

UE Conception Orientée Objet

plugins

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

Préliminaire

L'objectif de ce TP est la mise en place *progressive* d'une application qui s'adapte dynamiquement en fonction de "plugins" installés 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 le TP !

Un exemple de l'application que vous l'on va réaliser au cours de ce TP 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 exécutez la commande :

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

La fenêtre qui apparaît contient une barre de menu. L'item du menu Outils propose un certain nombre de fonctionnalités (une pour l'instant). Ces outils correspondent à des opérations qui s'appliquent au texte contenu dans la zone de texte constituant la partie inférieure de l'application.

Vous pouvez tester cette application en appliquant l'outil proposé sur un texte (que vous saisissez au clavier ou chargez depuis le menu Fichier).

Cette application a la particularité qu'il est possible de l'enrichir en lui ajoutant dynamiquement, c'est-à-dire pendant qu'elle est en cours d'exécution, de nouveaux outils. Ce sont ces outils qui vont constituer les plugins évoqués ci-dessus.

Un certain nombre de plugins sont proposés, il se trouve dans le dossier `disponibles`.

Copiez un par un les `.class` situés dans le répertoire vers `extensions`, observez ce qui se passe entre chaque copie dans le menu Outils et tetez chacun des nouveaux outils proposés. Vous pouvez placer dans le même dossier un autre fichier (y compris d'extension `.class`) quelconque et, normalement, rien ne se passe au niveau de l'application. Pour être reconnu comme étant un plugin un fichier doit en effet satisfaire un certain nombre de contraintes.

Il est également possible de supprimer un outil, il vous suffit pour cela d'effacer le fichier `.class` correspondant du répertoire `extensions`.

L'objectif de ce TP sera l'implémentation de 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` (donc sans s'occuper s'il s'agit ou non de "plugins" pour ce premier temps).

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 plugin est ajouté dans le répertoire spécifié. L'application sera abonée à ces évènements et réagira donc en conséquence.

L'objet émetteur de l'évènement doit donc examiner régulièrement le contenu du répertoire. Nous allons pour faire cela utiliser une instance de la classe `javax.swing.Timer`. Cette classe permet de définir des objets réalisant régulièrement une certaine 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()`).

Q 1 . Lisez la javadoc de `javax.swing.Timer`.

Q 2 . Pour tester votre compréhension de cette classe, écrivez un programme qui crée un timer dont la fonction est d'afficher sur la sortie standard la date courante (`java.util.Calendar.getInstance().getTime()` toutes les secondes puis démarre ce timer.

Nous souhaitons qu'en l'occurrence l'`ActionListener` du timer 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`. Elle retourne le tableau des noms de fichier acceptés (`accept()`) par l'objet de type `FileNameFilter` passé en paramètre.

La classe `File` permet de créer des objets qui "représentent" un répertoire et non simplement un fichier. On doit donc créer un objet représentant le répertoire `extensions`, un `FileNameFilter` qui accepte les fichiers d'extension `.class` et "lister" ce répertoire avec ce filtre.

Q 3 . Lisez la javadoc de `java.io.FileNameFilter`

Q 4 . Lisez la javadoc de la méthode `public String[] list(FileNameFilter filter)` de la classe `java.io.File` (lire le texte d'introduction de la javadoc de cette classe est certainement utile également).

Pour chaque nom accepté on déclenchera un évènement (appelé `PluginEvent` dans la suite) pour avertir les applications intéressées par l'ajout d'une nouvelle classe. On applique ici la démarche présentée dans les notes de cours pour le design pattern Observer/Observable.

Il est assez aisé de gérer également la suppression des classes.

Q 5 . Créez une classe `PluginEvent` qui correspondra à l'évènement émis lorsqu'un nouveau plugin est ajouté.

Le nom de la classe du plugin impliqué est passé en paramètre du constructeur de l'évènement.

Q 6 . Créez l'interface de `PluginListener` associé à ces évènements et définissant les méthodes `pluginAdded` et `pluginRemoved`.

Q 7 . 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 7.1. Créez une classe implémentant `FileNameFilter` dont la méthode `accept` renvoie `true` pour tous les fichiers `*.class`.

Q 7.2. 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 évènement de suppression (`firePluginRemoved(...)`).

Q 7.3. Complétez la classe `NewPluginChecker` pour qu'elle permette la gestion des "plugin listeners" et la propagation de l'évènement vers ces listeners.

Q 8 . Créez une classe implémentant `PluginListener` qui affiche le message "nouveau `.class` : `xxxxx` détecté dans `extensions`" pour chaque évènement émis.

Q 9 . 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...

Plugins

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

- implémenter l'interface `Extension` (éventuellement indirectement - cf. `isAssignableFrom` dans la classe `Class`)

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

- n'appartenir à aucun paquetage ("default package")⁴.

- fournir un constructeur sans paramètre.

Q 10 . Créez une nouvelle classe implémentant `FileNameFilter` qui n'accepte que les plugins : 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 11 . 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.

²ne pas déclencher l'évènement 2 fois pour le même fichier

³`new File("extensions").list(une instance de votre FileNameFilter)`

⁴Pour simplifier.

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

Q 12 . Créez maintenant l'application complète, avec l'interface graphique. Celle-ci se base sur une nouveau `PluginListener` à définir pour qu'il gère l'ajout ou le retrait d'élément de menu lors de l'apparition ou de la disparition de fichier de plugin.

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. (`JMenuItem` est en fait une sous-classe de `AbstractButton`).

Jetez un œil au lien "*How to Use Menus*" aux documentations des classes `JMenu`, `JMenuItem` et `JMenuBar` du paquetage `javax.swing`.