T5 : Laboratoire de simulation de systèmes multi-agents réactifs

A.W.E.

Abstract World Engine 1

Documentation Technique

Sommaire

Introduction

Abstract World Engine, comme son nom l'indique, est un moteur abstrait de simulation d'environnement tridimensionnel. La plupart des systèmes multi-agents réactifs sont spécifiquement conçus pour résoudre des problème bien précis (fourmis, simulateurs autoroutiers…). En général, les différents agents de la simulation sont codés en dur et l'utilisateur est limité à la modification d'une quantité de paramètres définis par le programmeur.

L'approche d'**AWE** est fondamentalement différente dans le sens où les agents ne sont pas qualifiés directement dans leur code, mais plutôt par des liaisons vers les **actions qu'on les considère capables d'effectuer**.

En d'autres termes, si l'on désire créer des fourmis, on ne cherchera pas à créer une classe paramétrable, composée du code correspondant à ce qu'elles savent faire, mais plutôt un ensemble de classes séparées décrivant chacune un qualificatif de l'insecte (se déplacer, rechercher de la nourriture, poser des phéromones…). Ces qualificatifs où méthodes sont aussi appelées des **possibilités.**

L'intérêt principal de cette séparation agent/qualificatif et de pouvoir **partager** les **possibilités** avec d'autres agents. Ainsi, une **possibilité** de mouvement codée de manière suffisamment abstraite sera utilisable aussi bien par un mammifère qu'une voiture.

On peut aussi imaginer créer des entités totalement irréelles comme des chats volants ou des lapins mangeurs d'hommes. Par le jeu des possibilités, AWE offre à l'utilisateur les **opportunités d'un dieu**.

Par ailleurs, chaque **possibilité** peut être ajoutée ou enlevée aux agents à tous moments sans briser leur fonctionnement ni même la **cohérence** de leurs actions.

AWE répond ainsi aux questions suivantes : et si cette entité qui fonctionne ce cette manière et sait faire ceci, à cet instant précis ne savait plus le faire ? Et si elle savait faire cela ? Et si elle résonnerait plutôt comme ça ? Quels en seraient les conséquences ?

Aussi, ce qui fait la l'essence d'**AWE** est certainement sa **conception totalement évolutive**, en effet, plus on programme des possibilités, plus les agents peuvent devenir complexes, plus ils ont des modes de raisonnement différents, plus ils peuvent vivre différemment.

Cette versatilité ouvre les portes de simulations complexes et sans limites : les agents ayant déjà un vécu pourront, sans problèmes, accéder à de nouvelles possibilités au cours de leur existence. On appelle aussi ceci : **l'évolution**.

Les Possibilités (package world.possibilities)

Définition

La somme des possibilités et des entités décrit l'ensemble que forme l'univers. Une possibilité représente un concept unique que les entités, appréhendent ou non.

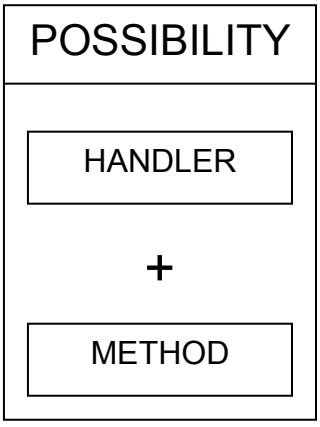
Dans la philosophie d'AWE, les entités sont appelés des agents et les possibilités représentent des notions comme : manger, attaquer, se reproduire, être violent, être amical, voir, entendre, comprendre, assimiler, etc...

Les agents sont donc essentiellement définis par leur perception des possibilités déjà présentes dans le macrocosme.

Pourtant, si le fait qu'une fourmi s'alimente et qu'un homme s'alimente se réduit conceptuellement à une même problématique qu'est la survie, certaines particularités ne doivent pas être négligées en fonction des l'espèces (l'homme peut, par exemple, manger par plaisir).

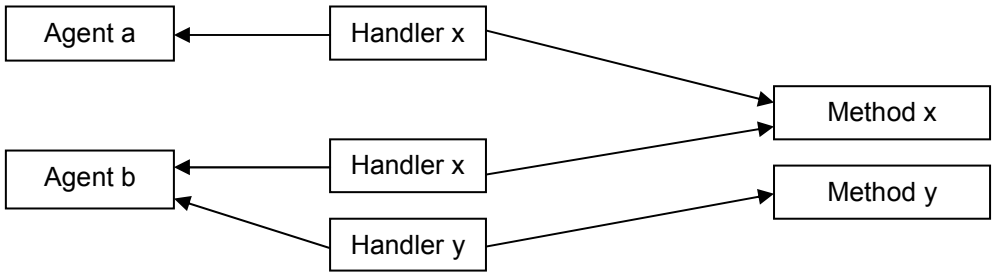
Ainsi, il est judicieux de définir une possibilité de la manière la plus abstraite possible, puis de descendre progressivement dans la hiérarchie de la complexité en stratifiant les particularités. L'intérêt de cette approche, bien évidemment liée au concept Objet, permettra une meilleure réutilisation du code qui accélérera le développement de nouvelles possibilités.

Par ailleurs, si une possibilité est clairement définie pour une espèce donnée, elle varie légèrement selon les individus. Tout homme peut, à priori, bander un arc, mais qu'en est-il du néophyte ? Même s'il est agile, il n'a pas les dizaines d'années de pratique du champion Olympique. Aussi, chaque possibilité se décompose en deux parties que nous appellerons le **support** (**Handler**) et la **méthode** (**Method**).



La **Method** contient **l'essence** de la possibilité, c'est à dire son **code**. Le **Handler**, quand à lui, contient ce qui **détaille** la possibilité d'un individu à l'autre, c'est à dire ses **variables,** il effectue aussi la **liaison** entre **un** agent et **une** Method.

Pour une possibilité donnée, il n'existe qu'une seule instance de la Method correspondante en mémoire, par contre, il existe autant d'instances d'Handlers que d'Agents ayant accès à des possibilités.



Cette distinction étant faite, il reste encore un problème : comment coder une possibilité pour qu'elle soit utilisable par tout agent de manière transparente tout en s'intégrant à l'environnement ?

Pour cela, la partie **Method** doit implémenter quatre concept : la **volonté**, la **cohérence**, le **fait** et la **dépendance** respectivement appelés **will**, **react**, **act et init**. Dans les prochaines versions d'AWE devrait apparaître un cinquième concept (agreact) qui aura en charge la notion de réaction chez les agent(s) observants ou subissants la manifestation de la possibilité.

Concrètement une possibilité est définie par deux classes. Pour que le serveur de possibilités (cf ServerPossibilities) soit capable de les charger convenablement, il faudra nommer les parties Handler et Method de la manière suivante :

Handler<type_de_possibilité><nom_de_la_possibilité>
Method<type_de_possibilité><nom_de_la_possibilité>

Par exemple, les deux classes de la possibilité de pensée générale (MindingGeneral) sont nommées :

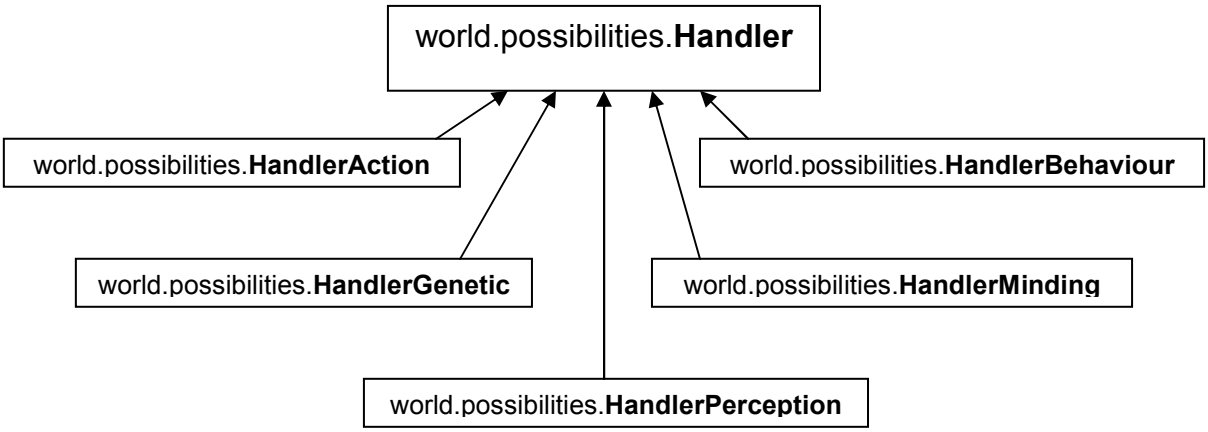
HandlerMindingGeneral.class
MethodMindingGeneral.class

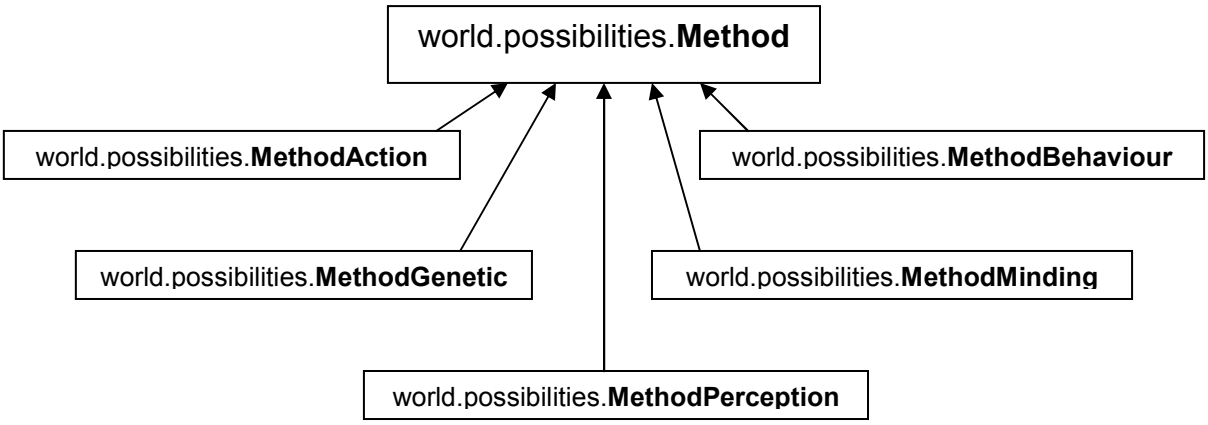
Cette approche peut paraître rébarbative, c'est pour cela que les prochaines versions d'AWE autoriserons l'utilisation de noms libres, néanmoins, il est conseillé de s'y tenir. En effet, le nombre de possibilités grandissant, il sera de plus en plus difficile de les identifier (est-ce une possibilité de pensée ou une possibilité d'action ? Est-ce un Handler ou une Method ?), avec cette spécification, on connaît le type et la partie en rien qu'en lisant le nom de la classe.

Dans la version actuelle d'AWE, les handlers et les methods des possibilités codées par le programmeur doivent être placés dans le répertoire possibilities/.

Précisons que le tutorial décrit en détail le processus de création des possibilités.

Arborescence hiérarchique (héritage)





Interfaces

world.possibilities.Handler

La classe abstraite Handler est la mère de toutes les classes Handler*. Elle contient une référence vers un Agent, ainsi qu'une référence vers une Method. La mise à jour de ces attributs est faite **automatiquement** par le moteur (cf ServerPossibilities). Rappelons que figureront dans les Handlers, les variables utilisée par la partie Method d'une possibilité.

Cette structure permet de dissocier le code qui est commun à tous les agents disposants d'une possibilité donnée, des variables qui la détaille.

Dans un souci de simplicité pour le programmeur, Handler dispose aussi des prototypes **will**, **act**, **react** et **init** qui ne font qu'appeler, en se passant en paramètre, leurs identiques de la classe Method correspondante, ainsi le programmeur pourra plus aisément les utiliser en codant des possibilités (cf Tutorial).

```
// m = référence sur la classe Method correspondante

public void init(){
     this.m.init(this) ;
}

public void act(){
     this.m.act(this);
}

public int will(){
     return this.m.will(this);
}
public boolean react(){
     return this.m.react(this);
}
```

Par ailleurs, il dispose d'un attribut world (cf World) qui permet notamment au Handler de fournir une référence vers le serveur de possibilités (cf.ServerPossibilities) dans le cadre de la méthode **init.**

Méthodes utiles

public final AgentActive getAgent()
renvoie l'agent propriétaire du Handler

public final Method getMethod()
renvoie la méthode correspondante à ce Handler

public final World getWorld()
renvoie la référence vers l'instance de la classe World courante

world.possibilities.Method

La classe abstraite Method est la mère de toutes les classes Method*, elle se décompose en trois sous méthodes qui sont appelées par le ou les Handlers qui la référent, nous verrons plus tard dans quelles conditions, pour le moment intéressons-nous à comment les sous classes de Method devront les implémenter.

```
abstract public boolean init(Handler h) ;
```

Cette méthode doit gérer l'initialisation de la possibilité. Si, par exemple, la possibilité nécessite que l'agent dispose d'une autre pour fonctionner, elle va automatiquement lui ajouter. Renvoie true si les dépendances ont pu être effectuées, false sinon. Dans ce dernier cas, le moteur n'affecte pas cette possibilité à l'agent.

```
abstract public int will(Handler h) ;
```

Déterminera un facteur de priorité correspondant à la manifestation de la possibilité. Une possibilité de se nourrir, par exemple, enverra un facteur de priorité proportionnel à l'état d'affaiblissement de l'agent.

```
abstract public boolean react(Handler h) ;
```

Renvera true si la manifestation de la possibilité est **cohérente**, false sinon. Dans ce dernier cas, le moteur n'exécutera pas la méthode **act**. Dans le cas d'une possibilité de déplacement, c'est cette méthode qui vérifiera que l'agent ne risque pas de prendre la place d'un mur.

```
abstract public void act(Handler h) ;
```

Contiendra le code correspondant à la **manifestation** de la possibilité. Dans le cas d'une possibilité de combat, par exemple, elle ira supprimer de l'énergie à la victime.

Méthode utile

public String getDescription()
renvoie la description de la possibilité

world.possibilities.MethodMinding
world.possibilities.HandlerMinding

C'est de ces classes abstraites qu'héritent les possibilités de pensée. Dans la version actuelle d'AWE, chaque agent n'a qu'une seule possibilité de pensée dont l'**act** est appelé par sa méthode **cycle**, elle-même appelée par le synchronizeur (cf Synchronizer).

Le principe d'une possibilités de pensée est d'appeler les méthodes **will** des possibilités de comportement dont dispose l'agent. Ces méthodes renvoient des facteurs de priorités calculés selon les besoins de l'agent. Ensuite, elle appelle la méthode **act** de la possibilité de comportement qu'elle a choisit.

Le choix le plus simple (actuellement utilisé par MindingGeneral) est de considérer la possibilité de comportement renvoyant le facteur de priorité le plus important.

Mais, on peut imaginer les traiter d'une foultitute de manières différentes : en tenant compte de la santé mentale de l'agent, par exemple. S'il est fou, ses actions peuvent être incohérentes, on choisira le comportement aléatoirement, s'il est en furie, on appellera systématiquement les comportements violents, etc…

Il est important de bien distinguer les notions qui différencient la pensée des comportements :

- un comportement correspond à un ensemble d'actions possibles. C'est donc une tendance constituée d'un certain nombre de désirs, la tendance effectue un choix parmi ses désirs : c'est un premier filtre.

- la pensée se situe au-dessus, elle traite les différents comportements ou tendances et effectue un deuxième choix : c'est le deuxième filtre.

En croisant la philosophie freudienne et celle d'AWE, l'ensemble des comportements constitue le « Ca », la pensée, le « Surmoi ». Le « Moi » n'est pas encore implémenté mais c'est en cours ☺

world.possibilities.MethodBehaviour
world.possibilities.HandlerBehaviour

C'est de ces classes abstraites qu'héritent les possibilités de comportement. Les possibilités actuellement fournies avec la version d'AWE définissent le comportement de manière relativement simpliste.

Un comportement correspond à un certain nombre d'actions que sait faire l'agent.

Un comportement s'opère plutôt qu'un autre suivant un facteur de priorité calculé par rapport aux besoins de l'agent.

La partie **react** renvoie pour le moment true, un comportement est donc toujours cohérent. Dans le cas d'un comportement agressif, il serait ici intéressant d'étudier le type de la victime pour éviter qu'une fourmi aille s'attaquer à un Eléphant. Mais il est aussi possible de laisser cette tâche aux possibilités d'action sous-jacentes.

De futures versions tenteront d'étoffer le concept.

Cependant, le « principe moteur » des possibilités de comportement ne devrait pas changer :

- quand elle est affectée à un agent, le moteur appelle sa méthode **init** qui se charge d'ajouter à l'agent les possibilités d'actions qu'elle sait gérer (dépendances).

- lorsque sa méthode **will** est appelée par la possibilité de pensée, elle appelle les **will** des possibilités d'action qu'elle connaît,

- en fonction des priorités renvoyées, elle détermine sa propre priorité qu'elle renvoie à la possibilité de pensée.

- lorsque sa méthode **act** est appelée par une méthode de pensée (elle a donc été choisie), elle empile l'action ayant renvoyée la plus grande priorité dans la pile d'intentions de l'agent (cf AgentActive). Il est néanmoins possible d'empiler plusieurs actions dans la pile d'intention (cf Parallelable).

world.possibilities.MethodAction
world.possibilities.HandlerAction

C'est de cette ces classes abstraites qu'héritent les possibilités d'action, pour l'instant ces deux classes sont vides et ne font que donner un type.

Une action s'apparente à un désir. Un ensemble de possibilités d'action liées à un comportement forment une tendance. Une action est obligatoirement liée à un ou plusieurs comportements.

Dans les prochaines versions d'AWE, et après avoir développé un certain nombre de possibilités d'action, il sera possible d'y remonter des méthodes ou variables utilisées en général par les classes filles de la ramification action.

world.possibilities.MethodPerception
world.possibilities.HandlerPerception

C'est de cette ces classes abstraites qu'héritent les possibilités de perception. On entend par possibilité de perception, la fenêtre sensorielle de l'agent sur le monde.

La méthode **will** des possibilités de perception n'est actuellement pas utilisée par la moteur, pourtant, il serait intéressant de l'utiliser pour vérifier si l'agent à toutes ses capacités sensorielles (s'il est aveugle temporairement, on renverra false, etc…).

world.possibilities.MethodGenetic
world.possibilities.HandlerGenetic

C'est de cette ces classes abstraites qu'héritent les possibilités de génétique. Malheureusement, aucune spécification n'existe à ce niveau. Les prochaines versions d'AWE devraient étoffer les concept.

Ces possibilités s'occuperont de gérer les parties reproduction, mutation, émergence de propriétés et génération des informations génétiques. Il sera notamment intéressant de travailler sur la notion de mutation entre les possibilités.

world.possibilities.Parallelable

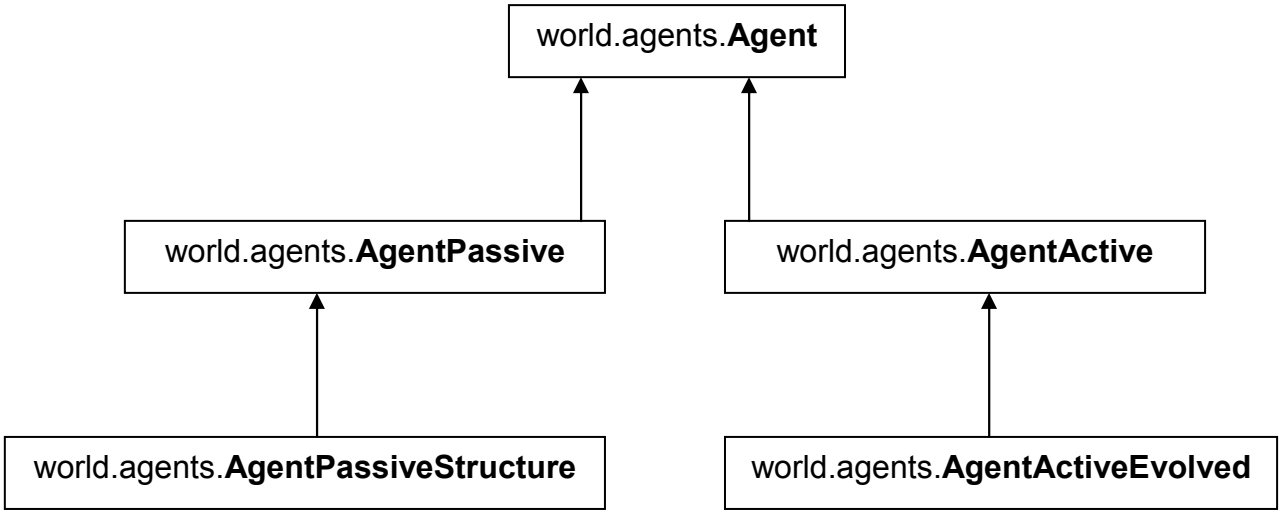
Les Handlers qui implémentent cette interface, peuvent être exécutés en même temps dans un même cycle (cf Environment, Synchronizer).

Les Agents (package world.agents)

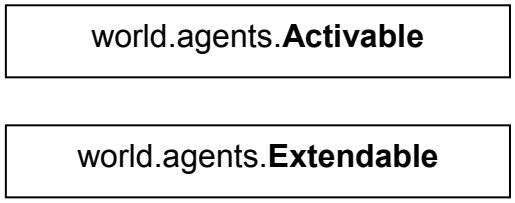
Définition

Les agents sont des entités qui composent l'environnement, on distingue les agents actifs des agents passifs. Globalement, les premiers on accès à des possibilités, ils peuvent donc penser, agir, réfléchir, se reproduire et ont un ou plusieurs comportements… Les seconds sont plutôt des éléments matériels ou des composantes physiques qui peuvent, dans certains cas, agir et/ou réagir, mais en aucun cas par le biais de la réflexion.

Arborescence hiérarchique (héritage)



Interfaces



world.agents.Agent

La classe Agent est une classe abstraite, elle ne peut donc être instanciée. Elle fourni le type générique d'un agent, c'est à dire son nom, sa description, son image, sa position ainsi que ses dimensions et l'environnement auquel il est attaché. Les noms des agents doivent être uniques car c'est la clé qui les identifie, notamment dans le serveur d'agents (cf ServerAgents).

Tous les agents implémentent l'interface clonable et doivent définir une méthode **clone**. Ces méthodes sont définies pour toutes les classes de la hiérarchie courante.

L'accès à leur position se fait par l'attribut public « position » qui est de type Position3D et admet des méthodes :

- setX(int)
- setY(int)
- setZ(int)
- getX(int)
- getY(int)
- getZ(int).

L'accès à leur dimension se fait par l'attribut public « dimension » qui est de type Dimension3D et admet les méthodes :

- setWidth(int)
- setLength(int)
- setHeight(int)
- getWidth(int)
- getLength(int)
- getHeight(int).

Méthodes utiles

public final void setName(String s)
affecte un nom à l'agent

public final String getName()
Renvoie le nom de l'agent

public final void setDescription(String s)
Affecte une description à l'agent

public final String getDescription()
Renvoie la description de l'agent

public final void setEnvironment(Environment e)
affecte un environnement à l'agent

public final Environment getEnvironment()
retourne l'environnement dans lequel l'agent se trouve

public final void setImage(Image i)
affecte une image à l'agent

public final Image getImage()
retourne l'image de l'agent

public final void setImageName(String imageName)
affecte le chemin d'accès à l'image de l'agent

public final String getImageName()
retourne le chemin d'accès à l'image de l'agent

public String getInfos()
retourne une chaîne contenant les informations de l'agent

public String toString()
retourne le nom de l'agent et rien d'autre (quand on ajoute un objet dans une zone de liste swing, elle utilise sa méthode toString pour l'identifier)

world.agents.AgentActive

C'est aussi une classe abstraite. Elle fourni le type générique d'un agent ayant accès à des possibilités, pour cela elle est composée de 5 tables pouvant contenir les parties handlers des possibilités d'action, de réflexion, de comportement, de perception et de génétique.

Un agent actif dispose aussi d'une pile d'intentions où seront empilés les Handlers, sélectionnés par la possibilité de comportement choisie par la possibilité de pensée, qui correspondent aux actions qu'il désire effectuer.

L'agent actif dispose d'une méthode abstraite **cycle** qui est appelée par le synchroniseur à chaque cycle (cf Synchronizer).

Cette méthode correspond au fait d'agir et doit être implémentée dans les sous-classes non abstraites de AgentActive.

Par ailleurs, un agent actif peut se situer dans les 4 états, qui peuvent être affectés par l'utilisateur ou les méthodes.

- STATE_SLEEP : l'agent est endormi, il ne peut exercer qu'une activité cérébrale. Cette fonctionnalité n'est pas encore implémentée au niveau du synchroniseur.

- STATE_IDLE : l'agent est oisif, il est invisible aux autres et n'agit plus. C'est état n'est théoriquement qu'affecté par l'utilisateur au niveau de l'interface.

- STATE_ACTIVE : l'agent est actif. Comportement normal, la méthode **cycle** est appelée par le **synchroniseur** à chaque cycle.

- STATE_DEAD : l'agent est mort. Il n'agit plus ni ne réagit mais reste physiquement présent pour les autres agents. La méthode **cycle** n'est plus appelée.

Méthodes utiles

public final boolean addHandler(Handler h)
objet : ajoute le handler h à l'agent
principe : le type du handler h est testé pour pouvoir le ranger dans la table appropriée, sa référence « agent » est mise à l'agent courant
retourne : true, si le handler a été ajouté à l'agent.

public final boolean removeHandler(String name)
objet : supprime le handler de nom name à l'agent
principe : recherche du handler dans les différentes tables
retourne : true, su le handler à bien été supprimé

public final Handler getHandler(String name)
retourne le handler de nom name, null si introuvable

public final Enumeration getHandlersAction()
retourne, sous forme d'énumération, les handlers d'action dont dispose l'agent.

public final Enumeration getHandlersMinding()
retourne, sous forme d'énumération, les handlers de pensée dont dispose l'agent.

public final Enumeration getHandlersPerception()
retourne, sous forme d'énumération, les handlers de perception dont dispose l'agent.

public final Enumeration getHandlersBehaviour()
retourne, sous forme d'énumération, les handlers de comportement dont dispose l'agent.

public final Enumeration getHandlersGenetic()
retourne, sous forme d'énumération, les handlers de génétique dont dispose l'agent.

public final Stack getWillStack()
retourne la pile d'intentions de l'agent

public final void pushIntention(Handler h)
empile le handler h dans la pile d'intentions

public final Handler popIntention()
dépile le handler situé au sommet de la pile d'intentions

public final Handler peekIntention()
retourne le handler situé au sommet de la pile d'intentions

public final boolean hasMoreIntentions()
renvoie true si la pile d'intentions n'est pas vide

public final byte getState()
retourne l'état de l'agent

public String getStateToString()
retourne l'état de l'agent sous forme de chaîne

public void setState(byte state)
change l'état de l'agent, si la valeur passée n'est pas valide, l'agent passe en état STATE_IDLE

public final void setStateSleep()
met l'état à STATE_SLEEP

public final void setStateIdle()
met l'état à STATE_IDLE

public final void setStateActive()
met l'état à STATE_ACTIVE

public final void setStateDead()
met l'état à STATE_DEAD

world.agents.AgentActiveEvolved

C'est la classe d'agent actuellement utilisée par les possibilités inclues avec le moteur. Elle implémente la méthode **cycle** qui ne fait qu'appeler la méthode **will** de la première possibilité de pensée, mais on peut l'imaginer plus complexe s'il on décide d'affecter plusieurs possibilités de pensée à l'agent.

```
public void cycle()
{
     if( !minding.isEmpty() )
     ((HandlerMinding)minding.elements().nextElement()).act();
}
```

La classe contient aussi un objet de type AgentActiveEvolvedAttributes décrit ci-après.

world.agents.AgentActiveEvolvedAttributes

Cette classe contient des valeurs comprises entre 0 et 100 pour les attributs suivants :

- brainpower (bra) intelligence
- willpower (wil) volonté
- sensitivity (sen) perception, sensibilité
- agility (agi) agilité
- dexterity (dex) dextérité
- charisma (cha) charisme
- strength (str) force
- constitution (con) constitution (résistance physique)
- luck (luc) chance

Ces attributs sont actuellement utilisés par les possibilités inclues avec le moteur et qualifient l'agent de manière abstraite. Certaines propriétés des Handlers sont calculées à partir de celles-ci.

Par ailleurs, elle contient aussi un attribut energy qui représente les forces vitales de l'agent. Quant elle atteint 0, l'agent est, théoriquement, mort.

L'énergie de départ est calculée de la manière suivante, c'est aussi le seuil maximum que l'agent ne peut dépasser :

(Constitution x 100) + (Force x 50)

Ce seuil maximum n'est remis à jour que si les attributs de l'agents sont tous régénérés (avec les méthodes seRandomAttributes, setAverageAttributes, setAttributes), mais pas si ils le sont fait indépendamment.

Méthodes utiles

public void setAttributes(byte bra, byte wil, byte sen, byte agi, byte dex, byte cha, byte str, byte con, byte luc)
affectation des valeurs passées en paramètres aux attributs

public void setAverageAttributes()
affectation de valeurs moyennes aux attributs

public void setRandomAttributes()
affectation aléatoire des attributs

public void generateEnergy()
cette méthode recalcule l'énergie en fonction de la force et la volonté

public final boolean setBrainpower(byte v)
affecte une nouvelle valeur à l'attribut brainpower, revoie true si modifié

public final boolean setWillpower(byte v)
affecte une nouvelle valeur à l'attribut willpower, renvoie true si modifié

public final boolean setSensitivity(byte v)
affecte une nouvelle valeur à l'attribut sensitivity, renvoie true si modifié

public final boolean setAgility(byte v)
affecte une nouvelle valeur à l'attribut agility, renvoie true si modifié

public final boolean setDexterity(byte v)
affecte une nouvelle valeur à l'attribut dexterity, renvoie true si modifié

public final boolean setCharisma(byte v)
affecte une nouvelle valeur à l'attribut charisma, renvoie true si modifié

public final boolean setStrength(byte v)
affecte une nouvelle valeur à l'attribut strength, renvoie true si modifié

public final boolean setConstitution(byte v)
affecte une nouvelle valeur à l'attribut constitution, renvoie true si modifié

public final boolean setLuck(byte v)
affecte une nouvelle valeur à l'attribut luck, renvoie true si modifié

public final byte getBrainpower()
renvoie la valeur de l'attribut brainpower

public final byte getWillpower()
retourne la valeur de l'attribut willpower

public final byte getSensitivity()
retourne la valeur de l'attribut sensitivity

public final byte getAgility()
retourne la valeur de l'attribut agility

public final byte getDexterity()
retourne la valeur de l'attribut dexterity

public final byte getCharisma()
retourne la valeur de l'attribut charisma

public final byte getStrength()
retourne la valeur de l'attribut strength

public final byte getConstitution()
retourne la valeur de l'attribut constitution

public final byte getLuck()
retourne la valeur de l'attribut luck

public final void setCurrentEnergy(int v)
affecte une nouvelle valeur à l'attribut currentEnergy, renvoie true si modifié

public final int getCurrentEnergy()
retourne l'énergie courante

public final int getMaxEnergy()
retourne l'énergie maximale

world.agents.AgentPassive

Les agents passifs sont des agents n'ayant pas accès à des possibilités, ils n'ont donc, en principe (cf. Activable) pas de méthode **cycle**. Ces agents peuvent néanmoins réagir à des actions effectuées par des agents actifs (un agent passif de type feu brûlerait un agent s'en approchant de trop près, un agent passif de type mur empêcherait la traversée…). C'est de cette classe abstraite que vont hériter les agents passifs crées par le programmeur.

L'ors de la programmation de nouveaux agents passifs, le programmeur devra impérativement définir le clonage de son nouvel agent en appelant tout d'abord **super.clone()** puis en définissant les nouvelles spécificités de clonage. Ceci s'explique par le fait que les agents passifs sont ajoutés par clonage dans l'environnement (cf ServerAgents).

Quand on programme un nouvel agent passif, Il est possible de définir des composants graphiques qui permettent d'éditer les attributs de l'agent au niveau de l'interface.

Il existe deux méthodes, la première, getComponent, renvoie un éditeur pour la souche à cloner, la deuxième, getClonedComponent, renvoie l'éditeur pour les clones (en général, le même composant que la souche).

Dans la classe abstraite AgentPassive, les méthodes sont implémentées de la manière suivante :

```
public JPanel getComponent()
{
     return null;
}

public JPanel getClonedComponent()
{
     return null;
}
```

Elles ne renvoient donc pas de composant, au niveau de l'interface, un processus de réflexion (cf. Doc Interface, Tutorial) permet tout de même d'en éditer les attributs publiques si ce sont des types de base (int, char, byte, etc…).

Dans la version actuelle d'AWE, on ne peut plus modifier l'orientation d'un clone de mur (AgentPassiveWallRect) une fois qu'il a été ajouté à un environnement, mais nous voulons tout de même fournir un éditeur graphique pour la souche. Pour cela, on ne redéfinit que la méthode getComponent.

De futures versions d'AWE tenteront d'exploiter la technologie Beans qui s'apparente en certains points à cette approche.

world.agents.AgentPassiveStructure

C'est de cette classe abstraite que vont hériter les agents passifs de structure crées par le programmeur.

Ces agents passifs incluent des notions de pression de température et de luminosité (actuellement non utilisées par le moteur).

Ils font souvent parti du monde microscopique et définissent des attributs structurels à des endroits de l'environnement.

Ainsi, l'environnement autorise un agent de type structure à occuper la même position qu'un agent passif classique ou qu'un agent actif. C'est le cas, par exemple, des phéromones (cf. Tutorial) qui dans la version actuelle d'AWE sont de type AgentPassiveStructure.

world.agents.Activable

Si un agent passif implémente cette interface et sa méthode **cycle**, il peut outrepasser sont statut de passif (l'agent passif AntHill fourni avec le moteur implémente cette interface pour générer des fourmis). Rappelons que « passif « est a prendre au sens « n'a pas accès à des possibilités ».

world.agents.Extendable

Un agent passif implémentant cette interface et sa méthode getPoints peut se situer à plusieurs positions (contiguës ou non). Ceci est utile pour des grandes étendues d'eau, ou de longs murs par exemple. En effet, il est inutile de créer autant d'agents passifs qu'il y a de morceaux de murs s'ils ont tous les mêmes propriétés.

Les Serveurs (package world.servers)

Définition

Les serveurs sont le noyau d'AWE, leur objectif est de mettre à la disposition d'une interface quelconque, une gestion client-serveur des différentes structures et concepts qui forment le moteur.

La version actuelle d'AWE fourni trois serveurs :

- le **serveur d'agents** (ServerAgents) s'occupe de l'enregistrement et du chargement des agents actifs, et de l'instanciation des classes d'agents passifs,

- le **serveur d'environnements** (ServerEnvironments) qui s'occupe de la sauvegarde d'un environnement et des instances d'agents passifs qui le compose.

- le **serveur de possibilités** (ServerPossibilities) s'occupe de la gestion des parties Handler et Method que constituent les possibilités,

Les répertoires de travail des serveurs sont définis en paramètres de leurs constructeurs, dans la version actuelle d'AWE, ils sont définis comme il suit :

ServerAgents	:	_agents/active/ et _agents/passive
ServerPossibilities	:	_possibilities/
ServerEnvironments	:	_environments/

Chaque objet est sérialisé avec son contenu dans une archive compressée au format GZIP du GNU.

Il est probable que de futures versions améliorent ces serveurs. Notamment en ce qui concerne la possibilité d'avoir plusieurs serveurs d'un même type sur des machines distantes.

world.servers.ServerAgent

Le serveur d'agent à la tâche de gérer la sérialisation, la desérialisation et la suppression des agents actifs du répertoire _agents/active/, ainsi que le chargement dynamique des classes d'agents passifs situées dans le répertoire _agents/passive/.

Les agents actifs n'étant pas traités de la même manière que les agents passifs, ils sont rangés dans deux tables distinctes.

La clé des agents passifs est égale au nom de leur classe, la clé des agents actifs est égale à leur attribut name.

Il est à préciser que les **noms des classes des agents passifs doivent toutes commencer par « AgentPassive »** suivi de ce que l'on veut. Ceci s'explique par le fait qu'au début les agents passifs et les agents actifs étaient dans le même répertoire et que nous ne savions pas encore que les uns seraient présents sous forme de classe et les autres sous forme d'objets sérialisés, il fallait donc trouver un moyen de les distinguer. Cette spécification rébarbative devrait disparaître dans les futures version d'AWE

Sérialisation d'un agent actif

Le processus de sérialisation permet d'enregistrer l'état d'un objet et de son contenu sur le disque dur. Les spécifications de sérialisation des agents actifs sont les suivantes :

- en ce qui concerne les possibilités, on ne sérialise que la partie Handler et non la partie Method. Rappelons que seule la partie Handler est propre à l'agent, la partie Method, qui est unique pour une classe de possibilité donnée, est chargée par le serveur de possibilités (cf ServerPossibilities).

- l'image de l'agent n'est pas sérialisée, par contre, on enregistre, sous forme de chaîne de caractère, son chemin d'accès.

- on ne sérialise pas l'environnement auquel est rattaché l'agent, ce processus est effectué par le serveur d'environnement (cf ServerEnvironment). Ainsi, un agent sérialisé n'appartient plus à un environnement, il se retrouve donc dans le chaos jusqu'à qu'il soit à nouveau ajouté à un environnement.

- les autres attributs de l'agent sont normalement sérialisés

- le nom donné au fichier est celui de l'attribut name de l'agent.

Les prochaines versions d'AWE se devront de spécifier si la pile d'intentions doit être sérialisée (ce qu'elle est actuellement) ou non. De plus, il sera possible de maintenir la jointure des agents avec leur environnement.

Désérialisation d'un agent actif

Le processus de déserialisation permet de rétablir un objet sérialisé sur le disque dur. Les spécifications de la désérialisation des agents actifs sont les suivantes :

- pour chaque Handler dont dispose l'agent, rechercher si la partie Method à été chargée par le serveur de possibilités (cf ServerPossibilities), si c'est le cas, mettre à jour la référence « method » du Handler ainsi que la référence « world », sinon supprimer la possibilité et réinitialiser la pile d'intentions.

- si le chemin d'accès de l'image est valide, charger l'image et l'affecter à l'agent, sinon affecter l'image par défaut (cf World)

- ajout dans la table des agents actifs si le nom de l'agent est unique.

Chargement des classes d'agents passifs

Les agents passifs contenus dans un environnement ne sont pas sérialisés par le serveur d'agent, mais par le serveur d'environnement (cf ServerEnvironment). L'objectif ici est de charger les **classes** d'agents passifs et **non une instance sérialisée**. Le processus de chargement des classes d'agents passifs se décrit comme il suit, c'est un ClassLoader (cf doc JDK) :

- recherche des fichiers *.class se situant dans le répertoire _agents/passive/

- chargement en mémoire si le nom de la classe commence par « AgentPassive »

- transformation en objet Class si la classe est accessible en mode « public »

- instanciation et ajout dans la table des agents passifs, l'objet Class est détruit.

Par ailleurs, si le programmeur désire qu'une classe d'agent passif ne soit pas chargée par le serveur tout en restant utilisable par un agent actif, il suffit de ne pas la déclarer en « public class … ».

C'est le cas, par exemple, de la classe AgentPassiveStructurePheromone qui représente des phéromones que seules certaines possibilités peuvent générer, et non l'utilisateur.

Par la suite, le fait d'ajouter un agent passif dans un environnement se ramène à ajouter un clone d'une des souches des classes que le serveur d'agent à chargé. L'intérêt de cette approche est de pouvoir paramétrer une souche, puis d'ajouter des instances identiques sans avoir besoin de redéfinir à chaque fois ses attributs.

Par exemple, l'agent passif WallRect, peut prendre des tailles différentes. Si l'on désire ajouter plusieurs murs d'une certaine taille, il aurait été ennuyeux de la redéfinir à chaque fois.

Ajouter un agent actif, du fait de son unicité, se ramène à ajouter sa référence.

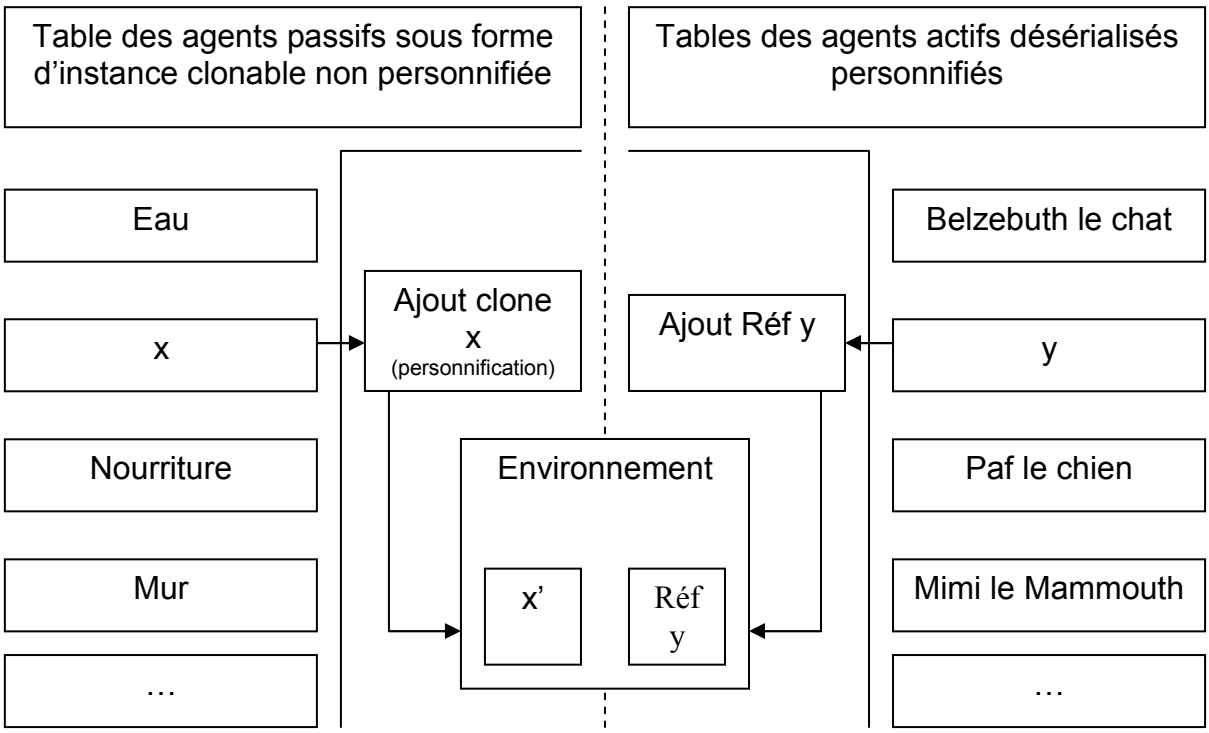


Table des agents passifs sous forme d'instance clonable non personnifiée
Tables des agents actifs désérialisés personnifiés
Eau
x
Nourriture
Mur
…
Ajout clone
x
(personnification)
Ajout Réf y
Environnement
x'
Réf
y
Belzebuth le chat
y
Paf le chien
Mimi le Mammouth
…

Méthodes utiles

public void loadAgents()
objet : mise à jour des tables d'agents passifs et actifs.
principe : appel des méthodes loadAgentsPassive et loadAgentsActive.
retourne : rien.

public int loadAgentsPassive()
objet : mise à jour de la table des agents passifs.
principe : recherche de fichiers *.class dans le répertoire réservé aux agents passifs, appel de la méthode loadAgentPassive pour chaque fichier.
retourne : le nombre d'agents passifs chargés.

public int loadAgentsActive()
objet : mise à jour de la table des agents actifs.
principe : recherche d'objets sérialisés dans le répertoire réservé aux agents actifs, appel de la méthode loadAgentActive pour chaque fichier.
retourne: le nombre d'agents actifs chargés.

public boolean loadAgentPassive(String filename)
objet : chargement d'un agent passif de nom filename.
principe : classloading du fichier *.class, instanciation et ajout dans la table des agents passif.
retourne: true, si l'agent passif à bien été chargé.

public boolean loadAgentActive(String filename)
objet : chargement d'un agent actif de nom filename.
principe : désérialisation du fichier correspondant à l'agent précédemment sérialisé, mise à jour des possibilités et ajout dans la table des agents actifs.
retourne: true, si l'agent actif à bien été désérialisé.

public int writeAgentsActive()
objet : enregistrement des agents actifs contenus dans le serveur.
principe : appel de la méthode writeAgentActive pour chaque agent contenu dans la table des agents actifs.
retourne: le nombre d'agents actifs sérialisés.

public boolean writeAgentActive(AgentActive a)
objet : enregistrement de l'agent actif a.
principe : sérialisation de l'objet agent actif a et de son contenu avec compression.
retourne: true, si l'agent actif à bien été sérialisé.

public boolean addAgentActive(AgentActive a)
objet : ajout de l'agent actif a dans le serveur.
principe : vérification qu'un agent de même nom n'existe pas encore dans la table des agents actifs, dans ce cas l'agent a est ajouté dans la table.
retourne: true, si l'agent actif à été ajouté dans le serveur.

public int removeAgentsActive()
objet : suppression des agents actifs contenus dans le serveur.
principe : appel de la méthode removeAgentActive pour chaque agent contenu dans la table des agents actifs.
retourne: le nombre d'agents actifs supprimés.

public boolean removeAgentActive(String a)
objet : suppression de l'agent actif de nom a du serveur et du disque.
principe : vérification qu'un agent de même nom existe dans la table des agents actifs, dans ce cas l'agent a est supprimé dans la table et du disque(si c'est un agent précédemment sérialisé).
retourne: true, si l'agent actif à été supprimé du serveur.

public boolean removeAgentActive(AgentActive a)
objet : suppression de l'agent actif a du serveur et du disque.
principe : vérification qu'un agent de même nom existe dans la table des agents actifs, dans ce cas l'agent a est supprimé dans la table et du disque (si c'est un agent précédemment sérialisé).
retourne: true, si l'agent actif à été supprimé du serveur.

public AgentActive getAgentPassive(String name)
objet : acquisition d'un clone de l'agent passif de nom name (name est le nom d'une classe d'agent passif moins « AgentPassive », pour récupérer un clone d'une agent passif s'appelant « AgentPassiveWall », appeler la méthode avec « Wall » en paramètres.
principe : recherche de l'agent dans la table des agents passif, s'il existe, on retourne sa référence.
retourne: la référence de l'agent si trouvé, null dans le cas contraire.

public AgentActive getAgentActive(String name)
objet : acquisition de l'agent actif de nom name.
principe : recherche de l'agent dans la table des agents actifs, s'il existe, on retourne sa référence.
retourne: l'agent si trouvé, null dans le cas contraire.

public final void clearAgentsPassive()
suppression du contenu de la table des agents passifs. Cette méthode n'est utile que si, au cours de l'exécution, le chemin d'accès aux agents passifs est changé.

public final void clearAgentsActive()
suppression du contenu de la table des agents actifs. Les agents actifs sérialisés ne sont pas supprimés du disque. Cette méthode n'est utile que si, au cours de l'exécution, le chemin d'accès aux agents sérialisés est changé (dans le cas d'une extension réseau du serveur, on pourra spécifier des adresses contenant d'autres agents).

public final int getAgentsPassiveCount()
retourne le nombre d'agents passifs présents dans le serveur.

public final int getAgentsActiveCount()
retourne le nombre d'agents actifs présents dans le serveur.

public final Enumeration getAgentsPassive()
retourne, sous forme d'énumération, les agents passifs contenus dans le serveur.

public final Enumeration getAgentsActive()
retourne, sous forme d'énumération, les agents actifs contenus dans le serveur.

public final Vector getAgentsPassiveVector()
retourne, sous forme de vecteur (Liste), les agents passifs contenus dans le serveur. Utile pour les zones de liste Swing (cf. JDK).

public final Vector getAgentsActiveVector()
retourne, sous forme de vecteur (Liste), les agents actifs contenus dans le serveur. Utile pour les zones de liste Swing (cf. JDK).

public final String getPath()
retourne le répertoire de travail du serveur d'agents, les agents actifs sont stockés dans le sous répertoire active/ de ce répertoire, les agents passifs dans passive/.

public final void setPath(String agentsPath)
change le répertoire de travail du serveur d'agents, attention les tables ne sont pas vidées.

world.servers.ServerEnvironment

Le serveur d'environnement à la tâche d'enregistrer, de charger et de supprimer des environnements crées par l'utilisateur ainsi que les agents passifs le constituant.

Le serveur est composé de deux listes, l'une contient les noms des environnements disponibles, l'autre contient la liste des agents passifs contenus dans l'environnement courant.

Sérialisation d'un environnement

Les agents actifs contenus dans un environnement, ni le synchroniseur (cf Synchronizer), ne sont sérialisés.

Désérialisation d'un environnement

La déserialisation d'un environnement s'opère comme il suit :

- on vérifie que le fichier est bien un environnement
- on balaie les cases de l'environnement à la recherche d'agents passifs qu'on ajoute dans la liste qui leur est réservée
- le serveur assume cet environnement comme l'environnement courant, on appelle sa méthode initialize (cf Environment)

De futures versions d'AWE tenteront de permettre la sérialisation en cascade des agents actifs tout en maintenant la possibilité de les déliés des environnements à volonté.

Méthodes utiles

public int checkEnvironments()
objet : mise a jour de la liste des environnements.
principe : on ne fait qu'ajouter dans la liste des environnements les noms des fichiers trouvés dans le répertoire réservé aux environnements sérialisés. Attention, on ne vérifie pas encore si ces fichiers sont bien des environnements, car la désérialisation prend trop de temps.
retourne : le nombre de fichiers trouvés.

public Environment loadEnvironment(String name)
objet : déserialisation de l'environnement portant le nom de fichier name.
principe : on recherche le fichier dans le répertoire réservé aux environnements sérialisés, on le déserialise, on vérifie que c'est bien un environnement, si c'est le cas on appelle la méthode checkAgents pour mettre la liste des agents passifs contenus dans cet environnement à jour
retourne : l'environnement déserialisé, null si la procédure a échouée.

public int checkAgents(Environment environment)
objet : mise à jour de la liste des agents passifs.
principe : on balaie les cases de l'environnement à la recherche d'agents passifs qu'on ajoute dans la liste qui leur est réservée.
Retourne : le nombre d'agents passifs trouvés dans l'environnement.

public boolean writeEnvironment(Environment environment)
objet : enregistrement de l'environnement et de ses agents passifs
principe : sérialisation de l'objet environment et de son contenu en un fichier compressé.
retourne : true si l'environnement et ses agents passifs on bien été sérialisés.

public boolean delEnvironment(String name)
objet : suppression d'un environnement.
principe : si le fichier existe il est supprimé du disque, la liste des agents passifs est vidée et la méthode checkEnvironments est rappelée.
retourne : true si le fichier a été supprimé.

public final Vector getAgentsPassiveVector()
retourne la liste des agents passifs contenus dans l'environnement courant.

public final Vector getEnvironmentsVector()
retourne la liste des fichiers contenus dans le répertoire réservés aux environnements sérialisés

world.servers.ServerPossibilities

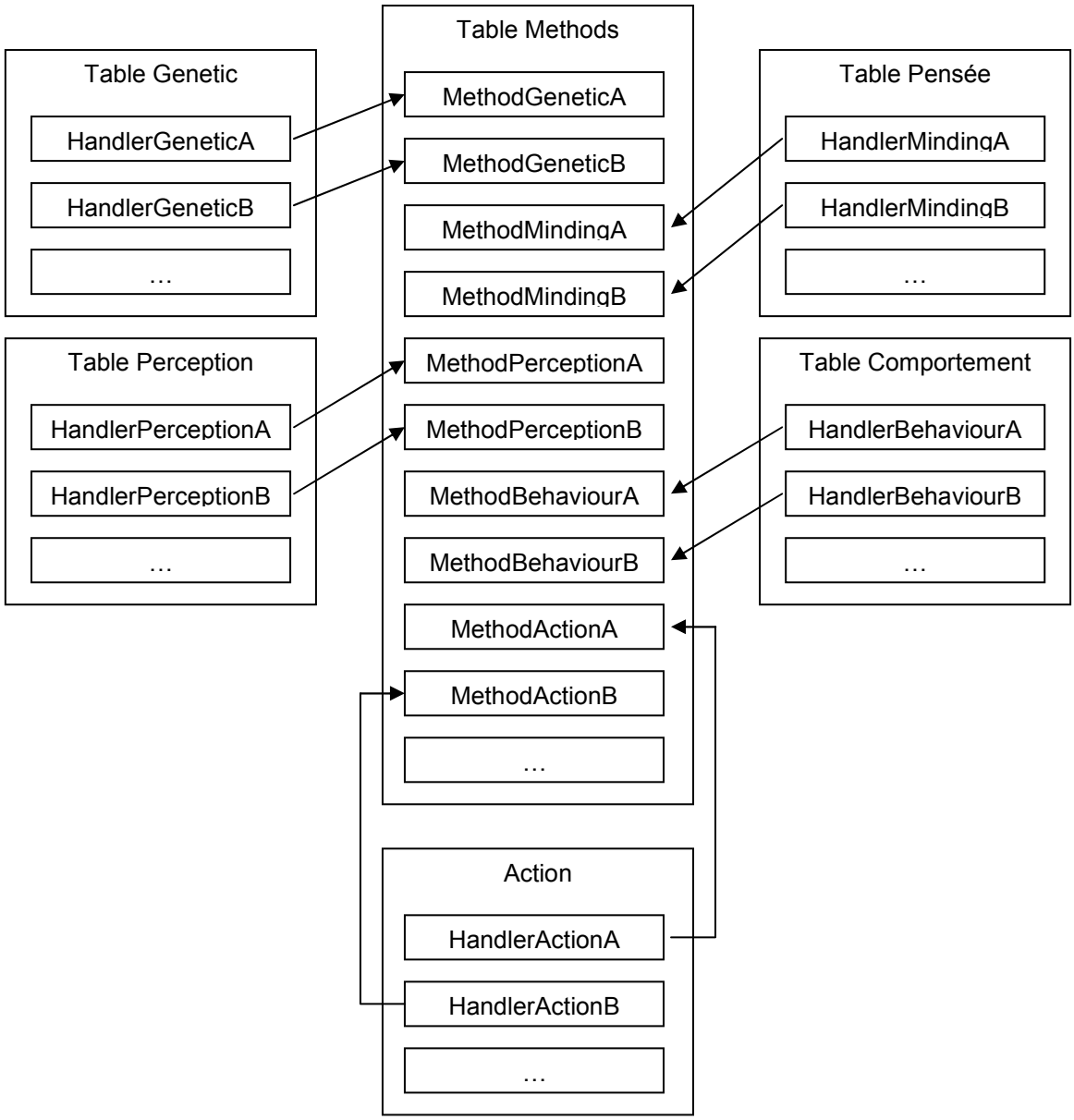
C'est au serveur de possibilité qu'incombe la tâche de charger les parties Handler et Method des possibilités qui se trouvent dans le répertoire _possibilities/. Ainsi, la liste des possibilités disponibles est récupérable par une interface.

Par ailleurs, le fait d'ajouter une nouvelle possibilité à un agent se fera aussi par le biais du serveur.

Le serveur de possibilités contient six tables dont les clés sont les noms des classes :

- une table pour chacun des cinq types de possibilités existants dans lesquelles on placera les objets Class des parties Handler des possibilités. Ceci permet à une interface de connaître rapidement les possibilités d'action, de perception, de pensée, de comportement ou de génétique disponibles.

- une table contenant les instances des parties Method des possibilités.



Chargement des possibilité

Précisons tout de suite que le serveur ignorera toute possibilité qui n'est pas nommée suivant les spécifications décrites dans la définition des possibilités. (cf. Les Possibilités)

Pour qu'une possibilité soit chargée, il faut déposer sa partie Handler et sa partie Method dans le répertoire de travail du serveur, le chargement s'opère comme il suit :

- recherche de fichiers commençant par « Handler » et ayant l'extension *.class

- pour chaque handler trouvé, rechercher le fichier qui commençant par « Method » correspondant. Si on a ces deux fichiers, tout ce qu'il faut est réuni pour charger la possibilité.

- on transforme le fichier de la method en objet Class qu'on instancie et que l'on place dans la table des methods. L'objet Class est détruit

- on transforme le fichier du handler en objet Class que l'on place dans l'une des cinq tables des handlers (suivant le type).

Ajout d'une possibilité à un agent

La procédure d'ajout d'une possibilité à un agent se déroule de la manière suivante :

- recherche du Handler « possibilitéA » dans la table correspondante au type de la possibilité

- recherche de la Method « possibilitéA » correspondante dans la table des methods

- si les deux parties sont trouvées : instanciation du Handler « possibilitéA »

- ensuite on affecte à l'attribut « method » du nouveau handler la référence de la Method « possibilitéA ».

- enfin on ajoute le nouveau handler à l'agent.

Maintenant, le nouvel agent pourra utiliser cette nouvelle possibilité dont il ne conserve que la partie Handler, qui elle-même pointe vers la partie Method. Nous voyons ici l'illustration d'un des mécanismes qui fait la force d'AWE : la gestion intelligente de la mémoire.

Notons que les possibilités implémentant la méthode **init** (cf. Possibilities) utilisent aussi cette fonctionnalité du serveur de possibilité pour gérer l'ajout des possibilités dont elles sont dépendantes.

Méthodes utiles

public int loadPossibilities()
objet : mise à jour des tables des possibilités.
principe : recherche des fichiers commençant par Handler*.class dans le répertoire réservé aux possibilités, appel de la méthode loadPossibility pour chaque fichier trouvé.
retourne : *le nombre de possibilités chargées*

public boolean loadPossibility(String name)
objet : charge la possibilité de nom name dans le serveur.
principe : recherche des parties Handler et Method. Si ils sont trouvés, l'objet Class du handler est placé dans la table appropriée, la partie Method est instanciée dans la table des methods.
retourne : true, si la possibilité à été chargée (partie Handler et Method).

public boolean setPossibility(String name, AgentActive a)
objet : ajoute la possibilité de nom name à l'agent a.
principe : recherche de la partie handler dans les tables, recherche de la partie method correspondante, instanciation de la partie handler et mise à jour de sa référence vers la partie method, ajout du handler à l'agent.
retourne : true, si la possibilité à été ajoutée à l'agent.

public boolean setHandler(String name, AgentActive agent)
idem que setPossibility mais l'argument name doit être précédé de « Handler ». Cette méthode est dépréciée.

public Handler getHandler(String name)
retourne une instance de Handler de nom name. Cette méthode est dépréciée.

public Enumeration getPossibilitiesAction()
retourne, sous forme d'énumération, les possibilités d'action présentes dans le serveur

public Enumeration getPossibilitiesBehaviour()
retourne, sous forme d'énumération, les possibilités de comportement présentes dans le serveur

public Enumeration getPossibilitiesGenetic()
retourne, sous forme d'énumération, les possibilités de génétique présentes dans le serveur

public Enumeration getPossibilitiesMinding()
retourne, sous forme d'énumération, les possibilités de pensée présentes dans le serveur

public Enumeration getPossibilitiesPerception()
retourne, sous forme d'énumération, les possibilités de perception présentes dans le serveur

public Vector getPossibilitiesActionVector()
retourne, sous forme de vecteur (Liste), les possibilités d'action présentes dans le serveur

public Vector getPossibilitiesBehaviourVector()
retourne, sous forme de vecteur (Liste), les possibilités de comportement présentes dans le serveur

public Vector getPossibilitiesGeneticVector()
retourne, sous forme de vecteur (Liste), les possibilités de génétique présentes dans le serveur

public Vector getPossibilitiesMindingVector()
retourne, sous forme de vecteur (Liste), les possibilités de pensée présentes dans le serveur

public Vector getPossibilitiesPerceptionVector()
retourne, sous forme de vecteur (Liste), les possibilités de perception présentes dans le serveur

public Hashtable getPossibilitiesActionHashtable()
retourne, sous forme de table, les possibilités d'action présentes dans le serveur

public Hashtable getPossibilitiesBehaviourHashtable()
retourne, sous forme de table, les possibilités de comportement présentes dans le serveur

public Hashtable getPossibilitiesMindingHashtable()
retourne, sous forme de table, les possibilités de pensée présentes dans le serveur

public Hashtable getPossibilitiesPerceptionHashtable()
retourne, sous forme de table, les possibilités de perception présentes dans le serveur

public Hashtable getPossibilitiesGeneticHashtable()
retourne, sous forme de table, les possibilités de génétique présentes dans le serveur

public Enumeration getMethodsElements()
retourne, sous d'énumération, les parties Method des possibilités présentes dans le serveur.

public void setPath(String path)
affecte un nouveau répertoire de travail au serveur

public String getPath()
retourne le répertoire de travail du serveur

L'environnement (package world.environment)

Définition

Un environnement est un espace tridimensionnel constitué d'agents passifs, dans lequel évoluent des agents actifs.

world.environment.Environment

La classe Environment est constituée d'une Map qui est un tableau tridimensionnel d'objets MapPoint, ainsi que d'un synchroniseur.

Réaction de l'environnement

L'environnement se doit, en plus de contenir les agents, de vérifier la cohérence de leurs actions, ainsi le processus de réaction se déroule comme il suit :

pour chaque agent présent dans l'environnement

- on appelle la méthode react du Handler situé au sommet de la pile d'intentions de l'agent
- si elle renvoie true, l'action est cohérente donc on appelle la méthode act de ce même Handler
- on dépile le Handler
- si le Handler dépilé est de type Parallelable, on réitère le processus tant que le sommet de la pile d'intention contient un Handler de type Parallelable

Méthodes utiles

public void initialize(Viewport v)
cette méthode est, en général, uniquement appelée lors de la création d'un environnement ou l'ors de sa désérialisation, elle vide la liste des agents actifs et crée un nouveau synchroniseur

public synchronized void start()
cette méthode gère le départ de l'activité du synchroniseur

public synchronized final void setInPause(boolean b)
cette méthode gère le départ et l'arrêt de l'activité du synchroniseur

public final void react()
objet : vérification de la cohérence des actions des agents actifs
principe : appelle le react du handler au sommet de chaque pile d'intentions, si elles renvoient true, il appelle leur act.
retourne : rien.

public final synchronized boolean addAgent(Agent ag)
objet : ajout de l'agent ag dans l'environnement.
principe : met l'environnement en pause, vérifie le type de l'agent, vérifie que sa position est libre et l'y place. Dans ce cas, l'attribut « environment » de l'agent est mis à l'environnement courant.
retourne : true, si l'agent a bien été ajouté à l'environnement.

public final synchronized boolean removeAgent(Agent ag)
objet : suppression de l'agent ag de l'environnement.
principe : met l'environnement en pause, vérifie le type de l'agent, le supprime de sa position et met à null l'attribut « environment » de l'agent.
retourne : true, si l'agent a été supprimé de l'environnement.

public boolean indexAreInBounds(int x, int y, int z)
retourne true si la position (x,y,z) existe dans l'environnement

public boolean indexAreInBounds(Position3D p)
retourne true si la position p existe dans l'environnement

public Synchronizer getSynchronizer()
retourne le synchronisateur courant

public final Vector getActiveAgents()
retourne un vecteur (Liste) contenant la liste des agents actifs contenus dans l'environnement

public final Dimension3D getDimension()
retourne sous forme d'objet Dimension3D, les dimensions de l'environnement

public final Map getMap()
retourne l'objet Map courant de l'environnement

public final void setName(String name)
affecte un nouveau nom à l'environnement

public final String getName()
retourne le nom de l'environnement

public final void setDescription(String description)
affecte une nouvelle description à l'environnement

public final String getDescription()
retourne la description de l'environnement

world.environment.map.Map

La classe Map est un tableau tridimensionnel composé d'objets de type MapPoint.

Méthodes utiles(en principe elle sont utilisées par l'objet Environment)

public final MapPoint getMapPoint(int x, int y, int z)
retourne l'objet MapPoint situé à la position (x,y,z)

public final MapPoint getMapPoint(Position3D p)
retourne l'objet MapPoint situé à la position p

public final Agent getAgentAt(int x, int y, int z)
retourne l'agent situé à la position (x,y,z)

public final Agent getAgentAt(Position3D pos)
retourne l'agent situé à la position p

public final void setAgent(Agent ag)
place l'agent ag à la position qui correspond à son attribut position

public final void setAgentAt(Agent ag, Position3D p)
place l'agent ag à la position p

public final void removeAgentAt(int x, int y, int z)
supprime l'agent de sa position (x,y,z)

public final void removeAgentAt(Position3D pos)
supprime l'agent de sa position pos

public final AgentPassiveStructure getStructureAt(int x, int y, int z)
retourne l'agent passif de type structure situé à la position (x,y,z)

public final AgentPassiveStructure getStructureAt(Position3D p)
retourne l'agent passif de type structure situé à la position p

public final void setStructureAt(AgentPassiveStructure struct, int x, int y,int z)
place l'agent passif de type structure à la position (x,y,z)

public final void setStructure(AgentPassiveStructure struct)
place l'agent passif de type structure à la position qui correspond à son attribut position

public final void setStructureAt(AgentPassiveStructure struct, Position3D p)
place l'agent passif de type structure à la position p

public final void removeStructureAt(Position3D p)
supprime l'agent passif de type structure situé à la position p

public final int getMaxX()
retourne la longueur de la map

public final int getMaxY()
retourne la largeur de la map

public final int getMaxZ()
retourne la dimension de l'axe z de la map

world.environment.map.MapPoint

En principe, chaque MapPoint ne peut contenir qu'un seul agent actif ou passif en plus d'un éventuel agent passif de type structure (cf AgentPassiveStructure).

De futures versions d'AWE, géreront les MapPoints comme des listes de MapPoints, cette construction chaînée permettra de définir des environnements où coexisteront des entités de dimensions différentes (échelle universelle, planétaire, humaine, atomique, subatomique, etc…).

Méthodes utiles

public final void setAgent(Agent ag)
place l'agent ag

public final Agent getAgent()
retourne l'agent

public final void setStructure(AgentPassiveStructure struct)
place l'agent passif de type structure

public final AgentPassiveStructure getStructure()
retourne l'agent de type structure

public final Position3D getPosition()
retourne la position de ce MapPoint

world.environment.synchronizer.Synchronizer

L'objectif du synchroniseur est de gérer la synchronisation entre les agents, l'environnement l'affichage. Il est possible de modifier sa vitesse d'exécution car c'est un Thread (cf. JDK).

En principe, le synchroniseur n'est pas piloté directement par le programmeur, mais par les méthodes **start** et **setInPause** de l'environnement.

L'unité de temps la plus petite dans AWE est le cycle, au cours de chaque cycle, le synchroniseur effectue les opérations suivantes :

- appel des méthodes **cycle** des agents passifs activables (cf. Activable)
- appel des méthodes **cycle** des agents actifs (cf. AgentActive)
- demande de réaction de l'environnement
- rafraîchissement de l'affichage

Méthodes utiles

public final void updateDisplay(boolean b) {
détermine si l'affichage soit être mis à jour

public final void setInPause() {
met le synchroniseur en pause

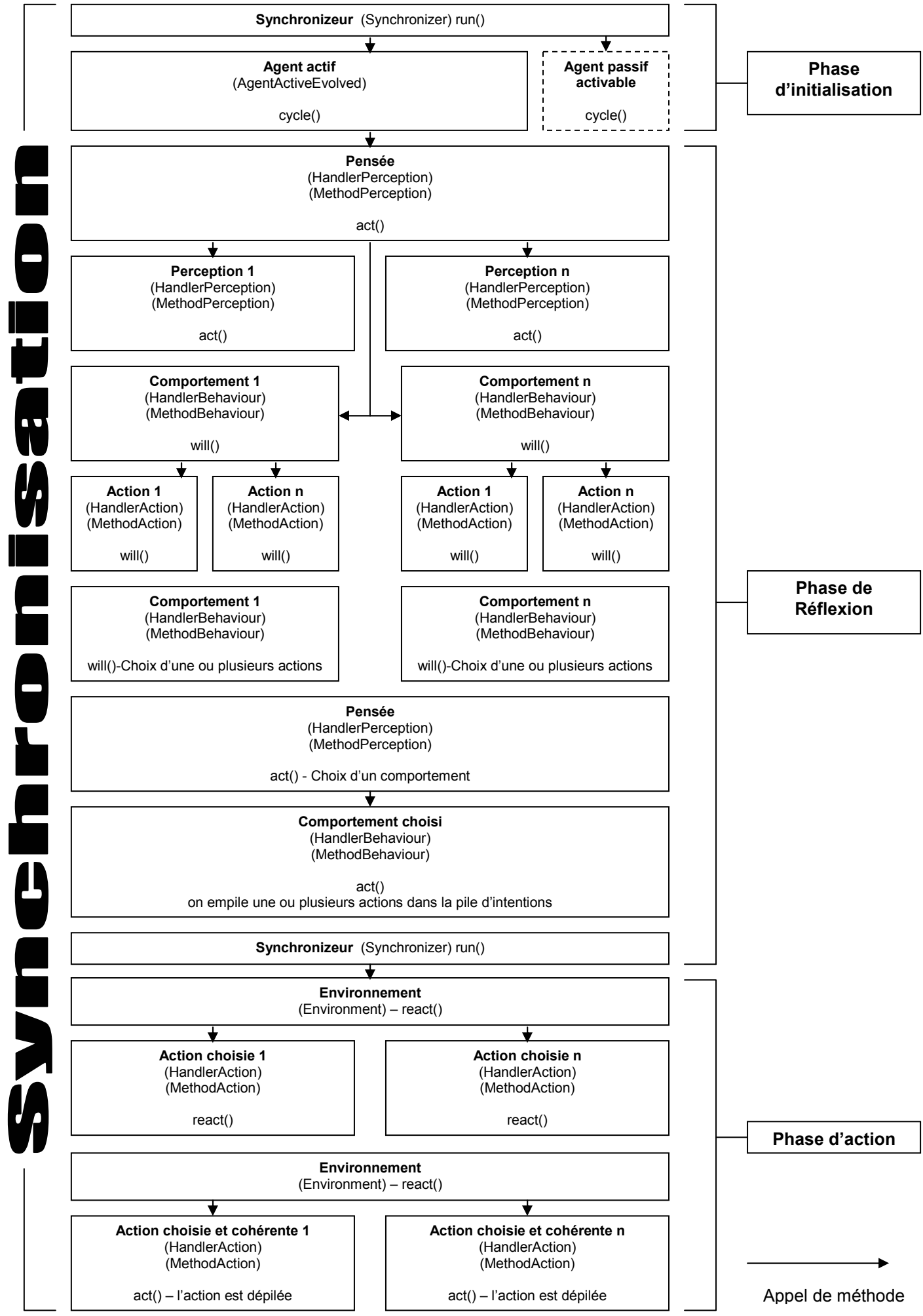
public final boolean isInPause() {
retourne true si le synchroniseur est en pause

public final void wakeUp() {
réveille le synchroniseur après une pause

public final void finish() {
force le synchroniseur à terminer son exécution

public final void setDelay(int delay) {
modifie la vitesse d'exécution du synchroniseur, delay est exprimé en millisecondes.

public final int getDelay () {
retourne la vitesse d'exécution du synchroniseur exprimée millisecondes, elle correspond au temps d'attente entre deux cycles.


Synchronisation
Synchronizeur (Synchronizer) run()
Agent actif
(AgentActiveEvolved)
cycle()
Agent passif activable
cycle()
Phase d'initialisation
Pensée
(HandlerPerception)
(MethodPerception)
act()
Perception 1
(HandlerPerception)
(MethodPerception)
act()
Perception n
(HandlerPerception)
(MethodPerception)
act()
Comportement 1
(HandlerBehaviour)
(MethodBehaviour)
will()
Comportement n
(HandlerBehaviour)
(MethodBehaviour)
will()
Action 1
(HandlerAction)
(MethodAction)
will()
Action n
(HandlerAction)
(MethodAction)
will()
Action 1
(HandlerAction)
(MethodAction)
will()
Action n
(HandlerAction)
(MethodAction)
will()
Comportement 1
(HandlerBehaviour)
(MethodBehaviour)
will()-Choix d'une ou plusieurs actions
Comportement n
(HandlerBehaviour)
(MethodBehaviour)
will()-Choix d'une ou plusieurs actions
Phase de Réflexion
Pensée
(HandlerPerception)
(MethodPerception)
act() - Choix d'un comportement
Comportement choisi
(HandlerBehaviour)
(MethodBehaviour)
act()
on empile une ou plusieurs actions dans la pile d'intentions
Synchronizeur (Synchronizer) run()
Environnement
(Environment) – react()
Action choisie 1
(HandlerAction)
(MethodAction)
react()
Action choisie n
(HandlerAction)
(MethodAction)
react()
Phase d'action
Environnement
(Environment) – react()
Action choisie et cohérente 1
(HandlerAction)
(MethodAction)
act() – l'action est dépilée
Action choisie et cohérente n
(HandlerAction)
(MethodAction)
act() – l'action est dépilée
Appel de méthode

World (world.World) et WorldLauncher

La classe WorldLauncher constitue le main d'AWE, elle commence tout d'abord par créer une instance de la classe World, puis elle crée le serveur de possibilités, le serveur d'agent et le serveur d'environnements qui chargent respectivement leur modules, enfin elle crée les composants de l'interface avec un environnement de base.

La classe World peut être considérée comme un central téléphonique. Actuellement, elle contient des références vers la plupart des composants graphiques de l'interface ainsi que vers serveurs.

Aussi, tous les composants et objets d'AWE peuvent se joindre via sa référence. L'architecture encapsulée au modèle client-serveur de la plupart des classes assurent la sécurité des accès, ainsi cette facilité de communication n'est pas dangereuse pour le bon fonctionnement du moteur.

Conclusion

Abstract World Engine n'en est qu'a sa genèse car, conceptualiser la réalité pour l'automatiser tend à se complexifier au fur et à mesure de la compréhension humaine de l'univers.

Cependant, la structure numérique et les processus qui définissent AWE maintiennent un niveau d'abstraction suffisant pour s'adapter à la nouveauté.

Ainsi, AWE n'est pas un logiciel, mais une tentative de virtualisation du réel, ses limites sont donc celles de notre capacité d'abstraction.

Actuellement, beaucoup de concepts basiques ne sont pas encore implémentés, beaucoup de choses doivent être améliorées et ajoutées, notamment en ce qui concerne la gestion de la pensée.

Mais la structure restera la même car l'approche par les possibilités ne fait que représenter la définition numérique des notions métaphysique se fait de la réalité.

AWE peut s'apparenter à une germe qui va pousser lentement, la tâche est énorme et prendra certainement plusieurs vies, plusieurs siècles. AWE représente la possible réalisation de l'un des rêves les plus fous de l'humanité : celui d'égaler les dieux.

Arborescence du moteur

_agents/active/
contient les agents actifs sérialisés (enregistrés sur le disque)

_agents/passive/
contient les classes d'agents passifs

_environments/
contient les environnements sérialisés

_possibilities/
contient les possibilités (Handler* et Method*) programmeur

WorldLauncher.class
cette classe contient le main du programme, elle initialise la classe World, l'interface, ainsi qu'un environnement de base

world/
c'est dans ce répertoire que se trouvent les classes du moteur

world/World.class
cette classe contient des références vers les serveurs, l'environnement courant et les composants de l'interface

world/agents/
contient les classes des agents

world/agents/Activable.class
interface permettant à un agent passif d'implémenter une méthode cycle

world/agents/Agent.class
classe abstraite, mère de toutes les classes d'agent, contient le nom, la description, l'image, la position et la dimension

world/agents/AgentActive.class
classe abstraite des agents actifs, contient les tables des possibilités, l'état et la pile d'intentions

world/agents/AgentActiveEvolved.class
classe des agents actifs, contient une référence vers l'objet attributs et une méthode cycle qui appelle la méthode de pensée

world/agents/AgentActiveEvolvedAttributes.class
classe contenant les attributs (volonté, intelligence, etc…) des agents actifs évolués

world/agents/AgentPassive.class
classe abstraite des agents passifs

world/agents/AgentPassiveStructure.class
classe abstraite des agents passifs de type structure

world/agents/Extendable.class
interface permettant à un agent passif d'occuper plusieurs positions

world/environment/
contient les classes relatives à la gestion de l'environnement

world/environment/Environment.class
cette classe représente l'environnement dans lequel évoluent les agents, elle dispose notamment d'un attribut Map et d'un synchroniseur

world/environment/EnvironmentConstants.class
cette classe contient un ensemble de constantes relatives aux lois physiques

world/environment/map/
contient les objets relatifs à la gestion de l'espace

world/environment/map/Map.class
cette classe représente un espace tridimensionnel, elle dispose d'un tableau de MapPoint

world/environment/map/MapPoint.class
cette classe représente une position dans l'espace

world/environment/pathfinder/
contient les classes relatives à la gestion de l'algorithme « A* »

world/environment/pathfinder/Pathfinder.class
cette classe contient l'algorithme de recherche de chemin par coût « A* »

world/environment/pathfinder/PathNode.class
cette classe est utilisée par Pathfinder.class, elle représente un nœud

world/environment/synchronizer/
contient les classes relatives à la synchronisation du moteur

world/environment/synchronizer/Synchronizer.class
cette classe s'occupe de la synchronisation entre les agents, l'environnement et l'affichage

world/possibilities/
contient les classes de possibilités desquelles héritent les possibilités utilisateur

world/possibilities/Handler.class
classe abstraite représentant la partie variables d'une possibilité. C'est la mère de toutes les parties Handler des possibilités

world/possibilities/HandlerAction.class
classe abstraite représentant des Handlers d'action, hérite de Handler, mère des parties Handler des possibilités d'action

world/possibilities/HandlerBehaviour.class
classe abstraite représentant des Handlers de comportement, hérite de Handler, mère des parties Handler des possibilités de comportement

world/possibilities/HandlerGenetic.class
classe abstraite représentant des Handlers de génétique, hérite de Handler, mère des parties Handler des possibilités de génétique

world/possibilities/HandlerMinding.class
classe abstraite représentant des Handlers de pensée, hérite de Handler, mère des parties Handler des possibilités de pensée

world/possibilities/HandlerPerception.class
classe abstraite représentant des Handlers de perception, hérite de Handler, mère des parties Handler des possibilités de perception

world/possibilities/Method.class
classe abstraite représentant la partie code d'une possibilité. C'est la mère de toutes les parties Method des possibilités

world/possibilities/MethodAction.class
classe abstraite représentant les Methods d'action, hérite de Method, mère des parties Method des possibilités d'action

world/possibilities/MethodBehaviour.class
classe abstraite représentant les Methods de comportement, hérite de Method, mère des parties Method des possibilités de comportement

world/possibilities/MethodGenetic.class
classe abstraite représentant les Methods de génétique, hérite de Method, mère des parties Method des possibilités de génétique

world/possibilities/MethodMinding.class
classe abstraite représentant les Methods de pensée, hérite de Method, mère des parties Method des possibilités pensée

world/possibilities/MethodPerception.class
classe abstraite représentant les Methods de perception, hérite de Method, mère des parties Method des possibilités de perception

world/possibilities/Parallelable.class
interface permettant à une possibilité de s'exécuter en même temps que les autres

world/servers/
contient les serveurs d'AWE

world/servers/ServerAgents.class
cette classe se charge de la gestion des agents actifs et passifs

world/servers/ServerEnvironment.class
cette classe se charge de la gestion des environnements

world/servers/ServerPossibilities.class
cette classe se charge de la gestion des possibilités

world/utils/
contient des classes utilitaires diverses

world/utils/Dimension3D.class
cette classe décrit une position dans l'espace, elle est utilisée par les agents et l'environnement

world/utils/Position3D.class
cette classe décrit une dimension dans l'espace, elle est utilisée par les agents et l'environnement

world/utils/priorityQueue/PriorityQueue.class
cette classe décrit une structure de type pile avec des notions de priorités, elle est utilisée par le Pathfinder