

MODULECO

Philippe Le Goff

12 juillet 2001

Option Informatique des télécommunications
ENST Bretagne, Technopôle Brest Iroise 29280 Plouzané

Résumé

Depuis un an les départements économique et informatique de l'ENST-Bretagne collaborent pour la réalisation d'un outil de simulation économique modulaire, flexible, évolutif. La première version de ce logiciel a déjà permis la simulation d'une grande variété de modèles. Afin de généraliser la notion d'interaction entre agents économiques, l'objectif de ce projet est d'intégrer dans le système de simulation moduleco la notion de marché - vue comme un composant de communication. Il faudra s'adapter au système existant, intégrer les composants de communication - leur gestion, leur activité -, et adapter les modèles économiques existants pour qu'ils fonctionnent dans la nouvelle version.

Table des matières

1	Introduction	4
2	Moduleco	4
2.1	Présentation	4
2.1.1	Définition d'un système multi-agent	4
2.1.2	Définition d'un agent	6
2.1.3	Quelques exemples	6
2.2	Architecture, modèle	10
2.2.1	le modèle de base	10
2.2.2	Les voisinages	10
2.2.3	Zone d'activation	10
2.2.4	Ordonnanceur ou scheduler	13
2.2.5	Représentation graphique	13
2.3	Les paquetages	13
2.3.1	modeleco	17
2.3.2	grapheco	18
2.3.3	medium	19
2.4	Paquetages de simulation	19
2.4.1	Classes de base	19
2.4.2	Adaptation des classes graphiques	19
3	Le Framework Medium	21
3.1	Présentation	21
3.2	Architecture générale du framework (D'après la documentation de Eric Cariou ENST Bretagne)	22
3.2.1	Connexion d'un composant à un médium	22
3.2.2	MediumsRegistry et objets associés	24
3.3	Architecture détaillée	25
3.4	Création de nouveaux médiums	25
3.5	Création d'une nouvelle couche de communication	26
3.5.1	Classe LocationAddress	26
3.5.2	Classe ConnectionManager	26
3.5.3	Classe Connection	28
4	Utilisation du Framework Medium dans Moduleco	29
4.1	Le monde et les agents	29
4.1.1	Introduction	29
4.1.2	La classe EAgent	29
4.1.3	La classe EWorld	29
4.1.4	Le Scheduler TimeScheduler	29
4.2	Le voisinage	33
4.3	Les graphiques	33
4.3.1	La classe Canvas	33
4.3.2	La classe Graphique	33

4.4	Le Marché virtuel	33
4.4.1	Le monde et les agents	34
4.4.2	Les entreprises	34
4.5	Le paquetage medium	34
4.6	Premiers tests	34
5	Conclusion	36

1 Introduction

Une des applications de l'informatique est de modéliser ou de vérifier des théories scientifiques. L'architecture classique basée sur des algorithmes séquentiels a modélisé beaucoup de choses mais elle ne résout pas tout.

Les plateformes multi-agent permettent de modéliser des comportements complexes car elle repose sur des fondements inspirés du vivant (interaction, coopération et évolution d'entités autonomes).

Moduleco est un prototype de plateforme multi-agent dont le but est de modéliser des marchés économiques. La première version, initiée par Denis Phan¹ et modélisée par Antoine Beugnard² au début de l'année 2000, a permis de mettre en place l'architecture et reproduire des modèles existants sur d'autres plateformes.

La deuxième version a vu le jour lors d'un stage de deuxième année auquel j'ai participé avec Frédéric Falempin et Camille Monge³ durant l'été 2000. De nouveaux modèles ont été implantés et l'architecture consolidée et généralisée.

La troisième version qui fait l'objet de mon projet de troisième année voit son architecture enrichie par le concept de médium ou composant de communication.

Ce document est composé de trois parties. Une première est consacrée à la présentation de la version actuelle du logiciel, une deuxième partie présente le concept de médium et la troisième partie présente l'intégration de ce concept dans moduleco.

2 Moduleco

2.1 Présentation

“Moduleco une plate-forme multi-agent à la convergence de plusieurs disciplines, conçue pour simuler les marchés et les organisations, les phénomènes sociaux et la dynamique des populations.” (figure 1)

2.1.1 Définition d'un système multi-agent

C'est une architecture logicielle qui permet de traiter de grands volumes et qui repose sur la coopération et l'interaction d'entités autonomes que l'on nomme agents.

A l'instar d'une colonie de fourmis, ce système permet de résoudre des problèmes complexes irréalisables par un seul individu (ou agent). En programmant un comportement individuel simple, nous voyons émerger un comportement global élaboré grâce à la collaboration et à l'interaction des agents.

Ce type de logiciel peut être utilisé soit pour résoudre un problème soit pour simuler un système. L'architecture repose essentiellement sur des concepts de programmation objet.

¹Département Economie ENST Bretagne

²Département Informatique ENST Bretagne

³Option CHMEST

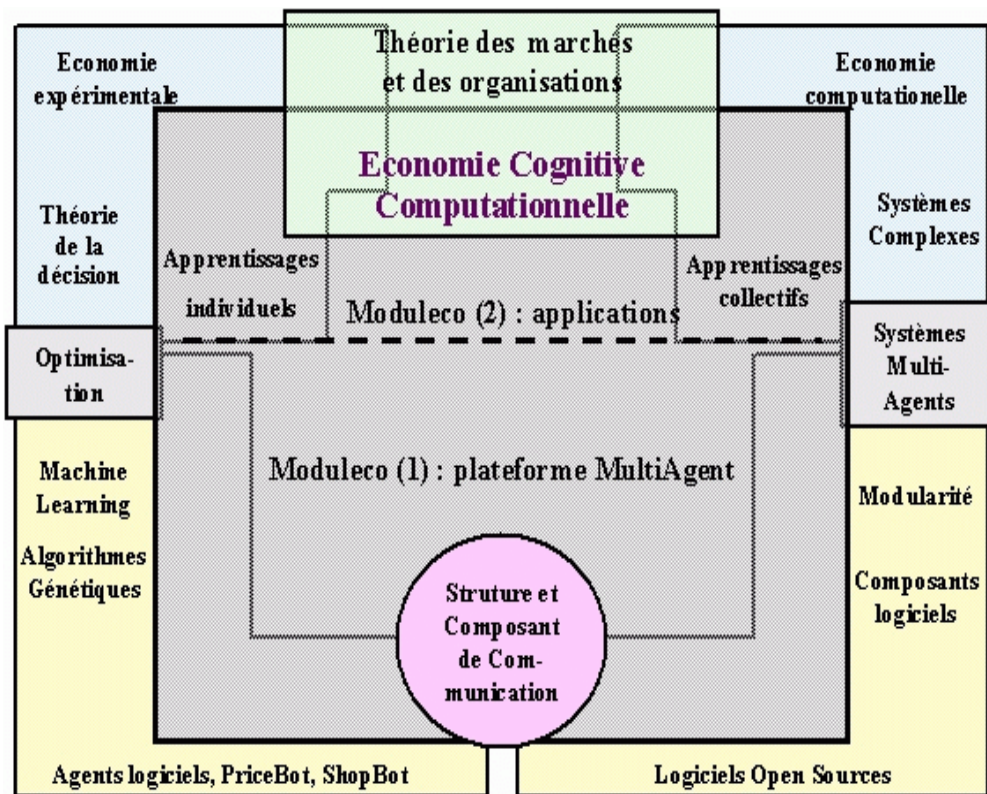


FIG. 1 – Domaines d’application (source A. Beugnard)

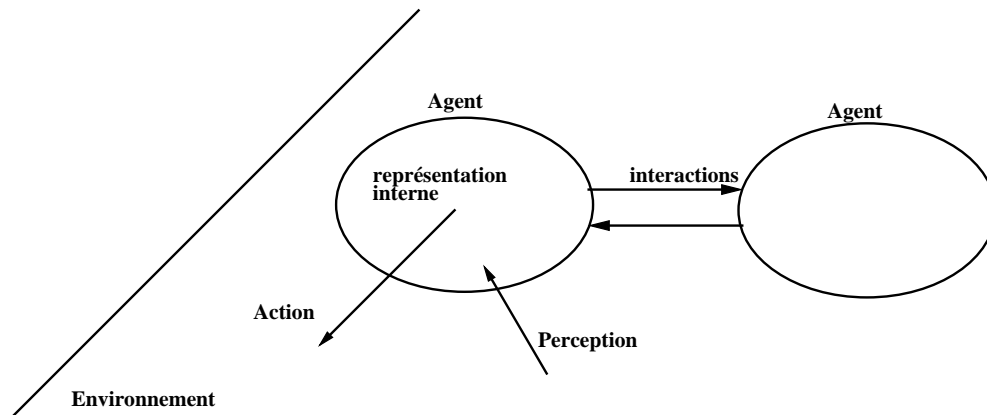


FIG. 2 – Système multi-agent

2.1.2 Définition d'un agent

C'est l'entité de base du système comme une personne dans une population ou un insecte dans une colonie. Il existe plusieurs définitions d'agents. Généralement on le définit comme une entité qui possède les propriétés suivantes :

- interagit avec l'environnement
- communique avec les autres agents
- possède des ressources propres
- capable de percevoir l'environnement et de réagir en conséquence

Une système composé d'agents peut être représenté comme l'indique la figure 2

En fonction de la tâche à accomplir, ces propriétés seront plus ou moins importantes. On peut toutefois distinguer les agents réactifs des agents délibératifs. Alors que les premiers sont programmés pour répondre à des stimuli de l'environnement extérieur ou de leurs voisins, les seconds intègrent un processus plus "intelligent" qui contient des fonctions d'apprentissage et de représentation élaborée de l'environnement.

2.1.3 Quelques exemples

Afin de mettre en place le prototype, des modèles connus et déjà testés sur d'autres plateformes ont été développés sur moduleco. Après cette phase il est possible d'implanter de nouveaux modèles.

Les exemples présentés sur les figures 3, 4 et 5 proviennent du site <http://digemer.enst-bretagne.fr/~phan/moduleco/> de D. Phan et A. Beugnard.

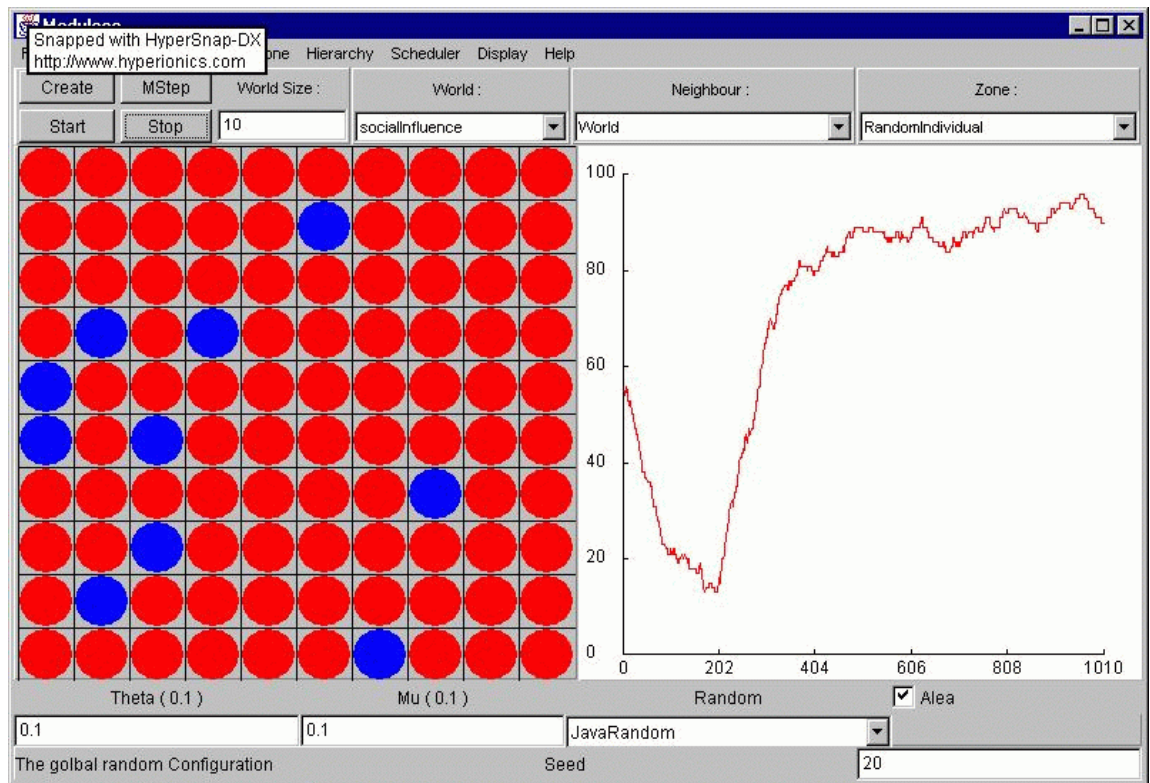


FIG. 3 – Influence sociale

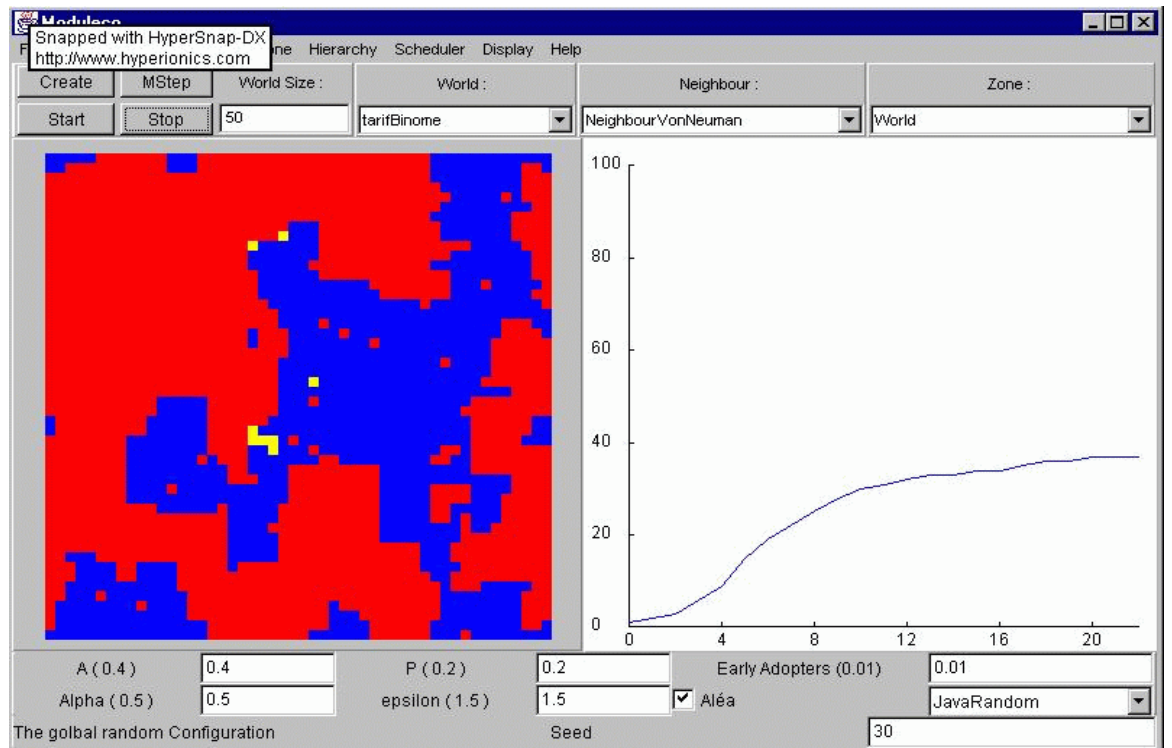


FIG. 4 – Tarif binôme

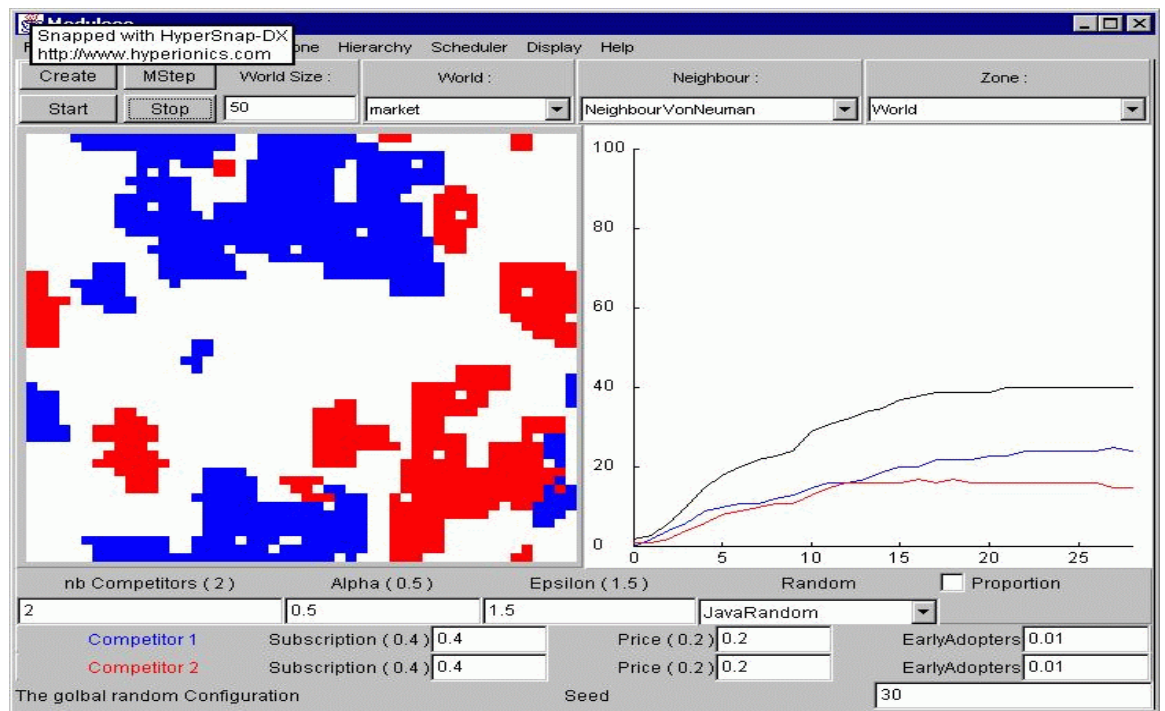


FIG. 5 – Compétition

2.2 Architecture, modèle

2.2.1 le modèle de base

L'approche objet qui répond bien aux concepts multi-agent et le langage Java sont les solutions techniques retenues pour développer la plate-forme.

Moduleco contient un noyau central autour duquel s'ajoutent des paquetages spécifiques aux simulations. Le but est de simplifier au maximum ces paquetages additionnels afin de permettre au plus grand nombre de créer des simulations.

Ajouter une simulation revient à ne coder que les lignes spécifiques au comportement voulu.

Les objets de base sont l'agent et le monde représentés par les classes abstraites **EAgent** et **EWorld**. Le monde contient des agents et évolue dans une structure d'interaction grâce à un ordonnanceur. (figure 6)

- L'agent **EAgent** possède une variable d'état et une méthode de calcul qui la modifie (en fonction de l'état de ses voisins ou du monde).
- Le monde **Eworld** qui est un ensemble d'agents possède les caractéristiques de l'agent (état et méthode calcul).
- Ces deux classes sont définies par l'interface **CAgent**.
- Le monde est attaché à une structure d'interaction qui a une dimension spatiale : **le voisinage** et une dimension temporelle : **la zone d'activation**.
- Le monde évolue grâce à un **ordonnanceur** ou **scheduler** qui appelle régulièrement la méthode de calcul du monde et des agents.
- Une simulation est composée principalement de l'objet **Agent** et de l'objet **World** qui héritent respectivement des classes abstraites **EAgent** et **EWorld**. Seule la méthode de calcul **compute()** est modifiée. Chaque simulation fait l'objet d'un paquetage additionnel.

2.2.2 Les voisinages

Chaque agent possède un voisinage, c'est à dire un ensemble d'agents susceptibles de faire évoluer son état. La méthode de calcul de l'agent peut avoir connaissance de l'état de voisins et agir en conséquence. Les voisinages sont choisis par l'utilisateur et ont une signification par rapport à la simulation choisie. Actuellement plusieurs types de voisinages ont été définis dans moduleco. La liste n'est pas figée, il est possible d'en ajouter d'autres. (figure 7)

Chaque objet "agent" possède un objet voisinage **Neighborhood** contenant une liste d'agents voisins. **Neighborhood** construit sa liste à partir de l'objet **ZoneSelector**.(figure 8)

2.2.3 Zone d'activation

Elle représente la zone active du monde. En d'autres termes il s'agit de l'ensemble des agents qui seront activés lors du processus de calcul. Les zones sont construites de la même façon que les voisinages et sont utilisées par l'ordonnanceur

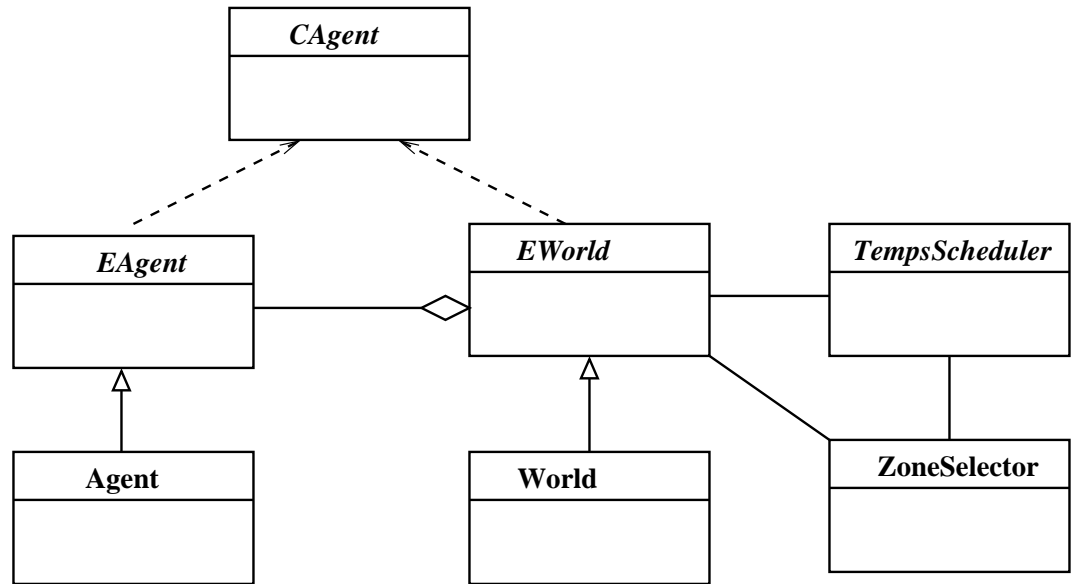


FIG. 6 – modèle UML de base

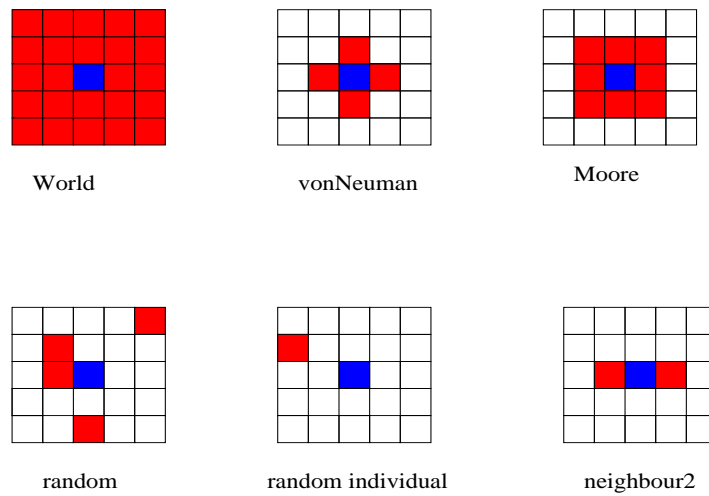


FIG. 7 – Exemples de voisinages

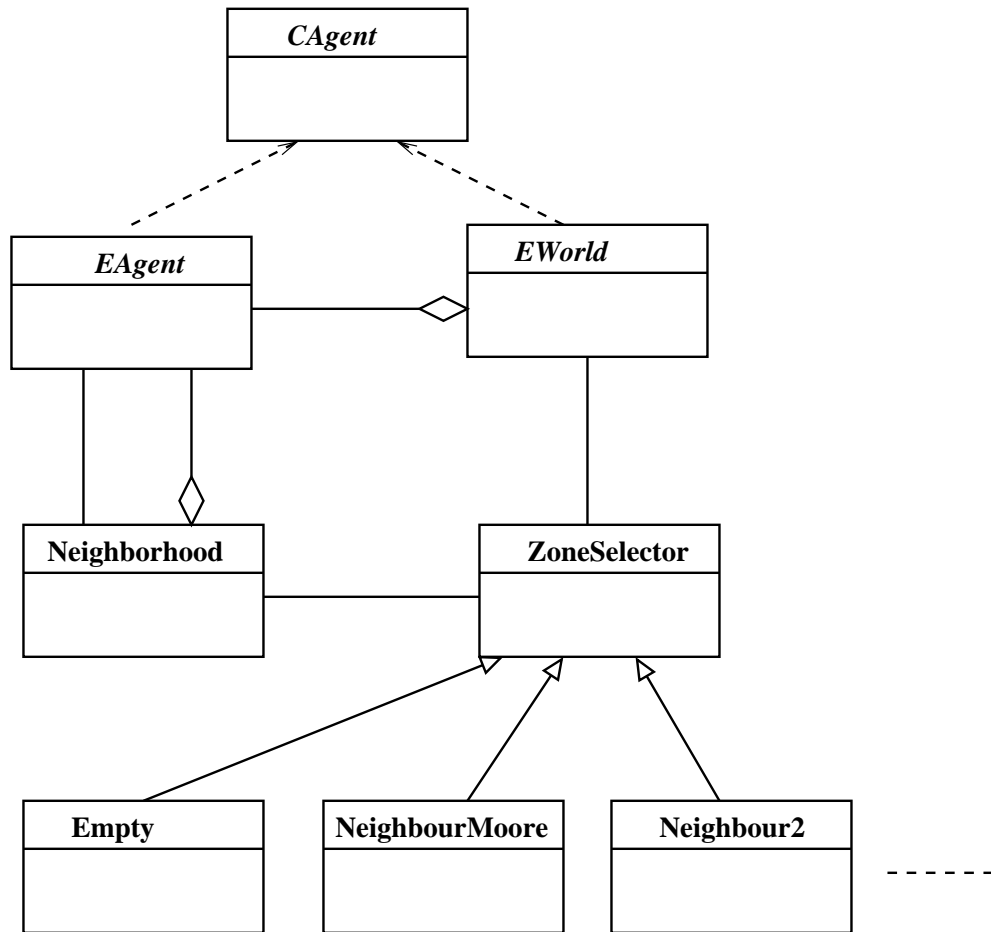


FIG. 8 – voisinage (UML)

2.2.4 Ordonnanceur ou scheduler

Le monde possède un ordonnanceur ou *TempsScheduler* (à renommer en TimeScheduler) classe abstraite réalisée par deux classes :

1. **LateCommitScheduler** : les agents calculent leur nouvel état et le valide à la fin de la période de temps , c'est à dire quand tous les agents ont été activés. Valider un état signifie que l'agent rend public l'état courant (avant la validation il s'agit de l'état de la période t-1).
2. **EarlyCommitScheduler** : les agents calculent et valide immédiatement leur nouvel état.

Le diagramme de classes est représenté figure 9 .

2.2.5 Représentation graphique

L'interface de l'application est composée de quatre parties (figure 10) :

1. un panel qui permet à l'utilisateur de paramétrer les simulations (voisinage, scheduler, zones d'activation) : **ControlPanel**
2. un panel graphique représentant le monde sous forme de tableau de points à deux dimensions ou plus précisément un tore à deux dimensions : **Canevas**
3. un panel graphique qui contient des courbes statistiques : **Graphique** (à renommer en Graphic)
4. un panel de saisie des valeurs du monde à simuler : **WorldEditor**

Il est aussi possible de visualiser les données d'un agent par le panel **AgentEditor**. (figure 11)

Globalement l'IHM est gérée par la classe **CentralControl**.

Le panel **Canevas** représente le monde et s'appuie sur l'état de chaque agent (figure 12). Une couleur représente chaque état et ses transitions :

- Bleu : False
- Rouge : True
- Jaune : l'état passe de True à False
- Vert : l'état passe de False à True

Le panel statistique **Graphique** s'appuie sur le **statManager** qui gère l'évolution de l'état des agents.

Ces deux panneaux sont programmés pour des simulations simples (états booléens). Chaque simulation peut posséder un **Canevas** et un **Graphique** spécifiques qui héritent des deux classes de bases.

Le panel **AgentEditor** est par contre un objet abstrait. Il doit être réalisé dans chaque paquetage de simulation.

2.3 Les paquetages

(figure 13)

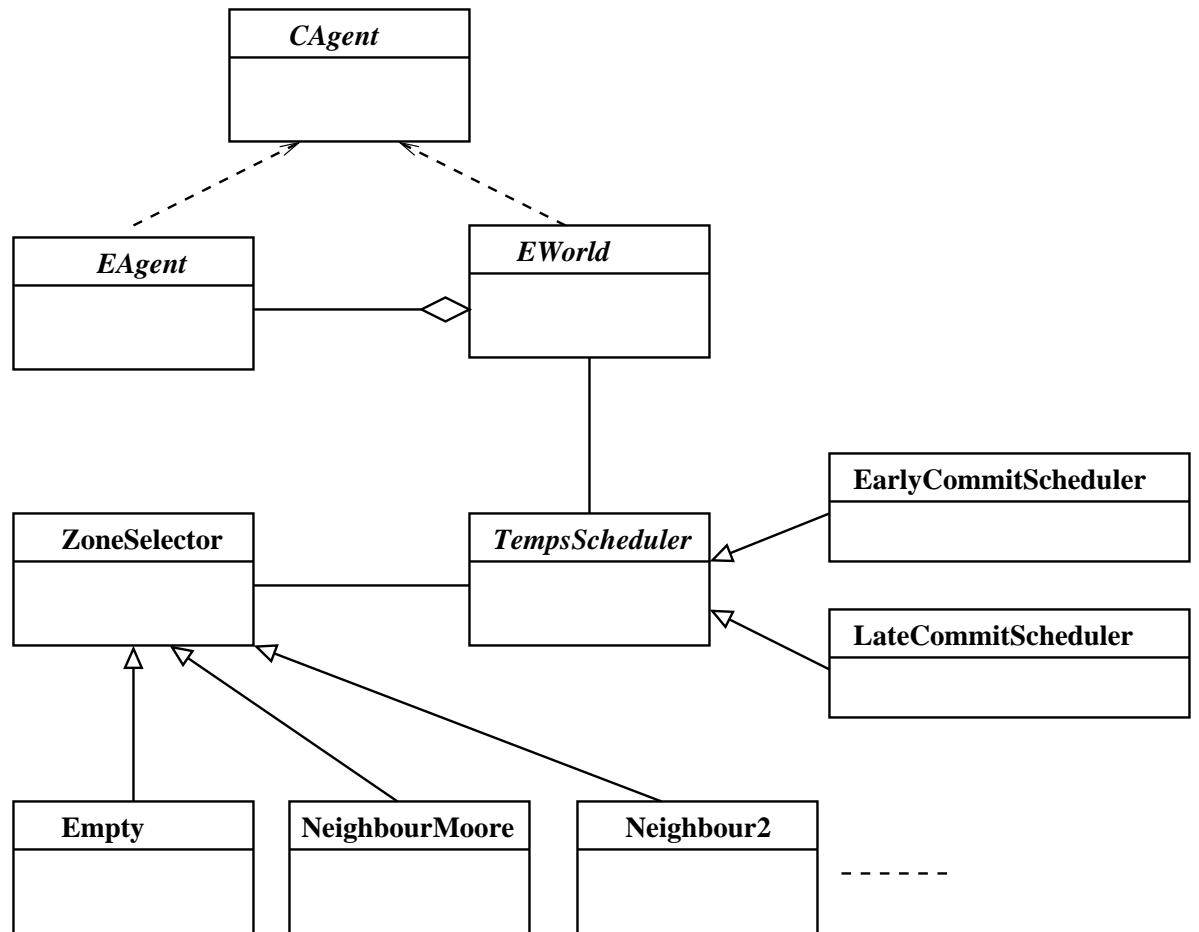


FIG. 9 – Scheduler (UML)

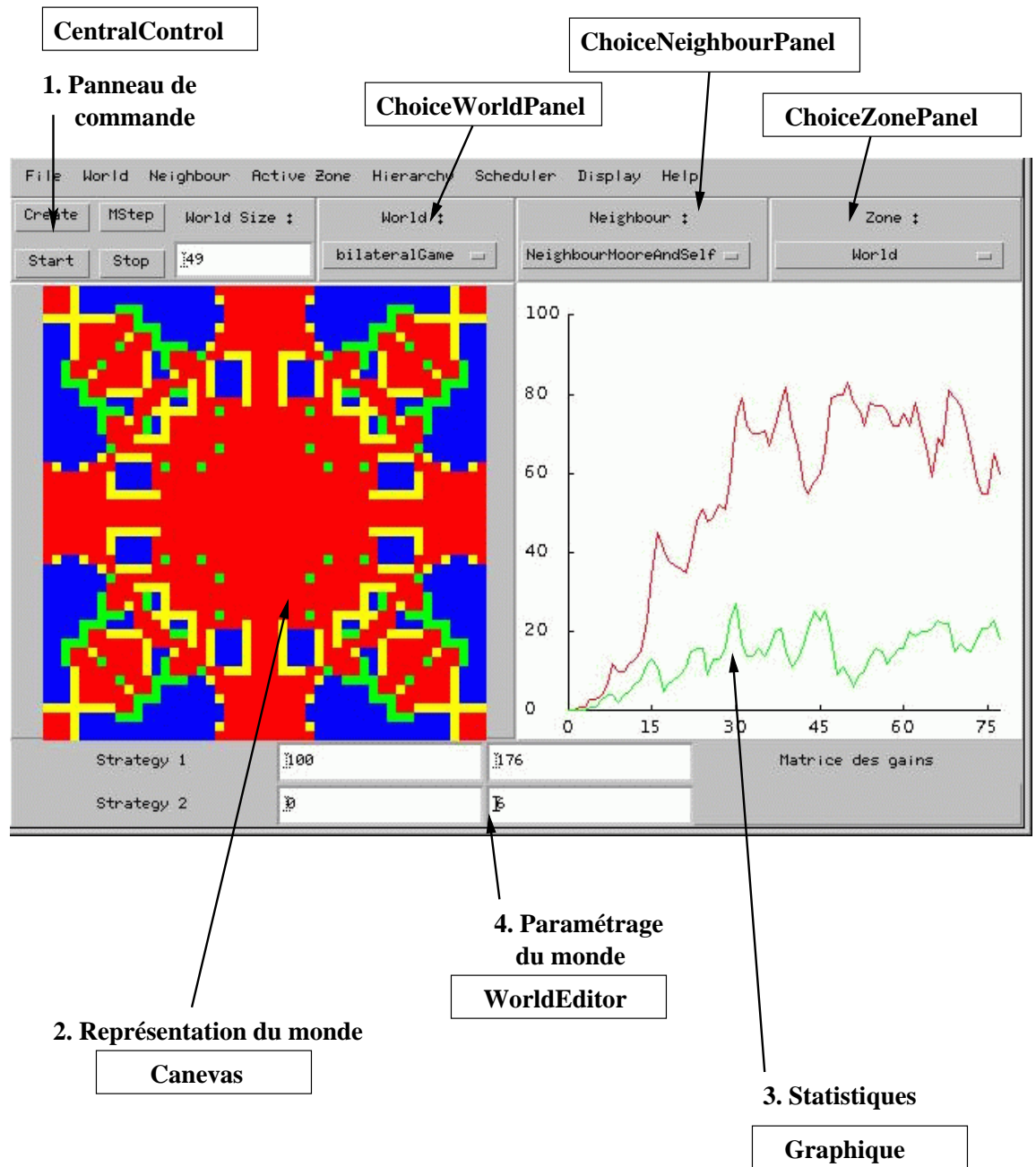


FIG. 10 – Représentation graphique

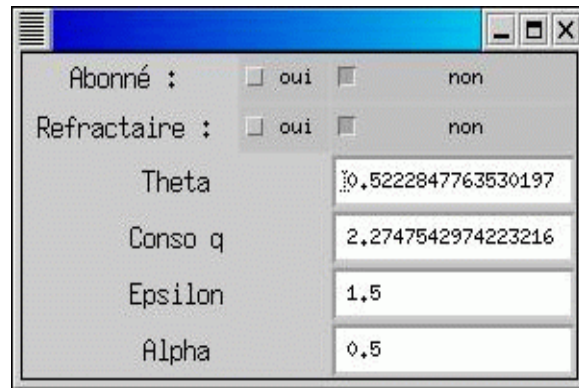


FIG. 11 – Agent Editor

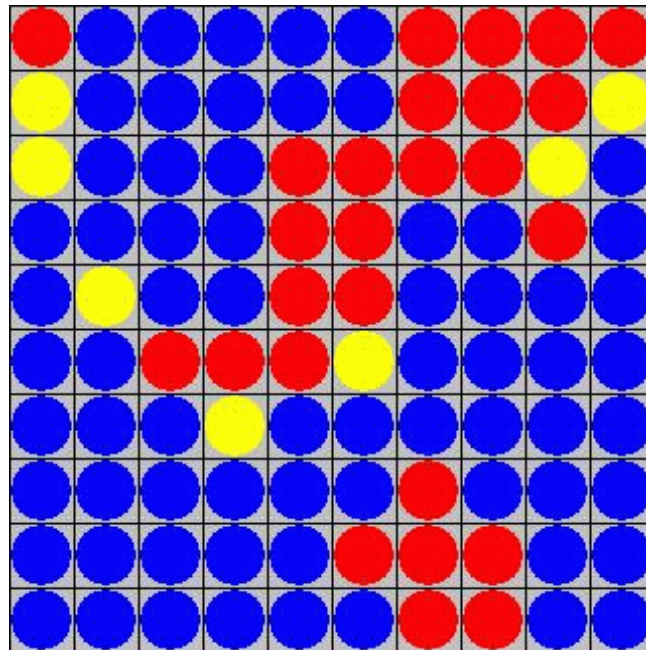


FIG. 12 – Représentation des agents

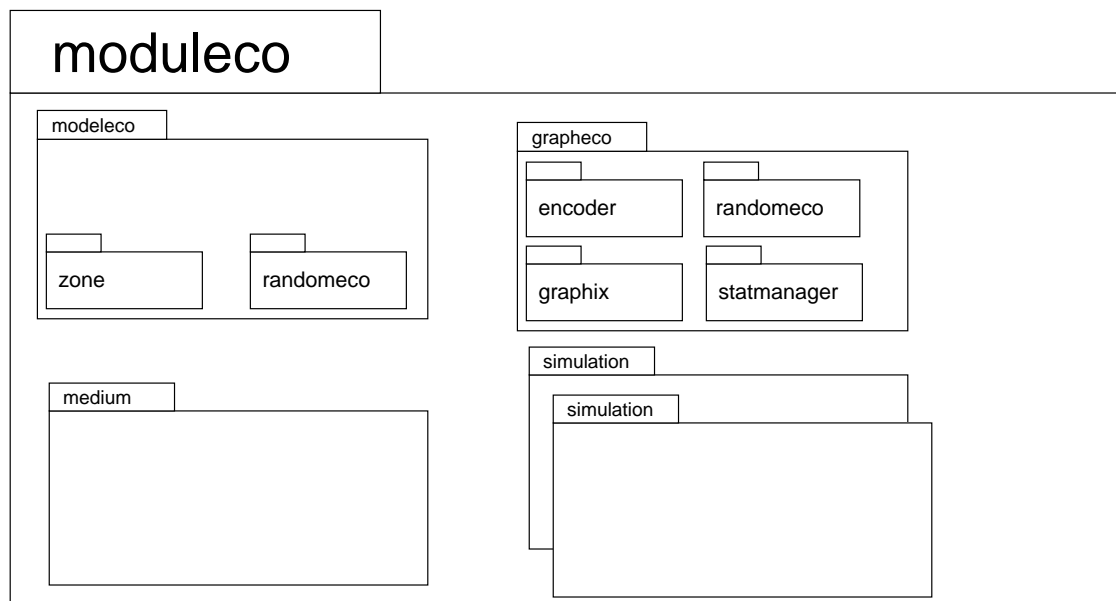


FIG. 13 – Packages

2.3.1 modeleco

C'est le coeur de l'application. Il contient les classes suivantes :

- **CAgent.java** : Interface définissant l'agent et le monde
- **EAgent.java** : Classe abstraite de l'agent. Définit les fonctions génériques.
- **EWorld.java** : Classe abstraite du monde = ArrayList l'agent. Définit les fonctions génériques du monde
- **TempsScheduler.java** : Ordonnanceur (Classe abstraite). Fait évoluer le monde (gère les appels à la méthode **compute** du **World**).
- **EarlyCommitScheduler.java** : réalisation concrète de l'ordonnanceur
- **LateCommitScheduler.java** : réalisation concrète de l'ordonnanceur
- **ZoneSelector.java** : Classe abstraite. Permet de construire les listes de voisins et des zones actives.

modeleco.zone Sous paquetage qui contient les réalisations concrètes de l'objet **ZoneSelector**.

- **BoundedRandom.java**
- **Empty.java**
- **Neighbour2.java**
- **Neighbour4.java**
- **NeighbourMoore.java**
- **NeighbourMooreAndSelf.java**

- **NeighbourVonNeuman.java**
- **NeuronalNetwork.java**
- **Random.java**
- **RandomIndividual.java**
- **RandomPairwise.java**
- **World.java**

modeleco.randomeco Ce sous paquetage a été conçu pour enrichir les objets Random fournis par Java.

- **Default.java**
- **JavaGaussian.java**
- **JavaRandom.java**
- **Random.java**

2.3.2 grapheco

C'est le paquetage qui gère l'IHM. L'élément principal est la classe **CentralControl** qui est lancée à l'initialisation de l'application. Cette classe crée le monde et gère les interactions avec l'utilisateur. Le paquetage contient tout ce qui est relatif à la représentation de la simulation à l'écran. (Voir aussi figure 10)

- **CAgentEditor.java** : Interface de l'objet AgentEditor
- **CAgentRepresentation.java** : Interface de Cannevas
- **CAgentRepresentationContainer.java** : Hérite de Cpanel
- **CBufferedCanvas.java** :
- **CPanel.java** : hérite de java.awt.Panel
- **CWorldEditor.java** : Interface, définit le WorldEditorPanel
- **Canevas.java** : Représentation du monde
- **CentralControl.java** : Classe principale de l'IHM
- **ChoiceNeighbourPanel.java** : Choix du type de voisinage sur le ControlPanel.
- **ChoiceWorldPanel.java** : Choix du monde sur le ControlPanel.
- **ChoiceZonePanel.java** : Choix de la zone d'activation sur le ControlPanel.
- **ControlPanel.java** : Panel de contrôle de moduleco.
- **EAdditionalPanel.java** : Classe abstraite à partir de laquelle héritent les WorldEditor.
- **EAgentEditor.java** : Classe abstraite : édite les paramètres de l'agent.
- **EDialog.java** : Boite de dialogue java.awt.dialog adaptée à moduleco.
- **EPanel.java** : java.awt.Panel enrichi pour moduleco
- **EWarningDialog.java** : Permet d'afficher des messages d'erreur
- **EWorldEditor.java** : Classe abstraite : affiche les valeurs du monde à simuler
- **Histo.java** : Affiche les statistiques sous la forme d'histogrammes
- **ImageENSTB.java** : Affiche l'image par défaut.
- **ItemComponent.java** : utilisé dans EAgentEditor, item dans une liste.
- **ModulecoTextField.java** : Textfield enrichi pour les besoins de l'application
- **PanelSud.java** : Bas d'écran à l'initialisation
- **PermanentControlPanelBar.java** : Utilisé par ControlPanel

grapheco.encoders Sous paquetage permettant d'enregistrer les graphiques des simulations dans un format image.

grapheco.graphix Sous paquetage qui affiche le graphique de statistiques. **Graphique** et **Trace** tracent les courbes dont les données proviennent du **StatManager**.

grapheco.randomeco Ce sous paquetage permet de choisir via l'IHM les objets définis dans `modeleco.randomeco`.

grapheco.statManager Gestion des valeurs à afficher dans la partie statistique. Le monde en s'initialisant fournit au **StatManager** les méthodes à appeler pour calculer les valeurs des données à afficher.

- **StatManager.java** : instancié par le monde, cette classe calcule et garde les valeurs qui seront affichées par le Graphique.
- **CalculatedVar.java** : utilisé par le Statmanager : définit une variable à calculer.
- **Var.java** : Accède à la méthode déclarée par le monde pour récupérer une variable.
- **VarCalculator.java** : Calcule les statistiques.

Exemple issu de `tarifBinome.World` : `statManager.add(new CalculatedVar("TrueState", Class.forName("tarifBinome.Agent").getMethod("getState", null), CalculatedVar.NUMBER, new Boolean(true)));`

2.3.3 medium

C'est un paquetage intermédiaire en vue de l'implémentation du framework **Medium**. Concrètement il est utilisé pour gérer le voisinage. Chaque agent est lié à l'objet **Neighbours** qui contient la liste de ses voisins.

2.4 Paquetages de simulation

2.4.1 Classes de base

Créer une simulation revient à créer un nouveau paquetage. Dans ce paquetage la première chose à faire est de redéfinir les méthodes **compute()** de la classe **Agent** et la classe **World** (qui héritent de **EAgent** et **EWorld**).

2.4.2 Adaptation des classes graphiques

- **WorldEditor** hérite de **grapheco.EWorldEditor** : Saisie des paramètres du monde, à adapter en fonction des zones à saisir ou à afficher.
- **AgentEditor** hérite de **grapheco.EAgentEditor** : permet d'afficher les attributs de l'agent à la demande (click droit)
- **Canevas** : par défaut est adaptée à des états booléens. Si elle ne convient pas, il faut la redéfinir (méthodes `setCagent` et `drawPoint`) dans le nouveau paquetage.

- De la même façon la classe **Graphique** doit être redéfinie afin d'indiquer les courbes à tracer et leur couleur. La classe **World** indique au **StatManager** les méthodes à appeler pour accéder aux variables d'état des agents afin de calculer les statistiques ainsi que les courbes qui leur sont associées.

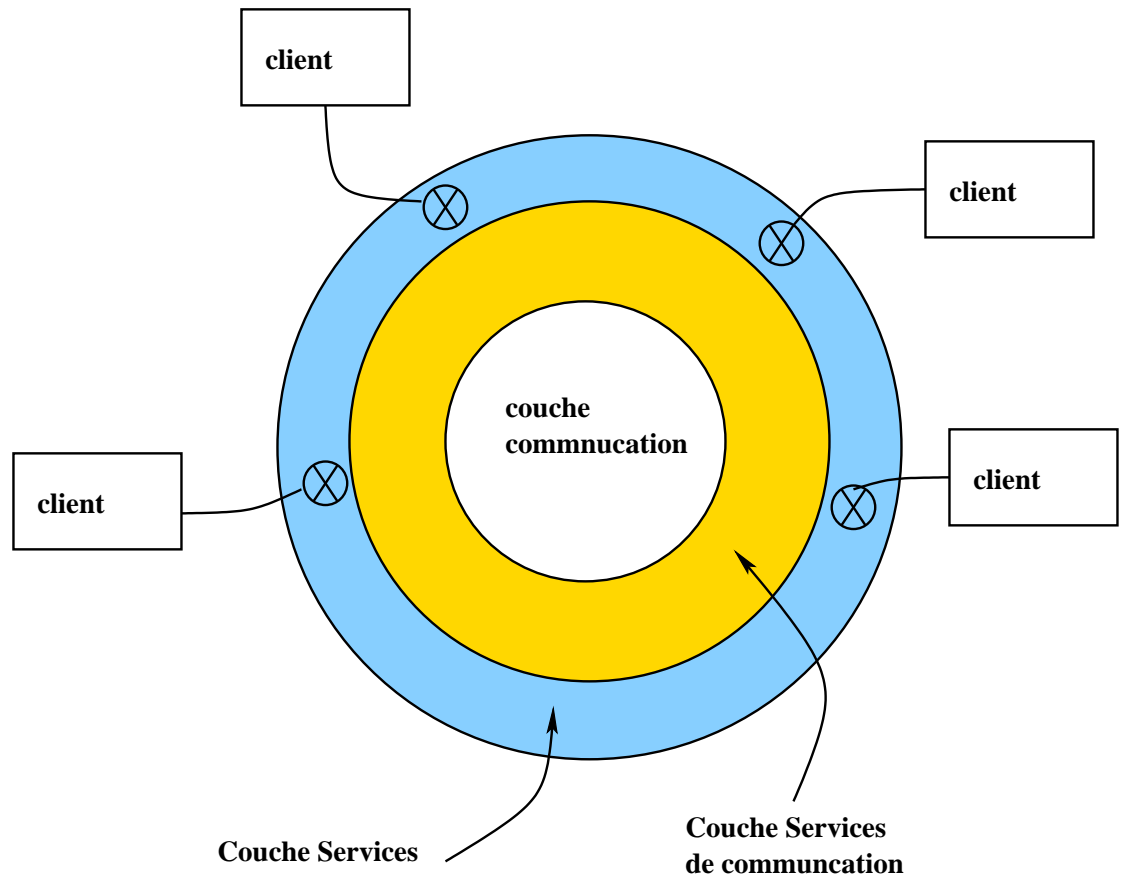


FIG. 14 – Représentation schématique du framework

3 Le FrameWork Medium

3.1 Présentation

Le framework Medium est un package développé par Erci Cariou dans le cadre de sa thèse au département informatique de l'ENST Bretagne.

Le site <http://www-info.enst-bretagne.fr/medium/index-fr.html> présente en détail le concept de médium. Je ne le décrirai pas plus dans ce document.

Schématiquement le package Medium peut se représenter comme l'indique la figure 14.

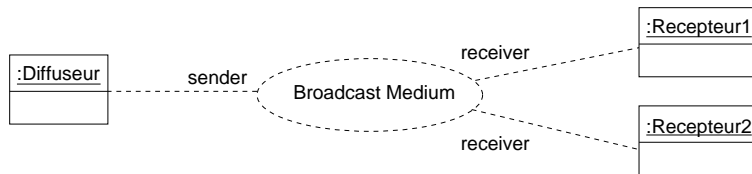


FIG. 15 – Diagramme d’instances UML : “snapshot” d’une application utilisant un médium de diffusion d’événement

3.2 Architecture générale du framework (D’après la documentation de Eric Cariou ENST Bretagne)

La figure 15 est un diagramme d’instances UML représentant une application composée de 4 composants :

- **Broadcast Medium**, un médium de diffusion de messages acceptant deux types de rôles :

1. Le rôle **sender** qui peut appeler le service `send (Message msg)` sur le médium pour envoyer le message `msg` à tous les composants connectés en tant que **receiver**. L’appel de ce service est non-bloquant.
2. Le rôle **receiver** qui peut appeler le service `Message receive()` pour recevoir le premier message non-lu de sa file de message (le message lu en est alors supprimé). Si aucun message n’est disponible, on lui renvoie `null`. L’appel de ce service est non bloquant (lecture de type << boîte aux lettres >>).

- Un objet `Diffuseur` connecté au médium en tant que **sender**.
- Deux objets `Recepteur [1 / 2]` connecté au médium en tant que **receiver**.

La figure 16 montre les différents éléments lors du déploiement de cette application dont le médium a été implémenté grâce à notre framework.

La première chose à prendre en considération est que les différents éléments formant le médium sont physiquement distribués sur plusieurs sites (un site correspondant plus précisément à une machine virtuelle java (JVM), plusieurs JVMs pouvant tourner sur la même machine physique).

Le site B contient des éléments particuliers qui servent à la gestion des différents médiums présents dans l’application (il n’y a qu’un seul médium dans notre cas).

3.2.1 Connexion d’un composant à un médium

Un composant est connecté à un médium en ayant une référence (réciproque si le médium doit appeler des services sur ce composant) sur un objet << role manager >>. Un `role manager` est un objet gérant la partie du protocole ou du service de communication pour un rôle donné. Dans notre cas, le médium accepte deux rôles de connexion de composant, il y aura donc deux `role managers` :

- `SenderManager` pour le rôle **Sender**, associé au composant `Diffuseur` et qui implémente le service `send`.

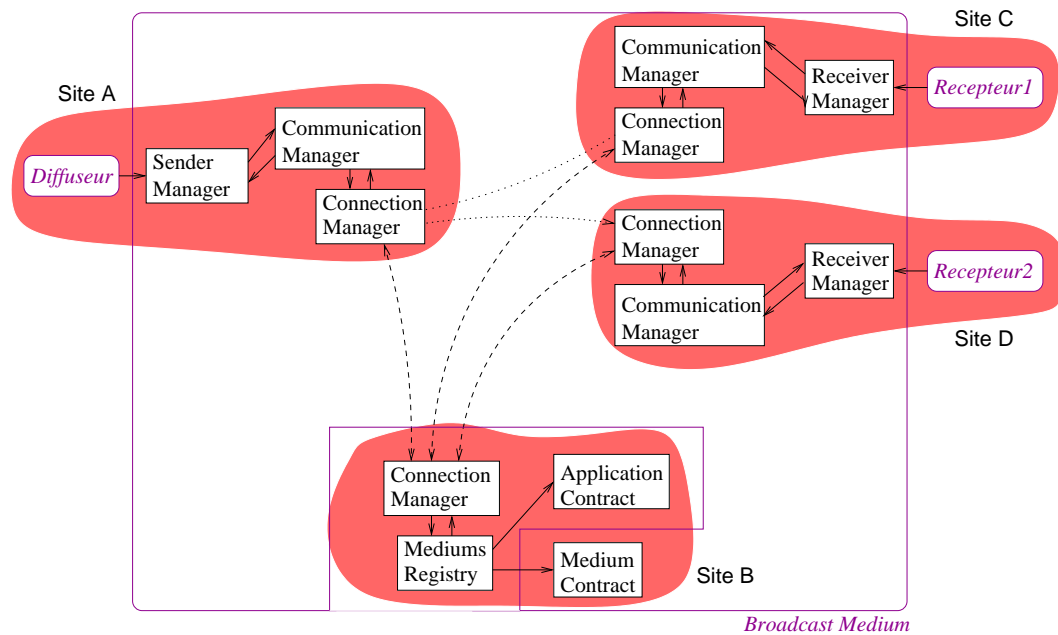


FIG. 16 – Framework

- *ReceiverManager* pour le rôle *Receiver*, associé aux composants *RecepteurX* et qui implémente le service *receive*.

Ces role managers forment la seule partie du médium dépendante du service ou du protocole qu'il implémente.

Les role managers distants doivent communiquer entre eux pour réaliser le service. Par exemple, l'appel du service `send(msg)` sur *SenderManager* nécessite d'envoyer le message `msg` vers tous les *ReceiverManager* présents au moment de l'émission.

Pour cela, les role managers s'appuient sur des services offerts par un objet *CommunicationManager* permettant notamment d'effectuer des appels de méthodes sur un ensemble déterminé de role managers distants.

C'est cet objet qui connaît les autres managers distribués. C'est lui qui possède les références sur les connexions (objets *Connection*) avec les autres managers.

L'objet *CommunicationManager* est identique dans tous les médiums. C'est le même pour tous les médiums, indépendamment du protocole ou du service implémenté.

L'objet *ConnectionManager* sert à ouvrir des connexions vers des managers distants et à accepter des connexions de leur part. Il est spécifique au système de << communication physique >> choisi (sockets TCP, RMI, Corba...). Une connexion étant gérée par un objet de la classe *Connection*.

Chaque (role) manager est identifié par quatre données (regroupées dans un objet de classe `RoleIdentifier`) :

- Le type du médium auquel il appartient
- L'identificateur du médium auquel il appartient
- Son rôle
- Son adresse physique (qui sera utilisée pour ouvrir des connexions avec ce manager)

Toute communication entre deux managers ou le << medium registry >> se fait par l'envoi d'un événement (classe `MediumEvent` et ses diverses sous-classes).

3.2.2 `MediumsRegistry` et objets associés

Le `MediumsRegistry` est un objet particulier. C'est l'objet qui connaît tous les role managers (et donc les sites) de tous les médiums présents dans une application à un instant donné. Il gère une application dans son ensemble.

L'objet `ApplicationContract` est le contrat de l'application. C'est lui qui doit vérifier que les différentes connexions entre composants et médiums sont correctes.

À chaque médium est associé un objet `MediumContract`, géré par le `mediums registry`. Il a pour but de vérifier que les connexions de composants sont légales ou non pour un médium donné.

La connexion d'un composant à un médium implique l'instantiation des trois objets `RoleManager`, `CommunicationManager` et `ConnectionManager`. Une fois ces instantiations effectuées, les actions suivantes sont réalisées :

1. Une connexion est ouverte avec le `mediums registry`.
2. Le role manager envoie son identificateur au `mediums registry` afin de l'informer de sa demande d'appartenir au médium.
3. Le `mediums registry` interroge l'application contract et/ou le medium contract pour savoir si le nouveau manager doit être accepté puis en notifie le role manager.
4. Si la connexion est acceptée, le role manager va effectuer des demandes d'abonnements auprès du `mediums registry`. Il va envoyer une liste de noms de rôle qui correspondent aux types des role managers avec lequel le role manager aura besoin d'ouvrir des connexions.
5. Le `mediums registry` interroge le medium contract pour savoir si les abonnements sont valides ou non. Pour les valides il renvoie les adresses des managers étant de ces roles et également la liste des rôles pour lesquels l'abonnement a été refusé par le medium contract. Par la suite, à la nouvelle connexion d'un composant, si son role manager est d'un rôle auquel notre manager s'est abonné, ce dernier recevra l'adresse de ce nouveau manager afin de pouvoir ouvrir une connexion avec celui-ci.
6. Le role manager ouvre des connexions avec tous les roles managers distants dont il vient de recevoir les adresses.

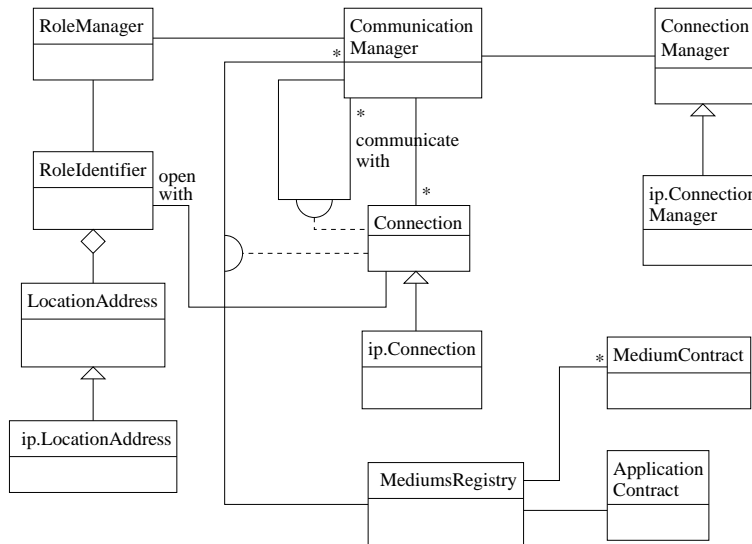


FIG. 17 – Diagramme de classe UML de framework

3.3 Architecture détaillée

Cf figure 17 pour le diagramme de classe UML du framework.

3.4 Création de nouveaux médiums

L'écriture d'un nouveau role manager se fait par la création d'une classe héritant `RoleManager` dont les attributs et les méthodes suivants sont à redéfinir :

protected Vector subscribedRoleNames la liste des noms (objets `String`) de rôles auxquels le manager veut s'abonner. L'initialisation de ce vecteur peut se faire de manière simple dans le constructeur de la nouvelle classe.

protected String getMyRoleName() retourne le nom du rôle du manager.

void managerDisconnection(RoleIdentifier remoteRoleId) cette méthode est appelée lorsque la connexion avec un manager distant (d'identificateur `remoteRoleId`) vient de se terminer.

void newManagerConnection(RoleIdentifier remoteRoleId) cette méthode est appelée lorsqu'une nouvelle connexion a été ouverte avec un manager distant (d'identificateur `remoteRoleId`).

Utiliser ensuite les services de l'interface `ICommunicationManagerServices`.

Note : medium contract à redéfinir également.

3.5 Création d'une nouvelle couche de communication

Cela se fait en créant trois classes héritant chacune d'une des trois classes suivantes du package `fr.enstb.medium.kernel` : `ConnectionManager`, `Connection` et `LocationAddress`.

Mettre ces trois nouvelles classes (en conservant leur nom) dans un package `fr.enstb.medium.kernel-xxx` où `<< xxx >>` est le type du support physique utilisé. Par exemple, l'implémentation au dessus de IP est dans le package `fr.enstb.medium.kernel.ip`.

3.5.1 Classe `LocationAddress`

Cette classe représente une adresse physique dont le format dépend du support choisi. Par exemple, l'implémentation au dessus de IP utilise un couple [Adresse IP, numéro de port] comme adresse physique (sur le port sera connecté une socket serveur TCP en attente de connexion).

Méthodes à redéfinir dans cette classe, en plus des attributs et de leur méthodes d'accès nécessaires aux stockage de l'adresse physique :

`boolean equals(java.lang.Object obj)` : pour pouvoir comparer deux objets de cette classe. Typiquement : tester le type du paramètre (il doit être de classe `...kernel.xxx.LocationAddress`) puis comparer les valeurs des attributs un à un.

`String toString()` : pour afficher la valeur de l'adresse. Pas obligatoire mais utile en phase de test.

`LocationAddress(...)` : prévoir un nouveau constructeur pour instancier un objet directement avec les bonnes valeurs de ses attributs.

3.5.2 Classe `ConnectionManager`

Cette classe à plusieurs buts :

- Déterminer l'adresse physique associée au manager (l'objet `LocationAddress`, attribut de l'objet `RoleIdentifier`). En effet, le format et le contenu de l'adresse physique associée à un role manager dépend du support physique utilisé. L'adresse physique déterminée sera utilisée par les managers distants pour ouvrir des connexions avec notre role manager.
- Ouvrir des connexions avec des managers distants dont on connaît l'identificateur complet ou juste son adresse physique.
- Accepter des connexions de managers distants. Le communication manager est informé de ces nouvelles connexions (et également des déconnexions lorsqu'elles sont détectées).

Une connexion correspond à un objet de classe `Connection`.

Les quatre méthodes à redéfinir sont :

Connection openConnection(LocationAddress location) : ouvre une connexion avec un manager distant dont on ne connaît que l'adresse physique puis renvoie l'objet **Connection** (correctement initialisé) permettant de communiquer avec ce manager.

Connection openConnection(RoleIdentifiant remoteRoleId) : ouvre une connexion avec un manager distant dont on connaît le role identifiant puis renvoie l'objet **Connection** (correctement initialisé) permettant de communiquer avec ce manager.

Connection openConnectionWithMediumsRegistry(LocationAddress location) : ouvre une connexion avec le mediums registry dont on connaît l'adresse physique puis renvoie l'objet **Connection** (correctement initialisé) permettant de communiquer le mediums registry.

LocationAddress setAndInitializeLocationAddress() : initialise le connection manager. Cela implique notamment de construire l'adresse physique associée au role manager et à être prêt à recevoir des connexions (dans le cas de l'implémentation IP, un thread qui ouvre une socket server liée au numéro de port de l'adresse physique est lancé). Puis il retourne l'adresse physique construite.

Quand un manager distant ouvre une connexion avec le manager courant, par l'intermédiaire du système adéquat mis en place à l'initialisation du connection manager, ce dernier doit en informer le communication manager en appelant la méthode `newRemoteManagerConnected(remoteRoleId, connection)` sur celui-ci, où :

- `remoteRoleId` est l'identificateur du role manager distant. Il faut donc que l'événement `RoleIdentifiantEvent`⁴ soit lu avant d'informer le communication manager.
- `connection` est l'objet de classe **Connection** permettant de communiquer avec le manager distant qui vient de se connecter.

Si le connection manager détecte qu'une connexion avec un manager distant vient d'être perdue, il en informe le communication manager qui se chargera de fermer la connexion (appel de la méthode `close()`) en appelant la méthode : `remoteManagerDisconnected(remoteRoleId)` où `remoteRoleId` est l'identificateur du manager avec lequel la connexion a été perdue⁵.

Si le connection manager reçoit un événement provenant d'un manager distant, ou du mediums registry, il déposera ce dernier (en l'encapsulant dans un événement **ReceivedEvent** permettant d'indiquer l'émetteur de l'événement) dans le buffer d'événements (classe **BufferEvent**) adéquat : l'objet `managerBuffer` si l'événement vient d'un manager ou l'objet `registryBuffer` ⁵ s'il vient du mediums registry.

⁴ quand un manager ouvre une connexion avec un manager distant, la première chose qu'il fait, une fois la connexion ouverte, est d'envoyer un événement de type `RoleIdentifiantEvent`. afin d'indiquer son identificateur au manager distant

⁵ ce cas ne se présente jamais dans l'implémentation IP.

3.5.3 Classe Connection

Les méthodes à redéfinir sont les suivantes :

initialization() : initialise la connexion (facultatif).

void closeConnection() : ferme la connexion.

void sendEvent(MediumEvent evt) throws Exception : envoyer un événement au manager distant associé à la connexion. Si une exception est levée, c'est qu'un problème a eu lieu, la connexion est considérée comme rompue (c'est le communication manager qui récupère cette exception et qui s'occupera d'appeler la méthode `close` sur l'objet `Connection`).

Si un objet `connection` détecte que la connexion avec son manager distant vient d'être perdue, il en informe le `communication manager` qui se chargera de fermer cette connexion (appel de la méthode `close()`) en appelant la méthode : `remoteManagerDisconnected(remoteRoleId)` où `remoteRoleId` est l'identificateur du manager avec lequel la connexion a été perdue.

Si un objet `connection` reçoit un événement provenant d'un manager distant (qui pourra être le `mediums registry`), il déposera ce dernier (en l'encapsulant dans un événement `ReceivedEvent` permettant d'indiquer l'émetteur de l'événement) dans le buffer d'événements (classe `BufferEvent`) dont il possède la référence : l'objet `event-Buffer`

Note : une connexion est *obligatoirement* bidirectionnelle.

4 Utilisation du Framework Medium dans Moduleco

4.1 Le monde et les agents

4.1.1 Introduction

Avant d'appliquer le concept de médium au marché virtuel il est nécessaire de revoir l'architecture de moduleco. La principale difficulté est de désolidariser les agents du monde. Le monde est actuellement une liste d'agents et toute l'application repose sur ce lien.

L'idée de départ est que le monde est un médium. Les agents s'y connectent avec un rôle d'"agent" et l'ensemble évolue en utilisant les possibilités de communication et de synchronisation du médium. La notion de monde en tant qu'objet concret doit toujours exister pour coordonner l'ensemble (en gardant le scheduler). Le modèle de base devient donc celui présenté sur la figure 18.

4.1.2 La classe *EAgent*

Elle est rattachée au médium avec le rôle "agent" via la classe **AgentManager**. En distribuant les agents se pose le problème de les retrouver. Nous allons donc leur attribuer un identifiant et pour cela nous utiliserons l'adresse de localisation du Framework medium (qui identifie de manière unique le composant).

Ainsi, les modifications apportées à la classe *EAgent* sont :

- une phase de connexion au médium
- la récupération de l'objet *LocationAdress* comme identifiant.

4.1.3 La classe *EWorld*

Le monde ne connaît plus ses agents que par leur identifiant. La classe *EWorld* devient une *ArrayList* d'identifiants d'agents. Elle est rattachée au médium avec le rôle "world" via la classe **WorldManager**. (figure 19)

Les modifications apportées se situent au niveau du constructeur :

- Connexion au médium.
- Création des agents (pourrait être faite ailleurs).
- Récupération des identifiants d'agents via le médium (**role.getListAgentId()**).
- Déclenchement de la phase d'initialisation des agents (**role.initAgent()**)
- Déclenchement de la phase de construction des voisinages (**role.setNeighborhood("type de voisinage")**)

4.1.4 Le Scheduler *TimeScheduler*

Nous pouvons garder le même mécanisme d'activation puisque le monde est toujours une liste.

Jusqu'à présent le Scheduler appelait directement les méthodes **compute()** de chaque agent (ou plus précisément les méthodes **progress()** et **commit()**). Maintenant ces méthodes sont appelées via le **WorldManager** en précisant les identifiants des agents concernés. Il n'y a pas de modifications côté agent. (cf figure 20).

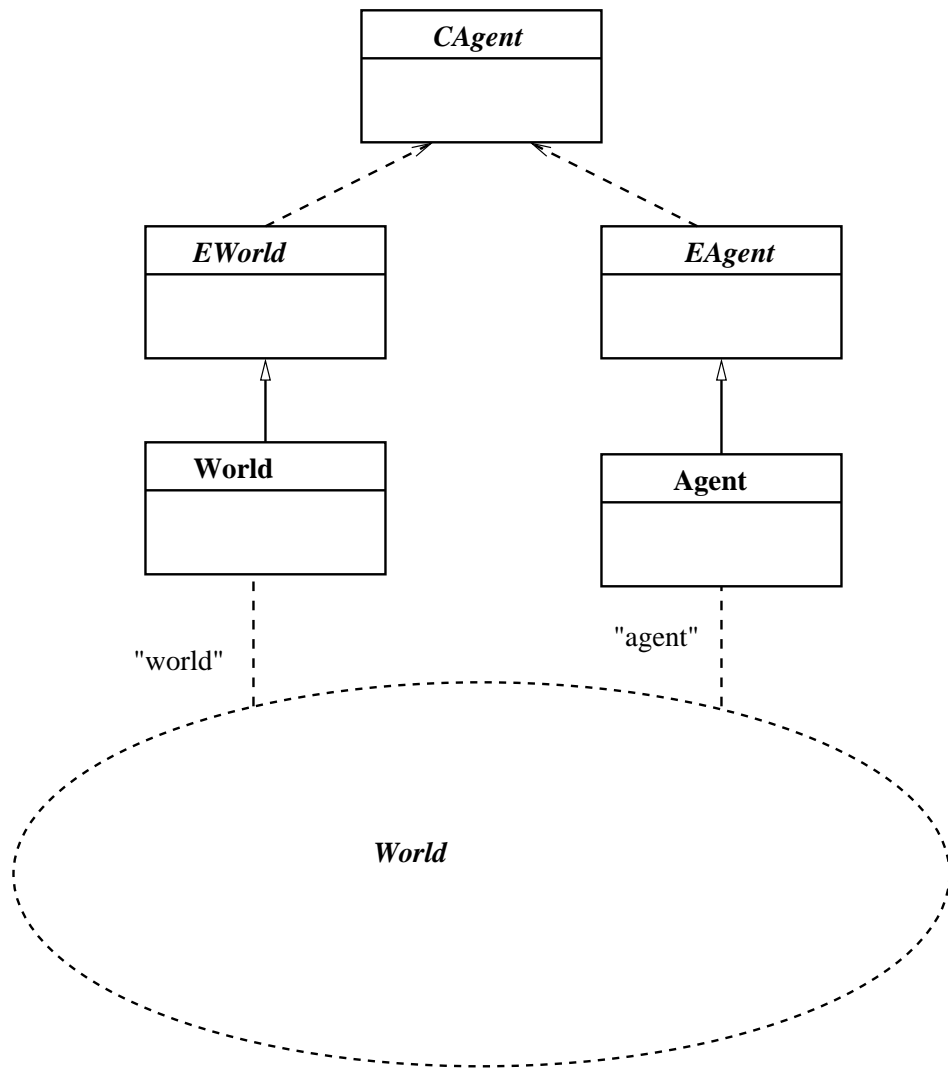


FIG. 18 – modeleco avec medium

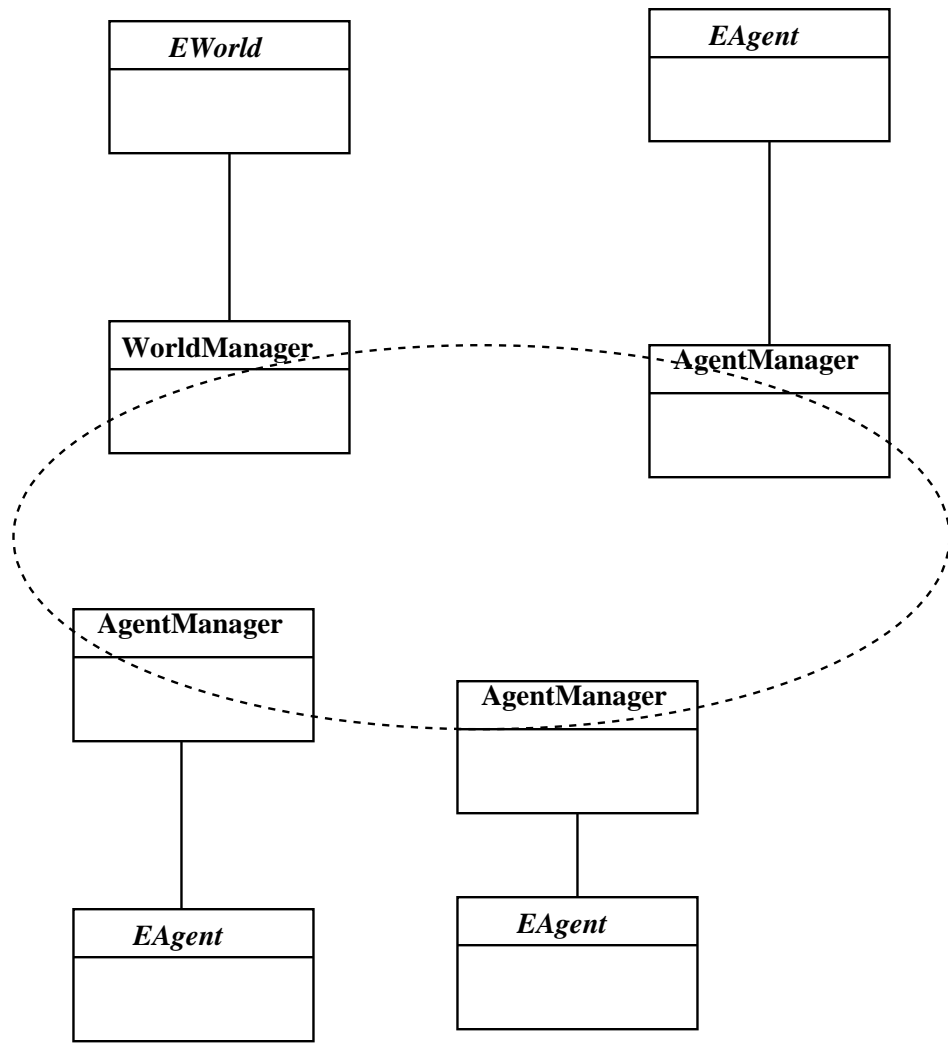


FIG. 19 – modeleco avec RoleManager

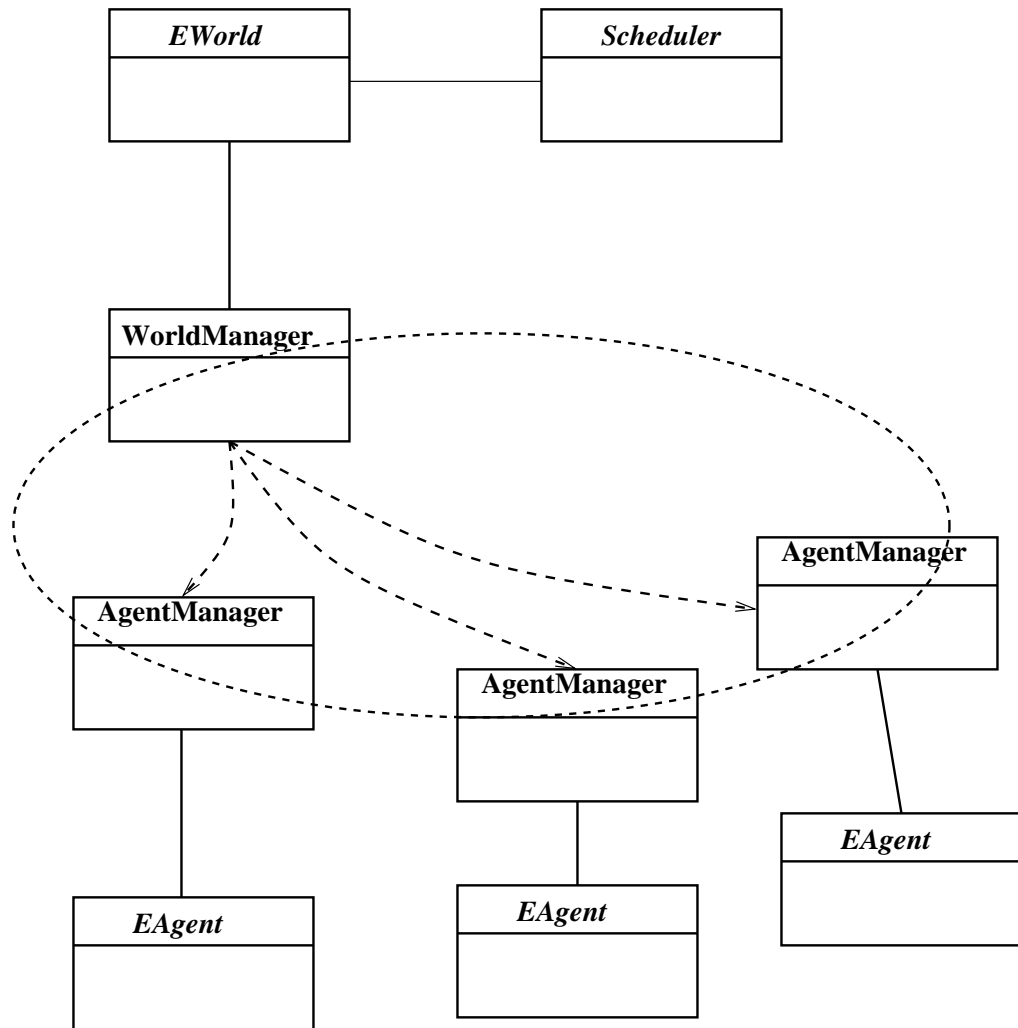


FIG. 20 – Scheduler avec médium

4.2 Le voisinage

C'est l'agent connaît ses voisins par son roleManager **AgentManager**.

1. Le monde initialise la phase de construction des voisinages dès qu'il sait que tous les agents sont connectés.
2. L'agentManager possède une ArrayList d'identifiants de voisins. Il obtient cette liste en demandant au monde "qui sont les voisins de l'agent x ?" : **agentManager.setNeighborhood("type de voisinage")** appelle **worldManager.buildNeighborhood("type de voisinage")**. Le monde construit la liste grâce au zoneSelector.

Evolutions à prévoir

- Arrivée d'agents en cours de simulation
- Créer un rôle "neighbour" qui permettrait à un agent de savoir de qui il est le voisin (rôle attribué par l'initiateur du voisinage). Ce rôle permettrait aussi de créer des voisinages symétriques (random pairwise par exemple).

Répercussions sur les agents C'est le médium qui effectue les calculs relatifs au voisinage. Un agent qui veut connaître la proportion d'agents se trouvant dans l'état xxx demande au medium de la calculer. Ce n'est plus l'agent qui effectue cette tâche au niveau de sa méthode compute.

J'ai implanté une première méthode **getNeighboursProportion(String valueType, Object value)** qui demande la proportions d'agents dont l'état de type *valueType* est égal à *value*. C'est une méthode de l'**AgentManager**.

En fonction des besoins des simulations il sera peut être nécessaire de la faire évoluer ou d'en créer d'autres.

4.3 Les graphiques

4.3.1 La classe Canevas

Ce panel est attaché au monde. Pour connaître l'état d'un agent il doit s'adresser au monde. Pour cela le monde possède une méthode **getAgentState(int)** ayant pour paramètre l'indice de l'agent dans liste. Le monde peut alors connaître l'identifiant de l'agent et lui demander sont état via le médium.

4.3.2 La classe Graphique

Ici ça se complique. Les graphiques appellent dynamiquement des méthodes sur les agents. Il faut aussi modifier ces appels pour qu'ils se fassent via le medium. Je n'ai pas encore effectué les modifications nécessaires. A priori, c'est le statManager qu'il faut revoir et essentiellement ses appels dynamiques vers les agents.

4.4 Le Marché virtuel

Je n'ai pas pu implémenter cette partie mais tout est en place pour implémenter cette partie.

4.4.1 Le monde et les agents

On garde ce qui a été décrit plus haut.

4.4.2 Les entreprises

Les entreprises sont des agents ayant un rôle différent. L'entreprise ou **Competitor** se connecte au médium avec le rôle "competitor" (figure 21)

Les échanges entreprises/agents ou entreprise/monde se font par les `roleManager` respectifs.

Pour l'instant les entreprises ne sont pas représentées à l'écran. Il sera souhaitable de les intégrer dans la liste des agents et de les faire évoluer de la même façon. (prévoir peut être la notion d'héritage de rôle (*competitor* hérite de *agent*) pour éviter de dupliquer les opérations).

Le principal problème vient du nombre non fixé d'entreprises et donc du nombre d'`AdditionalPanels`. En fait les entreprises étant des agents, il serait souhaitable de les voir apparaître à l'écran parmi les autres agents (couleur différente) et de leur attribuer un `AgentEditor` spécifique. Cet `AgentEditor` serait affiché automatiquement à l'initialisation et permettrait de saisir les valeurs liées aux entreprises.

4.5 Le paquetage medium

Le découpage en paquetage de `moduleco` reste le même. Seul le paquetage `medium` change. Il contient à présent tout ce qui concerne les accès au framework `medium` qui est une entité à part de `moduleco` (la couche de communication peut évoluer). Nous allons donc y trouver :

- **Registry** : Thread qui centralise les connexions des composants
- **IAgentServiceMedium** : interface définissant le rôle d'agent
- **IWorldServiceMedium** : interface définissant le rôle de monde
- **AgentManager** : lie l'agent au médium
- **WorldManager** : lie le monde au médium
- **ModulecoApplicationContract** : précise les règles de connexion à l'application `moduleco`
- **ModulecoMediumContract** : précise les règles de connexion pour chaque rôle.

4.6 Premiers tests

Le premier problème survenu est lié à l'utilisation de la couche de communication IP sur une machine locale. Un nombre trop important d'agents a impliqué trop d'ouvertures de sockets IP sur la même machine.

Il fallait donc prévoir une utilisation locale du framework. C'est pour cela qu'il intègre maintenant une couche de communication locale. Cependant Des problèmes subsistent en raison d'un nombre trop important de thread.

L'étape suivante est donc de voir s'il n'est pas possible de créer une version sans threads. `Moduleco` pourra ainsi fonctionner en local.

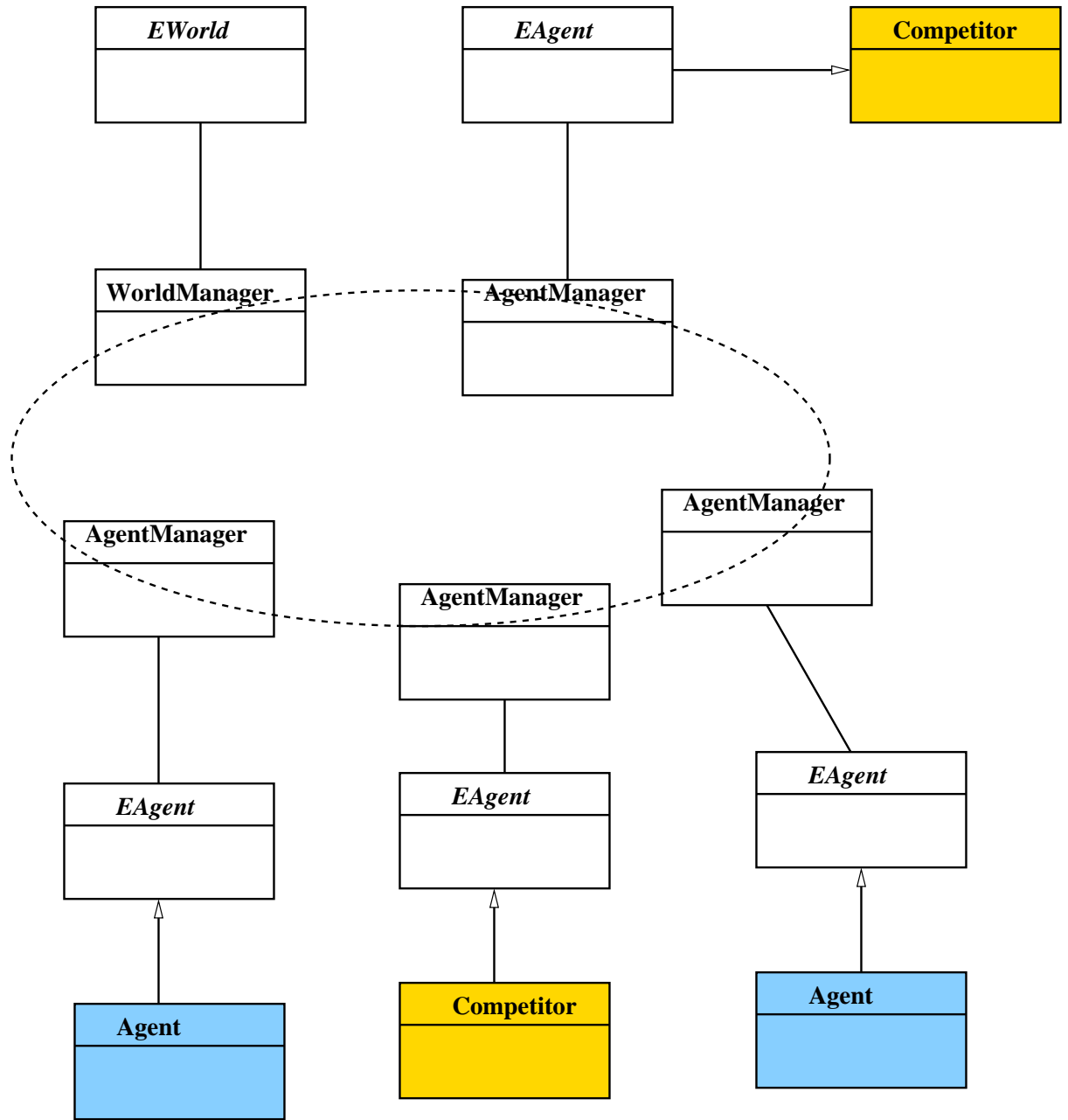


FIG. 21 – market avec médium

Ensuite, si l'on veut distribuer l'application, seule la couche de communication sera changée. On pourra même prévoir de réunir partie locale et une partie distante (IP ou Corba).

5 Conclusion

Le concept de médium s'applique parfaitement aux systèmes multi-agent. L'utilisation d'un package existant comme le framework médium permet aux développeurs de moduleco de ne se concentrer que sur la partie comportement des agents en utilisant au maximum ses possibilités de communication.

Après la phase de construction et de validation du prototype en implémentant des modèles existants, s'ouvre une deuxième phase de développement du prototype incluant les médiums.

J'ai souhaité rester le plus près possible du modèle initial en gardant le monde attaché à l'ordonnanceur et à l'IHM. Les premiers tests donnent des résultats concluants. Il faudra ensuite faire évoluer le modèle en séparant ses notions. L'IHM doit se connecter au médium pour échanger les informations avec l'utilisateur. De la même façon, l'ordonnanceur est attaché au médium et coordonne l'ensemble. Le modèle de base de moduleco pourrait alors devenir : figure 22.

Ce qu'il reste à faire

- Résoudre les problèmes de performance au niveau du médium
- Afficher les courbes statistiques (attachées au monde dans un premier temps)
- Définir correctement les interfaces CAgent, IAgentManager, IWorldManager
- Evolution du modèle : désolidariser la vue et l'ordonnanceur du monde.

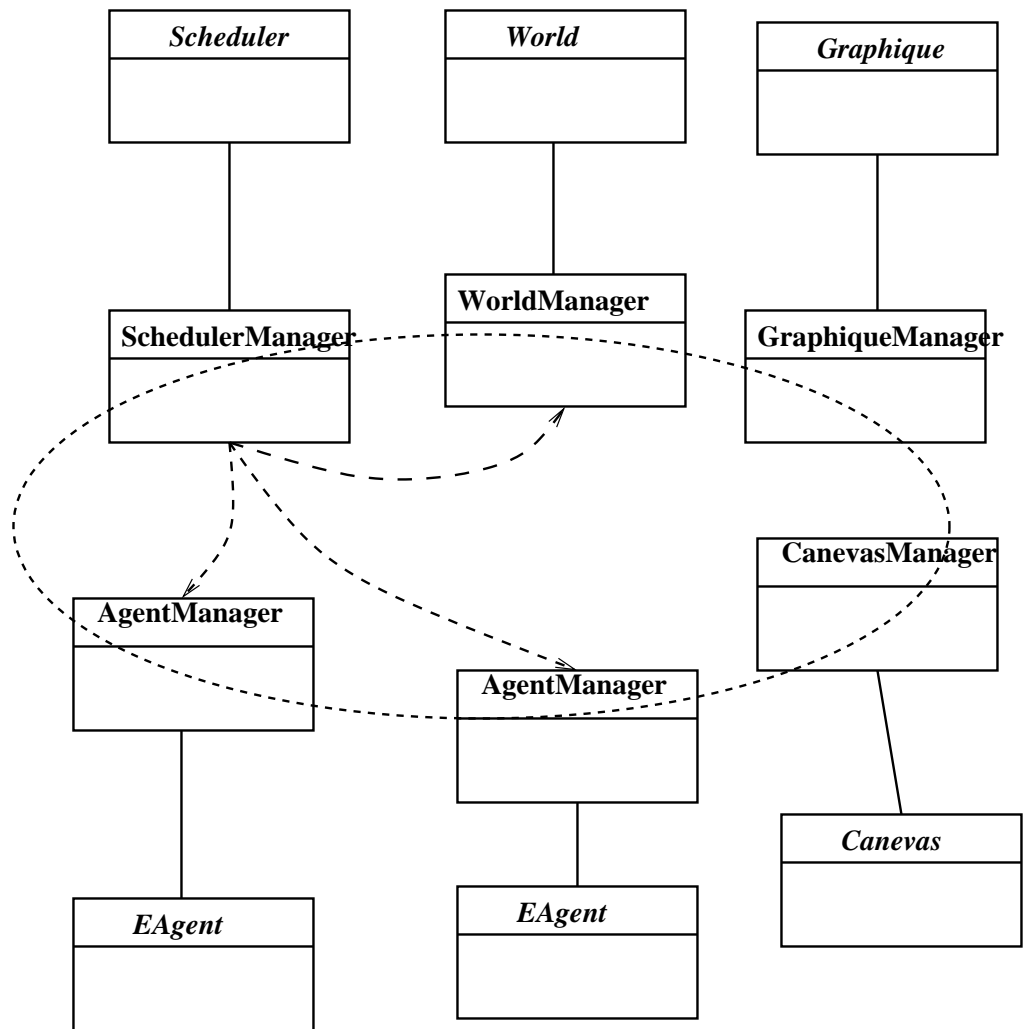


FIG. 22 – modeleco en séparant vue et ordonnancement