

MGS plugin framework manual

Antoine Forgerou David Drevet Jérémy Bardon

Sommaire

1	Recommandations et pré-requis	1
2	Installation	1
2.1	Récupérer le projet et installer la plateforme	1
2.2	Configuration de la plateforme	2
2.3	Compiler les plugin et démarrer le jeu	2
3	Les plugins	3
3.1	Introduction	3
3.2	Fichier de configuration	3
3.3	Détail sur les plugins runnable	3
3.4	Format et organisation	4
4	Créer des plugins	5
4.1	Générer le plugin principal	5
4.2	Créer le patron pour les plugins secondaires	6
4.3	Générer les plugins secondaires	6
4.4	Compilation et réglages du framwork	7
5	Appels au framework	8
6	Fonctionnement interne	9

Résumé

Le framework MGS permet de gérer de manière transparente la modularité des applications basées sur l'utilisation de plugins.

Plutôt que d'obliger les utilisateurs à remplir de long fichiers de configurations, nous avons choisi le plus possible de nous concentrer sur une architecture de fichier la moins contraignante possible.

1 Recommandations et pré-requis

L'ensemble du framework et des plugins sont tous gérés sous la forme de projets *maven* il est donc nécessaire d'avoir cet outil installé pour développer sur la plateforme.

Notez qu'il est tout à fait possible d'utiliser le framework sous forme de d'archive *jar* pour l'utiliser dans un projet n'utilisant pas *maven*. Le choix de maven permet de faciliter l'intégration des dépendances entre la plateforme, les plugins principaux et les plugins secondaires.

Pour des raisons de clareté, tout les chemins indiqués dans ce manuel ont pour racine le dossier du projet *snake* cloné depuis GitHub.

2 Installation

Cette section explique comment installer la plateforme localement et y ajouter les plugins développés par l'équipe pour l'exemple du snake. La plupart des commandes ne seront pas expliquées ici mais l'installation des plugins est détaillée dans la section suivante.

2.1 Récupérer le projet et installer la plateforme

Le projet est hébergé sur GitHub, pour le récupérer il suffit de le cloner avec la commande suivante :

```
$ git clone https://github.com/masters-info-nantes/snake.git
```

Listing 1 – Télécharger le projet

Pour installer la plateforme localement, il faut utiliser la commande « *mvn install* » dans la répertoire de la plateforme à savoir */platform*.

Ceci va avoir pour effet de rendre disponible la plateforme pour les plugins que nous allons compiler dans l'étape suivante.

2.2 Configuration de la plateforme

Il existe un seul fichier de configuration qui permet d'ajuster le fonctionnement du framework. Ce fichier se nomme *settings.txt* et se trouve dans le répertoire */platform/resources*.

<code>pluginspath</code>	Chemin absolu vers le dossier qui contient tout les plugins.
<code>startplugin</code>	Nom du plugin ¹ à charger au démarrage. Obligatoirement runnable.

TABLE 1 – Paramètres du fichier settings.txt

Ce répertoire contient également un dossier *plugins* qui rassemble les fichiers *jar* des plugins développés par l'équipe. La valeur par défaut de *pluginspath* est donc */platform/resources/plugins*.

Il est nécessaire de changer la valeur de *pluginspath* afin qu'elle corresponde à l'endroit où se trouve le dossier */platform/resources/plugins* localement.

2.3 Compiler les plugin et démarrer le jeu

Les plugins se trouvent dans le répertoire */plugins*. Avant de tous les compiler et les importer dans la plateforme, il est nécessaire d'installer les plugins principaux *platforminspector* et *snakecore* pour pouvoir compiler leurs plugins secondaires par la suite.

```
$ cd /plugins/snakecore/  
$ mvn install  
  
$ .cd /plugins/platforminspector/  
$ mvn install
```

Listing 2 – Installation des plugins principaux

Nous pouvons maintenant compiler et importer dans la plateforme tout les plugins. Pour cela, il suffit de lancer le script *importPluginsOnPlatform* mis à disposition dans le dossier */plugins*.

```
$ cd /plugins/  
$ ./importPluginsOnPlatform.sh
```

Listing 3 – Importer les plugins

Tout les plugins sont maintenant importés dans la plateforme. Pour la démarrer, il faut importer la plateforme dans Eclipse et lancer la classe principale *App.java*

```
$ cd /platform/  
$ mvn eclipse:eclipse
```

Listing 4 – Démarrer la plateforme

3 Les plugins

3.1 Introduction

Le dossier plugin renseigné par *pluginpath* est le répertoire qui contient tout les plugins que le framework pourra charger. On distingue deux types de plugins :

Runnable Plugin principal qui peut être démarré par le framework s’il est renseigné comme *startplugin* dans le fichier de configuration du framework.

Classic Plugin annexe qui fournit des classes respectant les interfaces définies par un ou plusieurs plugin(s) runnable qui l’utiliseront.

3.2 Fichier de configuration

Chaque plugin – quelque soit son type – doit respecter une certaine architecture en termes d’organisation de son dossier. En effet, un plugin doit obligatoirement fournir un fichier *plugin.txt* qui le décrit et donne des informations sur comment il peut être utilisé.

Le dépôt git du projet² regroupe les plugins développés par l’équipe (répertoire plugins).

3.3 Détail sur les plugins runnable

Les plugins principaux sont capables de définir des interfaces afin de s’assurer que les plugins qu’ils utiliseront répondent à leurs besoins.

Plutôt que de donner la liste des interfaces dans le fichier de configuration – ce qui est lourd – nous avons choisi d’obliger l’utilisateur à placer ces interfaces dans un sous-package *interfaces*. Ceci est moins contraignant en plus du fait

2. Projet hébergé par GitHub : <https://github.com/masters-info-nantes/snake>

runnable	Indique si le plugin est un plugin runnable ou classic. Peut être égal à <i>0/1</i> ou <i>true/false</i> .
category	Classe ou interface que respecte le plugin. Pour un plugin runnable ce sera toujours <i>fr.univnantes.snake.framework.MGSApplication</i> qui sera rempli automatiquement par le framework. Dans le cas d'un plugin classique ce sera une interface définie par un plugin runnable.
mainClass	Classe principale qui sera chargée par le framework. Elle doit implémenter ou hériter de celle donnée par la catégorie.

TABLE 2 – Paramètres d'un fichier plugin.txt

que des utilisateurs organisées les auraient de toutes façons rassemblés dans un package à part.

3.4 Format et organisation

Tout plugin est fourni à la plateforme sous forme d'une archive *jar*. Quelle soit générée par un projet maven – avec l'outil de création de plugins – ou autres elle doit toujours respecter une certaine organisation interne.

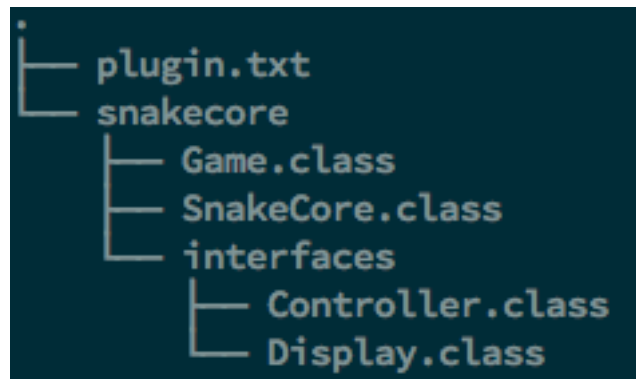


FIGURE 1 – Contenu d'une archive d'un plugin

Dans l'exemple ci-dessus, on remarque que le fichier de configuration *plugin.txt* se trouve à la racine de l'archive. La classe principale de ce plugin est *SnakeCore* dont le package est *snakecore* ce qui se traduit par le fait que

la classe se trouve dans le sous-répertoire *snakecore*.

Il s'agit d'un plugin runnable qui définit des interfaces – *Controller* et *Display* – et qui doit donc obligatoirement les placer dans le sous-package *interfaces* se trouvant au niveau de la classe principale du plugin.

4 Créer des plugins

Le but de cette partie est de créer un plugin principal qui peut dire bonjour dans plusieurs langues. Le plugin principal *Hello* sera chargé de dire bonjour et il va s'appuyer sur les plugins secondaires *Francais* et *Anglais* pour le dire dans ces deux langues.

4.1 Générer le plugin principal

Pour commencer, il est nécessaire de vérifier où le framework va chercher les plugins en regardant le fichier *settings.txt*.

L'équipe a mis à disposition le script *createPlugin* dans le répertoire */plugins* et nous allons l'utiliser pour générer un projet maven pour notre plugin.

```
$ ./createPlugin.sh
This script will create a plugin skeleton for MGS framework

Your plugin name (will be folder name):
hello

Is it a runnable plugin ? (y or n)
y

Full qualified name of your plugin class (like com.plugin.
  MyPlugin):
com.hello.Hello

Plugin description:
Plugin principal qui dit bonjour

Generating plugin skeleton....
Plugin hello has been created in current directory
```

Listing 5 – Création du plugin hello

4.2 Créer le patron pour les plugins secondaires

Comme expliqué précédemment (voir 3.3), il est possible de définir des plugins secondaires qui seront utilisés par le plugin principal. Tout d’abord, il faut commencer par définir le périmètre fonctionnel des plugins secondaires que l’on veut à travers une interface.

Pour cela il faut créer le package *interfaces* au niveau de la classe principale du plugin et à l’intérieur définir notre interface *Speak* qui permet de dire bonjour.

```
package com.hello.interfaces;  
  
public interface Speak {  
    public void sayHello();  
}
```

Listing 6 – /plugins/hello/src/main/java/com/hello/interfaces/Speak.java

4.3 Générer les plugins secondaires

Nous allons de nouveau utiliser le script *createPlugin* pour générer les projets pour les plugins.

```
$ ./createPlugin.sh  
This script will create a plugin skeleton for MGS framework  
  
Your plugin name (will be folder name):  
français  
  
Is it a runnable plugin ? (y or n)  
n  
  
Full qualified name of plugin category (like com.plugin.  
    interface.MyInterface):  
com.hello.interfaces.Speak  
  
Full qualified name of your plugin class (like com.plugin.  
    MyPlugin):  
com.français.Français  
  
Plugin description:  
Plugin secondaire pour hello , dit bonjour en français  
  
Generating plugin skeleton....  
Plugin français has been created in current directory
```

Listing 7 – Création du plugin français

Il faut maintenant définir l'action du plugin lorsqu'on lui demande de dire bonjour. Pour ce faire il faut compléter le code de la méthode *sayHello* fournie par l'interface *Speak*.

```
package com.francais;

import com.hello.interfaces.Speak;

public class Francais implements Speak{

    @Override
    public void sayHello() {
        System.out.println("Bonjour");
    }
}
```

Listing 8 – /plugins/francais/src/main/java/com/francais/Francais.java

La création du plugin *Anglais*, se fait exactement de la même manière à ceci près qu'il faut attention au nom du plugin et à celui de la classe principale.

4.4 Compilation et réglages du framework

Nous avons à cette étape nos trois plugins *hello*, *francais* et *anglais* qui sont créés et implémentés mais il faut maintenant les compiler pour ensuite les importer dans la plateforme.

```
$ cd /plugins/hello
$ mvn install

$ cd /plugins/
$ ./importPluginsOnPlatform.sh
```

Listing 9 – Compilation des plugins

Avant d'importer les plugins il est nécessaire d'installer le plugin principal afin de pouvoir satisfaire la dépendance des plugins secondaire par rapport à ce dernier.

Un fois l'importation terminée, tout les plugins seront présents – sous la forme d'archives *jar* – dans le répertoire */platform/resources/plugins/*.

La dernière chose à faire est de dire au framework de démarrer notre plugin *hello* au démarrage. Pour cela, il faut renseigner le nom du plugin – nom du fichier *.jar* – dans le paramètre *startplugin* du fichier de configuration du framework.

5 Appels au framework

Etant donné qu'un plugin principal peut utiliser des plugins secondaires gérés par le framework, il est possible d'instancier des plugins secondaires à partir d'un plugin principal.

La classe *Hello* du plugin *hello* doit implémenter la méthode *run* qui prend en paramètre un objet de type *AppContext* et c'est ce lien qui va permettre de faire des appels au framework. Cette classe possède deux attributs accessibles par le plugin (via des getters) : *currentPlugin* et *pluginLoader*. Le premier regroupe tout simplement la configuration du plugin courant – hello – mais le second donne accès au gestionnaire de plugins.

<code>String getPluginsPath()</code>
Chemin vers le dossier où se trouvent les plugins
<code>Collection<Plugin> getClassicPlugins()</code>
Configuration de tout les plugins chargés par le framework (exepté les plugins principaux).
<code>List<Plugin> getClassicPluginsByCategory(String category)</code>
Tout les plugins secondaires ayant la catégorie donnée
<code>Set<String> getClassicPluginsList()</code>
Nom de tout les plugins chargés par le framework (exepté les plugins principaux).
<code>Object loadPlugin(Plugin plugin) throws IOException</code>
Charge le plugin donné en paramètre. Cette fonction assure que le plugin donné est bien une sous-classe de sa catégorie.

TABLE 3 – Méthodes du gestionnaire de plugins

6 Fonctionnement interne

Lors d’une première phase, le framework va lire son fichier de configuration – *settings.txt* – pour trouver le chemin vers le dossier des plugins. Ensuite, il va scanner les jars, lire le fichier de configuration *plugins.txt* et stocker la liste des plugins dont la configuration est valide.

Attention ! Le framework indique au démarrage pour chaque plugin s’il a été chargé ou non.

Certaines propriétés du fichier *plugin.txt* sont déduites par le framework : *runnable* est faux par défaut et ce n’est pas obligé de donner la *category* pour un plugin principal car c’est toujours la même (voir 3.2).

Une fois cette étape de chargement de la liste des plugins, le framework va démarrer le plugin par défaut qui a été indiqué avec *startplugin* dans son fichier de configuration.

La liaison entre le framework et le plugin principal se fait à travers la relation d’héritage avec la classe *MGSApplication*.

En fait, nous avons fait ce choix car l’utilisation d’un singleton aurait pu donner accès au framework à n’importe quel plugin (même secondaire). De plus, si nous avions choisi de passer l’accès en paramètre de la fonction surchargée – *run* – ceci aurait obligé l’utilisateur à ajouter un attribut pour stocker l’accès au framework.