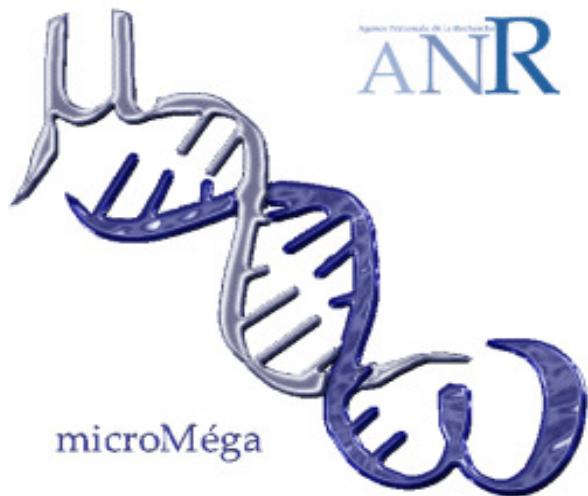


ANR-BLAN/NT05-2_42128



microMéga

Dossier de conception

Sommaire

1.Introduction.....	3
2.Structure générale.....	3
2.1.Arborescence.....	3
2.2.Dépendances externes.....	4
3.Le paquetage « core ».....	4
3.1.Aperçu.....	4
3.2.Paquetage « agent ».....	4
3.2.a Implémentations de IAgent et IAgentPool.....	6
3.2.b Les modules de Modulable.....	8
3.2.c Implémentation par défaut des modules de comportement.....	9
3.2.d Implémentation par défaut du module de représentation.....	10
3.2.e Implémentation par défaut du module IInteraction.....	10
3.3.Paquetage « simulation ».....	12
3.4.Paquetage « result ».....	13
3.5.Paquetage « util ».....	16
3.5.a Paquetage « xml ».....	16
3.5.b Paquetage « importation ».....	16
3.5.c Paquetage « log ».....	17
3.6.Le paquetage « solver ».....	18
4.Le paquetage « functional ».....	19
4.1.Aperçu.....	19
4.2.Les différents types d'agent.....	19
4.2.a Les agents réactionnels	19
4.2.b Les agents éléments.....	19
4.3.Paquetage « representation ».....	20
4.4.Paquetage « interaction ».....	21
4.5.Paquetage « behavior » (nominal).....	23
4.5.a Paquetage « util.activation ».....	23
4.6.Paquetage « behavior » (tuning).....	24
4.6.a Paquetage « util.activation.adaptive ».....	26
4.6.b Paquetage « util.activation.context.adaptive ».....	28
4.6.c Paquetage « util.parameterAdjuster ».....	29
5.Le paquetage « view ».....	29

1. Introduction

Le but de ce document est de présenter l'architecture de la plateforme microMéga développée dans le cadre du projet ANR de même nom.

La plateforme a pour objectif de pouvoir effectuer la simulation fonctionnelle de cellules vivantes en se basant sur la modélisation computationnelle des processus formant les voies métaboliques au sein de cette dernière. A cette fin, l'architecture de l'application est principalement composée d'une partie dédiée à la simulation fonctionnelle, d'outils pour la conception adaptative des modèles et d'une interface graphique permettant l'usage de ces fonctionnalités.

Ce document de conception décrit la structure générale du code source de la plateforme dans un premier temps, puis le cœur générique de l'application permettant la mise en œuvre des simulations, et enfin les paquetages plus spécialisés implémentant les différents types d'entités participant à la simulation et à la conception adaptative. Nous ne décrirons pas ici la partie interface graphique de la plateforme.

2. Structure générale

Nous exposons dans cette partie l'organisation du code source de la plateforme.

Remarque : Nous utilisons dans la notation des noms de paquetages, classes, méthodes, etc., les conventions d'écritures suivantes :

- Les noms de paquetage et de méthodes commencent par une minuscule ;
- Les noms des classes et interfaces commencent par une majuscule.

De plus, nous utilisons régulièrement les interfaces comme moyen de définition de types qui sont par la suite « implémentés » sous forme de classes concrètes ou abstraites. De fait, nous utiliseront beaucoup le polymorphisme de type et la délégation dans la manière de concevoir plutôt que l'héritage.

2.1. Arborescence

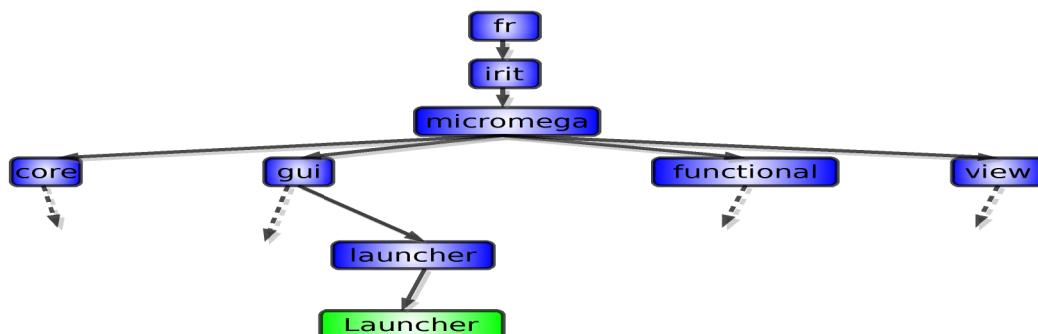


Figure 1: Arborescence générale

La Figure 1 décrit l'organisation des paquetages (rectangles bleus) depuis la racine du projet. Les quatre paquetages principaux (« core », « gui », « functional » et « view ») sont détaillés dans la suite de ce document.

Le paquetage « core » est dédié au cœur opérationnel de l'application. Il contient le code relatif aux agents (uniquement la partie générique), à la gestion de la simulation, au traitement des résultats ainsi que quelques utilitaires (logs, xml, imports...).

Le paquetage « gui » contient l'ensemble des éléments (ressources incluses) de l'interface graphique ainsi que la classe de lancement de l'application (fr.irit.micromega.gui.launcher.Launcher).

Le paquetage « functional » contient tout le code relatif aux agents fonctionnels de MicroMéga (à l'exception des parties génériques des agents qui sont dans « core »).

Le paquetage « view » contient tout le code relatif aux agents *vue* de MicroMéga (à l'exception des parties génériques des agents qui sont dans « core »).

2.2. Dépendances externes

Un certain nombre de bibliothèques externes sont utilisées :

1. jung : bibliothèque utilisée pour l'affichage des graphes (jung-1.7.6.jar)
2. jFreeChart : bibliothèque utilisée pour l'affichage des courbes (jfreechart-1.0.0.jar)
3. colt : bibliothèque utilisée par jung (colt.jar)
4. commons-collections : utilisée par jFreeChart (commons-collections-1.0.0.jar)
5. jcommon : utilisée par jFreeChart (jcommon-1.0.0.jar)

L'interface graphique utilisant la bibliothèque SWT qui est dépendante de la plateforme utilisée, la bibliothèque graphique est variable.

Attention : pour utiliser SWT, il faut que les versions de l'archive .jar et des bibliothèques système (.dll pour windows, .so pour linux) correspondent.

Lors de l'archivage en .jar du projet, il est conseillé d'utiliser un manifeste (qui sera dépendant du système d'explotation ciblé).

Exemple de manifeste pour la version linux 32 bits :

```
Manifest-Version: 1.0
Class-Path: lib/LINUXlib32/swt.jar lib/colt.jar lib/commons-collections-3.1.jar
lib/jcommon-1.0.0.jar lib/jfreechart-1.0.0.jar lib/jung-1.7.5.jar
Main-Class: fr.irit.micromega.gui.launcher.Launcher
```

3. Le paquetage « core »

Ce paquetage contient tous les éléments génériques permettant la mise en oeuvre de simulations fonctionnelles.

3.1. Aperçu

Le modèle sous-jacent utilisé pour les simulations dans la plateforme MicroMéga est basé sur le paradigme des systèmes multi-agents : chaque entité du système cible (ici les éléments et réactions chimiques participant aux voies métaboliques) est modélisée comme un agent autonome.

3.2. Paquetage « agent »

Ce paquetage contient tout le code générique des agents. Dans le diagramme 1 qui décrit l'interface *IAgent*, on peut voir que l'agent est défini par un nom, un type et un identifiant unique. L'agent

implémente les interfaces suivantes :

- *AgentRunnable* : permet de lancer le comportement nominal ou coopératif (adaptatif) de l'agent.
- *Modulable* : définit trois modules opérationnels de l'activité de l'agent : les représentations, les interactions et le comportement fonctionnel.
- *NcsValuable* : une mesure de non coopération est définie (*getNcsValue()*) renvoie la dernière valeur calculée, il n'y a pas de mise à jour).
- *NcsComparable* : permet de comparer les agents sur la base d'une mesure de non coopération.

De plus, l'interface *IAgentPool* définit l'interface pour gérer un ensemble d'agents. Outre les méthodes d'accès, d'ajout et de retrait, des méthodes supplémentaires sont définies :

- *renameAgent(String name, String newName)* : modifie le nom d'un agent.
- *UpdateReferences()* : les références à d'autres agents contenues dans tous les modules de représentation des agents connus sont mises à jour (méthode utilisée après chargement de fichier ou après renommage d'agents).
- *GetLastChange()* : indique la date de la dernière modification de l'ensemble des agents connus (ajout, retrait, renommage).
- *suspendLastChangeUpdating()/resumeLastChangeUpdating()* : permet de suspendre ou reprendre la mise à jour de la date de dernière modification.

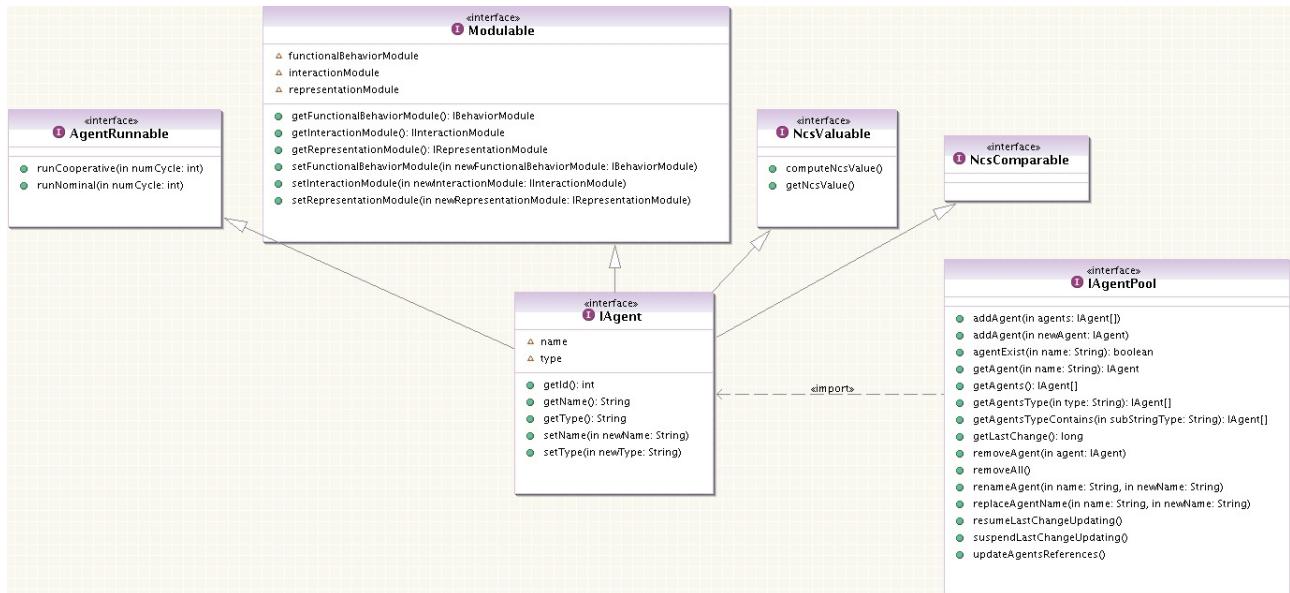


Diagramme 1: L'interface *IAgent*

3.2.a Implémentations de *IAgent* et *IAgentPool*

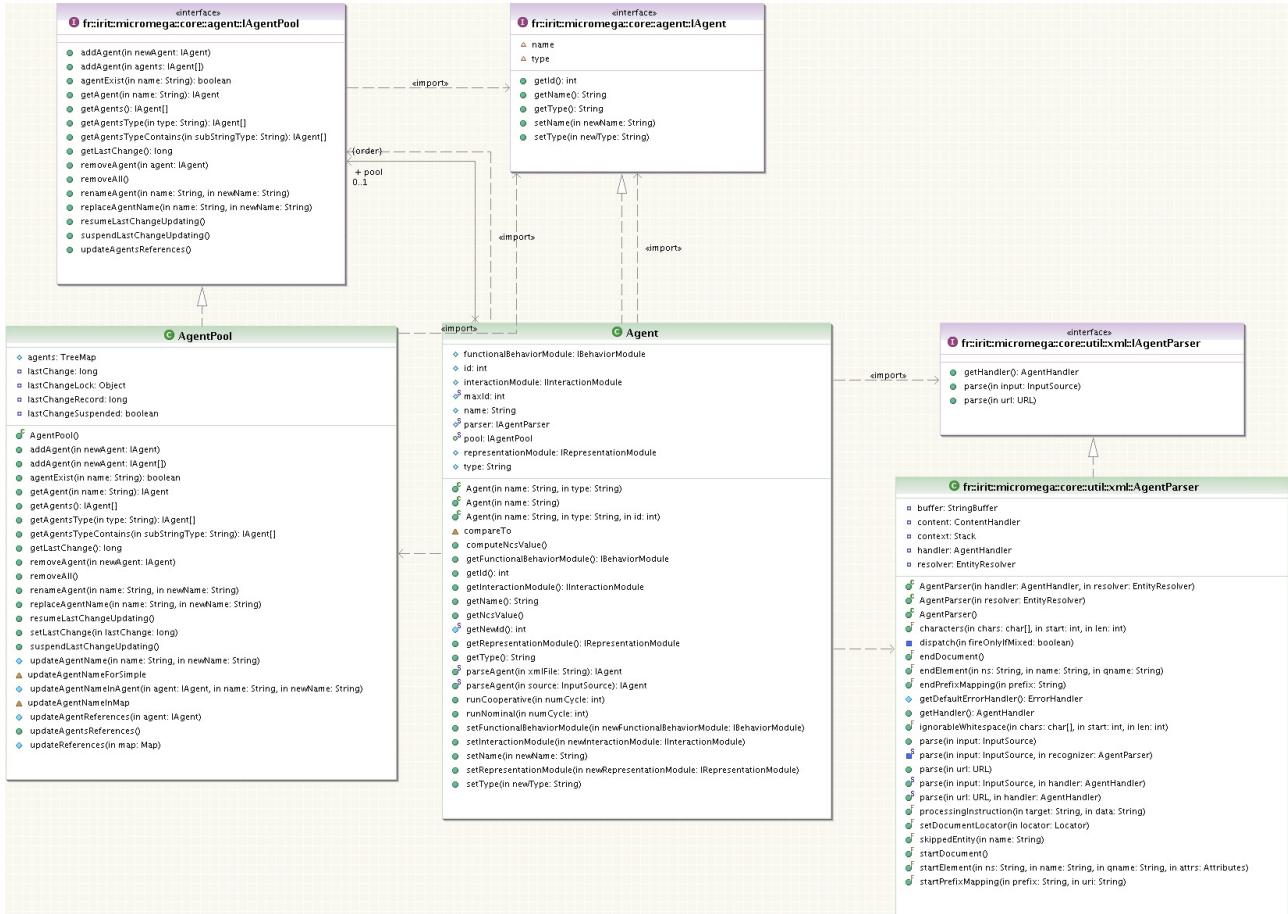


Diagramme 2: Implémentation de *IAgent* et *IAgentPool*

`fr.irit.micromega.core.agent.internal.Agent` est une implémentation de l'interface *IAgent*. Cette implémentation permet de créer un agent en se basant sur un fichier xml de description. A cette fin, le parser `fr.irit.micromega.core.util.xml.AgentParser` est utilisé. Ce parser est basé sur la dtd (grammaire de description) suivante :

```
<?xml encoding="UTF-8" ?>
<!ELEMENT Agent ( Data , Modules )>
<!ATTLIST Agent
    name CDATA #REQUIRED
    type CDATA #REQUIRED
>
<!ELEMENT Comment (#PCDATA)>
<!ATTLIST Comment
    locale (fr | en) #REQUIRED
>
<!ELEMENT Data ((Simple|Complex)*)>
<!ELEMENT Simple (Comment*)>
<!ATTLIST Simple
    name CDATA #REQUIRED
    type (int|float|double|boolean|String|IAgent) #REQUIRED
    isConstant (true|false) #IMPLIED
    value CDATA #IMPLIED
>
```

```

<!ELEMENT Complex (Comment*,(Simple|Complex)*)>
<!ATTLIST Complex
  name CDATA #REQUIRED
  type CDATA #IMPLIED
  isConstant (true|false) #IMPLIED
  value CDATA #IMPLIED
>
<!ELEMENT Modules (InteractionModule|RepresentationModule|
FunctionalBehaviorModule)*>
<!ELEMENT InteractionModule (Comment*)>
<!ATTLIST InteractionModule
  moduleClassName CDATA #REQUIRED
  moduleXMLFile CDATA #IMPLIED
>
<!ELEMENT RepresentationModule (Comment*)>
<!ATTLIST RepresentationModule
  moduleClassName CDATA #REQUIRED
  moduleXMLFile CDATA #IMPLIED
>
<!ELEMENT FunctionalBehaviorModule (Comment*)>
<!ATTLIST FunctionalBehaviorModule
  moduleClassName CDATA #REQUIRED
  moduleXMLFile CDATA #IMPLIED
>

```

Des exemples de définitions d'agent en xml sont disponibles dans les sous-paquettages de *fr.irit.micromega.fonctional*.

Il est à noter que d'autres parsers peuvent être utilisés (afin de parser des agents décrits par d'autres dtd) en utilisant l'interface *fr.irit.micromega.core.util.xml.IAgentParser*.

La classe *Agent* référence aussi l'ensemble des agents créés dans un *IAgentPool* accessible de manière statique (*Agent.pool*). Par défaut, la classe *fr.irit.micromega.core.agent.internal.AgentPool* est utilisée pour gérer cet ensemble.

fr.irit.micromega.core.agent.internal.AgentPool est une implémentation de l'interface *IAgentPool*. Les opérations liées au suivi de date de modification sont synchronisées dans cette implémentation. Attention : les autres opérations ne sont pas synchronisées.

3.2.b Les modules de *Modulable*

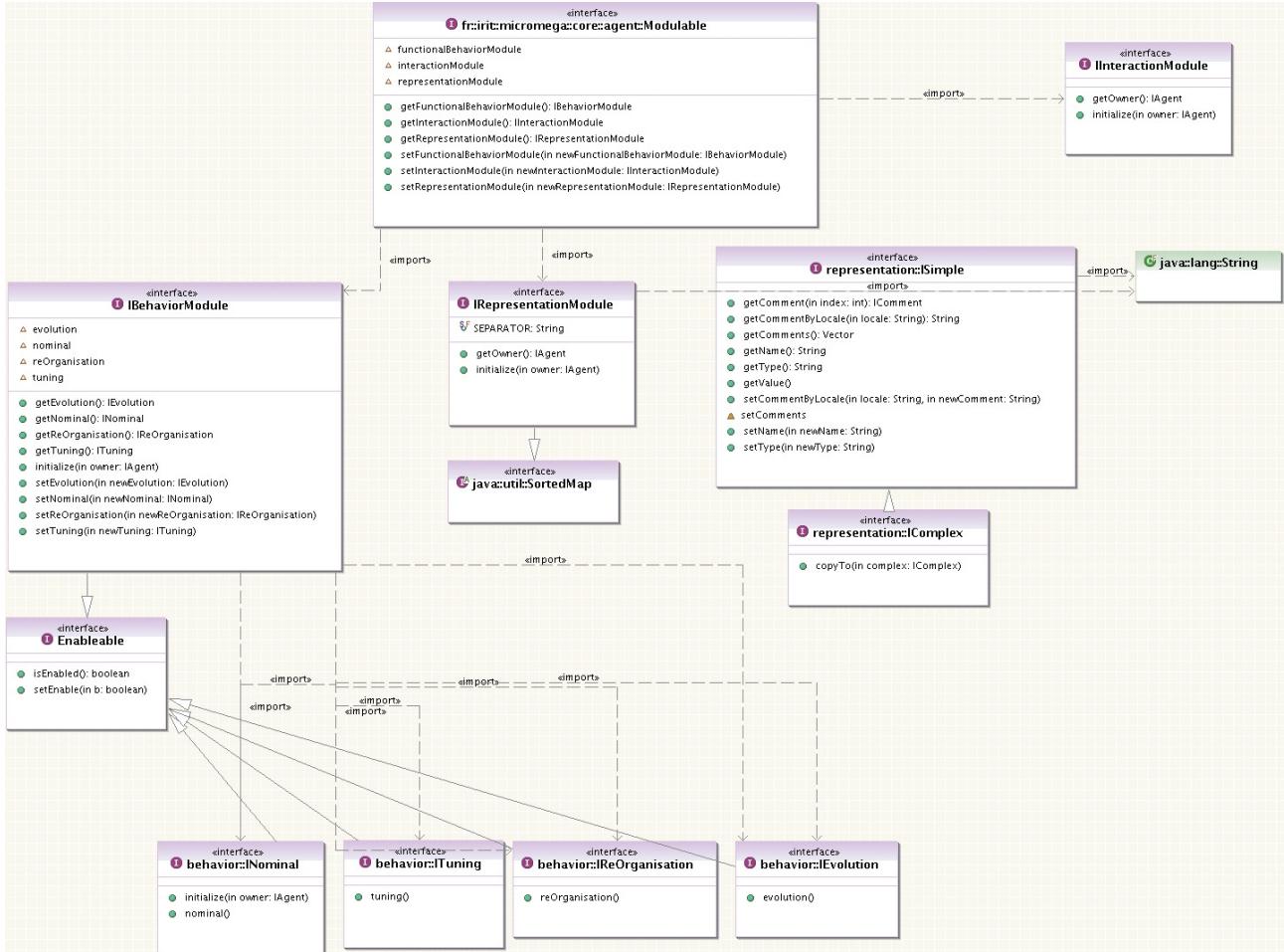


Diagramme 3: Définition des interfaces des modules

Les trois modules induits par *Modulable* sont (cf. Diagramme 3) :

- *IbehaviorModule* : définit le comportement d'un agent en décomposant ses activités en quatre sous-comportements : la description du comportement nominal (*INominal*), et la description du comportement adaptatif, i.e. l'ajustement du comportement (*ITuning*), la ré-organisation (*IReorganisation*) et l'évolution (*IEvolution*). Chacun de ces comportements peut être activé ou désactivé (*Enableable*).
- *IrepresentationModule* : définit le module de représentations comme une table d'association triée (*SortedMap*) contenant des associations de type *<String, ISimple>*. Un *ISimple* permet de stocker une valeur d'un type quelconque en lui associant un nom, une description du type et un commentaire. Un élément *IComplex* est un *ISimple* dont la valeur stockée est du type *SortedMap<String, ISimple>*, i.e. une table d'association d'éléments *ISimple*. De plus, une méthode de copie est définie sur *IComplex*. Ainsi, les représentations sont stockées sous la forme d'une arborescence dont les feuilles sont des *ISimple* et les noeuds des *IComplex*.
- *IinteractionModule* : permet de décrire le ou les modes d'interaction de l'agent.

Tous les modules doivent être initialisés, après leur création, à l'aide d'un appel de la méthode *void initialize(IAgent owner)* afin de déterminer l'agent sur lequel ils interviennent.

3.2.c Implémentation par défaut des modules de comportement

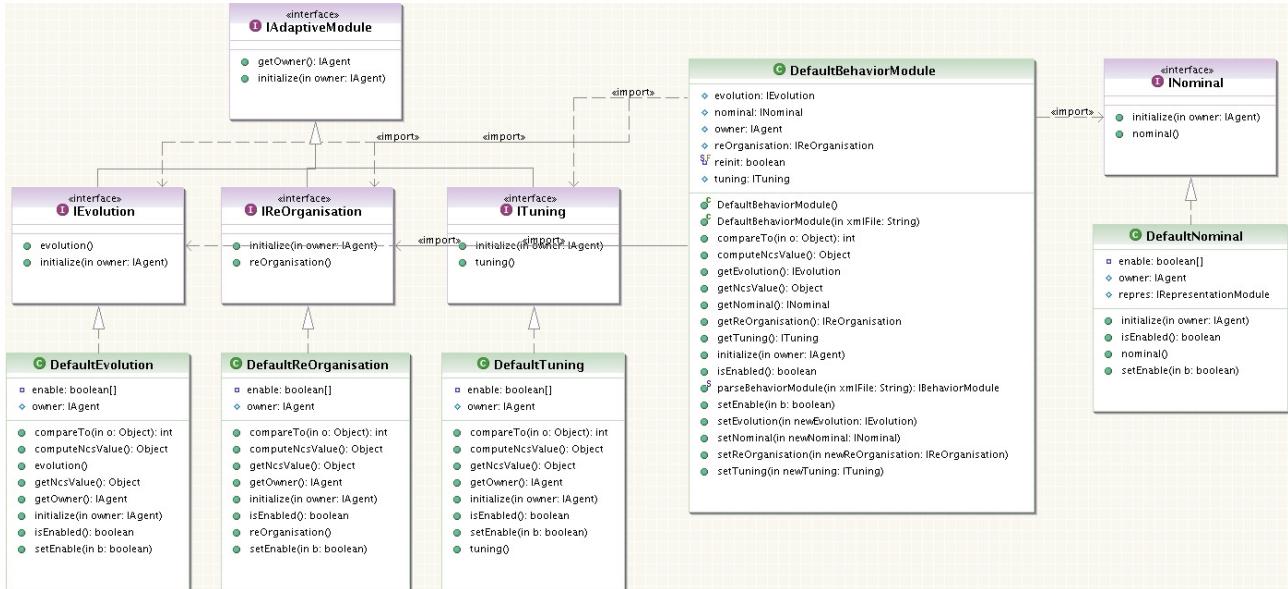


Diagramme 4: Implémentation des modules de comportement

Les classes `fr.irit.micromega.core.agent.behavior.Default*` fournissent des implémentations par défaut des différentes interfaces de comportement. De plus, la classe `DefaultBehaviorModule` possède une méthode statique (`public static IBehaviorModule parseBehaviorModule(String xmlFile);`) permettant la création d'un module de comportement à partir d'une description xml respectant la dtd suivante :

```

<!ELEMENT BehaviorModule (Nominal?, Tuning?, ReOrganisation?, Evolution?)>
<!ELEMENT Nominal (Comment*)>
<!ATTLIST Nominal
    moduleClassName CDATA #REQUIRED
    moduleXMLFile CDATA #IMPLIED
>
<!ELEMENT Tuning (Comment*)>
<!ATTLIST Tuning
    moduleClassName CDATA #REQUIRED
    moduleXMLFile CDATA #IMPLIED
>
<!ELEMENT ReOrganisation (Comment*)>
<!ATTLIST ReOrganisation
    moduleClassName CDATA #REQUIRED
    moduleXMLFile CDATA #IMPLIED
>
<!ELEMENT Evolution (Comment*)>
<!ATTLIST Evolution
    moduleClassName CDATA #REQUIRED
    moduleXMLFile CDATA #IMPLIED
>

```

`DefaultBehaviorModule` rajoute par défaut dans les représentations un complexe `FunctionalBehaviorModule` contenant lui-même un simple (String) `moduleType` ayant pour valeur le nom de la classe (type) utilisée comme `IBehaviorModule`. Les sous-modules créent de manière similaire des sous-complexes (`NominalModule`, `TuningModule`, `ReOrganisationModule`, `EvolutionModule`) dans `BehaviorModule` avec leur type respectif.

3.2.d Implémentation par défaut du module de représentation

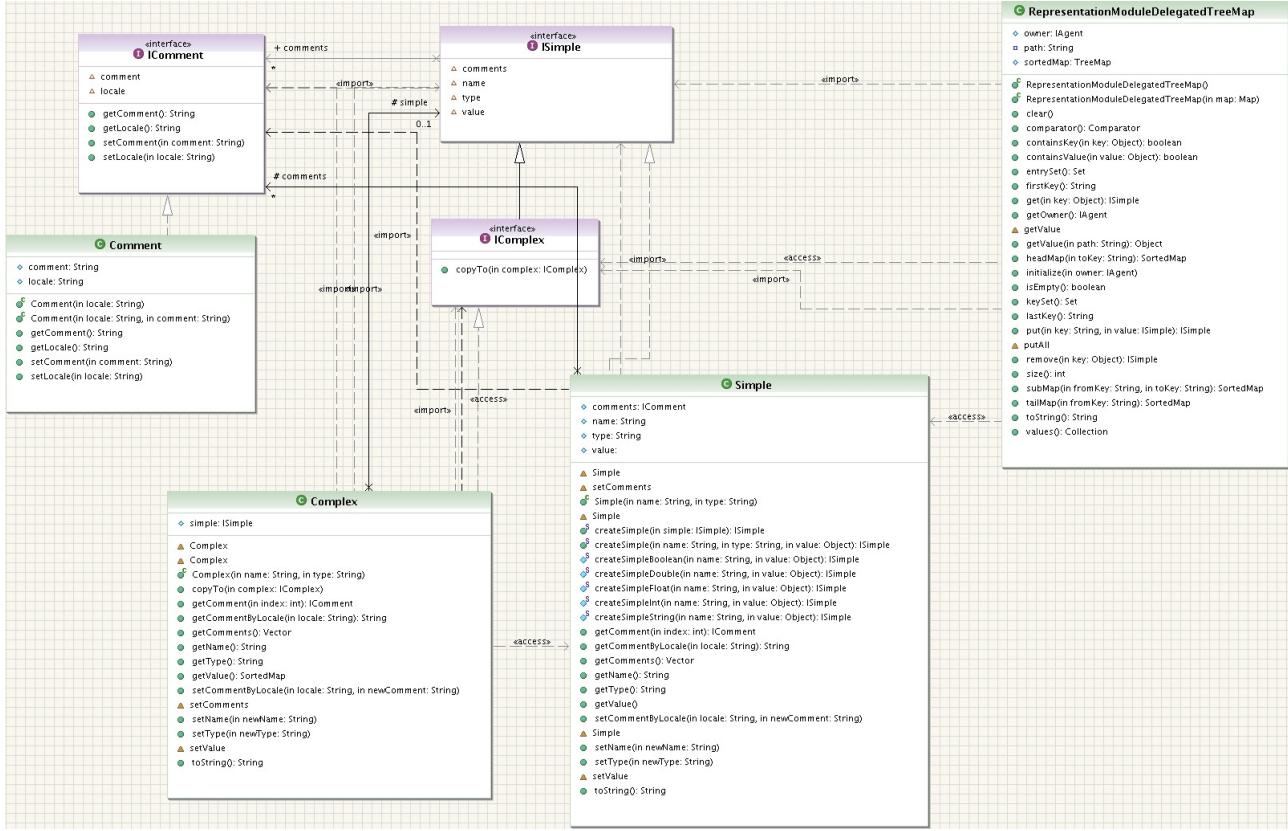


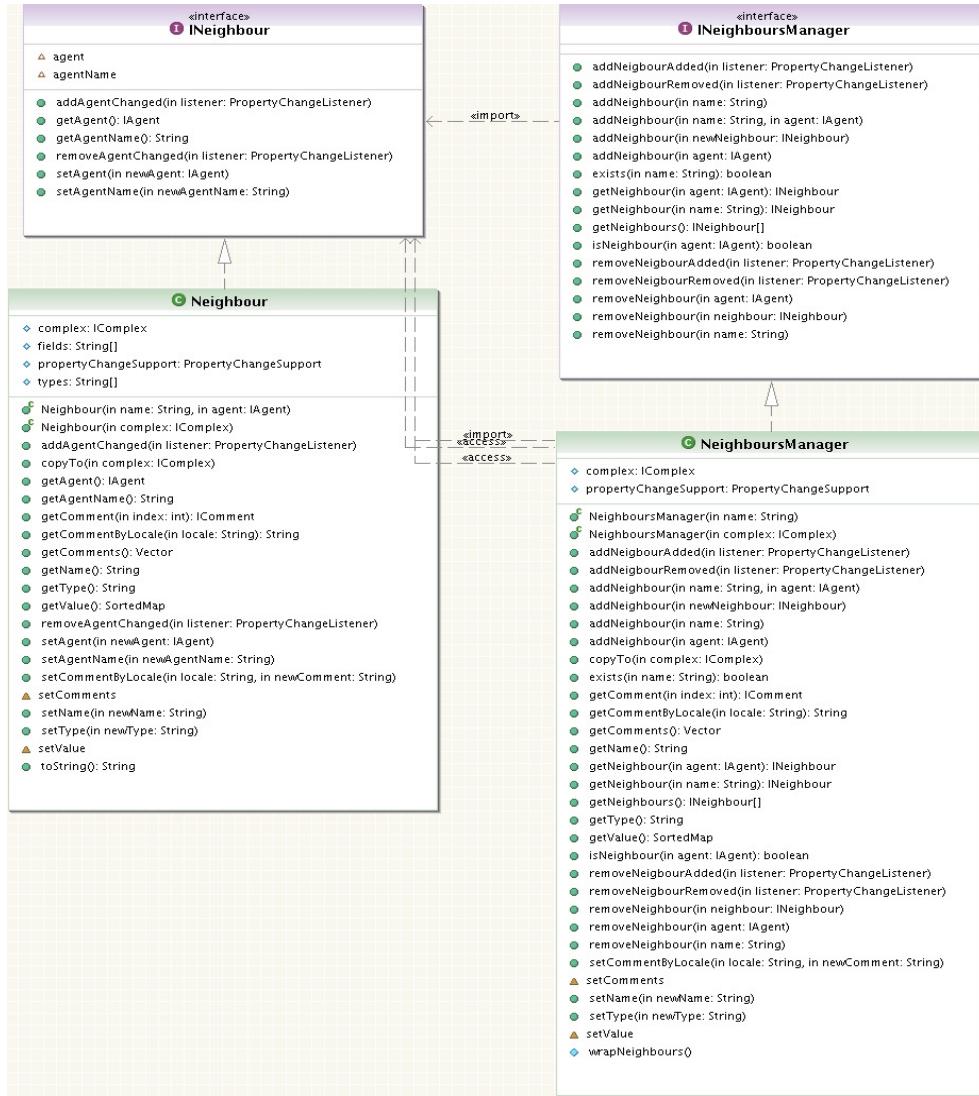
Diagramme 5: Implémentation de *IRepresentationModule*

`fr.irit.micromega.core.agent.modules.representation.RepresentationModuleDelegatedTreeMap` est une implémentation de *IRepresentationModule* basée sur l'utilisation d'une table d'association *TreeMap<String, ISimple>*. Une méthode `getValue(String path)` permet de récupérer la valeur d'un *ISimple* en indiquant son chemin d'accès (avec '`:`' comme caractère de séparation), un *IComplex* si le chemin en désigne un ou *null* si le chemin ne désigne pas un élément existant. *RepresentationModuleDelegatedTreeMap* contient par défaut un complexe *RepresentationModule* contenant lui-même un simple (String) *moduleType* ayant pour valeur le nom de la classe (type) utilisée comme module de représentation.

Simple est une implémentation de *ISimple* avec la définition de méthodes statiques de création d'éléments pour certains types de données tels que les entiers, les booléens, les chaînes de caractères, les doubles, les flottants et les références à des agents *IAgent*.

3.2.e Implémentation par défaut du module *IIInteraction*

En supplément de l'implémentation très simple de *IIInteraction* que fournit la classe `fr.irit.micromega.core.agent.modules.interaction.DefaultInteractionModule`, un petit ensemble d'outils est fourni pour la gestion de voisinage dans le sous-paquetage *neighbour* (cf. Diagramme 6). *DefaultInteractionModule* rajoute, par défaut, dans les représentations, un complexe *InteractionModule* contenant lui-même un simple (String) *moduleType* ayant pour valeur le nom de la classe (type) utilisée comme *IIInteractionModule*.

Diagramme 6: Le paquetage `neighbour`

Ce paquetage `neighbour` contient les interfaces définissant un voisin (`INeighbour`) et un module de gestion d'un ensemble de voisins (`INeighboursManager`).

3.3. Paquetage « simulation »

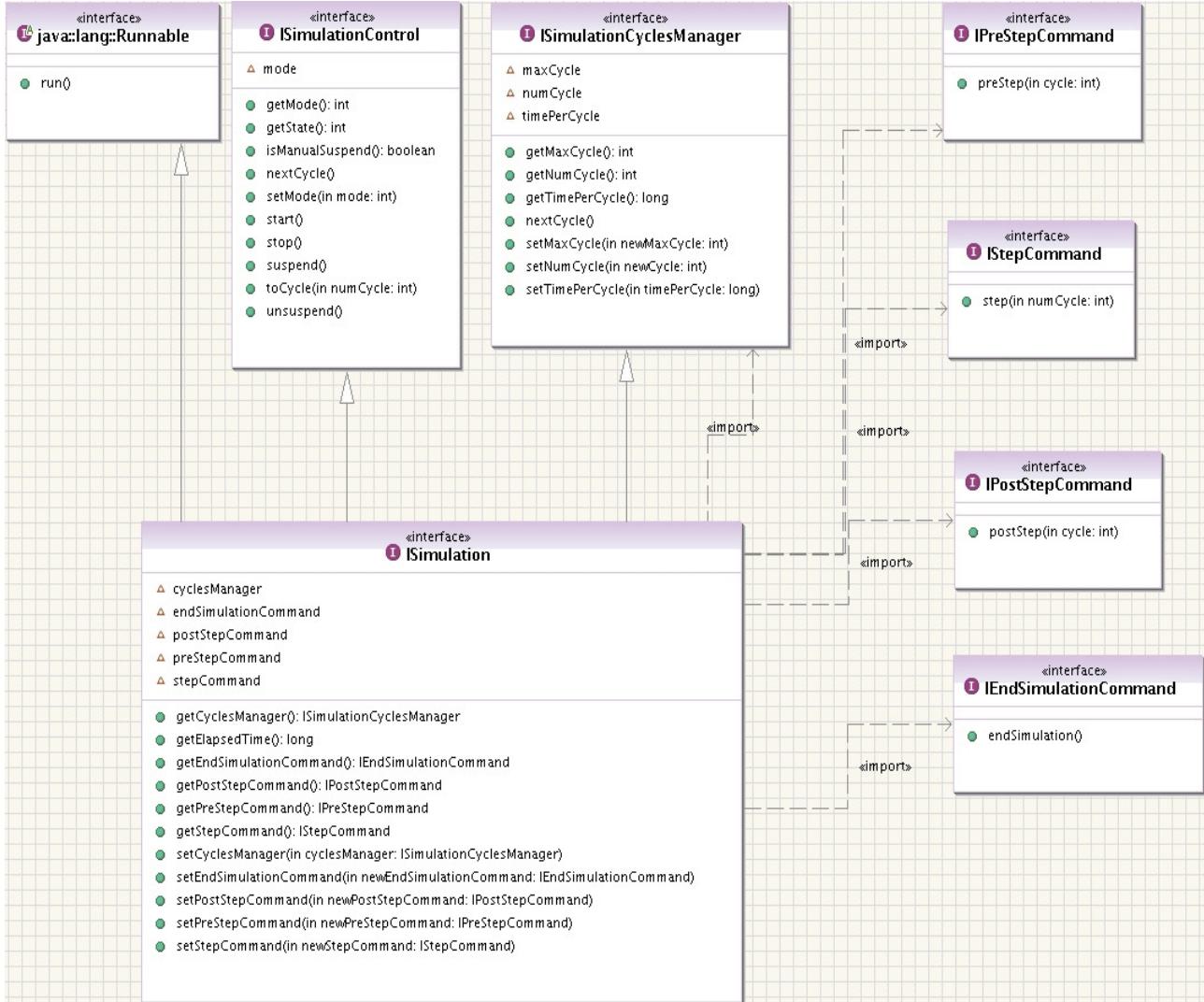


Diagramme 7: L'interface *ISimulation*

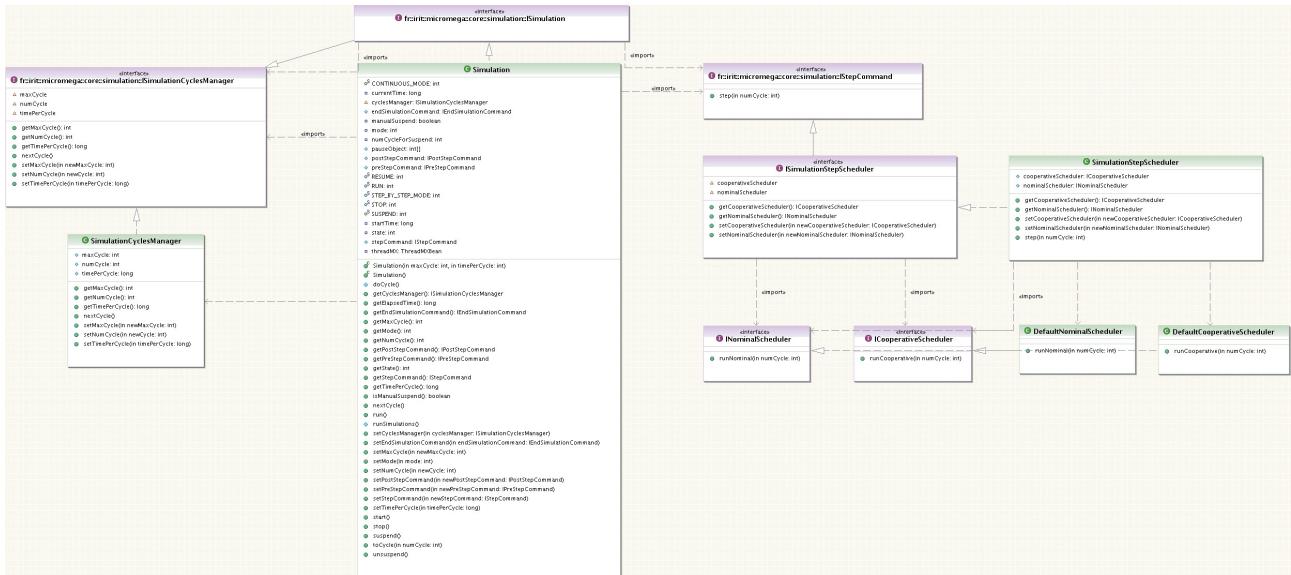
L'interface *ISimulation* décrite dans le Diagramme 7 montre qu'une simulation est composée d'un gestionnaire de contrôle (*ISimulationControl*) permettant de contrôler le déroulement de la simulation (start, stop, pause, etc.) et d'un gestionnaire de cycle (*ISimulationCyclesManager*). Afin de définir une simulation, il faut définir les commandes à exécuter pour chaque cycle (*IPreStepCommand*, *IStepCommand*, *IPostStepCommand*) et à la fin de la simulation (*IEndSimulationCommand*). La gestion du déroulement de la simulation est gérée par la méthode `run()` (héritée de *Runnable*).

Le Diagramme 8 décrit l'implémentation de *ISimulation* utilisée par défaut. L'exécution de chaque pas de la simulation consiste en l'exécution des activités nominales de tous les agents (dans l'ordre défini dans *DefaultNominalScheduler*) puis des activités coopératives/adaptatives de tous les agents (dans l'ordre défini dans *DefaultCooperativeScheduler*). Les fonctions d'ordonnancement fournies par défaut sont celles utilisées par le projet microMéga :

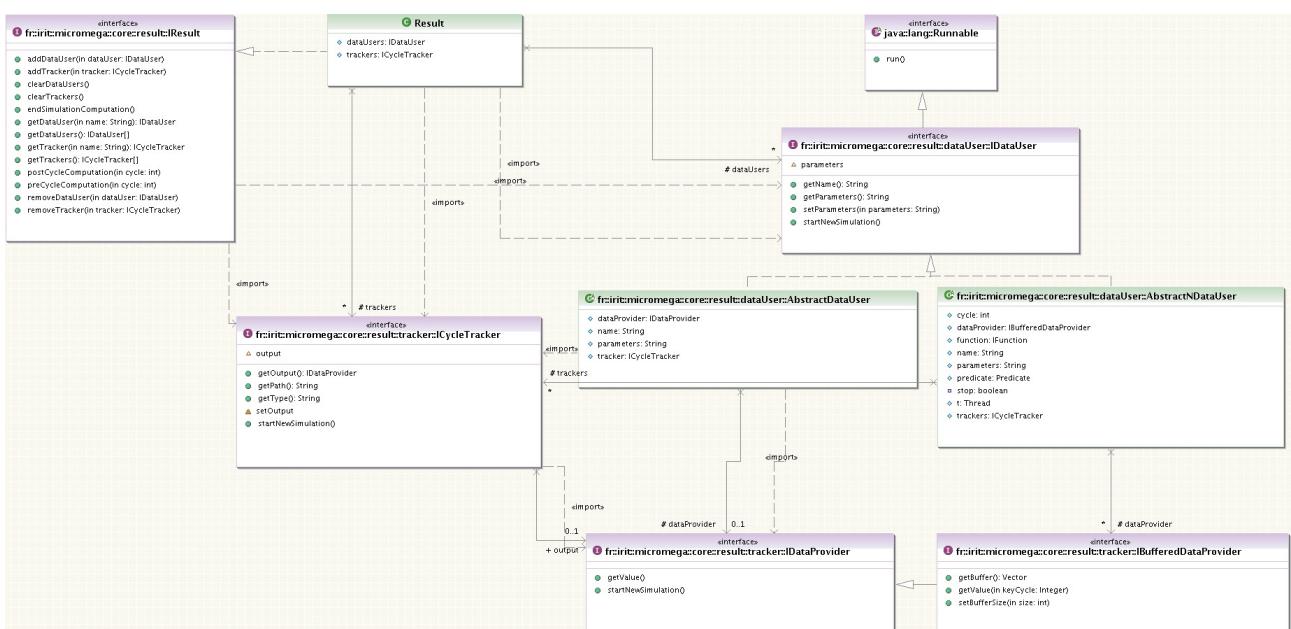
- *DefaultNominalScheduler* : exécute les comportements nominaux des agents de type « non element » et « non view » (toutes les réactions/transporteurs, donc), puis les comportements nominaux des agents éléments puis des agents vues en dernier.

- *DefaultCooperativeScheduler* : exécute les comportements coopératifs des agents vues puis des agents éléments puis des autres agents (réactions et transporteurs).

On peut noter que cette implémentation essaie de mesurer le temps écoulé depuis le début de la simulation en cours (*long getElapsedTIme()*). On notera aussi que les fonctions de contrôle de *Simulation* affectent le thread courant : ce type de simulation est censé être exécuté dans un thread créé spécifiquement ! (sinon l'application peut être bloquée).



3.4. Paquetage « result »



La gestion des résultats se fait à l'aide de deux outils : les « trackeurs » de données (*ICycleTracker*) et les utilisateurs de données (*IDataUser*). Un trackeur porte un nom et relève la valeur d'un *ISimple* présent dans les représentations (à l'aide d'un chemin) d'un agent donné. Un utilisateur de données récupère les données récoltées par un ou plusieurs trackeurs.

Comme l'illustre le Diagramme 9, *IResult* contient (en plus de la gestion des trackeurs et utilisateurs de données) les méthodes suivantes :

- *endSimulationComputation()* qui est appelée en fin de simulation et est utilisée pour réinitialiser les trackeurs de données.
- *preCycleComputation(int cycle)* et *postCycleComputation(int cycle)* qui sont appelées avant et après chaque pas simulation. Dans *Result*, *postCycleComputation* déclenche la mise à jour de chaque trackeur de données listé.

Les trackeurs de données (*ICycleTracker*) utilisent des *IDataProvider* pour fournir les données aux utilisateurs qui le demandent. Des fournisseurs bufferisés (*IbufferedDataProvider*) sont aussi définis.

Le Diagramme 10 illustre les implémentations de *ICycleTracker* et de *IDataProvider* réalisées (les classes concrètes pour tracker les types de données prédéfinis dans *Simple* sont implémentées dans le sous-paquetage *representation*).

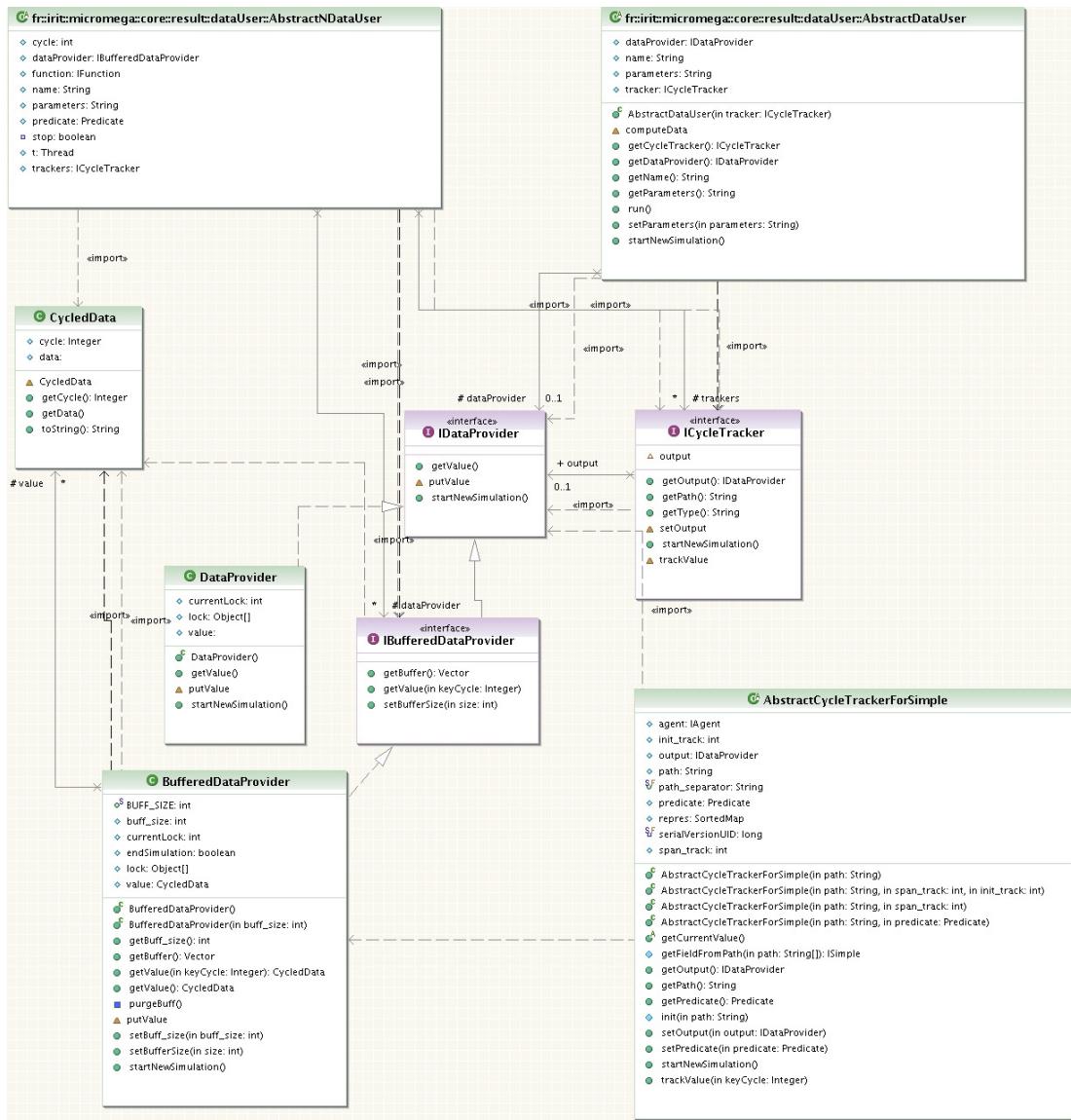


Diagramme 10: Implémentations des trackeurs de données

Les utilisateurs de données implémentés par défaut sont représentés sur le diagramme 11. Deux grandes catégories d'utilisateurs ont été considérées :

- les utilisateurs simples : ils traitent les données provenant d'un trackeur (*AbstractDataUser*).
- les utilisateurs N-aires : ils traitent des données provenant de plusieurs (N) trackeurs (*AbstractNDataUser*).

L'un des intérêts des utilisateurs N-aires est de pouvoir entrelacer deux données ; par exemple, si une valeur est stockée au cours du temps dans différentes variables (dans microMéga par exemple, les quantités des éléments sont stockées à deux endroits différents selon que le cycle soit pair ou impair).

Les implémentations concrètes présentées dans le Diagramme 11 sont :

- DataUserToScreen : pour afficher la donnée « trackée » sur la sortie standard.
- DataUserToFile : pour enregistrer la donnée trackée dans un fichier.
- DataUserCycledDataToXYSeries : pour générer des séries exploitables pour l'affichage sous forme de courbes (jFreeChart).

- NdataUserCycledDataToXYSeriesWithInt : pour générer des séries exploitables pour l'affichage sous forme de courbes (jFreeChart) et uniquement sur des données entières.
- NDataUserToFile: pour enregistrer les données trackées dans un fichier.

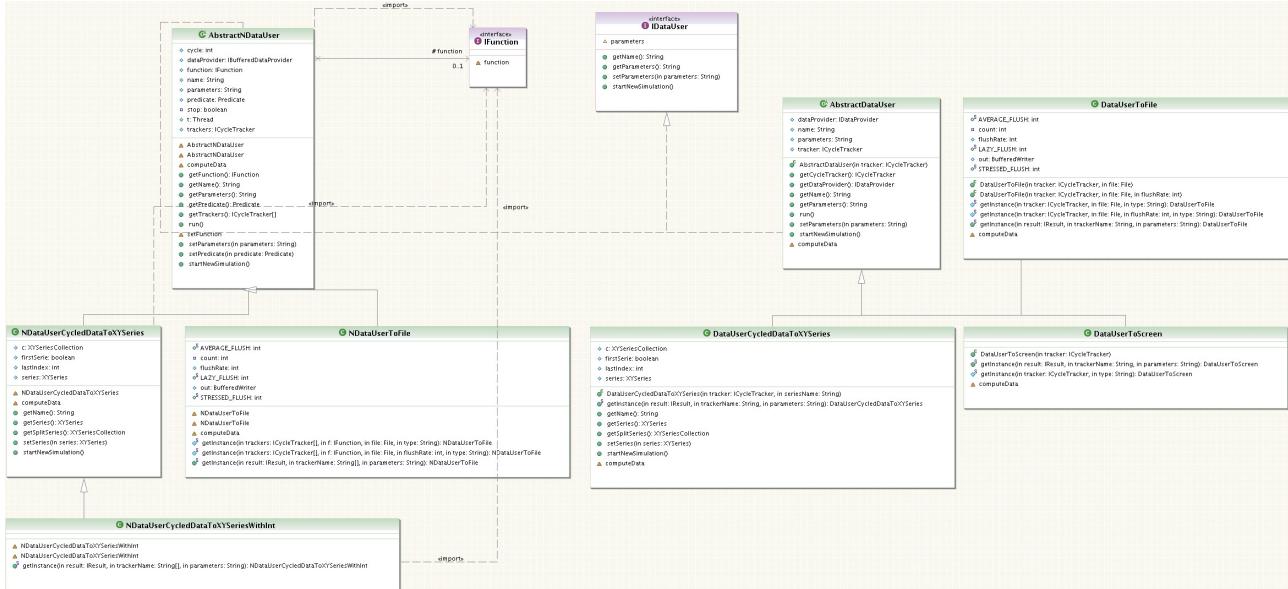


Diagramme 11: Les utilisateurs de données implémentés

Ces classes concrètes implémentent les utilisateurs de manière parallèle (un thread pour chacun). Cela permet de ne pas arrêter le déroulement de la simulation en cas de traitement bloquant ou très long (accès disque, affichage, enregistrement, envoi sur réseaux) mais ne garantit pas que toutes les valeurs seront traitées au cours du temps (en cas de retard, l'utilisateur sera désynchronisé et « ratera » des données).

3.5. Paquetage « util »

Ce paquetage contient différents utilitaires tels que :

- les parseurs xml : dans le sous-paquetage *xml*
- les parseurs de données biologiques : dans le sous-paquetage *importation*
- le système de trace : dans le sous-paquetage *log*

3.5.a Paquetage « xml »

Le sous-paquetage *dtd* contient les dtd utilisées par le projet (pour décrire les agents, les simulations, les résultats etc.)

Les sous paquetage *loader* et *saver* contiennent les classes de sauvegarde et chargement sous forme de fichiers .xml et .mws (mws est un format compressé et mieux organisé que son équivalent xml).

3.5.b Paquetage « importation »

La classe *Importation* permet le parsing de fichiers contenant des réseaux sous forme de matrices d'interaction. La classe *ComplexReaction* parse les matrices d'interaction réactionnelles.

3.5.c Paquetage « log »

Ce paquetage contient les outils permettant de tracer l'exécution du système.

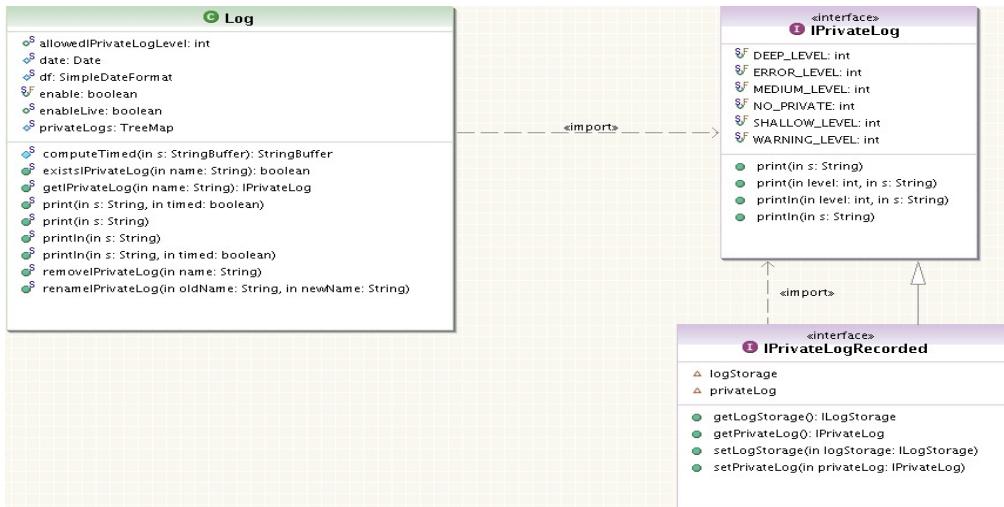


Diagramme 12: Le paquetage Log

Le Diagramme 12 donne une vision très simplifiée et utilitariste de ces outils. La classe principale est la classe *Log* qui ne contient que des champs et méthodes statiques.

La fonctions de base de *Log* sont d'afficher sur la sortie standard à l'aide des méthodes *print**(*)).

Ce paquetage fournit aussi un service de log « privatif » sur le principe de canaux nommés du type *IPrivateLog*. Le nom de ces canaux est un String, et par défaut, le contenu de ces canaux est aussi affiché sur la sortie standard (cf. le champs *allowedIPrivateLogLevel* pour les restrictions).

Les champs de contrôle accessibles de *Log* sont :

- *enable* : permet d'activer ou de désactiver l'outil de manière statique (AVANT compilation).
- *EnableLive* : si *Log.enable* est vrai, alors on peut activer/désactiver l'outil de manière dynamique (lors de l'exécution) à l'aide de ce champ.
- *AllowedIPrivateLogLevel* : niveau à partir duquel les logs privés sont aussi affichés sur le log principal.

Les canaux privés *IPrivateLog* permettent un marquage des messages par niveaux. Les niveaux prédéfinis sont (par ordre hiérarchique décroissant de restriction) :

1. NO_PRIVATE : aucun affichage (0)
2. ERROR_LEVEL : trace de niveau erreur (<32)
3. WARNING_LEVEL : trace de niveau warning (<64)
4. SHALLOW_LEVEL : trace superficielle (<128)
5. MEDIUM_LEVEL : trace intermédiaire (<256)
6. DEEP_LEVEL : trace détaillée en profondeur (<512)

Exemples d'utilisation :

```
if (Log.enable) Log.println("IAgent.parseAgent succed", true);
```

Affichage du message « IAgent.parseAgent succed » sur la sortie standard avec l'heure de diffusion.

```
if (Log.enable)
```

```
Log.getIPrivateLog("test").println(IPrivateLog.MEDIUM_LEVEL, "message" + 5);
```

Affichage du message « message5 » sur le canal privé nommé « test » (s'il n'existe pas, il est créé), avec un niveau intermédiaire.

Attention : Toute utilisation de *Log* se fait en commençant par « if (Log.enabled) Log. »+ accès statique à une fonctionnalité (soit directement un affichage, soit l'accès à un canal). Ne pas utiliser un *IPrivateLog* sans passer par un accès statique *Log.getIPrivateLog(String)* et donc en faisant le test « if (Log.enabled) ».

3.6. Le paquetage « solver »

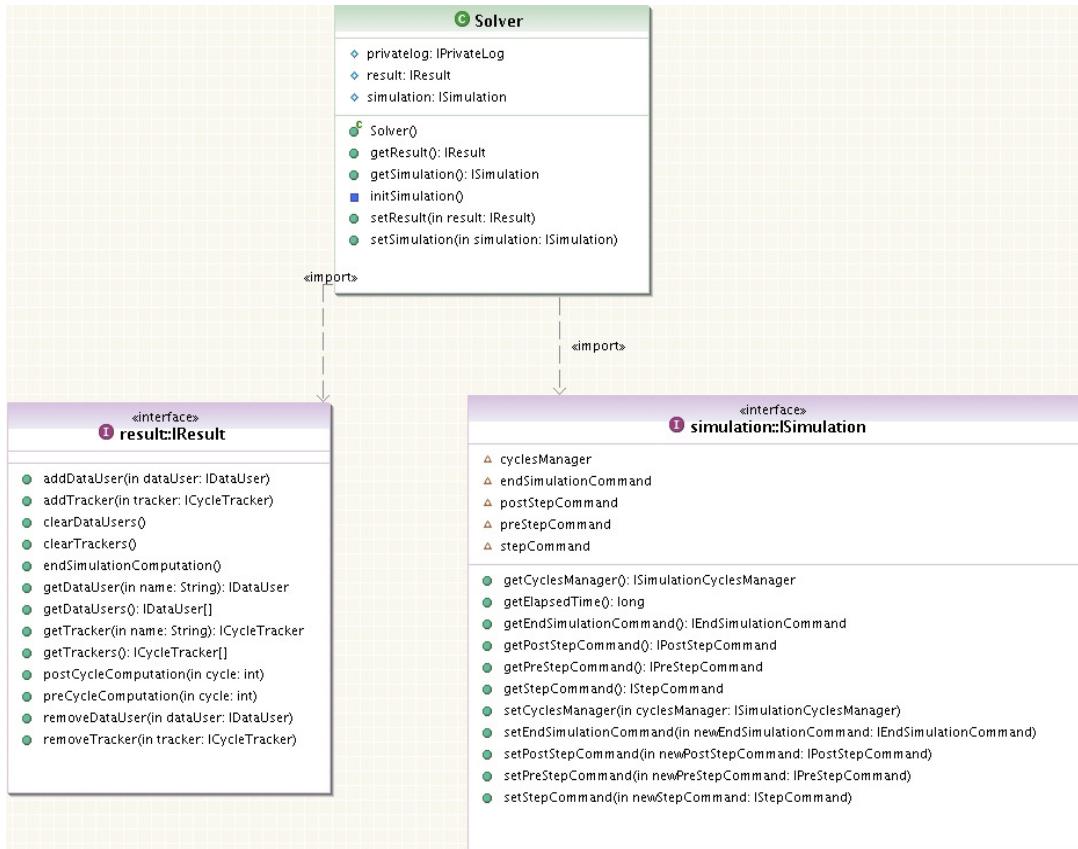


Diagramme 13: La classe Solver

La classe *Solver* est le point d'entrée du moteur de simulation de microMega et gère donc la simulation ainsi qu'un module responsable de collecter les résultats.

4. Le paquetage « functional »

4.1. Aperçu

Ce paquetage contient le code relatif aux agents fonctionnels utilisés dans le projet. Ces agents sont construits par ajout des fonctionnalités spécifiques liées à leur type par « dessus » les fonctionnalités fournies par les modules par défaut de « core ». Attendu que nombre de ces fonctionnalités sont utilisées par plusieurs (ou tous les) types d'agent, le paquetage *utils* contient des couches fonctionnelles intermédiaires.

4.2. Les différents types d'agent

Les cinq types d'agents définis dans le projet sont les éléments (*elements*), les transporteurs (*carriers*), les réactions de synthèse (*synthesizers*), les réactions de catalyse (*catalysts*) et les gènes (*dnas*). Pour chaque type, l'agent est défini à l'aide d'un fichier xml le décrivant en terme de contenu des représentations (souvent optionnel en fait) et de type de modules. Dans l'implémentation courante, chaque agent est donc défini par :

- un fichier xml ;
- une classe *BehaviorModule* ;
- une classe *Nominal*, implémentant *INominal*, et décrivant l'activité nominale ;
- une classe *Tuning*, implémentant *ITuning*, qui décrit l'activité coopérative de « tuning » ;
- une classe *<type>ReOrganisation*, implémentant *IReOrganisation*, et décrivant l'activité coopérative de réorganisation ;
- une classe *<type>Evolution*, implémentant *IEvolution*, qui décrit l'activité coopérative d'évolution.

On peut classer les cinq types d'agent en deux grandes catégories : les éléments et les réactions (on considère les gènes et les différents types et sous-types de transporteurs comme des réactions du point de vue fonctionnel).

Remarque : L'essentiel du « code » décrivant les agents de chaque catégorie est générique (factorisé dans le sous-paquetage *utils*).

4.2.a Les agents réactionnels

Pour les agents réactionnels, la partie spécifique de l'activité nominale consiste à implémenter *IReactionManager* pour donner une description de la réaction (cf. *fr.irit.micromega.functional.util.behavior.nominal.** pour les détails) :

PassiveCarrierAgent : « input1 » => « sortie1 » + « energy »

CatalystAgent : « entree1 » => « sortie1 » + « sortie2 »

SynthesizerAgent : « entree1 » + « entree2 » => « sortie1 »

DnaAgent : « input1 » + « input2 » + « input3 » + « input4 » => « output1 » + « output1 »

4.2.b Les agents éléments

L'activité nominale des éléments est codée dans la classe *ElementNominal* : au cycle 0, la quantité courante est initialisée à la quantité initiale; aux cycles > 0, la quantité courante est égale à la quantité précédente plus la somme des productions/consommations (ajouts/retraits) effectuées par

les agents réactionnels sur l'élément : $Q_t = Q_{t-1} + \sum_{n \in Produit \cup Consom} \Delta Q_n$

4.3. Paquetage « representation »

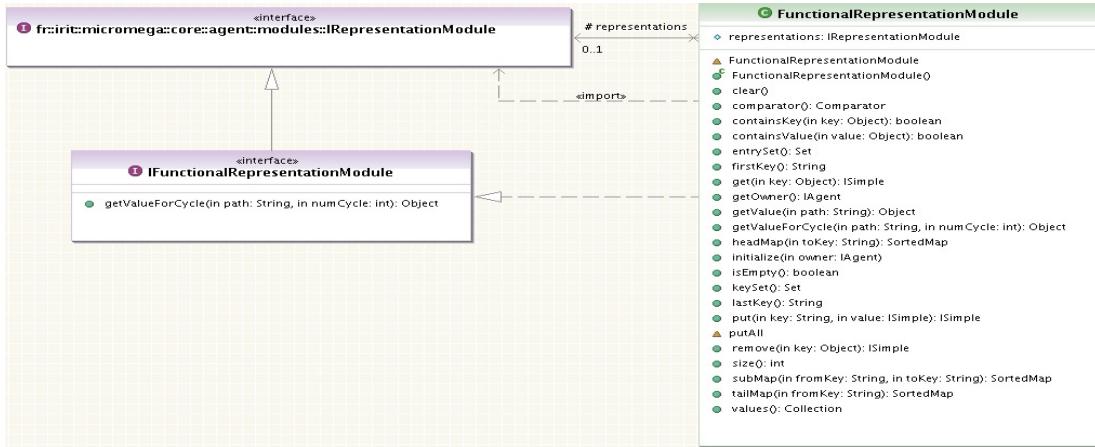


Diagramme 14: *IFunctionalRepresentation*

`IFunctionalRepresentation` est le module de représentation des agents fonctionnels. Ce module rajoute la fonctionnalité `getValueForCycle(String, int)` qui permet de récupérer la valeur associée à un chemin à partir des complexes « Cycle0 » ou « Cycle1 » selon que le second paramètre (cycle) est pair ou impair (utile essentiellement pour les quantités des éléments).

4.4. Paquetage « interaction »

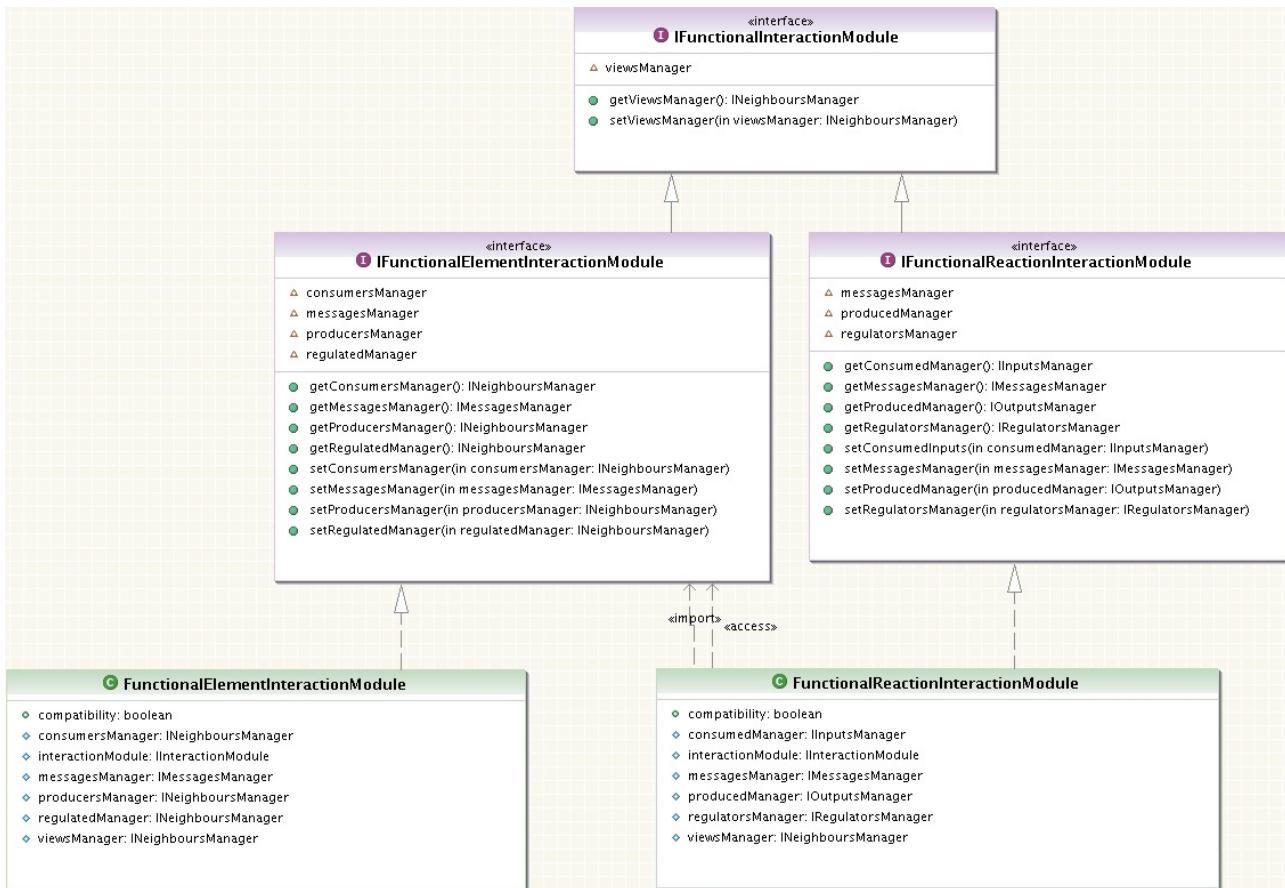


Diagramme 15: Les modules d'interaction des agents fonctionnels

IFunctionalInteractionModule est le module de gestion des interactions dédié aux agents fonctionnels. Comme le montre le diagramme 15, les interactions sont de deux types différents :

1. Les interactions directes qui sont gérées par des *INeighboursManager* qui gèrent les entrées et sorties liées aux activités de production, de consommation et de régulation ;
2. Les interactions par le biais de messages à l'aide d'un *IMessagesManager* dédié aux échanges d'information liés à la coopération (notamment lors des phases de tuning).

Ce module d'interaction est décliné en deux spécialisations correspondant aux deux catégories d'agents fonctionnels :

1. Les éléments : ces agents ont des voisins qui les consomment, qui les produisent ou alors ils régulent certaines réactions ;
2. Les réactions : ces agents consomment leurs entrées (*IInputManager*), produisent sur leurs sorties (*IOutputsManager*) et sont régulés par leurs régulateurs (*IRegulatorsManager*).

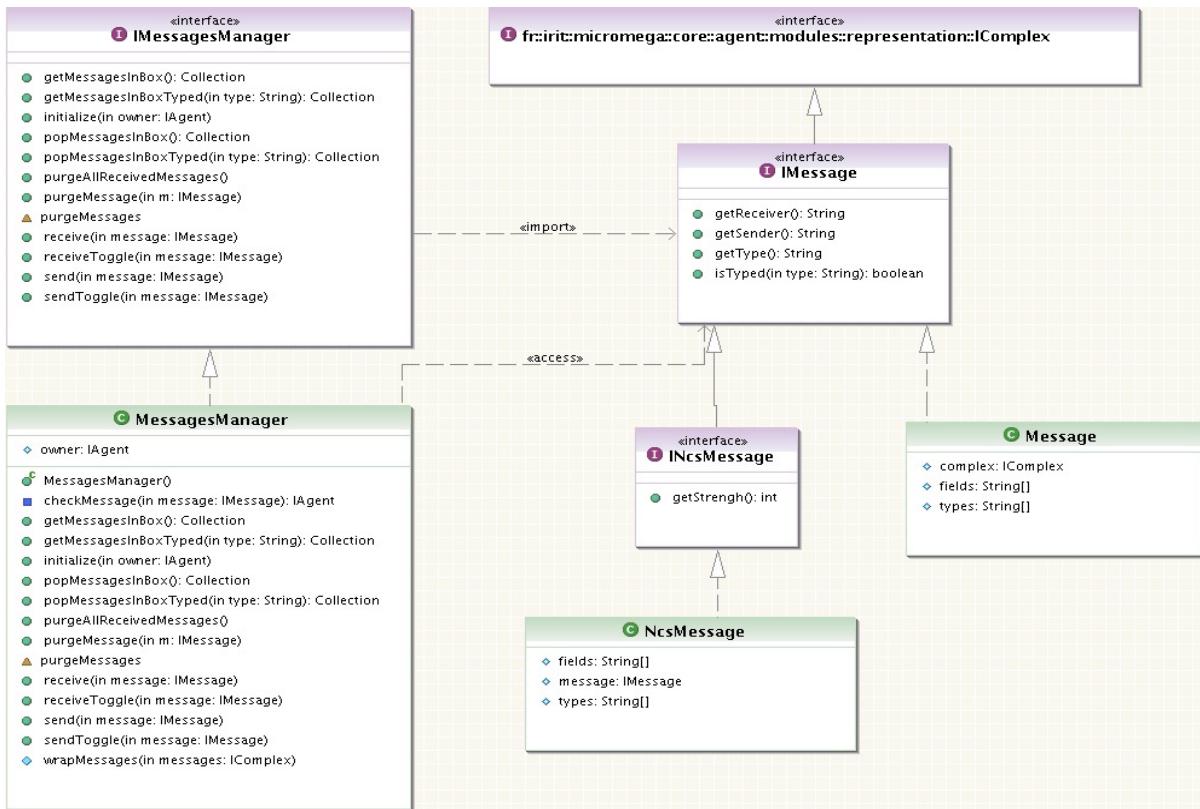


Diagramme 16: *IMessageManager et les types de message (Imessage)*

Le diagramme 16 décrit l'implémentation du module de gestion des messages. Ce module utilise un système de boîte aux lettres privé (chaque agent possède sa propre boîte à lettres) et définit une interface **IMessage** pour les messages qui seront échangés.

Comme les cycles de vie des agents sont fonction de deux états (l'état courant et l'état précédent) chaque agent dispose de deux boîtes aux lettres. De plus, les méthodes `send()` et `receive()` qui envoient, au temps t, un message qui sera lu au temps t+1, sont déclinées en `sendToggle()` et `receiveToggle()` afin de pouvoir envoyer au temps t un message qui sera lu au même temps t (utile dans certains cas durant l'adaptation).

Les états des boîtes aux lettres (les messages qu'elles contiennent) sont automatiquement représentés dans le module de représentation des agents pour les implémentations fournies (*MessageManager*, *Message*, *NcsMessage*).

4.5. Paquetage « behavior » (nominal)

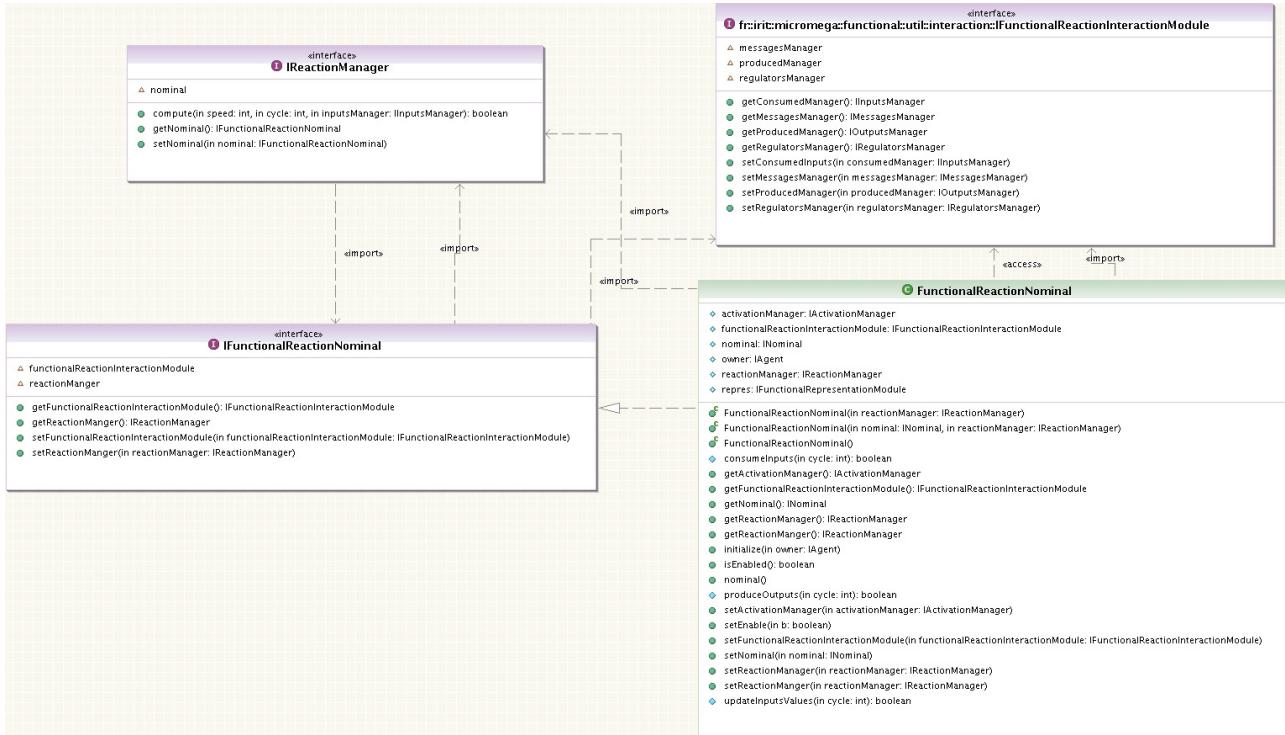


Diagramme 17: *IFunctionalReactionNominal* pour le comportement des réactions

L'implémentation du module de comportement nominal pour les agents fonctionnels modélisant une réaction dépend de l'utilisation par l'agent du module d'interaction adéquat (*IFunctionalReactionModule*) et délègue la partie traitement à un *IReactionManager* correspondant à la réaction (cf. définition d'un agent réactionnel). Cette architecture permet de factoriser au maximum le code réutilisable des spécificités de chaque réaction. Par exemple, la gestion de la vitesse de réaction est gérée pour tous les agents réactionnels par un module (*ActivationManager*, cf. sous-section suivante) gérant des contextes auxquels sont associés une vitesse ainsi que l'activation du meilleur contexte en fonction de la situation courante.

4.5.a Paquetage « util.activation »

Le rôle du *IActivationManager* est de gérer la définition d'un ensemble de *IContext* qui associent à une plage de situations donnée une vitesse spécifique. Le *IActivationManager* doit ainsi fournir la vitesse associée au « meilleur » contexte stocké en fonction de la situation courante de l'agent, i.e. en fonction des valeurs des régulateurs de l'agent (accessibles via un *IContextRegulatorsManager*). Le diagramme 19 montre comment est défini l'*ActivationManager* utilisé par les agents réactionnels de microMéga.

Dans le diagramme 18 est précisée la composition d'un contexte pour microMéga avec, notamment, un module de gestion des plages de valeurs associées aux régulateurs et une mesure de distance inter-contexte qui est ici une simple distance euclidienne. Les *Context* sont aussi des *IComplex* afin de pouvoir être stockés facilement dans les modules de représentation des agents.

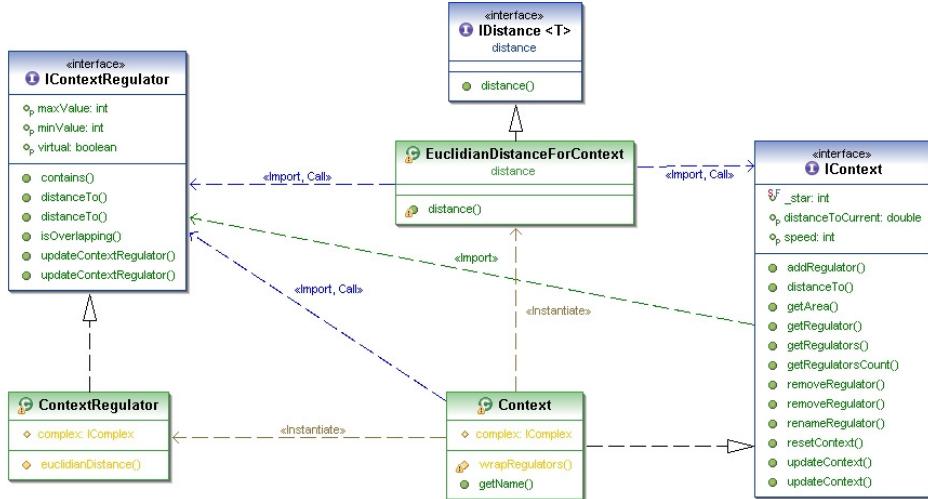


Diagramme 18: IContext et son implémentation

4.6. Paquetage « behavior » (tuning)

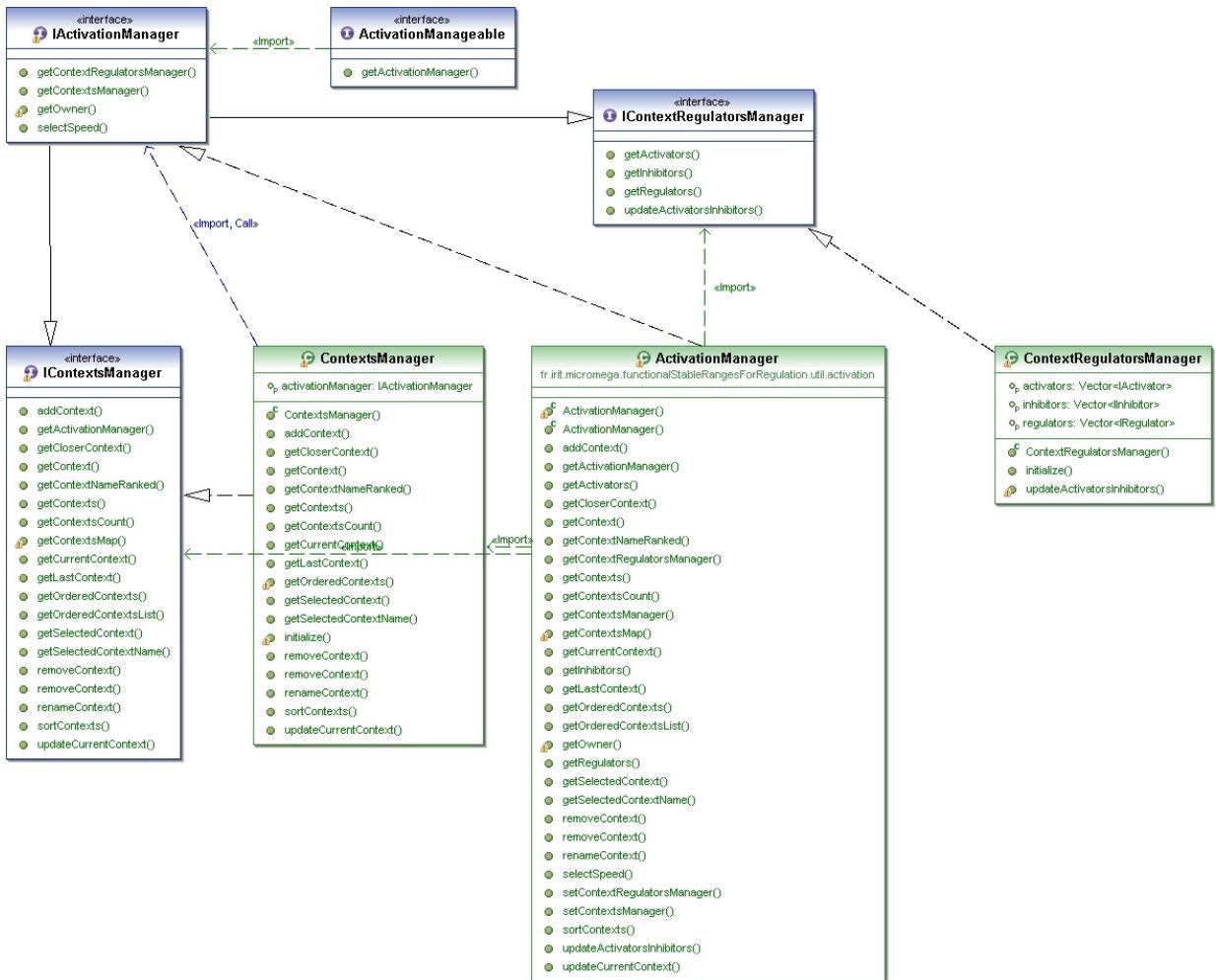


Diagramme 19: *IActivationManager et la gestion des contextes*

Le module de comportement d'ajustement pour les agents réactionnels est basé sur la définition des INCS correspondantes aux problèmes communiqués par les entrées ou sorties de la réaction et sur les actions de modification soit de la stœchiométrie, soit de la vitesse fournie par le module d'activation de contexte (voir sous-section suivante).

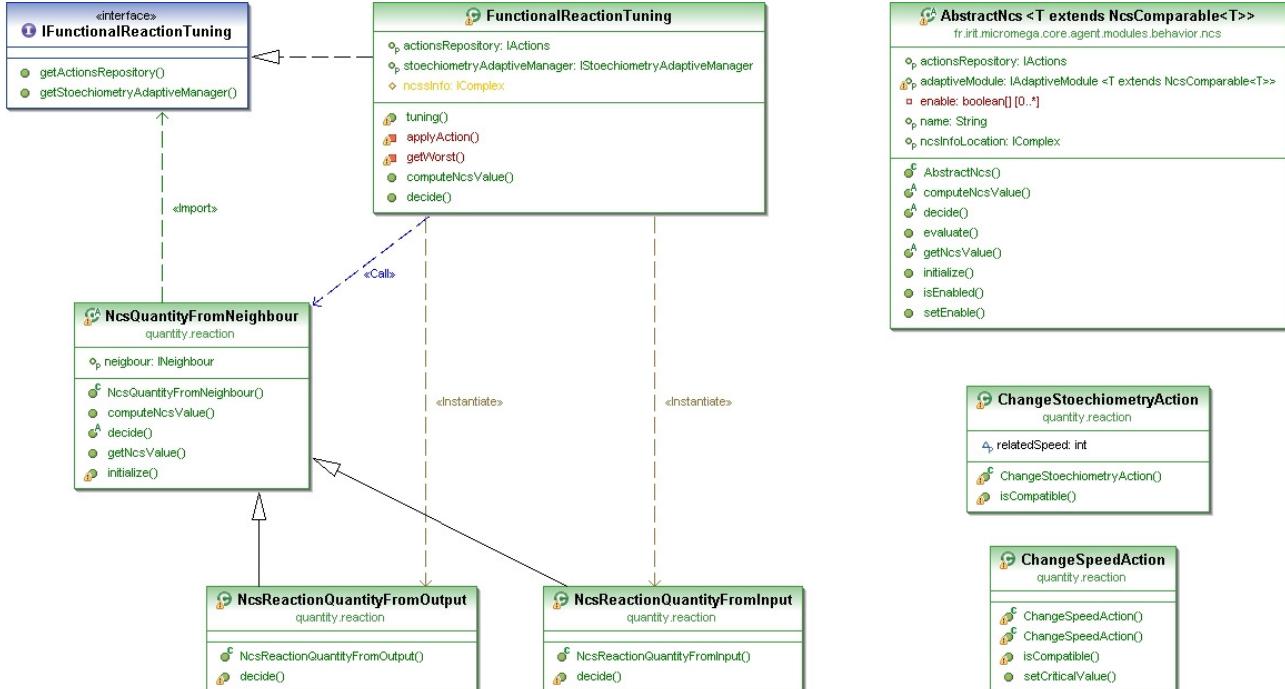


Diagramme 20: Module de Tuning pour les réactions

Pour les agents fonctionnels représentant des éléments (voir diagramme 21), les situations non coopératives sont soit des messages envoyés par les réactions que régule l'élément, soit des messages envoyés par des agents d'observation, soit la détection d'une quantité négative d'éléments. Les actions correctrices associées sont les modification de la quantité courante associée à l'élément ou bien la modification de la quantité initiale.

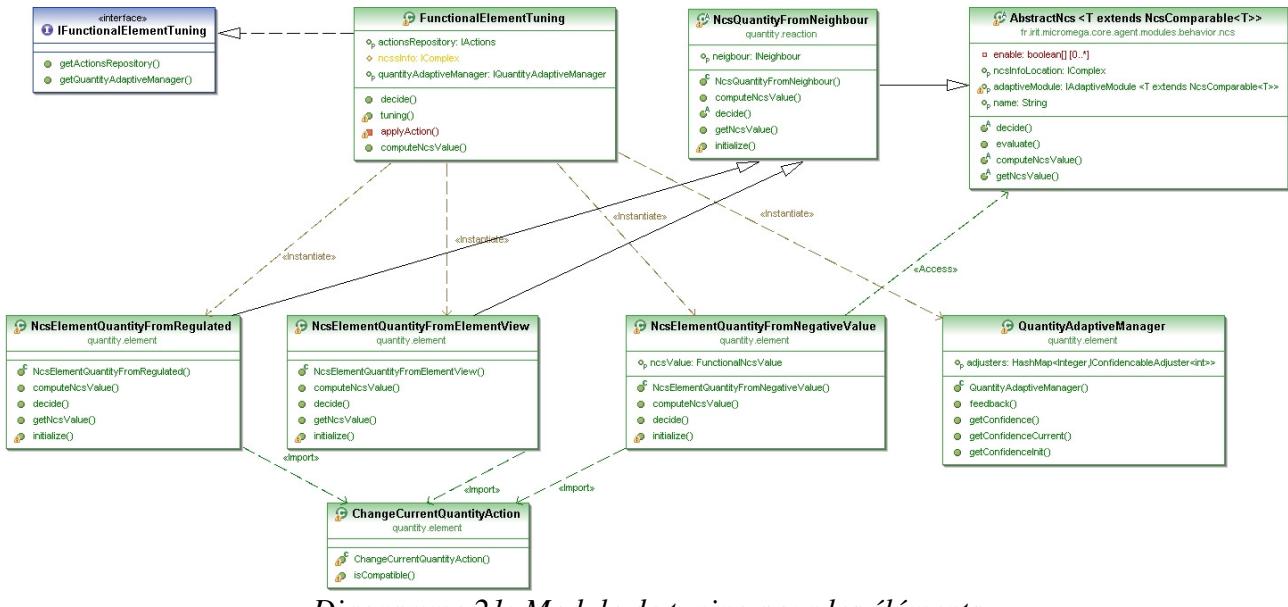


Diagramme 21: Module de tuning pour les éléments

4.6.a Paquetage « util.activation.adaptive »

Lors de l'analyse des situations non coopératives de la phase d'ajustement des agents réactionnels, le module de *IFunctionalReactionTuning* peut choisir de demander une modification de la vitesse fournie par le *IActivationManager*. Afin de rajouter la possibilité de modifier les contextes à la demande, nous avons défini un type dérivé de gestionnaire d'activation décrit dans le diagramme 22 (le *IAdaptiveActivationManager*) qui permet notamment d'adapter la vitesse fournie en fonction d'une requête donnée.

Le traitement proprement dit de cet ajustement est délégué à un *ISpeedAdaptator* et les contextes initiaux définis dans le *ActivationManager* sont encapsulés dans des *IAdaptiveContext* qui utilisent des modules d'ajustement de valeur (voir détails dans la sous-section qui suit).

Afin de gérer ces contextes modifiés, un *IAdaptiveContextsManager* basé sur l'ancien *IContextsManager* est utilisé. Ce nouveau gestionnaire de contextes gère aussi l'accès aux informations relatives à l'état précédent des contextes. En effet, à chaque pas *t* de décision des agents, ces derniers corrigeant les problèmes détectés en se basant sur l'état précédent ayant mené à ces problèmes. Il est donc nécessaire de garder trace du contexte qui avait été alors utilisé (*lastSelectedContext*), dans quelle situation (*lastCurrentContext*) et en se basant sur quel ordonnancement des contextes (*sortLastContexts()*).

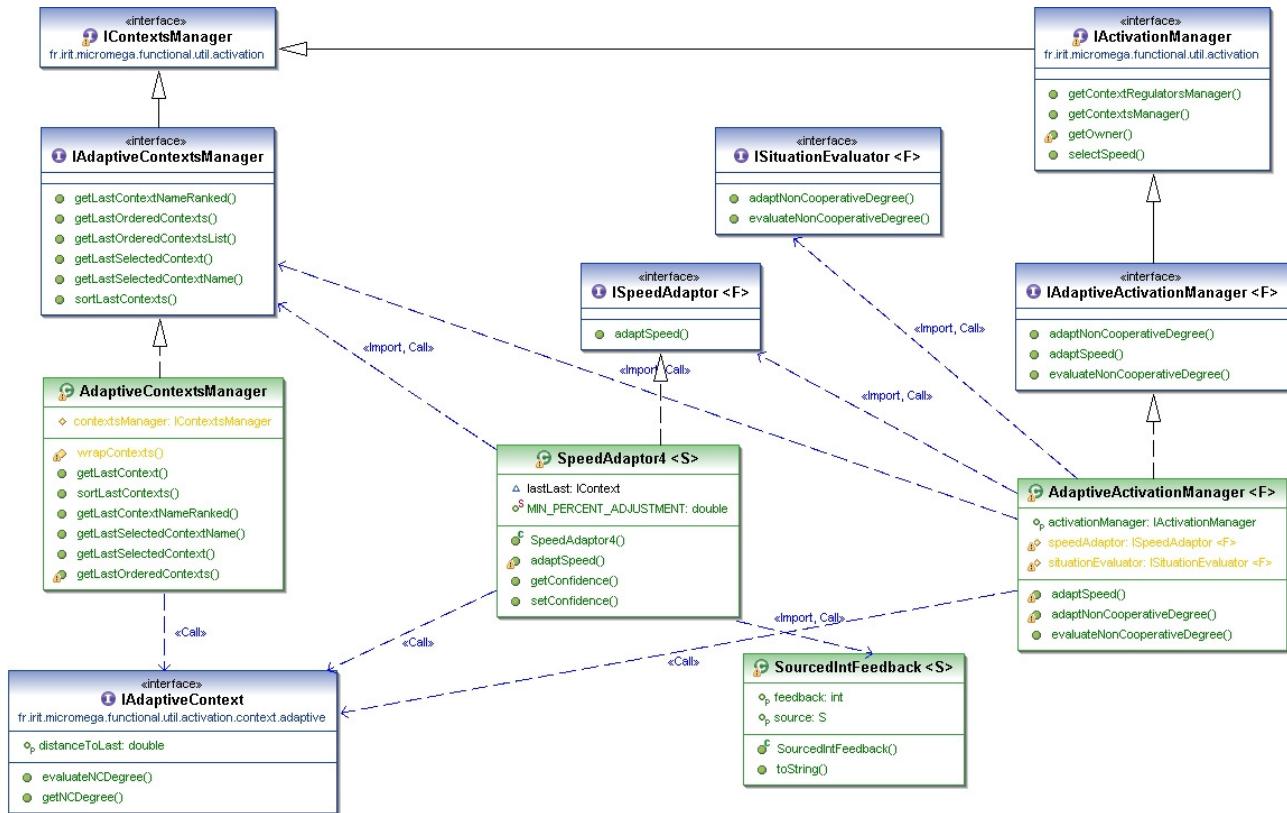


Diagramme 22: Ajout de fonctionnalités d'adaptation à l'IActivationManager

4.6.b Paquetage « util.activation.context.adaptive »

Afin de pouvoir adapter efficacement les contextes, ces derniers sont encapsulés automatiquement dans des *IAdaptiveContext* comme l'illustre le diagramme 23. Des ajusteurs de paramètres sont rajoutés sur la valeur de vitesse ainsi que sur les bornes de chaque plage de régulateur.

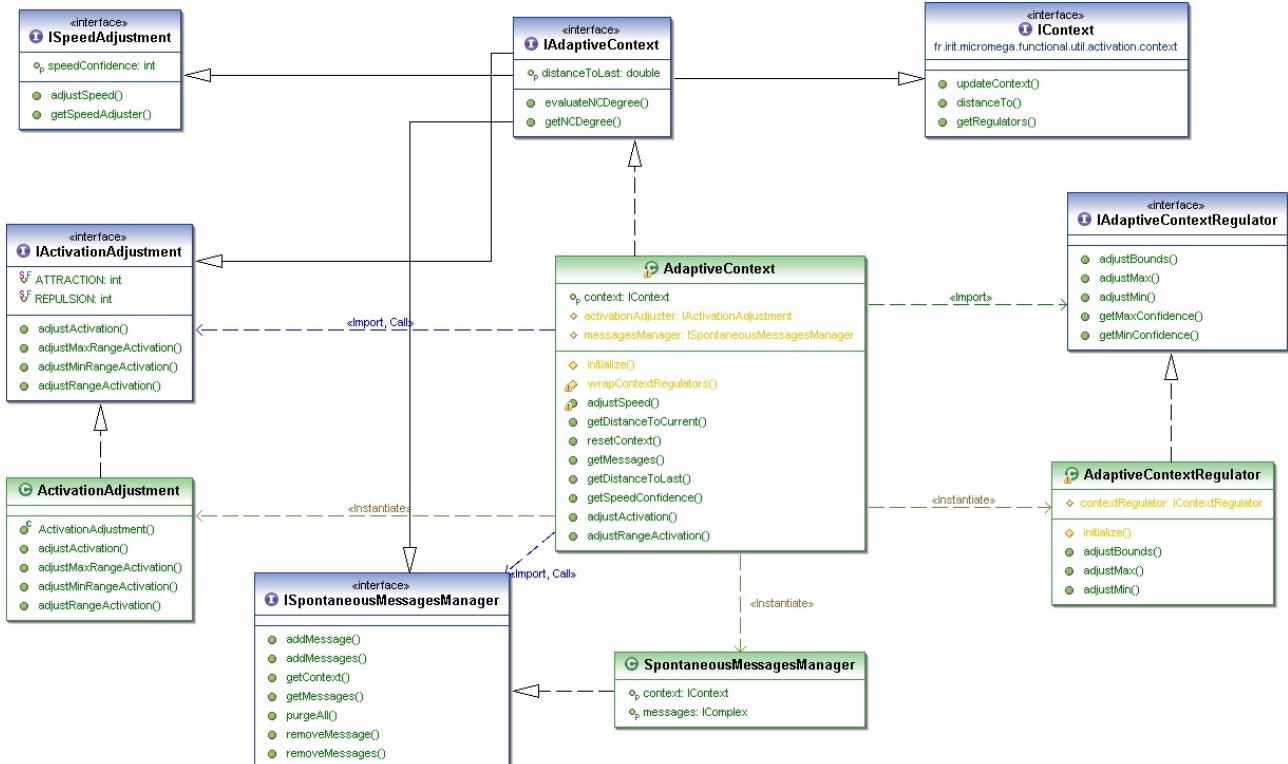


Diagramme 23: *IAdaptiveContext* et son implémentation

De plus, un petit module de messagerie permet de gérer les messages à envoyer au agents régulateurs pour modifier leur quantité si besoin est. Ces messages seront envoyés en mode « toggle » (voir module de messagerie en section 4.4) lors de la prochaine activation de ce contexte.

4.6.c Paquetage « util.parameterAdjuster »

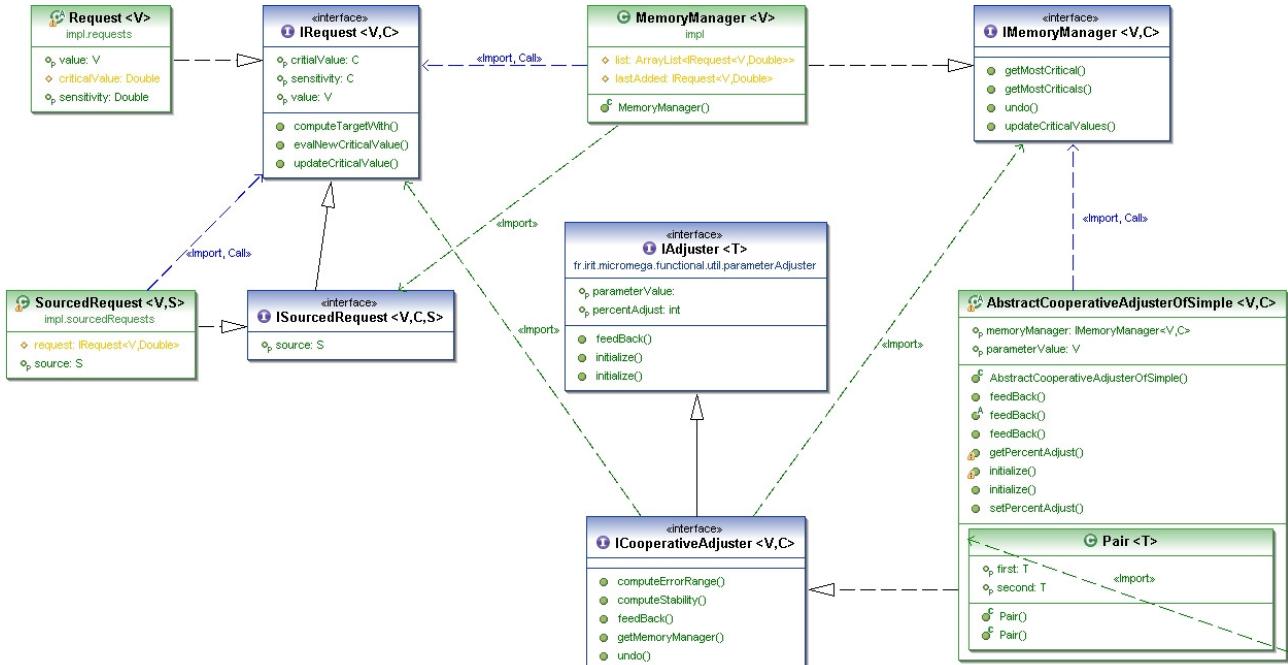


Diagramme 24: `IAdjuster` et l'implémentation d'un ajusteur coopératif

Nous utilisons dans microMéga des ajusteurs par paramètre générique (`IAdjuster<T>`) permettant de modifier au mieux sa valeur en fonction d'une succession de « feedbacks ». Plus précisément, nous utilisons des ajusteurs coopératifs qui enregistrent les feedbacks comme des requêtes dans un `IMemoryManager` et les traitent selon un algorithme coopératif (prise en compte de la « pire » requête et modification conséquente qui ne génère potentiellement pas une nouvelle pire requête dans le sens de modification opposé).

Notre implémentation rajoute de plus (non visible sur le diagramme) toutes les requêtes dans le module de représentation de l'agent au même niveau de l'arborescence que le paramètre qui est associé à l'ajusteur.

5. Le paquetage « view »

Les agents vues sont définis de la même manière que les agents fonctionnels (fichiers xml et classes spécifiques pour les comportements), cependant leur implémentation est relativement plus aisée car leur comportements sont plus simples.

L'activité première des agents vues est de surveiller spécifiquement des aspects donnés d'un ou de plusieurs agents fonctionnels. Dans microMéga, seuls deux types très simples d'agents vues ont été utilisés jusqu'ici :

1. Les *ElementViewer* agents : ils relèvent, à chaque cycle, les valeurs de quantité d'un groupe d'agents éléments donné et mesurent éventuellement l'écart de ces valeurs avec des valeurs références. En phase de coopération, ces agents envoient les erreurs relevées aux agents concernés. Ces agents sont utilisés comme point d'entrée pour permettre au système de s'adapter à des données expérimentales connues.

2. Les *ElementSetterView* : ils relèvent, à chaque cycle, les valeurs de quantité d'un groupe d'agents éléments donné et les remettent éventuellement à leur valeur de référence. Ces agents sont utilisés pour « piloter » directement certaines valeurs pour pouvoir, par exemple, maintenir stable la quantité de O2.

Du fait de leurs fonctionnalités nominales très proches, le comportement des *ElementSetterView* est étendu à partir de celui des agents *ElementViewer*. Ce dernier est basé sur la donnée d'un fichier externe contenant le nom des agents éléments à surveiller et d'une liste de couples <numéroCycle, valeur> pour chacun de ces éléments. A chaque pas, un *ElementViewer* :

- vérifie s'il doit recharger les données du fichier (utilisation du *readDataFromFile()* dans ce cas) ;
- récupère la valeur de quantité courante de chacun des agents éléments et, pour les éléments ayant un couple défini pour le cycle courant, calcule et stocke l'erreur.

Les *ElementSetterView* corrigent directement l'erreur au lieu de la stocker.

En phase d'ajustement, les *ElementViewer* envoient un message de non coopération si l'erreur relevée n'est pas nulle. Les *ElementViewerSetter* ne font rien pendant la phase de tuning.

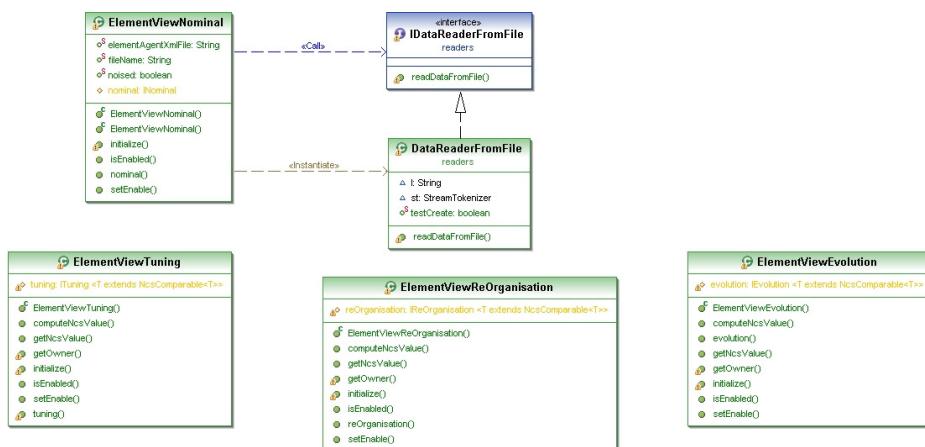


Diagramme 25: Comportements des agents d'observation

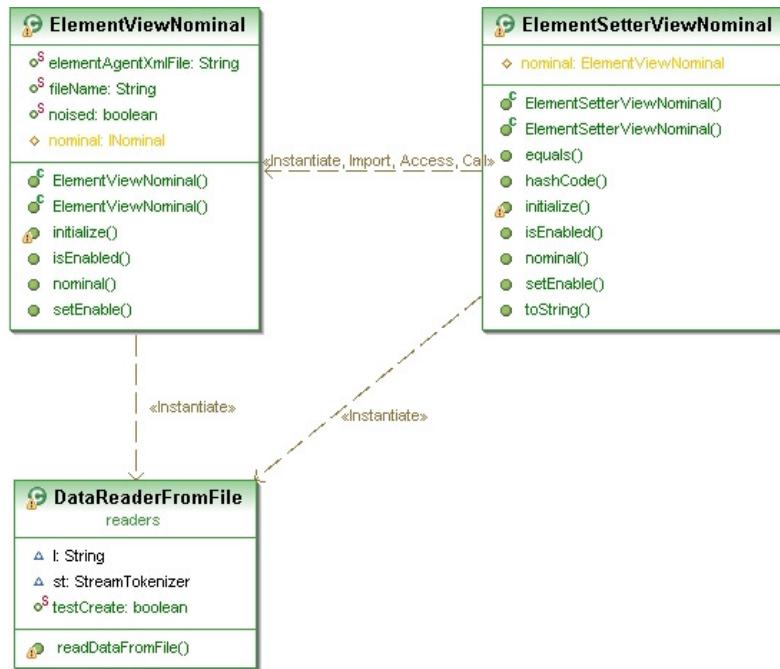


Diagramme 26: Comportement nominal des agents "setter"