UNIVERSITÉ DE NANTES

Master 1 ALMA

Projet de Conception de Logiciels Extensibles

Plateforme de plugins - Architecture et fonctionnement

Étudiants :
Coraline MARIE, Vincent RAVENEAU et Quentin MORICEAU

Encadrant :
Gilles ARDOUREL

30 mars 2014

Table des matières

Introduction **2**

1 Organisation **3**
1.1 Cahier des charges 3
1.2 Organisation du projet 3

2 Architecture de la plateforme **5**
2.1 Package `interfaces` 5
2.2 Package `main` 6
2.3 Package `pluginManager` 6

3 Fonctionnement de la plateforme **7**
3.1 Fonctionnement général 7
3.2 Fonctionnement spécifique : chargement d'un plugin 8
3.2.1 Chargement d'un plugin précis 8
3.2.2 Chargement d'un plugin aléatoire 8
3.3 Exemple de fonctionnement 9

Conclusion **10**
Conclu
ce qu'il manque (éditer les données)
(création par l'interface graphique)

Introduction

Une plateforme est une base de travail, à partir de laquelle il est possible d'utiliser un ensemble de logiciels. Un Gestionnaire de plugins, quant à lui, permet aux développeurs de gérer et d'installer de nouveaux plugins, en toute simplicité. Cela permet donc d'enrichir un logiciel, sans le surcharger, en lui rajoutant les fonctionnalité désirées. C'est aussi pourquoi les architectures à plugins sont aujourd'hui considérées comme étant l'avenir des logiciels.

L'actuel Master Alma de l'Université de Nantes, offre la possibilité aux étudiants d'étudier la *Conception de logiciels extensibles*. Les points abordés dans cette matière présentent plusieurs techniques d'architectures à plugins, permettant de développer des logiciels extensibles.

L'Éditeur d'Armure se trouve être un logiciel extensible réalisé comme projet de fin de semestre du module *Conception de logiciels extensibles*.

Ce rapport a été écrit dans le but de commenter le projet Éditeur d'Armure. Il présente donc les différentes étapes de notre travail, depuis l'imagination, jusqu'à la réalisation de notre projet.

Organisation

1.1 Cahier des charges

Dans le cadre du cours de *Conception de logiciels extensibles*, les étudiants du Master 1 Alma avaient pour objectif de concevoir un logiciel extensible, ainsi que quelques plugins agissant sur des données quelconques. Les fonctionnalités du logiciel ainsi que celles des plugins étaient libres de choix, cependant il fallait qu'elles permettent de créer, d'afficher et de modifier des données.

Pour répondre à ce cahier des charges, nous avons créé l'Éditeur d'Armure, un logiciel de gestion d'armures de protection/combat(comme celle d'Iron Man). Nous avons également créé quelques plugins qui permettent de créer, de modifier et d'afficher des armures de plusieurs façons différentes :

- création d'armure automatique ;
- création d'armure par fichier texte ;
- modification d'armure ;
- affichage console d'une armure ;
- affichage graphique d'une armure.

1.2 Organisation du projet

Ce projet, réalisé par trois étudiants, c'est étalé sur environ 3 mois de programmation. Pour ce faire, le travail a été divisé en plusieurs parties :

- l'imagination/la conception du projet ;
- la création de la base ArmorEditor ;
- la mise en place du pluginManager ;
- la production des premiers plugins : CreationArmure et AffichageConsole ;
- la mise en place du plugin pour l'interface graphique ;
- la création des autres plugin de manipulation de données ;
- le refactoring ;
- l'optimisation de l'application.

Pour que la programmation soit optilisée, chaque étudiant s'est spécialisé dans une partie du code :

- Quentin : plugin d'affichage graphique ;
- Coraline : plugins d'affichages et de manipulations des données ;

— Vincent : coeur de l'application (pluginManager, fichiers de configuration, ...) et corrections des différentes parties du projet.

Architecture de la plateforme

La plateforme à plugins, fournie dans le projet Eclipse `GestionnairePlugin`, s'articule autour de 3 packages, dont le contenu sera détailé dans les parties suivantes.

2.1 Package `interfaces`

Ce package contient uniquement des interfaces, utilisées par d'autres projets souhaitant utiliser les fonctionnalités de la plateforme de plugins. Ces interfaces sont les suivantes :

- `interfaces.IPlugin` : Cette interface permet à la plateforme d'identifier une classe Java comme un plugin valide. Elle est destinée à être implémentée par les plugins basiques, ne nécessitant pas le chargement d'autres plugins pour fonctionner. Une fois implémentée, elle impose des méthodes, permettant à la plateforme d'exécuter le plugin ou de récupérer son type. Elles lui permettent aussi de transmettre un objet, contenant les informations trouvées dans le fichier de configuration, donné lors de la demande de chargement du plugin.
- `interfaces.IComplexPlugin` : Cette interface, qui étend l'interface `interfaces.IPlugin` précédemment décrite, est destinée à être implémentée par les plugins complexes ayant eux-même besoin de charger d'autres plugins pour fonctionner. En plus des méthodes d'`interfaces.IPlugin`, elle impose une méthode qui permet à la classe gérant la création des plugins, de transmettre son instance aux plugins complexes créés. Ainsi, ces plugins seront en mesure de faire appel au gestionnaire de plugins pour charger tout ce dont ils ont besoin pour s'exécuter.
- `interfaces.IPluginManager` : Cette interface permet d'identifier une classe comme étant un gestionnaire de plugins. Utilisée par les plugins complexes pour demander le chargement des plugins dont ils ont besoin, elle impose pour ce faire 2 méthodes. L'une permet de demander un plugin précis, en donnant le nom de la classe Java associée et le chemin vers son fichier d'initialisation. L'autre permet de demander le chargement d'un plugin aléatoire d'un type donné.

2.2 Package `main`

Ce package est composé d'une unique classe Java, `main.PluginsPlatform`. Cette classe contient la méthode `main` à exécuter pour démarrer la plateforme, qui va créer une instance de la classe `pluginManager.PluginManager`, qui constitue le cœur de la plateforme.

2.3 Package `pluginManager`

Ce package contient la classe `pluginManager.PluginManager`, qui est la classe gérant le chargement et l'exécution de tous les plugins de la plateforme. Implémentant l'interface `interfaces.IPluginManager` précédemment décrite, son comportement repose sur le contenu du fichier d'initialisation qui lui est donné (`GestionnairePlugins/resources/init`).

Ce fichier est une suite de paires *clef/valeur*, destinées à être récupérées par une instance de `java.util.Properties`. Afin d'avoir un fonctionnement correct de la plateforme, les 3 clefs suivantes doivent avoir une valeur :

- `loadAtStart` : Contient le nom complet des plugins devant être chargés au lancement de la plateforme. Dans notre cas, il s'agit de l'éditeur d'armure.
- `homePath` : Contient le chemin commun à tous les fichiers qui devront être chargés lors de la suite de l'exécution de la plateforme. Ces fichiers regroupent aussi bien les `.class` devant être instanciés que les fichiers de configuration des différents plugins utilisés. Ce chemin peut être absolu, ou relatif au répertoire `$home` de la machine courante. Si plusieurs valeurs sont données, leur existance sera vérifiée dans l'ordre indiqué, et seule la dernière valide sera considérée.
- `binPaths` : Contient la liste des chemins vers les dossiers contenant les fichiers `.class` des plugins susceptibles d'être utilisés lors de l'exécution de la plateforme. Ces chemins sont relatifs au répertoire indiqué par la valeur de la clef `homePath`.

Le fichier contient ensuite une paire *clef/valeur* pour chaque valeur de la clef `loadAtStart`. Ces paires ont pour clef la valeur correspondante de la clef `loadAtStart`, et pour valeur le chemin relatif (depuis le répertoire indiqué par la valeur de `homePath`) vers le fichier d'initialisation associé à la classe en question.

Fonctionnement de la plateforme

Le fonctionnement de la plateforme est directement lié au fonctionnement des classes implémentant l'interface `interfaces.IPluginManager` qui sont exécutées. Ce document ayant pour but d'expliquer le fonctionnement de l'implémentation actuelle, cette partie illustrera le comportement de la classe `pluginManager.PluginManager` fournie. Il est cependant tout à fait possible de créer une classe gérant les plugins d'une façon complètement différente de celle-ci, tant qu'elle respecte l'interface nécessaire pour être reconnue comme telle.

3.1 Fonctionnement général

Lors de son instanciation, la classe `pluginManager.PluginManager` lit son fichier de configuration (*GestionnairePlugins/resources/init*) et enregistre son contenu. Ce contenu est ensuite analysé afin de récupérer les éléments suivants :

- Le chemin commun vers tous les fichiers à charger ultérieurement.
- Les dossiers dans lesquels se trouvent les fichiers `.class` contenant les classes à charger. Les URLs de ces dossiers sont ensuite récupérées et passées en argument à l'`URLClassLoader` qui chargera toutes les classes utilisées.

Ces informations sont mémorisées par le gestionnaire de plugins, afin de pouvoir les transmettre aux plugins en ayant besoin.

Une fois ces actions effectuées, le gestionnaire effectue un scan de l'arborescence des dossiers gérés par l'`URLClassLoader` créé précédemment. Ce scan permet de trouver les fichiers `.class` contenant des plugins valides. Ces plugins sont mémorisés et classés selon leur type, qui est obtenu par l'appel à leur implémentation de la méthode `public String type()` de l'interface `interfaces.IPlugin`. L'inconvénient de cette façon d'obtenir le type est qu'elle nécessite d'instancier temporairement chaque plugin, ce qui peut être problématique suivant leur implémentation.

Une fois ce scan fini, le gestionnaire de plugins affiche dans la console qui l'a exécuté la liste des plugins disponibles classés par type, afin que l'utilisateur en prenne connaissance.

Le gestionnaire charge ensuite tous les plugins demandés dans le fichier de configuration, en leur fournissant le fichier de configuration associé. Si le plugin

en question est un plugin complexe, le gestionnaire lui fourni également une référence vers son instance, afin que le plugin puisse demander le chargement des plugins dont il aura besoin.

3.2 Fonctionnement spécifique : chargement d'un plugin

3.2.1 Chargement d'un plugin précis

Le chargement d'un plugin est effectué par un appel à la méthode `public IPlugin loadPlugin(String pluginName, String initPath)` déclarée dans l'interface `IPluginManager`. Les deux arguments requis sont le nom complet de la classe à charger, et le chemin vers le dossier contenant le fichier d'initialisation associé.

Le gestionnaire commence par récupérer les informations contenues dans le fichier d'initialisation, auxquelles il ajoute les clefs *pathToInit* et *pathToHome* contenant respectivement le chemin vers le fichier d'initialisation (au cas où le plugin ait besoin d'analyser son contenu à nouveau) et le chemin vers l'arborescence commune à tous les éléments gérés par la plateforme (afin de fournir un point de départ pour accéder à un fichier).

La classe demandée est ensuite chargée à l'aide du ClassLoader du gestionnaire de plugins, puis passe à travers une série de tests, qui prend fin dès que l'un est positif :

— La classe implémente-t-elle l'interface `interfaces.IComplexPlugin` ?
— La classe implémente-t-elle l'interface `interfaces.IPlugin` ?

Si un de ces tests est positif, alors la classe est instanciée. Cette instance fait ensuite appel à sa méthode `public void receiveProperties(Properties prop)` afin de récupérer l'objet précédemment créé mémorisant le contenu de son fichier d'initialisation. De plus, si le plugin est un plugin complexe, l'instance fait également appel à se méthode `void receivePluginManager(IPluginManager pluginManager)` afin d'obtenir une référence vers le gestionnaire de plugins. L'instance est ensuite exécutée via sa méthode `public void run()` et retournée en résultat de la fonction.

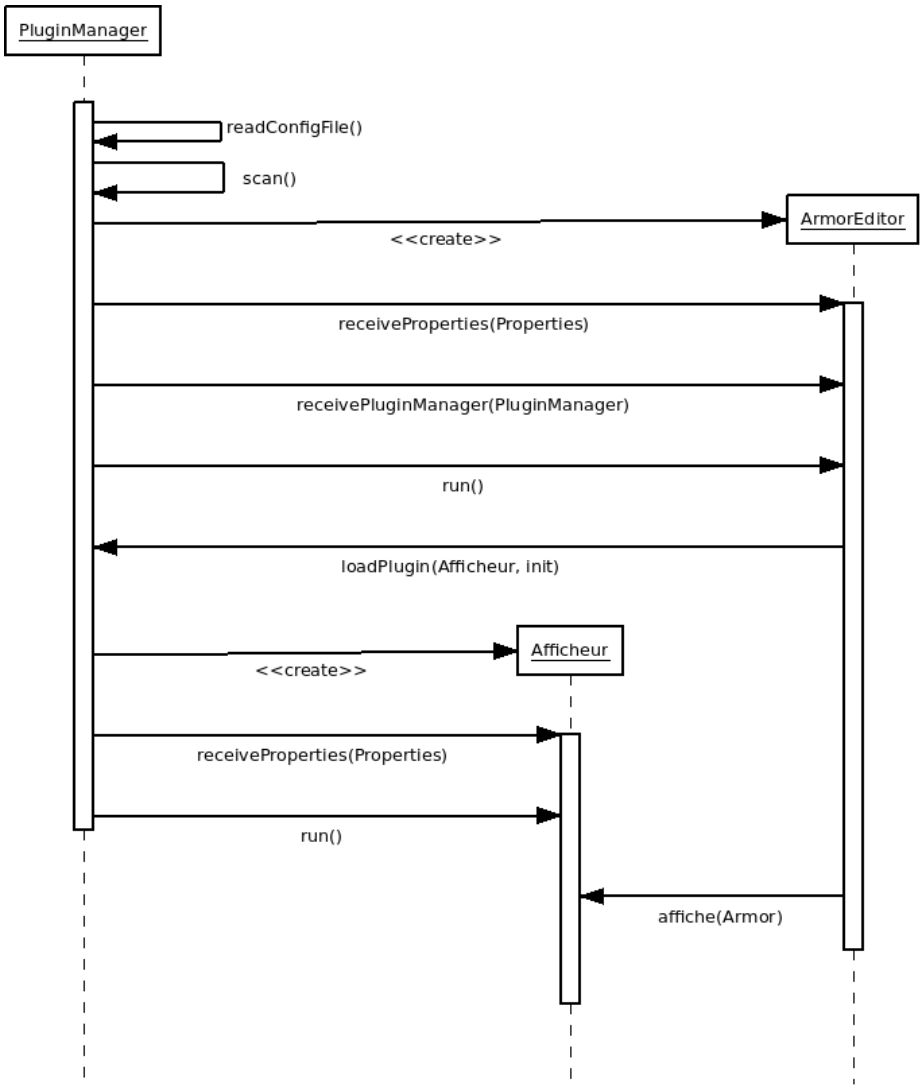
3.2.2 Chargement d'un plugin aléatoire

Il est également possible de demander le chargement d'un plugin aléatoire parmis ceux appartenant à un type précis de plugins. Cette action est affectuée par un appel à la méthode `public IPlugin loadRandomPlugin(String pluginType)` du gestionnaire de plugins, et prend en argument le type de plugin voulu.

Cette fonction récupère parmis l'ensemble des plugins trouvés lors du scan une valeur aléatoire correspondant au type voulu, et demande son chargement via la méthode décrite dans la partie ci-dessus.

3.3 Exemple de fonctionnement

Le diagramme de séquence suivant permet d'illustrer le comportement global du gestionnaire de plugins tel que paramétré actuellement :



Conclusion