



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

Année 2014-2016

**Version 3.3.2.a du Manuel
Correspond à la version 3.3 (et suivantes) de Warbot**

du 30 novembre 2016

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 plateforme 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 fondamentaux dont il s'agit de programmer le comportement : **bases**, **explorers** et **rocket-launchers**.

1) Bases

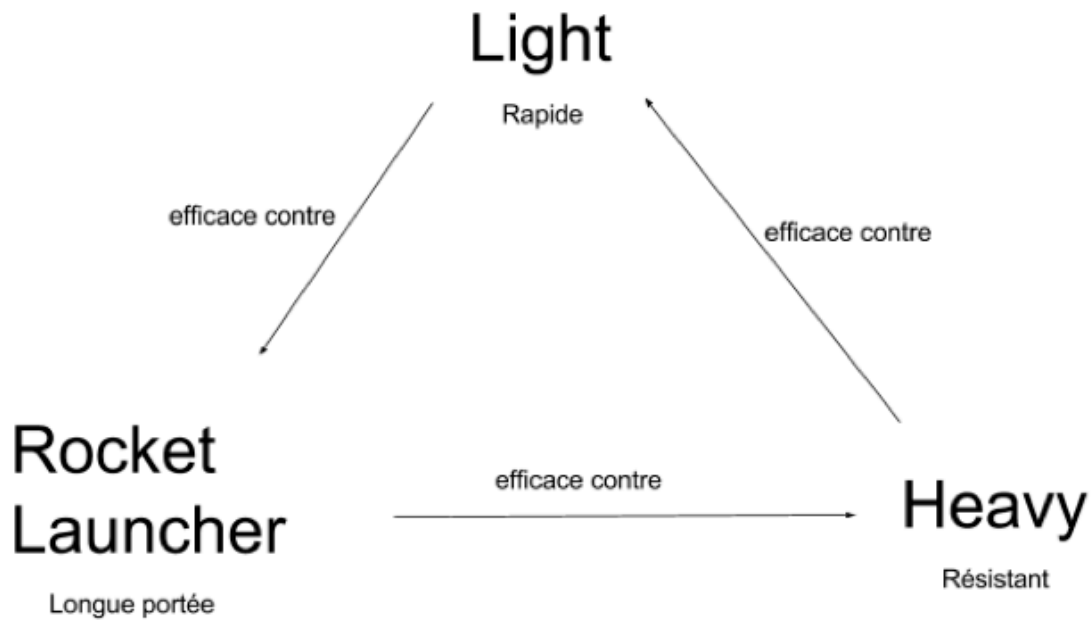
Les **bases** représentent le quartier général de votre équipe. Ce sont des agents ne pouvant pas se déplacer physiquement dans le monde. En revanche elles peuvent construire des unités de type exploreurs et rocket-launchers.

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) Les chars et lanceurs de missiles

Il existe (depuis la version 3.3) trois types d'unités capable de produire des dommages à d'autres unités en lançant des obus ou des missiles. Ce sont les chars légers (WarLight), les chars lourds (WarHeavy) et lanceurs de missiles (WarRocketLauncher). Ces trois types d'unités sont comme feuille-caillou-ciseau, chacune étant efficace contre une autre, mais pouvant être défaite par une troisième.



- Les **WarHeavy** sont lents mais résistants, ils sont efficaces contre les WarLight car ne subissent que peu de dégâts de leur part grâce à leur armure, et ont une cadence de feu suffisante pour les détruire rapidement. Cependant, leur faible vitesse les rendent vulnérables aux WarRocketLauncher.
- Les **WarLights** sont rapides mais fragiles, ils sont efficaces contre les WarRocketLauncher car peuvent facilement éviter les missiles ennemis grâce à leur vitesse, et arriver au corps à corps contre des WarRocketLauncher où ces derniers sont inefficaces. Cependant, ils doivent rester à distance des WarHeavy ennemis contre lesquels ils sont inefficaces. Les projectiles des WarLight et des WarHeavy sont différents des missiles des WarRocketLauncher : ils n'infligent des dégâts qu'à la première unité rencontrée dans sa course.
- Les **WarRocketLauncher** ont une grande portée mais tirent lentement, ils sont efficaces contre les WarHeavy car peuvent les atteindre de loin et les WarHeavy sont trop lents pour éviter leurs missiles. D'un autre côté, ils ne peuvent tirer au contact ce qui les rends particulièrement faibles face aux WarLight. De plus, désormais les projectiles des WarRocketLauncher ne sont plus détruits en entrant en collision avec un autre agent, ils continuent

leur course jusqu'à une distance précise, définie avant le tir, similaire à un mortier.

3) Autres unités

- **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 des unités 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.

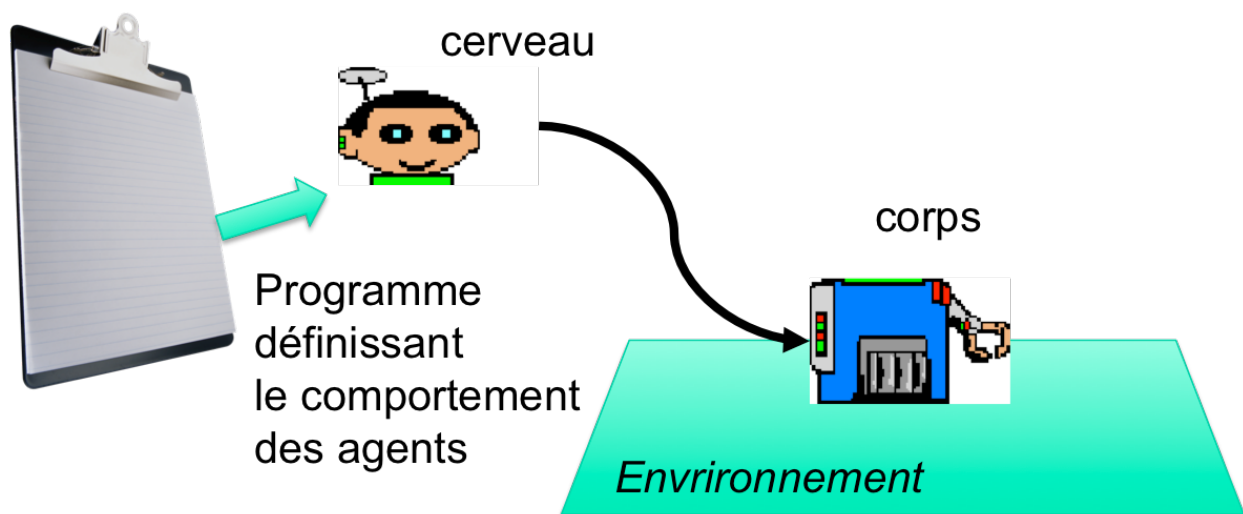


Figure 1. les agents ont un corps et un cerveau. Les joueurs doivent programmer le cerveau des agents.

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 exprimée 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 réduites. 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 de lancement

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

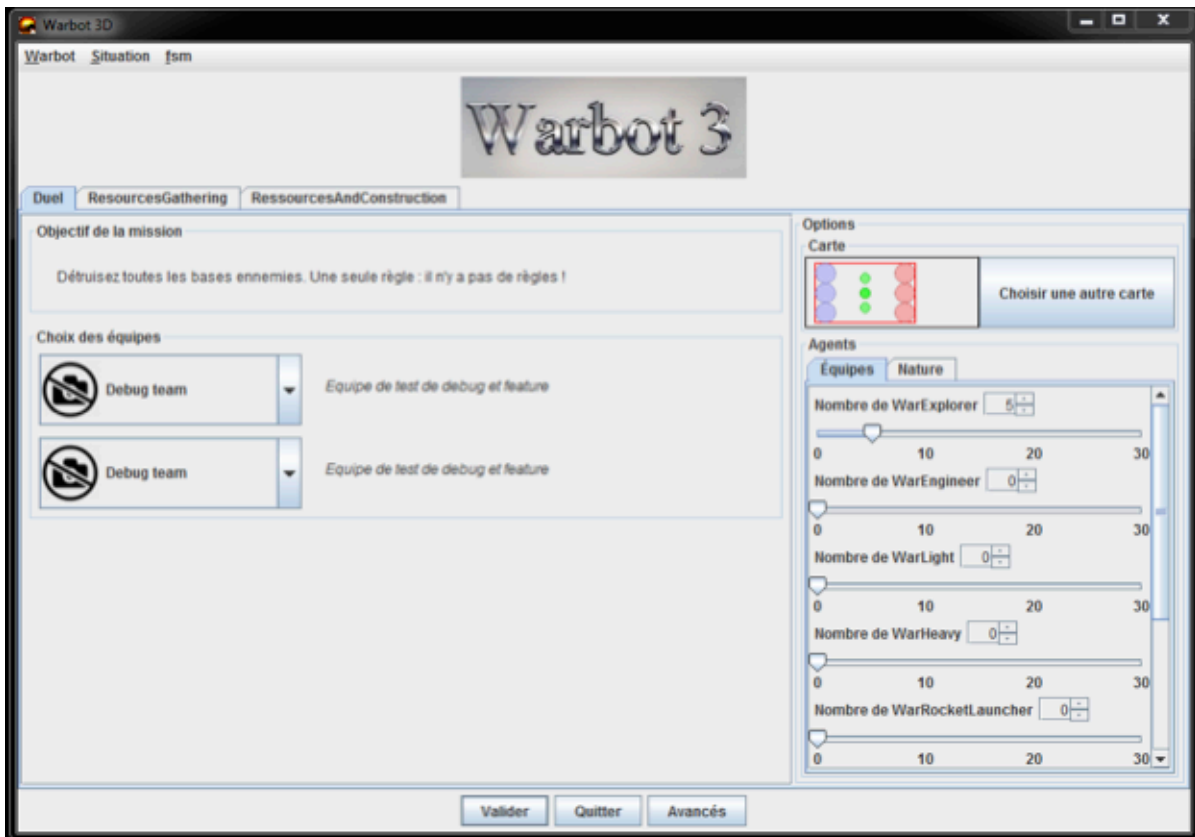


Figure 2: Capture d'écran de l'interface de lancement de Warbot

La fenêtre est divisée en plusieurs parties :

une barre de menu permettant de charger une situation ou de lancer l'éditeur de FSM (en cours de développement);

- le panneau central est composé de trois onglets correspondant aux modes possibles : duel (un combat entre deux équipes), récupération de ressources ou constructions. 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.

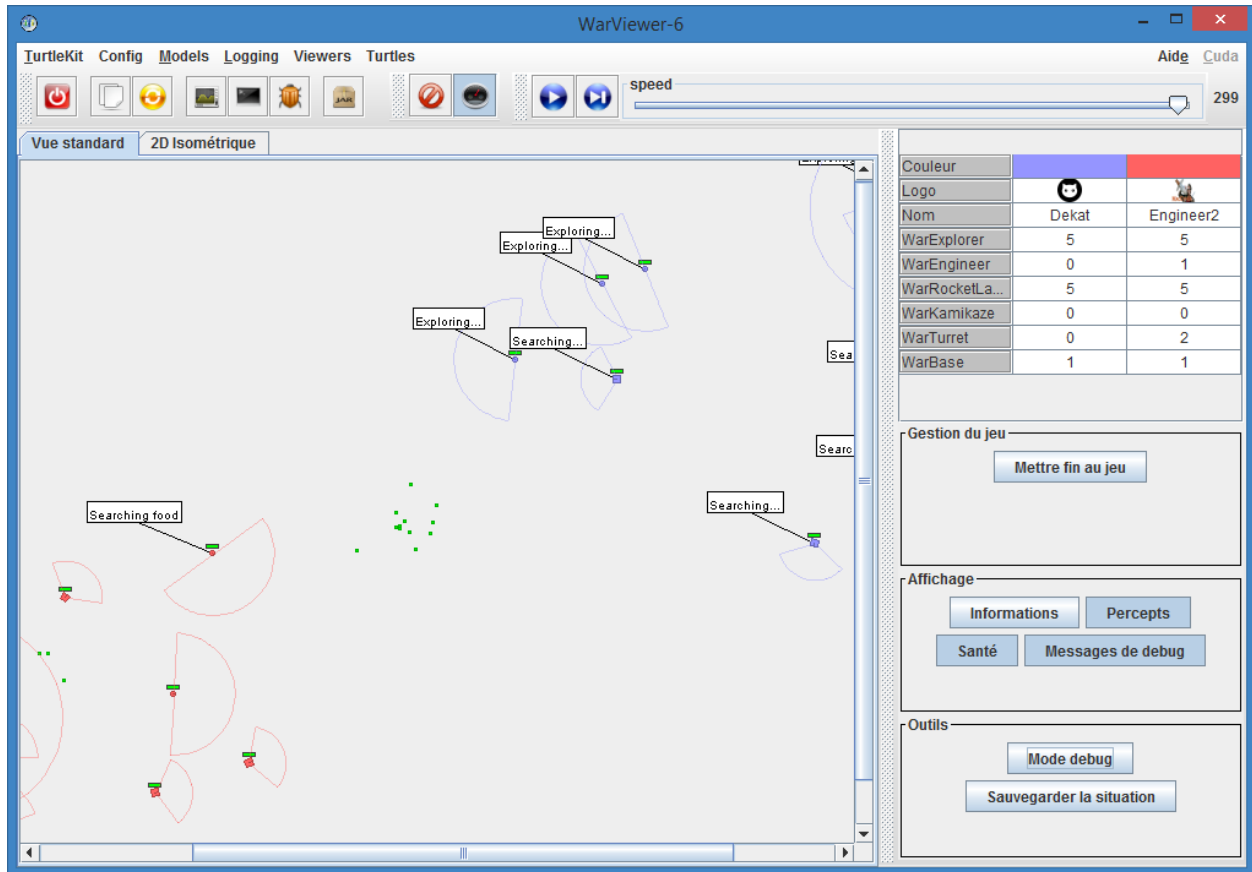


Figure 3: Capture d'écran de l'interface

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

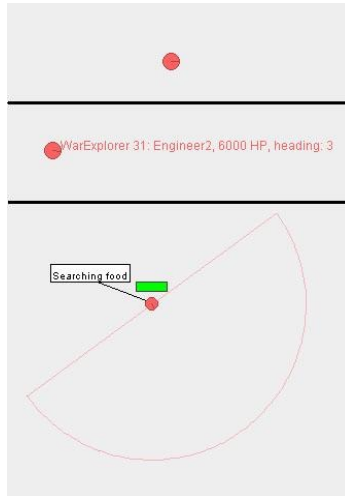
Tableau d'information des équipes

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 :

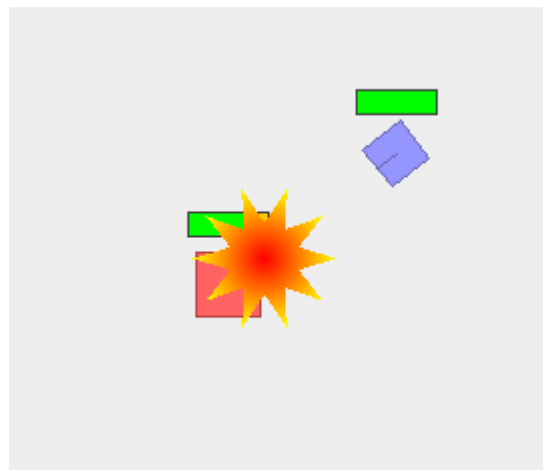
- un texte d'information général : type d'unité, numéro d'identification de l'agent, nom de son équipe, santé, angle ;
- affichage des percepts : les champs de vision des agents sont alors affichés ;
- les barres de santé ;
- les messages de debug : bulles reliées à chaque agent, qui permettent à l'utilisateur 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

V. Modes de jeux

Afin d'avoir plusieurs stratégies possibles, il est possible de jouer dans plusieurs modes. Chaque mode décrit un certain type de scénario et un objectif à atteindre pour terminer le jeu.

1. Récolte de ressources

Dans ce mode de jeu, une équipe composée d'une base et de plusieurs explorateurs doivent récolter un certain nombre de ressources sur la carte, mais la carte est défendue par des tourelles ennemies, placées hors de portée de la base. Il n'y a qu'une seule ressource à la fois sur la carte, et quand une est ramassée, une nouvelle est placée aléatoirement sur la carte, hors de portée des tourelles. La construction de nouvelles unités est interdite. La partie prend fin quand les explorateurs ont ramassés suffisamment de ressources, où quand les tourelles ont détruits tous les explorateurs.

2. Duel

C'est le mode de jeu standard dans lequel deux équipes s'affrontent et cherchent à détruire les bases ennemies. Chaque équipe est composée du même nombre d'agents. Au milieu se trouve des zones où les ressources apparaissent au cours de la partie. La partie est terminée quand toutes les bases d'une équipe sont détruites.

2. Ressources et construction

Dans le mode de jeu « ressources et constructions », le placement initial est le même que dans le mode de jeu duel. Néanmoins, aucune unité offensive n'est autorisée. Le but est de récolter un maximum de ressources et de créer de nouvelles unités, bases, murs, etc.. L'équipe gagnante est la première à posséder suffisamment de constructions

VI. Liste des actions

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 des 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)`.
- **Take** : Permet de ramasser une ressource présente sur le terrain. Si le sac est plein, rien ne se passe.
- **Move** : fait avancer l'unité (sauf pour les bases et les tourelles bien entendu).

3) Actions de la base et de l'ingénieur

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

4) Actions des unités de combat

- **Fire** : Envoyer une roquette en direction d'un agent ennemi. Le projectile est envoyé dans la direction du véhicule (heading).
- **Reload** : Une fois enclenché, le RocketLauncher doit attendre un laps de temps de 50 ticks avant de pouvoir tirer.

VII. 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...

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.

- **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.

- **getMaxHealth ()** : int

Retourne l'énergie maximale de cet 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 groupes et roles

- **requestRole** (String group, String role)

Demande de jouer un rôle dans un groupe. Quand le groupe n'existe pas, il est créé. Retourne un code d'erreur en fonction du résultat de l'acceptation. Par défaut c'est un succès.

- **leaveRole** (String group, String role) :

Arrête de jouer un rôle dans un groupe.

- **leaveRole** (String group) :

Quitte un groupe. S'il n'y a plus d'agents dans le groupe, le groupe est supprimé.

- **myGroups ()** : ArrayList<String>

Retourne la liste des groupes dans lequel se trouve l'agent.

- **myRoles** (String group) : ArrayList<String>

Retourne la liste des rôles dans lequel se trouve l'agent dans le groupe group.

- **getNumberOfAgentsInRole** (String group, String role) : int

Retourne le nombre d'agents jouant le rôle role dans le groupe group.

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) : Return-Code :

Envoie un message de réponse au message passé en argument.

- **broadcastMessageToAll**(String message, String ... content) (int idAgent, String message, String ... content) : ReturnCode

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

- **broadcastMessageToAgentType**(WarAgentType agentType, String message, String ... content) : ReturnCode

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

- **broadcastMessageToGroup** (groupName, message, content)

Envoie un message aux agents du groupe groupName.

- **broadcastMessage** (groupName, roleName, message, content)

Envoie un message aux agents du groupe groupName, ayant pour rôle roleName.

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.

6) Primitives de création (base et ingénieur)

- **setNextAgentToCreate** (WarAgentType type)

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

```
getBrain().setNextAgentToCreate(WarAgentType.WarExplorer);
```

sera appliquée avant de lancer l'action WarBase.[ACTION_CREATE](#);

- **getNextAgentToCreate**() :

Retourne le type de l'agent à créer prochainement

- **isAbleToCreate**(WarAgentType type) :

Retourne vrai si et seulement si l'agent peut créer cette unité

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

Retourne la liste des percepts correspondant à des alliés.

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

Retourne la liste des percepts correspondant à des ennemis.

VIII. 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 indiquant la direction de l'objet perçu.

- **getDistance()** : int

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

- **getHealth()** : int

Retourne le niveau de vie actuel de l'unité perçue.

- **getMaxHealth()** : int

Retourne le niveau de vie maximum 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()** : String

Retourne le type de l'unité perçue.

- **getHeading()** : String

Retourne la direction vers laquelle se dirige l'unité perçue

IX. Primitives liées à des chaînes 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

X. Primitives liées aux projectiles

Les projectiles sont associés à des unités de combat. Les chars légers (Light) tirent des balles, les chars lourds (Heavy) tirent des obus et les lanceurs de missiles (RocketLauncher) des missiles.

Types de projectiles

WarBullet (tiré par des WarLight)

WarShell (tirés par des WarHeavy)

WarRocket (tirés par des WarRocketLauncher)

Les balles et les obus tirent en ligne droite et occasionnent des dégâts à la première unité trouvée.

Les missiles passent au dessus des unités et explosent lorsqu'ils arrivent au bout de leur destination (définie par **setTargetDistance(double target-Distance)** dans un RocketLauncher)

Ces projectiles sont caractérisés par les constantes suivantes :

SPEED : vitesse des projectiles (en nombre de tics)

AUTONOMY : durée du projectile

RANGE : portée maximale du projectile (=SPEED*AUTONOMY)

Et pour les Rockets uniquement :

DAMAGE : dégâts occasionnés par les missiles

EXPLOSION_RADIUS : étendue de l'explosion du missile.

Ex : WarBullet.RANGE donne la portée maximale des balles.

XI. Primitives liées aux messages

L'ensemble de ces primitives s'appliquent aux messages 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.

XII. Définition de comportement

Vous pouvez utiliser le langage Python. On peut aussi programmer en Java (Il existe aussi une version JavaScript, mais elle n'est pas à jour version 3.2.4).

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

```
def actionWarRocketLauncher():  
    percepts = getPercepts();  
  
    for percept in percepts:  
        if (percept.getType().equals(WarAgentType.WarRocketLauncher)):  
            if (isEnemy(percept)):  
                setDebugString("Mode hunter")  
                setHeading(percept.getAngle())  
  
                if (isReloaded()):  
                    return fire()  
                else :  
                    return reloadWeapon()  
            else:  
                setDebugString("No cible")  
  
        elif (percept.getType().equals(WarAgentType.WarBase)):  
            if (isEnemy(percept)):  
                setDebugString("Mode hunter")  
                setHeading(percept.getAngle())  
  
                if (isReloaded()):  
                    return fire()  
                else :  
                    return reloadWeapon()  
            else:  
                setDebugString("No cible")  
        else:  
            setDebugString("No cible")  
  
    if (len(percepts) == 0):  
        setDebugString("No cible")  
  
    if(isBlocked()):  
        RandomHeading()  
  
    return move();
```

Cela va vous permettre 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.

Voici un exemple de définition d'un cerveau d'un lanceur de missile (le nom de la fonction est important). On trouvera quelques exemples de tels comportements dans les dossiers démo fournis à cet effet.

IX. Création d'une équipe en Python

WARBOT III, à son démarrage, scanne l'ensemble des dossiers situés dans le dossier 'team'. Il suffit de créer donc un nouveau dossier et d'y placer les ressources associées.

Les fichiers correspondent au Brain des robots. Voilà leur nom par défaut (on peut changer leur nom dans le fichier de configuration, voir ci-dessous):

- **WarBase.py**: pour décrire le cerveau d'une Base
- **WarRocketLauncher.py**: pour décrire le cerveau d'un lanceur de Rocket
- **WarExplorer.py**: pour décrire le cerveau d'un explorateur
- **WarEngineer.py** : pour décrire le cerveau d'un ingénieur
- **WarTurret.py**: pour décrire le cerveau d'une tourelle
- **WarKamikaze.py**: pour décrire le cerveau d'un kamikaze

Un fichier de configuration, nommé 'config.yml' doit se trouver dans ce dossier. il comprend les paramètres de votre équipe. Voici celui de l'exemple par défaut:

```
Name: PYTHON_FunctionLv0
IconPath: dekat.png
SoundPath: dekat.wav
Description: |
    Equipe de super Python avec fonction niveau 0
BrainsPackage: inGameTeams.python_teamFunction_level_0
ScriptImplementation: python
AgentsBrainClasses:
    WarBase: WarBase.py
    WarExplorer: WarExplorer.py
    WarRocketLauncher: WarRocketLauncher.py
    WarKamikaze: WarKamikaze.py
```

```
WarEngineer: WarEngineer.py
WarTurret: WarTurret.py
```

Pour créer des attributs et donc mémoriser des éléments, il est possible de créer un dictionnaire de données locales à l'agent et d'y accéder depuis la fonction principale:

```
dico = {}
dico['n'] = 0

def actionWarExplorer():
    dico['n'] += 1 #incrémente l'attribut 'n' de l'agent
```

XI. Installation

Pour lancer Warbot3, aller dans le dossier de Warbot et lancer le jar avec la commande:

```
java -jar warbot3.x.jar
```

(ou double cliquez sur le jar, mais vous n'aurez pas nécessairement accès aux messages dans la console)

XII. Création d'une équipe en Java

Il est possible de programmer des équipes en Java ou en Python. Pour programmer une équipe en Java, il suffit de créer l'ensemble des fichiers Java correspondant aux types de robots et de créer un Jar, c'est à dire une archive Java contenant les fichiers Java compilés ainsi qu'un fichier de description de votre équipe.

Une description du «cerveau» d'un agent. Chaque cerveau doit être créer dans un fichier Java à part. On pourra utiliser l'équipe MyTeam comme exemple pour démarrer:

```
package myteam;

import edu.warbot.agents.enums.WarAgentType;
import edu.warbot.agents.percepts.WarAgentPercept;
import edu.warbot.brains.brains.WarRocketLauncherBrain;
```

```

public abstract class WarRocketLauncherBrainController
    extends WarRocketLauncherBrain {

    public WarRocketLauncherBrainController() {
        super();
    }

    @Override
    public String action() {
        for (WarAgentPercept wp : getPerceptsEnemies()) {
            if (wp.getType().equals(WarAgentType.WarBase)) {
                setHeading(wp.getAngle());
                if (isReloaded())
                    return ACTION_FIRE;
                else if (isReloading())
                    return ACTION_IDLE;
                else
                    return ACTION_RELOAD;
            }
        }

        if (isBlocked())
            setRandomHeading();
        return ACTION_MOVE;
    }
}

```

Le fichier de description d'une équipe

Le fichier de description est utilisé par Warbot pour savoir quelle équipe est présente, quels en sont les classes décrites, etc.

Il s'agit d'un fichier noté 'config.yml' dont la syntaxe doit obligatoirement respecter celle qui suit :

```

Name: MyTeam
IconPath: MyTeam.png
Description: |
    Descriptions de mon équipe
BrainsPackage: myteam
AgentsBrainClasses:
    WarBase: WarBaseBrainController
    WarExplorer: WarExplorerBrainController
    WarRocketLauncher: WarRocketLauncherBrainController
    WarKamikaze: WarKamikazeBrainController

```

WarEngineer: WarEngineerBrainController
WarTurret: WarTurretBrainController

Tout ce qui est en bleu est imposé par le format, ce qui est écrit en noir peut être modifié, et dépend de votre équipe.

- **Name:** le nom de votre équipe qui sera affiché dans la liste des équipes au démarrage de Warbot.
- **IconPath:** le nom d'une image qui sera affichée dans la liste des équipes.
- **BrainsPackage:** le nom du package Java de votre équipe
- **AgentsBrainClasses:** le nom des classes des «cerveaux» de vos agents, pour chaque type d'agent.

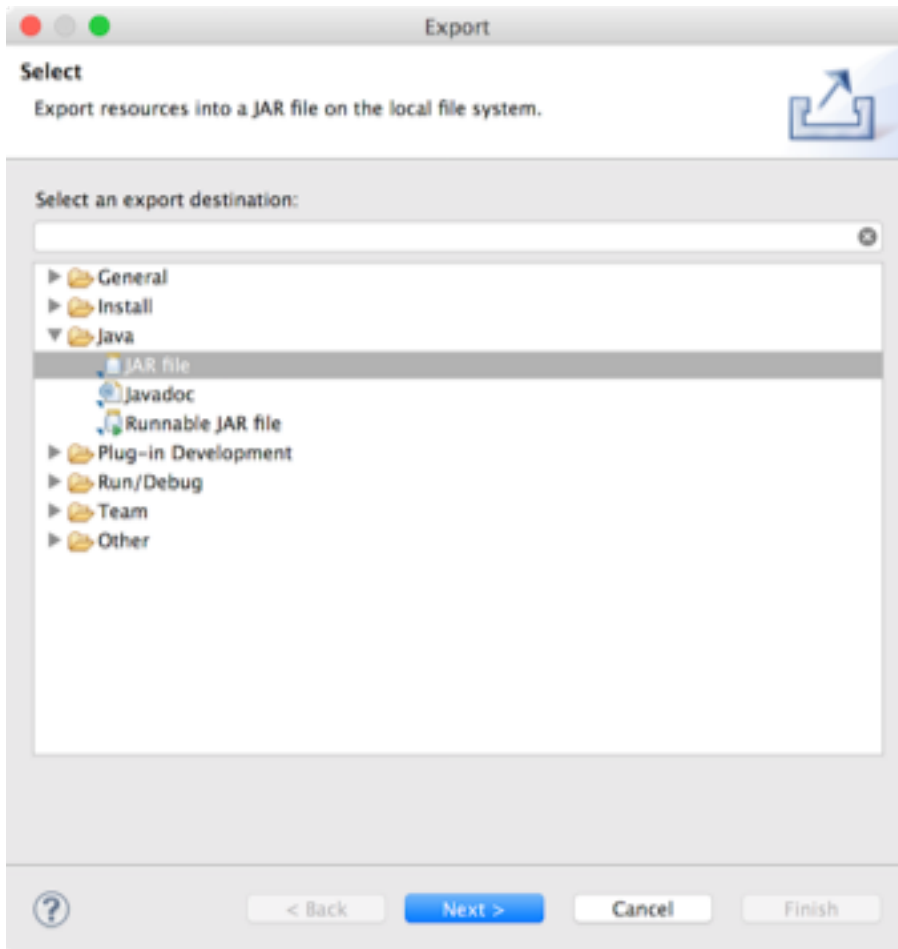
Note: *n'appellez surtout pas votre équipe 'myTeam', car 2 équipes, pour entrer en compétition doivent avoir des noms différents. Prenez le nom de votre équipe.*

Création du Jar sous Eclipse

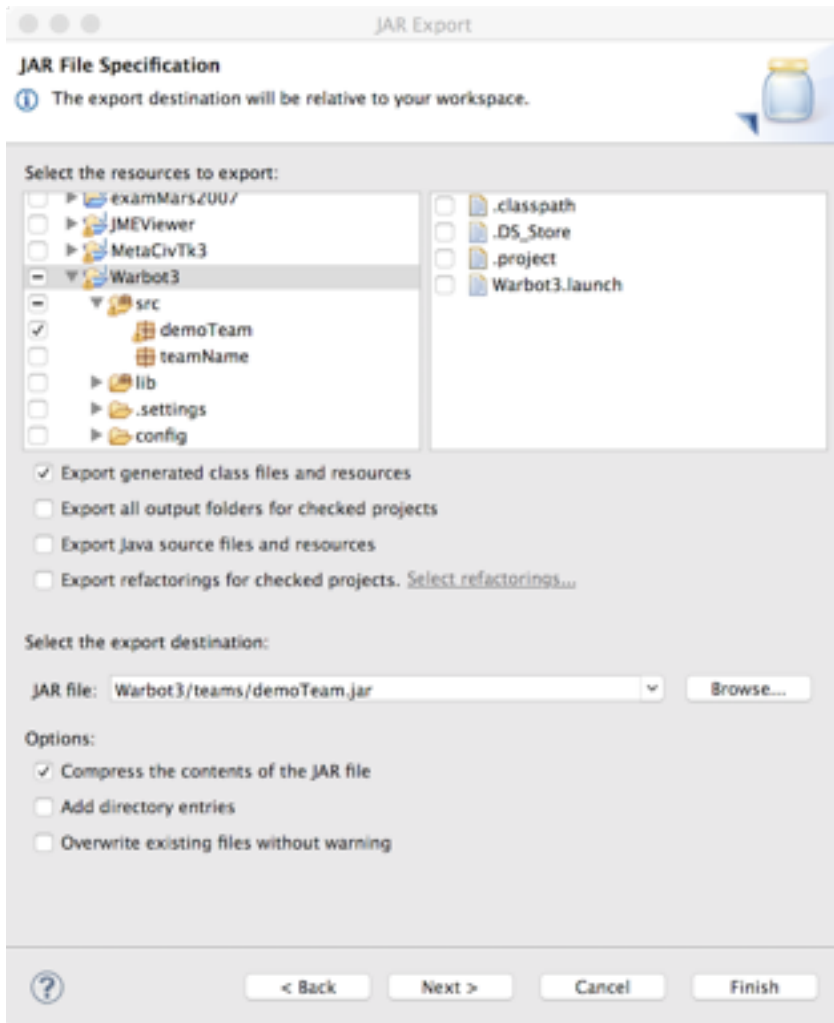
WARBOT III, à son démarrage, scanne l'ensemble des JARs présents dans le dossier 'teams'. Il faut donc le générer, mais il ne contiendra pas que les classes constituant vos brains. Une image représentant votre logo d'équipe devra être ajoutée, le format n'étant pas important (png, jpeg, ...).

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 'teams' de Warbot. présent à la racine du . De cette façon, vous verrez votre équipe apparaître lors du lancement de WARBOT. 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.

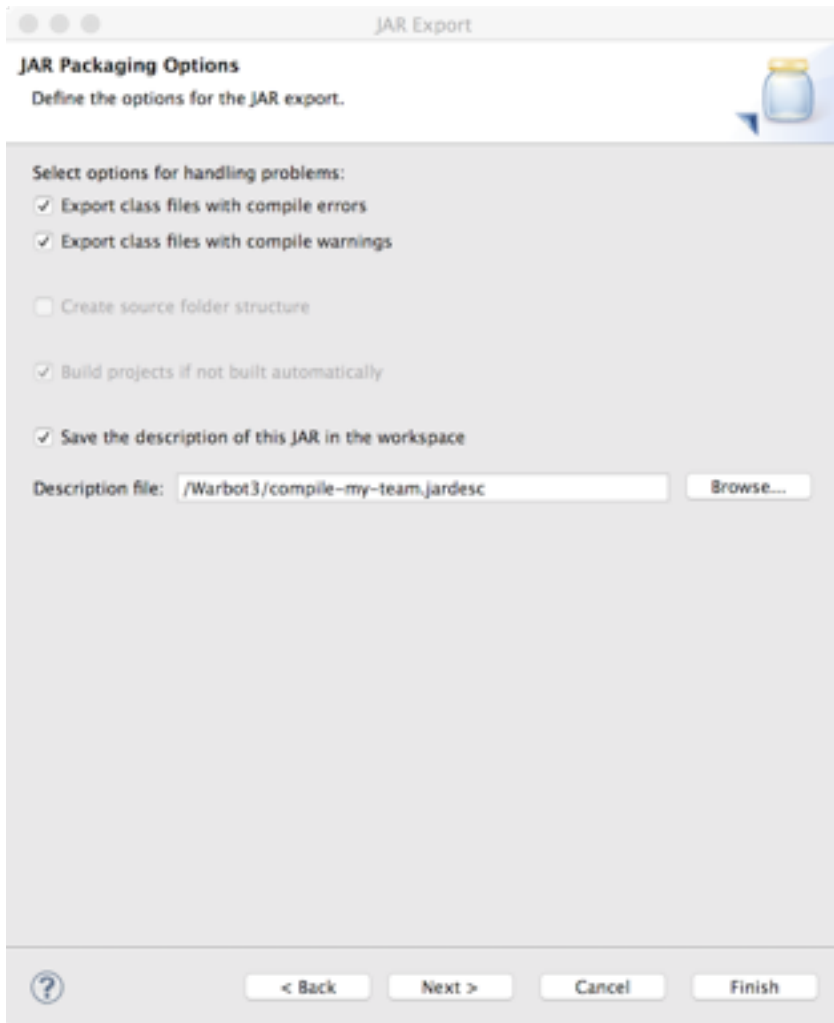
1) Pour générer un JAR sous Eclipse. Clic droit sur le package de votre équipe, puis export / Java / JAR File. Vous obtiendrez la fenêtre suivante:



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



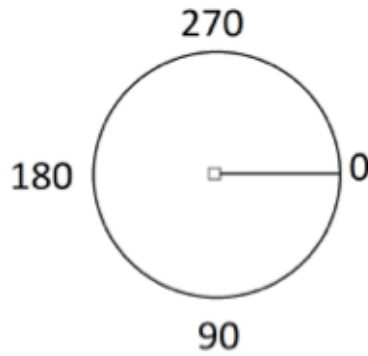
1) Cliquez sur 'next'. Et vous obtiendrez l'image suivante. Il suffit alors de valider l'item: 'save the description of this Jar...' et de choisir avec 'browse' le dossier 'teams' où doivent se trouver les jars. Puis faites 'finish'. Votre Jar devrait se trouver dans le dossier 'teams'.



XIII. Quelques éléments de compréhension du jeu

1. Mouvements

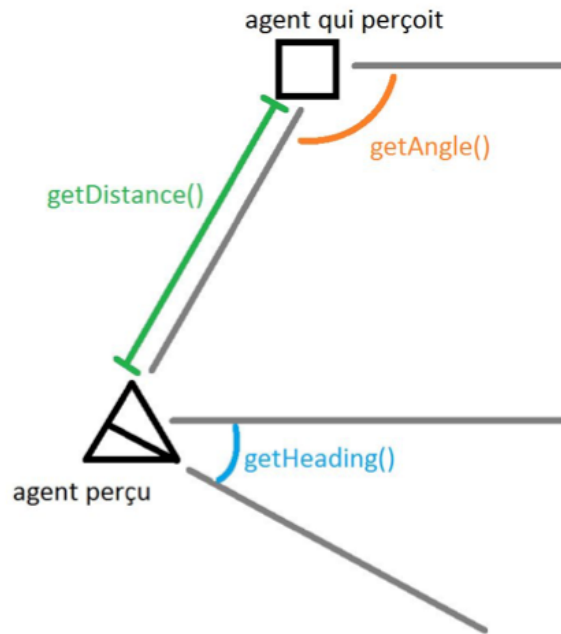
- **return move()** (action terminale) : Effectue une action de mouvement. Cette action termine le tour de l'agent. L'agent se déplace dans la direction du dernier `setDirection(int angle)`, d'une distance prédéfinie (ex : `WarExplorer.SPEED`), si son déplacement est restreint par un obstacle, il s'arrête devant.
- **setDirection(int angle)** : Tourne l'agent en fonction de la valeur `angle`, selon le repère donné ci-dessous.



- **WarExplorer.SPEED / WarLight.SPEED** / etc... retourne la vitesse d'un type d'agent, c'est-à-dire la distance parcourue à chaque tick où l'agent fait une action move().
- **isBlocked()** : retourne vrai si l'agent a rencontré un obstacle lors du tick précédent, faux sinon.
-

2. Perception

- **getPercepts()** : retourne l'ensemble des percepts dans le champ de vision de l'agent.
- **percept.getType()** : retourne le type d'agent perçu (WarFood, WarExplorer, WarBase, etc...).
- **isEnemy(percept)** : vrai si l'agent perçu est un ennemi, faux sinon.
- percept.getAngle() / percept.getDistance() / percept.getHeading()



3. Récupération de ressources

- **percept.getDistance()** WarFood.MAX_DISTANCE_TAKE
- **isBagFull()** / **isBagEmpty()**
- **return take()** : action

Récupère une ressource à proximité, dont la distance est inférieure à WarFood.MAX_DISTANCE_TAKE.

- **setIdNextAgentToGive(idBase)**
- **return give()**

Donne une ressource à l'agent indiqué par le précédent setIdNextAgentToGive(...), la distance entre les deux agents doit être inférieure à WarFood.MAX_DISTANCE_TAKE.

4. Construction

La base peut construire des unités (explorateur, ingénieur, light, heavy, et rocketLauncher). L'ingénieur peut construire des bâtiments (base, tourelle, mur).

- **setNextAgentToCreate(agentType)**

- **return create()** : Construit une nouvelle unité. Consomme de la vie.
- **return eat()** : Consomme une WarFood dans le sac de l'agent pour régénérer sa vie. `setNextAgentToGive(idAgent)`
- **return give()** : Donne une food contenue dans le sac à l'agent indiqué dans le dernier `setNextAgentToGive`. La distance entre les agents doit être inférieure à `MAX_DISTANCE_GIVE`.

5. Tir

- **return fire()** : effectue une action de tir. Cette action termine le tour de l'agent. L'agent tire dans la direction du dernier `setDirection(int angle)`. Seuls les agents offensifs (Light, Heavy, RocketLauncher) peuvent tirer. Pour plus d'informations sur les forces et faiblesses de ces agents, lire la partie associée.
- **return reloadWeapon()** : effectue une action de rechargement. Cette action termine le tour de l'agent. Lors du rechargement, l'agent ne peut plus effectuer d'autres actions.
- **isReloaded() / isReloading()**
- **setTargetDistance(double targetDistance)** : les agents RocketLauncher peuvent, grâce à cette fonction, régler la distance de détonation de leur projectile. ATTENTION : les rockets passent au dessus des autres unités et explosent à la distance précisée.

XIV Les paramètres de base des unités

Chaque unité est décrite par un ensemble de paramètres qui sont modifiables globalement par le fichier `config/warbot_settings.yml`.

Pour accéder à ces paramètres il suffit d'écrire :

`<type unité>.<paramètre>`

Ex :

`WarBase.AngleOfView` retournera l'angle de vue d'un agent de type `WarBase`.

`WarExplorer.speed` retournera la vitesse d'avancement à chaque tick d'un `WarExplorer` (s'il fait un 'move' bien sûr).

Voici la liste de ces paramètres :

WarBase:

AngleOfView:
DistanceOfView:
Cost:
MaxHealth:
BagSize:
Armor:
Hitbox:

WarTurret:

AngleOfView:
DistanceOfView:
Cost:
MaxHealth:
BagSize:
TicksToReload:
Armor:
Hitbox:

Wall:

Cost:
MaxHealth:
Armor:
Hitbox:

Worker:

WarExplorer:
 AngleOfView:
 DistanceOfView:
 Cost:
 MaxHealth:
 BagSize:
 Speed:
 Armor:
 Hitbox:
WarEngineer:
 AngleOfView:
 DistanceOfView:
 Cost:
 MaxHealth:
 BagSize:
 Speed:
 MaxRepairsPerTick:
 Armor:
 Hitbox:

Soldier:

WarLight:
 AngleOfView:

```

    DistanceOfView: 20.0
    Cost: 250
    MaxHealth: 200
    BagSize: 4
    Speed: 1.8
    TicksToReload: 1
    Armor: 1
    Hitbox: { Shape: Square, SideLength: 5.0 }
WarHeavy:
    AngleOfView: 120.0
    DistanceOfView: 40.0
    Cost: 500
    MaxHealth: 800
    BagSize: 4
    Speed: 0.8
    TicksToReload: 5
    Armor: 20
    Hitbox: { Shape: Square, SideLength: 5.0 }
WarRocketLauncher:
    AngleOfView: 120.0
    DistanceOfView: 20.0
    Cost: 1000
    MaxHealth: 200
    BagSize: 4
    Speed: 1.2
    TicksToReload: 50
    Armor: 1
    Hitbox: { Shape: Square, SideLength: 3.0 }
WarKamikaze:
    AngleOfView: 150.0
    DistanceOfView: 20.0
    Cost: 3000
    MaxHealth: 3000
    BagSize: 4
    Speed: 1.0
    Armor: 0
    Hitbox: { Shape: Diamond, Radius: 2.0 }

```

Projectile:

```

WarRocket:
    Speed: 5.0
    ExplosionRadius: 10.0
    Autonomy: 30
    Damage: 200
    Hitbox: { Shape: Circle, Radius: 1.0 }
WarShell:

```



```
    Speed: 10.0
    ExplosionRadius: 1.0
    Autonomy: 5
    Damage: 50
    Hitbox: { Shape: Arrow, Radius: 1.0 }
WarBullet:
    Speed: 10.0
    ExplosionRadius: 0.5
    Autonomy: 3
    Damage: 20
    Hitbox: { Shape: Triangle, Radius: 0.5 }
WarBomb:
    Speed: 0.0
    ExplosionRadius: 40.0
    Autonomy: 0
    Damage: 2000
    Hitbox: { Shape: Circle, Radius: 1.0 }
WarDeathRocket:
    Speed: 3.0
    ExplosionRadius: 20.0
    Autonomy:
    Damage:
    Hitbox:
```

Resource:

```
WarFood:
    HealthGived:
    Hitbox:
```

XIV. 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 , Matthieu Polizzi. Cette version a été écrite au dessus de TurtleKit 2.0 dont l'ateur est Fabien Michel.

- La version 3.1 de Warbot a été développée par: Félix Vonthron, Olivier Perrier, Bastien Schummer, Valentin Cazaudebat de Février à Mai 2014. Cette version a été écrite au dessus de TurtleKit 3.0 dont l'auteur est Fabien Michel.
- La version 3.2 (standalone) a été développée et améliorée par Sébastien Beugnon et Jimmy Lopez, aidés de Quentin Philbert et Geoffrey Dumas de Février à Mai 2015.
- La version 3.3 a été développée par Thomas Bonnin et Pascal Wagner de Février à Mai 2016.