Patron de conception / Design pattern – 4 EII

TP2

1 Buts

Le but principal de ce TP est de bien comprendre le rôle des interfaces et l'intérêt de séparer les interfaces de leurs implémentations (*i.e.* les classes). L'intérêt est principalement de faciliter la maintenance et l'évolution du code, ainsi que la mise en place de patrons de conception tels que *fabrique abstraite* (qui a également pour but de faciliter la maintenance et l'évolution du code) ou *adaptateur*.

Ce TP avancera progressivement dans ce sens pour terminer sur la mise en place d'une fabrique abstraite (ce qui est relativement complexe).

2 TP

Tout comme le TP précédent, l'énoncé ainsi que le code Java nécessaire à la réalisation du TP sont fournis sur Moodle dans la matière *Qualité Logicielle*. Vous devez donc télécharger le projet Eclipse que nous vous fournissons et l'importer dans Eclipse (*File -> Import -> Existing Projects*).

Quelques raccourcis très utiles sous Eclipse : *ctrl+shift+0*, mettre à jour les *imports*; *alt+shit+r*, permet de renommer une classe, une operation, *etc.*

2.1 Séparation *entre interfaces et classes* : les listes Java

Il existe plusieurs types de liste en Java, notamment *ArrayList* (`http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html`).

Q.1a Dans l'opération *main* de la classe *q1.Main*, créez 2 listes (*ArrayList*) d'entiers, ajoutez quelques valeurs dans la première et faites en sorte que la seconde contienne les mêmes valeurs mais dans l'ordre opposé (utilisez une boucle *for*). Affichez ensuite les 2 listes (*System.out.println(liste1)*).

On veut maintenant changer de type de liste pour utiliser la classe *LinkedList* `http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html`.

Q.1b Effectuez ce changement. Combien de modifications avez-vous effectués ? Imaginez que votre code fasse plusieurs milliers de lignes utilisant un peu partout *ArrayList*, quels sont problèmes pour d'effectuer de tels changement ?

Les classes *LinkedList* et *ArrayList* partagent un certains nombres d'opérations regroupées dans plusieurs interfaces.

Q.1c Étudiez la javadoc des 2 classes pour identifier l'interface commune à utiliser dans votre code et changez votre code.

Q.1d Quelle est la différence entre une interface et une classe ? Quel est l'avantage à utiliser une interface comme type d'une variable ? Pourquoi ne pas utiliser une classe abstraite plutôt qu'une interface pour définir le type d'une variable ?

2.2 Mise en place d'une *fabrique abstraite*

Imaginons un jeu vidéo dans lequel différentes civilisations se combattent avec des unités. Les civilisations ont le même type d'*Unité* : *Blindé* et *Infanterie*. Une Unité possède des opérations pour retourner le nombre de points de vie, le nombre de points d'attaque. Une *Infanterie* possède une opération permettant de savoir si l'unité peut se camoufler où non. Un *Blindé* possède une opération pour pilonner une position (x et y).

Q.2a Créez les interfaces décrites dans le texte ci-dessus dans le package *q2.unit*.

Il existe deux civilisations implémentant ces interfaces : FR et US. La civilisation FR possède des *Char*s et des *BeretVert*s. Un *Char* possède 4 d'attaque et 10 de vie, un *BeretVert* 2 d'attaque et 3 de vie et ne peut pas se camoufler. La civilisation US possède des *Tank*s et des *Marine*s. Un *Tank* possède 5 d'attaque et 9 de vie, un Marine 1 d'attaque et 5 de vie et peut se camoufler.

Q.2b Créez les classes correspondant à chacune des civilisations décrites dans le texte ci-dessus dans les package *q2.unit.fr* et *q2.unit.us* (laisser les opérations *pilonner* vide).

Q.2c Dans l'opération *main*, déclarez une variable *Blindé* et une autre *Infanterie* (utilisez les interfaces) et initialisez-les avec les classes FR. S'il vous fallait changer de civilisation, quel type de changement devriez-vous effectuer ?

On vient de voir l'utilité de séparer les interfaces de leurs différentes implémentations. Cependant l'appel au constructeur pose toujours problème car doit être changé à chaque changement d'implémentations. Nous allons voir comment faire pour éviter ce problème grâce à *Fabrique Abstraite*. Utilisez le cours pour vous aider dans les questions suivantes.

Q.2d Ajoutez dans le package des interfaces une interface *FabriqueUnité*. Cette interface aura 1 opération pour créer chaque type d'unité (comme vu en cours).

Q.2e Implémentez cette interface pour les 2 civilisations.

Q.2f Modifiez votre *main* pour utiliser ces nouvelles classes / interfaces.

Q.2g Commentez les avantages de ce patron de conception. Quelle est la contrainte de ce patron ?

2.3 Proxy

Nous disposons d'une interface MyObject, définissant un objet quelconque, et d'une implémentation *TimeConsumingObject*. Le problème est que la création d'un objet TimeConsumingObject prend beaucoup de temps. Nous voulons accélérer de démarrage de notre application en ne l'instanciant que lors de sa première utilisation (principe de l'instanciation paresseuse). Il existe plusieurs solutions. Nous allons utiliser ici un proxy.

Q.3a Créez une nouvelle sous-classe LazyObject de MyObject. Cette nouvelle classe sera le proxy pour instancier un object *TimeConsumingObject* uniquement à la première utilisation d'une de ses méthodes. Trouvez comment en vous aidant du cours.

2.4 Visiteur / *Visitor*

Dans cet exercice nous allons modéliser des expressions arithmétiques sous la forme d'un arbre binaire. Seules l'addition et la soustraction devront être considérées. Un *Arbre* possède un *Noeud racine* et un *nom*. Un *Noeud* est soit un *NoeudPlus*, soit un *NoeudMoins*, soit un *NoeudValeur*. *NoeudPlus* et *NoeudMoins* possède chacun un *noeud droit* et un *noeud gauche*. *NoeudValeur* possède une *valeur* (un entier). Vous modéliserez *Noeud* sous la forme d'une interface.

Q.4a Nous voulons parcourir l'arbre pour effectuer différentes opérations (calcul, affichage en console). Quelle est la première manière de faire qui vous vient à l'esprit ? Regarder le patron *Visiteur* dans le cours et comparez à votre solution.

Q.4b Ajoutez le patron de conception *Visiteur* à l'arbre : ajoutez les méthodes *accept(VisiteurArbre)* dans les bonnes classes et définissez l'interface *VisiteurArbre* et ses méthodes comme le montre l'image fournie dans le package.

Q.4c Pourquoi les méthodes *accept* sont-elles nécessaires ?

Q.4d Implémentez en Java un visiteur permettant d'afficher dans la console et en notation post-fixée (ex., 1 2 +) la formule arithmétique que représente l'arbre.

Q.4e Implémentez en Java un visiteur permettant de calculer la formule arithmétique que représente l'arbre. Un indice : le visiteur possédera une pile stockant les valeurs visitées. Vous utilisez alors le patron *Visiteur* pour faire un *Interpréteur* de manière propre.