

Création d'un système de plugins en java

par [Lainé Vincent](#)

Date de publication : 15/02/2006

Dernière mise à jour : 15/02/2006

Dans cet article nous aborderons la mise en oeuvre d'un système de plugins en java.

- I - Introduction
- II - Conception du système
 - II-A - Problématique
 - II-B - Les différentes classes et interface(s)
- III - Programmation du système
 - III-A - Le code fonctionnel
 - III-B - Le(s) interface(s)
 - III-C - Création d'une archive java pour les interfaces de définition de plugins
- IV - Programmation d'un plugins
- V - Remerciements
- VI - Bibliographie et téléchargement

I - Introduction

Dans les applications d'aujourd'hui nous voyons souvent la possibilité d'ajouter des fonctionnalités par l'intermédiaire de plugins.

Ces fameux plugins sont en effet une façon très simple, tant au niveau de la programmation que du déploiement, d'ajouter des fonctionnalités à une application tournant déjà et cela sans recompilation et/ou redéploiement de l'application cible

Nous verrons dans cet article comment ajouter à vos applications cette possibilité à la fois au niveau de la conception de l'application que au niveau de la programmation.

Enfin nous verrons rapidement les aspects "sécurité" de la gestion des plugins.



*Tout au long de cet article je m'enforcerais de respecter la convention suivante : les paquetages sont écrit en italique, les classes en **gras,italique** et les méthodes en **gras***

II - Conception du système

II-A - Problématique

Avant de se lancer dans la programmation proprement dite, arrêtons-nous un moment et réfléchissons un peu à comment notre système de plugins va fonctionner.

En effet, il faut commencer par déterminer quels sont les services que nous allons offrir aux plugins, quels seront les droits sur les données que les plugins auront, comment les plugins seront intégrés à notre application, etc ...

Une fois ces questions résolues, il faut se poser la question de la sécurité. Pourquoi ? Tout simplement par ce que vous ne contrôlerez pas le code des plugins et que par conséquent, ils sont un danger potentiel pour votre application et ses données. C'est pourquoi il faut définir une politique stricte d'accès aux données et s'y tenir. Par exemple vous pouvez choisir de passer les données par copie plutôt que par référence afin de garantir que tout changement effectué par le plugins n'ai pas de répercussion sur le reste de l'application.

Mais dès lors si la modification des données par le plugins est systématiquement annulé à quoi sert-il ?

C'est le genre de questions types auxquelles il vous faudra répondre avant de commencer à penser à coder.

II-B - Les différentes classes et interface(s)

Afin de mettre en oeuvre notre système de plugins nous allons devoir coder un certain nombre de classe et d'interface(s).

Dans cet exemple nous allons donc créer :

- Une classe qui va s'occuper de charger les .jar en mémoire, de déterminer quelle(s) interface(s) de notre système sont implémentées et de créer les objets correspondants.
- Au moins une interface qui va permettre de définir les actions que les plugins devront implémenter.

Et c'est tout au niveau même du système de plugins.

III - Programmation du système

Dans cette partie nous allons voir le code qui permet de charger les plugins en mémoire et d'en obtenir des objets.

Nous verrons ensuite une interface de base pour plugins, mais gardez à l'esprit que ce n'est qu'un exemple qui est loin d'être complet.

III-A - Le code fonctionnel

Attaquons-nous maintenant au code proprement dit de notre système de plugins.

Voyons d'abord le code puis les explications liées à ce code.

```

/**
 * Fonction de chargement de tout les plugins de type StringPlugins
 * @return Une collection de StringPlugins contenant les instances des plugins
 * @throws Exception si file = null ou file.length = 0
 */
public StringPlugins[] loadAllStringPlugins() throws Exception {

    this.initializeLoader();

    StringPlugins[] tmpPlugins = new StringPlugins[this.classStringPlugins.size()];

    for(int index = 0 ; index < tmpPlugins.length; index ++ ){

        //On créer une nouvelle instance de l'objet contenu dans la liste grâce à newInstance()
        //et on le cast en StringPlugins. Vu que la classe implémente StringPlugins, le cast est toujours correct
        tmpPlugins[index] = (StringPlugins)((Class)this.classStringPlugins.get(index)).newInstance() ;

    }

    return tmpPlugins;
}

private void initializeLoader() throws Exception{
    //On vérifie que la liste des plugins à charger à été initialisé
    if(this.files == null || this.files.length == 0 ){
        throw new Exception("Pas de fichier spécifié");
    }

    //Pour éviter le double chargement des plugins
    if(this.classIntPlugins.size() != 0 || this.classStringPlugins.size() != 0 ){
        return ;
    }

    File[] f = new File[this.files.length];
    // Pour charger le .jar en memoire
    URLClassLoader loader;
    //Pour la comparaison de chaines
    String tmp = "";
    //Pour le contenu de l'archive jar
    Enumeration enumeration;
    //Pour déterminer quels sont les interfaces implémentées
    Class tmpClass = null;

    for(int index = 0 ; index < f.length ; index ++ ){
        f[index] = new File(files[index]);
        if( !f[index].exists() ) {
            break;
        }
    }

    URL u = f[index].toURL();

```

```

//On créer un nouveau URLClassLoader pour charger le jar qui se trouve ne dehors du CLASSPATH
loader = new URLClassLoader(new URL[] {u});

//On charge le jar en mémoire
JarFile jar = new JarFile(f[index].getAbsolutePath());

//On récupère le contenu du jar
enumeration = jar.entries();

while(enumeration.hasMoreElements()){

    tmp = enumeration.nextElement().toString();

    //On vérifie que le fichier courant est un .class (et pas un fichier d'informations du jar )
    if(tmp.length() > 6 && tmp.substring(tmp.length()-6).compareTo(".class") == 0) {

        tmp = tmp.substring(0,tmp.length()-6);
        tmp = tmp.replaceAll("/", ".");

        tmpClass = Class.forName(tmp ,true,loader);

        for(int i = 0 ; i < tmpClass.getInterfaces().length; i ++ ){

            //Une classe ne doit pas appartenir à deux catégories de plugins différents.
            //Si tel est le cas on ne la place que dans la catégorie de la première interface correct
            // trouvée
            if(tmpClass.getInterfaces()[i].getName().toString().equals("tutoPlugins.plugins.StringPlugins") ) {
                this.classStringPlugins.add(tmpClass);
            }
            else {
                if( tmpClass.getInterfaces()[i].getName().toString().equals("tutoPlugins.plugins.IntPlugins") ) {
                    this.classIntPlugins.add(tmpClass);
                }
            }
        }
    }
}
}
}
}

```



Ce code est celui de l'application de l'exemple, vous le retrouverez donc au complet dans les téléchargements à la fin de l'article

Voyons les lignes intéressantes ensemble :

- La création de l'**URLClassLoader** :

```

URL u = f[index].toURL();
//On créer un nouveau URLClassLoader pour charger le jar qui se trouve ne dehors du CLASSPATH
loader = new URLClassLoader(new URL[] {u});

```

Pourquoi créer un **URLClassLoader** ? Tout simplement pour que le loader de java puisse localiser les classes contenues dans le jar. En effet le jar ne figurant (normalement) pas dans le CLASSPATH, il faut créer un nouveau ClassLoader qui possède le chemin du jar dans son CLASSPATH. La méthode la plus simple consiste donc à créer un nouveau loader par le biais de **URLClassLoader**.



Je vous conseil d'aller lire la FAQ Java de [développez.com](#), la question y est traitée :

FAQ Java

- Le chargement de la classe dans le système :

```
tmp = tmp.substring(0,tmp.length()-6);
tmp = tmp.replaceAll("/", ".");

tmpClass = Class.forName(tmp ,true,loader);
```

Vous remarquerez ici que nous déterminons le nom de la classe grâce au convention de nommage java. En effet nous partons du chemin au niveau du jar et nous remplaçons les '/' par des '.' ce qui est réellement la définition des paquetages en java.

De plus le nom du fichier est celui utilisé pour le nom de la classe, ce qui est là aussi un impératif de java (pour les public au moins).

- La création d'un objet à partir d'une instance de la classes **Class** :

```
//On créer une nouvelle instance de l'objet contenu dans la liste grâce à newInstance()
//et on le cast en StringPlugins. Vu que la classe implémente StringPlugins, le cast est toujours correct
tmpPlugins[index] = (StringPlugins)((Class)this.classStringPlugins.get(index)).newInstance() ;
```

Ce code permet de créer une instance d'un objet à partir d'un objet de type **Class** qui contient la définition de la classe que l'on veut instancier.

En effet la méthode **newInstance()** de **Class** appelle un constructeur sans argument afin de créer une instance de la classe. Cela nous donne donc une contrainte en plus : avoir un constructeur sans argument. Cette restriction est contournable en récupérant la collection de constructeur de la classe grâce à la méthode **public Constructor[] getDeclaredConstructors()**. Une fois cette collection de constructeurs récupérée vous pouvez appeler la méthode **newInstance(Object[] params)** de la classe **Constructor** afin de créer une instance de cette classe.

Voilà nous en avons fini avec le code fonctionnel du système de plugins.

Comme vous avez l'avez vu, la gestion basique des plugins n'est pas très compliquée.

III-B - Le(s) interface(s)

Dans cette partie nous allons rapidement voir les interfaces utilisées dans le projet exemple.

```
package tutoPlugins.plugins;

/**
 * Interface de base pour les plugins de notre application.
 * @author Lainé Vincent (dev01, http://vincentlaine.developpez.com/)
 *
 * Cette interface n'est destinée à être directement implémenté dans un plugins,
 * elle sert à définir un comportement commun à toutes les interfaces de plugins.
 */
```

```
public interface PluginsBase {  
  
    /**  
     * Obtient le libellé à afficher dans les menu ou autre pour le plugins  
     * @return Le libellé sous forme de String. Ce libellé doit être clair et compréhensible facilement  
     */  
    public String getLibelle();  
  
    /**  
     * Obtient la catégorie du plugins. Cette catégorie est celle dans laquelle le menu du plugins sera ajouté une fois chargé  
     * @return  
     */  
    public int getCategorie();  
  
}
```

Cette interface est l'interface de base pour tous les plugins de l'application. Elle fournit les services d'enregistrement du plugins auprès du menu principal de l'application grâce à la récupération du libellé et de la catégorie.

```
package tutoPlugins.plugins;  
  
/**  
 * Interface définissant les méthodes de manipulation de String ajoutés par le plugins qui l'implémente.  
 * @author Lainé Vincent (dev01, http://vincentlaine.developpez.com/)  
 *  
 * Cette interface est strictement destinée à l'utilisation par des plugins et en aucun cas par des classes internes à notre application  
 *  
 */  
public interface StringPlugins extends PluginsBase {  
  
    /**  
     * Fonction de traitement principale du plugins de manipulation de String  
     * @param ini La chaine initiale  
     * @return La chaine traitée  
     */  
    public String actionOnString(String ini);  
  
}
```

Cette interface comme **IntPlugins** permet seulement de faire des tests. Il n'y a rien de particulier à remarquer si ce n'est l'extends **PluginsBase**

Pour conclure sur les interfaces, je rappellerais seulement que ce ne sont là que des exemples **simplistes** qui n'ont rien à voir avec de vraies interfaces de vrais plugins.

III-C - Création d'une archive java pour les interfaces de définition de plugins

Pourquoi créer une archive java contenant les interfaces des plugins ? Tout simplement pour la pratique.

En effet, lors du développement d'un plugins, il est plus pratique de travailler avec une archive réduite qu'avec toute l'application. Dès lors la question de "Que dois-je mettre dans le jar ? " peut devenir un vrai casse-tête à cause des dépendances. Une chose quand même qui est valable tout le temps : il faut séparer l'interface du corps du programme.

Voyons rapidement comment créer un jar avec nos interfaces du programme d'exemple :

```
:>cd ./bin/  
> jar cvf tutoPlugins.plugins.jar ./tutoPlugins/plugins/*.class
```

Cette ligne de commande vous donne un jar contenant la totalité du dossier plugins, c'est à dire les interfaces et la classe de chargement

IV - Programmation d'un plugins

Dans cette partie nous verrons rapidement un exemple de programmation de plugins à partir des interfaces définies plus haut.

```
package stringPluginsExemple;

import tutoPlugins.plugins.StringPlugins;

public class classTest implements StringPlugins {

    public String actionOnString(String arg0) {

        return "** - " + arg0 + " - **";

    }

    public String getLibelle() {

        return "Ajouter de jolis caractères à la String";

    }

    public int getCategorie() {

        return 0;

    }

}
```

Comme vous le voyez, il n'y a rien de plus simple que la création d'un plugins grâce à nos interfaces.

V - Remerciements

VI - Bibliographie et téléchargement

[L'article au format pdf \(miroir http \)](#)

[Les sources du projet eclipse](#)

[Les sources du plugins exemple](#)