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`, `depot` et `src`. Placez vous dans le répertoire `plugins` et éxécutez la commande :

```
java -classpath classes:depot exercice5.Main &
```

La fenêtre qui apparaît contient une barre de menu. L'item du menu Tools 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 File).

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, ils se trouvent dans le dossier `disponibles`.
Copiez un par un les `.class` situés dans de répertoire vers `depot`, observez ce qui se passe entre chaque copie dans le menu Tools 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 `depot`.

L'objectif de ce TP sera l'implémentation de cette application. Nous allons procéder progressivement. Vous devez valider chaque étape (chaque exercice) avant de passer au suivant.

Exercice 1 : Les timers
La classe `javax.swing.Timer` permet de définir des objets exécutant régulièrement une certaine tâche[1]. 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 grâce à la méthoode `start()`.

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

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

Vous créerez en classe interne le type d'`ActionListener` dont vous aurez besoin.

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`.

[1] La classe `java.util.Timer` aurait aussi pu être utilisée.

Exercice 2 : Lister le contenu d'un répertoire
La classe `File` permet de créer des objets qui "représentent" des fichiers mais aussi un répertoire.

Q 1 . Lisez la javadoc de `java.io.FilenameFilter` et en particulier 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). Cette méthode a pour résultat le tableau des noms des fichiers acceptés (`accept()`) par l'objet de type `FilenameFilter` passé en paramètre.

Q 2 . En vous appuyant sur l'écriture de 2 classes qui implémentent cette interface, écrivez une classe dont les instances sont paramétrées par un répertoire dont le nom (chemin) est donné à la création et dispoe de deux méthodes :

1. l'une pour afficher les noms de tous les fichiers de ce répertoire dont le nom commence par un `C`
2. l'autre pour affiches les noms de tous les fichiers de ce répertoire dont l'extension est `.class`

Exercice 3 : Un vérificateur de nouveaux fichiers `.class`
Nous allons maintenant nous intéresser à la mise en place d'un mécanisme événementiel pour gérer la détection de l'apparition de fichiers dans un répertoire choisi.
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 fichier est ajouté dans le répertoire spécifié. Un objet "client" sera abonné à ces évènements et réagira donc en conséquence.
L'objet émetteur de l'évènement, que nous appellerons `FileChecker`, examinera régulièrement le contenu du répertoire concerné à l'aide d'un objet `Timer`.
Nous souhaitons qu'en l'occurrence l'`ActionListener` du timer examine le contenu du répertoire voulu et informe de l'apparition d'un **nouveau** fichier `.class`. Cela sera fait à l'aide d'un objet `FilenameFilter` comme vu précédemment. Pour chaque nom accepté par ce filtre on déclenchera un évènement (appelé `FileEvent` dans la suite) pour avertir les objets abonnés (les "*Observer*").

On applique ici la démarche présentée dans le notes de cours pour le design pattern Observer/Observable.

Q 1 . Créez une classe `FileEvent` qui correspondra à l'évènement émis lorsqu'un nouveau plugin est ajouté.

Le nom du fichier impliqué est passé en paramètre du constructeur de l'objet évènement et mémorisé.

Q 2 . Créez l'interface `FileListener` associée à de tels évènements et définissant le type des obervateurs pour ces événements. Cette interface imposera la méthode `fileAdded`.

Q 3 . Créez une classe `FileChecker` qui sera la classe des objets émetteurs des évènements `FileEvent`. Celle-ci dispose du timer évoqué et déclenche les évènements à chaque ajout. Le constructeur de cette classe prendra en paramètre l'objet `FilenameFilter` qui permet de filtrer les fichiers ainsi que le chemin du répertoire surveillé.

Il faut donc :

Q 3.1. gérer les `FileListener` qui pourront être les observateurs de cette classe. Il faut donc ajouter le couple de méthode permettant l'abonnement et le désabonnement de ces *listeners*.

Q 3.2. coder la méthode `fireFileAdded` qui crée et propage propage l'événement à tous les *listeners* abonnés. Le nom du fichier pourra être passé en paramètre de cette méthode.

Q 3.3. créer la classe interne implémentant `ActionListener` et dont une instance est fournie au timer. Dans la méthode `actionPerformed`, pour chacun des **nouveaux** fichiers "*acceptés*" du répertoire choisi on propage un évènement d'ajout grâce à l'appel à `fireFileAdded(...)`.

Il sera nécessaire de maintenir la liste des fichiers "connus" par le `FileChecker`.

Q 4 . Créez une classe implémentant `FileListener` et dont la seule réaction à l'événement d'ajout de fichier est d'afficher le message "`nouveau .class :` *xxxxx* `détecté`" pour chaque évènement émis.

Q 5 . Testez l'ensemble. N'oubliez pas d'abonner le `FileListener` au `FileChecker` ni de démarrer le timer du `FileChecker` (à nouveau un `while (true);` sera nécessaire).

Q 6 . On souhaite maintenant détecter également la suppression de fichiers. Faites les modifications nécessaires :

- dans `FileListener` pour ajouter une méthode `fileRemoved`
- dans `FileChecker` pour détecter la disparition d'un fichier "connu" et propager l'événement à 'aide d'une méthode `fireFileRemoved(...)`
- dans la classe implémentant `FileListener` pour prendre en compte la modification de l'interface, on affichera dans ce nouveau cas "`.class` *xxxxx* `supprimé détecté`".

Exercice 4 : Première prise en compte des plugins
Nous allons maintenant nous attaquer à l'application initialement annoncée en étendant ce qui a déjà été réalisé.
Commençons par poser certaines contraintes pour définir ce qui sera considéré comme un plugin. Nous allons ainsi supposer que les classes candidates doivent :

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

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

- n'appartenir à aucun paquetage ("default package")[2].
- fournir un constructeur sans paramètre (cf. `getConstructor()` dans `Class`).

Q 1 . Créez une nouvelle classe `PluginFilter` implémentant `FilenameFilter` qui n'accepte que les fichiers correspondant à un plugin. Il faut donc :

1. avoir un `.class`,
2. charger l'instance `Class` associée grâce à `Class.forName(...)` (attention à supprimer l'extension du nom du fichier),
3. vérifier que c'est une classe qui satisfait les conditions mentionnées ci-dessus.

Q 2 . Ecrivez une classe `SimplePluginObserver` qui affiche un message d'information pour chaque fichier *plugin* ajouté ou retiré du répertoire `extensions`.

Q 3 . Testez votre `SimplePluginObserver` avec un `FileChecker` basé sur un`PluginFilter`

NB : *Il faut que le répertoire contenant les `.class` des plugins soit dans le classpath pour permettre au `forName` de bien fonctionner. Si besoin sous* ECLIPSE *vous pouvez l'adapter par le 4ème onglet du menu de paramétrage de Run...). sous Eclipse, vous pouvez également adapter le répertoire de travail du programme dans le 4ème onglet du menu de paramétrage de Run...) (tout en bas).*

Exercice 5 : L'application graphique
On va maintenant créer l'application demandée complète, c'est-à-dire avec l'interface graphique.
Outre le fait de disposer d'un `FileChecker` qui utilise le `PluginFilter` précédent, l'application se base sur un nouveau type implémentant `FileListener` à définir pour qu'il gére l'ajout ou le retrait d'élément de menu lors de l'apparition ou de la disparition de fichier de plugin.
Il s'agit en fait d'ajouter (ou de supprimer) des `JMenuItem` dans un `JMenu` défini. Le label de cet item est le `getLabel` du plugin et l'action sur ce `JMenuItem` consiste à appeler la méthode `transformer` du plugin. Il faudra donc construire une instance de la classe de plugin détectée (voir `newInstante` dans `Class`) et la lier au `JMenuItem`. Il sera certainement pertinent de définir une classe de `JMenuItem` spécifique.

Q 1 . Réalisez l'application avec l'interface graphique.

Pour la zone de texte vous pouvez utiliser un `JScrollPane` et voir ci-dessous pour les menus. Si vous souhaitez le mettre en place, pour le choix d'un fichier texte à afficher vous pouvez utiliser un `JFileChooser`.

Vous pouvez imaginer d'autres extensions que celles suggérées.

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` par `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`.

[2]Pour simplifier.