TP design patterns

* L’évolution de la modélisation

Après le premier TP, nous avons fait un premier modèle UML dans une optique de Design Pattern Observer, que vous pourrez retrouver dans le répertoire sous le nom UML.png. Par rapport au modèle UML final, les classes Publisher et Subscriber sont remplacées par deux interfaces, car il était un peu ambigu de mettre les attributs de Pile dans la classe Publisher. Ainsi, la classe Pile réalise l’interface Publisher, et les classes qui ont besoin peuvent réaliser l’interface Subscriber. Par ailleurs, pour simplifier, un ArrayList est utilisé pour gérer la liste de Subscribers.

Nous pensions que la classe Log réalisait également l’interface Subscriber, cependant chaque pile doit avoir son propre fichier log. C’est pourquoi la classe PILE comporte un nouvel attribut PILLOG. Le nom du fichier log est donné en entrée lorsque l’on crée un objet Pile.

Lors de la phase de programmation, nous avons ajouté une classe Command pour recevoir les instructions saisies depuis le clavier.

Concernant les alarmes, une classe mère et des classes filles réalisent l’interface Subscriber, et n’ont que peu changé au court du TP.

La classe Stock réalise aussi l’interface Subscriber. À la suite d’une précision des consignes, une fonction pour l’affichage du dernier élément produit et une autre pour l’affichage de tout le contenu du stock ont été ajoutées.

Pour la gestion d’alarmes, nous avons suivi la DP Stratégie. Il y a tout d’abord une interface Warn qui gère les différents types d’alarmes qui correspondent à la saisie SET WARN 1, SET WARN 2. Cela a impliqué d’ajouter un attribut PILWarn dans la classe Pile, qui permet de changer dynamiquement le mécanisme de la gestion des alarmes. Les alarmes du paragraphe précédent étant similaires à Warn, nous avons ajouté une interface Alert. Les deux méthodes d’alarmes la réalisent.

Concernant la classe Warn1, elle analyse chaque modification du stock pour définir s’il y a besoin d’alerter ou non l’utilisateur, en fonction de nombre d’éléments restants. Cela nécessite une réalisation de l’interface Subscriber. Pour Warn2, elle compte le nombre d’éléments toutes les deux secondes.

Pour finir, nous avons ajouté deux classes Producer et Consumer pour créer des threads producteur et consommateur. Le producteur produit un entier toutes les 20s et le consommateur consomme un entier toutes les 30s. Il faut savoir dans quelle pile il produit ou consomme, donc il y a une agrégation entre la classe Pile et les deux nouvelles classes.

* La justification des choix et les limites

Réponses aux questions des séances 2 et 3 :

1° - Pour supprimer une alarme, on peut ajouter une ligne PIL.PILRemoveSubscriber(...) dans la classe main. Pareil pour supprimer un consommateur.

Pour modifier la construction du log, on peut ajouter une ligne PIL.PILLogWrite(...) dans la classe main.

* Il faut changer seulement la partie de code qui s’occupe de l’applicatif.
* Il n’a y pas besoin de modifier la classe de stockage

2° J’ajoute une fonction dans la classe Stock qui permet d’afficher tout le contenu du stock.

3° Oui les méthodes/classes s’occupant des alarmes sont sollicitées bien plus souvent que nécessaire, lors de chaque modification du stock.

* Également n'hésitez pas expliquer votre modélisation (DP utilisés, de quelle façon, adaptations ...)

DP Observer

Publisher : Pile

Subscriber : Stock, Alert1(Alert1Empty, Alert1AlmostEmpty, Alert1AlmostFull, Alert1Full), Warn1.

DP Strategy pour gérer dynamiquement les alarmes. La classe Warn1 réalise à la fois l’interface Warn et Subscriber.

Warn1 : afficher une alarme quand le nombre d’éléments est supérieur à 7.

Warn2 : afficher une alarme s’il y a plus de 2 éléments produits ou consommés dans les deux secondes.

* La ligne de commande à utiliser pour exécuter votre programme sachant que je ferai un copié/collé dans la console à partir du répertoire où se trouve le jar

java –jar Stock.jar

(PUSH ..., POP, Clear, SET WARN 1, SET WARN 2, q)