Rapport projet programmation avancée

Jérémy JAGUT, Samir ZENNANI, Chamir MOUSTAPHA-ABLOH et Melvin MOREAU

MASTER 1 MIAGE

UNIVERSITÉ NICE SOPHIA ANTIPOLIS

MIAGE NICE

12/01/2018

Contenu

La participation de chacun des membres de l'équipe 2
Chamir MOUSTAPHA-ABLOH 2
Samir ZENNANI 2
Jérémy JAGUT 2
Melvin MOREAU 2
La procédure à suivre pour tester votre projet 3
La procédure pour créer de nouveaux plugins 4
La procédure pour utiliser de nouveaux plugins 6
Quelques exemples de plugins : 6
Un calendrier des grandes étapes du projet 7
30 décembre 7
5 au 6 janvier 7
7 janvier 7
8 janvier 7
9 janvier 8
10 janvier 8
11 janvier 8
12 janvier 9
Fonctionnalités 9
Mécanisme de gestion de plugins et de chargement dynamique 9
Persistance 10
Modularité et dépendances 10
Gestion de projet : 11

La participation de chacun des membres de l'équipe

Chamir MOUSTAPHA-ABLOH

Chamir fut l'un des premiers à créer le dossier qui allait nous servir pour le projet. Il a aussi et surtout déployé et paramétré MAVEN. Il a donné des idées concernant les plugins que nous avons réalisés et il en a implémenté un d'attaque qui permet à un robot d'absorber la vie de son adversaire. Il a aussi réalisé un plugin permettant au robot de se déplacer en avant et en arrière sur la grille.

Il a aussi contribué à la réalisation de ce rapport. Il s'est occupé de la création et de la maintenance de la documentation.

Samir ZENNANI

Lors de ce projet, Samir s'est chargé principalement du développement de la classe Grille et de la classe Cellule. Ces classes ont été réalisées dans le but de pouvoir placer dans l'arène des robots et de pouvoir plus facilement gérer les déplacements d'un robot ou bien les distances entre eux. Le travail de Samir, c'est centré par la suite sur la création de méthode permettant de mieux gérer les différentes interactions avec cette grille. Il a surtout participé à la partie gestion de projet, fonctionnalités, modularité et persistance du rapport.

Jérémy JAGUT

Jérémy a été le premier à commencer le développement, en créant les premières classes du projet dans le but de réaliser une fenêtre. Il a ainsi créé la première version de la classe robot et la première version du template des plugins. Ces premiers plugins ont porté sur l'ajout au robot de couleur aléatoire. Après différente version de template pour les plugins et l'ajout, il a trouvé la version actuelle du template qui est la plus simple pour ajouter des plugins dynamiquement au core de l'application.

Ses connaissances lui ont permis d'adapter la classe grille réalisé par Samir Zennani afin que cette dernière soit présente dans une fenêtre pour qu'elle puisse être affiché. Dès que la grille a pu être affichée, nous avons pu avoir une première version des combats de robots.

Ses plus grosses participations ont été pour tout ce qui concerne la partie affichage de l'application mais aussi l'organisation et l'implémentation de la partie « core » de l'application.

Pour ce qui est du core de l'application, il s'est notamment occupé des classes MyClassLoader, OutilsReflection et du chargement dynamique des plugins de l'application. Tout cela dans le but de charger des plugins qui sont en dehors du classpath.

Enfin, c'est lui qui a réussi à structurer le projet en un projet parent et deux sous-projets, où l'un contient les plugins de l'application et l'autre contient le core de cette dernière. Il a aussi participé activement à la réalisation du rapport en travaillant sur les parties plus techniques du rapport.

Melvin MOREAU

Au début du projet, Melvin a réalisé une première conception de l'application qui avait pour but de faciliter le développement tout au long du projet. Dans cette conception, il avait noté qu'elle serait les plugins utiles à faire comme les attaques courtes et longues ou les déplacements aléatoires.

Mais aussi la façon dont les plugins pouvaient fonctionner entre eux sans qu'il soit dans le même projet.

Lors de la phase de développement du projet, il réalisa une première version du template permettant de créer et d'ajouter facilement un nouveau plugin à l'application. Ce template contenait la forme que devraient avoir tous les plugins d'attaques et la structure de l'annotation permettant de récupérer les plugins.

Il a aussi participé au développement des plugins d'attaques et notamment les plugins d'attaques courtes, longue et lourde. Il a dû vérifier que chaque attaque était équilibrée afin qu'aucun d'entre elle ne soit trop forte comparé aux autres. Comme cela aurait peut-être le cas pour les attaques à distance. Il fut aussi chargé des tests unitaires sur les plugins avant la mise à jour majeure et il a réalisé les tests unitaires des classes robot et App.

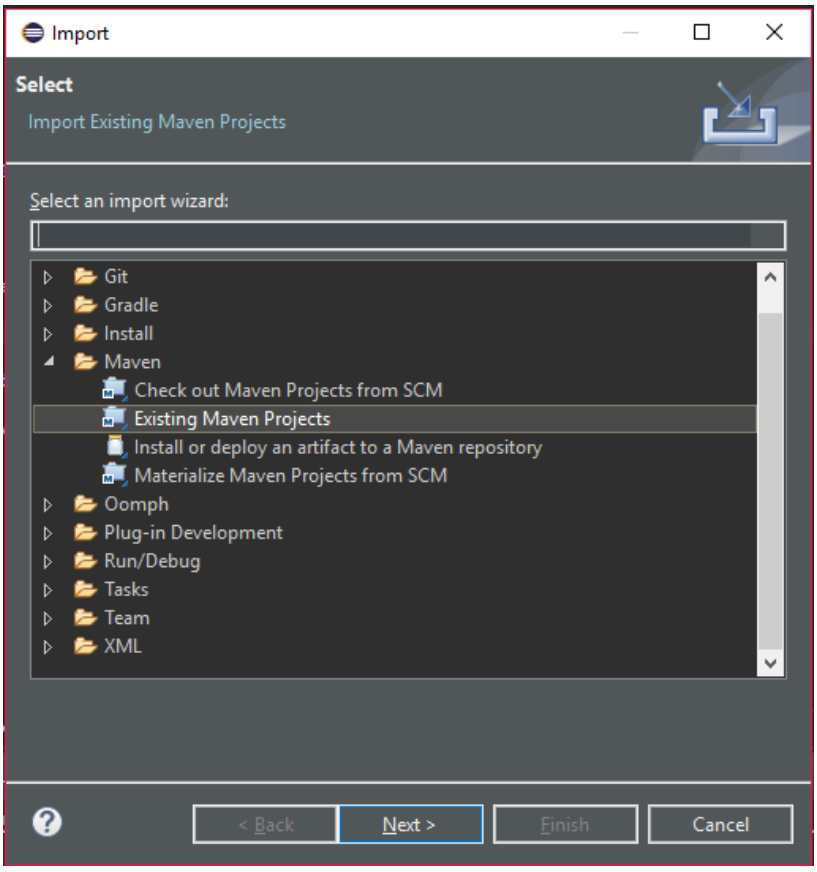
Enfin, il a été chargé de commencer et d'avancer le rapport de projet, le diaporama pour la soutenance de projet et la documentation pendant que ses collègues avançaient le code. Il a été chargé de la relecture et de la rédaction du rapport.

La procédure à suivre pour tester votre projet

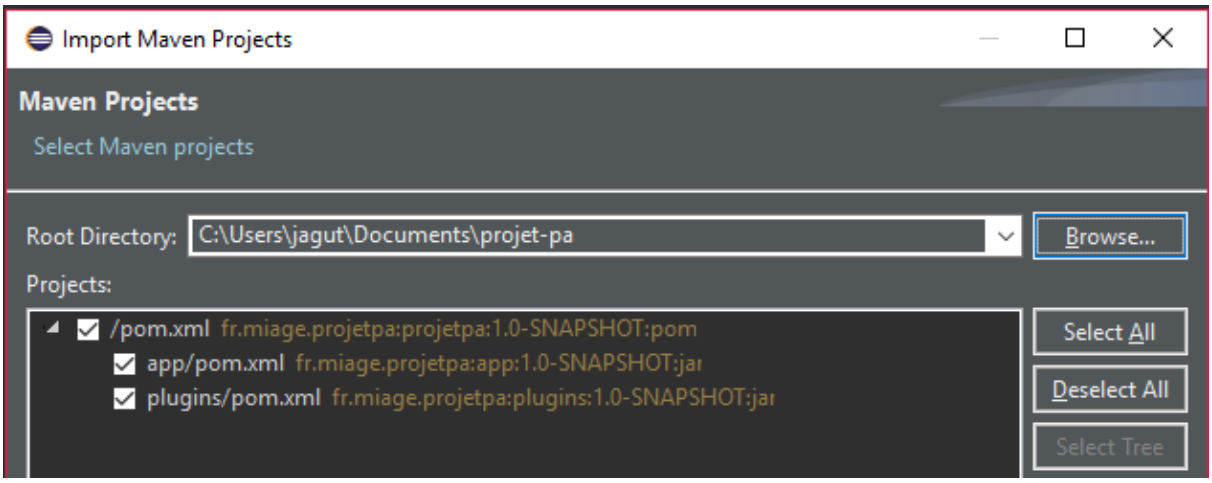
Après de nombreux efforts, nous ne sommes pas parvenus à exécuter notre projet depuis un. jar, le lancement se fera donc sous Eclipse.

Sous Eclipse :

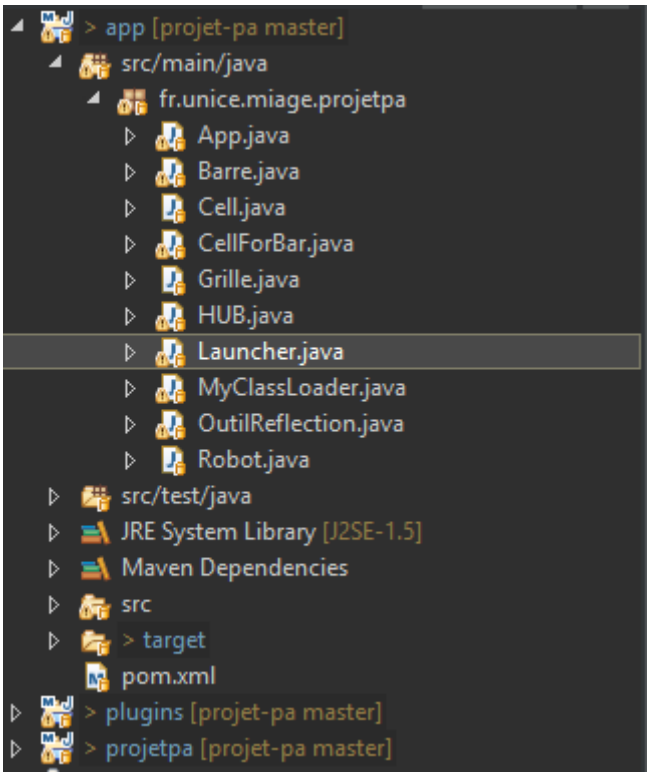
- Importer le dossier du projet



- Bien vérifier que les sous modules du projet soient cochés

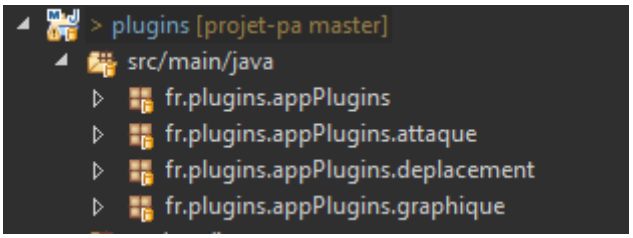


- Lancer la classe « Launcher. java » dans le module « app »



La procédure pour créer de nouveaux plugins

Créer une nouvelle classe dans le module plugin, dans le package correspondant à son type



Lui ajouter l'annotation « @Plugin (String nom, String type, Type typeOf) »

```
@Plugin(Nom = "attaqueCourte", Type = "attaque", TypeOf = Type.Attaque)
public class AttaqueCourte {
```

Si le plugin est de type « attaque » (Type = « attaque »), lui ajouter les méthodes et attributs ci-dessous (seul la valeur des attributs est modifiable) pour qu'il puisse fonctionner correctement :

```
private int degat = 10;
private int energyUse = 5;
private int distanceAtk = 1;

@PluginInfos(name = "getDegat")
public int getDegat() {
    return degat;
}

@PluginInfos(name = "getEnergyUse")
public int getEnergyUse() {
    return energyUse;
}

@PluginInfos(name = "getDistanceAtk")
public int getDistanceAtk() {
    return distanceAtk;
}
```

Si le plugin est de type « robotcolor » :

```
@PluginInfos(name = "getColor")
public Color getColor() {
    Color c = ...
    return c;
}
```

Si le plugin est de type « move » :

```
@PluginInfos(name = "nextMove")
public int nextMove() {
    return //int 1(HAUT), 2(BAS), 3(GAUCHE), 4(DROITE);
}
```

Enfin, si vous voulez ajouter un nouveau type de plugin ou de nouvelles méthodes, vous le pouvez. Mais il sera important de préciser le type dans l'annotation @Plugin de la classe, et d'ajouter @PluginInfos (name = « nom de la méthode ») au-dessus des méthodes que vous voudrez utiliser.

La procédure pour utiliser de nouveaux plugins

Tous les plugins du dossier « plugins » (les classes avec l'annotation @Plugin), seront chargés dans une HashMap (avec en clé un String du nom du plugin et en valeur son fichier) au démarrage de l'application.

Pour récupérer un plugin par son nom :

- Class cl = myCl. loadPluginFromPluginFile (pluginMap.get (nom du plugin));

Pour utiliser une de ses méthodes :

- Object obj = OutilReflection. construire (classe du plugin) ; → la classe doit être chargée comme ci-dessus.
- Object result = (Object) OutilReflection. invokeMethod (type du plugin, nom de la méthode, null) ;

Quelques exemples de plugins :

```java
package fr.plugins.appPlugins.attaque;

import fr.plugins.appPlugins.Plugin;
import fr.plugins.appPlugins.PluginInfos;
import fr.plugins.appPlugins.Plugin.Type;

 * Ce plugin permet a un robot d'effectuer une attaque lourde sur un autre robot
@Plugin(Nom = "attaqueLourde", Type = "attaque", TypeOf = Type.Attaque)
public class AttaqueLourde {

    private int degat = 25;
    private int energyUse = 50;
    private int distanceAtk = 1;

    @PluginInfos(name = "getDegat")
    public int getDegat() {
        return degat;
    }

    @PluginInfos(name = "getEnergyUse")
    public int getEnergyUse() {
        return energyUse;
    }

    @PluginInfos(name = "getDistanceAtk")
    public int getDistanceAtk() {
        return distanceAtk;
    }
}
```

Ce plugin est de type « attaque », il contient des informations sur les dégâts, l'énergie utilisée et la distance d'attaque de l'attaque.
Ces informations-là sont accessibles par l'application avec les méthodes annotées @PluginInfos retournant leur valeur.

Dans l'application, lorsqu'un robot souhaite attaquer, on charge le plugin correspondant à son type d'attaque (Robot. atkType), ici « AttaqueLourde » afin d'utiliser ces informations.

```java
package fr.plugins.appPlugins.deplacement;

import java.util.Random;

import fr.plugins.appPlugins.Plugin;
import fr.plugins.appPlugins.PluginInfos;
import fr.plugins.appPlugins.Plugin.Type;

@Plugin(Nom = "RandomMove", Type = "move", TypeOf = Type.Deplacement)
public class RandomMove {

    @PluginInfos(name = "nextMove")
    public int nextMove() {
        Random rnd = new Random();
        int value = rnd.nextInt(4) + 1;
        return value;
    }
}
```

Ce plugin est de type « move », il servira donc à faire se déplacer un robot.
Il comporte une méthode nextMove () renvoyant une valeur entre 1 et 4, indiquant une direction. Ici, la méthode renvoie une direction aléatoire.

Un calendrier des grandes étapes du projet

30 décembre

- Chamir :
  - Création du github et du projet en maven
  - Création du pom.xml

5 au 6 janvier

- Melvin :
  - Rapide conception de l'application

7 janvier

- Jérémy :
  - Création des premières classes du projet ainsi que de la première interface ;
  - Création d'un premier plugin graphique permettant de changer la couleur du robot aléatoirement.
- Melvin :
  - Création d'une version de template pour les plugins.
- Samir :
  - Création de l'architecture de la classe Grille
  - Création de la classe Cellule.

8 janvier

- Jérémy :
  - Création d'un template final pour les plugins ;
  - Création du plugin permettant le déplacement random des robots ;
  - Ajout de la grille dans la fenêtre et mise en mouvement des robots.

- Melvin :
  - Création des plugins d'attaque courte, lourde et à distance ainsi que l'équilibrage pour empêcher qu'une attaque soit trop puissante ;
  - Ajout des plugins d'attaques au core de l'application pour que les robots puissent attaquer :
  - Début de la rédaction du rapport.
- Chamir :
  - Création du plugin d'attaque qui permet d'absorber la vie d'un autre robot.
  - Création du plugin déplacement qui permet au robot d'aller en avant et en arrière
- Samir :
  - Ajout des méthodes permettant à une cellule de savoir où se trouvent les autres ;
  - Finalisation de la classe Grille avec les méthodes adjacentes.

9 janvier

- Jérémy :
  - Ajout de la possibilité de faire des combats a plus de deux robots ;
  - Ajout d'une première version d'affichage de la vie et de la barre d'énergie.
- Melvin :
  - Création des premiers tests sur les plugins et du test sur la classe Robot et App.
  - Avancement du rapport ainsi que début de la création du diaporama des soutenances.
- Samir :
  - Tests sur la classe grille

10 janvier

- Jérémy :
  - Modification de la structure du code (en 3 projets) pour permettre la gestion dynamique des plugins ainsi que pour pouvoir séparer les plugins du path.
  - Implémentation dans le code : que faire si aucun plugin de ce type n'est trouvé ?
- Melvin
  - Continuation du rapport et du diaporama de la soutenance.
- Samir :
  - Participation à l'élaboration du dossier et de la soutenance.
- Chamir :
  - Participation à l'élaboration du dossier et de la soutenance.

11 janvier

- Jérémy :
  - Participation à l'élaboration du dossier et de la soutenance.
- Melvin :
  - Finalisation du diaporama de soutenance et participation au rapport.
- Chamir :

        - Participation à l'élaboration du dossier et de la soutenance.
    - Samir :
        - Participation à l'élaboration du dossier et de la soutenance.

12 janvier

- Jérémy :
    - Rédaction de la partie technique du rapport du rapport.
- Melvin :
    - Rédaction de la partie finale du rapport du rapport.
- Chamir :
    - Finalisation du rapport.
- Samir :
    - Rédaction de la partie gestion de projet du rapport, modularité et persistance, et fonctionnalité.

Fonctionnalités

Notre application propose plusieurs fonctionnalités que nous allons détailler ici. Tout d'abord, notre jeu présentera une fenêtre qui est composée d'une grille, elle-même composée de cellule (arène de combat) ainsi que la présence d'un HUD pour chacun de nos robots (HUD = infos sur le robot).

Concernant les robots, chacun des robots a la possibilité d'effectuer une attaque. Nous avons défini plusieurs types d'attaques. En premier lieu, l'attaque courte qui provoque dix points de dégâts. Cette attaque utilise cinq points d'énergie et la distance nécessaire à l'attaque est d'une case. L'attaque lourde, quant à elle cause 25 points de dégâts et utilise 50 points d'énergie pour une même distance d'attaque d'une case. Enfin, l'attaque à distance cause deux points de dégâts en utilisant 25 points d'énergie mais peut être lancée à une distance d'au moins deux tiers de l'arène.

Les robots peuvent également se déplacer dans l'arène, en suivant le principe des cellules de la grille. Le déplacement se fait cellule par cellule. Ces mêmes robots sont colorés : en effet, la couleur propre d'un robot change de façon aléatoire avant chaque partie. Notre conception nous permet de redéfinir la taille de l'arène et le nombre de robots sera modulable et s'adapte à la taille de la fenêtre (combien de cellule dans notre grille).

Enfin, nous avons procédé à l'ajout dynamique de nos plugins dans l'application.

Mécanisme de gestion de plugins et de chargement dynamique

Dans notre projet, tous les plugins sont gérés grâce à des annotations. Nous avons créé une classe MyClassLoader qui nous permet d'ajouter les plugins au core de l'application même si ses derniers ne sont pas dans le même projet, et de les charger. Elle contient deux méthodes:

- «findAllPlugin (File file, HashMap < String, File > map) » nous permet de parcourir récursivement un répertoire et d'ajouter à une HashMap les plugins trouvés (classe comportant l'annotation @Plugin).

- loadPluginFromPluginFile (File pluginFile) qui nous permet de charger une Class depuis un fichier File.

Une classe OutilReflection qui nous permet de construire un plugin et d'utiliser ses méthodes. Elle contient deux méthodes :

- La méthode « construire » qui nous permet de retourner une instance d'une classe donnée en paramètre.
- La méthode « invokeMethod » qui nous permet d'utiliser une méthode d'un plugin que nous venons de construire.

Dans notre application pour récupérer des informations sur les plugins nous utilisons deux annotations :

- Le premier est l'annotation « @plugin (Nom = nomPlugin, Type = type de plugin) » cette annotation est a placé avant la avant un plugin.
  Cette annotation prend 3 paramètres, il prend le nom du plugin, son type, et une enum de Plugin. Type (graphique, attaque ou mouvement). Le deuxième paramètre est le plus important, il nous sert lorsque nous voulons récupérer les plugins des différents types. Cela nous permet d'avoir le type et le nom du plugin que nous allons utiliser par la suite.
- La seconde annotation est « @PluginInfos (name = nomMéthode) ». Cette annotation se place juste avant une méthode d'un plugin et permet d'identifier le nom d'une méthode afin de l'utiliser.

Persistance

Nous n'avons pas de persistance sur l'état des plugins, si un plugin possède des attributs, ils doivent être instanciés directement dans la classe. Si on supprime un plugin utilisé, par exemple « RobotColor », les robots n'auront pas de couleurs (tous noirs). Si on supprime un plugin de déplacement ou d'attaque mais qu'il y en a d'autres, les robots vont choisir le premier qu'ils trouvent. Si aucun plugin de déplacement ou d'attaque, respectivement les robots ne se déplacent pas ou n'attaquent pas.

Modularité et dépendances

Nous avons convenu pour structurer notre projet de le découper en trois parties. La partie principale, que l'on nomme le projet « parent » est lui-même composé de deux sous modules : les modules « App » et « plugins ».
Le module « app » contient le cœur de l'application. Quant à lui, le module plugins comporte tous les plugins nécessaires à l'élaboration du jeu.

Au niveau des dépendances, nous précisons que le module « app » dépend du module « plugins » et il ne pourra donc utiliser aucuns plugins sans ce dit module. Plus globalement, le module principe dépend de ses deux sous modules. Nous avons décidé de découper le projet de cette façon dans le but de faire fonctionner un. jar exécutable, qui peine à être réalisé pour souci technique.

Enfin, il faut savoir que sans ce découpage en trois modules, l'application est aussi capable de charger les plugins d'un dossier quel qu'il soit et où qu'il soit.

Gestion de projet :

Nous avons commencé par analyser le sujet de notre côté pour voir ce que nous comprenions et nous nous sommes réunis par la suite pour avoir la vision de chacun concernant l'application.

Une fois que pour nous tout le sujet était clair et que nous étions sur la même longueur d'onde, nous avons commencé à nous demander ce dont nous avons besoin. Une rapide remise à niveaux concernant les principes à connaître a été faite à l'aide des TP vues en cours. Il a été réalisé une rapide conception dans le but de savoir quel plugin nous allions commencer à faire et comment nous allions pouvoir utiliser les annotations pour récupérer les plugins.

La création d'un git à travers la plateforme GitHub, afin de permettre le travail en groupe de l'équipe autour d'un projet, a été réalisée en début de projet. Suivant les consignes du projet, la création du projet sous maven a été fait en parallèle de cela.

Nous avons ensuite réalisé une première version de l'application ou les plugins étaient dans le path afin de rendre plus facile la première implémentation. Il a été décidé de séparer les plugins du core même de l'application une fois que le projet nous paraîtrait assez bon.

Au fur et à mesure de notre développement nous avons réalisé trois versions différentes de template pour permettre à tous de faire des plugins.