Informatica e Società

complemento d'esame

UML class diagrams of "Agent classes simulator" AESOP-ACP

Supervisore lavoro: Prof. Guido Fioretti
Studente: Hichri Mohamed Ali

Sommaire

AESOP-ACP framework programming
UML class diagrams of "Agent classes simulator"

Introduction

Ce document constitue une représentation des diagrammes des classes UML des agents constituants les deux pièces du progiciel **AESOP-ACP framework programming** avec une brève explication.

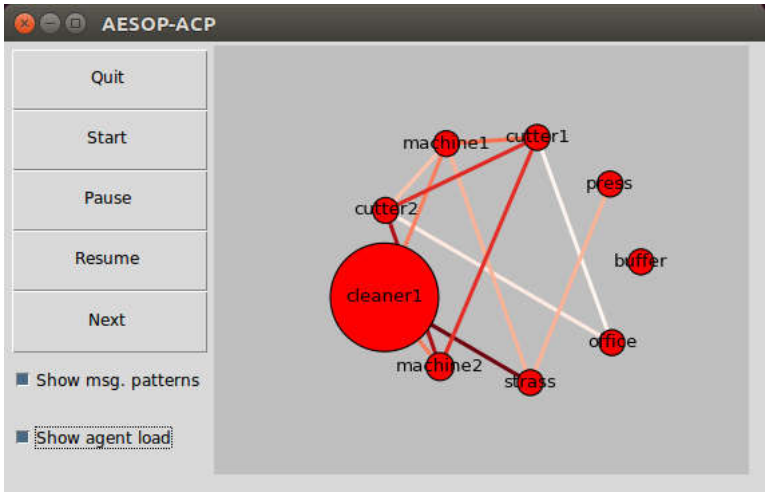
Pour faire voir le fonctionnement avec l'exemple «**The embroidery**» fourni dans cette distribution AESOP-ACP où un fragment de l'application est simulé, il y a des **objets** qui se cachent derrière, principalement des **Agents**.

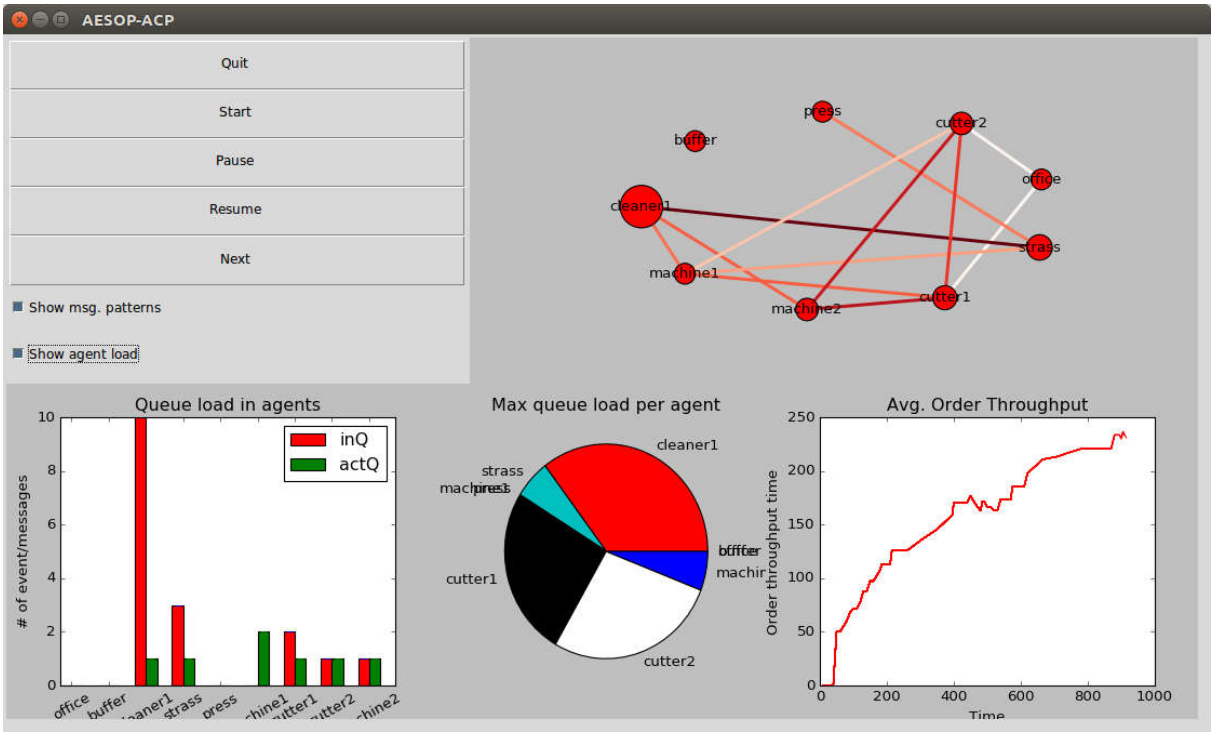
Ces « **Agents** » se sont les entités qui composent l'environnement virtuel de simulation. Chaque agent doit être une instance de la classe « **Agent** » ou l'une de ses sous-classes. C'est pour cela qu'on va s'intéresser dans tous ce qui suit aux classes « **Agents** », à leur modélisation conceptuelle UML en détaillant les différents diagrammes des classes d'objets qui les constituent.

1. Contexte de la modélisation

Durant la réalisation d'un projet, l'étape fondatrice reste toujours la conception et la modélisation.
Pareil pour l'application **AESOP-ACP** existante, qui est une application structurée et surtout modulable qui a été écrite dans le langage Python. Et pour ressortir la modélisation de cette application, il fallait faire l'étude du code existant avant d'éplucher toutes les classes et trouver les liaisons grâce aux créations d'objets.

Mais avant de construire tous ces diagrammes, il a d'abord fallu comprendre le fonctionnement de l'application. C'est pour cela, j'ai recouru également au fonctionnement de l'exemple « **embroidery** » après la configuration adéquate de telle sorte que tous les liens vers les librairies nécessaires soient corrects et ne posent aucun problème lors de l'interprétation et l'exécution.
La figure suivante donne un aperçu sur l'interface de l'exécution de l'exemple « The embroidery » à un instant t.





Cette seconde figure, représente l'interface graphique de production dans le cadre AESOP-ACP. Les trois nouveaux widgets en bas indiquent respectivement :

(i) La charge de la file d'attente entrante en cours de traitement pour chaque agent

(ii) La charge maximale atteinte par chaque agent

(iii) Le temps de traitement de la commande.

2. Diagramme de classes principal des « Agents »

L'application AESOP-ACP existante est très complète au niveau des fonctionnalités implémentées et surtout elle est riche et composée de nombreux modules, c'est pour cela on se délimite juste à la modélisation des « Agents » qui présentent le module le plus important, en donnant dans un premier lieu le diagramme de classes principal et ensuite on détaillera toute classe et liaison avec une brève explication.

Le diagramme de classes principal complet des « Agents » est donné par la figure suivante :

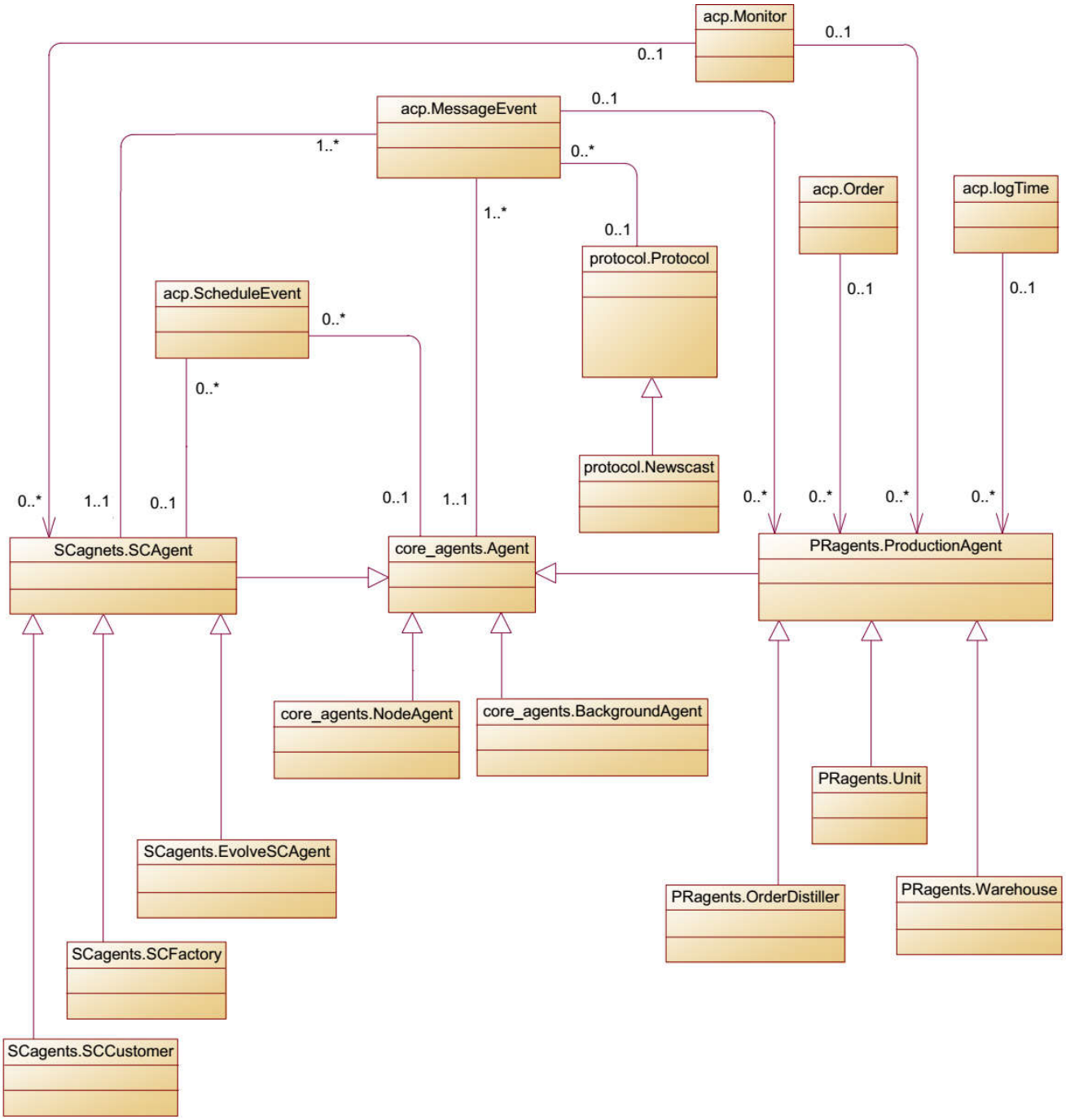


Diagramme de classe principal des "Agents"

Ce diagramme de classes principal « **Agents** » donne une vue statique du simulateur, il décrit les types et les objets qui le composent.

Le modèle de simulation AESOP-ACP est basé sur l'abstraction de l'aspect temporel et des événements dont les entités qui composent l'environnement virtuel de simulation sont principalement des **agents**. Chaque agent doit être une instance de la classe « Agent » ou de sa sous-classe.

Ce diagramme de classes ne contient ni les méthodes ni les attributs, car ils seront évoqués dans ce qui suit.

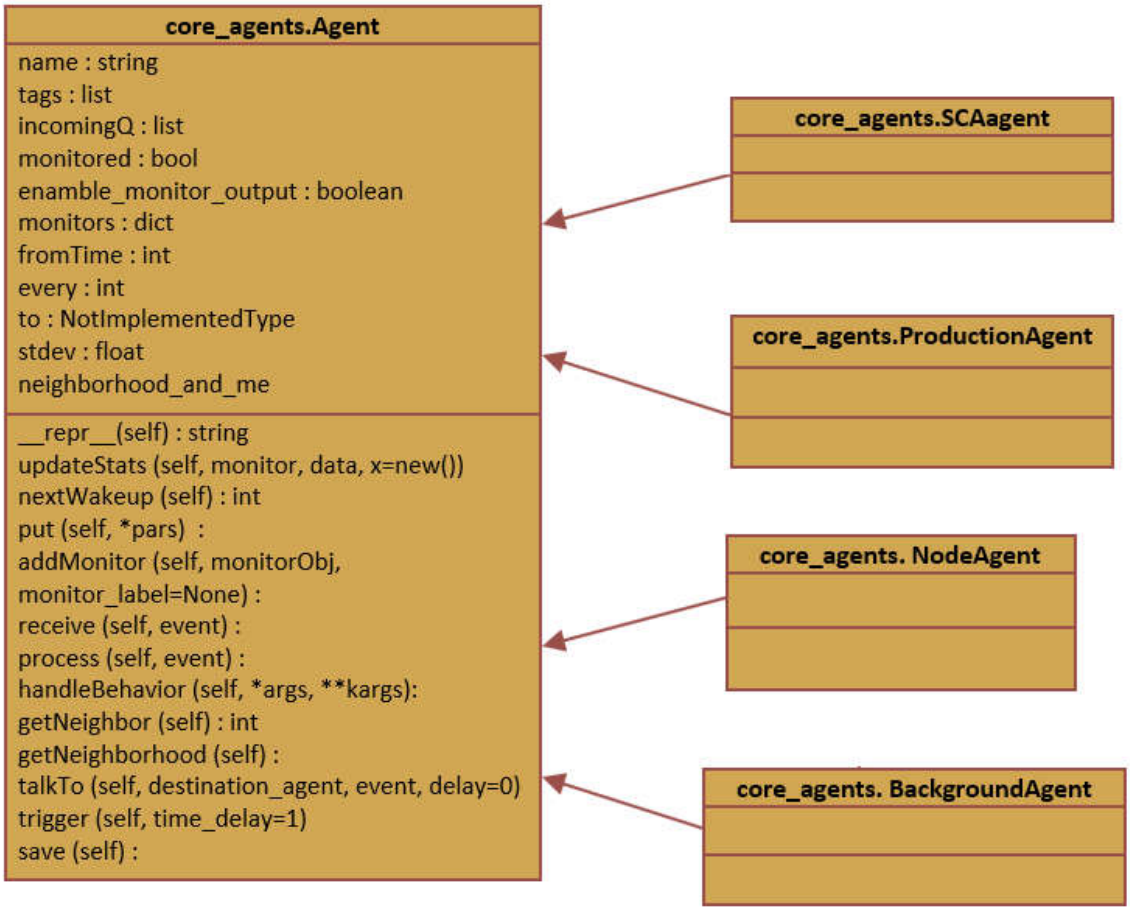
3. Diagramme de classes des « Agents » - Détails et fonctionnalités

Les entités dans le diagramme de classe élaboré ci-dessus, se sont des agents actives, autonomes et qui sont capables de mettre en œuvre certains comportements pour répondre à d'autres agents. Se sont des agents principalement réactifs dont les connaissances ne sont pas modifiables mais ont des objectifs précis et des comportements spécifiques dans le cas de l'occurrence d'actions dans l'environnement de simulation AESOP-ACP.

3.1 La classe « Agent »

La classe « Agent » est l'entité de base la plus importante qui est héritée de la super-classe « object ».

La figure suivante illustre la classe « **Agent** » avec ses sous-classes « **BackgroundAgent** », « **NodeAgent** » , « **SCAgent** » et « **ProductionAgent** » qui composent entre autre le cœur du simulateur AESOP-ACP .



Les agents communiquent par envoi de messages, leurs identiants (nom), leur permettent de s'envoyer des messages en précisant l'agent destinataire et l'emetteur du message. Tout grâce à la méthode « **talkTo()** » qui envoie un événement de message à destination agent référencié par son nom. Il convertit la chaîne de nom destination agent en object. Par défaut, il envoie un message instantanément avec un délai delay = 0. Ces derniers interagissent dans cet environnement constituant ainsi un système multi-agents, où chaque agent possède un rôle bien déterminé et doit être capable de finir la tâche qui lui a été attribuée pour atteindre les objectifs attendus de la simulation.

Concernant l'explication de diagramme de classe, on s'intéresse plus particulièrement aux méthodes et aux liaisons et interactions (communication) entre les entités (agents). En effet, l'explication des méthodes et les liaisons permet de mieux comprendre le fonctionnement et le comportement de différentes entités notamment les entités "Agents" où leurs méthodes peuvent être appelées par d'autres agents, et les agents peuvent se coordonner les uns avec les autres en utilisant le moyen de communication par message pour qu'ils régissent proactivement afin de réaliser le but de simulateur AESOP-ACP.

Nous pouvons noter que l'attribut le plus important est l'identifiant des agents. En effet chaque agent possède un nom unique (**name**) qui l'identifie.

Nous détaillons dans le tableau suivant les méthodes appropriées à la classe "Agent"

Méthodes de la classe « Agent »	
Méthode	**Rôle**
__repr__(…)	Retourne une présentation de l'agent sous forme de chaine de caractères.
updateStats(…)	Vérifie si l'agent en question est surveillé (monitored=True) il met à jour la date et le temps dans la liste Monitor.
nextWakeup(…)	Calcule le délai (delay) de la prochaine activation du comportement actif de l'agent déclanchée par la méthode nadleBehavior().
put(…)	Cette méthode peut accéder à n'importe quel champ de la planification en mode écriture ou lecture.
addMonitor(…**)**	Cette méthode permet d'enregistrer un moniteur dans le dictionnaire des agents. L'argument peut être un simple objet moniteur ou une liste
receive(…**)**	Cette méthode est appelée par par le moteur sim lorsqu'un nouveau message est délivré à l'agent. Cela place simplement l'événement entrant dans la file d'attente locale. L'événement est un objet "MessageEvent" et son contenu est spécifique au traitement.
process(…**)**	Cette méthode implémente un comportement «passif»: il réagit à un message reçu. La valeur par défaut n'imprime que l'événement (message), doit être redéfinie par les sous-classes (BackgroundAgent, NodeAgent, SCAgent, ProductionAgent). Notons bien que l'événement est un objet "MessageEvent" et son contenu est spécifique à la simulation.

handleBehavior(…)	Elle permet la gestion du comportement actif et le déclanchement du prochain réveil. Chaque classe fille de la classe Agent doit étendre cette méthode si elle doit ajouter un comportement "actif" spécifique. Elle est appelée par la méthode do() en réponse à un événement temporel de type "ScheduleEvent".
getNeighborhood(…)	Cette méthode permet de referencier le noeud voisine par leur identifiant "name".
talkTo(…)	Cette méthode envoie un événement de type message à agent destinataire référencié par son nom.
trigger(…)	Cette méthode doit être exécutée lorsqu'un nouvel agent est injecté dans l'environnement de simulation. Autrement, elle permet l'automatisation d'appel de la méhtode handleBehavior() à l'étape temporelle successive.
save(…)	Elle pemet d'enregistrer les données du moniteur sur un f_handle et mis à jour les fichiers existants.

3.2 Les sous-classes de la classe « Agent »

Comme nous avons montré précédemment qu'une entité "Agent" possède quatre instances (sous-classes). Chacun de ces quatre types d'entité est représenté par un agent.

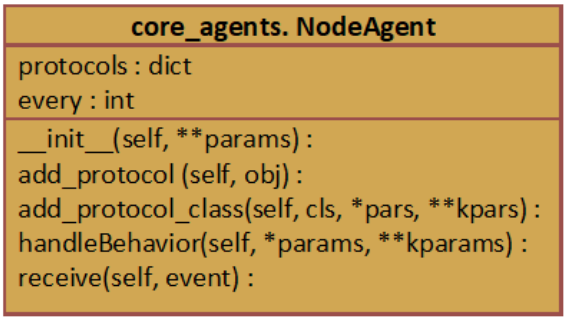
Les quatre agents « **NodeAgent** », « **BackgroundAgent** », « **SCAgent** » et « **ProductionAgent** » existent en deux niveaux et trois niveaux hiérarchiques, chaque agent commande les agents du niveau inférieur, et fournir l'agent du niveau supérieur par le biais de communication par messages et évènements.

Nous détaillons dans ce qui suit chaque entité instance "agent" en donnant la portion de diagramme de classe correspondante avec une brève explication des méthodes.

3.2.1 La sous-classe « NodeAgent »

Cette entité sous-classe "Agent" est spécialisée pour la simulation de réseau Peer-to-Peer. Elle permet l'ajout du support pour les classes de protocoles en jeu, où le travail réel de la simulation est effectué. Cette implémentation d'agent agit en tant que répartiteur pour les messages "MessaageEvents" qui portent le nom de protocole correspondant.

Lorsque AESOP est utilisé en tant que simulateur P2P, la classe "NodeAgent" se comporte comme un conteneur pour les protocoles qui effectuent réellement le calcul. Elle fait appel à la méthode "handleBehavior(…)" sur toutes ses instances de protocole. Elle fait appel à chaque protocole via la méthode process() de la classe mère "Agent". Cette classe est donnée par la figure suivante :



On voit bien que l'entité "NodeAgent" possède deux attributs :

protocols : de type dictionnaire qui sert à stocker les protocoles d'agents.

every : de type entier qui sert à remplacer la valeur de l'entité mère, généralement prend 1 ou 0

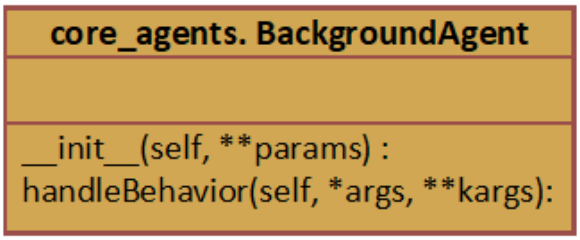
Pour les méthodes de cette implémentation agents, sont détaillées dans le tableau suivant :

Méthodes de la classe « NodeAgent »	
Méthode	**Rôle**
__init__(…)	Constructeur (initialisation des attributs et des paramètres effectifs)
add_protocol()	Permet l'ajout d'une instance protocole dans le dictionnaire des protocoles d'agent.
add_protocol_class()	Permet la création d'une instance de la classe de protocole à partir de son FQDN
handleBehavior()	Elle délègue une partie du traitement de l'appel à la classe "Agent" et elle exécute la partie du protocole actif.
receive(…)	Comme pour la classe mère, cette méthode est réimplémentée ici avec une légère modification toujours dans le but de placer un objet événement "MessageEvent" dans la file d'attente locale qui va servie pour la simulation.

3.2.2 La sous-classe « BackgroundAgent »

Cette entité sous-classe "Agent" s'occupe des agents qui font simplement partie de l'arrière-plan et ne participent pas à la simulation encours. En effet, lorsque l'interface graphique est active, certains agents ne sont nécessaires que pour remplir la représentation graphique, mais ne participent pas à l'interaction.

La classe "BackgroundAgent" est donnée par la figure suivante :



Pas des attributs appropriés pour cette classe, mais tous se fait au niveau de constructeur "__init__(…)" pour l'initialisation des attributs hérités surtout pour attribuer un nom à la classe comme identificateur unique.

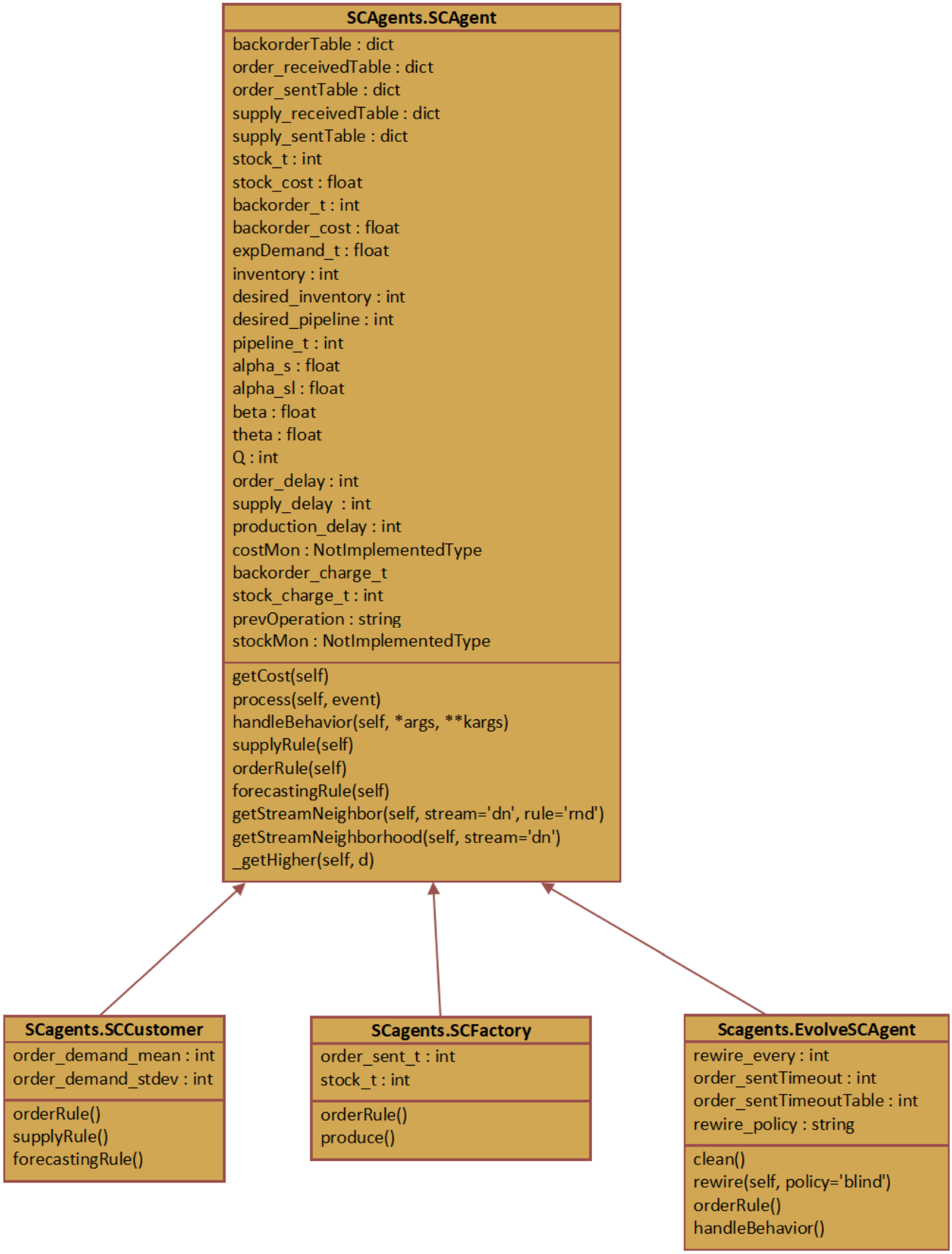
La méthode "handleBehavior(…)" joue le même rôle comme dans la classe mère "Agent" qui se manifeste par la gestion du comportement actif et le déclanchement du prochain réveil.

3.2.3 La sous-classe « SCAgent »

On peut classifier cette entité sour-classe "SCAgent " de la classe "Agent" comme un agent hybride possédant un croisement entre le comportement réactif et le comportement interactif puisque est la classe de base de la chaîne de simulation.

A partir de la classe principale "Agent", on arrive ici à trois niveaux hiérarchique vu que la sous-classe "SCAgent" possède trois instances de classe "SCCustomer", "SCFactory" et "EvolveSCAgent".

Pour avoir une vision plus claire, la figure suivante illustre la classe "SCAgent" et ses sous-classes :



Ainsi selon le diagramme de classe donné ci-dessus, et selon une vue structurale, les trois classes "SCCustomer", "SCFactory" et "EvolveSCAgent" sont héritées de la classe "SCAgent". Elles coopèrent entre eux dans l'objectif de la simulation selon une règle de conception qui permettra d'éviter la redondance et le chevauchement des informations circulantes au niveau de la chaîne soi-disant logistique de messages. C'est pour ça que ces trois classes ont une partie commune de méthodes et fonctions qui sont définies dans la classe mère "SCAgent" et une partie appropriée dans chaque classe fille.

On détaille dans ce qui suit les méthodes de chaque classe, pour les attributs sont définis avec leurs types dans la branche de diagramme de classe donnée par la figure précédente.

Méthodes de la classe « SCAgent »	
Méthode	**Rôle**
__init__(…)	Constructeur (initialisation des attributs et des paramètres fictifs)
getCost(…)	Renvoie le coût de l'inventaire à l'instant t, alias heure actuelle du système.
process(…)	Elle redéfinie la méthode process de la super-classe "Agent". Autrement, elle la remplace. Il compte essentiellement les demandes et les approvisionnements.
handleBehavior(…)	La méthode de la super-classe est redéfinie. Ici, l'agent doit trier les événements de message reçus en fonction de leur contenu. Ensuite, il met en œuvre sa stratégie de comportement «actif».
supplyRule(…)	Elle détermine l'agent cible d'approvisionnement et calcule le coût à envoyer.
orderRule(…)	Calcule la valeur à commander depuis l'amont.
forecastingRule(…)	Calcule les prévisions sur les prochaines demandes.
getStreamNeighbor(…)	Cette méthode permet de renvoyer le nom du nœud voisin aval. Elle calcule autrement le chemin entre le nœud producteur et le nœud receveur (client).
getStreamNeighborhood(…)	Elle renvoie le voisinage de l'agent actuel en fonction du flux sélectionné. L'ensemble des voisinages se trouve dans une liste d'agents identifiés par noms.
_getHigher(d)	Retourne la valeur la plus haute du dictionnaire 'd' passé en paramètre.

a. La sous-sous-classe «SCCustomer»

Les agents de cette sous-sous-classe de la classe "Agent" servent à passer des ordres en amont et à recevoir les approvisionnements. Fondamentalement, c'est un générateur d'ordres.

Méthodes de la classe « SCCustomer»	
Méthode	**Rôle**
__init__(…)	Constructeur (initialisation des attributs et des paramètres fictifs)

supplyRule(…)	La fonction supplyRule() ne fait rien à ce niveau, un travail sérieux s'effectue toujours en appelant la même méthode de la classe parent "SCAgent".
orderRule(…)	La méthode orderRule() est redéfinie ici pour générer un nombre aléatoire d'ordres selon une distribution gaussienne bien définie.
forecastingRule(…)	Au niveau de la classe "SCAgent" la méthode forecastingRule() ne fait rien et comme toutes les méthodes de cette classe l'appel se fait à la même fonction de la classe mère quand il y a un traitement à faire.

b. La sous-sous-classe « SCFactory»

Les agents de cette classe peuvent seulement recevoir des commandes et générer des fournitures en aval. En outre, un agent de production génère un "MessageEvents" (envoie à lui-même) pour la production des éléments de simulation

Méthodes de la classe « SCFactory »	
Méthode	**Rôle**
__init__(…)	Constructeur (initialisation des attributs et des paramètres fictifs)
orderRule(…)	Cette méthode permet de déclencher une production en utilisant Une usine qui ne peut pas satisfaire les commandes, déclenche la production en utilisant "produire" EventMessages.
produce(…)	Au niveau de la classe "SCAgent" la méthode forecastingRule() ne fait rien et comme toutes les méthodes de cette classe l'appel se fait à la même fonction de la classe mère quand il y a un traitement à faire.

c. La sous-sous-classe « EvolveSCAgent »

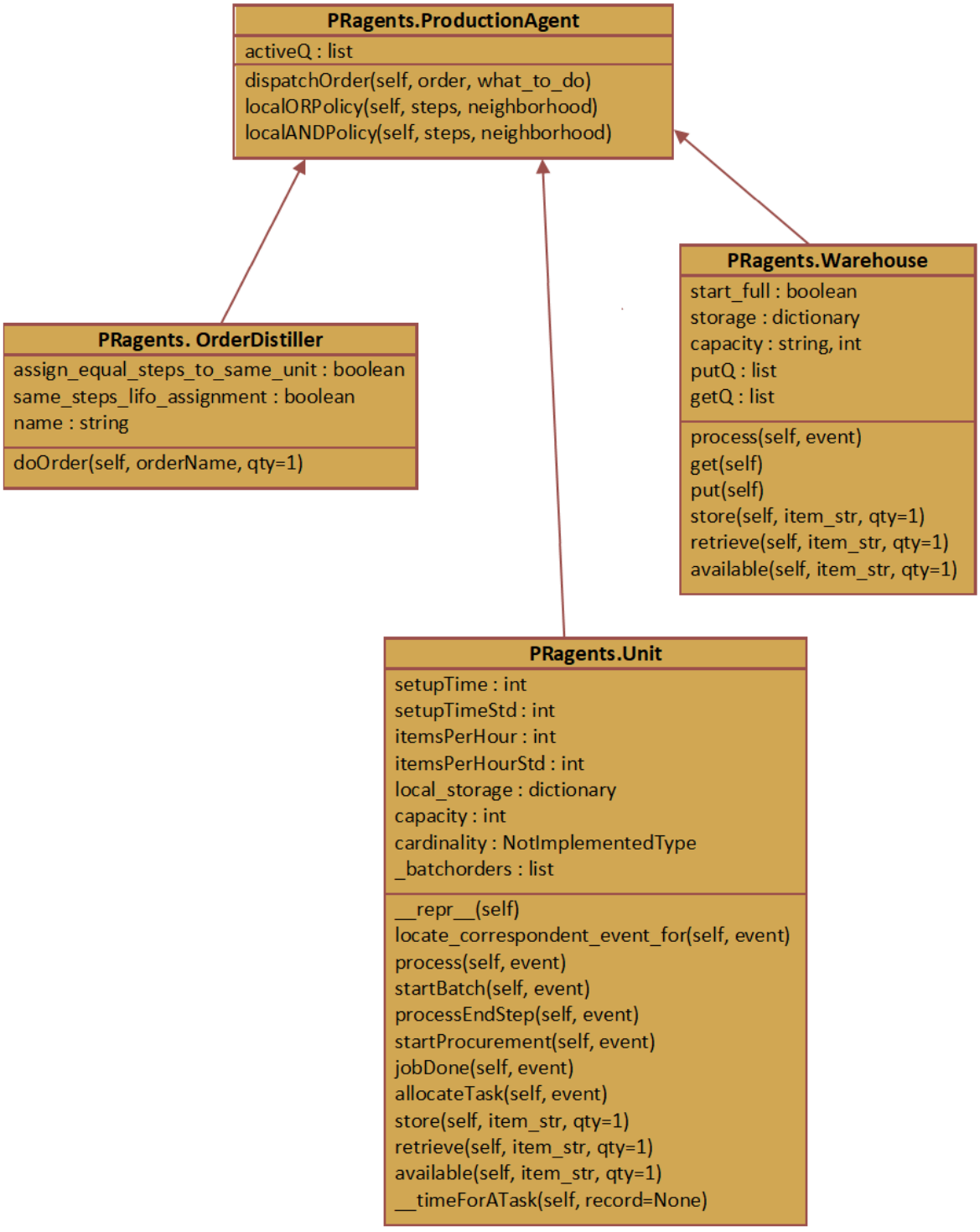
Méthodes de la classe « EvolveSCAgent »	
Méthode	**Rôle**
__init__(…)	Constructeur
clean()	Cette méthode se charge de la suppression des noeuds voisins qui n'ont reçu aucun ordre depuis l'unité du temps "oreder_sentTimeout".
rewire(self, policy='blind')	Permet d'obtenir un candidat au hasard parmi ceux qui ont la même étiquette (même classe) et qui ne se trouve pas dans le voisinage et lié avec celui-ci. Le principe de la politique «aveugle»: très basique.
orderRule()	La méthode orderRule() a été redéfinie à ce niveau pour mettre à jour les ts correspondants aux nœuds voisins.
handleBehavior()	Cette méthode a été redéfinie ici dans le but d'ignorer le comportement de la simulation dynamique (std) en nettoyant et en recâblant sa topologie à des intervalles réguliers.

3.2.4 La sous-classe « ProductionAgent »

La classe "ProductionAgent" est la classe principale dans le module de la simulation, elle définit les méthodes de répartition des ordres et des méthodes abstraites de personnalisation du comportement permettant de gérer les instructions et les ordres. C'est la classe de base des agents spécifiques à la simulation graphique.

Un agent de cette classe envoi un ordre à l'agent successeur sur le graphe suivant des caractéristiques demandées ou bien selon les étiquettes (tags) indiqués par les agents.

La figure suivante détaille cette classe avec ses trois classes dérivées :



La classe « ProductionAgent » possède trois méthodes :

Méthodes de la classe « ProductionAgent »	
Méthode	**Rôle**
dispatchOrder(…)	Cette méthode permet d'envoyer l'ordre aux bons agents. Les noms des agents et la prochaine étape à accomplir sont spécifiés dans le dictionnaire d'arguments 'what_to_do' passé en paramètre.
localORPolicy(…)	Cette méthode ne fait rien à ce niveau, mais elle est surchargée par les sous-classes. Elle permet entre autre la gestion de l'instruction 'OR' dans la liste de recettes.
localANDPolicy(…)	Comme pour la méthode précédente (localORPolicy()), cette méthode ne fait rien dans la classe "ProductionAgent" mais peut être surchargée dans les sous-classe pour assurer la gestion de l'instruction 'AND' dans les recettes.

a. La sous-sous-classe «OrderDistiller»

Cette classe génère des objets d'ordre et gère les messages de chaque recette et les envoie aux agents responsables. Elle reçoit les messages avec les ordres au format texte depuis le planificateur (scheduler) de simulateur au déclanchement de la simulation. Notons qu'au moins un "OrderDistiller" est requis dans une simulation.

Cette classe dispose d'une seule méthode "*doOrder(self, orderName, qty=1)*" qui est la méthode qui sera appelée par défaut lorsque l'agent reçoit un ordre. Cette méthode se charge d'envoyer un ordre à plusieurs agents au cas où des actions simultanées sont requises. L'ordre est empaqueté dans un objet "ScheduleEvent" tout en indiquant le nom de l'ordre et la quantité adéquate.

b. La sous-sous-classe «Unit»

Cette classe implémente un agent d'unité de production. Les messages directs mis en attente sont pris en compte à un temps donné par un objet cette classe selon son comportement temporel. Car il prend du temps pour mener bien sa tâche dans le jeu de la simulation. Sa réaction est plus ou moins autonome de l'objet et/ou de la communication à d'autres objets.

Le tableau suivant détaille les méthodes de cette classe :

Méthodes de la classe « Unit »	
Méthode	**Rôle**
__init__(…)	Constructeur
__repr__(self)	Cette méthode retourne une brève description de l'état de l'objet relativement aux messages actifs.

locate_correspondent_event_for(self, event)	Cette méthode permet de trouver l'événement correspondant de l'événement argument. Essentiellement, elle cherche l'événement qui est lié au même ordre. Elle est requise dans les transactions asynchrones lorsqu'elle reçoit une réponse ou un signal de confirmation.
process(self, event)	La méthode process() est redéfinie ici pour une tâche différente de celle dans les super-classes. En effet, elle permet ici de réagir aux ordres des évenements pour: - Allouer une nouvelle tâche si suffisament de ressources sont disponibles. - Acheminer une tâche expirée vers l'agent successeur. - suspendre une tâche en cours d'exécution et la retourner à la waitQ.
startBatch(self, event)	Se charge du traitement des instructions par lots dans la recette.
processEndStep(self, event)	Traite la déclaration de la dernière étape dans la recette. Cette implémentation considère une interaction synchrone avec un agent "warehouse".
startProcurement(self, event)	Traite la déclaration d'approvisionnement dans la recette. Elle Implémente une interaction synchrone avec la classe "warehouse".
jobDone(self, event)	Comme son nom l'indique, cette méthode permet d'informer sous forme d'un message à afficher si le traitement est accompli.
allocateTask(self, event)	Cette méthode attribue une tâche ou l'étape de recette et mettre à jour les statistiques. Elle génère un message à lui-même pour supprimer la tâche lorsque celle-ci est terminée.
store(self, item_str, qty=1)	C'est une méthode marchandise, elle est utile pour une simulation simple lors de l'accès à "warehouse" en mode synchrone.
retrieve(self, item_str, qty=1)	Elle renvoie True si la quantité souhaitée est disponible dans la mémoire, False sinon puis elle supprime la quantité requise de l'inventaire.
available(self, item_str, qty=1)	Elle renvoie True si la quantité souhaitée est disponible dans la mémoire, False sinon. Cette méthode est appelée lors d'une tâche de simulation simple et lorsque le temps de stockage et de récupération n'a pas d'importance.
__timeForATask(self, record=None)	Elle renvoie l'heure en unités de temps de simulation pour accomplir l'enregistrement pas à pas.

c. La sous-sous-classe «Wirehouse»

Cette sous-sous-classe de la classe "Agent" représente un agent "Wirehouse" (entrepôt). Elle peut également imiter un approvisionnement externe ou interne en utilisant un comportement déclenché par le temps fourni par la méthode do().

Un agent "Wirehouse" utilise des méthodes (comme get() et put()) pour régir aux commandes des événements.

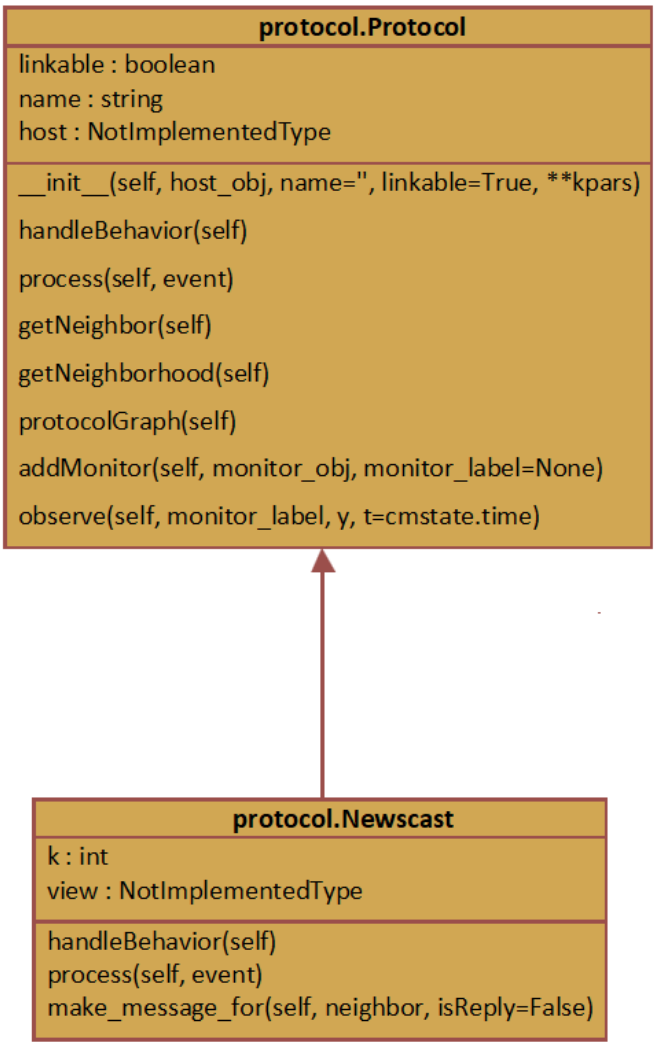
Le tableau suivant résume les méthodes définies par cette classe :

Méthodes de la classe « Wirehouse »	
Méthode	**Rôle**
__init__(self, capacity='unbounded', start_full=False, **params)	constructeur
process(self, event)	Redéfinie ici pour mettre à jour l'évènement passé en paramètre.
get(self)	Cette méthode gère les demande d'évenements de type 'get'. Elle permet d'allouer une nouvelle tâche si les ressources requises sont disponibles en répondant par un messagne contenant la liste des éléments, sino la demande reste en attente.
put(self)	Cette méthode prend en charge une tâche déjà allouée et qui a été expirée pour la router vers l'agent suivant. Autrement, elle gère une demande d'événement de type 'set'.
store(self, item_str, qty=1)	Elle renvoie True si la quantité souhaitée est disponible dans la mémoire, False sinon. Cette méthode est appelée lors d'une tâche de simulation simple et lorsque le temps de stockage et de récupération n'a pas d'importance.
retrieve(self, item_str, qty=1	Renvoie "True" si la quantité de ressources souhaitées est disponible dans la mémoire, "False" sinon.
available(self, item_str, qty=1)	Cette méthode est appelée lors d'une tâche de simulation simple et lorsque le temps de stockage et de récupération n'a pas d'importance. Elle renvoie "True" si la quantité de ressources souhaitées est disponible dans la mémoire, "False" sinon.

3.3 La Classe « Protocol »

Ce module prend en charge les protocoles de réseau. C'est la classe qui tient le graphe de chaque protocole, elle se charge de mappage. En effet, chaque instance de protocole porte le même nom du graphe auquel elle appartient. Elle n'est pas liée à aucune classe de type "Agent" directement, mais elle communique avec toutes les classe par messages à travers la classe "MessageEvent". C'est une classe un peu indépendante des classes agents qui présente une instance de la super-classe "object".

Cette classe implémente un niveau d'instance simple, elle possède une seule classe descendante "Newcast". La portion de diagramme de classe relative à la classe "Protocol" et sa classe fille est illustrée par la figure suivante :



Pour accomplir sa tâche convenablement, la classe protocole fait appel à ses méthodes régulièrement :

Méthodes de la classe « Protocol »	
Méthode	**Rôle**
__init__(…)	Constructeur.
handleBehavior(self)	Cette méthode ne fait rien à ce niveau.
process(self, event)	Comme la méthode "handleBehavior()", cette méthode ne fait rien à ce niveau.
getNeighbor(self)	Permet de sélectionner un voisin de façon aléatoire de tous les nœuds de graphe.
getNeighborhood(self)	Retourne cmstate.sim.agentsGraph.neighbors(self.host.name)
protocolGraph(self)	Renvoie la référence du protocole correspond au graphe.
addMonitor(self, monitor_obj, monitor_label=None)	Cette méthode ajoute un objet de type "Monitor" à l'agent correspondant. Cet agent "Monitor" est référencié par son nom de type chaine de caractère (string) et représente aussi la base du nom du fichier avec lequel les données assemblées sont collectées par un collecteur externe.
observe(self, monitor_label, y, t=cmstate.time)	Cette méthode permet d'ajouter une observation au moniteur spécifié. En effet, les protocoles utilisent cette méthode pour collecter des données via des moniteurs.

La méthode de la classe descendante "Newcast" sont détaillées dans le tableau suivant :

Méthodes de la classe « Newcast »	
Méthode	**Rôle**
__init__(…)	Constructeur.
handleBehavior(self)	Cette méthode est surchargée ici pour l'activation du "newcast thread".
process(self, event)	Pareil pour la méthode process(), elle surchargée à ce niveau pour faire le travail inverse de la fonction 'handleBehavior()' qui se manifeste par l'inactivation de "newcast thread".
make_message_for(self, neighbor, isReply=False)	Cette méthode permet la compression d'un "MessageEvent" pour le protocole 'Newcast' en question puis le renvoie.

4 Les classes d'évènements

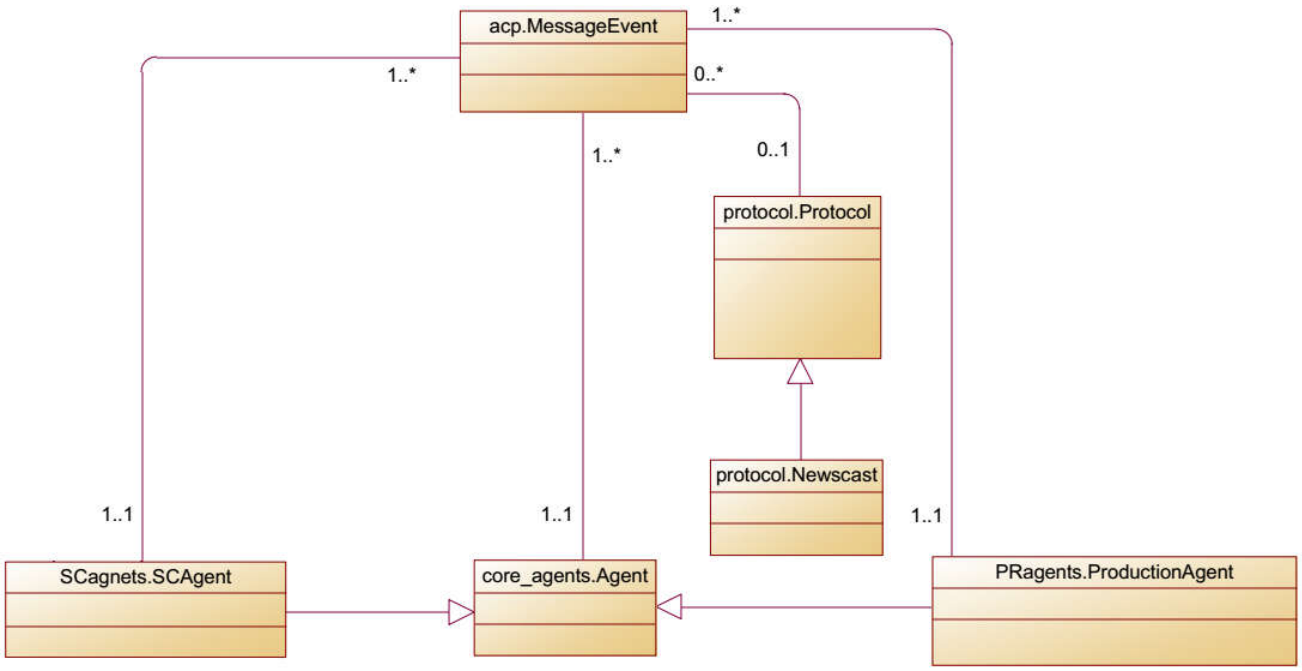
On ne peut pas parler de la modélisation des classes "Agent" sans parler des classes d'évènements qui jouent un rôle primordial dans le processus de simulation. En fait, durant la simulation (instance de simulation), le noyau extrait les événements de la file principale et les envoies aux agents de destination correspondants.

Les deux types d'événement qui son disponibles et qui ont l'importance vitale pour le cœur du simulateur AESOP-ACP sont : les instances de classe « MessageEvent » et « ScheduleEvent ».

4.1 La classe « MessageEvent »

Un événement est un message envoyé par un objet "Agent" pour signaler la présence d'une action. L'événement est généralement un membre de l'émetteur d'événements.

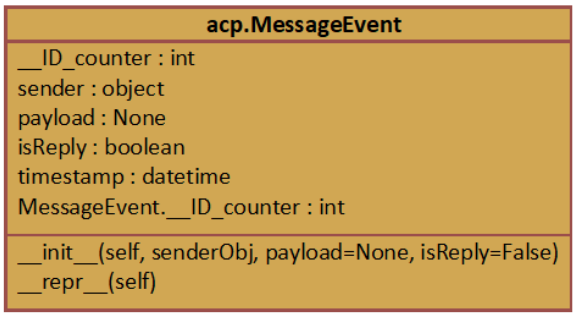
La figure suivante illustre cette classe avec les classes qui sont en relation avec :



Toute classe « Agent » peut avoir une ou plusieurs instances "MessageEvent", alors qu'une instance "MessageEvent" n'appartient qu'à un seul objet "Agent". Sauf pour la classe « Protocol », où un objet "Protocol" peut avoir ou non un "MessageEvent".

Un évènement message informe la cible (l'objet receveur) qu'un évènement a été déclenché tout en précisant l'émetteur.

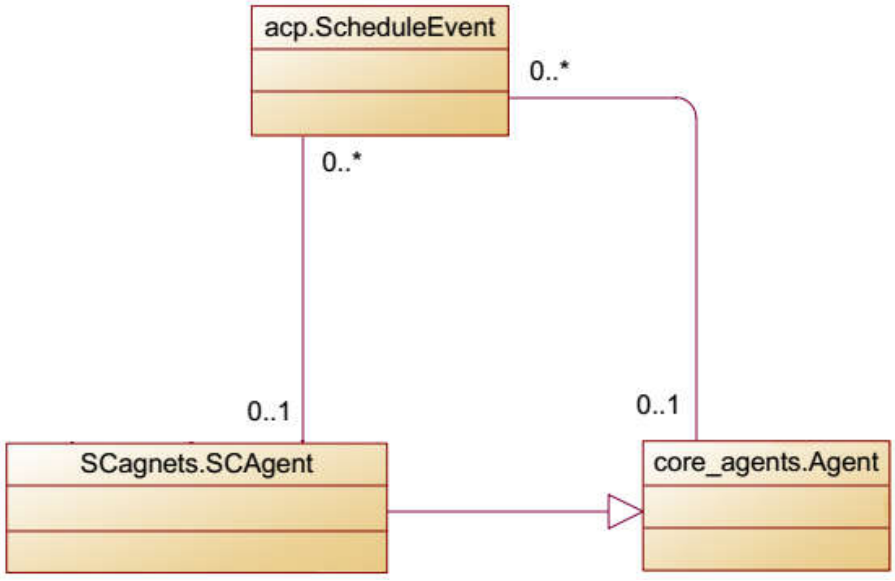
Un "MessageEvent"étant un événement qui implémente aussi ces propriétés et ses méthodes. La figure suivante donne un aperçu sur une instance de la classe « MessageEvent » :



Méthodes de la classe « MessageEvent »	
Méthode	**Rôle**
__init__(self, senderObj, payload=None, isReply=False)	Constructeur pour initialiser les propriétés de l'instance événement.
__init__(self, senderObj, payload=None, isReply=False)	Renvoie tous les détails sur le message évènement en question.

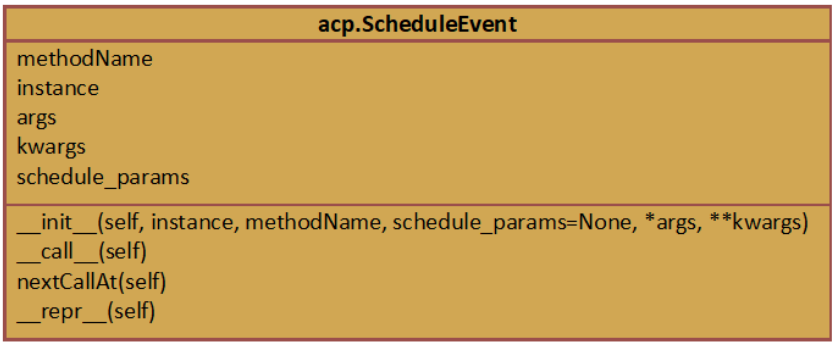
4.2 La classe « ScheduleEvent »

La classe "ScheduleEvent" sert à la signalisation du temps de l'événement déclenché par l'ordonnanceur (scheduler). Cette classe est en relation avec les deux classes « Agent » et « SCagent » comme le montre la figure suivante :



Une instance de la classe « Agent » ou la classe « SCAgent » peut avoir plusieurs "ScheduleEvent" comme ne peut avoir rien tout dépendant s'il y a un événement déclenché par l'objet agent en question.

La figure suivante détaille les propriétés et les méthodes de cette classe :



Et pour avoir une idée éclaircissante sur le comportement de cette classe, le tableau suivant détaille brièvement les méthodes de cette classe :

Méthodes de la classe « ScheduleEvent »	
Méthode	**Rôle**
__init__(self, instance, methodName, schedule_params=None, *args, **kwargs)	Constructeur pour initialiser les propriétés de l'instance ScheduleEvent.
__call__(self)	Cette méthode appelle l'événement afin de déclencher la méthode sur l'objet correspondant et génère le prochain événement à appeler.
nextCallAt(self)	Génère le prochain temps sur l'appel en cours.
__repr__(self)	Renvoie tous les détails sur l'évènement "ScheduleEvent" en question.