Programmation Orientée Objet et Génie Logiciel

Rapport du projet

SIMULATEUR EXTENSIBLE MULTI-AGENTS

Irénée Briquel, Léonard Gérard, Boris Golden

Vendredi 13 mai 2005

SOMMAIRE

Introduction et présentation du projet

I) Sema, le package de simulation

1) Structure globale et hiérarchie des classes

2) Réalisation, problèmes rencontrés et choix effectués

II) Agents et modélisation

1) Element et problèmes de visibilité et d'héritage

2) La structure extensible d'un agent : action, selector et satisfactor

3) Modélisation de comportements et d'aptitudes

III) L'interface utilisateur

1) Fonctionnalités et ergonomie de l'interface

2) Gestion de l'affichage du monde

3) Synchronisme avec le temps simulé

4) Les fichiers SIM

Conclusion et synthèse des points forts

Annexe 1 : tutorial pour la création d'un monde

Annexe 2 : syntaxe des fichiers SIM

Annexe 3 : diagrammes UML

INTRODUCTION

Notre projet est baptisé Simulateur Extensible Multi-Agents (**SEMA**). Notre programme permet la **création de mondes** où les lois, comportements d'agents et conditions initiales sont prédéfinies, et de visualiser et d'interférer avec leur évolution dans le temps, et ce de façon souple et aisée.

L'environnement est limité à **deux dimensions**, et modélisé par une grille rectangulaire de cases carrées. Néamoins, il est possible de tenir compte de la hauteur, et les agents peuvent évoluer dans un espace continu. **L'échelle de temps est continue**, et utilise une gestion **événement-centrée** : toute évolution est décomposée en évènements datés instantanés. Le monde est caractérisé par la donnée d'un moteur de simulation, d'une carte, d'une classe nature gérant les rapports des agents avec le monde et des classes définissant chaque type d'agent.

Nous offrons **une interface graphique conviviale** pour visualiser l'évolution de la simulation en direct, disposant d'un zoom, d'un contrôle de l'écoulement du temps, de champs d'information sur le monde, les cases et les individus. Le monde est vu du dessus, **sans soucis de graphismes élaborés** puisqu'il ne s'agit pas d'un jeu. En revanche, l'utilisateur pourra à tout moment interagir avec le monde, par la souris, en déplaçant et réorientant des agents, ou en modifiant les champs d'information qui sont éditables lorsque cela est pertinent, permettant des évolutions tant libres qu'orientées.

Concernant la création des mondes, notre but est de fournir une structure aboutie pour la gestion globale du monde, ainsi qu'une hiérarchie de classes de base pour la construction d'un modèle. Les objectifs sont d'offrir un **maximum de souplesse, de simplicité et d'économie de programmation**. Pour cela, un cadre de travail clair mais non limitatif doit permettre de créer un monde sans connaître tout le code existant, en réutilisant tout ce qui est utile, et sans être expert en Java. L'utilisateur est invité à ne pas modifier les classes d'éléments existantes, mais à en hériter pour redéfinir ses propres agents, et élargir ainsi sa bibliothèque. Enfin, tout a été fait pour cacher au maximum les détails techniques internes et ne fournir à l'utilisateur qu'un cadre de simulation agréable: nous avons encapsulé et cloisonné le code pour que l'utilisateur puisse modifier facilement certaines propriétés (par exemple, n'autoriser qu'un seul individu par case) sans avoir à digérer notre code. Un effort est accompli pour veiller à l'accessibilité des différents champs, et à l'utilisation pertinente des champs *public*, *private* et *protected.*

Nous ne souhaitons pas créer nous même des agents évolués, mais plutôt **fournir un cadre de simulation** qui rende la programmation très rapide et facile. Nous avons cependant implémenté de base plusieurs fonctions essentielles. La partie pleinement "intelligente" de l'agent reste cependant à la charge de celui qui veut créer son monde!

Nous nous efforçons de rendre **la structure de SEMA claire et simple**, ceci pour favoriser son extensibilité. Il est facile à l'utilisateur d'identifier quelle classe correspond à quoi, et alors rapide de la redéfinir. Ainsi, un utilisateur pourra sans difficultés travailler avec un temps discret.

Dans l'ordre, nous détaillerons :

- Le package Sema de simulation, qui regroupe ce qui concerne la simulation de l'espace et du temps. A priori, ces classes ne demandent pas à être modifiées, sauf usage particulier .

- La partie modélisation , qui comprend tout ce qui modélise les éléments et leurs interactions avec le monde. C'est le niveau le plus important pour la création d'un monde. C'est ici que l'utilisateur devra hériter des classes existantes pour créer les siennes.

- Enfin l'interface utilisateur, et les fonctionnalités mises en place.

I) SEMA, LE PACKAGE DE SIMULATION.

1) Structure globale et hiérarchie des classes.

Le package ***SEMA*** concerne la partie purement simulation. C'est donc le noyau du projet, la partie qui a demandé le plus de réflexion, tant pour la hiérarchie des classes que leur contenu : il faut rester généraliste et souple tout en fournissant une structure solide pour la modélisation et la simulation de comportements multi-agents, mais aussi garder à l'esprit que la modification d'une de ces classes par héritage doit rester aisée pour l'utilisateur.

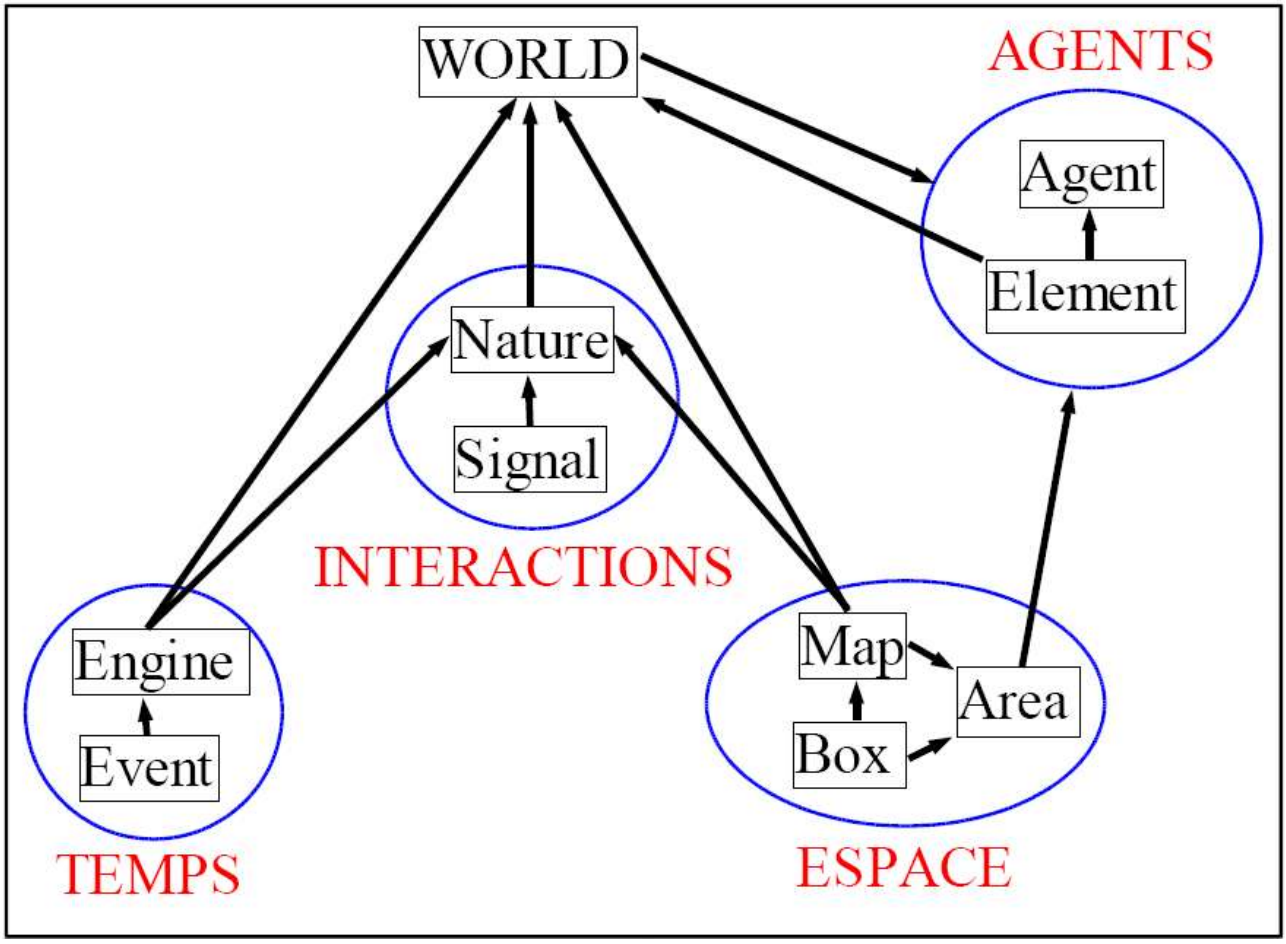


Diagramme des classes constitutives de Sema

La simulation proprement dite couvre quatre domaines:

- LE **TEMPS** : simulé à travers des événements ponctuels ***Event***, gérés par un moteur temporel ***Engine*** qui trie les événements et fait office d'horloge.

- L'**ESPACE** : simulé par l'intermédiaire de cases ***Box***, qui sont les constituants de base du terrain. Le terrain lui-même est géré par la carte ***Map***, qui est subdivisée en un ensemble de cases. Une ***Area*** est une portion quelconque de la carte, utilisée pour donner une forme à un élément du monde ou pour déterminer une zone, par exemple pour la diffusion d'un signal.

- LES **INTERACTIONS** passant par le monde : s'opèrent par le biais de signaux ***Signal***, et sont gérées par la ***Nature***. Il est à remarquer que les autres interactions c'est à dire celles directes entre les éléments sont gérées directement entre eux.

- LES **AGENTS**: domaine le plus important et le plus évolutif, présenté dans la partie II. Un ***Element*** est un agent primitif (ou *objet*), alors qu'un ***Agent*** est un agent basique du monde.

La classe ***World*** est la classe de plus haut niveau qui assure la cohésion de la simulation en regroupant temps, espace, interactions et agents. Une *simulation* correspond donc à une instanciation d'une classe ***World*** avec les paramètres propres à la simulation. L'objet obtenu est indifféremment désigné par monde, modèle, modélisation ou simulation.

2) Réalisation, problèmes rencontrés et choix effectués.

Le modèle épuré présenté dans la partie précédente ne rend pas compte des difficultés conceptuelles que nous avons eu, ni des choix qu'il nous a fallu faire ou des motivations d'une telle structure. Nous allons donc faire un petit compte-rendu des nos choix et des problèmes rencontrés durant la réalisation du temps, de l'espace, des interactions, et du monde lui-même.

a) Le temps.

L'idée est de **déconnecter totalement le temps réel du temps simulé** (le moteur ne s'occupe que du temps simulé). Après avoir hésité avec une approche centralisée donnant la main successivement à chaque agent en l'informant du temps courant, nous avons finalement choisi **une vision événement-centrée** (un événement est une méthode à exécuter, associée à une date).

Tout ce qui est effectué dans le monde passe par un enregistrement préalable auprès du moteur temporel. L'écoulement du temps simulé se fait de façon discrète, à travers les événements, c'est-à-dire que l'on saute un intervalle de temps ne contenant aucun événement (l'écoulement est discret, mais l'interface graphique gère son écoulement par rapport au temps réel). Mais le temps est bien continu, un événement a lieu a une date arbitrairement précise (codé sur un *double*).

Après un long débat, choix d'un *TreeMap* pour gérer la liste des événements du moteur. Il faut en effet une structure de données qui trie les éléments, avec une insertion et une obtention du premier élément rapides. Dans le *TreeMap*, on associe à un instant *t* la pile (*Stack*) d'événements de date *t*. Cela a l'avantage de mettre au même niveau les événements ayant lieu au même instant (fait normalement rare de part la randomisation subie par chaque individu) et d'interdire un rafraîchissement graphique entre les événements qui ont lieu au même instant (mais qui sont traités en série par le moteur : cohérence et apparente simultanéité).

Le moteur est en fait le processeur du monde virtuel, et c'est lui qui donne la main à chaque individu désirant effectuer une action. Un élément crée ses propres classes internes d'événement héritant de la classe ***Event***, ce qui permet à l'événement effectué par un certain élément d'avoir accès à tout l'individu. La seule vraie limitation de notre moteur est le fait que les événements simultanés sont gérés en série et qu'un événement ayant une durée non nulle est exécuté en un temps nul dans le monde simulé, ce qui oblige à fractionner un événement long en plusieurs sous événements (cf action de l'agent), et à attendre après avoir effectué un événement un certain temps, pour lui donner une durée.

La seule méthode pour palier à ce problème aurait été de gérer les individus en parallèle par des threads, mais le problème de cette démarche est qu'elle lie temps de calcul et temps simulé, sans compter les conflits lors d'accès commun à une case par exemple, ou encore les imprécisions de la gestion du temps en utilisant des threads, voire même, à un certain niveau, une discrétisation du temps, car entre deux cycles processeurs, on ne peut insérer une infinité d'événements, etc. **Nous estimons donc avoir choisi la solution la plus adaptée à notre but**, à savoir simuler en parallèle des agents.

b) L'espace.

La carte du monde est constituée d'une **matrice de cases carrées** de côté connu en unités spatiales simulées, et l'on peut affiner le niveau de détail du monde (au niveau de la détection des collisions, notamment) avec un facteur de subdivision de chaque case, appelé boxfactor, ou granulosité, dont la fonction précise est détaillée dans l'annexe 3. L'espace en lui-même n'est pas difficile à simuler: il est en 2D et rectangulaire. Le caractère rectangulaire est naturel à l'écran d'un ordinateur et ne constitue pas vraiment une limitation du point de vue de la simulation. Les limites de l'espace sont invisibles.

L'idée initiale des cases était de faciliter les déplacements et de n'autoriser qu'un seul agent par case, au plus. Nous nous sommes rendus compte par la suite que cela constituait une limitation importante, différentes alternatives ont été évoquées et nous avons finalement utilisé les cases comme une simple table de hashage : tous les éléments s'enregistrent automatiquement auprès des cases qu'ils occupent (càd les cases que leur forme intersecte), et donc, il est aisé, en passant par les cases, de savoir quels éléments se trouvent dans une zone donnée de la carte. Les cases doivent être en nombre suffisant mais pas trop: s'il y en a trop, cela ralentit la simulation, car les calculs case-basés ont beaucoup de cases à analyser, mais s'il n'y en a pas assez, on doit traiter beaucoup éléments à chaque fois que l'on détecte les collisions dans une zone donnée (par exemple), puisque le faible nombre de cases augmente le nombre d'éléments détectés dans une zone donnée. Une fois les cases utilisées pour connaître les éléments dans une zone, on peut bien sûr affiner la détection en regardant, pour chaque élément, s'il se trouve effectivement dans la zone: les cases ne sont qu'un biais pour limiter le nombre d'éléments à traiter (il serait stupide de calculer une collision éventuelle entre deux fourmis distantes de dix mètres!).

L'autre fonctionnalité des cases est le dessin du terrain du monde (chaque case dessine la portion du terrain sur laquelle elle se trouve). Les cases peuvent également être utiles pour créer des obstacles: en effet, une case peut refuser la venu d'un individu. **La seule vocation des cases est donc une pseudo discrétisation de l'espace utile pour les collisions ou pour la diffusion de signaux (analogie avec une table de hashage)**. Dans l'absolu, on pourrait imaginer un monde où les cases ne serviraient à rien (les éléments gèreraient les collisions eux-même), mais la complexité de la gestion des collisions serait alors quadratique en le nombre d'éléments (alors qu'elle peut être linéaire si on met un nombre de cases adéquat). En pratique, le plus petit individu d'une simulation doit prendre quelques cases (entre 1 et 10 pour obtenir de bonnes performances).

Nous avons crée des ***Area*** qui sont des zones du terrain que l'on peut faire évoluer facilement dans le monde, l'idée étant qu'elles permettent de donner simplement forme à un agent. En effet la gestion de l'individu en termes de cases et d'éventuelles collisions se fait automatiquement dans l'***Area*** de l'agent et est donc souvent transparente pour l'utilisateur.

c) Les interactions.

A qui est-ce de gérer la détection d'obstacle, les lois physiques, etc? L'idée est que l'on peut suivre certaines règles généralistes gérées par la nature et par les cases, mais que les agents doivent pouvoir avoir des comportements particuliers (penser en termes de souplesse de code).
Notre simulateur reste dans une **vision agent-centrée**, et c'est donc à priori chaque agent qui gère ses interactions. Cependant, il est parfois nécessaire de **centraliser certaines interactions** au niveau de la nature, qui s'occupe de diffuser les signaux, ou de la vision. Cela permettrait, par exemple, de changer du point de vue de la nature les capacités de vision des individus quand il y a du brouillard. Typiquement, dans ce cas, les fonctionnalités sont partagées entre l'agent et la nature: l'agent doit pouvoir voir et la nature doit considérer que l'agent peut voir pour qu'au final, l'agent voit. **Cette dualité retranscrit bien la réalité du monde terrestre: il faut une volonté de l'agent, accompagnée d'une possibilité matérielle de réalisation de l'action.**

La plupart des interactions se font par le biais de signaux, qui utilisent des ***Area*** pour indiquer la zone où ils peuvent être diffusés. Notre nature diffuse le signal de manière instantanée, mais l'on peut très bien créer une nouvelle nature (par héritage) qui diffuse le signal pas à pas.

d) La structure générale de Sema et le monde.

Le monde regroupe toutes les classes de la simulation. Il s'occupe aussi de fournir un ID unique à chaque élément pour l'identifier.

Au niveau de la structure générale de la simulation, on peut la résumer ainsi:
le temps et l'espace sont deux domaines fondamentaux qui ne dépendent d'aucune autre classe. Par dessus, on rajoute la couche gérant les interactions, puis la couche monde. Enfin, on ajoute les agents. Le monde est donc un temps simulé, un espace simulé, des interactions, et une liste d'agents du monde.

La structure de la simulation, si elle peut paraître limpide et simple, a demandé un travail énorme de réflexion et d'expérimentation. **Nous pensons avoir réussi à rester généralistes tout en fournissant un package fonctionnel et structuré**.

Nous avons aussi été amenés à faire certains choix de simulation, mais en aucune façon ces choix ne sont définitifs: ils peuvent être changés par l'utilisateur par héritage, ce qui montre bien le caractère extensible de notre simulateur.

II) Agent et Modélisation.

1) Element et problèmes de visibilité et d'héritage.

L'élément est l'entité primitive du monde, qui peut servir à modéliser aussi bien des agents que des objets, ou toute autre entité susceptible de se trouver dans le monde. Pour être créé, l'élément doit avoir connaissance du monde dans lequel il va évoluer (on lui passe donc un objet ***World*** comme premier argument). Toute chose du monde peut donc être castée en un ***Element*** (typiquement, un agent ou un objet).

Chaque élément se voit attribuer par le monde **un ID unique** qui permettra, si besoin est, de l'identifier parmi les autres éléments. Un élément possède également une forme dans le monde, qui définit l'espace qu'il occupe (cette forme est une ***Area***), une hauteur (même si le monde est 2D, la hauteur peut servir pour des calculs primitifs de volume ou de vision d'obstacles), ainsi qu'un type qui est une chaîne de caractères permettant d'identifier une catégorie à laquelle il appartient, et une image qui permet de matérialiser graphiquement cet élément (en interne, on gère les priorités d'affichage des éléments pour éviter les bugs d'affichage). Un élément peut mourir ou être tué, mais également être pris par un autre (ce qui peut s'avérer utile dans le cas d'objets) ; ceci se fait par accord de l'objet en fonction du déclencheur de l'action.

Tout élément hérite dans une classe interne ***ElementEvent*** de la classe ***Event***, ce qui lui permet de définir **un événement généraliste** qu'il pourra poser par la suite pour planifier des tâches. Enfin, l'élément possède des méthodes qui permettent d'afficher dans l'interface certaines de ses propriétés et de les modifier depuis l'interface, ce qui permet d'interagir en direct avec les éléments du monde au cours d'une simulation (cf. partie III).

La structure d'un élément se veut modulable, mais doit offrir de base ce qui est important pour un élément du monde. La partie plus active et comportementale se trouve dans l'agent.

En termes de visibilité des méthodes et champs d'un élément, nous avons effectué un travail important. En effet, il faut que l'on puisse hériter d'un ***Element*** sans perdre certaines de ses caractéristiques (attention au modificateur *private*), tout en ne rendant pas visible certaines méthodes internes à l'agent (attention au modificateur *public*). On utilise **des accesseurs pour tous les champs** (ce qui permet une plus grande flexibilité à l'héritage) ; les champs sont donc tous privés. Concernant les méthodes, les méthodes internes à l'agent sont protégées et les méthodes qui peuvent être visibles par d'autres agents sont publiques. Certaines méthodes qui sont appelées par l'interface (par exemple, la méthode de dessin de l'élément ou les méthodes pour lire ou écrire des propriétés depuis l'interface) doivent être visibles hors package et sont donc accessibles de l'extérieur, car déclarées publiques.

La réflexion qui a accompagné la création de la classe ***Element*** assure **une flexibilité totale**, tout en minimisant la visibilité des champs restreints (pour éviter d'éventuelles erreurs de programmation). Lorsque nous avions un doute, nous avons déclaré les méthodes publiques pour une plus grande souplesse. Rien n'empêche l'utilisateur de rendre publiques certaines des méthodes protégées pour le besoin de sa simulation, toujours par héritage.

2) La structure d'un agent : action, selector et satisfactor.

L'agent hérite d'***Element***, car c'est un élément du monde. Par défaut, il randomise son délai de 2% pour donner par une diversité de réactivité parmi les mêmes agents. La spécificité de ***Agent*** est surtout **sa capacité à gérer des actions** : il va par exemple avoir l'action marcher en direction de tel point, ou l'action scruter, pour voir s'il n'y a pas telle personne dans son entourage physique, etc.

De part le caractère discret des événements, ce genre d'action demande une sorte d'événement auto-régénérant, qui correspond à l'étalage d'une action sur une longue durée avec une série d'événements. Cela peut être codé, bien sûr, avec les événements qui, quand ils s'exécutent, rajoutent un futur événement, mais la gestion globale des différentes actions d'un agent n'est pas évidente, la mise en pause ou l'arrêt de certaine action non plus.

Pour faciliter tout cela, nous avons créé l'objet ***Action***, qui a pour but d'encapsuler tout l'aspect continu d'une action et toute la gestion de son interaction avec les autres.

Pour encore plus faciliter le tout, l'agent a **une fonction centrale de décision** pour arrêter, mettre en pause, etc, les actions en cours. Cette fonction s'appelle *acceptAction* et prend en argument l'action qui veut continuer de s'effectuer. Cette fonction va être appelée à chaque fois qu'une action veut effectuer un de ses pas élémentaire. La gestion globale des actions au niveau de l'agent est aussi assurée par une fonction d'enregistrement (*registerAction*) et de désenregistrement (*unregisterAction*) des actions, appelées à la création et à la destruction de l'action. Par défaut, ces fonctions ne font rien, mais on pourrait imaginer qu'elles permettraient de savoir qu'est-ce qui est en cours et donc d'avoir des comportements différents avec mémoire et conscience des actions.

Pour gérer les actions, on peut les mettre en pause, les reprendre, les démarrer, les stopper. L'action en elle-même encapsule ces ordres ordonnés par l'extérieur de l'action et les applique sans discuter. Nous avons pendant un certain temps hésité à la possibilité de l'action de refuser mais cela rendait le code de l'intelligence trop compliqué. De plus, il y a une possibilité de contourner cela, comme nous le verrons plus loin.

Pour créer une action précise, il suffit donc d'hériter d'Action et de redéfinir les fonctions que l'on veut :

- à chaque changement d'état, une fonction redéfinissable est appelée; par exemple, à la mise en pause, *whenPause* est appelée. Ces fonctions ne sont pas des pas de l'action, mais juste du code à effectuer pour permettre la mise de l'action dans l'état demandé. En effet puisque l'action ne peut pas refuser ce changement d'état, elle doit s'adapter.

- ensuite il y a tout ce qui est considéré comme le code d'exécution de l'action : les différents pas. Le principal est défini par la méthode *step*. Ces fonctions renvoient si elles veulent faire continuer l'action ou pas et si oui dans combien de temps effectuer le pas suivant.

- finalement une fonction qui permet la destruction de l'action est aussi redéfinissable. C'est par celle-ci par exemple que l'on peu contourner l'obligation de stopper l'action, en en relançant une autre par exemple à ce moment là.

Nous nous sommes rendus compte que pour gérer la plupart des actions, l'agent avait besoin d'être capable de sélectionner des individus parmi les autres, d'où la notion de ***Selector***, qui est un objet avec une méthode qui détermine si un certain élément est parmi l'ensemble que l'on veut sélectionner. Il est également souvent utile de savoir si une condition donnée est satisfaite, d'où la création du ***Satisfactor***, qui est une classe encapsulant une méthode booléenne sans arguments, et qui exprime le caractère satisfait ou insatisfait d'un but de l'agent (typiquement: être rassasié).

A partir de là, il était clair que l'on pouvait rendre les choses faciles au niveau des actions pour permettre de faire des actions à la suite, avec des conditions, ce qui a donné le ***ActToSatisfaction*** qui permet, avec l'encapsulation de toutes les fonctions (et une action avec une condition d'arrêt, un futur en cas de réussite et un futur en cas d'échec), de faire agir l'agent pour satisfaire un but.

Nous avons donc crée une structure solide et généraliste qui permet à l'agent de répartir ses événements dans le temps et avec cohérence, ou de suivre un but, l'aspect technique de la chose étant géré totalement et de façon transparente en interne, et la souplesse étant toujours privilégiée.

III) L'INTERFACE UTILISATEUR.

1) Fonctionnalités et ergonomie de l'interface.

L'interface utilisateur est conçue pour permettre de visualiser, dans les meilleures conditions, l'évolution d'une simulation. Pour cela, il est naturel d'avoir une **zone d'affichage** de l'espace simulé, avec **zoom** et **déplacement de la zone visible de la carte**. La disposition de la carte du monde au centre de la fenêtre est logique ; la gestion claire du zoom et une **zone d'aperçu** globale (permettant de se repérer sur la carte) vont de paire et sont donc regroupées. Le bouton **fit,** pour donner une vision globale de la carte et adapter le zoom en conséquence, et le bouton **default,** pour revenir au zoom par défaut proposé par la simulation, se révèlent souvent utiles pour changer rapidement d'échelle d'observation. Le choix de deux moyens complémentaires pour régler le zoom, avec un slider ou une zone de saisie, est nécessaire pour proposer des bornes visibles à l'utilisateur et un réglage visuel et intuitif, par le slider, mais également de la précision, avec la zone de saisie. Les valeurs extrêmes du zoom sont indiquées dans le fichier SIM (cf III-4). Globalement, on peut d'ailleurs définir les **paramètres par défaut de l'interface dans le fichier SIM**, ce qui apporte une grande ***souplesse*** et permet de personnaliser et d'***optimiser, pour chaque simulation, l'interface***.

Pour la gestion du **facteur temporel** (ou **time factor**), qui permet de corréler le temps (virtuel) de la simulation et le temps réel, on procède comme pour le zoom. Cependant, borner la simulation à un écoulement temporel fixé est peu pratique pour des utilisateurs cherchant à voir quelque chose émerger de la simulation à partir d'un certain temps écoulé, et qui souhaitent donc aller le plus vite possible à une date donnée ; d'où l'utilité d'une fonction **gotodate** qui permet de faire un saut dans le temps virtuel. La pratique nous a aussi vite montré l'utilité d'une fonction de **pause**, d'avancement pas à pas **next date,** et, pour atteindre les limites de notre simulateur, d'une fonction **boost** qui ne lie plus les temps et utilise toutes les ressources du système. Après ce dernier point, la volonté de savoir quel facteur temporel était effectivement atteint s'est faite sentir : la zone **achieved** donne donc le facteur temporel effectivement atteint par la simulation et indique par une coloration rouge que l'écart à la valeur demandée par l'utilisateur est trop important (supérieur à 5%).

Tous les détails de gestion du temps et de l'espace réglés, l'intérêt de voir **les propriétés des individus, des cases et du monde,** à tous les niveaux de la simulation, reste un point primordial dans une interface utile et donc informative. Le choix des propriétés à afficher est faite par l'objet en question ; par exemple, chaque élément dans le monde peut donner à afficher certaines propriétés (celles-ci n'apparaissent que si l'élément est sélectionné par l'utilisateur), ce qui apporte une grande souplesse. Les propriétés du monde sont affichées en permanence, et la liste des propriétés à afficher est modifiable par l'utilisateur en modifiant nos classes par héritage ; par exemple, si le moteur veut afficher l'événement en cours, il le peut par l'intermédiaire du monde (qui par défaut regroupe les propriétés de la carte, de la nature et du moteur en plus de celles de la simulation). L'édition de certaines propriétés étant quelque chose de très puissant, l'implémentation en a été décidée, toujours dans le modèle agent-centré : c'est l'objet en question qui accepte ou pas la nouvelle valeur que l'on essaye de lui donner ; typiquement, un agent que l'on veut positionner hors de la carte risque de refuser ce déplacement non pertinent. Un détail important à l'utilisation : l'édition des propriétés, entre autres, met en **pause temporaire** la simulation, pour avoir une édition cohérente et possible qui n'interfère pas avec la simulation. L'affichage des propriétés des éléments et des cases nécessite naturellement un **système de sélection à la souris** sur la carte : une fois sélectionné, on peut obtenir les informations de l'élément ou la case. Quelques facilités d'utilisation y sont associées, comme la possibilité de **suivre l'élément** sélectionné ou de **centrer la sélection** (case ou élément, pour pouvoir zoomer facilement dessus, par exemple), mais aussi **de marquer** un certain nombre d'individus, pour les observer plus facilement parmi les autres.

L'affichage est donc puissant, complet et fonctionnel, mais en contre partie, cela a un coût en terme de complexité, et il n'est clairement pas toujours nécessaire de mettre à jour l'affichage 20 fois par seconde pour suivre correctement une simulation. D'où la possibilité de **régler les délais entre deux rafraîchissements**, en temps réel et/ou en temps virtuel. L'interface alors obtenue permet une observation complète d'une simulation avec toutes les facilités usuelles d'un logiciel ; le menu permet de gérer les simulations sur lesquelles on travaille, avec **l'ouverture d'un fichier** et **le reset**, et une aide sur l'interface est disponible à tout moment dans le menu **Help.**

La dernière fonctionnalité importante pour faire un logiciel complet et puissant est la possibilité d'éditer graphiquement un monde ; en effet, dans la pratique de la création des mondes, l'aspect rebutant de passer par la syntaxe écrite du fichier de simulation nous a incité à faire **un mode d'édition** de notre interface, ainsi qu'un mini **wizard** pour régler l'ensemble des propriétés d'une nouvelle simulation. Le mode d'édition permet aussi de rajouter des éléments au monde, par l'utilisation d'un bouton **Add element**, mais aussi de copier un élément déjà présent sur la carte par **Duplicate element**, ce qui évite d'avoir à réfléchir à « quelle ligne de code entrer pour créer une nouvelle fourmi ». Ensuite, **le positionnement et la rotation**, bien que possibles par les propriétés, étaient naturels **à la souris**, ce que nous avons rendu possible. C'est très utile pour choisir la disposition initiale des agents dans le monde, chose extrêmement rebutante à la main. Le positionnement et la rotation sont même **autorisés pendant une simulation,** en déverrouillant l'option **Locked for mouse**. La dernière possibilité à offrir à l'utilisateur pour lui permettre d'avoir autre chose comme case que celle par défaut, est l'option **Modify box**. Finalement, la création automatique du fichier de simulation (cf III-4) par une fonction **save** et la possibilité de lancer directement la simulation en cours d'édition par un **run** sont les derniers points pour rendre le logiciel convivial et totalement fonctionnel.

Pour conclure, notre interface propose de suivre le déroulement d'une simulation avec une cohérence temporelle, de façon simple et conviviale, avec de multiples options utiles, mais aussi d'interagir en direct avec la simulation en consultant/modifiant ses propriétés et celles de ses agents, de pouvoir faire une utilisation naturelle de la souris en sélectionnant, déplaçant et tournant les éléments, mais aussi de créer un monde à partir de l'interface, ce qui rend beaucoup plus accessible la création d'un monde. Il nous paraît utile de préciser que tous les éléments de l'ergonomie ont aussi été pensés pour **les faibles résolutions**, avec les ascenseurs et les barres pour régler les différentes tailles des zones d'information. Les boutons ont été **regroupés par fonctionnalité** et la structure globale de l'interface répond aux besoins de l'utilisateur qui veut lancer une simulation. Enfin, tous les paramètres de l'interface utilisateur modifiables à l'écran ont des accesseurs publics, ce qui pourrait permettre d'**englober Sema dans un autre logiciel** (par exemple, un créateur complet de simulation, avec édition des comportements des individus en pseudo-code, qui pourrait être un projet futur...).

2) Gestion de l'affichage du monde.

Comment gérer l'affichage des cases et des éléments? Les cases sont à voir comme un morceau inerte du terrain, qui constitue l'image de fond du monde et SUR lesquelles on pose les éléments. On dessine donc d'abord les cases, puis les éléments. Auparavant, on délimite graphiquement l'espace simulé par un fond blanc.

Pour la gestion de l'ordre des éléments, cela est fait par la classe ***Drawing*** du package ***Sema***. C'est cette classe qui s'occupe du graphisme du monde. Par défaut, il faut bien « fixer » l'ordre d'affichage des éléments les uns par rapport aux autres, pour ne pas avoir de problèmes de scintillement lorsqu'il y a chevauchement des éléments. Cet ordre est appelé la **priorité d'affichage**, et c'est une donnée interne à l'élément (quand deux éléments ont même priorité, ils sont départagés par leur ID, numéro unique de l'élément dans un monde).

Ensuite pour l'efficacité de l'affichage, seules les cases nécessaires sont dessinées. Pour ce qui est des éléments, sachant qu'ils n'ont pas de zone définie d'affichage, le problème restait entier. La première approche utilisée fût un choix simplificateur : dessine-toi si ton centre de gravité est dans la fenêtre. Cet affichage était d'une complexité dépendant du nombre d'éléments exclusivement, ce qui était certes intéressant. Mais ne considérer que le centre de gravité provoquait des « bug » d'affichage : disparition de certains éléments par le zoom ou disparition soudaine en bordure de la fenêtre...

L'autre vision que nous avons finalement utilisée après des tests est la **vision case-centrée** : chaque élément ayant la charge de s'enregistrer dans les cases ou il est, ce qui nous permet de récupérer l'ensemble des éléments qui ont une partie dans la fenêtre d'affichage en fusionnant l'ensemble des sous-listes d'éléments de chaque case affichée.

Nos tests ont montré qu'avec l'utilisation de *TreeSet*, cette fusion n'était **pas plus coûteuse**, même en fine granulosité (ie beaucoup de cases par éléments pour une optimisation de la détection des collisions). Nous avons là une complexité ne dépendant que du nombre de cases visibles, ce qui offre un gain de complexité confortable lorsque l'on zoome, par rapport au parcours linéaire de tous les éléments (avec le test du centre de gravité), et qui n'est pas moins rapide, même pour afficher l'ensemble de la carte. **Ce sont ces résultats expérimentaux et de multiples tests qui ont finalement motivé ce choix**.

Par-dessus tout cela, **le monde peut dessiner des choses,** par l'intermédiaire de sa fonction draw. Typiquement, il peut dessiner du brouillard ou de la pluie. En résumé, les couches d'affichage sont : effacement de l'affichage précédent et remplissage en blanc de l'espace simulé, dessin des cases, dessins des éléments avec gestion des priorités d'affichage, puis dessin éventuel par le monde de phénomènes particuliers. **Le dessin du monde s'accompagne toujours du rafraîchissement des propriétés affichées** dans l'interface et de la mise en évidence de l'élément sélectionné (par effet de transparence), lorsqu'il y en a un.

3) Synchronisation entre l'affichage et la simulation.

L'interface et le graphisme constituent la partie externe de l'interface, mais, en parallèle, **il faut gérer le déroulement de la simulation**! C'est le coeur de l'interfaçage des threads de simulation et de graphisme. Voici, en pseudo-code, la méthode utilisée (remarque : une subtilité a été masquée au niveau de la fonction *OnDoitRafraîchir()*, pour ne pas précalculer trop d'événements du moteur et devoir attendre alors trop longtemps pour maintenir la cohérence d'écoulement du temps).

```
threadSema() =

    TantQue(Simulation en cours) {      /*thread de la simulation actif*/

        BloqueL'Affichage {             /*phase de précalcul des événements*/
            TraiterEvenementSuivantDuMoteur();
        }

        Si OnDoitRafraîchir() {
            Attendre(DuréePourRespecterLeFacteurTemporel);
            DessinerLeMonde();  /*thread du graphisme de Java*/
        }

    }
```

OnDoitRafraîchir() = /*détermine s'il est temps de redessiner pour respecter les contraintes*/

durée processeur depuis le dernier rafraîchissement > DélaiRéelMaximum
ou
durée simulée depuis le dernier rafraichissement > DélaiVirtuelMaximum

Le blocage mutuel de l'affichage et du moteur est nécessaire, car l'interaction avec l'utilisateur est dans un thread à part et donc, à tout moment, un ordre d'affichage peut être lancé. Pour assurer une cohérence et éviter des erreurs lors d'accès simultanés, **les deux threads ne sont donc jamais actifs en même temps**.

La phase d'attente avant le rafraîchissement n'a pas lieu lorsque le facteur temporel est trop élevé pour la puissance processeur disponible.

Ce code étant au coeur de la simulation, nous l'avons particulièrement soigné **pour le rendre rapide** (réflexion au niveau des types utilisés pour optimiser le temps de calcul, de l'utilisation de variables, etc.) Cette méthode donne d'excellents résultats en termes d'écoulement du temps simulé dans le temps réel, mais nous a demandé **de nombreux tests** avant d'être validée.

4) Les fichiers SIM.

Pour rendre notre logiciel agréable et convivial, nous avons crée un type de fichier d'extension SIM. Un fichier SIM contient toutes les informations nécessaires à la création d'une simulation. On peut notamment y définir les classes de base que l'on utilise [pour le moteur, le dessin, la carte, la nature et le monde (si l'on a redéfini ces classes par rapport à SEMA.base, ce qui n'est pas le cas dans une simulation basique)], les caractéristiques de la carte (càd les cases et les dimensions du monde), les éléments du monde avec leurs propriétés, et d'autres paramètres de simulation utiles pour l'interface (constantes pour les slides, etc).

Les fichiers SIM peuvent être soit générés automatiquement par SEMA, soit crées à la main (pour la syntaxe, cf l'annexe). L'intérêt d'un fichier SIM réside dans le fait que l'**on peut sauvegarder facilement un monde crée avec l'éditeur graphique de Sema**, et créer une nouvelle simulation sans compilateur Java si l'on a sous la main les classes compilées que l'on souhaite utiliser.

En effet, on aurait très bien pu imaginer une classe SIM qui contienne toutes les informations de la simulation, et qu'il faille éditer pour modifier le monde, mais il faudrait alors recompiler à chaque fois le fichier en question! Ce n'est pas l'esprit de Sema, qui, une fois des comportements définis et les classes correspondantes compilées, doit être **utilisable sans compilateur Java**.

La démarche typique de création d'un monde (expliquée en annexe dans le tutorial pour la création d'un monde), repose sur une première partie de réflexion-héritage-programmation, suivie ensuite de la création de différentes simulations auxquelles correspondent un fichier SIM à chaque fois. Un exemple tout simple: une fois défini l'agent Fourmi, on peut définir facilement une fourmilière avec un fichier SIM. Cette même classe Fourmi pourra être incorporée dans un fichier zoo.sim qui simule un zoo, sans que l'on ait besoin de recompiler quoi que ce soit.

Du point de vue de la réalisation technique, la création d'un **lecteur/écriveur de fichiers SIM** a été laborieuse, tant pour la définition d'une syntaxe riche mais naturelle que pour la mise en oeuvre en terme de programmation. Pour rendre les fichiers SIM vraiment puissants, on peut y définir des variables, faire référence à des classes de la simulation précédemment instanciées, etc.

Dans l'interface utilisateur de Sema, on peut générer automatiquement un fichier SIM à partir du monde en cours d'édition, ce qui est une fonctionnalité extrêmement pratique, en particulier pour positionner les élémens dans le monde (emplacement et orienation avec la souris, cf mode édition).

Les fichiers SIM constituent donc **une fonctionnalité fondamentale** pour la création facile d'un monde qui rend notre logiciel complet et utilisable.

CONCLUSION

Nous avons atteint les buts les plus optimistes que nous nous étions assigné, puisque nous avons débouché sur un logiciel complet comprenant un simulateur abouti, des modélisations basiques de comportements d'agents, une interface graphique, tant pour la visualisation que l'édition de cartes avec fichiers générés, et disposant d'une documentation complète en anglais.

Pour ce qui est de l'utilisation, **les performances atteintes par notre simulateur en terme de vitesse sont très satisfaisantes,** et n'ont donc pas vraiment souffert du caractère généraliste. Il est facile de privilégier, suivant la taille de la simulation, la vitesse de calcul ou la complexité des comportements. **Pour l'interface graphique, nous avons implémenté tout ce que nous voulions** (zoom, sélection à la souris, écoulement du temps, raccourcis clavier, champs modifiables, etc).

Les exemples fournis montrent bien qu'il est simple de créer et d'éditer un monde, et la lecture de fichiers SIM donne à SEMA toute sa puissance, et son évolutivité, puisque elle permet de redéfinir toutes les classes pertinentes.

Au final, nous sommes satisfaits, mais **ce projet s'est avéré très ambitieux**. Il a demandé un travail considérable, tant en matière de réflexions théoriques sur la modélisaiton la plus souple possible que pour la structure générale du projet, pour le contenu d'un fichier de simulation, ou pour une programmation extensible facilement. L'interface graphique a aussi constitué un gros travail, sans compter l'écriture du code elle-même.

Pour un projet demandant à son utilisateur de programmer lui-même ses simulations, et de bien comprendre SEMA pour éventuellement le faire évoluer, **il était essentiel de rendre le code lisible et compréhensible**. Cela a également demandé une réflexion essentielle sur l'utilisation des différents champs protected, private et public, pour cacher tout ce qui est inutile de connaître, et empêcher, autant que possible, les conflits. Nous avons en outre été confrontés au problème de la création d'agents aux caractéristiques diverses (par exemple mobile et communiquant), et ce **SANS héritage multiple**; l'utilisation de classes internes qui avait été initialement choisie a en fait été abandonnée, car il s'est avéré préférable de réimplémenter chaque caractère pour tous les agents, pour permettre l'interaction de ces différentes facultés (la vision et le déplacement, par exemple).

LES POINTS FORTS DE SEMA?

Notre projet est **un vrai projet**, avec des idées de fond derrière et une vraie théorie. Il a nécessité des choix importants, et nous avons souvent réfléchi pour faire ceux qui nous semblaient les plus judicieux. Il est naturellement et **fondamentalement orienté objet**, la réutilisabilité du code étant indispensable. Sa **structure hiérachisée, claire et mûrement réfléchie** est totalement pensée dans l'optique de la création aisée de mondes.

L'interface utilisateur a été particulièrement soignée, avec une interface graphique fonctionnelle et aboutie, un **lecteur et écriveur de fichiers de simulation**, et un mode édition qui nous a semblé indispensable pour un confort d'utilisation maximal et la **création/édition graphique d'une simulation**.

L'intégralité du code est authentique, et n'a pas fait l'objet de réutilisations, à l'exception de la classe FileFilter (classe anecdotique utilisée pour n'afficher que les fichiers SIM dans la fenêtre de sélection de cartes). **Les idées du projet et la théorie de modélisation sont aussi personnelles**, et correspondent à notre vision propre, mais se veulent évidemment réalistes et généralistes.

Enfin, nous nous sommes astreints à une discipline de programmation pour uniformiser le code : nous avons opté pour **une utilisation de l'anglais uniquement**, pour les noms de classes, variables et méthodes, mais aussi pour la documentation javadoc qui a été soignée, et qui donnera éventuellement à notre projet, à notre connaissance **inédit**, un second souffle, au-delà de POOGL...

Une *dernière* remarque: **le but de notre projet n'est *pas* de créer des mondes**, mais bien de donner la possiblité d'en créer facilement. Nos réalisations personnelles sont donc **un plus** pour valoriser notre projet, et non une fin en soit. Elles doivent surtout servir à **tester notre logiciel et son interface.** La phase de modélisation de comportements vivants dépasse donc largement le cadre de notre simulateur, qui représente déjà un travail énorme. Ils nous a cependant paru intéressant d'inclure de base quelques comportements types pour faciliter la création d'un monde, et initier ainsi la création par l'utilisateur d'une bibliothèque d'agents.

ANNEXE 1 : TUTORIAL POUR LA CRÉATION D'UN MONDE

Pour créer un monde, il vous faut:

-ce tutorial
-la documentation Javadoc du package ***Sema***
-des bases en Java et un compilateur Java
-quelques images sympas pour matérialiser les agents à l'écran (les images doivent être tournées vers la droite, pour signifier que l'individu regarde à droite lorsqu'il a un angle nul).
-des idées et un peu de motivation

Avec tout ça, il est possible de créer à peu près n'importe quel monde, sans avoir à regarder notre code: c'est bien le but de notre logiciel!

La création d'un monde se déroule en plusieurs étapes:

1) Redéfinition facultative de certaines classes structurelles de ***Sema***.

Etape la plus rare, elle consiste à hériter des classes du package ***Sema*** pour les redéfinir et profiter ainsi de la flexibilité de notre système. Typiquement, un utilisateur souhaitant modifier notre moteur à événements pour rendre le temps discret dans sa simulation va hériter de la classe ***Engine*** et la redéfinir, ou redéfinir la gestion des signaux dans ***Nature***. Pour hériter et connaître les propriétés des classes de ***Sema***, il suffit de consulter la documentation Javadoc. La seule contrainte est que les constructeurs des classes filles doivent prendre comme premiers arguments les mêmes que ceux du constructeur des classes mères, dans le même ordre, cela pour la bonne raison que ces arguments sont automatiquement insérés par le lecteur du fichier SIM, et pour préserver une cohérence dans les dépendances entre classes.

La méthode la plus simple et la plus rapide correspond à l'utilisation de l'environnement de simulation fourni avec l'interface utilisateur de base sans rien y modifier.

2) Programmation de nouveaux agents modélisant des comportements ou des aptitudes.

L'utilisateur peut définir de nouvelles classes pour les agents, en héritant des classes ***Element*** (plutôt pour créer un objet) ou ***Agent***. L'implémentation la plus simple d'un nouveau comportement se fait en définissant de nouvelles actions pour l'agent (cf la sous-classe ***Action*** de ***Agent***, partie III) et en appelant une première action à l'instanciation de l'objet. Attention, les classes héritant d'un élément ou d'un agent ont des contraintes sur leur constructeur : il doit prend comme premier argument le monde pour être reconnu comme valide par notre simulateur!
La classe ***NewAgent*** du package ***Modelling*** peut fournir une base pour créer un nouvel agent.

Si on veut faire des cases particulières (case puits, case nourriture, etc), on crée un objet "puits", ce qui est plus logique, puisque la seule vocation des cases est une discrétisation de l'espace utile pour les collisions (ou encore, pour la diffusion de signaux).

3) Création d'un fichier de simulation .SIM.

Il va regrouper toutes les classes à utiliser et tous les paramètres du monde, ainsi que la nature et la disposition des éléments/agents dans le monde. Pour créer ce fichier SIM, il y a deux moyens:

- **le mode automatique**, en passant par l'interface: on crée une nouvelle simulation en indiquant les propriétés du monde, les classes à utiliser pour le simulateur, puis on place les agents un par un après les avoir instanciés. Pour instancier un agent, il suffit de cliquer sur « Add Element » dans l'interface en mode édition, puis d'entrer la ligne de commande avec la syntaxe suivante:

className(type1 arg1, type2, arg2, ..., typeN argN)

si le constructeur de l'agent était: *public className(World world, type1 arg1, ..., typeN argN)*,
en remplaçant évidemment arg1...argN par leur valeur pour instancier cet agent. Pou plus de détails, consulter l'annexe 3 sur la syntaxe d'un fichier SIM.
- **le mode manuel**, qui est expliqué dans cette même annexe 3, et qui, bien que plus technique, peut permettre de définir grossièrement mais plus rapidement un monde (mais **le positionnement des agents est plus rapide en passant par l'interface** et en utilisant la souris, puis le générateur automatique).

Une fois compilés, les fichiers .class utilisés dans un fichier SIM doivent être placés dans un **répertoire connu par Java** (java/lib par exemple) pour que Sema puisse les instancier. Une bonne pratique est de réunir toutes les nouvelles classes en un seul package.

Après la théorie, la pratique: voici un exemple concret de **création d'une colonie de fourmis,** partant à la recherche de nourriture et rapportant le tout à la fourmilière, en prenant soin de marquer le chemin suivi avec des phéromones.

La phase 1 de redéfinition d'éléments structurels étant inutile, on passera directement à la phase 2, comme ce sera le cas pour la plupart des simulations. On créera les agents suivants:

1) **La fourmi**, héritant de ***PheromoneUserAgent***, auquel on ajoutera
   - la variable privée « réserve de nourriture » initiée à zéro.

   - l'action **« chercher de la nourriture »**, dont l'action récurrente sera de chercher de la nourriture dans les cases autour (grâce à la fonction ***getElements*** de ***Nature***), s'il y en a commencer l'action « collecter »; sinon chercher une phéromone de message « nourriture », et commencer une action « suivre chemin vers la nourriture » ; sinon avancer au hasard, et déposer une phéromone « retour ».

   - l'action **« collecter »**, prenant une nourriture en argument, dont l'action récurrente est d'appeler les fonctions *canEat* et *eaten* de la nourriture, et ajoutant en conséquence ce qu'elle collecte à sa variable « réserve de nourriture ». Se finit quand il n'y a plus rien a manger, et appelle alors l'action « retour fourmilière ».

   - l'action **« retour fourmilière »** cherchant si la fourmilière est à côté, commançant alors une action « déposer nourriture »; cherchant la phéromone suivante marquée « retour » et la suivant pas à pas, déposant une phéromone « nourriture ».

   - l'action **« suivre le chemin vers la nourriture »,** cherchant si de la nourriture se trouve à proximité, et commençant alors l'action « collecter »; sinon cherchant la phéromone suivante marquée « nourriture » et la suivant pas à pas. Déposant une phéromone « retour ».

   - enfin l'action **« déposer nourriture »**, la fourmi vidant sa réserve de nourriture et en remplissant d'autant la réserve de la fourmilière; puis entamant une action « chercher de la nourriture ».

   On initialisera dans son constructeur la fourmi avec une action « chercher de la nourriture ».

2) **La phéromone**, agent déjà défini, immobile, à durée de vie limitée, porteur d'un message.

3) **La nourriture**, héritant simplement de ***EatableAgent*** et dont le nom d'espèce *kind* sera nourriture.

4) **La fourmilière**, qui aura une variable interne « réserve de nourriture » initiée à zéro et une méthode « recevoir de la nourriture », augmentant en conséquence sa réserve de nourriture et par exemple se faisant grossir grâce à la methode rescale de son aire obtenue par *getArea()*.

Tous ces agents seront initialisés avec une aire circulaire : la classe ***CircularArea***.

Reste alors à **créer la carte**. La phase à la main est alors inutile pour nos besoins. On ouvre donc l'éditeur pour demander une nouvelle carte, de taille par exemple 100 * 100, avec des cases de longeur 1.

On utilise le bouton **Add Element** pour ajouter une fourmilière comme indiqué dans le tutorial, en prenant soin de choisir l'image (la déposer dans le même dossier que la carte) et l'aire.

De même, on ajoute une fourmi, et un objet nourriture.

Ensuite, on utilise le bouton **Duplicate Element** pour ajouter autant de fourmis et nourritures que souhaité. On répartit alors harmonieusement les nourritures sur la carte et on place toutes les fourmis dans la fourmilière, le tout avec la souris.

La simulation est alors prête. On pourra la faire évoluer en empêchant les superpositions de fourmis, ou en ajoutant d'autres paramètres, comme leur faim...

ANNEXE 2 : STRUCTURE D'UN FICHIER SIM.

Ce qui suit explique en détails les possibilités multiples d'un .SIM et la syntaxe s'y rattachant.

Dans un fichier SIM, une ligne commençant par un / est considérée comme un **commentaire**.

Tout d'abord, on définit des **balises**, à raison d'une balise par ligne, avec la syntaxe "BALISE= valeur". Les balises définissent certaines propriétés de la simulation. Les balises utilisables sont indiquées plus bas. Toutes les balises sont facultatives, et elles reçoivent une valeur par défaut si elles ne sont pas redéfinies. **Les balises les plus importantes** pour un monde basique sont XMBOXES, YMBOXES, BOXFACTOR, MBOXLENGTH. Si la balise n'est pas reconnue, elle est lue comme NAME=fichierImage, et l'on essaye de lire une image dans fichierImage (par défaut, dans le répertoire du .SIM si le chemin d'accès n'est pas précisé), et l'on mémorise l'association avec NAME. Plus tard, on pourra faire référence à cette image dans le fichier SIM en écrivant: Image NAME. Exemple: FOURMI= c:\\fourmi.gif. Plus tard, en argument d'un élément demandant une image, on pourra lui passer Image FOURMI. Remarque: on ne respecte pas la casse pour les noms des balises!

Puis, **on définit la liste des cases**, précédée de la marque "BOXES:". Ici, on fait une distinction subtile: en fait, à ce niveau, **ce sont des mBoxes que l'on définit** (métacases), qui seront par la suite subdivisées en BOXFACTOR² cases (l'idée étant que ce **BOXFACTOR** permet de changer la granulosité spatiale des cases **sans avoir à redéfinir entièrement la carte**!). La longueur d'une case dans le monde sera donc de MBOXLENGTHBOXFACTOR, de même, on aura XBOXES= BOXFACTOR * XMBOXES, etc. L'idée importante à retenir est que l'on donne la possibilité d'**affiner une carte** par ce BOXFACTOR, et que, donc les données relatives aux cases peuvent évoluer avec le BOXFACTOR.

Ici, on peut opter pour deux modes de saisie :

En mode d'entrée par défaut, la liste des cases est facultative. On procède ainsi :

- Soit on met la marque "BOXES:" suivi de la liste des cases (une par ligne) avec la syntaxe: "x,y=boxClass(..)", où x,y désigne les **coordonnées entières de la case** (la case supérieure gauche a pour coordonnées (0,0)), et boxClass(...) est la définition de la classe avec les paramètres qui permettront de l'instancier (la syntaxe pour la définition de boxClass(...) et les contraintes sur boxClass sont explicitées plus bas). **Les cases non définies sont initialisées à la case par défaut** DEFAULTBOX.
- soit on ne met ni marque "BOXES:", ni cases (et toutes les cases sont alors initialisées à la case par défaut DEFAULTBOX).

En mode de saisie matricielle (mode activé par la balise "MATRIXINPUTMODE= on"), on définit, après la marque "BOXES:", une suite de **raccourcis** "#raccourci= boxClass(...)", (avec boxClass(...) qui suit aussi la syntaxe de définition d'une classe), puis une matrice de raccourcis, matrice dont les dimensions doivent être XMBOXES de large et YMBOXES de haut (représentation très visuelle de la carte), les raccourcis la composant étant séparés par un espace.

Dans tous les cas, une fois que la marque "BOXES:" est posée, **on ne peut que définir des cases**, jusqu'à tomber sur la marque "ELEMENTS:".

Enfin, **on définit les éléments** du monde à partir de cette marque "ELEMENTS:", obligatoire dans un .SIM valide. Suivent, toujours à raison d'un part ligne, les définitions des éléments, éventuellement aucun. On définit un élément de la façon suivante: "className(...)". Pour

plus de détails sur cette syntaxe, voir le paragraphe qui suit et qui explique précisemment comment faire appel à une classe depuis un fichier SIM. Dans la liste des éléments, on peut définir des variables de la forme "#VARIABLE=elementClass(...)" pour pointer par la suite vers cet élément (le nom de variable est insensible à la casse, comme les balises). L'élément est alors immédiatement crée dans le monde, et l'utilisation de la variable "VARIABLE" précédée du mot-clef "selftype" dans la suite du fichier .SIM **fait référence à cet élément** (cela peut être utile pour définir des agents "élèves" qui prennent en argument leur chef commun, exemple détaillé ensuite). Ainsi, à la création d'un monde, on peut donner un élément en argument à un autre élément, ce qui peut s'avérer utile dans certains cas.

Syntaxe pour la définition d'une classe:

classname(type1 arg1, type2 arg2,...), où type$_i$ est le type du ième argument et arg$_i$ le ième argument. Typiquement, pour instancier une classe, il suffit donc de copier la première ligne de son constructeur, en enlevant « public » et en remplaçant les noms des paramètres par leur valeur! (La syntaxe est donc **calquée sur la définition d'une méthode ou d'un constructeur**, pour plus d'intuitivité et permettre à l'utilisateur ne maîtrisant pas la syntaxe d'un fichier SIM de s'en sortir).

classname est le nom complet de la classe (*java.lang.String*, par exemple). type$_i$ peut être un type de base de Java (*float, int, double, boolean, short, long, byte, char*), ou *String*, ou Image (qui sera suivi, au choix, du chemin du fichier dans lequel se trouve l'image, ou du nom d'une image précédemment définie dans les balises par IMAGE=fichier), ou selftype (les explications suivent), ou bien un type d'un objet quelconque suivi de son instanciation avec la syntaxe précédemment décrite. Ainsi, on peut imaginer des déclarations de la forme: "Object Exception(String "hello")" qui instancie une exception et la caste en objet.

***selftype* est un type particulier** qui indique que l'argument qui suit **fait référence à un élément précédemment instancié** avec #var=class.

Dans le cas du chef et des élèves, on définira donc : #chef=chef(String "Caniou"), puis : eleve(string "boris", selftype chef) si le constructeur de la classe élève prend comme argument le nom de l'élève et l'élément chef, et si la classe chef a un constructeur prenant en argument le nom du chef (le premier argument du constructeur est le monde mais n'a pas besoin d'être précisé ici). Si par la suite, on déclare d'autres éléments, comme eleve(string "léonard", selftype chef) ou eleve(string "irénée", selftype chef), les 3 élèves Boris, Irénée et Léonard seront 3 éléments du monde qui pointent vers **la même instance** de l'élément chef "Caniou".

Une fonctionnalité supplémentaire de selfType est que "selftype map" désigne **la carte du monde** (castée en sema.Map) et "selfType world" **le monde lui-même** (casté en sema.World), ce qui est utile pour instancier un élément qui prend une Area en argument, par exemple, puisque cette Area a besoin de map pour être instanciée. Cela constitue également une flexibilité pour d'éventuelles nouvelles classes de l'utilisateur qui auraient besoin de prendre le monde en argument.

Exemple évolué d'une définition de classe:
#ANT=sema.Agent(sema.Area sema.RectangularArea(selfType map, float 1, float 1, float 1, float 1)",float 1, Image FOURMI).

Ici, il est nécessaire de caster ***RectangularArea*** en ***Area***, car le constructeur de ***Agent*** prend en argument une ***Area*** (sinon, Java ne trouvera pas le constructeur de la classe ***Agent*** prenant un ***AreaRectangular*** en argument!).

"Image FOURMI" fait référence à l'image précédemment définie dans le .SIM par : FOURMI= c:\\fourmi.gif. Plus tard, si l'on veut faire référence à cet élément qui vient d'être instancié pour le passer en argument à un autre, il faudra écrire selftype ANT.

ATTENTION, il y a des contraintes sur les classes appelées dans un .SIM (normal, **on ne peut pas utiliser n'importe quelle classe pour un élément du monde ou une case Box**...).

Ces contraintes s'expriment en terme de constructeurs et d'héritage.

- ELEMENTS : le constructeur de la classe d'un élément (et donc d'un agent) **prend toujours le monde en premier argument** (argument **implicitement ajouté** par le lecteur de .SIM lors de la définition d'un élément par "element(type1 arg1,...)" entraîne : new element(world, arg1, arg2, ...)). De plus, la classe utilisée dans un .SIM pour instancier un élément **hérite nécessairement de sema.Element**.

- BOXES : le constructeur d'une case prend comme **4 premiers arguments** la carte *Map,* la classe de dessin *Drawing*, l'abscisse entière de la classe, l'ordonnée entière de la case. Encore une fois, ces arguments sont **implicites et ne doivent pas apparaître dans le fichier SIM**. Ainsi,

(x,y)=sema.Box(float 0, String "DefaultBox", String "box.jpg")

entraîne : new sema.Box(map, drawing, x, y, 0, "DefaultBox", "box.jpg"). De plus, la classe d'une Box **hérite nécessairement de la classe sema.Box**.

Il existe également des contraintes sur engine, drawing, map, nature et modelisation:
une classe 'class2' en remplaçant une autre 'class' (class=drawing, map, ...) **doit en hériter**, et posséder un constructeur dont les premiers arguments soient les mêmes que la classe d'origine (ainsi, pour créer un moteur qui rende le temps discret par pas de x unités temporelles,
on peut faire une classe ***discretEngine*** qui hérite de sema.Engine, et dont le constructeur prend en argument un flottant x (le pas temporel) ; on mettra alors la balise :

ENGINE=discretEngine(float 1)

dans un fichier SIM pour faire une simulation travaillant avec un temps discret (les éléments travaillant toujours sur un temps continue mais le moteur arrondissant leurs événements).

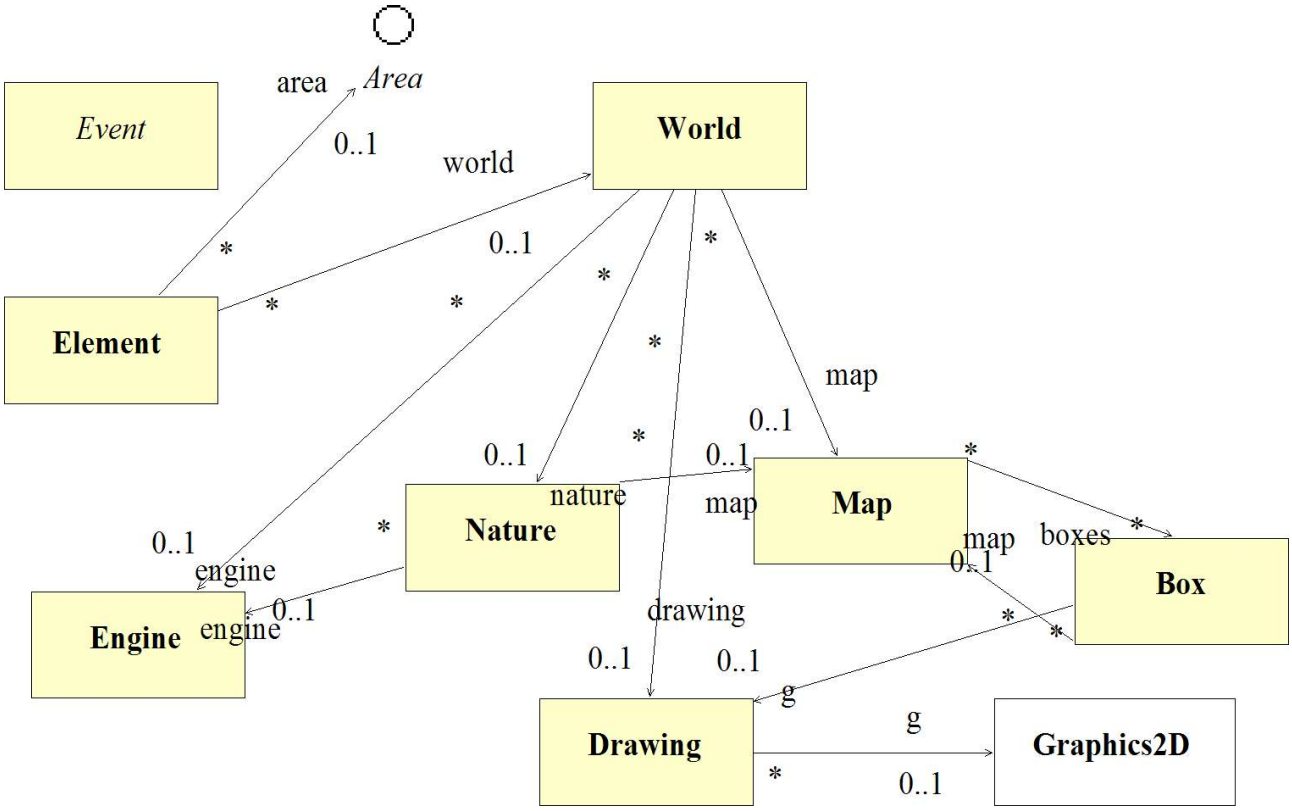
Pour plus de détails sur la création d'un monde, il faut se reporter au tutorial pour la création d'un monde, également en annexe.

Voici maintenant un **exemple d'un fichier de simulation**, avec liste exhaustive des fonctionnalités et des balises (les valeurs indiquées pour les balises sont celles par défaut lorsqu'elles ne sont pas précisées dans le fichier SIM):

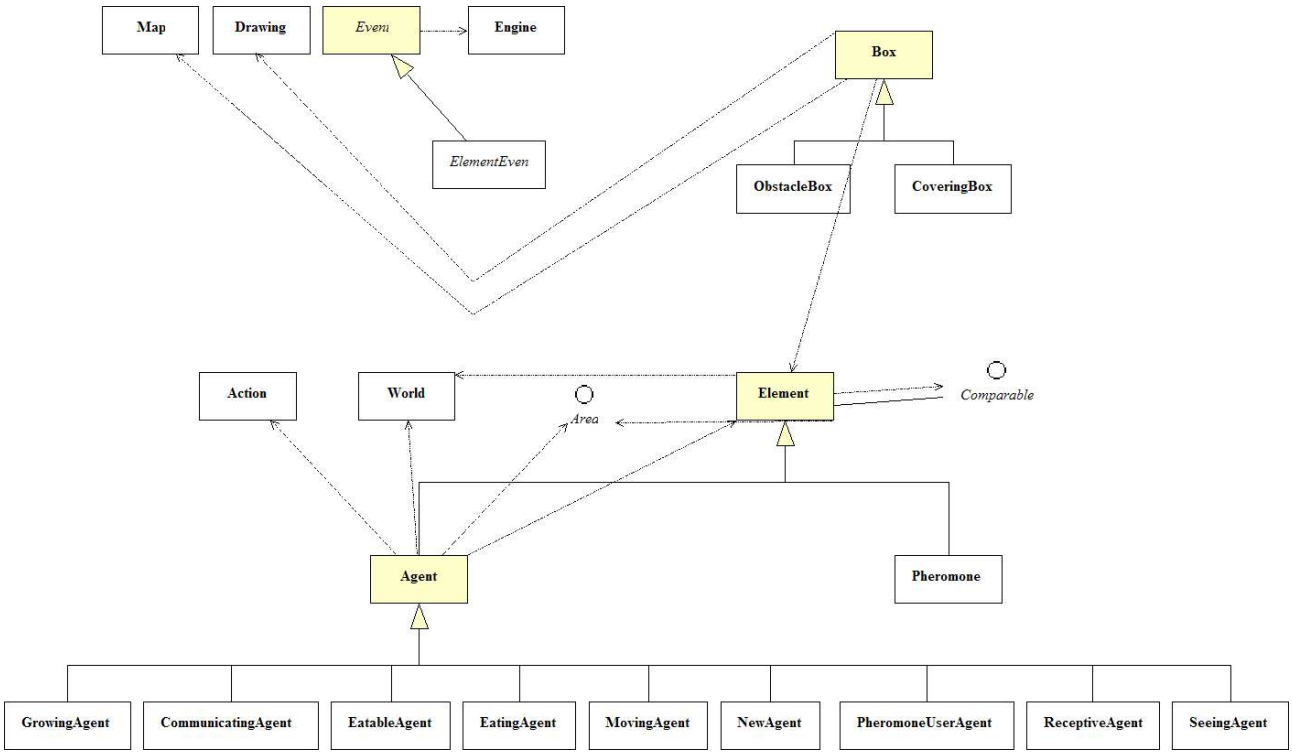
```
//informations sur la simulation
NAME= no name
COMMENTS= no comments

//classes utilisées pour le moteur, le dessin, la carte, la nature et la modélisation
ENGINE= sema.Engine
DRAWING= sema.Drawing
MAP= sema.Map
NATURE= sema.Nature
MODELISATION= sema.Modelisation

//propriétés de la carte, nombre de cases, granulosité, et dimensions
XMBOXES=10
YMBOXES=10
BOXFACTOR=1
MBOXLENGTH=1
BOXFACTOR= 1
```

```
//propriétés en rapport avec l'interface utilisateur et le graphisme
PIXELSBYMBOX=80
MINZOOM= 0
MAXZOOM= 10
MINTFACTOR= 0
MAXTFACTOR= 100
TIMEFACTOR= 1
ZOOM= 1
CENTERX= XBOXES*BOXLENGTH2
CENTERY= YBOXES*BOXLENGTH2
REALDISPLAYDELAY= 200
VIRTUALDISPLAYDELAY= 1000

//définition d'un raccourci vers une image
FOURMI= c:\\fourmi.gif

//Il y a deux options pour l'entrée des cases constitutives de la carte:

//Option 1: entrée des cases une par une lorsqu'elles diffèrent de la case par défaut

//case par défaut utilisée pour les cases qui ne sont pas explicitement définies
DEFAULTBOX= sema.Box(float 0, String "DefaultBox", String "")

BOXES:
1,1=sema.Box(float 0, String "dBox", String "")
0,0=sema.Box(float 0, String "dBox", String "")
2,2=sema.Box(float 0, String "dBox", String "")


//Option 2: entrée matricielle des cases
//il faut alors indiquer explicitement que l'on choisit l'entrée matricielle, par:
MATRIXINPUTMODE= on

//définition des cases de la carte
BOXES:
#b= sema.Box(float 0, String "DefaultBox", String "")
#i= sema.Box(float 0, String "dBox", String "")

i b b b b b b b b
b i b b b b b b b
b b i b b b b b b
b b b b b b b b b
b b b b b b b b b

//définition des éléments et agents du monde
ELEMENTS:
#ANT=sema.Agent(sema.Area sema.RectangularArea(selfType map, float 1, float 1, float 1, float
1)",float 1, Image FOURMI).
//ici, on nomme l'élément pour le réutiliser plus tard avec la syntaxe: seltype ANT
//exemple d'appel d'une variable avec selftype map qui fait référence à la carte du monde crée.

//Dans un fichier généré automatiquement par Sema, on trouve également les propriétés des
//éléments, des cases, et du monde.
```

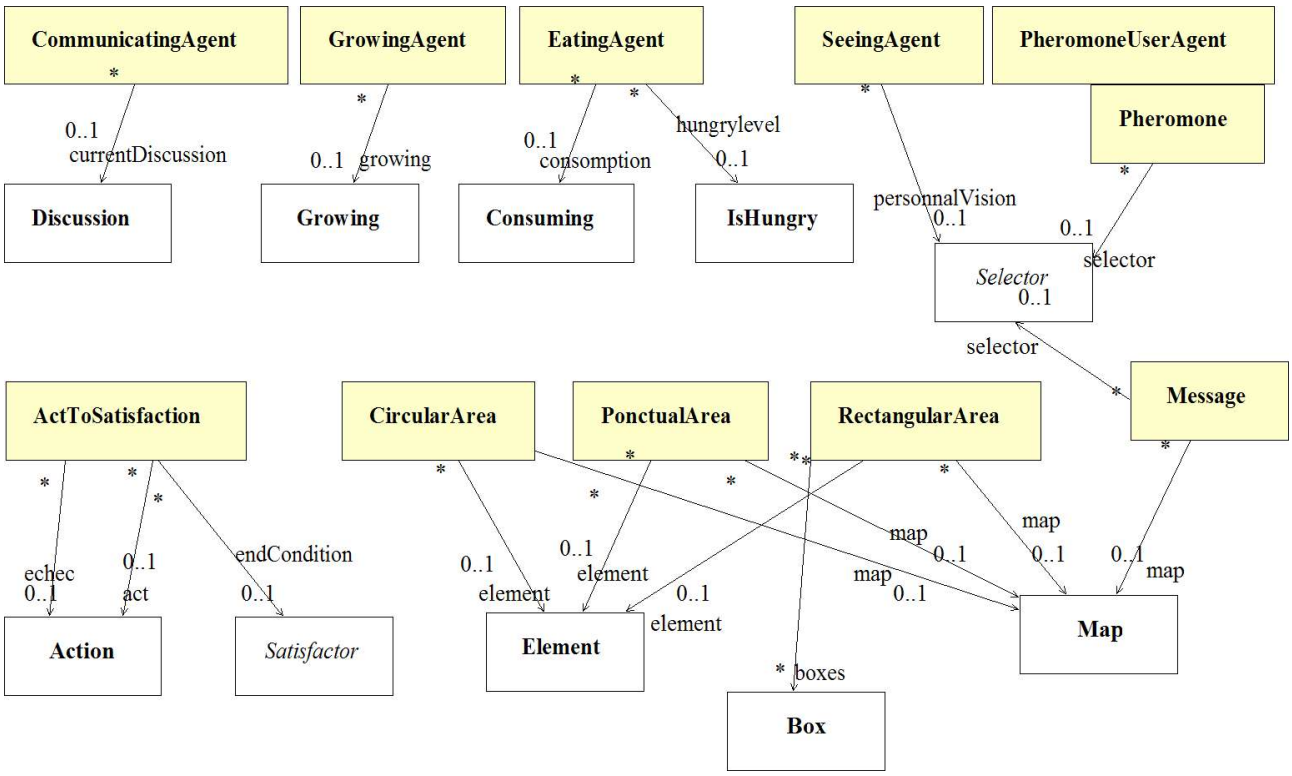
ANNEXE 3 : DIAGRAMMES UML.



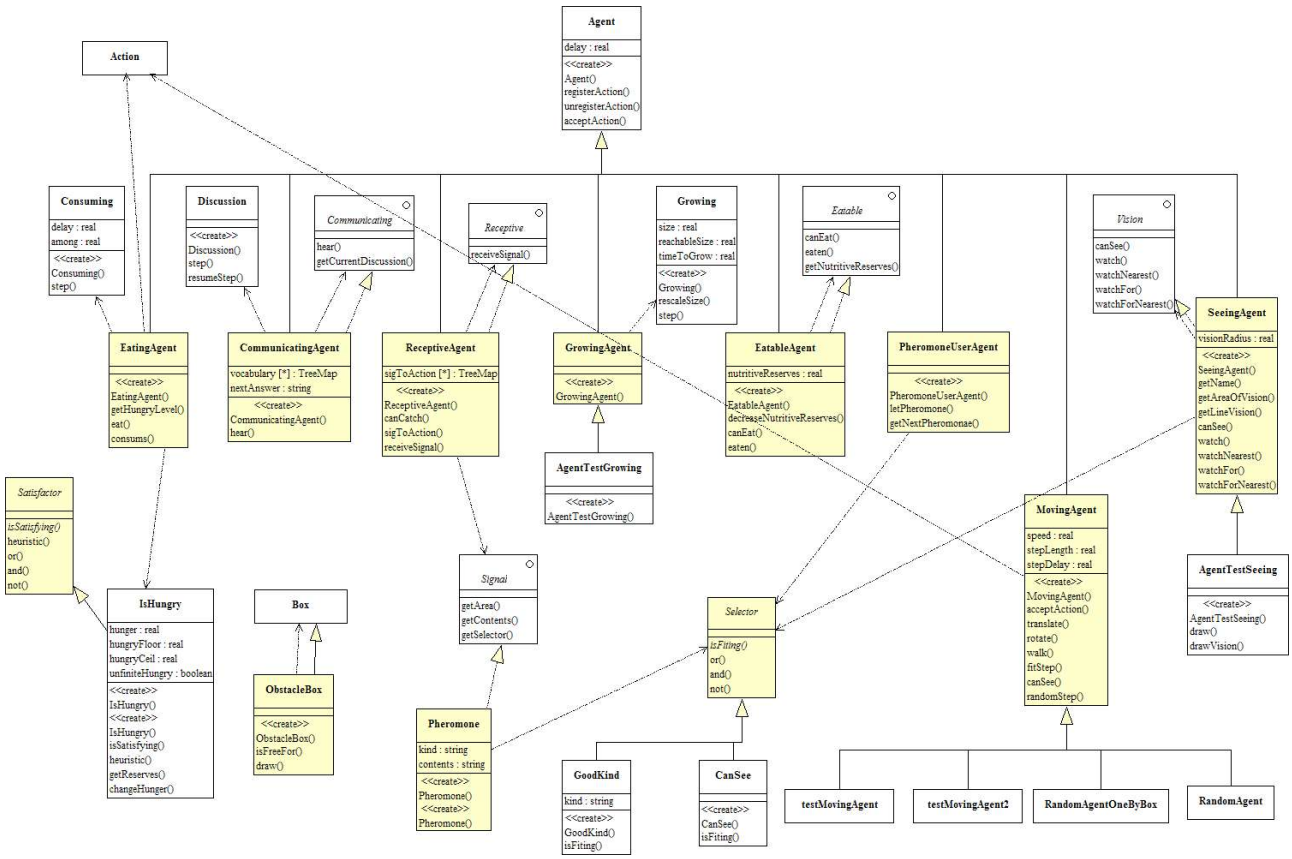
Structure du package Sema, responsable de la simulation



Classes héritant d'une classe de Sema



Modélisation et agents de base



Structure détaillée de la modélisation de base et des exemples