **hasSortie(direction)** renvoie vrai ou faux selon si la **Sortie** dans une direction existe ou non. Pensez au fait que plus tard, peut-

être, il faudra implémenter des sorties de type Nord-Est, Nord-Ouest, Sud-Est, Sud-Ouest.

.Au fur et à mesure que les **Joueurs** passent de salle en salle, les **Salles** sont générées (et mémorisées par le **Labyrinthe**). Quand une salle est générée, il faut s'assurer que le **Joueur** peut atteindre la **SalleSortie** de son côté du **Labyrinthe** (qu'il ne soit pas enfermé sans espoir de gagner).

.Les **SallesAvecMonstre** possèdent un **Monstre**, les **SallesAvecPiege** possèdent un **Piège** et les **SallesAuTresor** possèdent un **Trésor**. On pourra créer différents types de monstres, de piège et de trésor.

.Les **Joueurs** ont un niveau qui augmente au fur et à mesure de leur progression dans le **Labyrinthe.** Un joueur qui tombe sur un montres plus fort que lui doit le battre (on peut penser à des combats du type Pierre-Ciseau-Papier avec handicap), et s'il perd, doit rebrousser chemin. Il peut aussi soudoyer le monstre avec son or (gagné dans les trésors). Enfin, les pièges peuvent paralyser le joueur pendant un certain temps, ou l'envoyer ailleurs dans le labyrinthe (le joueur peut éviter le piège en tirant un nombre au hasard).

L'implémentation est assez libre, mais réfléchissez bien aux différents patterns à mettre en place pour que la structure de votre labyrinthe puisse être améliorée plus tard facilement.