

TD5 : Proxy, Command, Observer (et Factory et Singleton !)

Exercice 1 - Proxy

Voici quelques classes permettant de gérer un album de photo [td4photo.zip](http://igm.univ-mlv.fr/ens/IR/IR2/2008-2009/POO/td4photo.zip) (<http://igm.univ-mlv.fr/ens/IR/IR2/2008-2009/POO/td4photo.zip>). On cherche pour déboguer à afficher un message lorsque l'on entre ou sort d'une méthode particulière d'un objet.

1. Créez un objet `java.util.logging.Logger` et affichez un message avec le niveau (Level) ALL.
Affichez en utilisant la méthode `log` un message de niveau WARNING.
2. Rappelez le principe du design pattern Proxy.
Écrivez la méthode `createPhotoProxy(Photo photo, Logger logger)` dans la classe `PhotoFactory` qui prend un objet de type `Photo` et un logger et qui renvoie un objet de type `Photo`. Lorsqu'elle est appelée, elle doit afficher sur le logger, lors de l'entrée dans une méthode le message `enter method` suivi du nom de la méthode, et lors de la sortie de la méthode le message `exit method` suivi du nom de la méthode.
Pour tester le proxy, changez le code de la méthode `createPhoto(File file)` de sorte que, si l'on crée la factory avec un logger, on utilise le proxy.

Pour les pro ! Ou chez vous :

3. On souhaite écrire un proxy générique qui affiche les messages d'entrée et de sortie quel que soit l'objet.
Pour cela nous allons utiliser la classe [java.lang.reflect.Proxy](#).
Écrivez une méthode `createProxy()` générique et correctement typée créant un proxy générique.
4. Modifiez votre code pour que ne soient affichées que les méthodes ayant l'annotation `Log`.

Exercice 2 - Automate

Une société de fabrication d'automates programmables souhaite développer sa nouvelle gamme de produits basés sur la technologie Java.

Le cœur du moteur de commandes doit être facilement extensible à de nouvelles fonctionnalités.

Le Robot est composé d'un bras articulé muni d'une pince pouvant se déplacer linéairement dans les trois directions: X, Y et Z. Pour cela il dispose d'un moteur pas à pas pour chaque axe.

Chaque moteur peut se déplacer en avant (positif) ou en arrière (négatif) d'un nombre de pas fixé, sur les trois axes du robot: X,Y et Z. La méthode `move(int steps)` permet de donner le nombre entier de pas du déplacement sur le moteur correspondant.

La consigne `light(boolean on)` permet d'allumer ou éteindre la lumière au bout du bras. Enfin les consignes `open()` et `close()` permettant d'ouvrir ou fermer la pince.

A terme, on souhaite pouvoir :

- rendre asynchrone l'exécution des actions sur le robot
- pouvoir effectuer un arrêt d'urgence
- pouvoir visualiser l'état d'avancement des commandes de plusieurs manières :
 - affichage texte dans une console

- affichage graphique des mouvements
- remettre le robot dans son état initial après une série d'actions

1/ Rappeler le schéma du Patern Command

2/ Modifier le code donné (<http://igm.univ-mlv.fr/ens/IR/IR2/2011-2012/PrOO/automate0.zip>) pour mettre en place le pattern Command.

Transformer le main() du Controller en conséquence.

Note: pour l'accès au Robot, vous pouvez simplement utiliser le pattern Singleton

3/ simplifier le code du Controller en mettant en place une Factory de Commande's

4/ On veut découpler réellement les demandes d'actions au Robot et leur exécution : ajouter la gestion d'une pile de commandes, ainsi que l'exécution asynchrone

note : vous pouvez vous aider des implémentations de BlockingQueue.

5/ On souhaite surveiller les mouvement des axes en implémentant des messages indiquant si l'axe démarre son déplacement ou s'il stoppe le mouvement.

Pour l'instant, la surveillance se traduira simplement par des affichages « sysout ». Mais la structure implémentée devra permettre de « brancher » une interface graphique qui montrera l'évolution des actions sur le Robot.

- Rappeler le schéma du Pattern Observer.

- On implémentera cette surveillance en utilisant le Pattern Observer. N'utilisez pas les Observable du JDK.

Ajouter un observer qui loggiera dans un fichier l'historique de toutes les actions demandées

6/ Ajouter la gestion de priorité d'une commande : une commande Abort, de priorité maximum, doit pouvoir s'exécuter avant toutes les autres commandes en attente (on ne s'intéressera pas ici à l'interruption de la commande en cours d'exécution).

note : vous pouvez vous aider de PriorityBlockingQueue.

note : après l'exécution de Abort, vider la liste des commandes en attente

7/ On veut pouvoir facilement remettre le robot dans son état initial. Par exemple :

```
Controller.startSession()
```

```
.... envoi de commandes ...
```

```
// "annule" toutes les commandes envoyées et
```

```
// remet donc le robot à sa position initiale, avec la pince et la lumière dans l'état initial
```

```
Controller.undoSession() ;
```