CHEROUVRIER Léonard
LAUNAY Corentin
TEMIN Alison

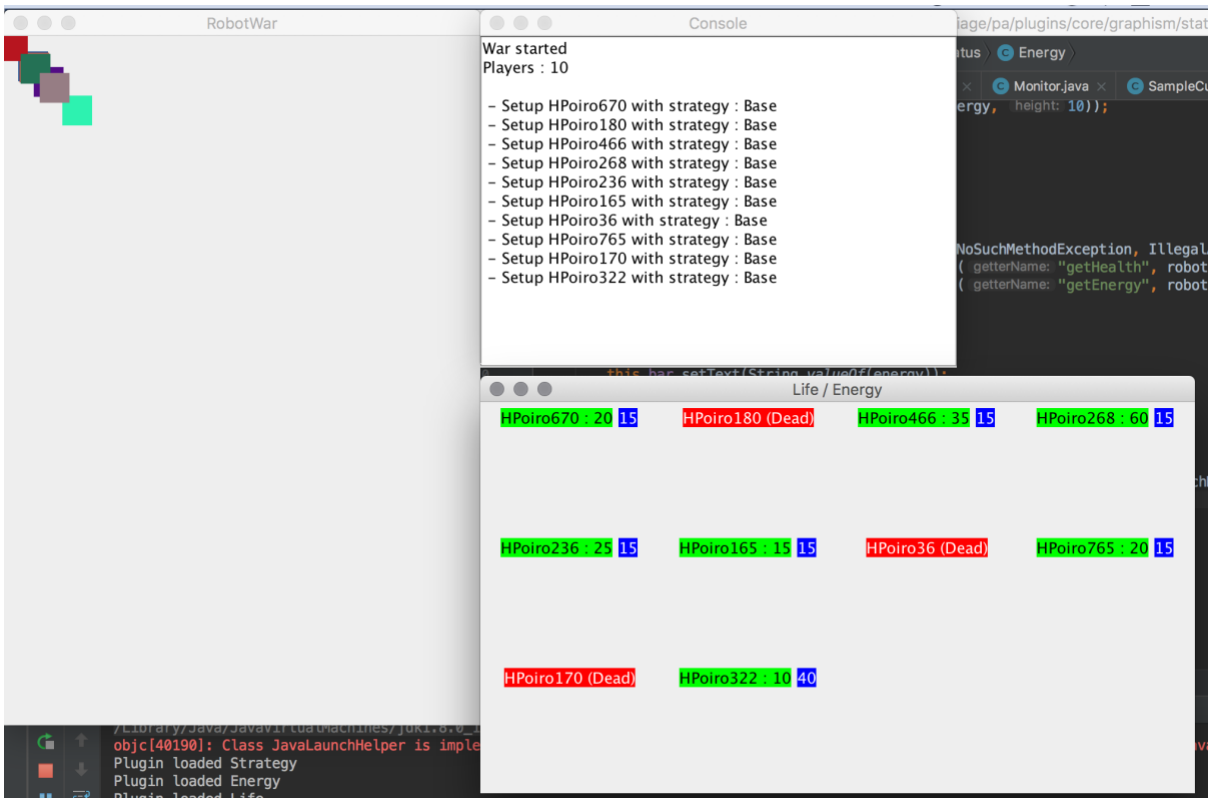
Rapport
Programmation avancée

Equipe : CHEROUVRIER Léonard
LAUNAY Corentin
TEMIN Alison

Master 1 Miage

Rendu final :

CHEROUVRIER Léonard
LAUNAY Corentin
TEMIN Alison



<u>Participation de chaque membres de l'équipe :</u>

Launay Corentin :

- Création de la console et conversion en plugin
- Création de la classe Life, ses graphismes et conversion en plugin
- Création de la classe Energy, ses graphismes et conversion en plugin
- Mise en place du tour à tour des différents joueurs et déclaration du winner
- Update de la vie et de l'energie
- Création du début du BoardMonitor

Cherouvrier Léonard :

- Conversion en projet Maven
- Définition de l'architecture de base

- Modification classLoader pour coller au système de plugins
- Système de trait permettant d'appeler facilement une méthode par reflection
- Amélioration BoardMonitor permettant de gérer le déroulement complet du jeu
- Système d'override des plugins de type stratégie / mouvement
- Canevas de Plugin (type stratégie / mouvement)
- Amélioration de la stratégie et modification de 1 contre 1 vers mêlée
- Conversion des barres d'énergie et refactor global du plugin de graphisme pour gérer la mêlée

Temin Alison :

- Graphisme Robots
- ClassLoader pour charger les plugins
- Méthode pour bouger un robot (axe des x)
- 1ère stratégie d'attaque d'un robot vs un robot
- Déplacement robot sur axe des y
- Participation à l'élaboration d'une seconde stratégie (slack)
- Participation à l'élaboration afin de passer d'un combat 1 contre 1 à un combat de plusieurs robots
- Initiation de la création d'une fenêtre pour y mettre les barres d'énergie / vie

Procédure à suivre afin de tester le projet :

Dans le module plugins lancer :
mvn clean compile -X
Dans le module core lancer :
mvn clean install

Un path va être généré, il faut utiliser le jar nommé "with-dependencies" puis exécuter à la racine du projet : (là ou se trouvent les modules core et plugins)
java - jar *pathGénéré nombreDeRobots*

Calendrier des grandes étapes du projet :

LAUNAY Corentin	Cherouvrier Léonard	Temin Alison
04/01 : Life & Energy	**08/12** : Architecture	***08/12 :*** Graphisme robot

05/01 : Console	**04/01** : Maven / Annotation Trait	***12/12 :*** ClassLoader
05/01 : BoardMonitor	**05/01** : ReflectionUtil	***06/12 :*** Stratégie
06/01 : Tour par tour	**10/01** : Stratégies modifiables	***08/12 :*** Elaboration seconde stratégie

- **Fonctionnalité :**

L'utilisateur décide d'un nombre de robot qui vont se combattre.
Les robots possèdent de l'énergie et de la vie.
Ils se déplacent sur la carte jusqu'à trouver un autre robot, le combat commence alors.
Lors d'un combat, le robot attaquant perd de l'énergie mais fait diminuer la vie de l'adversaire.
L'énergie remonte automatiquement afin de permettre à un robot d'attaquer.
Lorsqu'un robot voit sa vie descendre à zéro, il disparait du plateau et laisse alors son adversaire rejoindre un autre combat.
Le jeu se termine lorsqu'il ne reste qu'un robot en vie.

- **Chargement dynamique :**

Les plugins sont chargés via un système d'Autoloading, à la manière de PHP
Le système d'autoload utilise l'annotation Plugin pour décider ou non de quoi charger.

Le mécanisme de plugins utilise une annotation PluginTrait permettant d'appeler de manière simple une méthode par reflection.

- **Persistance :**

Chaque plugin est compilé par maven en .class (configuration Plugins)
Les Plugins sont chargés automatiquement en fonction des .class trouvés, en somme une fonction récursive se charge de chercher les .class en partant du point ou le jeu est lancé

- **Modularité** :

Le projet est un Maven multi-project, aucune dépendance externe à Java / Maven n'est utilisée. Les plugins sont un module, le core un autre.
Il est relativement aisé de créer un Plugin générique, néanmoins le coeur peut nécessiter quelques adaptations.

En revanche il est très aisé de créer des stratégies personnalisées qui seront ensuites appliquées à tous les robots. (Uniquement si le fichier commence par Custom)

- Suivi : Le suivi a été fait par Slack, chacun détaillant son travail à chaque fois qu'un push était réalisé pour que tout le monde comprenne où le projet en était concrètement.
  Pour le Git, nous avons utilisé GitHub.

CHEROUVRIER Léonard
LAUNAY Corentin
TEMIN Alison

Procédure pour créer de nouveaux plugins :

Au sein du module plugins, on distingue deux packages, l'un contenant les plugins requis au bon fonctionnement du jeu, de l'autre côté un package "Custom" contenant un exemple de stratégie / mouvement.

```
package fr.unice.miage.pa.plugins.custom;

@Plugin(name="SampleCustomStrategy", type="core", required=0)
public class SampleCustomStrategy {
  private final String name;
  private final Object monitored;
  private final ArrayList opponents;
  private final HashMap weaponCapabilities;
  private final HashMap<String, Class<?>> plugins;
  private Integer nextMoveY;
  private Integer nextMoveX;

  public SampleCustomStrategy(Object monitored, ArrayList opponents, HashMap weaponCapabilities, HashMap<String,
Class<?>> plugins){
    [...]
    this.plugins = plugins;
    this.name = "SampleCustomStrategy";
  }
  @PluginTrait(type="movements", on="strategy")
  public void movements() throws InvocationTargetException, IllegalAccessException {
    // Implement random strategy by setting this.nextMove values
    // core plugin RandomMove could be used

    this.nextMoveX = 0;
    this.nextMoveY = 0;
  }

  @PluginTrait(type="decide", on="robot")
  public Object decide() throws Exception {
    // Implement decide strategy
    // Iterate over this.opponents check health for example
    // Return wanted bot
    // 404 Bot not found atm
    return null;
  }

  /**
   * Get strategy name
   * @return Base
   */
  @PluginTrait(type="strategyName", on="strategy")
  public String getName(){
    return this.name;
  }
```

Comme montré dans l'extrait de code ci-dessus, il suffit de renommer son fichier en "CustomStrategy" ou en tous cas un nom commençant par Custom, et de modifier les deux méthodes decide() et movements() pour que le jeu se charge automatiquement de mettre la stratégie à un robot sur deux.

La méthode movements() contient un exemple montrant qu'un plugin de type "Movements" peut être utilisé. Les stratégies contenant l'ensemble des plugins du jeu (ci-fourni constructeur), il est aisé de mêler plusieurs plugins ensembles.

Pour créer un autre type de Plugin, il suffit de le mettre dans le package **custom**, avec un nom commençant par Custom pour que la classe soit chargée (il faudra bien sûr l'annoter avec @Plugin..), néanmoins le coeur du jeu devra probablement être modifié pour permettre l'instanciation (à part une nouvelle stratégie de mouvement), ou alors il faudra invoquer la méthode d'un autre plugin via un trait (sans oublier d'instancier le dit plugin) au moment opportun.

Les stratégies contiennent par défaut tous les plugins afin de faciliter la modification du jeu.

```
@Plugin(name="CustomMove", type="movement", required=0)
public class CustomMove {
  @PluginTrait(type="move", on="robot")
  public int nextPlace(){
     Random generator = new Random();
     return generator.nextInt(30) + 1;
  }
  @PluginTrait(type="moveY", on="robot")
  public int nextPlaceY(){
     Random generator = new Random();
     return generator.nextInt(400) +1 ;
  }
}
```

Pour créer un nouveau type de mouvement qui animera les robots, il suffit de créer une nouvelle classe commençant par "Custom" et se terminant par "Move" pour qu'elle soit directement reliée, y compris sur une stratégie personnalisée.

Pour ces deux types de plugins personnalisables, il suffit de renommer les "Sample" pour avoir le rendu avec une stratégie différente (stupide, mais différente).

Description des plugins : *(la flèche signifie sous-plugin du plugin pré-cité)*

Graphisme :

Dessin des robots grâce à des JLabel basiques, mis à jour en même temps que la position des X / Y du robot change.

> Console

Affichage des System.out au sein d'une JFrame, elle servait beaucoup en cas de debug, et fournit un moyen simple et pratique de communiquer à l'utilisateur. Étant directement reliée au System.out une fois instanciée elle récupère tous les messages et les affiche (y compris les erreurs).

> Life (Affichage des barres de santé) et Energy (barres d'énergies)

Ces plugins (qui ont un comportement relativement similaire) sont en fait des JLabel contenant les informations avec une couleur de fond définie, le monitor se charge de les placer dans une nouvelle frame et de les mettre à jour à chaque action d'un robot, ils récupèrent directement les informations sur le robot en utilisant getterOnBot de la classe utilitaire "PluginUtil".

Attacks :

> Weapons : *Contient l'ensemble des armes possibles, les capacités des armes sont directement écrites comme Annotation. Chaque arme est en réalité représentée par sa propre classe, annotée afin de récupérer rapidement les attributs qui lui sont propres (distance, attaque de base…).*

Strategy : *Plugin de stratégie de base*

La stratégie de base est décomposée en plusieurs traits permettant au jeu d'éviter au maximum la triche par plugin (le jeu vérifie via la stratégie de base si le joueur peut attaquer), deux de ces traits sont annotés Overridables pour clarifier le fait que ces méthodes sont modifiables au sein d'une stratégie personnalisée.

La stratégie de base cherche un joueur quelconque plus haut et l'attaque, les mouvements sont déterminés par un plugin de Mouvement, décris après.

RandomMove :

Plugin permettant de générer deux positions X / Y aléatoires, il se base sur la classe Random de Java avec des bornes définies de sorte à ne pas sortir de la fenêtre de jeu.