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

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 étand 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 coeur 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 relatif depuis le répertoire `$home` de la machine exécutant la plateforme vers un répertoire regroupant tous les plugins susceptibles d'être demandés lors de l'exécution de la plateforme. Les sous-dossiers étant pris en compte, tout comme les liens relatifs, il n'est pas nécessaire de
  !!!!!!!!!!!!!!!!!!! A REFORMULER, erreur dans l'explication (homePath/initPath) et il manque une clef!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Conclusion