Patron de conception / Design Pattern – 4 EII

TP1

1 Outils

Pour ces TP sur les patrons de conception vous utiliserez l'outil de développement Eclipse. Les TP seront réalisés en Java. Jusqu'à présent vous définissiez un "*main*" pour tester votre code en TP. Dans ces 2 TP vous utilisez des tests unitaires (*JUnit*) comme vu récemment.

Des tests vous sont fournis (dans le dossier *src/test*) pour assurer dans une certaine mesure que votre code fonctionne comme attendu. Ne prêtez pas une grande attention au code des tests, vous verrez la syntaxe des tests en 5EII. Ne modifiez pas ces tests. Dans le dossier src/main, un package par question existe, contenant éventuellement du code. Rajoutez votre code dans chaque package.

2 TP

Ces 2 TP sont composés de petits exercices dont le but est de comprendre aussi bien les problèmes que résolvent chaque patron que la solution qu'ils proposent.

Toutes les énoncés ainsi que le code Java nécessaire à la réalisation des 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 -> General -> Existing Projects into workspace*). Des erreurs seront présentes dans ce projet Eclipse dans le dossier *test*. Ne vous en souciez pas trop, elles disparaîtront au fur-et-à-mesure du TP (si vous faites les choses correctement).

2.1 Utilisation d'un singleton pour collecter des erreurs

Soit le programme Java suivant :

```
public class Q1 {
   public URL toURL(final String path) {
      URL url;
      try { url = new URL(path); }
      catch(MalformedURLException e) {
         // We want to gather the exceptions here.
         url = null;
      }
      return url;
   }
}
```

Lors de la conversion du *String* en *URL*, des erreurs (des exceptions) peuvent survenir (*MalformedURLException*). On voudrait collecter ces erreurs dans un collecteur global à tout le programme. Nous voulons que ce collecteur soit unique (une unique instance) et accessible facilement dans tout le programme. C'est pourquoi l'utilisation d'un singleton semble approprié : une classe qui ne peut être instanciée qu'une seule fois.

Q.1a Codez un tel collecteur (classe *CollecteurErreur*) et modifiez le code ci-dessus pour collecter les erreurs. Ce collecteur devra donc être un singleton possédant au moins une opération *add(Exception ex)* ajoutant une exception au collecteur. Pour contenir les erreurs (des objets de la classe *Exception*), votre collecteur pourra hériter de la classe Java *ArrayList* (`http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html`). Il n'est pas utile de re-écrire une opération *add* puisque la classe *ArrayList* en définit déjà une.

Q.1b Assurez-vous que les test unitaires que vous nous fournissons passent. Examinez ces tests pour comprendre leur fonctionnement.

2.2 Utilisation d'une fabrique pour lire une sauvegarde

Voici une énumération définissant les quatre couleurs d'un jeu de carte.

```
public enum ColourCard {
  SPADE, CLUB, HEART, DIAMOND;
}
```

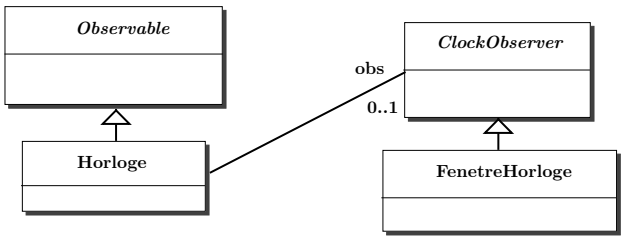

Figure 1: Diagramme de classes

Q.2a Développer en Java un moyen pour obtenir une couleur à partir d'une chaîne de caractères correspondant au nom de la couleur recherchée (le nom d'un élément d'une énumération s'obtient grâce à la méthode *name()*, par ex. *SPADE.name()*, tandis que la méthode *values()* retourne un tableau contenant toutes les éléments de l'énumération, par ex *ColourCard[] items = values()* ;
Q.2b Assurez-vous que les test unitaires que vous nous fournissons passent. Examinez ces tests pour comprendre leur fonctionnement.

2.3 Utilisation d'un *Observateur* pour afficher l'heure

Lancez l'application Swing se trouvant dans le package *q3*. Cette application est sensée afficher l'heure courante. L'affichage est mis à jour toutes les seconde.
Q.3a Que constatez-vous ? Expliquez la raison du problème.
Q.3b Modifiez le code de l'application pour y introduire aux endroits indiqués le patron *observateur* pour résoudre le problème. Utilisez pour cela les interfaces fournies *Observable* et *ClockObserver*. Aidez-vous du diagramme de classes donné.

2.4 Utilisation d'une *Stratégie* et d'une *Fabrique* dans un jeu vidéo

Lancez l'application Swing se trouvant dans le package *q4*. Cette application est un jeu vidéo dans lequel le joueur (le carré gris) ne doit pas être attrapé par le méchant (le cercle rouge). Vous pouvez constater que le méchant bouge aléatoirement toutes les secondes tandis que vous restez immobile. La classe *MoveUnit*, dans le package *q4.controler* gère les événements provenant du clavier pour déplacer votre unité (opération *keyPressed*). Pour utiliser les informations provenant du clavier, il est nécessaire de convertir la touche du clavier pressée par le joueur en une direction (ou rien si la touche n'est pas valide) comme l'illustre la ligne suivant le *TODO Q.4a*.
Q.4a Décommentez ces lignes et implémentez l'opération manquante. Cette opération est une fabrique convertissant une touche clavier en une direction. *Utilisez la documentation fournie :* `http://docs.oracle.com/javase/7/docs/api/java/awt/event/KeyEvent.html` *. Étant donné qu'il est possible de bouger l'unité avec le pavé numérique ainsi que les flèches, les constantes à utiliser sont par exemple* KeyEvent.VK_UP *ou* KeyEvent.VK_1.

Nous voulons maintenant que l'intelligence artificielle commandant le méchant évolue au cours du jeu. Nous voulons notamment 3 stratégies : déplacement aléatoire; déplacement au plus proche du joueur sans mouvement en diagonale; déplacement au plus proche du joueur avec mouvement en diagonale.
Q.4b Expliquez en quoi la patron *Stratégie* est adapté à cette situation et gribouillez un diagramme de classes expliquant votre solution fondée sur *Stratégie*. Pour cela, étudiez les classes *StrategyManager* et *StrategyEvilAbs* ainsi que l'interface *StrategyEvil* fournies dans le package *q4.strategy* et que vous devrez utiliser par la suite (cf. le cours aux diapos sur *Stratégie*).
Q.4c Modifiez le code pour implanter votre solution en utilisant *StrategyManager, StrategyMechantAbs* et *StrategyMechant* (suivez les endroits indiqués par *TODO Q.4c*, il vous faudra également créer des classes dans le package *q4.strategy*). La première stratégie est définie dans *MechantWorker*. Vous avez juste à faire hériter cette classe de *StrategyMechantAbs* (qui est la classe abstraite mère des stratégies). Traiter le TODO se trouvant un bas de la classe *UI*.