USTL Licence STA Mention Informatique Semestre 5

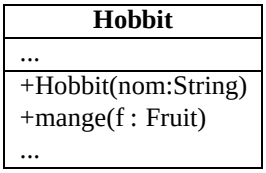
UE Conception Orientée Objet
Examen Première session – 2 février 2008

3 heures - notes et polycopiés de cours/TD/TP autorisés
livres et calculatrices interdits
dictionnaires de langue autorisés pour candidats étrangers

- ▷ les exercices sont totalement indépendants.
- ▷ le code sera conforme au jdk 1.5 de JAVA.

Exercice 1 : Mangez des fruits !

On dispose de la classe `Hobbit` suivante :



Dans le cadre d'une application permettant à des Hobbits de manger des fruits, les classes suivantes sont fournies :

```
package pasbo;
public class ArbreFruitier {
   private String type;
   public ArbreFruitier(String type) {
      this.type = type;
   }
   public Fruit cueille() {
      if (this.type.equals("pommier") {
         return new Fruit("pomme");
      else if (this.type.equals("poirier") {
         return new Fruit("poire");
      }
   }
   public String toString() {
      return this.type;
   }
}

package pasbo;
public class Fruit {
   private String type;
   public Fruit(String type) {
      this.type = type;
   }
   public String toString() {
      return this.type;
   }
}

package pasbo;
public class TheMain {
   public static void main(String[] args) {
      ArbreFruitier pommier = new ArbreFruitier("pommier");
      ArbreFruitier poirier = new ArbreFruitier("poirier");
      Hobbit maraudeur = new Hobbit("Merry");
      maraudeur.mange(pommier.cueille());
      maraudeur.mange(poirier.cueille());
   }
}
```

Q 1 . Critiquez la conception illustrée par ce code. Des remarques courtes et précises sont attendues.

Q 2 . Corrigez les erreurs de conception que vous avez relevées en proposant un diagramme UML et le code Java qui lui correspond.
Vous justifierez clairement pourquoi votre proposition est préférable, en précisant par exemple, le cas échéant, comment vous avez tirer profit de l'application du principe d'inversion des dépendances et du design pattern "factory method".

Exercice 2 : Des phrases

Contexte

L'étude de la structure des langages naturels est un sujet à l'intersection de la linguistique, des neurosciences et de l'intelligence artificielle. Un des objectifs est de pouvoir traiter automatiquement le langage, avec pour ambition de pouvoir faire de la traduction automatique ou de supporter un dialogue naturel (*i.e.* en français dans notre cas) entre l'humain et la machine.
Nous allons décrire ici, une définition formalisée extrêmement simple des phrases de la langue française. Une phrase est un ensemble de *syntagmes*. Ces syntagmes sont regroupés par rapport à leur catégorie.

Tout d'abord, un syntagme nominal (SN) est défini comme un déterminant (dét.), suivi éventuellement d'un ou plusieurs adjectifs (A), suivi(s) d'un nom (N), éventuellement suivi d'un ou plusieurs adjectifs. Ce qui peut se formaliser de la manière suivante :

$$SN \rightarrow dét\ A*\ N\ A*$$

Cette définition permet de construire des syntagmes nominaux comme:

le chien

le petit chien

le petit chien noir

le joli petit chien noir

Ensuite, un syntagme verbal (SV) est défini comme un verbe (V) suivi d'un syntagme nominal (complément du verbe). Ceci peut se formaliser de la manière suivante :

$$SV \rightarrow V\ SN$$

Cette définition permet de construire des syntagmes verbaux comme:

mordre la balle

mordre la vieille balle

mordre la vieille balle jaune

Enfin, une phrase (P) est constituée d'un syntagme nominal (SN), suivit d'un syntagme verbal (SV). Ce qui peut se formaliser de la manière suivante :

$$P \rightarrow SN\ SV$$

Cette définition permet de construire des syntagmes verbaux comme :

le joli petit chien mord la balle

le chien mord la vieille balle

le chien noir mord la vieille balle jaune

A titre d'exemples, voici quelques mots regroupés par catégorie.

Déterminants: le, la, du, des, les.

Adjectifs: petit, joli, vieille, noir, jaune.

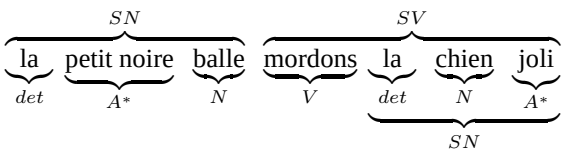
Noms: chien, balle, chat.

Verbes: mordre, jouer, courir.

De par les trois définitions ci-dessus, on peut voir qu'une phrase peut être représentée comme un arbre d'éléments de catégories diverses. La figure 1 propose un exemple d'arbre représentant une phrase en mémoire.
Dans la suite nous ne nous intéresserons qu'à la syntaxe des phrases et ne nous préoccuperons en aucun cas des genres et accords des mots et verbes, ni même de son sens. Ainsi la phrase

la petit noire balle mordons la chien joli

sera considérée comme correcte car elle respecte la syntaxe de phrase définie précédemment :



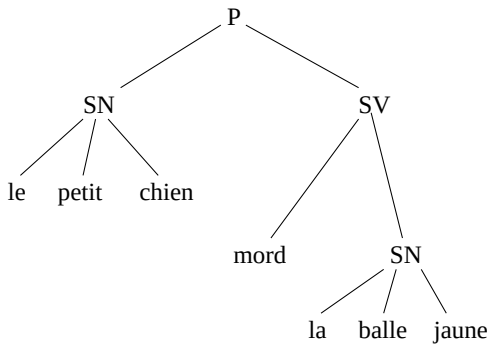


Figure 1: Exemple de représentation d'une phrase

Dans la suite on suppose :

- que les mots seront des instances de la classe `Mot`. Cette classe satisfait les contraintes suivantes :
  - elle contient un constructeur prenant en paramètre une chaîne de caractères (les lettres du mot),
  - elle offre des méthodes `hashCode`, `equals` et `toString` habituelles.
- que les verbes sont conjugués sans auxiliaire (donc pas de forme composée).

Q 1 . On souhaite disposer de quatre "dictionnaires". Un (et un seul) pour chacune des catégories de mots : les déterminants, les adjectifs, les noms et les verbes. Il doit être possible d'ajouter ou de retirer un mot à un dictionnaire et de tester l'appartenance d'un mot à ce dictionnaire.

Ces quatre dictionnaires sont des instances de la même classe qui doivent être accessibles au reste de l'application.

Donnez le **code java** qui vous semble le mieux adapté pour représenter ces dictionnaires.

Dans la suite on suppose que les dictionnaires sont construits et correctement informés. Ainsi le dictionnaires des adjectifs contiendra toutes les formes d'un même adjectif (masculin, féminin, pluriel), celui des verbes toutes les formes conjuguées d'un verbe (à tous les temps et toutes les personnes), etc.

On fera l'hypothèse qu'un mot est présent dans au plus un dictionnaire ainsi le mot *court* sera de manière exclusive soit dans le dictionnaire des noms, soit celui des adjectifs, soit celui des verbes). En conséquence pour savoir si un mot est, par exemple, un adjectif il faut et il suffit de tester si ce mot appartient au dictionnaire des adjectifs.

Q 2 . En **utilisant** le design pattern COMPOSITE, proposez une organisation de classes/interfaces pour représenter les phrases et les différents éléments qui les composent. Votre solution doit contenir au moins une classe `Phrase` et la classe `Mot`.

On souhaite disposer d'un méthode `affiche()` qui affiche les mots constituant l'élément.

Vous ne donnerez ici **que le diagramme UML** en précisant les attributs, constructeurs, méthodes avec leurs modificateurs, types, paramètres (et types), valeurs de retour.

Q 3 . Donnez le **code Java** correspondant au participant *Component* du pattern COMPOSITE de votre proposition.

Q 4 . Donnez le code Java de la classe `Mot`.

Q 5 . Donnez le code Java de **toutes** les classes correspondant aux participants *Composite* du pattern COMPOSITE de votre proposition.

Q 6 . On souhaite **ajouter** à la classe `Phrase` la méthode

```
public static Phrase buildPhrase(String texteDePhrase) throws PhraseMalFormeeException
```

qui à partir d'une chaîne de caractères construit l'objet phrase associé, si la phrase est mal formée (càd. non conforme à la syntaxe présentée dans le sujet) une exception est levée. Bien évidemment cette validité dépend du contenu de chacun des quatre dictionnaires mentionnés précédemment.

Par exemple l'appel de la méthode `build` avec le paramètre `"le petit chien mord la balle jaune"` construira l'objet phrase qui correspond à la racine de l'arbre de la figure 1.

On suppose que que les phrases ne contiennent aucun signe de ponctuation et les mots sont séparés par un et un seul espace.

Vous pourrez utilisez les méthodes des classes rappelées en annexe après avoir remarqué que l'élément pivot d'une phrase est le verbe et donc une fois celui-ci trouvé, il est facile de contruire la phrase.

Donnez le code nécessaire pour cette méthode, en **respectant** l'esprit du design pattern COMPOSITE, c'est-à-dire en laissant chacune des différentes classes de composants construire sa partie. Cela devrait donc vous amener à modifier au moins une partie des autres classes, vous préciserez alors clairement à chaque fois la classe modifiée (inutile de tout réécrire ! précisez seulement la partie ajoutée). Si des interfaces/classes sont ajoutées, vous en donnerez le code complet.

Annexe

Extrait de la javadoc de `java.util.String` et `java.util.StringTokenizer`

`java.lang.String`

`int indexOf(String str)` If the string argument occurs as a substring within this object, then the index of the first character of the first such substring is returned; if it does not occur as a substring, -1 is returned.

`String substring(int beginIndex)` Returns a new string that is a substring of this string.The substring begins with the character at the specified index and extends to the end of this string.

`String substring(int beginIndex, int endIndex)` Returns a new string that is a substring of this string. The substring begins at the specified *beginIndex* and extends to the character at index *endIndex* - 1. Thus the length of the substring is *endIndex-beginIndex*.

`java.util.StringTokenizer`

`public StringTokenizer(String str, String delim)` Constructs a string tokenizer for the specified string. All characters in the *delim* argument are the delimiters for separating tokens. Delimiter characters themselves will not be treated as tokens.

`boolean hasMoreTokens()` Tests if there are more tokens available from this tokenizer's string.

`String nextToken()` Returns the next token from this string tokenizer.