Rapport de programmation

Wollenburger Antoine, Chénais Sébastien, Ménoret Clément, Barillère Céline

Introduction

L'objectif de ce projet était l'élaboration d'une plate-forme de plugins, programme capable d'en exécuter d'autres et de gérer leurs interactions. Un tel code permet entre autres l'ajout de propriétés sur les plugins, tel le chargement paresseux. Ce rapport présentera d'abord brièvement l'architecture du programme, puis le détail des diverses évolutions.

Chapitre 1

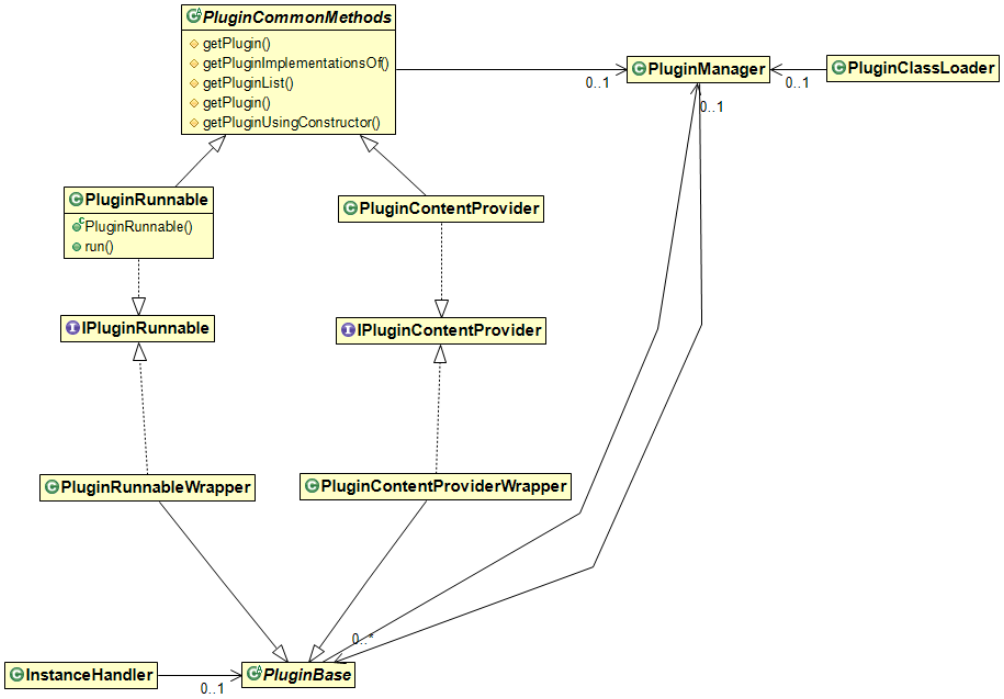
Architecture logicielle

Tout d'abord, voici, grossièrement, l'organisation de notre programme. Deux types de plugins sont disponibles :

- Les plugins exécutables : destinés à être exécutés, ils héritent de PluginRunnable, contenant une méthode run qui nécessite d'être ré implémenté.
- Les autres plugins : ils ne sont destinés qu'à fournir du contenu. Ils héritent de PluginContentProvider.

Chaque plugin est interfacé avec la plateforme via une instance enfant de PluginBase. Toutes ces instances sont connues du PluginManager.

FIGURE 1.1 – Diagramme de classe



Chapitre 2

Versions, évolutions, et choix de conceptions

2.1 Le commencement

De simples lignes destinées à exécuter une méthode quelconque d'un jar tout aussi anodin. C'est ce qu'était la première version de notre plateforme, et c'est ainsi qu'elle naquit. Après un bref test, nous optâmes pour l'utilisation d'URLClassLoader dédié au chargement du jar. Par la même occasion, nous avons adopté les jars.

2.2 L'expansion

Très rapidement, nous réécrivîmes le code pour non plus charger un jar particulier, mais tous les jars d'un même dossier (par défaut « Plugins »). Nous avons fixé par la même occasion le nom de la méthode d'exécution : elle se nommera dorénavant « run ».

2.3 La découverte des autres

S'en suivit la découverte d'une limite rédhibitoire : nos plugiciels ne pouvaient pas communiquer entre eux ! Ceci pour une raison toute bête : chaque plugin possède son propre URLClassLoader, qui lui-même a pour père celui de la plate-forme. Face à cela, notre première réaction, et non pas forcément la meilleure, fut de réécrire un ClassLoader alternatif. Ce ClassLoader, faisait une chose d'une perversion assez incroyable, nous nous en rendons maintenant compte : il redescendait l'arborescence des ClassLoaders pour rechercher la classe demandée (réécriture de findClass). Ce ne fut que plus tard qu'il fût expurgé. Ceci étant dit, nos plugins pouvaient désormais communiquer, via leurs emplacements respectifs. Pour ceci, il fallait indiquer le jar du plugin que l'on voulait utiliser comme dépendance de notre projet. Fonctionnel, mais ne permettant pas l'implémentation du chargement paresseux, et surtout n'utilisant la plateforme que dans le but de lancer la méthode run()... limitée me diriez-vous.

2.4 Vers la connaissance de l'autre

C'est bien beau de pouvoir lancer un jar, mais bon, être obligé de le faire manuellement c'est rédhibitoire. Dans cette optique, nous avons introduit un fichier properties donnant une rapide description du plugin : son nom, s'il contient une run, la classe contentant la run, doit-il être chargé paresseusement ou encore est-ce un singleton ? (les deux dernières se verraient implémentées de façons formelles bien plus tard). S'en suivit une refonte, que dis-je, une réécriture complète du code pour créer des descripteurs de plugins génériques (à savoir « PluginBase »).

2.5 Qui se ressemble s'assemble !

Face à notre monstruosité de ClassLoader descendant, nous nous décidâmes à l'éradiquer. Naquirent deux autres concepts : la notion de dépendance, indiquée par un « depend = plugin1, plugin2,... » dans le fichier properties, et celle de groupe de dépendance. Cette notion de groupe est fondamentale. Elle se résume au fait qu'un plugin se trouvera dans le même groupe que celui de son graphe connexe de dépendance. Supposons que les dépendances soient les suivantes :

Listing 2.1– Dependencies

```
A : B,C
B : C
C : D
D : null
E : F
F : null
G : null
```

Nous obtenons alors un diagramme de dépendance de cette forme (voir figure 2.1), créant par la même occasion trois groupes distincts. Chaque groupe aura son propre ClassLoader, et dans l'éventualité de la nécessité d'un plugin dont on ne dépend pas (par exemple pour un jeu chargeant des monstres à la volée, si Bob veut ajouter son monstre, il ne va pas modifier le properties de la base) on y aura accès via getPlugin.

2.6 La paresse est au rendez-vous !

Ceci résolu, nous nous attaquâmes à l'implémentation du chargement paresseux. L'InstanceHandler fit son apparition. Son utilité ? Permettre la communication de nos plugins via le plugin manager, et via des proxys. Un autre problème fit son apparition : dans cette optique nous nous sommes retrouvés à demander au concepteur du plugin de fournir une interface pour sa classe principale, proxys obligeants (ceux-ci permirent par la même occasion l'implémentation du chargement paresseux et du singleton). Peu après on vit apparaitre la notion de priorité au lancement, ce via des annotations. C'est ainsi que se termina la création de notre plate-forme de plugins.

FIGURE 2.1 – Groupes de dépendance