**Université de Carthage Année universitair2020/2021**

**Institut National des Sciences Appliquées et de Technologie Section : GL3**

Module : **UML et design Patterns Mme. Mouna JARRAYA**

**TP N°3**

**UML et design patterns**

**Exercice 1 :**

Au sein d’un système de modélisation de formes graphiques, nous souhaitons représenter des formes complexes à partir de formes primitives colorées, connaître les couleurs dont elles disposent et calculer leurs surfaces.

**Les formes constituées à partir d’autres formes** sont colorés par leur propre couleur (ou pas) et prennent en compte les couleurs de leurs formes primitives. Ainsi la couleur finale d’une forme est obtenue par la superposition de sa couleur et des couleurs des autres formes qui la constitue.

Une solution immédiate consiste à traiter différemment les formes primitives et celles possédant d’autres formes. Cependant, cette différence de traitement entre les deux types de formes rend l’application plus complexe et dépendante de la **composition** interne des formes.

Rq : Pour raison de simplification, on suppose que toutes les formes ont la même surface.

La classe abstraite forme est définie comme suit :

|  |
| --- |
| Public abstract class Forme{  protected static double surface= 55.0;  protected String couleur;  public void ajouteCouleur(String couleur){  this.couleur=couleur ;  }  public abstract double calculerSurface() ;  public abstract String afficherCouleur() ;  public abstract boolean ajouteForme(Forme f);  } |

1) Quel pattern permet-il de modéliser une solution à cette problématique.

2) Modélisez son utilisation par un diagramme de classe.

3) Programmez ce diagramme de classe en Java. Le programme principal doit pouvoir créer des formes et leur différentes formes primitives, de calculer leurs surfaces et d’afficher leurs couleurs.

**Exercice 2 :**

Dans un contexte d’un jeu vidéo en ligne, il est nécessaire de gérer les super-pouvoirs que le joueur peut choisir lorsqu’il commande un nouvel avatar (il existe différents types d’avatars avec des différentes caractéristiques).

Ces super-pouvoirs sont décrits par la classe *SuperPouvoirAvatar* qui contient plusieurs attributs comme le nom, la description, etc.

Pour chaque avatar que le joueur souhaite acheter, il est possible d’associer une nouvelle instance de cette classe. Cependant, un grand nombre de super-pouvoirs est souvent présent pour chaque avatar commandé, ce qui oblige le système à gérer **un grand ensemble d’objets de petites tailles**. Cette approche présente toutefois l’avantage de **pouvoir stocker au niveau du super-pouvoir des informations spécifiques à celui-ci et à l’avatar** comme le prix de vente du super-pouvoir qui peut différer d’un avatar commandé à un autre.

Une image contenant texte

Description générée automatiquement

1) Quel pattern est le mieux adapté pour concevoir les super-pouvoirs d’un avatar.

2) Modélisez son utilisation par un diagramme de classe.

3) Programmez ce diagramme de classe en Java. La classe *AvatarCommandé* gère la liste des super-pouvoirs ainsi que la liste de leurs prix de vente. Ces deux listes sont générées en parallèle. Le prix de vente d’un super-pouvoir se trouve au même indice dans la liste *prixDeVenteSuperPouvoir* que le super-pouvoir dans la liste *superPouvoirs*. Cette classe doit contenir aussi une méthode *ajouteSuperPouvoirs* et une méthode afficheSuperPouvoirs.

Le programme principal doit pouvoir créer deux avatars commandés en leur affectant des super-pouvoirs communs et d’autres non communs et d’afficher pour chaque avatar commandé la liste des super-pouvoirs choisis.

**Exercice 3 :**

Dans un système de vente en ligne de produits paramédicaux, la classe *VueCatalogueProduits* affiche la liste des produits destinés à la vente. Un algorithme d’affichage est utilisé pour calculer la mise en page en fonction de l’âge de l’utilisateur. Il existe **deux versions de cet algorithme** :

* Une première version pour les personnes âgées qui n’affiche qu’un seul produit par ligne (un produit prend toute la largeur disponible) et qui affiche les informations avec une grande Police;
* Une seconde version pour les personnes non agées qui affiche trois produits par ligne mais et qui affiche les informations avec une police de taille normale.

L’interface de la classe *VueCatalogueProduits* **ne dépend pas du choix** de l’algorithme de mise en page. Ce choix n’a pas d’impact sur la relation d’une vue d’un catalogue avec ses clients. Il n’y a que la présentation qui est modifiée.

Une première solution consiste à transformer la classe *VueCatalogueProduits* en une interface ou en une classe abstraite et à introduire deux sous-classes d’implantation différant par le choix de l’algorithme. Ceci présente l’inconvénient de complexifier inutilement la hiérarchie des vues de catalogue.

Une autre possibilité est d’implanter les deux algorithmes dans la classe *VueCatalogueProduits* et à l’aide d’instructions conditionnelles d’effectuer les choix. Mais cela consiste à développer une classe relativement lourde et dont le code des méthodes est difficile à appréhender.

1) Quel pattern faut-il mettre en œuvre pour adapter l’interface de la classe *VueCatalogueProduits* aux algorithmes d’affichage.

2) Modélisez son utilisation par un diagramme de classe. La classe *VueCatalogueProduits* comporte une méthode *affiche().*

3) Programmez ce diagramme de classe en Java. Le programme principal consiste à créer deux instances de *VueCatalogueProduits*, la première est paramétrée par le dessein sur trois lignes. La seconde, instance est paramétrée par l’affichage sur une ligne. Après les avoir créés, le programme principal invoque la méthode *affiche* de ses instances. La classe *VueProduit* est donnée en annexe.

Annexe :

|  |
| --- |
| public class VueProduit{  protected String description ;  public VueProduit (String description)  {  this.description=description;  }  public void affiche()  {  System.out.print(description)  }  } |