Programmation Java TP #4

A. Pigeau

1 Exercice : [Modélisation & Javadoc & Jar]

L'entreprise *PolyMusic* veut développer une application pour gérer ses futures ventes d'instruments de musique. Les différents types d'instruments vendus sont :

- Guitare ;
- Batterie ;
- Clavier.

Un instrument est défini par :

- une catégorie ;
- un prix ;
- un numéro de référence, unique pour chaque produit.

Chaque instrument appartient à une seule catégorie : `HauteGamme`, `MoyenGamme` et `BasGamme`. Les instruments appartenant à une même gamme ont un prix identique : respectivement 1000€, 500€ et 100€ pour le `HauteGamme`, `MoyenGamme` et `BasGamme`.

Le numéro de référence est divisé en deux parties : la première correspond au type du produit (`Bat`, `Guit` ou `Clav`) et la seconde au numéro du produit. Par exemple, le numéro de référence pour une guitare peut être `Guit-123`.

Pour chaque instrument, nous devons pouvoir obtenir son numéro de référence, sa catégorie et son prix. Il est impératif que chaque instrument implémente ce comportement : vous proposerez donc une modélisation impliquant que tout produit respecte cette contrainte.

Une classe `GestionInstrument`, contenant l'ensemble des produits à vendre, doit fournir les méthodes suivantes :

- ajouter un produit
- supprimer un produit (à l'aide de la référence)
- afficher l'ensemble des produits

Il est nécessaire de respecter la contrainte suivante : les produits contenus dans l'instance de la classe *GestionInstrument* doivent être distincts deux à deux (un même instrument ne peut pas apparaître deux fois dans la liste !).

Question :

1. proposer une solution de modélisation ;
2. implémenter la hiérarchie des types d'instruments ;
3. implémenter la classe `GestionInstrument` ;
4. fournir un test (créer une liste de produit et afficher tous les instruments contenus) ;
5. générer la `Javadoc` :
    1. `Project` > `Generate JavaDoc...` ;
    2. dans `javadoc command`, mettre le chemin du programme `javadoc.exe` (dans `C:\Program Files\Java\jdk1.6.0_12\bin`) ;
    3. sélectionner toutes les classes du module ;
    4. dans `Use Standard Doclet`, choisir la destination des fichiers html.

6. exporter votre programme dans un fichier `jar` : **Attention, le fichier JAR ne contient pas obligatoirement les source ! il faut cocher la case *Export Java sources files and ressources* pour les inclure**
   1. `file` > `Export...` ;
   2. choisir `JAR file` ;
   3. `Jar File Specification` : sélectionner le module à exporter ;
      (vérifier que tous les fichiers sont bien sélectionnés)
   4. `Jar File Specification` : inclure les fichiers `.class` et les fichiers `.java` ;
   5. `Jar File Specification` : choisir le répertoire de destination ;
   6. `Jar Manifest Specification` : sélectionner la classe à exécuter dans le module ;
   7. exécuter le programme en ligne de commande (`java -jar fichier.jar`).

Javadoc [1] :

Javadoc est un outil développé par Sun Microsystems permettant de créer une documentation d'API en format HTML depuis les commentaires présents dans un code source en Java. Javadoc est le standard industriel pour la documentation des classes Java. La plupart des IDEs créent automatiquement le javadoc HTML. Quelques tags sont donnés dans le tableau suivant.

Tag	Description
`@author`	Nom du développeur
`@deprecated`	Marque la méthode comme dépréciée. Certains IDEs créent un avertissement à la compilation si la méthode est appelée.
`@exception`	Documente une exception lancée par une méthode - voir aussi `@throws`.
`@param`	Définit un paramètre de méthode. Requis pour chaque paramètre.
`@return`	Documente la valeur de retour. Ce tag ne devrait pas être employé pour des constructeurs ou des méthodes définis avec un type de retour `void`.
`@see`	Documente une association à une autre méthode ou classe.
`@since`	Précise à quelle version de la SDK/JDK une méthode à été ajoutée à la classe.
`@throws`	Documente une exception lancée par une méthode. Un synonyme pour `@exception` disponible depuis Javadoc 1.2.
`@version`	Donne la version d'une classe ou d'une méthode.

Exemple de commentaires Javadoc :

```
/**
 * Cette classe est la super classe des differents instruments vendus
 * @author Antoine Pigeau
 *
 */
public abstract class Instrument{
 /**
 * constructeur de la classe Instrument
 * @param t le type de l'instrument instancie
 */
 public Instrument(Type t){

 }
}
```

Fichier Jar [2] :

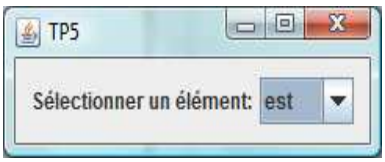
Un fichier JAR (Java ARchive) est un fichier ZIP utilisé pour distribuer un ensemble de classes Java. Ce format est utilisé pour stocker les définitions des classes, ainsi que des métadonnées, constituant l'ensemble d'un programme. Un tel fichier peut être destiné à être exécuté comme un programme indépendant, dont une des classes est la classe principale.

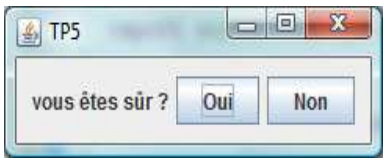
1. source Wikipedia
2. source Wikipedia

2 Exercice : [Modélisation]

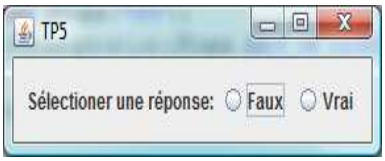
Votre client a besoin de créer différents types de `Panel` pour le développement de ses applications. Il vous demande de fournir des classes créant des `Panel` spécifiques.

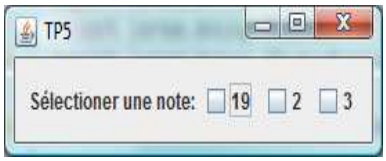
Deux types d'applications existent : les types A et B. Pour chaque type d'applications, deux styles de `Panel` existent. Pour les applications de type A, les `panel` à créer sont :





Pour les applications de type B, les `panel` à créer sont :





1. implémenter la hiérarchie des différents `Panel` à créer (voir le code ci-dessous pour les différents éléments à manipuler : chaque composant doit être ajouté à un `JPanel`, qui sera ensuite ajouté à un `JFrame`)
2. implémenter une classe abstraite `Fabrique`. L'interface de cette classe est la suivante :

```
Panel getPanelStyle(Style);
```

3. implémenter les classes `FabriqueA` et `FabriqueB`
4. implémenter un test : vos panels sont ajoutés à une fenêtre `JFrame`

```
public static void main(String[] args) {

  JFrame fenetre = new JFrame("TP5");
  // ferme le programme si je clique sur le 'X'
  fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  Factory f = new FactoryPanelAppA();

  JPanel p = new JPanel();
  p.add(f.getPanelStyle(Style.STYLE1));
  p.add(f.getPanelStyle(Style.STYLE2));
  fenetre.add(p);

  fenetre.pack(); // recalcule la dimension des composants
  fenetre.setVisible(true);   // fenetre visible
}
```

5. Ajouter un nouveau style de `Panel` pour chaque type d'applications. Combien de classes avez vous modifiées ?

Exemple de création d'un `JPanel` :

```
public class Panel extends JPanel {
   private JLabel text;
   private JComboBox combo;
   private JButton bt;
   private JRadioButton radio;
   private JCheckBox check;

```

```
    public Panel(){

        this.text = new JLabel("text");

        Vector<String> v = new Vector<String>();
        v.add("est");
        v.add("ouest");
        this.combo = new JComboBox(v);

        this.bt = new JButton("Oui");

        this.radio = new JRadioButton("Faux");

        this.check = new JCheckBox("3");

        this.add(text);
        this.add(this.combo);
        this.add(this.bt);
        this.add(this.radio);
        this.add(this.check);

    }
}
```

3 Exercice : [Modélisation]

Votre client désire modéliser des expressions mathématiques : la somme et la multiplication. Des exemples d'expression à modéliser sont $(2+1)$, $((4+6)+(7*2))$, $((6*7)*7)$, $((5*4)+7)$, ... Chaque élément de votre expression doit être représenté par un objet (par exemple, deux entiers 4 et 5 et un opérateur + pour l'expression $(4+5)$). Vous remarquerez qu'une expression peut elle-même être composée d'une expression.

Question :

1. proposer une solution de modélisation sous forme de diagramme de classes ;
2. implémenter votre solution
3. ajouter une méthode `public String toString()` permettant de retourner l'expression sous forme d'une chaîne de caractères ;
4. implémenter un test dans une classe `Test` comprenant la création et l'affichage de l'expression suivante : $((6*7)+(7+8))$.

4 Exercice : [Modélisation]

L'entreprise *PolyCocktail* veut une modélisation de ses cocktails.

- un cocktail est un mélange de plusieurs ingrédients
- les ingrédients possibles dans un cocktail sont : du jus d'abricot, du jus d'ananas, du jus de citron vert, du sirop de cerise, du sirop de grenadine et du soda à l'orange
- dans un cocktail, chaque ingrédient est associé à une dose. Par exemple un *abricot frappé* contient 4 doses de jus d'abricot, 3 de jus d'ananas, 1 de jus de citron vert et 1 de sirop de cerise. Un *indien* est composé de 1 dose de sirop de grenadine et de 10 doses de soda à l'orange

Question :

1. Modéliser les cocktails de *PolyCocktail*
2. Tester votre code dans une classe `Test` en créant un abricot frappé et en l'affichant sur la console
3. Proposer un moyen simple pour créer des instances de vos cocktails
4. Tester votre solution de création de cocktail

5. Proposer un moyen pour créer des copies (en profondeur) de vos cocktails
6. Tester votre solution de création de copie d'un cocktail (vérifier que la copie a les mêmes propriétés et que l'égalité sur les références, entre la copie et l'originale, est fausse)