Les patrons *Façade* et *Adaptateur*

Exercice 1 – *Une façade simple.*

Lors du traitement automatique du langage naturel (TALN), il est fréquent de réaliser un découpage d'un flux de texte en une séquence de morceaux correspondant au résultat d'une analyse lexicale. Ces morceaux, appelés *tokens* (ou *lexèmes*) sont identifiés de manière unique et penvent être annotés. Par exemple, la phrase « Un token est un couple. » pourrait être découpées de la manière suivante :

Lexème	Un	token	est	un	couple
Identifiant	0	1	2	0	3
Annotation	[det]	[noun]	[verb]	[det]	[noun]

Le processus de *tokenisation* (ou *analyse lexicale*) consiste donc à découper le flux d'entrée selon des critères (par exemple ici selon les espaces) puis à produire en sortie le flux de tokens correspondant selon un index (une carte des tokens déjà connus), en y ajoutant si nécessaire les tokens inconnus à la volée.

On souhaite ici réaliser un tel index. Pour cela, on s'appuiera sur la classe suivante (fournie) qui représente le concept de token (association d'une chaîne de caractères à un entier) :

```java
package tokenlib;

public class Token {
  // La chaîne de caractères représentant le token :
  private String token;
  // L'identificateur numérique du token :
  private int tokenId;
  // Une annotation quelconque, ex: "Adj", "Verb", "Noun"
  private String annotation;

  public Token(String token, int tokenId, String annotation) {
    this.token=token;
    this.tokenId=tokenId;
    this.annotation=annotation;
  }

  public String getTokenStr() { return token; }
  public int getTokenId() { return tokenId; }
  public String getAnnotation() { return annotation; }
}
```

Notre index devra fournir une relation bi-directionnelle, c'est-à-dire permettant de retrouver efficacement :

— la chaîne d'un token à partir de son identifiant, ou

— un identidiant de token à partir de la chaîne le représentant.

On se propose donc d'utiliser simultanément deux structures de données pour représenter notre index qui permettront d'effectuer efficacement les deux modes de recherche respectivement :

```java
List<Token> idToToken;
Map<String, Token> tokenToId;
```

Question 1.1 : Quel type de collection concret allez-vous choisir pour ces deux collections abstraites ?

Cet index est donc réalisé en utilisant trois types d'objets différents qu'il faut maintenir à jour et synchronisés. La manipulation de ces différentes structures de données étant complexe et pouvant conduire facilement à des erreurs (incohérence structurelle), on propose de les manipuler à travers le jeu d'opérations suivante :

```
int addToken(String tokenStr, String annotation);
String getTokenStr(int tokenId);
int getTokenId(String tokenStr);
String getAnnotation(int tokenId);
Token getTokenMapping(int tokenId);
int getNbTokens();
```

Question 1.2 : Pour chacune des fonctionnalités, décrivez en français le traitement à réaliser.
Remarquons enfin que le problème est instanciable : on pourrait vouloir disposer de plusieurs index.
Question 1.3 : Donnez le diagramme UML de votre solution.
Question 1.4 : Implémentez cette solution. Vous fournirez également un programme de test montrant le bon fonctionnement de votre système.
Question 1.5 : Notre index servant fréquemment à traduire des suites de chaînes de caractères en flux de tokens, on se propose d'ajouter une méthode permettant d'automatiser ce processus qui, pour simplifier, ne traitera pas les annotations (`null` sera utilisé dans les tokens insérés). Celle-ci aura la signature suivante :

```
List<Token> tokenize(iterator<String>);
```

La méthode `tokenize()` construit et renvoie une liste d'identifiants de tokens trouvés (ou ajoutés à la volée) dans l'index, chacun correspondant à la chaîne située à la même position dans l'itération. Lorsque `tokenize()` a besoin d'ajouter un token, elle utilisera l'annotation `"<default>"` par défaut.
Implémentez et testez cette méthode. (Notez que passer un itérateur en paramètre n'est pas une façon de faire très orthodoxe, mais cette méthode servir de prétexte pour l'exercice suivant).

Exercice 2 – *Conception d'un adaptateur*

La bibliothèque `tokenlib`, que nous avons codé à l'exercice précédent, met à notre disposition une classe nous permettant de cartographier les tokens issus d'une séquence de texte. Elle offre également une méthode `tokenize()` permettant d'ajouter une suite de *token* (chaînes de caractères) dans l'index en une seule fois. Nous souhaiterions toutefois pouvoir faire la même chose à partir d'une seule chaîne de caractères sans avoir à effectuer le découpage nous même. Le programme de test suivant montre les deux approches :

```
import java.util.*;

import tokenlib.TokenIndex;
import tokenlib.Utils;

public class Test {
  public static void main(String[] args) {
    TokenIndex ti=new TokenMapStd();  // Votre index.

    // Nous pourrons remplacer :

    LinkedList<String> tokenList=new LinkedList<String>();
    tokenList.add("Ceci");
    tokenList.add("est");
    tokenList.add("une");
    tokenList.add("liste");
    tokenList.add("de");
    tokenList.add("tokens");
    Iterator<String> it=tokenList.iterator();
    // Ajoute la séquence à l'indexe :
    List<Integer> is=ti.tokenize(it);
    Utils.displayTokenSequence(is); // Séquence de tokens
    Utils.displayTokenIndex(ti);  // État de l'indexe

    // Par :

    Iterator<String> it2=Utils.tokenize("Ceci est une manière plus simple de tokenizer");
    is=ti.tokenize(it2);
```

```
29      Utils.displayTokenSequence(is); // Séquence de tokens
30      Utils.displayTokenIndex(ti);  // État de l'indexe
31    }
32  }
```

Pour ce TP, une classe utilitaire `tokenlib.Utils` vous est donnée qui fournie les méthodes d'affichage employées dans ce test. Vous devrez écrire le contenu de la méthode `tokenize()` de la classe `tokenlib.Utils`.

```
1  package tokenlib;
2  import java.security.InvalidParameterException;
3  import java.util.Iterator;
4  import java.util.List;
5  import java.util.StringTokenizer;
6
7  public abstract class Utils {
8    public static Iterator<String> tokenize(String str) {
9      // TODO !
10   }
11
12   public static void displayTokenSequence(List<Integer> li) {
13     if (li==null) throw new InvalidParameterException("Null List.");
14     System.out.println("Token sequence "+li);
15   }
16
17   public static void displayTokenIndex(TokenIndex ti) {
18     if (ti==null) throw new InvalidParameterException("Null token index.");
19     System.out.println("TokenIndex "+ti+" :");
20     for (int i=0; i<ti.getNbTokens(); ++i)
21       System.out.println("\t"+i+" : "+ti.getToken(i)+" ["+ti.getAnnotation(i)+"]");
22   }
23 }
```

Afin de ne pas se lancer dans le processus complexe de découpage d'une chaîne de caractères, on observe que la classe `java.util.StringTokenizer` effectue parfaitement ce travail.
Note : on souhaite bien évidemment continuer à utiliser la bibliothèque `tokenlib`, sans la modifier, pour l'affichage et la manipulation de la liste de tokens produite.
Question 2.1 : Recherchez et lisez la documentation javadoc de l'interface *Iterator* sur internet. Attention ! Vérifiez que vous consultez bien la documentation version actuelle (ou ultérieure à *Java 2 Platform SE 5.0*) !
Question 2.2 : Maintenant, à l'aide de la javadoc, identifiez quelle interface, implémentée par *StringTokenizer*, est la plus proche de l'interface *Iterator*.
Question 2.3 : Identifiez, dans le code existant, les différents acteurs du patron *adaptateur*.
Question 2.4 : Rappelez les différents types d'adaptateurs vus en cours. Lesquels sont compatibles avec le programme à réaliser ?
Question 2.5 : Lequel allez-vous choisir d'implémenter ? Argumentez.
Question 2.6 : Dessinez le diagramme *UML* correspondant à votre solution en identifiant les participants (nom des acteurs du patron) à coté de chaque entité.
Question 2.7 : Programmez et testez l'adaptateur correspondant.
Question 2.8 : Si ce n'est déjà fait, proposez (et implémentez) une version la plus générique possible de l'adaptateur, c'est-à-dire adaptant les types les plus hauts possibles dans la hiérarchie. Cet adaptateur ultra-générique pourra être placé dans un paquetage utilitaire séparé puisqu'il n'est plus intrinsèquement lié à notre bibliothèque, et votre adaptateur « métier » pourra être ré-écrit en héritant de cet adaptateur générique. Attention toutefois, il vous faudra pour cela à un moment donné utiliser un transtypage non vérifié en raison de l'obsolescence de l'API `StringTokenizer`.

EXERCICES BONUS

Exercice 3 – *Visualiseur de liste de tokens*

Nous allons reprendre les travaux réalisés à l'exercice précédent et procéder à l'affichage graphique d'une liste de tokens sur deux colonnes de façon similaire à `TokenLib.displayTokens()` (la première sera le numéro séquentiel du token, la second le token). La classe suivante permet d'ouvrir une fenêtre et d'y afficher une table. Le constructeur attend en paramètre un objet dont la classe implémente l'interface `javax.swing.table.TableModel` :

```
1  import java.util.*;
2  import java.lang.*;
3  import javax.swing.*;
4  import java.awt.*;
5  import javax.swing.table.*;
6
7  public class TableViewer {
8       // Display a Swing component.
9       public static void display(Component c, String title) {
10           JFrame frame = new JFrame(title);
11           frame.getContentPane().add(c);
12           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13           frame.pack();
14           frame.setVisible(true);
15      }
16
17    public TableViewer(TableModel tm) {
18           JTable table = new JTable(tm);
19           JScrollPane pane = new JScrollPane(table);
20           pane.setPreferredSize(new java.awt.Dimension(300, 200));
21           display(pane, "Token Viewer");
22
23    }
24 }
```

Question 3.1 : Que devez-vous fournir pour faire en sorte que ce composant graphique soit en mesure d'exploiter une liste de tokens ?

Question 3.2 : Lisez la documentation javadoc de l'interface `TableModel` et rappelez le rôle de la classe `AbstractTableModel`. De quel type d'adaptateur s'agira-t-il si vous décidez de l'utiliser ?

Question 3.3 : Donnez le diagramme UML de votre solution annoté avec le nom des acteurs.

Question 3.4 : Programmez l'adaptateur correspondant.

Exercice 4 – *Façades multiples d'un sous-système.*

Une entreprise de vente de véhicules souhaite permettre à ses clients d'accéder à la liste des véhicules en vente ainsi que de commander de la documentation papier, via un service web. Voici les éléments de base de ce système d'information. Les véhicules sont représentés par :

```
1  public interface Vehicule {
2    /* returns the price of the vehicule */
3    public int getPrice();
4    /* returns the name of the vehicule */
5    public String getName();
6    /* gives an html description of the vehicule */
7    public String getWebDescription();
8    /* gives a text description for printing */
9    public String getPrintableDocument();
10 }
```

et sont stockés dans une base de données représentée par un objet *DatabaseImpl* implémentant :

```
1  public interface Database {
2    /* returns a vehicule by its name */
3    public Vehicule getVehicule(String name);
4    /* returns a list of vehicules by arange of prices */
5    public List<Vehicule> getVehiculesByPrice(int low, int hi);
```

```
    /* add a vehicule into the database */
    public void addVehicule(Vehicule v);
}
```

Les services d'impression et de mailing sont représentés par les classes *PrinterImpl* et *MailingImpl* implémentant respectivement :

```
public interface Printer {
  /* print the documentation and return the job id */
  public int print(String doc);
}

public interface Mailing {
  /* send a documentation with a given print id to a given address and return
   * a confirmation message */
  public String send(int printid, String address);
}
```

L'imprimante par défaut s'obtient par

```
Printer p=PrinterImpl.getDefaultPrinter();
```

On souhaite disposer de deux types d'accès métiers différent :

- Un service web pour les clients permettant de rechercher un véhicule par son nom, par intervalle de prix ou de commander de la documentation,
- Une interface d'administration permettant la gestion du parc de vehicule.

Voici les deux interfaces de ces deux façades :

```
public interface FacadeWeb {
  public String findVehicule(String name);
  public String findVehiculeByPrice(int lo, int hi);
  public String orderVehiculeDoc(String name, String address);
}

public interface FacadeAdmin {
  public void addVehicule(String name, int price, String wdoc, String pdoc);
}
```

Question 4.1 : Dessinez le diagramme UML du système complet,
Question 4.2 : implémentez les deux façades (récupérez les classes de base sur votre liste de diffusion),
Question 4.3 : testez les avec l'exemple de client suivant :

```
package tpfacade;

public class Client {
  public static void main(String[] args) {
    System.out.println("Debut client");

    FacadeAdmin admin=new FacadeAdminImpl();
    admin.addVehicule("Berline 5 portes", 6000,
        "Compact 3 portes<br>Moteur diesel<br>Neuve<br>",
        "Compact 3 portes\nMoteur diesel\nNeuve");
    admin.addVehicule("Espace 5 portes", 8000,
        "Espace 5 portes<br>Moteur essence<br>Neuve<br>",
        "Espace 5 portes\nMoteur essance\nNeuve");
    admin.addVehicule("Coupe 2 portes", 3000,
        "Utilitaire 3 portes<br>Moteur diesel<br>Occasion<br>",
        "Utilitaire 3 portes\nMoteur diesel\nOccasion");

    FacadeWeb web=new FacadeWebImpl();
    System.out.println("requete espace 5 portes :\n" +
        web.findVehicule("Espace 5 portes"));
    System.out.println("requete vehicule a moins de 7000 euros :\n" +
```

```
22          web.findVehiculeByPrice(0, 7000));
23
24      System.out.println("Demande de documentation :\n" +
25          web.orderVehiculeDoc("Berline 5 portes",
26          "Avenue de l'universite, 76800, Saint Etienne-du-Rouvray"));
27      System.out.println("Demande de documentation :\n" +
28          web.orderVehiculeDoc("Coupe 2 portes",
29          "Avenue de l'universite, 76800, Saint Etienne-du-Rouvray"));
30      System.out.println("Fin client");
31    }
32  }
```

Produisant la sortie suivante :

```
1  Debut client
2  requete espace 5 portes:
3  <html><body>
4  Espace 5 portes<br>Moteur essence<br>Neuve<br>
5  </body></html>
6
7  requete vehicule a moins de 7000 euros:
8  <html><body>
9  Compact 3 portes<br>Moteur diesel<br>Neuve<br>
10 Utilitaire 3 portes<br>Moteur diesel<br>Occasion<br>
11 </body></html>
12
13 Demande de documentation:
14 <html><body>
15 The documentation order no 1 will be sent to Avenue de l'universite, 76800, Saint Etienne-
       du-Rouvray
16 </body></html>
17
18 Demande de documentation:
19 <html><body>
20 The documentation order no 2 will be sent to Avenue de l'universite, 76800, Saint Etienne-
       du-Rouvray
21 </body></html>
22
23 Fin client
```