TD 9 : Adapter et Factory

Objectifs pédagogiques : manipulation des design patterns *Adapter* et *Factory*.

9.1 Télécommande universelle

9.1.1 Introduction

Pour pouvoir contrôler une télévision de marque *Soni*, on dispose d'une télécommande. Une télévision *Soni* est implémentée comme suit :

TVSoni.java

```java
package pobj.tele;

public class TVSoni {
  private final String modele;
  private boolean on = false;
  private int volume = 0;

  public TVSoni(String modele) { this.modele = modele; }
  public void powerOnOff() { on = !on; }
  public void volumeUp() { if (volume < 100) volume++; }
  public void volumeDown() { if (volume > 0) volume--; }
  public void mute() { volume = 0; }
}
```

La télécommande ne permet que de contrôler des télévisions `TVSoni` (par des appels aux différentes méthodes de sa classe).

⇒ Proposez une classe (`Telecommande`) capable de contrôler une `TVSoni`.

9.1.2 Adaptateur Soni vers Sarsung

Suite à l'achat d'une télévision de marque *Sarsung*, on aimerait que la télécommande *Soni* puisse commander la télévision *Sarsung*. Malheureusement, les télécommandes *Soni* ne commandent que des objets de type `TVSoni`. Étant donné cette implémentation de la classe `TVSarsung` :

TVSarsung.java

```java
package pobj.tele;

public class TVSarsung {
  public final int reference;
  private int volume = 0;
  private int etat = 0; // 0 éteint, 1 en veille, 2 allumé

  public TVSarsung(int reference) { this.reference = reference; }
  public int getEtat() { return etat; }
  public int getVolume() { return volume; }

  public void allume() { etat = 2; }
  public void veille() { etat = 1; }
  public void eteint() { etat = 0; }
  public void augmenteSon() { if (volume < 127) volume++; }
  public void diminueSon() { if (volume > 0) volume--; }
}
```

⇒ Proposez une solution (classe `TVSoniSarsungAdapter`) qui permettrait d'utiliser la télécommande *Soni* sur la télévision *Sarsung*.

9.1.3 Télécommande universelle

Avec la solution précédente, si *Soni* décide de produire de nouvelles télévisions (sous-classes de `TVSoni`), chaque nouvelle télécommande ne fonctionnera pas avec votre adaptateur.

⇒ À l'aide d'interfaces et en modifiant les classes `TVSoni` et `TVSoniSarsungAdapter`, comment régler ce problème ? Dessinez le diagramme de classe.

9.2 Une configuration *singleton*

On souhaite construire une application hautement configurable. Pour cela on propose de loger les options de configuration dans une classe `Configuration`, qui associe les noms des paramètres à leurs valeurs. Pour préserver une bonne généralité, on considérera que les paramètres et leurs valeurs sont des `String`.

Quelle structure de données vous paraît la plus adaptée pour stocker cette information ?

⇒ **Définissez la classe `Configuration`**, avec ses attributs, un constructeur qui initialise une configuration vide, et les méthodes `String getParamValue(String param)` et `void setParamValue(String param, String value)`.

On souhaite stocker les configurations dans des fichiers XML. La syntaxe d'un tel fichier est la suivante :

```
<configuration>
      <parametre name='param1' value='valeur1'/>
      <parametre name='param2' value='valeur2'/>
      ...
</configuration>
```

⇒ **Ecrivez une méthode statique qui**, étant donné une `Configuration` et une `String cheminFile` désignant un nom de fichier, **construit un fichier ayant cette forme**.

NB : On pourra utiliser des `append()` dans un `Writer` :

```
FileOutputStream fos = new FileOutputStream(cheminFile);
Writer out = new OutputStreamWriter(fos);
out.append("hello world");
out.close();
fos.close();
```

Pour *parser* des fichiers XML, on utilise fréquemment l'API SAX. Dans ce système, le parse du fichier est découpé en deux parties :

- L'utilisateur doit fournir une classe qui étende le `DefaultHandler` fourni avec l'API. Dans cette classe il pourra redéfinir le comportement des opérations `startElement`, `endElement`.
- Le `SAXParser` lit le fichier source à la recherche de balises XML. Au moment de lancer le parse, on lui passe un `Handler`. Pour chaque (début ou fin de) balise, il émet un événement en invoquant une opération du `Handler`. Par exemple, pour chaque balise rencontrée il invoquera `startElement`, en lui passant l'URI du `namespace` (non utilisé ici), le nom local de l'objet (non utilisé ici), et la `String` qui correspond au nom de la balise (« parametre » ou « configuration » pour nos fichiers). Il lui passe aussi sous la forme d'un `Attributes`, un ensemble de clé/-valeur pour les propriétés de la balise (ici «name» et «value» sont les propriétés de la balise «parametre»).

La classe suivante représente une partie du code à définir pour parser nos fichiers.

```java
package pobj.conf;

public class ConfigHandler extends DefaultHandler {
  @Override
  public void startElement(String uri, String localName, String qName, Attributes attributes
      ) throws SAXException {
    if ("parametre".equals(qName)) {
      luParametre(attributes.getValue("name"), attributes.getValue("value"));
    }
  }

  private void luParametre(String name, String value) {
    // TODO Auto-generated method stub
  }
}
```

Ce code permet de lancer le parse d'un fichier :

```java
public static Configuration importConfigurationData(String fileName) throws IOException,
    ParserConfigurationException, SAXException {
  SAXParserFactory factory = SAXParserFactory.newInstance();
  SAXParser parser = factory.newSAXParser();

  Configuration conf = null ; // TODO : corriger
  ConfigHandler parseHandler = new ConfigHandler();
  parser.parse(fileName, parseHandler);

  return conf; // TODO : pas affecté
}
```

⇒ Reconnaissez-vous le pattern *singleton* dans ce code ? Quelle classe joue le rôle de `factory` ? Expliquez l'instanciation qui est faite ici du pattern Stratégie.

⇒ Comment faire circuler l'information entre les différentes classes pour correctement construire une Configuration ? Ajoutez un attribut typé `Configuration` dans la classe `ConfigHandler` et complétez et corrigez le code de chargement depuis XML.

⇒ Comment modifier le code de `Configuration` pour en faire un *singleton* ?