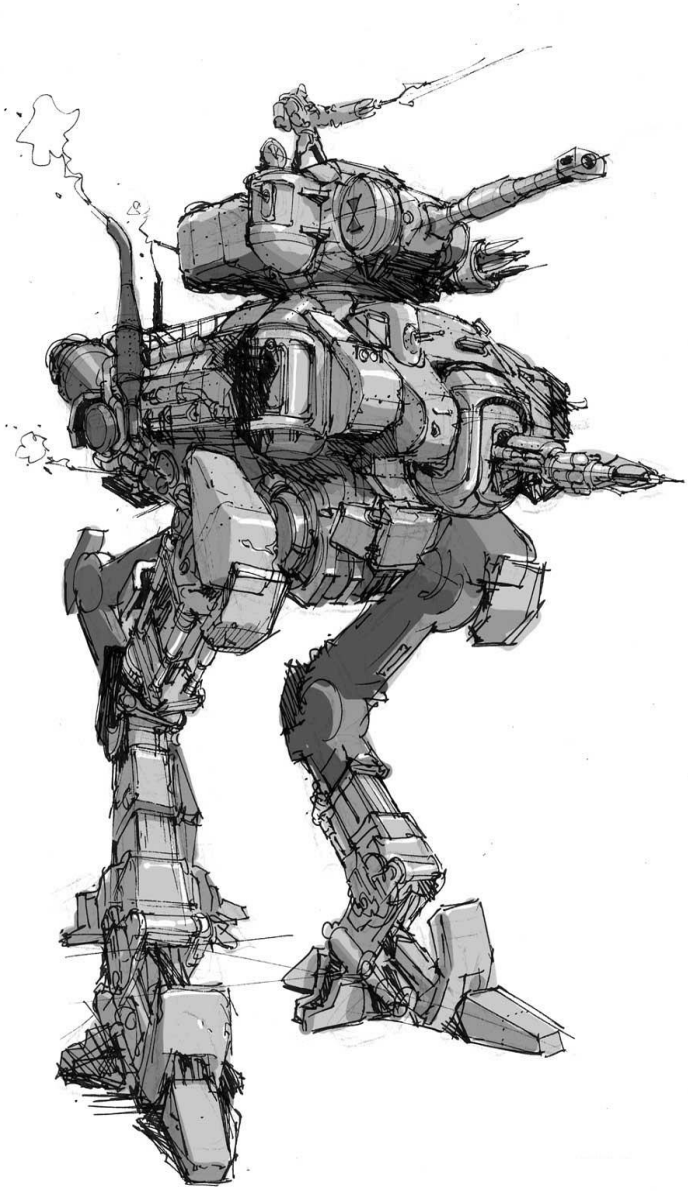
Université Nice Sophia Antipolis

miage nice



# RobotWar

## Projet de programmation avancée 2016-2017

Par
PIERRE Laeticia
TESTANIERE Frédéric
TUAILLON Pierre

Encadré par M. Fabrice Huet

Table des matières

# I. Participation des membres de l'équipe

Chaque membre est associé à des parties qu'il a principalement réalisées :

- Laeticia PIERRE : persistance et plugins
- Frédéric Testanière : interface graphique et plugin loader
- Pierre Tuaillon : menu graphique et moteur de jeu

Cependant, chaque membre a été volatile et a pu toucher à d'autres parties, autres que celles qui lui ont été associées, selon les avancées et difficultés rencontrées lors du projet de chacun pour assurer la cohésion de celui-ci.

# II. Procédure à suivre pour tester notre projet

## 1. Pour le jar

Il suffit de le lancer via un double click ou via la commande java : java -jar Game.jar

il faut impérativement qu'il y ait un dossier nommé Plugins, et contenant un dossier plugins qui lui même contiendra les .class des plugins, au même niveau que le .jar.

La création du jar s'est faite par maven de la maniére suivante. *Apache Maven* propose dans son cycle de vie de production la phase « package », c'est cette phase que j'ai configurée en y ajoutant le maven-assembly-plugin. J'ai ajouté un plugin maven au projet de cette maniére :
Clique droit sur le projet, « Maven », « Add Plugin », saisir « maven-assembly-plugin » dans la barre de recherche, sélectionner le résultat, cliquer sur « OK ».
Enfin j'ai modifié le pom.xml du projet pour qu'il ressemble à ceci :

```
<plugin>
```

```
 <groupId>org.apache.maven.plugins</groupId>

 <artifactId>maven-assembly-plugin</artifactId>

<configuration>

 <appendAssemblyId>false</appendAssemblyId>

 <finalName>${project.artifactId}</finalName>
<outputDirectory>${basedir}\\export</outputDirectory>
 <descriptorRefs>
 <descriptorRef>jar-with-dependencies</descriptorRef>
 </descriptorRefs>
 <archive>
 <manifest>
 <mainClass>moteur.MoteurDeJeu</mainClass>
 </manifest>
 </archive>
 </configuration>
<executions>
 <execution>
 <id>package-jar-with-dependencies</id>
 <phase>package</phase>
 <goals>
 <goal>single</goal>
 </goals>
 </execution>
 </executions>
</plugin>
```

La **donnée en rouge est à remplacer** dans le cas de notre application. C'est le chemin vers notre point d'entrée de l'application (classe où il y a le main).

Les **données en vert sont facultatives** mais bien utiles, elles permettent de personnaliser le nom du fichier généré ainsi que le dossier de destination.

Enfin Clique droit sur le projet, « Run As », « Maven build ».

### 2. Pour eclipse

Il suffit simplement de mettre le projet sous eclipse et de le lancer.

# III. Calendrier des grandes étapes du projet

**Laeticia**

Plugins : 21 Novembre au 4 décembre

Persistance : 21 Novembre au 2 décembre

**Frédéric**

Interface utilisateur : 30 Octobre au 4 Décembre

PluginLoader : 3 Novembre au 2 Décembre

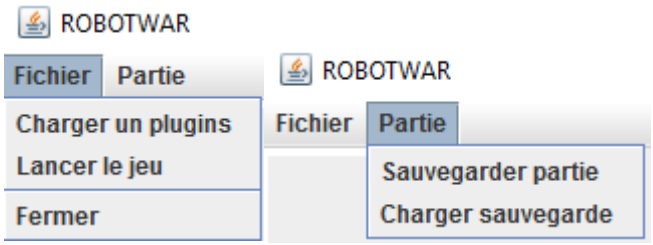
**Pierre**

Moteur de Jeu : 29 Octobre : 21 Novembre au 2 Décembre

Interface utilisateur : 1 Décembre au 3 Décembre

# IV. Fonctionnalité

Comme l'indique le menu graphique plusieurs fonctionnalitées sont disponibles. Elles sont séparées en 2 parties, une partie Fichier et une partie Partie.




La partie Fichier permet entre autres de charger des plugins. Lors d'un clique sur "Charger un plugins" une fenêtre graphique s'ouvre pour sélectionner le fichier voulu.

Le lancement du jeu permet de lancer le jeu pour la première fois ou de reprendre le jeu après qu'il ait été mis en pause via un accès au menu.

Et enfin "Fermer" qui permet de fermer la fenêtre graphique.

La seconde partie du menu (Partie) gére la sauvegarde.

Au clique sur "Sauvegarder Partie" ou sur "Charger sauvegarde" une fenêtre graphique s'ouvre pour, dans le cas de la sauvegarde, choisir l'emplacement de la sauvegarde et dans le cas du chargement, pour choisir le fichier à charger.

Le déroulement d'une partie est le suivant.
Les robots (dont le nombre est indiqué lors de l'instanciation du moteur) sont créés avec les plugins par défaut et une position aléatoire.
Chacun leur tour, ils procèdent à :

- Une phase de déplacement en faisant appel à leur plugin de déplacement
    - Si le déplacement demandé provoque un chevauchement avec un autre robot, on réinterroge le robot

- Une phase d'attaque en faisant appel à leur plugin d'attaque
    - Le moteur vérifie que le robot attaquant a assez d'énergie (coût déterminé par l'attaque) pour effectuer son attaque
        - Si oui : le moteur retire de l'énergie au robot attaquant
        - Si non : le robot passe sa phase d'attaque
    - Si des robots se trouvent aux cases attaquées, le moteur diminue le nombre de PV des robots touchés selon la puissance de l'attaque effectuée
    - Si un robot voit ses PV tomber à 0, celui-ci est retiré du jeu (et donc également de l'interface)

A la fin de chaque tour, de l'énergie est rendue à tous les robots.
Le jeu continue ainsi jusqu'au dernier survivant.

# V. Persistance

*Dans le fichier SystemeSauvegarde*

Nous sauvegardons l'état des robots : tous leurs attributs, dont les plugins utilisés. Pour ce faire, nous parcourons tous les attributs des robots par réflexivité (parcours *getDeclaredFields()* en traitant les cas primitifs, objets simples et plugins). Ces informations sont stockées dans une HashMap qui est elle-même sérialisée et stockée dans un fichier. Nous utilisons pour cela *ObjectOutputStream* avec la méthode *writeObject*.

Pour charger la sauvegarde, nous récupérons tout d'abord la HashMap désérialisée (méthode *readObject* de *ObjectInputStream*).

*Dans le fichier MoteurDeJeu*

Celle-ci est parcourue : pour chaque clé, c'est-à-dire pour chaque attribut, nous récupérons le setter associé (*invoke()*) pour affecter au robot les données. Si un plugin est rencontré, un appel au plugin loader est effectué en amont. Pour connaître le type en second paramètre

de la méthode *invoke()* dans le cas d'un plugin, nous parcourons notamment les interfaces *getInterfaces()*.

*Aspects remarquables*

Après la soutenance, on nous a fait part que les objets Robot auraient pu être directement "write", sans passer par une HashMap. Nous pensions en effet que celui-ci se serait appuyé sur le toString() et cela nous a permis de réutiliser ce que nous avions fait en TP dans une situation réelle.

De plus, nous pourrions dans une prochaine version apporter d'autres informations telles que la taille du plateau.

# VI. Exemples de plugins

*Plugins d'attaque*

Nous avons un plugin d'attaque courte portée. Le robot attaque toutes les cases autour de lui (diagonale comprise), avec une portée de 1. Il s'agit du plugin par défaut.
L'autre plugin d'attaque est une attaque de longue portée. Le robot attaque 4 cases aléatoires. Cette attaque est plus puissante que la première.
Un point de couleur (dont la couleur change selon le plugin d'attaque utilisé) apparaît sur un robot attaqué.
Nous avons fait en sorte que les cases visées ne soient pas hors borne (c'est-à-dire en dehors du plateau).

*Plugin de déplacement*

Nous avons un plugin de déplacement (donc par défaut) qui permet au robot d'avancer d'une case de manière aléatoire (diagonale comprise). Si le coup devait sortir le robot du plateau, celui-ci est annulé.

*Plugins graphiques*

Nous avons deux types de plugin graphique : un qui permet de sélectionner :

- La forme (par défaut rectangulaire, qui peut être changée par une forme ronde)
- La couleur (par défaut une couleur aléatoire ou une couleur aléatoire dont l'opacité diminue en fonction du pourcentage de PV restants)

# VII. Suivi (documentation/gestion du projet)

Le projet est entièrement alimenté par de la JavaDoc.

Les membres de l'équipe ont tous été actifs sur le projet et le travail a bien été distribué. La communication s'est très bien déroulée.

Nous nous sommes appuyés sur git, chacun déposait de nouvelles parties de manière régulière. Nous avons utilisé les tickets, ceux-ci étaient très globaux au départ puis se sont affinés au fil du projet et nous ont été très utiles.

# VIII. Chargement dynamique

Le chargement dynamique des plugins est assuré par la classe PluginLoader (singleton) accessible de partout dans le programme grâce à une unique instance. Son rôle est de récupérer et d'instancier une classe via URLClassLoader ainsi qu'un Path.

*CDep = PluginLoader.getInstance().loadPlugin("Plugin_Deplacement_Aleatoire");*

Le chargement s'effectue s'effectue une première fois au démarrage du programme avec des plugins par défaut puis peut se faire via le menu de l'interface graphique.

Une fois les plugins instancier en classe, ils sont typés en fonction de leurs interfaces (Attaque, Déplacement …) et ajouter au robot.

*robot.setPluginDeplacement() = (Plugin_Deplacement) CDep.newInstance();*

Quand le moteur de jeu demande au robot de faire son action de déplacement, on appelle alors la méthode du plugin qui nous renvoi la nouvelle position du robot.

*robot.position = robot.plugin_Deplacement.getActionDeplacement();*

# IX. Interface Graphique

L'objectif principal était de rendre indépendant l'interface graphique et le moteur de jeu de façon à ce qu'un changement de technologie graphique ne perturbe pas le projet si nécessaire. De ce fait, la communication entre le moteur et l'interface se fait via le patron Observer/Observable.

L'interface est découpée en deux grandes parties, la première contient un gridLayout de panel qui sert d'arène au robot (représenté dans le projet par la classe Arene). Cette arène est un ensemble de JPanel accueillant les représentations graphiques des robots (GRobot).

La seconde partie concerne le menu utilisateur, celui-ci permet à l'utilisateur de charger des plugins, lancer le jeu, sauvegarder sa partie, charger une partie et fermer le jeu. Lorsque l'utilisateur clique sur le menu, le jeu est automatiquement mis en pause, pour reprendre la partie il suffit de cliquer sur "Lancer le jeu".

Le rafraîchissement de l'interface se fait via la méthode *update()* de l'observer qui est appelé lorsqu'un robot effectue un déplacement (on déplace le GRobot dans le nouveau JPanel correspondant) ou si une attaque touche un robot (on affiche une animation sur le robot touché).

# X. Modularité

Le projet est découpé en 3 modules maven, le premier contient le jeu en lui même (Moteur de jeu, Interface graphique, PluginLoader). Le second est là pour stocker les interfaces des plugins, et le troisième contient les plugins disponibles par défaut.

Les dépendances sont faites de la façon suivante, Le moteur et les plugins dépendent des interfaces de plugins. Les interfaces quant à elles, ne possèdent aucune dépendance.

Les dépendances nous permettent d'utiliser le contenu des répertoires visés. Dans notre projet, elles se présentent de la façon suivante :

- Le moteur dépend des interfaces de plugins : les robots peuvent ainsi posséder des plugins en attributs, ils peuvent être déclarés...
- Les plugins dépendent des interfaces de plugins : ils peuvent ainsi les implémenter
- Les interfaces quant à elles ne possèdent aucune dépendance

La configuration maven a été effectuée via les pom.xml notamment par le biais des balises modules et dependencies.

# XI. Exemple de création d'un plugin

Créer une classe qui implémente l'interface du type du plugin que l'on veut créer ainsi que l'interface Serializable pour la sauvegarder, pour l'exemple nous créons un plugin de déplacement.

```
public class Plugin_Deplacement_Aleatoire_Une_Case implements Plugin_Deplacement, Serializable
```

Implémenter les méthodes de l'interface du plugin et les compléter.

```java
public class Plugin_Deplacement_Aleatoire_Une_Case implements Plugin_Deplacement, Serializable {

    private static final long serialVersionUID = -2187265944145739198L;

    /**
     * Nouvelles coordonnees aleatoires avec deplacement d'une case (inclus diagonale)
     *
     * @see plugins.Plugin_Deplacement#getNouvellePosition(java.awt.Point, int, int)
     */
    public Point getNouvellePosition(Point positionActuelle, int longueurArene,
            int largeurArene) {
```

Compiler votre classe de façon à obtenir le .class et chargé le via notre interface graphique.