



## **MANUEL WARBOT III**

**Prise en main d'un jeu de stratégie multi-agent de robots que vous avez à programmer**

**Jacques Ferber**

**LIRMM - Université de Montpellier II**

**Année 2014-2015**

**Version 1.0.1 du Manuel**

### **I. Introduction**

WARBOT est un jeu dédié à la simulation dans un environnement logique et graphique d'un jeu de stratégie pour y intégrer une intelligence artificielle agissant au niveau individuel.

Warbot est à la fois un jeu et une plate-forme d'évaluation et d'analyse de techniques de coordination entre agents, dans une situation de compétition où deux équipes de "robots" s'affrontent pour détruire la (ou les) base(s) de l'adversaire.

Dans ce projet, les joueurs sont en fait les développeurs des agents. Mais ils ne doivent faire qu'une seule chose: développer les "cerveaux" (brain) de ces robots sachant que les "corps" (body) sont définis une fois pour toutes par les règles du jeu. De ce fait, la compétition réside dans la qualité de la programmation de ces cerveaux, et dans les stratégies de coordination proposées. L'agent a des capacités de perception limitées (il ne peut percevoir ce qui se passe que dans son champ de perception) et décide de manière autonome les actions qu'il a à réaliser en fonction de ce qu'il perçoit et des messages qu'il reçoit des autres agents.

L'environnement décidera du résultat de son action (et celle des autres agents) en fonction des lois de cet environnement, produisant un nouvel état du monde.

## Histoire et enjeux

L'étude de la coopération entre agents constitue l'un des aspects les plus importants des systèmes multi-agents. Mais les situations de conflits ou plus exactement de compétition sont au moins aussi importantes que les situations de coopérations. De nombreux exemples ont été proposés dans l'histoire des systèmes multi-agent pour aider à comprendre et à étudier ces situations: poursuites " proies-prédateurs ", recherche d'échantillons, agence de voyage, robots footballeurs, etc. sont des exemples de telles situations.

Warbot constitue un environnement de tests de techniques de coordination entre agents dans un contexte similaire à celui des jeux video, dans lequel deux armées de robots s'affrontent. Le but d'une équipe est de détruire les bases du camp adverse avant que l'autre ne le fasse... Un peu bourrin, mais assez simple finalement à comprendre...

La particularité de Warbot vient de ce que les corps des robots ne peuvent pas (et ne doivent pas) être modifiés par les joueurs, lesquels n'ont qu'une possibilité: définir le comportement des robots de manière à ce qu'ils coopèrent entre eux.

Il existe une grande différence entre Warbot et un jeu vidéo: dans Warbot, lorsque un match est lancé, les êtres humains n'ont pas le droit d'intervenir dans le déroulement du match, à la différence des jeux video où les joueurs peuvent manipuler les entités avec la souris ou le clavier.

Le projet Warbot a débuté en 2002 sur une idée de Jacques Ferber. La plate-forme a été réalisée par Fabien Michel et Jacques Ferber.

Plusieurs tournois ont eu lieu, un par année en 2002, 2003, 2004 et 2005 dans le cadre du DEA de Montpellier. Les comportements des équipes des années 2002 à 2004 sont distribués avec MadKit. Les équipes 2005 et suivantes seront fournies sur ce site..

Dans ce document, vous trouverez tout ce dont vous avez besoin pour connaître le fonctionnement de l'environnement, des percepts et des actions.

## II. Les différents types d'agents

Il y a trois types de corps dont il s'agit de programmer le comportement : bases, explorers et rocket-launchers.

### 1) Bases

Les **bases** représentent le quartier général de votre équipe. Ce sont des agents ne pouvant pas se déplacer physiquement dans le monde. En revanche elles peuvent construire des unités de type exploreurs et rocket-launchers.

### 2) Explorers (explorateurs)

Les **explorers** sont des agents dédiés à l'exploration du monde. Ils sont pacifistes et ne peuvent donc pas attaquer d'unités ennemies. Ils peuvent se déplacer et servent donc essentiellement à découvrir le territoire, le surveiller, récupérer de l'énergie sous forme de nourriture dispersée dans l'environnement. À terme ils permettront de donner des informations sur le reste du monde aux autres agents grâce à un module de communication.

### 3) Rocket-launchers (lanceurs de missiles)

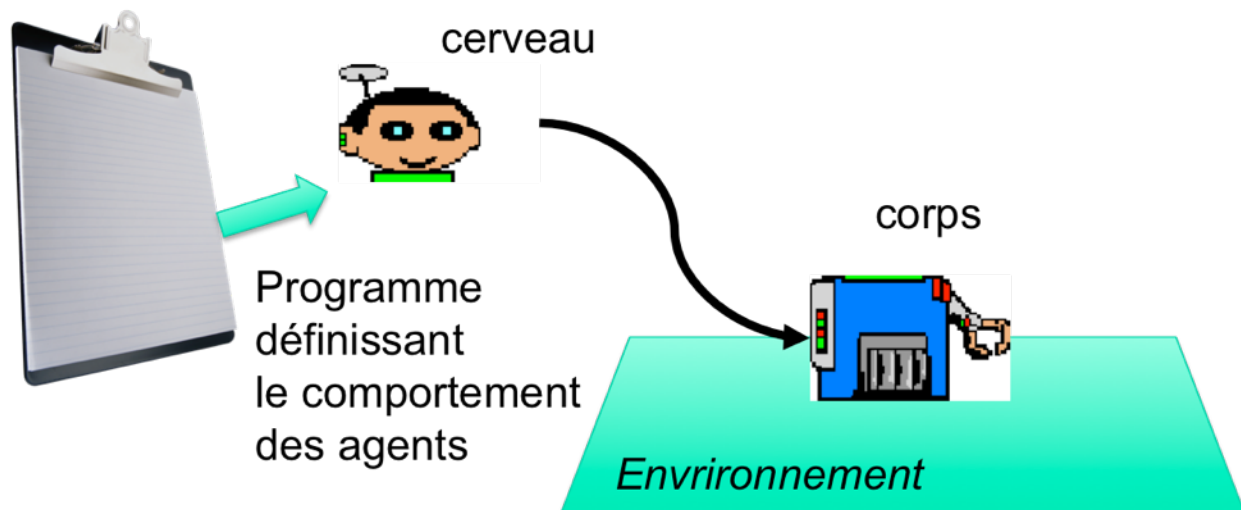
Les **rocket-launchers** (lanceurs de missile) sont les seules unités à pouvoir se battre. Ils peuvent se déplacer dans l'environnement et tirer des missiles (qui affectent les amis comme les ennemis). Ils ne peuvent tirer un missile que tout les  $k$  tours ( $k$  représentant un nombre de ticks prédéterminé). Contrairement à son prédécesseur, les rocket-launchers ne peuvent plus stocker plusieurs missiles. Les missiles seront construits automatiquement, quand l'action « tirer » sera appelée. Bien entendu, cette action entraînera une perte de vie du rocket-launcher. Les coûts des différentes unités sont indiqués plus bas dans le document.

Avec la version 3.0 de Warbot, de nouvelles unités ont été ajoutées:

- **Kamikaze** : ce sont des robots kamikazes qui ont peu de vie, sont rapides et ont un fort pouvoir de destruction. Leur principal avantage est leur force d'attaque mais ils meurent lorsqu'ils attaquent.
- **Turret (tourelles)** : ce sont des unités statiques qui ne peuvent se déplacer. Leur avantage est leur possibilité d'envoyer des missiles qui infligent de lourds dégâts. Elles ne peuvent être créées que par des ingénieurs.
- **Engineer** : ce sont les ingénieurs capables de créer des tourelles.

### III. Principes

Les agents sont composés d'un «corps» correspondant aux types d'unités mentionnées ci-dessus, et d'un cerveau. C'est le cerveau qui peut être programmé et qui contrôle le comportement des corps des robots (figure 1). Les cerveaux des robots contrôlent le comportement des «corps» des robots. Le langage de prédilection pour programmer ces cerveaux est Java.



**Figure 1.** les agents ont un corps et un cerveau. Les joueurs doivent programmer le cerveau des agents.

Warbot est défini par l'ensemble des principes suivants:

1. Les agents se déplacent dans un univers en 2D avec des coordonnées continues (ils ne se déplacent pas sur une grille). Ils se déplacent d'un certain nombre de «pixels» par unité de temps. Pas d'accélération, de masse, ni de physique du monde.

2. Pas de coordonnées globales: Les agents n'ont qu'une vision locale de leur environnement et ils ne connaissent pas leur position en termes de coordonnées x, y (pas de GPS, désolé, mais c'est voulu).
3. Formes simples: les agents prennent la forme d'un cercle dans l'espace et leur taille est exprimé par un 'radius' (rayon). Warbot prend en compte les collisions. Quand un agent rencontre un autre ou atteint les limites de l'espace du jeu, il ne peut aller plus loin.
4. Les perceptions des agents sont réduits. La plupart des agents n'ont qu'un cône de perception (certains ont un cercle complet de perception). A chaque unité de temps, ils peuvent obtenir la liste des choses qu'ils perçoivent.
5. Un agent ne peut faire qu'une action physique à la fois par unité de temps: avancer, tirer, etc. En revanche, il peut faire autant d'actions cognitives (raisonnement, envois de messages) qu'il désire.
6. Les agents ont des points de vie. Lorsque ces points de vie arrivent à zéro, ils meurent. Les points de vie ne diminuent que s'ils reçoivent sont blessés par des armes (missiles).

Programmer un agent c'est un peu comme diriger un sous-marin: imaginez-vous enfermé dans un sous-marin. Vous avez pour seule information un moniteur qui vous délivre certaines informations (écho sonar, état de la coque...) et un ensemble de manettes à tirer devant vous. La carlingue et l'océan décident pour vous du résultat. C'est exactement ce qui passe pour Warbot. Chaque agent est autonome et décide de ses actions en fonction du programme que vous lui avez placé, et le meilleur moyen de comprendre ce qui se passe dans la «tête» d'un agent, c'est de se placer en vue subjective.

## IV. L'interface

### 7. L'interface du launcher

L'interface du launcher (lanceur) s'affiche au lancement de Warbot 3, après avoir chargé les équipes. L'utilisateur peut y paramétrer sa simulation (choix des équipes, nombre et types d'unités, etc.).

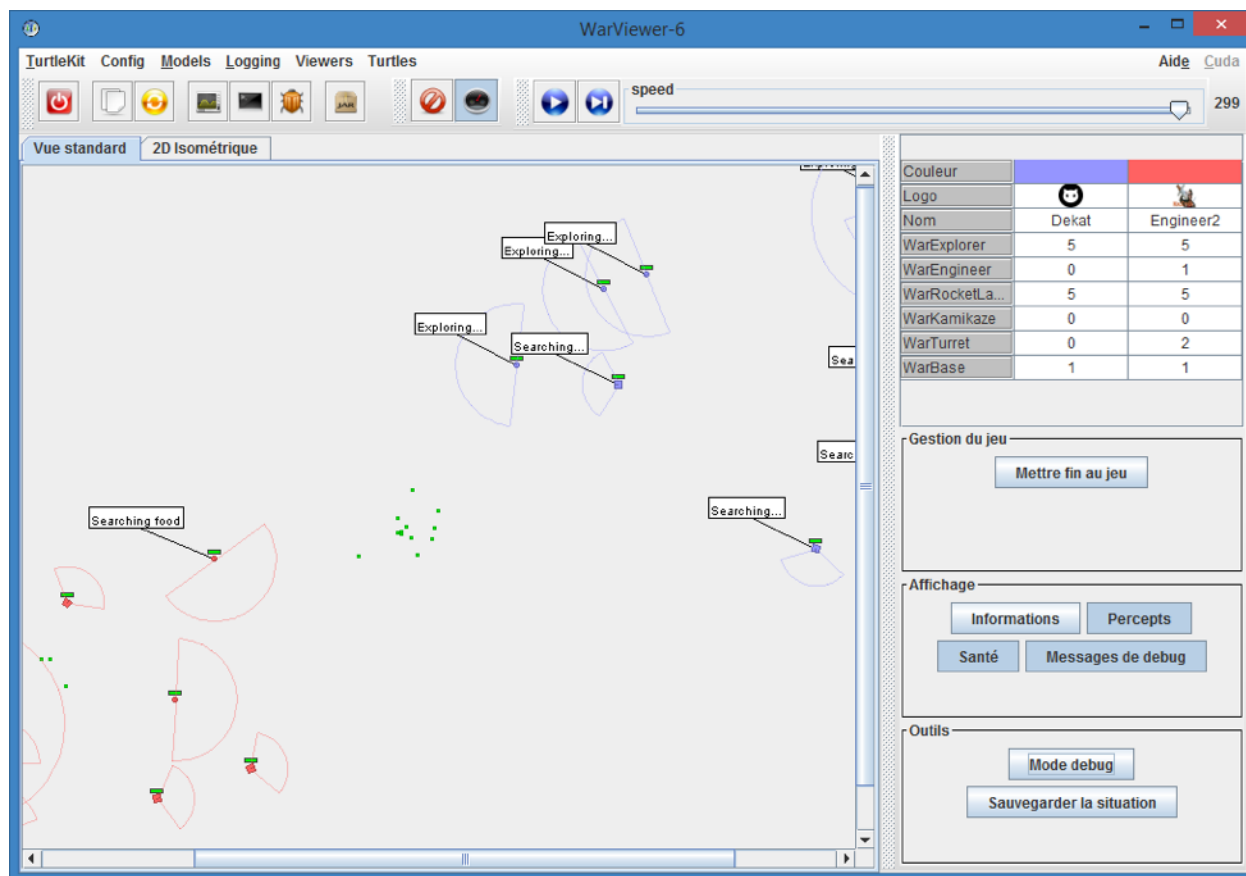


**Figure 2:** Capture d'écran de l'interface de lancement de Warbot

La fenêtre est divisée en plusieurs parties :

- une barre de menu permettant de charger une situation ou de lancer l'éditeur de FSM ;
- le panneau central est composé de deux onglets correspondant au mode duel (un combat entre deux équipes) et un mode tournoi, où plusieurs équipes se rencontrent. Dans ce panneau, on sélectionne les équipes. Le joueur peut faire jouer une équipe contre elle-même ;
- un panneau sur le côté droit, qui contient des options concernant la simulation avec des curseurs de défilement pour choisir le nombre de robots pour chaque type et par équipe au départ. Il est également possible de

choisir dans les options avancées le niveau d'affichage des logs à l'aide d'un menu déroulant et une case à cocher si l'utilisateur souhaite utiliser la visualisation en 2D isométrique.



**Figure 3:** Capture d'écran de l'interface



L'interface du Jeu provient de celle définie par Madkit/TurtleKit. Ainsi, la barre de menu ainsi que la barre d'outils proviennent de TurtleKit et permettent de manipuler la simulation.



*La barre d'outils de Turtlekit*

Au centre, nous pouvons voir le champ de bataille. L'utilisateur a la possibilité de zoomer grâce à la molette de la souris et de se déplacer grâce à un glisser-déplacer de la souris.

A droite se trouve la barre d'outils de Warbot. Elle contient d'abord un tableau décrivant la situation actuelle des équipes : leur couleur, leur nom, leur logo ainsi que le nombre restant de chacune leurs unités.

Couleur		
Logo		
Nom	Dekat	Engineer2
WarExplorer	5	5
WarEngineer	0	1
WarRocketLa...	5	5
WarKamikaze	0	0
WarTurret	0	2
WarBase	1	1

*Tableau d'information des équipes*

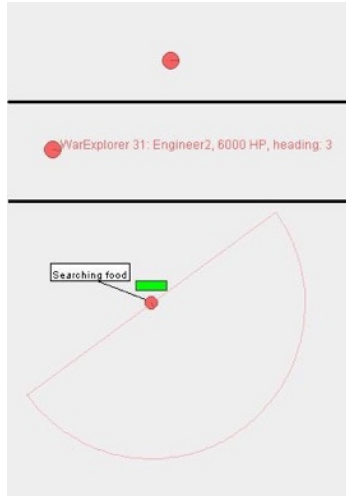
Ensuite pour interagir avec la simulation, l'utilisateur dispose de plusieurs outils :

- stopper la simulation à tout moment et revenir à la page du launcher
- lancer le mode Debug pour modifier le champ de bataille (ajouter, supprimer et déplacer des agents) : un panneau apparaît alors à gauche de la fenêtre ;
- sauvegarder la situation actuelle dans un fichier.

Enfin, l'affichage peut être modifié pour afficher plus ou moins d'informations sur les agents. Ces informations sont :

- un texte d'information général : type d'unité, numéro d'identification de l'agent, nom de son équipe, santé, angle ;
- affichage des percepts : les champs de vision des agents sont alors affichés ;
- les barres de santé ;
- les messages de debug : bulles reliées à chaque agent, qui permettent à l'utilisateur d'y afficher ce qu'il souhaite afin de l'aider au développement.

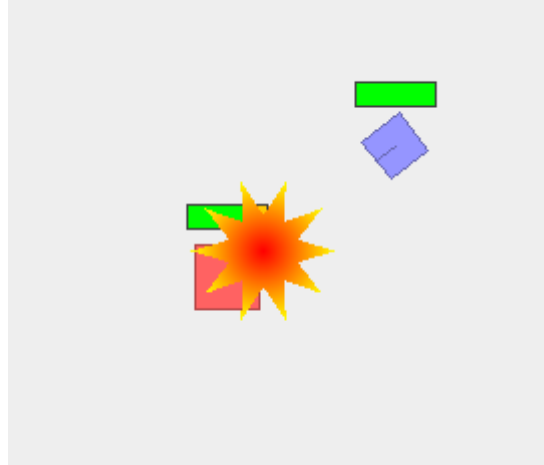




*Différents affichages possibles des informations de l'agent*

Un ensemble d'onglets permettent de passer de la vue standard à la vue 2D isométrique si cette dernière a été chargée (choix possible dans l'interface).

De plus, d'autres informations sont visibles comme le nombre de ticks par seconde et les explosions lors de la mort d'un agent, que ce soit un projectile ou un robot.

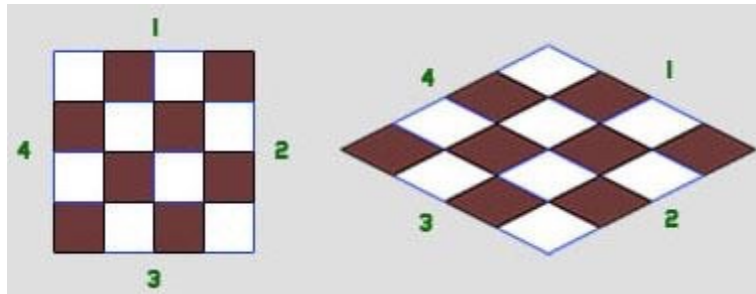


*Affichage d'une explosion*

### 3. La vue 2D isométrique (en cours de développement)

Une vue 2D isométrique a été développée pour Warbot. Elle est proposée en version beta. Une carte isométrique est composée de « tuiles » (tiles) qui sont représentées par une texture. Ainsi une texture de base pour carte orthogonale (vue aérienne ou de côté) diffère de celles pour une carte isométrique.

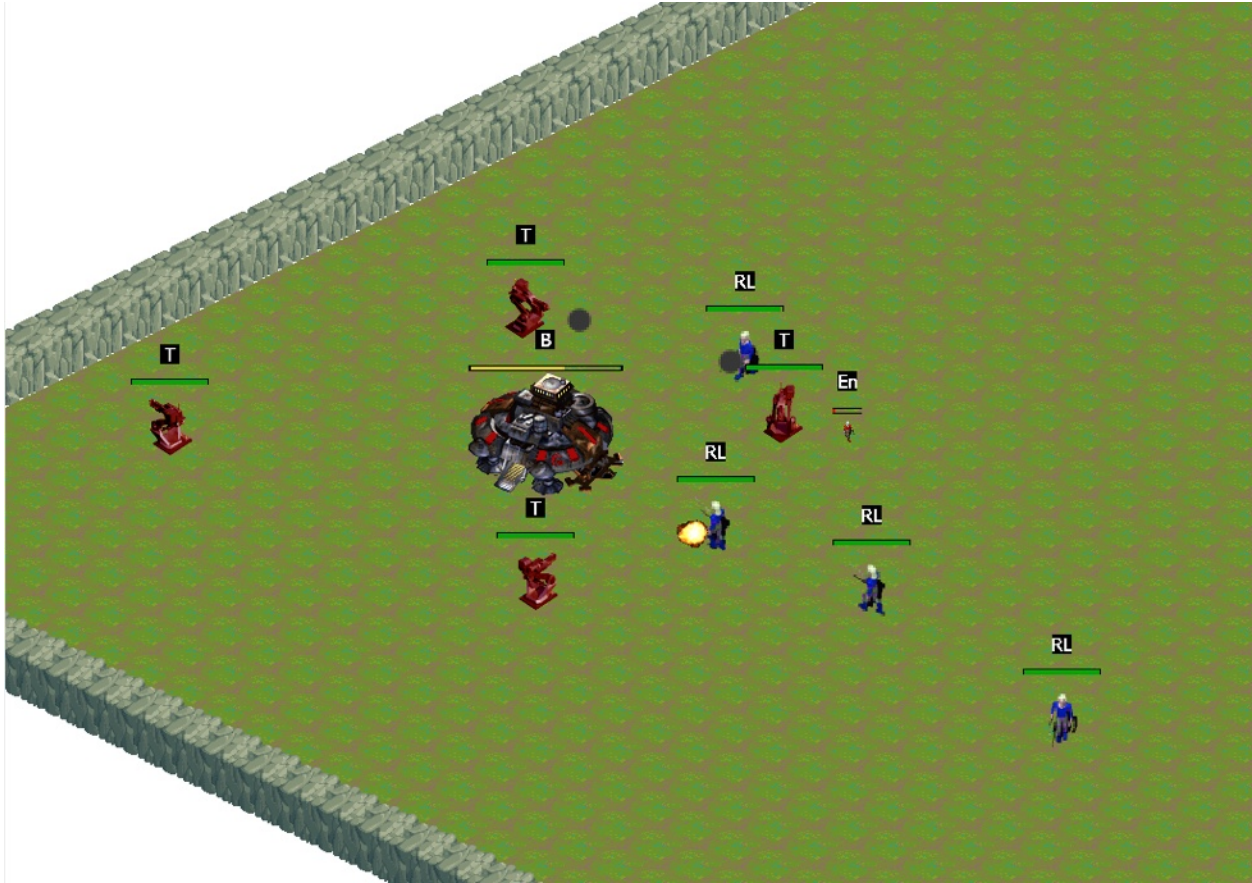
Pour passer une texture en isométrique, une case subit une rotation de 45 degrés et une réduction de sa hauteur. Ainsi, l'intégralité d'une carte peut être convertie en 2D isométrique en appliquant cette transformation sur toutes les cases, comme le montre le schéma ci-dessous.



*Transformation d'une carte en isométrique*

Enfin, chaque robot possède des informations importantes que l'utilisateur souhaite connaître :

- l'unité dont il s'agit, représenté par une icône avec les initiales des unités affichées ;
- la santé des unités ;
- un indicateur, représenté par un triangle jaune, permettant de savoir si l'agent possède de la nourriture.



*Capture d'écran d'une visualisation en mode 2D isométrique*

## V. Liste des actions

1. Chaque action coûte un tour à l'agent. Vous devez donc soigneusement choisir la meilleure action à lui faire effectuer en fonction du contexte qu'il perçoit et de son environnement.

### 1) Actions communes à toutes les unités

- **Eat** : Décrémente le nombre de food dans le sac de 1 et ajoute 200 PV à l'unité.
- **Idle** : l'agent ne bougera plus et ne fera aucune action.
- **Give** : Donner une ressource à un agent. Juste avant cette action, faire un `setAgentToGive(int id_agent)`.

## 2) Actions de l'explorateur

- **Take** : Permet de ramasser une ressource présente sur le terrain. Si le sac est plein, rien ne se passe.

## 3) Actions de la base

- **Create** : Permet de créer un agent de type Explorer ou RocketLauncher. Juste avant cette action, faire un `setNextAgentCreate(String agent)`.

## 4) Actions du rocket-launcher

- **Fire** : Envoyer une roquette en direction d'un agent ennemi. Avant cette action, faire un `setAngleTurret(int angle)`, pour diriger la tourelle en direction de l'ennemi.
- **Reload** : Une fois enclenché, le RocketLauncher doit attendre un laps de temps de 50 ticks avant de pouvoir tirer.
- **Take** : Permet de ramasser une ressource présente sur le terrain. Si le sac est plein, rien ne se passe.

## V. Primitives liées aux agents

Une multitude de primitives ont été mises en place pour permettre d'écrire ces comportements. Certaines renvoient des valeurs booléennes sur des choses que vous aurez très souvent à évaluer avant d'agir, d'autres renvoient des informations plus générales, comme l'identifiant de l'agent, son numéro d'équipe, etc...

Toutes les primitives liées aux agents sont citées ci-dessous, avec la signature et le rôle de chacune. Pour utiliser une primitive, dans un contrôleur, il est nécessaire de faire la commande:

**getBrain().<primitive>(<arg1>, ..., <argn>)**

Exemple:

`getBrain().getHealth()`

retourne la santé de l'agent courant.

## 1) Primitives communes à tous les agents

- **getBagSize ()** : int

Retourne la taille du sac.

- **getNbElementsInBag ()** : int

Retourne le nombre d'éléments dans le sac.

- **isBagEmpty ()** : boolean

Retourne vrai si le sac est plein, faux sinon.

- **isBagFull ()** : boolean

Retourne vrai si le sac est plein, faux sinon.

- **getEnergy ()** : int

Retourne l'énergie actuelle de l'agent.

- **getID ()** : int

Retourne l'id de l'agent.

- **getTeamName ()** : String

Retourne le nom d'équipe de l'agent.

- **getHealth ()** : int

Retourne l'énergie actuelle de l'agent.

## 2) Gestion des percepts

- **isEnemy** (Percept)

Indique si le percept correspond à un agent ennemi. Autrement dit, si l'on perçoit un agent ennemi.

- **getPercepts ()** ArrayList<WarPercept>

Retourne la liste de tous les percepts

- **getPerceptsAllies ()** ArrayList<WarPercept>

Retourne la liste des percepts correspondant à des alliés.

- **getPerceptsEnemies ()** ArrayList<WarPercept>

Retourne la liste des percepts correspondant à des ennemis.

- **getPerceptsResources ()** ArrayList<WarPercept>

Retourne la liste des percepts correspondant à des ressources.

- **getPerceptsAlliesByType**(WarAgentType agentType) ArrayList<WarPercept>

Retourne la liste des percepts d'alliés correspondant à un type d'agent.

- **getPerceptsEnemiesByType**(WarAgentType agentType) ArrayList<WarPercept>

Retourne la liste des percepts ennemis correspondant à un type d'agent.

- **getPerceptsAllies** () ArrayList<WarPercept>

Indique si le percept correspond à un agent ennemi. Autrement dit, si l'on perçoit un agent ennemi.

### 3) Gestion des messages

- **getMessages** () : ArrayList<WarMessage>

Retourne une liste contenant l'ensemble des messages reçus au tick actuel.

- **sendMessage** (int idAgent, String message, String[] content) : ReturnCode

Envoie un message à l'agent d'id iAgent, avec le sujet du message et le contenu.

- **reply**(WarMessage warMessage, String message, String[] content) : ReturnCode

Envoie un message de réponse au message passé en argument.

- **broadcastMessage**(WarAgentType agentType, String message, String[] content) : ReturnCode

Envoie un message à tous les agents (alliés bien sûr) de type WarAgentType.

- **broadcastMessage**(String groupName, String roleName, String message, String[] content) : ReturnCode

Envoie un message à tous les agents alliés qui jouent le rôle roleName dans le groupe groupName.

- **broadcastMessage**(String message, String[] content) (int idAgent, String message, String[] content) : ReturnCode

Envoie un message à tous les agents de l'équipe.

### 4) Primitives propres au Rocket-launcher et à l'Explorer

- **getHeading ()** : Integer

Retourne l'angle actuel de l'agent.

- **isBlocked ()** : Boolean

Retourne vrai si l'agent est bloqué contre un bord, faux sinon.

- **setAgentToGive** (int ID)

Méthode appelée juste avant l'action « give ». Permettra de donner une ressource à l'agent ayant l'identifiant ID.

- **setHeading (double angle)**

Permet de changer de direction en modifiant la trajectoire.

- **setRandomHeading ()**

Change aléatoirement la trajectoire de l'agent.

- **setRandomHeading** (int range)

Change aléatoirement la trajectoire de l'agent dans une étendue limitée.

## 5) Primitives propres au Rocket-launcher (en plus des précédents)

- **isReloaded ()** : Boolean

Retourne vrai si l'agent a déjà « rechargé », faux sinon.

- **isReloading()** : Boolean

Retourne vrai si l'agent est en train de « recharger », faux sinon.

- **setAngleTurret (int angle)**

Placer la tourelle dans un certain angle pour tirer par la suite.

## 6) Primitives propres à la base

- **setNextAgentToCreate** (WarAgentType agent)

Méthode appelée juste avant l'action « create ». Permet d'indiquer quel type d'agent sera créé. Ex:

```
getBrain().setNextAgentToCreate(Const.createExplorer);
```

sera appliquée avant de lancer l'action WarBase.ACTION\_CREATE;

- **getPerceptsAllies** () ArrayList<WarPercept>

Retourne la liste des percepts correspondant à des alliés.

- **getPerceptsEnemies** () ArrayList<WarPercept>

Retourne la liste des percepts correspondant à des ennemis.

## VI. Primitives liées aux percepts

L'ensemble de ces primitives s'appliquent aux objets de types WarPercept, donc aux objets obtenus grâce à la primitive getPercepts(), citée plus haut.

- **getAngle**() : int

Retourne l'angle où se trouve l'unité perçue, par rapport à notre propre direction.

- **getDistance**() : int

Retourne la distance qu'il y a entre l'unité perçue et nous-même.

- **getHealth**() : int

Retourne l'énergie actuelle de l'unité perçue.

- **getId**() : int

Retourne l'identifiant de l'unité perçue.

- **getTeamName**() : String

Retourne le nom d'équipe de l'unité perçue, ou une chaîne vide si l'agent est de type « WarFood ».

- **getType**() : WarAgentType

Retourne le type de l'unité perçue.

## VI. Primitives liées à des chaîne de caractères de debug

- **getDebugString**() : String

Retourne la chaîne de caractère qui est affichée par l'agent.

- **setDebugString**(String)

Affecte la chaîne à afficher .

- **getDebugStringColor**() : Color



Retourne la couleur de la chaîne de caractère affichée.

- **setDebugStringColor**(Color)

Affecte une couleur à la chaîne de caractère affichée

## VII. Primitives liées aux messages

L'ensemble de ces primitives s'appliquent aux objets de types `WarMessageFinal`, donc aux objets obtenus grâce à la primitive `getMessage()`, citée plus haut.

- **getAngle()** : double

Retourne l'angle où se trouve l'agent nous ayant envoyé un message, par rapport à notre propre direction.

- **getContent()** : String[]

Retourne, s'il y en a, les informations complémentaires envoyées en tant que troisième paramètre de la primitive **sendMessage()** ou **broadcastMessage()**.

- **getDistance()** : double

Retourne la distance qu'il y a entre l'agent receveur et l'émetteur.

- **getMessage()** : String

Retourne le libellé du message (le sujet), contenu dans le deuxième paramètre de la primitive `sendMessage()` ou `broadcastMessage()`.

- **getSenderID()** : Integer

Retourne l'identifiant de l'émetteur.

- **getSenderTeamName()** : String

Retourne le nom de l'équipe qui a envoyé le message.

- **getSenderType()** : String

Retourne le type de l'unité qui a envoyé le message.

## VIII. Définition de comportement

Plusieurs équipes sont fournies pour que vous puissiez les confronter les unes aux autres, pour voir les différents comportements. Cela va vous permet-

tre de pouvoir vous mesurer localement à des agents basiques avant de pouvoir vous mettre en situation de compétition avec les autres étudiants. Vous devez donc vous occuper uniquement de la programmation des brains de vos agents. Pour définir par vous-même un comportement pour chacun des trois types d'agents, vous allez devoir définir 3 classes dans un nouveau package que vous créerez, chacune correspondant à l'IA d'une unité. Le nom n'a aucune importance, mais ces 3 classes doivent hériter de WarBrain, vous permettant d'avoir accès à l'ensemble des primitives citées plus haut.

Voici un exemple, pour le brain d'un Explorer:

```
public class WarExplorerBrainController extends WarExplorerAbstractBrainController {

    public WarExplorerBrainController() {
        super();
    }

    @Override
    public String action() {
        // Ecrire le comportement ici

        if (getBrain().isBlocked())
            getBrain().setRandomHeading();
        return WarExplorer.ACTION_MOVE;
    }
}
```

C'est dans la méthode action() que l'on place le comportement de l'agent, avec bien entendu la possibilité de coder des fonctions annexes. La String retournée par action() correspond à l'action que l'agent effectue pour ce tour.

## VII. Installation

Pour installer Warbot3, créez un projet Eclipse, en faisant 'new Java Project' et donnez comme directory, le dossier où se trouve Warbot.

Pour lancer Warbot depuis Eclipse, clic droit sur le fichier 'Warbot3.launch' qui se trouve à la racine du dossier Warbot et faire 'run as...' et sélectionnez Warbot3

## IX. Création d'un JAR contenant le code de votre équipe

**Note:** La création d'un Jar n'est nécessaire que lorsque vous désirez donner votre équipe à quelqu'un pour faire une compétition par exemple. Cela évite d'avoir à recompiler les sources.

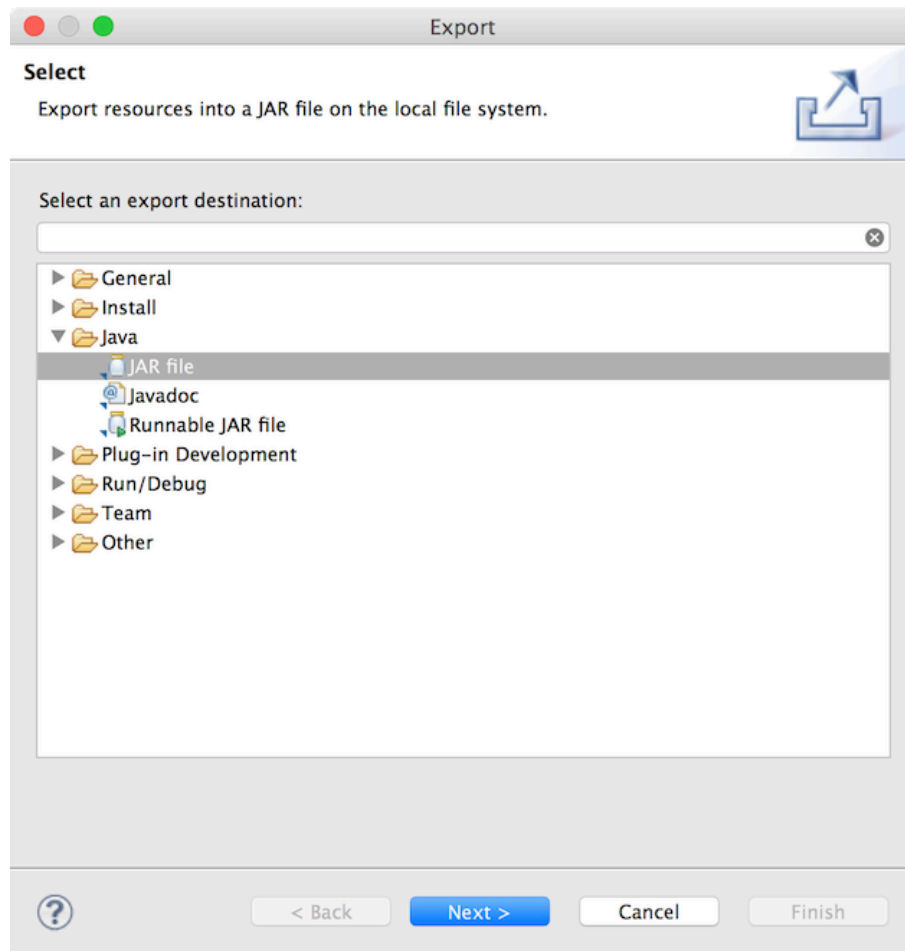
WARBOT III, à son démarrage, scanne l'ensemble des JARs présents dans le dossier 'teams'. Il faut donc le générer, mais il ne contiendra pas que les classes constituant vos brains. Une image représentant votre logo d'équipe devra être ajoutée, le format n'étant pas important (png, jpeg, ...).

Enfin, Il faudra un fichier structurant l'ensemble des précédents, sous forme XML et nommé « config.xml ». La syntaxe doit obligatoirement respecter celle qui suit :

```
<Team>
  <Name>myTeam</Name>
  <IconPath>myTeam.png</IconPath>
  <SoundPath>myTeam.wav</SoundPath>
  <Description>
    Ecrire une description ici !
  </Description>
  <AgentsBrainControllers>
    <WarExplorer>WarExplorerBrainController.class</WarExplorer>
  <WarRocketLauncher>WarRocketLauncherBrainController.class</WarRocketLauncher>
    <WarBase>WarBaseBrainController.class</WarBase>
    <WarKamikaze>WarKamikazeBrainController.class</WarKamikaze>
    <WarEngineer>WarEngineerBrainController.class</WarEngineer>
    <WarTurret>WarTurretBrainController.class</WarTurret>
  </AgentsBrainControllers>
</Team>
```

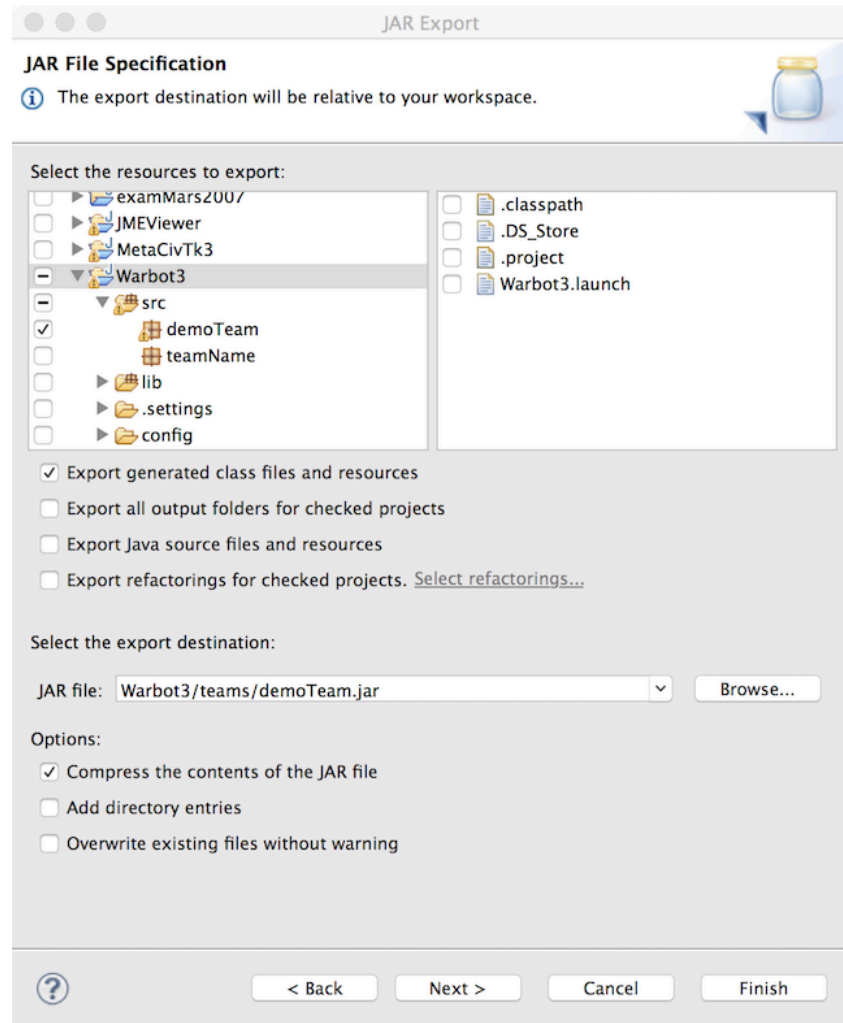
Une fois ces fichiers complétés, il vous suffit de générer le JAR à l'aide d'un IDE, puis de le placer dans le dossier jar présent à la racine du dossier Tk2. De cette façon, vous verrez votre équipe apparaître lors du lancement de WAR-BOT II. Une fois que le JAR est créé, il ne sera plus utile de le régénérer à chaque fois que vous effectuerez une modification dans votre code, cela sera détecté automatiquement.

1) Pour générer un JAR sous Eclipse. Clic droit sur le package de votre équipe, puis export / Java / JAR File. Vous obtiendrez la fenêtre suivante:

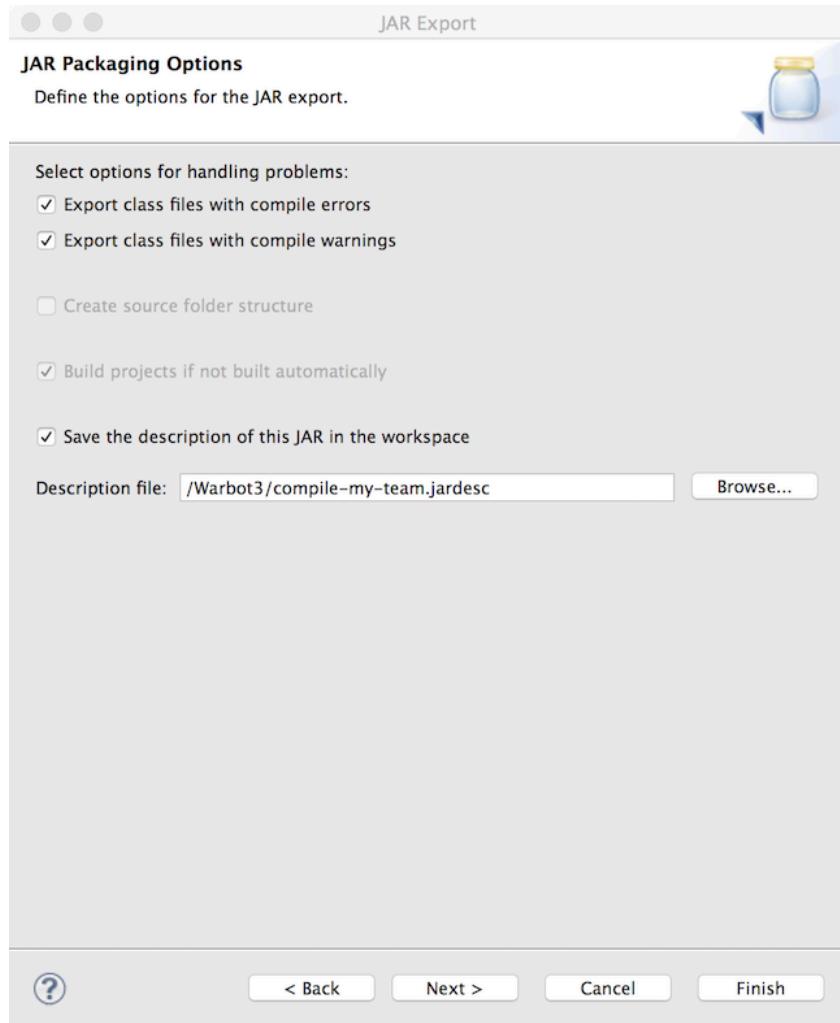


2) Continuez en cliquant sur 'next'. Vous avez alors une fenêtre comme le montre la figure suivante. Dans 'browse' sélectionnez le dossier <warbot>/teams

qui contient toutes les équipes, ainsi que le nom de votre Jar. Le chemin global aura la forme <warbot>/teams/<nom de votre équipe>.jar



3) Cliquez sur 'next'. Et vous obtiendrez l'image suivante. Il suffit alors de valider l'item: 'save the description of this Jar...' et de choisir avec 'browse' un lieu. Il est préférable de le situer à la racine de Warbot. Puis faites 'finish'. Votre Jar devrait se trouver dans la racine dans vos Teams.



## VIII.Crédits

- Responsable du projet: Jacques Ferber
- La version originale de Warbot a été écrite en 2002 par Jacques Ferber et Fabien Michel, au dessus de MadKit.
- La version Warlogo a été écrite par Loïs Vanhée et Fabien Hervouet en NetLogo.
- La version 2.x de Warbot a été réécrite à partir de Warlogo et la première version de Warbot par: Jessy Bonnotte , Pierre Burc, Olivier Duploux Mathieu Polizzi. Cette version a été écrite au dessus de TurtleKit 2.0 dont l'auteur est Fabien Michel.
- La version 3.x de Warbot a été développée par: Félix Vonthron, Olivier Perrier, Bastien Schummer, Valentin Cazaudebat. Cette version a été écrite au dessus de TurtleKit 3.0 dont l'auteur est Fabien Michel.