Plug-ins

(les fichiers mentionnés dans ce document sont disponibles sur le portail)

Préliminaire

L'objectif de ce TP est la mise en place *progressive* de cette "application" qui s'adapte dynamiquement en fonction de ressources (les "plug-ins") disponibles dans un répertoire.
Ce TP sera l'occasion de voir la mise en place du design-pattern observer et d'utiliser certains aspects de la réflexivité. La mise en place du design pattern *Observer* est détaillée étape par étape dans le poly de cours ! Relisez donc cette partie avant de vous lancer !

Un exemple de l'application que vous devez obtenir est disponible. Pour cela récupérez l'archive `fichiers-plugins.zip` disponible sur le portail et décompressez la dans votre espace de travail. Vous devez avoir maintenant un répertoire `plugins` contenant les répertoires `classes`, `disponibles`, `extensions` et `src`. Placez vous dans le répertoire `plugins` et éxécutez la commande :

```
java -classpath .:classes:extensions edit extensions &
```

La fenêtre qui apparaît contient une barre de menu (un seul menu pour l'instant) que vous pouvez tester.
Copiez un par un les `.class` situés dans `disponible` dans `extensions` et observez ce qui se passe entre chaque copie. Vous pouvez y placer un `.class` quelconque (et normalement rien ne se passe) car ne sont pris en compte que l'ajout de classes implémentant l'interface :

```
public interface Extension {
  public String filtrer(String s) ;
  public String toString() ;
}
```

et ne faisant partie d'aucun paquetage (donc en fait faisant partie du paquetage par défaut).

Quand une telle classe est ajoutée dans le répertoire `extensions`, (qui est l'argument passé à l'application sur la ligne de commande), un item de menu dont le nom correspond au retour de l'invocation de `toString()` sur une instance de la classe est ajouté dans le menu *Outils* et l'activation de cet item applique la transformation définie dans `public String filtrer(String s)` sur le texte du champ de texte de l'application.

Vous allez donc dans ce TP implémenter cette application.

Un vérificateur de nouveaux fichiers `.class`

Dans un premier temps, on va se contenter de détecter l'apparition de fichier d'extension `.class` dans le répertoire `extensions`.
Pour permettre à l'application de s'enrichir dynamiquement, celle-ci doit être avertie régulièrement de l'évolution du contenu du répertoire `extensions`[1]. L'ajout d'un nouveau `.class` correspond à un évènement auquel l'application réagit.
Pour pouvoir faire cela nous allons mettre en œuvre le design-pattern *Observer* vu en cours. Il s'agira d'émettre un évènement à chaque fois qu'un nouveau plug-in est ajouté dans lé répertoire spécifié. L'application sera abonné à ces évènements et réagira donc en conséquence.
L'objet émetteur de l'évènement doit donc examiner régulièrement le contenu du répertoire. Nous allons donc utiliser une instance de `javax.swing.Timer`. Cette classe permet de définir des objets réalisant régulièrement une certain tâche (ici examiner le répertoire). Son constructeur prend comme paramètre un `delay` (`int`) et un `ActionListener` dont la méthode `actionPerformed` est déclenchée tous les `delay` millisecondes une fois que le timer a été démarré (`start()`).
Nous souhaitons qu'en l'occurrence l'`ActionListener` examine le contenu du répertoire `extensions` et informe de l'apparition d'une **nouvelle** classe "intéressante". On peut pour cela utiliser la méthode `String[] list(FileNameFilter)` de la classe `File`. Qui retourne le tableau des noms de fichier acceptés (`accept()`) par l'objet de type `FileNameFilter` passé en paramètre.
Il est assez aisé de gérer également la suppression des classes.
Pour chaque nom accepté on déclenchera un évenement (appelé `PluginEvent` dans la suite) pour avertir les applications intéressées par l'ajout d'une nouvelle classe. On applique ici la démarche présnetée dans le notes de cours pour le design pattern Oberser/Observable.

[1] On rappelle que le nom de ce répertoire est communiqué au démarrage de l'application.

Q 1 . Créez une classe `PluginEvent` qui correspondra à l'évènement émis lorsqu'un nouveau plug-in est ajouté. Le nom de la classe du plug-in impliqué est passé en paramètre du constructeur de l'évènement.

Q 2 . Créez l'interface de `PluginListener` associé à ces évènements et définissant les méthodes `pluginAdded` et `pluginRemoved`.

Q 3 . Créez une classe `NewPlugInChecker` qui sera l'émettrice des évènements `PluginEvent`. Celle-ci dispose du timer évoqué et déclenche les évènements à chaque ajout :

Q 3.1. Créez une classe implémentant `FileNameFilter` dont la méthode `accept` renvoie `true` pour tous les fichiers `*.class`.

Q 3.2. Créez la classe implémentant `ActionListener` et dont une instance est fournie au timer. Dans la méthode `actionPerformed`, pour chacun des **nouveaux**[2] fichiers acceptés du répertoire `extensions`[3] on déclenche un évènement d'ajout (`firePluginAdded(...)`) ; pour les fichiers supprimés on déclenche un évènemment de suppression (`firePluginRemoved(...)`).

Q 3.3. Complétez la classe `NewPlugInChecker` pour qu'elle permette la gestion des "plug-in listeners" et la propagation de l'évènement vers ces listeners.

Q 4 . Créez une classe implémentant `PluginListener` qui affiche le message "`nouveau .class : xxxxx détecté dans extensions`" pour chaque évènement émis.

Q 5 . Testez l'ensemble (n'oubliez pas de démarrer le timer ! et pour éviter que l'application ne se termine aussitôt, placez un superbe `while (true) ;` à la fin de votre `main`).

Il vous faut préciser que le répertoire de travail est le répertoire `classes`, dans ECLIPSE il faut aller dans le "2nd onglet en bas" dans menu de paramétrage de Run....

Plug-ins

Nous allons poser certaines contraintes pour définir ce qui sera considéré comme un plug-in. Nous allons ainsi supposer que les classes candidates doivent :

- implémenter l'interface `Extension`
- n'appartenir à aucun paquetage ("default package")[4].
- fournir un constructeur sans paramètre.

Q 6 . Créez une nouvelle classe implémentant `FileNameFilter` qui n'accepte que les plug-ins : il faut donc avoir un `.class`, puis charger l'instance `Class` associée (`Class.forName(...)`) et regarder si c'est une classe qui satisfait les conditions mentionnées ci-dessus.

Q 7 . Dans l'"application" précédente, remplacez le filtre actuel par celui-ci (il faut mettre le répertoire `extensions` dans le classpath pour permettre au `forName` de bien fonctionner, sous ECLIPSE vous pouvez l'adapter par le 4ème onglet du menu de paramétrage de Run...) et testez.

Q 8 . Créez l'application complète. Celle-ci se base sur une nouveau `PluginListener` à définir pour gérer les menus. Vous pouvez imaginer d'autres extensions que celles fournies.

Menus (très vite)

Les classes permettant de gérer des menus sont dans le paquetage `javax.swing` :

JMenuBar la barre de menu

JMenu un menu, que l'on ajoute à un `JMenuBar` par `add`

JMenuItem un élément du menu, que l'on ajoute à un `JMenu` pas `add`

On peut abonner des `ActionListener` à un `JMenuItem`, leur méthode `actionPerformed` est déclenchée lors que l'on clique sur l'item.

[2] ne pas déclencher l'évènement 2 fois pour le même fichier
[3] `new File("extensions").list(une instance de votre FileNameFilter)`
[4] Pour simplifier.