Conception de Logiciels Extensibles

Manuel développement

Brian GOHIER, Yannis GREGO, Kevin MOKILI, Pierre NOYON, Aida TAPSOBA
28 avril 2013

Table des matières

1 Installation et lancement de la plate-forme

Pour installer la plate-forme il faut extraire tous les dossiers de l'archive dans le *workspace* d'eclipse. Le projet principal est "**ProjectCLE**" de l'archive. Vous pouvez l'importer via l'onglet *importer* d'eclipse (puis *from existing source*).

Avant de lancer la plate-forme il faut mettre à jour le "**pluginPath**" du fichier du "**config.ini**" du coeur (*ICore*). Par défaut le chemin pris en compte sera celui qui est présent dans le dossier du projet principal si le chemin entré est incorrect.
pluginPath : chemin permettant d'accéder au répertoire "**plugins**" (regroupe les plugins disponibles sur la plate-forme).
Exemple : "**pluginPath = /home/rep1/rep2/plugins**"

La plate-forme se lance ensuite en lançant l'exécutable du coeur (effectuer un *run* de la classe "**Core**").

2 Fichier de configuration

Le fichier contient les propriétés suivantes, qui fourniront les informations nécessaires pour le chargement au niveau de la plate-forme. Par défaut quand vous développez vous pouvez placer ce fichier à la racine du projet, mais lorsque vous importerez le projet il devra être placé obligatoirement à la racine du dossier du plugin (voir création de plugin 3.1.2). Ce fichier doit impérativement porter le nom "**config.ini**".

className = Nom de la classe principale du plugin
interfaces** = Noms des interfaces dont va dépendre le plugin (voir avec "**dependencies**")
active = booléen (**false** ou **true**), Si le plugin a accès à la plateforme
singleton = booléen, Si le plugin ne peut être instancié qu'une fois
default = booléen, Si le plugin se lance au démarrage de la plateforme
dependencies** = Noms des interfaces dont vont avoir besoin les plugins qui seront des extensions de ce plugin
libraries** = Noms des librairies externes (*.jar*) qu'utilise le plugin

****(Les valeurs sont séparées par une virgule)

2.1 className

Cette propriété correspond au nom du fichier java qui définit le plugin où l'on peut y trouver la définition de cette classe. Il s'agit de la classe principale qui va lancer le plugin concerné. Dans la classe principale du plugin concerné, il y a le constructeur de classe, les attributs de la classe et les méthodes qu'il doit surcharger dû aux interfaces qu'il implémente. Cette classe doit impérativement implémenter l'interface *IPlugin* présente dans la plateforme.
Exemple : "**className=NomPackage.NomSubpackage.NomClasse**"

2.2 interfaces

C'est une contrainte dont chaque plugin peut faire l'objet pour les possibles communications (plugin-plugin). Le coeur doit connaitre toutes les interfaces implémentées par chaque plugin. Dans ces classes se trouvent les primitives des méthodes nécessaires aux fonctionnalités du plugin.

2.3 active

Ce paramètre détermine l'état du plugin par rapport à la plate-forme. Si vrai (**true**) le plugin devra contenir un constructeur prenant en paramètre un objet de type *ICore*, qui est l'interface implémentée par le coeur. Dans ce cas il aura accès aux ressources du coeurs, et pourra utiliser les méthodes de cette interface. Si faux (**false**), plugin classique.

2.4 singleton

Si le plugin est un singleton ("**singleton=true**"), tant que son instance n'est pas détruite, on ne pourra pas le recharger. Si par contre le paramètre est défini à **false** on pourra le charger autant de fois qu'on le souhaite.

2.5 default

Indique à la plate-forme si le plugin doit être chargé initialement, au lancement du coeur. Si ce n'est pas le cas il faudra le charger manuellement.

2.6 dependencies

Toutes les interfaces d'autres plugins dont dépend le plugin. Pour assigner plusieurs interfaces depuis un seul plugin, regroupez toutes les interfaces dont vont dépendre les plugins dans un package et indiquez ce nom de package avec une étoile (« * ») (ex : Si ce plugin veut que ses extensions dépendent de 2 interfaces, *Interface1* et *Interface2*, on peut les regrouper dans le package "**myPlugin.interfaces.dependecies**" par exemple et indiquer le chemin :
"**myPlugin.interfaces.dependecies.***".
Dans ce cas les sous plugins devront avoir en attribut "**interfaces**" ce même chemin).

2.7 librairies

Toutes les librairies externes dont dépend le plugin. Pour les ajouter dans la plateforme il faut les placer dans le dossier "**libs**" présent dans le dossier des plugins ("**plugins**").

Tous les éléments présentés ci-dessus sont essentiels au chargement du plugin par la plate-forme. Si une valeur doit être nulle, laissez cette valeur vide (ex : Si le plugin n'utilise aucune librairie externe notez comme ceci : "**liraries=**").

3 Création d'un plugin

Un plugin est un projet Java (un répertoire "**bin**" pour les fichiers binaires (ex : toto.class) et "**lib**" pour les éventuelles librairies dont le projet a besoin. Pour ce projet, il faut rajouter un fichier de configuration (décrit au chapitre précedent). Le principe est le suivant, vous développez votre plugin puis vous l'étendez à notre plateforme en respectant un certain nombre de contraintes (liées à celle-ci) pour que la plateforme soit en mesure de faire l'extension (le chargement).

3.1 Créer un plugin simple

Ici nous vous expliquons comment créer un plugin tout à fait basique. C'est à dire un plugin qui ne dépend d'aucun autre donc totalement indépendant. Dans notre projet par exemple, existe le plugin "**PluginAffichage**" qui est un plugin tout simple qui ne fait qu'afficher un message dans la console.

3.1.1 Développement

- Créez un projet eclipse.
- Dans le build path de ce projet ajoutez le chemin vers la librairie "**core_lib.jar**".
- Créez votre classe principale, elle doit implémenter *IPlugin*.
- Donnez un nom à votre plugin (une méthode "**setName**" existe dans l'interface *IPlugin* dédiée à cet usage).
- Créez un fichier de configuration propre au plugin (Voir détails du fichier de configuration).
- Si vous voulez que votre plugin soit actif (puisse avoir accès à la plateforme), modifiez cet attribut dans le fichier de configuration et créez un constructeur qui prend en compte un objet de type *ICore*.
- Développez votre plugin.
- Compilez votre projet.

3.1.2 Importation

- Copiez les fichiers binaires de votre projet dans le dossier de la plateforme qui contient les plugins :
  . Copiez les dossiers et fichiers présents dans le dossier "**bin**" de votre projet (fichiers binaires "**.class**").
  . Le dossier des plugins de la plateforme est celui définit par l'attribut "**pluginPath**" du fichier de configuration de la plateforme.
  . Placez y les dossiers et fichiers copiés au préalable dans un dossier (Le mieux est d'utiliser le nom du plugin en tant que nom pour ce dossier).
  . Placez le fichier de configuration du plugin à la racine de ce dossier.
- Allez dans le fichier de configuration de la plateforme pour y ajouter une propriété qui correspond au nom du plugin et qui sera égale au nom du dossier dans lequel vous avez placé les fichiers binaires (Exemple pour un plugin nommé "**MyPlugin**" qui est placé dans le dossier "**MyPluginPath**" du dossier de la plateforme, la ligne sera : "**MyPlugin=MyPluginPath**").
- Relancez la plateforme et votre plugin sera reconnu.

3.2 Créer un plugin qui dépend d'un autre

Ici nous allons parler de plugins sous forme d'extension, c'est-à-dire de plugins qui vont dépendre d'un autre. Par exemple dans notre projet nous avons développé le plugin "**PluginEditor**", qui va pouvoir charger d'autres plugins à partir du coeur et qui seront les « formes » qui pourront être dessinées.

3.2.1 Développement

- Faites comme si vous développiez un plugin classic.
- Maintenant vous devez définir dans ce plugin les dépendances. Dans le fichier de configuration ajoutez l'interface (attribut "**interfaces**") qui correspond à l'attribut "**dependencies**" du fichier de configuration du plugin principal.
- N'oubliez pas d'importer une librairie qui contient ces interfaces dans le build path du projet si besoin est.

3.2.2 Importation

L'importation d'un plugin extensible se déroule de la même manière qu'un plugin classique.