# une plateforme générique et paramétrable nécessaire

**Plan du chapitre :**

*Le modèle formel et les algorithmes présentés dans le chapitre précédent sont avancés comme suffisants pour construire des simulations en conservant le morphisme entre modèle et implémentation. Afin d'en confirmer la faisabilité et ainsi montrer que les avantages avancés de l'approche* IODA *ne sont pas purement théoriques, nous décrivons dans ce chapitre :*

- *Une implémentation fidèle du modèle formel et des algorithmes en une plateforme de simulation paramétrable appelée* JEDI *(Java Environment for the Design of agent Interactions) ;*
- *Une implémentation de la méthodologie de conception* IODA *appelée* JEDI-BUILDER *permettant de construire graduellement un modèle, et de l'implémenter sur la plateforme* JEDI.

*En plus de confirmer la faisabilité de l'approche* IODA*, ces implémentations ont aussi servi de support à l'exploration des problématiques de simulation selon la perspective d'une approche centrée sur les interactions, que nous menons dans la partie III.*

## 5.1 Réification de IODA dans la plateforme JEDI

Le morphisme entre modèle et implémentation est garanti dans JEDI par l'implémentation quasi-systématique d'un élément du modèle formel de IODA par une classe, une classe abstraite, ou une interface lui étant dédiée. Nous ne décrivons ici que la part émergée de l'implémentation, *i.e.* la part de l'implémentation manipulée par les concepteurs afin d'implémenter les spécifications issues de la méthodologie IODA.

**Choix effectués dans JEDI** L'approche IODA est générique, et peut en particulier être utilisée pour implémenter des environnements, ou des modèles de sélection d'interaction très différents. Dans JEDI, nous avons fait le choix de nous restreindre aux environnements, modèles de sélection d'interaction que l'on retrouve le plus souvent en simulation : les environnements euclidiens en deux dimensions, et les modèles de sélection d'interaction réactifs. Nous avons de plus fait le choix d'effectuer l'implémentation en JAVA pour trois raisons :

- C'est un langage orienté-objet, ce qui facilite la construction d'une plateforme modulaire où presque chaque concept de IODA est représenté par une classe.
- C'est l'un des langages de programmation les plus utilisés actuellement. JEDI est ainsi ouvert à un public aussi large que possible.
- C'est un langage reposant sur un typage fort. Il permet ainsi de réifier la notion de signature sous la forme d'interface et donc de vérifier plus aisément si un agent se conforme à la signature de la

source ou la cible dans interaction.
Les langages permettant d'implémenter l'approche IODA ne se limitent toutefois pas au JAVA, ou plus généralement aux langages orientés-objet. En effet, des prototypes ont pu être implémentés dans le langage Netlogo [WC99] et dans le langage Prolog [Col90].

**L'environnement dans JEDI** Dans JEDI, nous avons fait le choix de restreindre les simulations aux environnements les plus couramment rencontrés en simulation informatique : les environnements euclidiens en deux dimensions, pouvant être non-toriques ou toriques selon l'axe des abscisses, des ordonnées, ou les deux. Ce choix a plusieurs implications sur l'implémentation que nous fournissons :

- Nous supposons l'ensemble des primitives de l'environnement fixe et ne pouvant pas être modifié.
- La signature de l'environnement dans les interactions ou le halo des entités ne peut contenir que des sous-ensembles de ces primitives. Dans JEDI, ces signatures n'ont pas à être définies : elles sont définies implicitement, en leur ajoutant l'intégralité des primitives définies pour les environnement euclidiens.
- Les entités ont une position dans l'environnement, et possèdent donc deux attributs *abscisse* et *ordonnée* ainsi que des primitives permettant d'y accéder.
- Les entités sont soit ponctuelles, soit occupent une surface dans la simulation. Cette surface est approximée dans JEDI par une « bounding box », *i.e.* un rectangle dont les côtés sont parallèles à l'axe des abscisses et des ordonnées.
- Le halo des entités est exprimé à l'aide d'une surface de perception centrée sur la position de l'entité, appelée *surface de perception référente*. Le voisinage d'une entité sont les entités dont la surface intersecte la surface de perception.

Le détail des principes de calcul de la distance, du calcul du voisinage, et plus généralement les primitives fournies par l'environnement, et leur implémentation sont décrites dans l'annexe A page 259. Il est à noter que l'extension de JEDI à d'autres formes d'environnement ne nécessite pas de modification profonde des diagrammes UML que nous décrivons ici.

**Le modèle de sélection d'interaction dans JEDI** Nous restreignons dans JEDI les modèles de sélection d'interaction au modèle réactif basé sur une matrice d'interaction raffinée, que nous avons présenté dans le chapitre précédent. Ce modèle fournit, à notre sens, un bon compromis entre la complexité du modèle utilisé, et la complexité des comportements qu'il est possible de modéliser simplement.

En effet, bien le modèle utilisé soit réactif, il est possible de modéliser des agents cognitifs. Par exemple, les primitives de perception et d'actions peuvent se baser sur un modèle fondé sur les croyances et les connaissances de l'entité, et utiliser ainsi des moteurs d'inférences de connaissance que l'on retrouve chez certains agents cognitifs, ou encore les interactions peuvent produire et consommer des buts poursuivis par une entité, et permettre ainsi de modéliser de la planification réactive.

**Vue d'ensemble sur le modèle IODA dans JEDI** Les éléments logiciels que nous présentons sont résumés sous la forme de diagrammes UML dans les figures 5.1 et 5.2. La plateforme JEDI est actuellement composée de 95 classes et interfaces, totalisant 14820 lignes de code physiques (commentaires et mise en page inclus), équivalentes à 5541 lignes de code logique [16].

**Remarque.** JEDI est une plateforme écrite en anglais. Tous les éléments du modèle décrits jusqu'à présent y sont donc traduits.

### 5.1.1 Interactions

Le modèle d'une interaction est un 8-uplet $\mathcal{I}$ tel que :

$$\mathcal{I} = \langle id, noms, cardinalite, signature, sign_{env}, preconditions, declencheur, actions \rangle$$

L'implémentation de ce modèle dépend de la façon d'implémenter la signature des entités dans une interaction et la signature de l'environnement dans une interaction.

16. comptabilisées selon la métrique SLOC de la commande unix `sloccount`

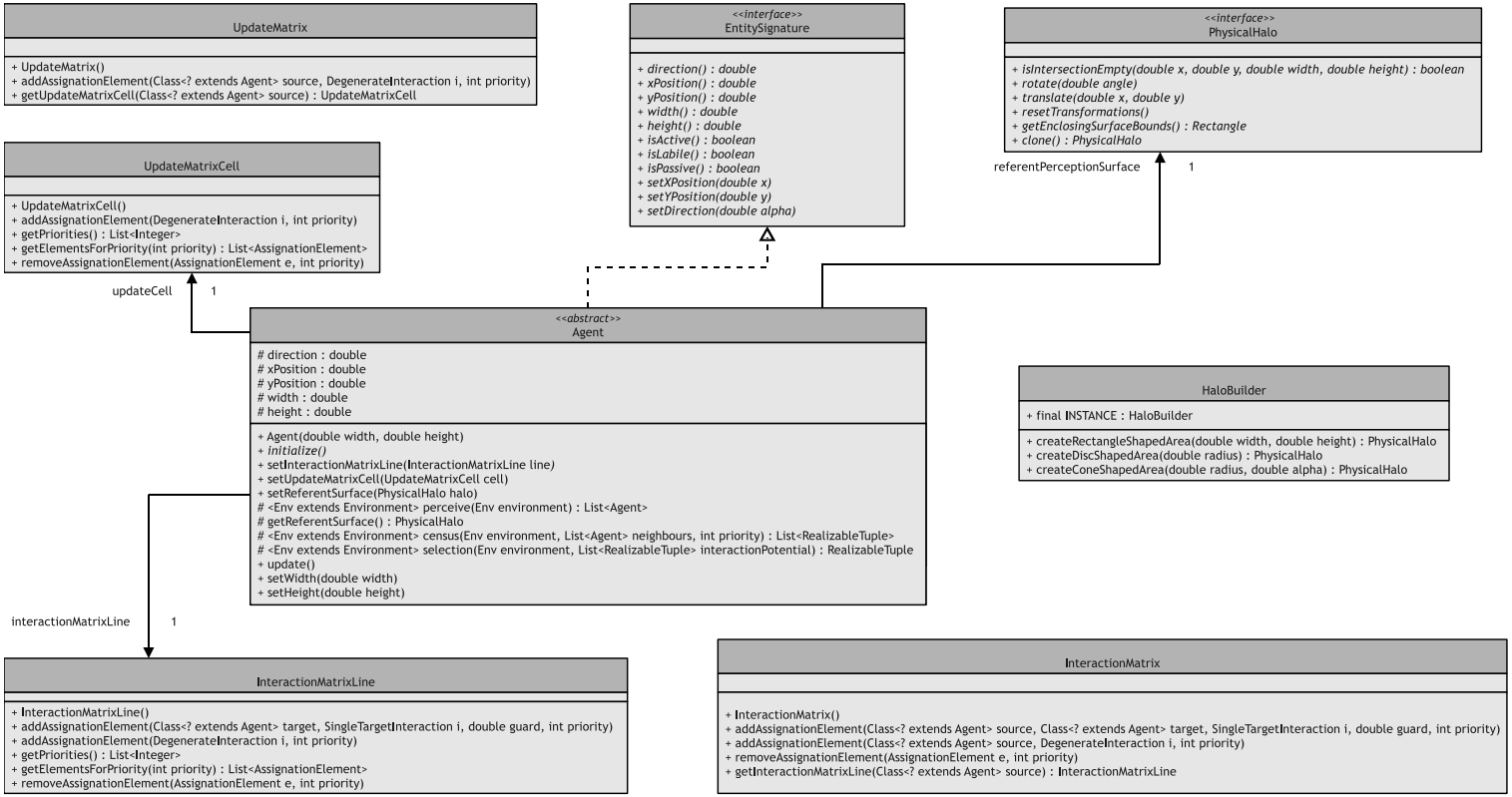


FIGURE 5.1 – Première partie du diagramme UML décrivant les classes et interfaces de la plateforme JEDI directement manipulées pour implémenter un modèle décrit avec la méthodologie IODA.

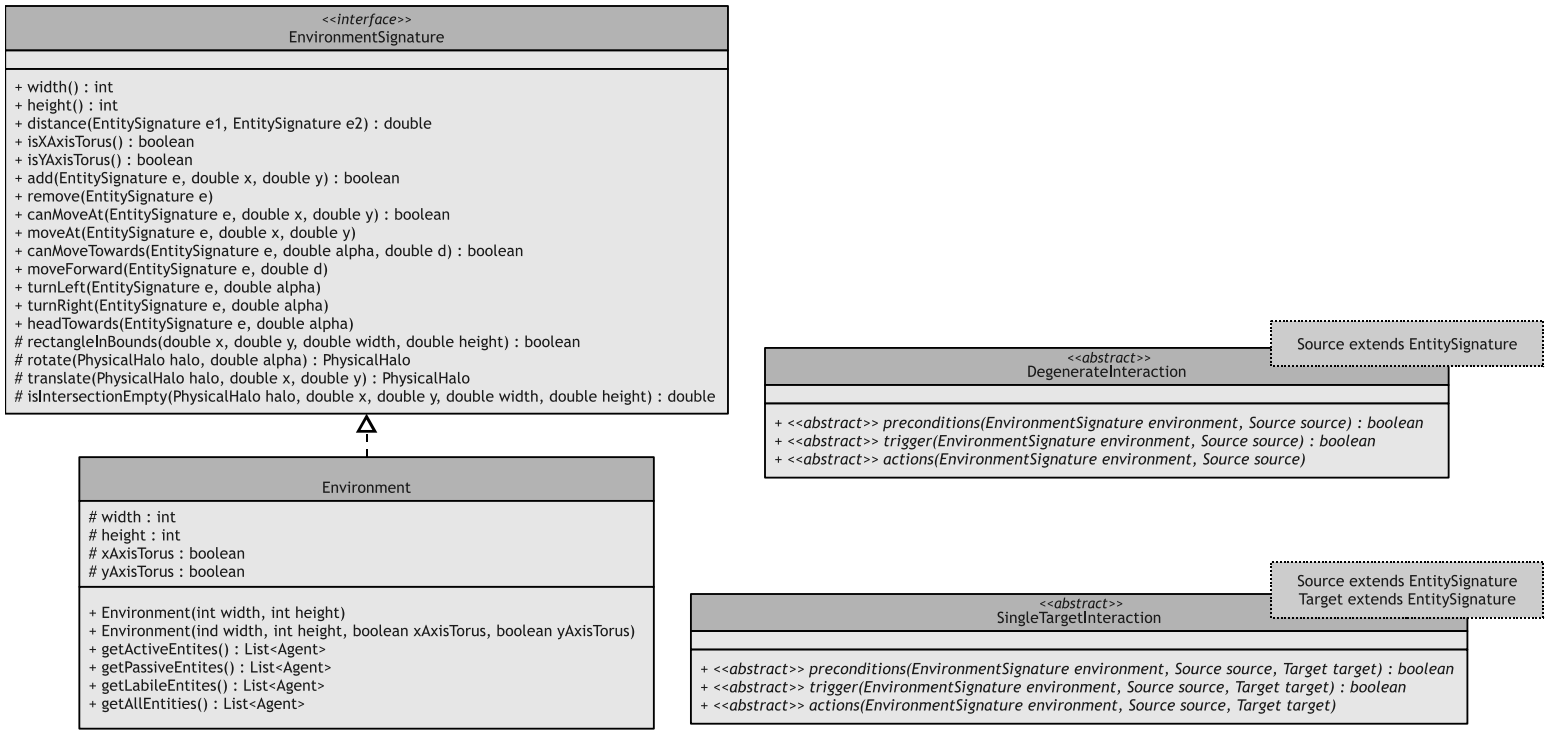


FIGURE 5.2 – Seconde partie du diagramme UML décrivant les classes et interfaces de la plateforme JEDI directement manipulées pour implémenter un modèle décrit avec la méthodologie IODA.

Dans le modèle IODA, la signature d'une entité dans une interaction est représentée par un couple $S$ tel que :

$$S = \langle id(S), primitives(S) \rangle$$

Dans JEDI, ce couple est implémenté sous la forme d'une interface étendant l'interface nommée *EntitySignature*. La valeur $id(S)$ correspond au nom de l'interface créée, et l'ensemble $primitives(S)$ des primitives abstraites de cette signature sont représentées par des méthodes dans l'interface. Ces interfaces correspondent aux éléments contenus dans l'ensemble $signatures(\mathcal{I})$ du 7-uplet définissant une interaction. Le diagramme UML de cette spécification apparaît sur la figure 5.1.

**Remarque.** Nous nous plaçons dans le cadre des environnements euclidiens en deux dimensions décrits précédemment. Par conséquent, toutes les entités d'une simulation doivent implémenter les primitives abstraites provenant de la signature des entités dans l'environnement, dont un résumé est fourni dans le tableau A.5 page 270 de l'annexe A. Pour des raisons exposées par la suite, ces primitives sont intégrées à l'interface *EntitySignature*.

Le modèle de la signature de l'environnement dans une interaction est un élément $S$ tel que :

$$S = \langle (p_i)_{i \in \mathbb{N}} \rangle$$

Sa structure est très similaire au modèle de la signature d'une entité dans une interaction. Par conséquent, son implémentation dans JEDI use de moyens similaires, sous la forme d'une interface nommée *EnvironmentSignature*, qui correspond à l'élément $sign_{env}$ du 8-uplet définissant une interaction. Les primitives de l'environnement de cette signature sont représentées par des méthodes dans l'interface.

Le diagramme UML de cette spécification apparaît sur la figure 5.2.

**Remarque.** Nous nous plaçons dans le cadre des environnements euclidiens en deux dimensions décrits précédemment. Par conséquent, toutes les entités d'une simulation doivent implémenter les primitives abstraites provenant de la signature des entités dans l'environnement, dont un résumé est fourni dans le tableau A.5 page 270 de l'annexe A. Pour des raisons exposées par la suite, ces primitives sont intégrées à l'interface *EntitySignature*.

Nous nous plaçons dans le cadre des environnements euclidiens en deux dimensions décrits précédemment. Par conséquent, l'environnement fournit un ensemble important de primitives de l'environnement, dont un résumé est fourni dans le tableau A.5 page 270 de l'annexe A. Pour des raisons exposées par la suite, ces primitives sont intégrées à l'interface *EnvironmentSignature*.

L'approche IODA est présentée dans ce chapitre uniquement dans le cadre d'interactions de cardinalité $(1, 1)$ (les interactions individuelles) et $(1, 0)$ (les interactions dégénérées). Dans JEDI, nous identifions la cardinalité $card(\mathcal{I})$ de l'interaction par la classe utilisée pour implémenter son modèle.

Les interactions individuelles sont représentées par une classe étendant la classe parente à toutes les interactions individuelles, nommée *SingleTargetInteraction*. L'identifiant *id* de cette interaction est alors utilisé pour nommer la classe ainsi créée. Cette classe est paramétrée par deux types génériques appelés *Source* et *Target*, qui correspondent respectivement à la signature de l'entité source, et à la signature de l'entité cible dans l'interaction. Puisque l'on sait que l'élément *signature* du modèle de l'interaction contient les deux signatures $sign_{source}(\mathcal{I})$ et $sign_{cible}(\mathcal{I})$, on a :

- $Source = id\Big(sign_{source}(\mathcal{I})\Big)$, où $id\Big(sign_{source}(\mathcal{I})\Big)$ est le nom de l'interface représentant la signature de l'entité source dans l'interaction ;
- $Cible = id\Big(sign_{cible}(\mathcal{I})\Big)$, où $id\Big(sign_{cible}(\mathcal{I})\Big)$ est le nom de l'interface représentant la signature de l'entité cible dans l'interaction.

Les interactions dégénérées sont représentées de manière similaire, par une classe étendant la classe parente à toutes les interactions dégénérées, nommée *DegenerateInteraction*. L'identifiant $id(\mathcal{I})$ de cette interaction est alors utilisé pour nommer la classe ainsi créée. Cette classe est paramétrée par un type générique appelé *Source*, dont la définition a déjà été fournie dans le paragraphe précédent.

Les interactions ne manipulent pas directement les entités et l'environnement, mais uniquement leurs primitives. Par conséquent, le type des paramètres des préconditions, du déclencheur et des actions d'une

interaction sont la signature des entités impliquées dans l'interaction, ainsi que la signature de l'environnement dans l'interaction, *i.e.* les types génériques *Source* et *Target*, et l'interface *EnvironmentSignature*. De par la forme préconisée pour spécifier les préconditions, le déclencheur, et les actions dans le modèle d'une interaction, l'implémentation des méthodes éponymes dans la classe *SingleTargetInteraction* est directe et non ambiguë. Le diagramme UML de la figure 5.2 résume la structure des interactions dans JEDI.

**Remarque.** Comme les préconditions, le déclencheur et les actions de l'interaction sont susceptibles de manipuler les primitives abstraites et les primitives de issues de la description de l'environnement euclidien, nous avons fait le choix d'intégrer toutes les primitives liées à ce type d'environnement dans les interfaces *EntitySignature* et *EnvironmentSignature*.

### 5.1.2 Matrice d'interaction

Dans IODA, le modèle d'une matrice d'interaction brute est une fonction $\mathcal{M}(S,T)$ telle que :

$$\mathcal{M} \ : \ \begin{array}{lll} \mathbb{F} \times \big(\mathbb{F} \cup \{\emptyset\}\big) & \rightarrow & \mathcal{P}\Big((\mathbb{I}_{(1,0)} \times \mathbb{F}) \cup (\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F})\Big) \\ (\mathcal{S},\mathcal{T}) & \rightarrow & a_{\mathcal{S}/\mathcal{T}} \end{array}$$

Dans cette notation, $\mathbb{F}$ est l'ensemble des familles d'entités de la simulation, $\mathbb{I}_{(1,0)}$ est l'ensemble des interactions dégénérées de la simulation, et $\mathbb{I}_{(1,1)}$ est l'ensemble des interactions individuelles de la simulation. Dans le cadre de cette thèse, le modèle de sélection d'interaction d'une entité consiste à construire une matrice d'interaction raffinée, qui prend la forme d'une fonction $\mathcal{M}_{raff}(a)$ telle que :

$$\mathcal{M}_{raff} \ : \ \begin{array}{lll} \Big(\mathbb{I}_{(1,0)} \times \mathbb{F}\Big) \cup \Big(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}\Big) & \rightarrow & \mathbb{Z} \\ (a) & \rightarrow & \mathcal{M}_{raff}(a) \end{array}$$

Dans cette notation, $a$ est un élément d'assignation, contenu dans l'une des assignations $\mathcal{M}(S,T)$ de la matrice d'interaction brute.

Dans JEDI, une famille d'entités est représentée par une classe étendant la classe abstraite parente de toute famille d'entités, nommée *Agent*. Par conséquent, les paramètres de la fonction $\mathcal{M}$ sont des instances de la classe *Class< ? extends Agent>*.

Dans JEDI, nous faisons le choix d'implémenter la matrice d'interaction brute et la matrice d'interaction raffinée dans une seule et même classe, nommée *InteractionMatrix*. Cette classe utilise des sous classes telles que *Assignation*, *AssignationElement* ou *InteractionMatrixLine* comme structure de données permettant de mémoriser les assignations $\mathcal{M}(S,T)$, ainsi que leur priorité définie par $\mathcal{M}(a)$. La description de ces éléments sort du cadre de ce chapitre, qui offre une présentation générale de JEDI. Deux méthodes sont définies pour ajouter les différents éléments d'assignation présents dans la matrice d'interaction. La première s'intitule *addAssignationElement(Class< ? extends Agent>, Class< ? extends Agent>, SingleTargetInteraction,double,int)*, et permet d'ajouter un élément d'assignation individuel à la matrice d'interaction brute, tout en lui associant une priorité dans la matrice d'interaction raffinée. La seconde méthode s'intitule *addAssignationElement(Class< ? extends Agent>, DegenerateInteraction,int)*, et permet d'ajouter un élément d'assignation dégénéré à la matrice d'interaction brute, tout en lui associant une priorité dans la matrice d'interaction raffinée. Une dernière méthode intitulée *getInteractionMatrixLine(Class< ? extends Agent>)* permet d'avoir accès à la ligne de la matrice d'interaction pour une famille d'entités source particulière. Le diagramme UML de cette classe est fourni sur la figure 5.1.

Une ligne de la matrice d'interaction est implémentée à l'aide de la classe *InteractionMatrixLine*. Cette classe est instanciée et initialisée automatiquement par un appel à la méthode *getInteractionMatrixLine(Class< ? extends Agent>)* de la classe représentant la matrice d'interaction. Cette ligne de la matrice d'interaction fournit en particulier deux méthodes nécessaires à l'implémentation du moteur de simulation. La première méthode, intitulée *getPriorities()*, est utilisée lors de la sélection d'interaction, et permet de connaître toutes les priorités présentes dans cette ligne de la matrice. La seconde méthode est intitulée *getElementsForPriotity(int)*, et permet de récupérer l'ensemble des éléments d'assignation ayant une priorité particulière. Cette méthode est utilisée pour calculer le potentiel d'interaction d'une priorité particulière. Le diagramme UML de cette classe est fourni sur la figure 5.1.

### 5.1.3 Matrice de mise à jour

Dans IODA, le modèle d'une matrice de mise à jour est une fonction $\mathcal{U}(S)$ telle que :

$$\mathcal{U} : \begin{array}{ccc} \mathbb{F} & \rightarrow & \mathcal{P}(\mathbb{I}_{(1,0)} \times \mathbb{F}) \\ \mathcal{S} & \rightarrow & (u^i_{\mathcal{S}}, \mathcal{S})_{i \in \mathbb{N}} \end{array}$$

Dans cette notation, $\mathbb{F}$ est l'ensemble des familles d'entités de la simulation, et $\mathbb{I}_{(1,0)}$ est l'ensemble des interactions dégénérées de la simulation. Afin d'implémenter la mise à jour des entités, le modèle d'une matrice de mise à jour ordonnée $\mathcal{U}_{ord}(u)$ a aussi été introduit :

$$\mathcal{U}_{ord} : \begin{array}{ccc} \mathbb{I}_{(1,0)} \times \mathbb{F} & \rightarrow & \mathbb{Z} \\ (u) & \rightarrow & \left\{ \begin{array}{ll} \mathcal{U}_{ord}(u) & \text{si } u \in \mathcal{U}(\mathcal{S}) \\ \text{non défini} & \text{sinon} \end{array} \right. \end{array}$$

Dans cette notation, $u$ est un élément d'assignation dégénéré, contenu dans l'une des assignations dégénérée $\mathcal{U}(S)$ de la matrice de mise à jour.

Dans JEDI, nous faisons le choix d'implémenter la matrice de mise à jour et la matrice de mise à jour ordonnée dans une seule et même classe, nommée *UpdateMatrix*. Cette classe utilise des sous classes telles que *DegenerateAssignation*, *DegenerateAssignationElement* ou *UpdateMatrixCell* comme structure de données permettant de mémoriser les assignations dégénérées $\mathcal{U}(S)$, ainsi que leur priorité définie par $\mathcal{U}_{ord}(u)$. La description de ces éléments sort du cadre de ce chapitre. Une méthode intitulée *addAssignationElement(Class< ? extends Agent>, DegenerateInteraction,int)* est définie afin d'ajouter les différents éléments d'assignation dégénérés présents dans la matrice de mise à jour. Elle permet d'ajouter un élément d'assignation dégénéré à la matrice de mise à jour, tout en lui associant une priorité dans la matrice de mise à jour ordonnée. Une autre méthode intitulée *getUpdateMatrixCell(Class< ? extends Agent>)* permet d'avoir accès à la cellule de la matrice de mise à jour pour une famille d'entités source particulière. Cette méthode est utilisée lors de la spécification d'une famille d'entités. Le diagramme UML de cette classe est fourni sur la figure 5.1.

Une cellule de la matrice de mise à jour est implémentée à l'aide de la classe *UpdateMatrixCell*. Cette classe est instanciée et initialisée automatiquement par un appel à la méthode *getUpdateMatrixCell(Class< ? extends Agent>)* de la classe représentant la matrice de mise à jour. Cette cellule de la matrice de mise à jour fournit en particulier deux méthodes nécessaires à l'implémentation du moteur de simulation. La première méthode est intitulée *getPriorities()*, et permet de connaître toutes les priorités présentes dans cette cellule de la matrice. Cette méthode est utilisée lors de la mise à jour de l'état d'une entité. La seconde méthode est intitulée *getElementsForPriotity(int)*, et permet de récupérer l'ensemble des éléments d'assignation ayant une priorité particulière. Cette méthode joue le rôle de $\mathcal{U}^{-1}_{ord}$ dans l'algorithme de mise à jour d'une entité. Le diagramme UML de cette classe est fourni sur la figure 5.1.

### 5.1.4 Environnement

Dans IODA, le modèle de l'environnement est un quadruplet $\varepsilon$ tel que :

$$\varepsilon = \langle id, primitives_{env}, sign_{entites}, attributs, valeur_{attr} \rangle$$

Dans JEDI, ce modèle est implémenté sous la forme de la classe *Environment*, qui étend l'interface *EnvironmentSignature*. Les primitives de l'environnement $primitives_{env}$ y sont implémentées par des méthodes, et les attributs *attributs* du modèle IODA sont implémentés par des attributs dans la classe ainsi créée. Puisque l'environnement est une structure figée dans JEDI (*i.e.* on ne peut y ajouter de nouvelles primitives de l'environnement), la signature de l'environnement et la signature des entités dans les diverses primitives de l'environnement sont égales respectivement à *EnvironmentSignature* et *EntitySignature*.

La classe *Environment* fournit une implémentation de toutes les primitives de l'environnement décrites dans l'annexe A. Elle offre de plus une implémentation à diverses méthodes nécessaires au fonctionnement du moteur de simulation. Parmi ces méthodes figurent *getActiveEntities()*, *getPassiveEntities()*, *getLabileEntities()* ou *getAllEntities()*, qui permettent d'avoir accès en lecture et en écriture à l'ensemble

des entités actives de la simulation (*i.e.* $\mathbb{E}_{active}$), à l'ensemble des entités passives de la simulation (*i.e.* $\mathbb{E}_{passive}$), à l'ensemble des entités labiles de la simulation (*i.e.* $\mathbb{E}_{labile}$), et enfin à l'ensemble des entités de la simulation (*i.e.* $\mathbb{E}$). Le diagramme UML de cette classe apparaît sur la figure 5.2.

### 5.1.5 Entités

Dans IODA, le modèle d'une famille d'entités est un 8-uplet $\mathcal{F}$ tel que :

$$\mathcal{F} = \langle id(\mathcal{F}), attributs(\mathcal{F}), primitives(\mathcal{F}), \mathcal{H}(\mathcal{F}), \mathcal{U}_{ord}(\mathcal{F}), ligne(\mathcal{F}), colonne(\mathcal{F}), selection(\mathcal{F})\rangle$$

Dans JEDI, ce modèle est implémenté sous la forme d'une classe étendant à la fois la classe abstraite *Agent*, et les interfaces représentant la signature de cette famille d'entités dans les diverses interactions auxquelles elle peut participer. La valeur $id(\mathcal{F})$ correspond au nom de la classe ainsi créée. L'ensemble $primitives(\mathcal{F})$ des primitives figurant dans les signatures sont implémentées sous la forme de méthodes, et l'ensemble des attributs $attributs(\mathcal{F})$ du modèle de la famille d'entités est implémenté sous la forme d'attributs dans cette classe. La ligne $ligne(\mathcal{F})$ dans la matrice d'interaction associée à cette famille d'entités est implémentée sous la forme d'un attribut nommé *interactionMatrixLine*, dont la classe est *InteractionMatrixLine*. La cellule $\mathcal{U}_{ord}(\mathcal{F})$ de la matrice de mise à jour ordonnée associée à cette entité est implémentée sous la forme d'un attribut nommé *updateMatrixCell*, dont la classe est *UpdateCellMatrix*. Le modèle de sélection d'interaction d'une entité est défini dans son attribut *interactionMatrixLine*.

Pour des raisons d'optimisation de calculs, la forme générale du halo $\mathcal{H}(\mathcal{F})$ d'une famille d'entités, où une fonction booléenne permet de vérifier l'appartenance d'une entité au voisinage, n'apparaît pas dans JEDI. À la place,une méthode intitulée *perceive(Environment)* est utilisée comme alternative. Elle est l'équivalent de la fonction $\mathcal{V}$ du modèle, qui calcule le voisinage d'une entité. De plus, nous fournissons une implémentation par défaut de cette méthode, où une entité perçoit les autres entités présentes dans une portion de l'environnement, identifiée à l'aide d'une surface. Ce type de halo implique la spécification d'une unique primitive abstraite appelée *referentPerceptionSurface()*, qui permet de connaître la surface de perception référente de l'entité (*i.e.* la surface de l'environnement dans laquelle les autres agents sont considérés comme perçus). Dans JEDI, nous considérons que cette méthode est un accesseur vers un attribut intitulé *referentPerceptionSurface*, dont la classe, intitulée *PhysicalHalo*, représente une surface de l'environnement. La valeur de cet attribut est initialisée par la méthode *setReferentPerceptionSurface(PhysicalHalo)*, dont le paramètre est obtenu à l'aide d'une fabrique abstraite, définie par la classe *HaloBuilder*. Cette classe définit trois types de surface de perception, construites à l'aide des méthodes :

- la méthode *createRectangleShapedArea(double, double)* permet de créer une surface de perception rectangulaire ;
- la méthode *createDiscShapedArea(double)* permet de créer une surface de perception prenant la forme d'un disque. Le voisinage est alors composé de toutes les entités située à une distance inférieure ou égale au rayon de ce disque ;
- la méthode *createConeShapedArea(double, double)* permet de créer une surface de perception en forme de cône.

Les surfaces de perception référentes n'y sont toutefois pas restreintes, et peuvent être complétées par des classes étendant l'interface *EntitySignature*.

La classe *Agent* définit aussi d'autres méthodes telles que census, selection ou upate, qui implémentent les différents algorithmes décrits dans le chapitre précédent. Nous ne décrivons pas ces méthodes dans ce chapitre, puisqu'elles ne constituent qu'une implémentation directe en JAVA de ces algorithmes. Le diagramme UML de cette classe apparaît sur la figure 5.1.

## 5.2 Un moteur générique et paramétrable nécessaire

Puisqu'une plateforme de simulation est un outil permettant de faciliter la conception de simulations, il apparaît normal d'y demander la spécification *a minima* d'un simulateur, et donc de réutiliser un maximum de concepts déjà implémentés. Toutefois, le simulateur ne doit pas devenir une boîte noire pour autant. En effet, dans une simulation explicative, le procédé suivi pour obtenir les résultats est tout aussi important, voire plus important que les résultats mêmes, contrairement aux autres types de

simulation qui ont été mentionnés dans la section 1.1.2 du chapitre 1, où seul le résultat obtenu compte, et pas le procédé (et les calculs) pour y parvenir. Dans ce cadre, les algorithmes utilisés pour spécifier une simulation particulière ne sont pas forcément adaptés à d'autres simulations. Nous illustrons ce point pour le modèle de tri collectif, et le modèle SugarScape.

***Exemple. Tri « collectif » et SugarScape***

*Dans un modèle de « tri collectif »*[17]*, par exemple dans le modèle décrit par Resnick [Res97], le nombre d'entités trieuses n'a pas d'influence sur l'apparition de tas [GRHT07]. Par conséquent, l'ordre dans lequel le processus comportemental des entités est déclenché n'a aucune influence sur les résultats obtenus.*

*Au contraire, dans la simulation* SugarScape *développée par Epstein et Axtell [EA96], l'ordre dans lequel le processus comportemental des entités est déclenché influe fortement sur la nature des résultats expérimentaux. En effet, l'objectif de cette simulation est de mesurer l'évolution d'une population d'entités se nourrissant d'une ressource disposée dans l'environnement sous la forme d'un tas. Si l'ordre de déclenchement du comportement des entités est identique pour chaque pas de temps, les entités les plus vieilles auront la priorité sur la consommation de ressources, ce qui a pour conséquence d'augmenter le taux de mortalité des entités plus jeunes. La pyramide des ages, qui constitue l'un des résultats étudiés de la simulation, s'en trouve modifiée.*

*Par conséquent, le moteur de simulation utilisé pour le tri collectif n'est pas adapté pour modéliser* SugarScape *[Mic04].*

La solution la plus simple à ce problème consiste à utiliser le moteur de simulation le plus complexe. Par exemple, dans les deux modèles mentionnés ci-dessus, un simulateur qui déclenche le comportement des entités dans un ordre aléatoire permet d'éviter les biais dans le modèle *SugarScape*, et permet aussi d'exécuter correctement une simulation de tri collectif. Toutefois, cette solution est obtenue au prix d'une complexification superflue du simulateur utilisé pour le tri collectif. Elle a pour conséquence la réduction du nombre d'expériences pouvant être effectuées en un temps donné, et donc une réduction de la viabilité des résultats expérimentaux obtenus. Elle fait de plus défaut au principe de parcimonie.

Ainsi, une plateforme supportant la conception de simulateurs doit prendre en compte une problématique à notre sens fondamentale : le contrôle fin du moteur de simulation, afin d'adapter la complexité du simulateur au problème simulé. Dans Jedi, nous offrons la possibilité de paramétrer finement trois aspects du moteur de simulation, via la classe singleton appelée *SimulationProperties* : le générateur de nombres aléatoires utilisé, l'ordonnanceur utilisé pour gérer l'activité des entités lors d'un pas de temps, ainsi que la façon de déterminer le sous-ensemble des entités à mettre à jour au début d'un pas de temps. Le diagramme UML résumant la structure logicielle mise en place pour gérer ces paramètres est résumée sur la figure 5.3.

### 5.2.1 Reproductibilité des résultats

La possibilité de reproduire exactement une simulation est l'un des fondements de la vérification d'une implémentation. En effet, la résolution d'erreurs dans un simulateur passe par la découverte d'une expérience particulière lors de laquelle des erreurs surviennent, puis par l'analyse de cette expérience afin d'identifier ce qui a participé à leur apparition. Cette analyse n'est possible que si l'expérience d'intérêt peut être exécutée à nouveau pas à pas, et donc être reproduite dans ses moindres détails. Ce déterminisme ne peut être atteint que par le contrôle des éléments régissant le non-déterminisme de la simulation dans l'implémentation : les générateurs de nombres aléatoires.

La génération d'un nombre aléatoire se fait dans Jedi via la méthode statique intitulée *random()* de la classe *SimulationProperties*. Cette méthode sans paramètres retourne un nombre à virgule flottante aléatoire (plus précisément, un double), situé dans l'intervalle $[0; 1[$. Puisque les algorithmes de simulation présentés dans le chapitre précédent utilisent par moment un générateur de nombre aléatoires, la reproductibilité d'une simulation ne peut être garantie dans Jedi que si ce générateur de nombre aléatoires est le seul à être utilisé.

17. Ce type de modèle est usuellement appelé « tri collectif », ce qui explique pourquoi nous utilisons cette terminologie ici. Il a toutefois été montré formellement par Gaubert *et al.*[GRHT07] que ce tri émerge aussi en la présence d'une unique entité trieuse. Le tri n'est donc pas « collectif ».

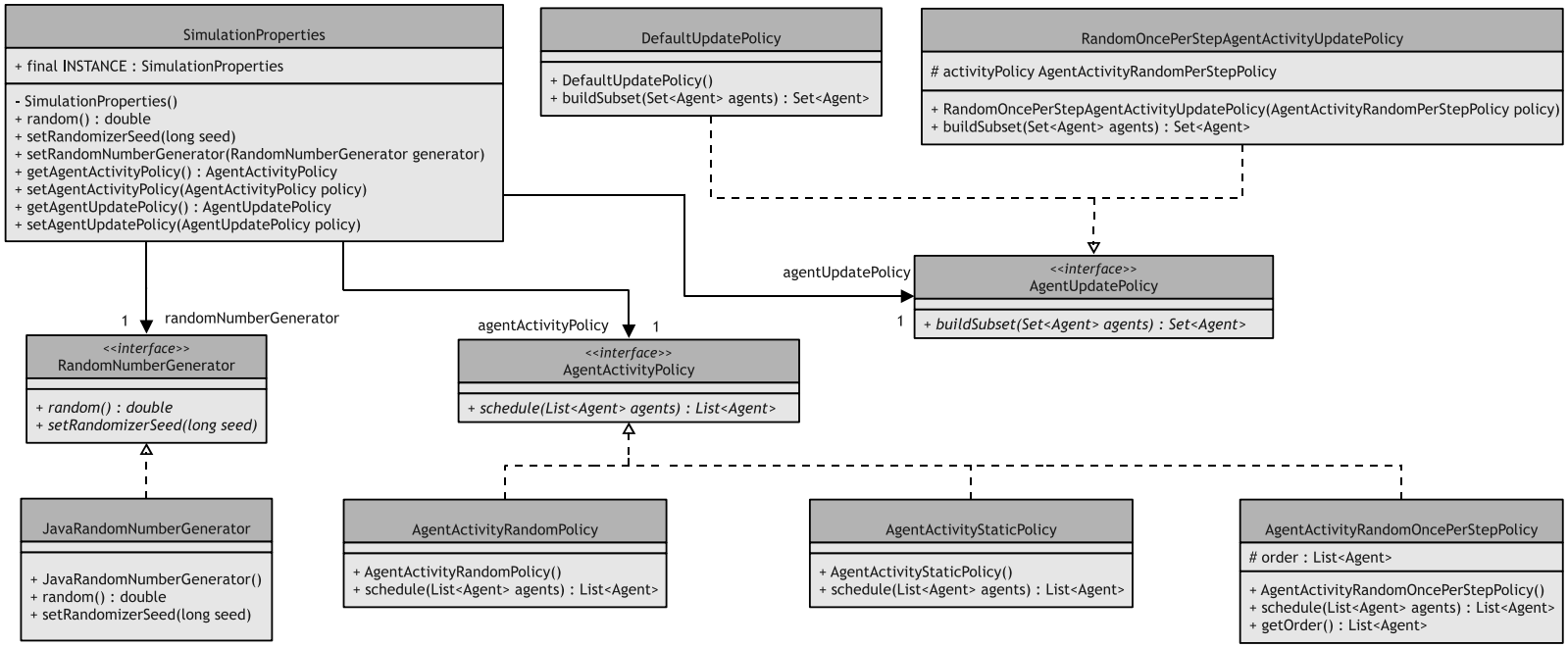


FIGURE 5.3 – Diagramme UML résumant la structure logicielle utilisée pour spécifier les paramètres de la plateforme JEDI.

Le premier paramètre de JEDI permet de caractériser finement comment les nombres aléatoires sont générés par la méthode *random()*. Ce paramètre est représenté sous la forme d'un attribut présent dans la classe *SimulationProperties* s'intitulant *randomNumberGenerator*, et qui implémente l'interface *RandomNumberGenerator*. Cette interface définit une méthode intitulée *setRandomizerSeed(long)* qui définit la graine utilisée pour initialiser le générateur de nombres aléatoires, ainsi qu'une méthode intitulée *random()*, qui retourne un nombre à virgule flottante aléatoire (plus précisément, un double), situé dans l'intervalle $[0; 1[$. La valeur de cet attribut peut être redéfinie par la méthode statique intitulée *setRandomNumberGenerator(RandomNumberGenerator)* de la classe *SimulationProperties*.

Pour simplifier l'initialisation et l'utilisation des générateurs de nombres aléatoires dans JEDI, nous avons de plus défini deux méthodes statiques intitulées *random()* et *setRandomizerSeed(long)* dans la classe *SimulationProperties*, dont le rôle est de déléguer l'appel aux méthodes éponymes de son attribut *randomNumberGenerator*. Nous proposons de plus une implémentation par défaut de l'interface *RandomNumberGenerator* que nous avons intitulée *JavaRandomNumberGenerator*, qui consiste a utiliser une instance de la classe *java.util.Random* fournie pour générer des nombres aléatoires.

### 5.2.2 Ordonnancement de l'activité des entités

Comme nous l'avons énoncé au début de cette sous-section, l'ordonnancement de l'activité des entités peut avoir une grande influence dans les résultats obtenus. Dans le cas où le temps est modélisé par un ensemble discret de pas de temps, au cours desquels chaque entité agit selon une séquence particulière, l'ordonnancement consiste à définir comment est construite cette séquence lors de chaque pas de temps.

Le second paramètre de JEDI permet de caractériser finement la construction de la séquence utilisée pour ordonner l'activité des entités, en ayant pour seule donnée l'ensemble des entités actives présentes dans l'environnement au début du pas de temps. Ce paramètre est représenté sous la forme d'un attribut présent dans la classe *SimulationProperties* qui s'intitule *agentActivityPolicy*, et qui implémente l'interface *AgentActivityPolicy*. Cette interface définit une unique méthode intitulée *schedule(List<Agent>)* qui est appelée dans le moteur de simulation au début de chaque pas de temps, en ayant pour paramètre l'ensemble des entités actives présentes dans l'environnement au début du pas de temps. Cette méthode construit puis retourne une instance de l'interface *List<Agent>*, qui représente la séquence dans laquelle les entités vont agir. La valeur de l'attribut *agentActivityPolicy* peut être redéfinie par la méthode statique intitulée *setAgentActivityPolicy(AgentActivityPolicy)* de la classe *SimulationProperties*.

La plateforme JEDI propose trois implémentations réutilisables de l'interface *AgentActivityPolicy*, qu'il est possible de compléter par des implémentations spécifiques à un problème de simulation particulier. Bien que le comportement le plus courant de la méthode *schedule(List<Agent>)* consiste à retourner une version réordonnée de liste des entités actives, l'ordonnancement dans JEDI n'y est pas restreint. Il peut aussi consister à extraire un sous-ensemble ordonné de cette liste, comme l'illustre la troisième implémentation que nous proposons.

La première façon d'ordonnancer, utilisée par défaut, est celle définie dans les algorithmes du chapitre 4. Elle consiste à réordonner aléatoirement la liste des entités actives fournie en paramètres, en utilisant le générateur de nombres aléatoires défini dans la classe *SimulationProperties*. Son implémentation est faite dans une classe intitulée *AgentActivityRandomPolicy*, dont l'implémentation de la méthode *schedule* est fournie dans l'algorithme 5.4). Cet ordonnancement est particulièrement utile pour des simulations telles que SugarScape, où l'ordre dans lequel les entités agissent a une forte influence sur les résultats obtenus.

FIGURE 5.4 – Ordonnancement aléatoire de l'activité des entités au sein d'un pas de temps, tel qu'implémenté dans la classe *AgentActivityRandomPolicy*

```
public List<Agent> schedule(List<Agent> list){
    Collections.shuffle(list, SimulationProperties.getInstance().getRandomizer());
    return list;
}
```

La seconde façon d'ordonnancer est définie dans la classe *AgentActivityStaticPolicy*. Elle consiste à ne pas modifier l'ordre dans lequel les entités agissent, et à retourner directement l'ensemble des entités

actives de la simulation. Cet ordonnancement est intéressant pour des simulations telles que le tri du couvain, où l'ordre dans lequel les entités agissent n'a pas d'influence sur les résultats obtenus.

La dernière façon d'ordonnancer que nous proposons est définie dans la classe *AgentActivityRandomOncePerStepPolicy*, et consiste à conserver un ordre rigoureusement identique à celui obtenu à l'aide de l'ordonnanceur *AgentActivityRandomPolicy*, en garantissant toutefois que seule une entité initiera une interaction au cours d'un pas de temps (voir algorithme 5.5). Il permet donc de réduire la granularité du temps de la simulation, tout en conservant la même façon d'ordonnancer l'activité des entités (voir figure 5.6). Cet ordonnanceur est défini afin de pallier à l'absence d'observateurs au niveau des individus : en s'assurant que seule une entité initie une interaction par pas de temps, l'observation au niveau supra-individuel (*i.e.* à la fin d'un pas de temps) est équivalent à l'observation au niveau des individus (sous réserve d'utiliser une mise à jour appropriée des entités, décrite dans la section qui suit).

FIGURE 5.5 – Ordonnancement aléatoire de l'activité des entités au sein d'un pas de temps, tel qu'implémenté dans la classe *AgentActivityRandomOncePerStepPolicy*. Le diagramme UML de la figure 5.3 résume la structure de la classe où cette méthode est déclarée.

```
public List<Agent> schedule(List<Agent> list){
    List<Agent> result = new LinkedList<Agent>();
    if(order.isEmpty()){
        // Début d'un nouveau pas de temps.
        // L'attribut order mémorise alors l'ensemble des sous-pas de temps qu'il faut exécuter
        list = AgentActivityRandomPolicy.INSTANCE.schedule(list);
        order.addAll(list);
    }
    if(! order.isEmpty()){
        // Ordonnancement du prochain sous-pas de temps
        result.add(order.remove(0));
    }
    return result;
}
```

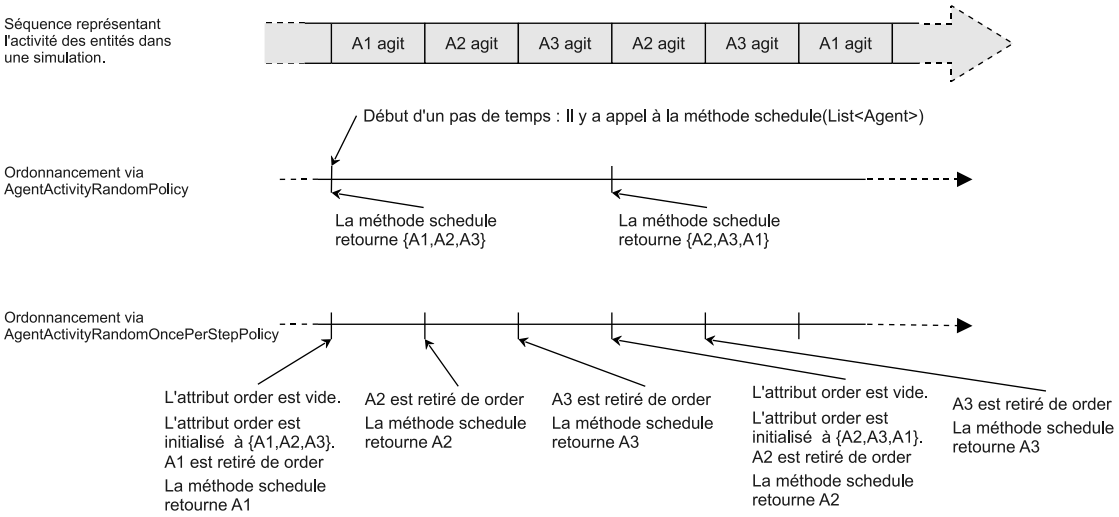


FIGURE 5.6 – Illustration de l'équivalence entre les ordonnanceurs de l'activité des entités, décrits par les classes *AgentActivityRandomPolicy* et *AgentActivityRandomOncePerStepPolicy*.

### 5.2.3 Gestion de la mise à jour des entités

Le changement de la granularité de la représentation du temps ne se fait pas uniquement au niveau de l'ordonnancement de l'activité des entités, mais aussi au niveau de la gestion de la mise à jour de l'état des entités. La gestion de la mise à jour consiste à déterminer quelles entités labiles doivent mettre à jour leur état au début d'un pas de temps.

Le dernier paramètre de JEDI permet de caractériser finement la construction du sous-ensemble caractérisant les entités étant mises à jour au début d'un pas de temps, en ayant pour seule donnée l'ensemble des entités labiles présentes dans l'environnement au début du pas de temps. Ce paramètre est représenté sous la forme d'un attribut intitulé *agentUpdatePolicy*, qui est présent dans la classe *SimulationProperties*, et qui implémente l'interface *AgentUpdatePolicy*. Cette interface définit une unique méthode intitulée *buildSubSet(Set<Agent>)*, qui est appelée dans le moteur de simulation au début de chaque pas de temps, en ayant pour paramètre l'ensemble des entités labiles présentes dans l'environnement. Cette méthode construit puis retourne une instance de l'interface *Set<Agent>*, qui représente le sous-ensemble des entités labiles qui est mis à jour au début du pas de temps courant. La valeur de l'attribut *agentUpdatePolicy* peut être redéfinie par la méthode statique intitulée *setAgentUpdatePolicy(AgentUpdatePolicy)* de la classe *SimulationProperties*.

La plateforme JEDI propose deux implémentations réutilisables de l'interface *AgentUpdatePolicy*, qu'il est possible de compléter par des implémentations spécifiques à un problème de simulation particulier.

La première façon d'extraire l'ensemble des entités mises à jour, utilisée par défaut, est définie dans la classe *DefaultUpdatePolicy*. Elle consiste à mettre à jour toutes les entités labiles au début de chaque pas de temps. Sa méthode *buildSubSet(Set<Agent>)* consiste donc à retourner l'ensemble des entités labiles de la simulation. Il s'agit de la politique de mise à jour décrite dans le modèle IODA.

La seconde façon d'extraire l'ensemble des entités mises à jour est définie dans la classe *AgentActivityRandomOncePerStepUpdatePolicy*. Elle est utilisée afin de garantir l'équivalence entre les ordonnanceurs de l'activité des entités des classes *AgentActivityRandomPolicy* et *AgentActivityRandomPerStepPolicy*. Pour cela, elle assure que les agents ne sont mis à jour que lors des pas de temps où l'attribut *order* de la politique d'ordonnancement *AgentActivityRandomOncePerStepUpdatePolicy* est réinitialisé. Une instance de la classe *AgentActivityRandomPerStepUpdatePolicy* mémorise donc dans un attribut nommé *activityPolicy* l'instance de la classe *AgentActivityRandomPerStepPolicy* à laquelle elle est associée. Sa méthode *buildSubSet(Set<Agent>)* consiste à retourner l'ensemble vide si la liste *order* de son attribut *activityPolicy* est non vide, et à retourner l'ensemble des entités labiles dans le cas contraire (voir l'algorithme 5.7).

FIGURE 5.7 – Méthode de la classe *AgentActivityRandomOncePerStepUpdatePolicy* assurant la mise à jour de toutes les entités labiles seulement lors de certains pas de temps. Cette politique de mise à jour n'est utilisée que conjointement à la politique d'ordonnancement de l'activité des entités de la classe *AgentActivityRandomPerStepPolicy*.

```
public Set<Agent> buildSubSet(Set<Agent> agents){
    if(this.activityPolicy.getOrder().isEmpty()){
        return agents;
    } else {
        return new HashSet<Agent>();
    }
}
```

## 5.3 Implémenter une simulation avec JEDI

L'implémentation d'expériences se fait avec JEDI en deux phases. La première consiste à construire un simulateur à l'aide des informations contenues dans le modèle. Lors de cette phase, chaque élément du modèle est transformé en un élément de l'implémentation, en utilisant des moyens tels que l'implémentation directe, la génération de code, ou la transformation de modèles. La seconde phase consiste à

spécifier l'expérience effectuée sur le simulateur ainsi créé. Elle comprend la spécification de l'initialisation de l'environnement de la simulation, l'identification du critère de terminaison de la simulation, ou encore l'identification des observateurs de la simulation.

Dans cette section, nous illustrons dans un premier temps comment utiliser les classes définies dans la section 5.1 afin de fournir une implémentation à un modèle. Cette implémentation n'est toutefois pas suffisante pour réaliser des expériences. En effet, en plus des interactions entre entités, du comportement des entités, et de l'environnement dans lequel les entités interagissent, une expérience est caractérisée par un état initial de la simulation, où des entités sont créées et ajoutées à l'environnement, mais aussi par la nature des résultats expérimentaux obtenus et la façon de les récupérer au fil de la simulation, ou encore par un critère d'arrêt déterminant quand stopper la simulation. Nous caractérisons ces deux points séparément dans un second temps dans cette section.

### 5.3.1 Implémentation d'un modèle avec JEDI

Les descriptions fournies dans la sous-sous section précédente ont pour but de décrire la structure utilisée pour implémenter un modèle construit avec l'approche IODA. Dans cette sous-sous section, nous spécifions comment implémenter un modèle en utilisant cette structure.

Puisque la plateforme JEDI fournit déjà l'implémentation de l'environnement et ses primitives, l'implémentation d'une simulation consiste à spécifier les interactions, spécifier la matrice d'interaction, spécifier la matrice de mise à jour, et enfin spécifier les primitives abstraites et le halo des entités. Nous profitons de cette section pour illustrer comment écrire une simulation simple de type « HelloWorld ».

**Interactions** L'implémentation d'une interaction individuelle $\mathcal{I}$ revient dans un premier temps à créer deux interfaces étendant l'interface *EntitySignature*, dont le nom est défini par la concaténation de la chaîne "SourceDe" et l'identifiant $id(\mathcal{I})$ de l'interaction, ou par la concaténation de la chaîne "CibleDe" et l'identifiant $id(\mathcal{I})$ de l'interaction. Ces interfaces représentent la signature des entités dans l'interaction créée. La seconde étape de cette implémentation consiste à créer une classe dont le nom correspond à l'identifiant $id(\mathcal{I})$ de l'interaction. Cette classe doit étendre la classe abstraite *SingleTargetInteraction*, ainsi qu'attribuer la concaténation de "SourceDe" et l'identifiant $id(\mathcal{I})$ comme valeur au type générique *Source*, et attribuer la concaténation de "CibleDe" et l'identifiant $id(\mathcal{I})$ comme valeur au type générique *Target*. La troisième étape de l'implémentation d'une interaction consiste à implémenter la spécification des préconditions, du déclencheur, et des actions de l'interactions, d'y identifier les primitives abstraites introduites, puis de reporter ces primitives dans les deux interfaces créées précédemment.

L'implémentation d'une interaction dégénérée est similaire, à cela près qu'une seule une interface, dont le nom est la concaténation de "SourceDe" et l'identifiant $id(\mathcal{I})$, est nécessaire à son implémentation.

Nous illustrons l'implémentation de ces deux types d'interactions, au travers de deux exemples.

***Exemple. Interaction dégénérée* VIEILLIR**
*La figure 5.8 décrit l'implémentation des classes et interfaces nécessaires à la définition d'une interaction dégénérée dont l'identifiant est* VIEILLIR. *Dans cette interaction, qui n'est soumise à aucunes conditions, une entité source voit son age augmenter.*

***Exemple. Interaction individuelle* SALUER**
*La figure 5.9 décrit l'implémentation des classes et interfaces nécessaires à la définition d'une interaction individuelle dont l'identifiant est* SALUER. *Dans cette interaction, qui n'est soumise à aucunes conditions, une entité source envoie un message de salutation à l'entité cible.*

**Remarque.** Si une signature n'introduit que des primitives abstraites figurant déjà dans l'interface *EntitySignature*, alors il n'est pas nécessaire d'introduire de nouvelle interface pour cette signature. Dans ce cas, le type générique lui étant associé (*i.e. Source* ou *Target*) vaut *EntitySignature*.

**Matrice d'interaction et Matrice de mise à jour** L'implémentation d'une matrice d'interaction ou d'une matrice de mise à jour ne nécessite pas la définition de nouvelles classes ou interfaces. Elle ne requiert que l'instanciation des classes *InteractionMatrix* et *UpdateMatrix*.

FIGURE 5.8 – Implémentation d'une interaction dégénérée VIEILLIR avec la plateforme JEDI.

```
public interface SourceDeVieillir extends EntitySignature {
   // Augmente l'age d'une entité d'une valeur t
   public void augmenterAge(int t);
}
```

```
public class Vieillir extends DegenerateInteraction<SourceDeVieillir>{
   public boolean declencheur(EnvironmentSignature environment,
                              SourceDeVieillir source){
      return true;
   }
   public boolean preconditions(EnvironmentSignature environment,
                                SourceDeVieillir source){
      return true;
   }
   public void actions(EnvironmentSignature environment,
                       SourceDeVieillir source){
      source.augmenterAge(1);
   }
}
```

FIGURE 5.9 – Implémentation d'une interaction individuelle SALUER avec la plateforme JEDI.

```
public interface SourceDeSaluer extends EntitySignature {
   // Retourne un message de salutation
   public String messageDeSalutation();
}
```

```
public interface CibleDeSaluer extends EntitySignature {
   // Permet de recevoir un message envoyé par une autre entité
   public void recevoirMessage(String message);
}
```

```
public class Saluer extends SingleTargetInteraction<SourceDeSaluer,CibleDeSaluer>{
   public boolean declencheur(EnvironmentSignature environment, SourceDeSaluer source,
                              CibleDeSaluer target){
      return true;
   }
   public boolean preconditions(EnvironmentSignature environment, SourceDeSaluer source,
                                CibleDeSaluer target){
      return true;
   }
   public void actions(EnvironmentSignature environment, SourceDeSaluer source,
                       CibleDeSaluer target){
      String message = source.messageDeSalutation();
      target.recevoirMessage(message);
   }
}
```

**Famille d'entités** L'implémentation d'une famille d'entités $\mathcal{F}$ dans JEDI revient dans un premier temps à créer une classe qui étend la classe abstraite *Agent*, de lui donner pour nom l'identifiant $id(\mathcal{F})$ du modèle de la famille d'entités, et de lui associer un constructeur vide. Dans un second temps, cette classe est complétée en lui faisant étendre toutes les interfaces représentant la signature de cette famille d'entités dans les interactions qu'elle est capable d'initier ou de subir. Dans un dernier temps, cette classe est complétée par l'implémentation des diverses primitives abstraites présentés dans les interfaces qu'elle étend, et par l'ajout des attributs manipulés dans ces primitives.

***Exemple. Famille d'entités*** `ExempleDeFamille`
*En supposant qu'une famille d'entités dont l'identifiant est* `ExempleDeFamille` *est capable d'initier les interactions* VIEILLIR *et* SALUER*, et qu'elle est capable de subir l'interaction* SALUER*, alors son implémentation dans* JEDI *correspond au code présent dans la figure 5.10.*

FIGURE 5.10 – Implémentation d'une famille d'entités `ExempleDeFamille` avec la plateforme JEDI. Cette famille peut initier et subir l'interaction SALUER, et initier l'interaction VIEILLIR.

```
public class ExempleDeFamille extends Agent implements SourceDeSaluer,CibleDeSaluer,
                                                        SourceDeVieillir {
    // Attributs de la famille d'entités
    protected int age;
    // Constructeur vide de la famille d'entités
    public ExempleDeFamille(){
        super();
        age = 0;
    }
    // Primitives abstraites provenant de la signature SourceDeSaluer
    public String messageDeSalutation(){
        return "Bonjour, j'ai " + this.age + " ans. Et vous ?";
    }
    // Primitives abstraites provenant de la signature CibleDeSaluer
    public void recevoirMessage(String message){
        System.out.println("J'ai reçu le message : \"" + message + "\"");
    }
    // Primitives abstraites provenant de la signature SourceDeVieillir
    public void augmenterAge(int t){
        this.age = this.age + t;
    }
}
```

**Instanciation d'entités** Les classes décrites dans la section 5.3.1 suffisent à implémenter un modèle. Toutefois, l'instanciation d'une famille d'entités requiert la définition de la ligne de la matrice d'interaction et de la cellule de la matrice d'interaction lui étant associées. Afin de faciliter cette instanciation, une classe particulière est mise en place pour être utilisée comme fabrique abstraite d'entités. Cette fabrique permet de centraliser la déclaration des matrices d'interaction et de mise à jour, ainsi que la déclaration des halo, tout en permettant de créer des instances de familles d'entités simplement, à l'aide d'une méthode appelée *createEntity(Class<T>, double,double)*, ou d'une méthode appelée *createEntity(Class<T>)*.

La fabrique mentionnée ici s'implémente sous la forme d'une classe étendant la classe abstraite *DefaultEntityFactory*, dont il faut spécifier une unique méthode abstraite intitulée *initializeFactory()*. Cette méthode est utilisée pour spécifier la matrice d'interaction, la matrice de mise à jour, et déterminer quelle est la surface de perception référente associée à chaque famille d'entités, à l'aide des méthodes dont le nom est *addAssignationElementToInteractionMatrix* (pour initialiser la matrice d'interaction), *addAssignationElementToUpdateMatrix* (pour initialiser la matrice de mise à jour) ou *setReferentSurfaceFor* (pour définir la surface de perception référente d'une entité).

***Exemple. Fabrique de la simulation HelloWorld***
*Si nous poursuivons notre exemple reposant sur les interactions* VIEILLIR *et* SALUER*, et reposant sur*

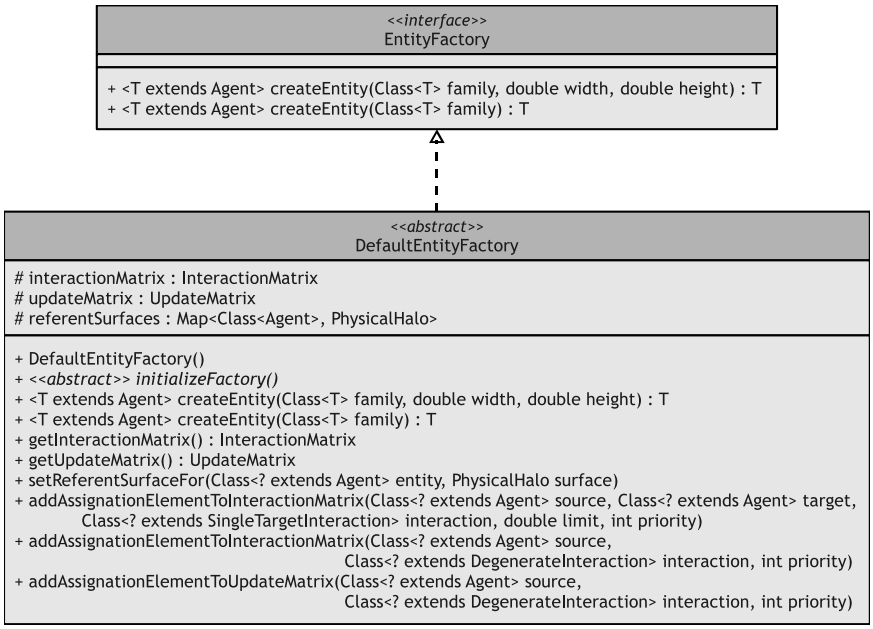


FIGURE 5.11 – Troisième partie du diagramme UML décrivant les classes et interfaces de la plateforme JEDI directement manipulées pour implémenter un modèle décrit avec la méthodologie IODA.

*la famille d'entités* `ExempleDeFamille`*, alors la fabrique permettant de créer des instances de la famille d'entités* `ExempleDeFamille` *est implémentée selon le code décrit dans la figure 5.12. Si* `fabric` *est une instance de cette classe nouvellement créée, alors une entité de la famille* `ExempleDeFamille` *est instanciée par exemple par l'appel de méthode :* `fabric.createEntity(ExempleDeFamille.class, 1, 1)`*.*

**Remarque.** Cette fabrique ne fait que créer une instance d'une famille d'entités dont la ligne de la matrice d'interaction, la cellule de la matrice de mise à jour, et la surface de perception référente sont initialisées correctement. L'initialisation à proprement parler d'une entité, dans le cadre d'une expérience particulière, n'est pas encore effectuée ici. Leur initialisation doit être faite lors de l'exécution d'une interaction, à l'aide d'une primitive abstraite, dans le constructeur de l'entité, ou lors de l'initialisation de la simulation.

### 5.3.2 L'observation des résultats d'une simulation

Un simulateur est un programme informatique dont l'objectif est de valider ou invalider un modèle, à l'aide d'expériences. Il constitue une implémentation plus ou moins fidèle du modèle et peut donc, dans certains cas, aboutir à des expérimentations fournissant des résultats erronés. Il se révèle alors primordial d'identifier l'origine des erreurs, et de les corriger. Cette vérification se fait au moyen de l'étude de la trace de simulations, *i.e.* l'observation pas à pas de l'exécution d'expériences conduites avec ce simulateur.

Un simulateur est aussi un moyen utilisé pour valider un modèle, *i.e.* pour s'assurer que si le modèle reproduit de manière satisfaisante le phénomène étudié. Il permet ainsi de déterminer si les hypothèses sur lesquelles se fonde le modèle fournissent une explication candidate à l'apparition du phénomène, ou si au contraire elles ne permettent pas d'expliquer l'apparition du phénomène étudié sous leur forme actuelle. La validation est faite au moyen de la comparaison de données extraites du phénomène, et de données extraites de différentes expériences menées sur un simulateur, et se fonde donc sur l'observation de l'exécution d'une simulation.

Dans les deux cas mentionnés ci-dessus, il est fondamental de pouvoir observer l'exécution de la simulation à plusieurs niveaux, et d'en retirer un maximum d'informations, afin de les interpréter. L'ob-

FIGURE 5.12 – Implémentation de la fabrique permettant de créer des instances de la famille d'entités `ExempleDeFamille` avec la plateforme JEDI. Cette famille peut initier et subir l'interaction SALUER, et initier l'interaction VIEILLIR.

```
public class ExempleDeFabrique extends DefaultEntityFactory {
    // Constructeur de la fabrique (optionnel)
    public ExempleDeFabrique(){
        super();
    }

    public void initializeFactory(){
        // Initialisation de la matrice d'interaction
        // Une entité peut en saluer une autre à une distance de 3.5, avec une priorité de 1
        this.addAssignationElementToInteractionMatrix(ExempleDeFamille.class,
                                        ExempleDeFamille.class,Saluer.class, 3.5, 1);
        // Initialisation de la matrice de mise à jour
        this.addAssignationElementToInteractionMatrix(ExempleDeFamille.class,Vieillir.class,1);
        // Définition de la surface de perception référente de la famille ExempleDeFamille
        // La perception se fait dans un cone de 90° de longueur 3.
        this.setReferentPerceptionSurface(ExempleDeFamille.class,
                                        HaloBuilder.createConeShapedArea(3, Math.PI / 2));
    }
}
```

servation de ces informations peut être effectué à deux niveaux selon [PMR+05] : l'« observation à l'échelle de l'individu », qui consiste à « [observer et éventuellement éditer] les informations spécifiques à un agent (valeurs des attributs, messages,courbes d'évolution des attributs) », et l'« observation à l'échelle supra-individuelle », qui consiste à « observer les interactions entre agents communiquant et négociant par échange de messages ». Dans le cas de la méthodologie IODA, l'observation prend un sens plus précis, permettant ainsi de faciliter l'interprétation des données observées : ces informations peuvent être la valeur des attributs d'une entité, le voisinage de l'entité lors de ce pas de temps, les interactions auxquelles l'entité a participé (et non plus seulement les messages échangés), le nombre d'entités dans l'environnement, ou toute valeur calculée à partir de ces informations (par exemple la densité en entités d'une zone de l'environnement, le nombre moyen de naissances et décès, l'age moyen des entités, *etc.*). Nous décrivons ci-après les moyens mis en place dans la plateforme JEDI afin de prendre en compte ce problème.

### L'observation : un problème complexe

La construction d'observateurs de la simulation est une tache s'avérant complexe, pour deux raisons. D'une part un observateur doit être non-invasif dans le simulateur, *i.e.* l'ajout d'un observateur ne doit pas influer sur l'implémentation du modèle. La résolution de ce problème passe par la mise en place d'une logicielle évoluée au sein du simulateur, utilisant par exemple des patrons de conception tels que le Modèle-Vue-Contrôleur ou Observateur/Observé, ou encore une structure logicielle utilisant la programmation par aspects, *etc.* Des travaux tels que ceux de Ralambondrainy *et al.* [RCP06, PMR+05] vont en ce sens, et proposent une ontologie pouvant être utilisée pour fournir une structure logicielle générique et réutilisable d'observateurs, pouvant autant observer au niveau de l'individu qu'au niveau supra-individuel. D'autre part, un observateur doit observer une simulation en un temps raisonnable, sans quoi le nombre d'expériences qu'il est possible de conduire avec le simulateur peut être réduit significativement, en particulier lorsque les observations portent sur le niveau de l'individu. En effet, l'observation de toutes les entités est extrêmement coûteux dès que le nombre d'entités dans une simulation augmente. Des travaux tels que ceux de Morvan *et al.* [MVDJ09] s'intéressent à ce problème, et proposent de réduire le nombre de calculs nécessaires à l'observation au niveau des individus. La réduction des calculs s'y fait en mesurant de manière approchée les résultats d'une observation, en se basant sur un filtrage statistique des entités observées.

Ces deux problèmes sortent du cadre de l'étude menée dans cette thèse, et nous n'avons pas ici

l'ambition de les résoudre. L'observation d'une simulation reste toutefois un problème fondamental à la constitution d'un simulateur, qu'il est nécessaire de prendre en compte. Nous présentons donc de manière minimale comment l'observation est prise en compte dans la plateforme JEDI, en limitant nos descriptions à la structure logicielle utilisée, ainsi que les différents observateurs spécifiés par défaut dans cette plateforme.

### Structure des observateurs dans JEDI

JEDI offre la possibilité d'inspecter le contenu de l'environnement d'une simulation au début d'une expérimentation, à la fin d'une expérimentation, ainsi qu'à la fin de chaque pas de temps. Les données observées incluent la valeur des attributs des entités, les interactions auxquelles ils ont participé (via les tuple réalisable dont l'interaction a été initiée au cours du pas de temps), ou plus généralement l'environnement d'une simulation.

L'observation repose sur le patron de conception *Modèle-Vue-Contrôleur*, où un observateur est une vue sur la simulation exécutée. Une simulation est exécutée dans un processus instance de la classe *SimulationThread*. Ce processus se fige dans un état stable, propice à l'observation, au début de la simulation, à la fin de la simulation, ainsi qu'à la fin de chaque pas de temps. Un contrôleur instancié par la classe *SimulationCore* est notifié de ces situation, et déclenche alors en réponse la mise à jour de chaque observateur. La simulation ne se poursuit qu'une fois la mise à jour de tous les observateurs terminée (voir figure 5.13).

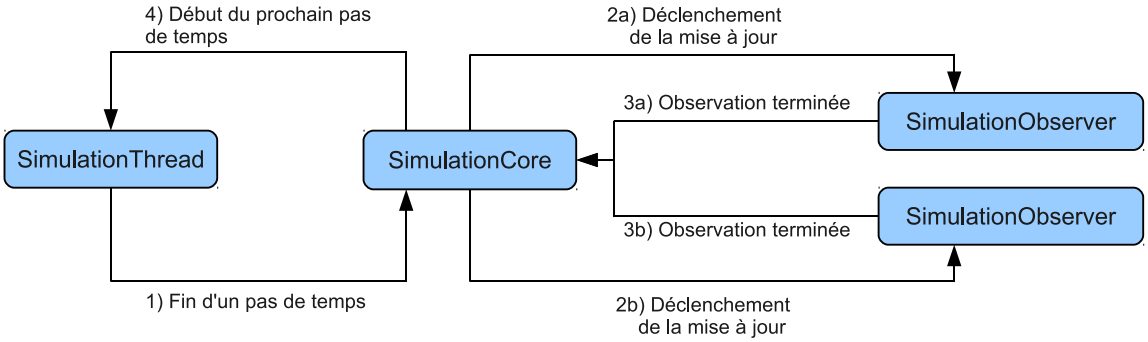


FIGURE 5.13 – Patron de conception Modèle-Vue-Contrôleur mis en place dans la plateforme JEDI afin d'observer des simulations. La classe « SimulationThread » correspond au processus où sont exécutés les pas de temps d'une simulation, la classe « SimulationCore » sert de pivot entre le modèle de la simulation et les observateurs, et contrôle la mise à jour des observateurs. La classe « SimulationObserver » permet de définir des observateurs dans JEDI.

La mise à jour d'un observateur consiste à extraire des informations de l'environnement ou des entités qui y résident, et à les mettre à la disposition des expérimentateurs, que cela soit sous la forme d'une interface graphique, ou sous la forme de fichier de données. Nous en fournissons deux exemples ci-après.

***Exemple. Grille en deux dimensions***
*L'observation d'un environnement euclidien en deux dimensions peut être faite par l'affichage d'une grille en deux dimensions, où les entités sont représentées par des formes géométriques placées dans la grille en fonction de leur position. Dans ce cas, la mise à jour de l'observateur consiste à construire une image représentant une telle grille pour l'état courant de l'environnement, puis à l'afficher.*

***Exemple. Fichier de données***
*L'observation d'une simulation en éthologie peut être faite en mesurant l'évolution des populations de chaque espèce animale contenue dans l'environnement. Si une espèce animale est représentée par une famille d'entités, alors la mise à jour de cet observateur consiste à recenser le nombre d'entités de chaque famille d'entités de la simulation, et à écrire ces valeurs dans un fichier de données.*

**Observateurs fournis par défaut**

La plateforme JEDI permet de greffer un nombre arbitraire d'observateurs à un simulateur, et offre une implémentation par défaut de quatre observateurs valides pour tout type de simulation ayant lieu dans un environnement euclidien continu en deux dimensions :

- Une interface graphique affichant l'environnement sous la forme d'une grille en deux dimensions, où une entité est représentée avec une forme géométrique colorée, ou avec un image ;
- Une interface graphique affichant la population actuelle de chaque famille d'entités ;
- Une interface graphique affichant l'historique des interactions auxquelles chaque entité a participé au cours de chaque pas de temps, ainsi que les pas de temps lors desquels une entité a été ajoutée ou retirée de l'environnement ;
- Un observateur mémorisant sous la forme d'un fichier des informations recensées à la fin de chaque pas de temps.

L'instanciation de ces quatre observateurs est décrite ultérieurement, dans la section 5.3.3.

Les valeurs observées lors d'une simulation, et la façon de les observer peut grandement varier d'une simulation à l'autre. Par conséquent, JEDI offre la possibilité de construire des observateurs spécifiques à une simulation, en créant une classe étendant la classe abstraite *SimulationObserver*. La spécification de l'observateur se fait par l'implémentation des trois méthodes *timeStepEnded(Environment,int)*, *simulationInitializationEnded(Environment)*, et *simulationEnded(Environment)*, qui décrivent la mise à jour de l'observateur respectivement lorsqu'un pas de temps se termine, lorsque l'initialisation de la simulation est terminée, et lorsque la simulation est terminée.

Dans JEDI, nous ne définissons pas de structure permettant de construire des observateurs au niveau des individus, *i.e.* au moment où chaque entité initie une interaction, qui se révèlent extrêmement coûteux en termes de performances à mettre en place. Leur définition reste toutefois possible en imposant qu'une seule entité puisse agir par pas de temps. Avec une gestion adaptée de l'ordonnanceur de l'activité et de la mise à jour des entités au cours des pas de temps, il est même possible d'obtenir des résultats strictement équivalent de ceux obtenus avec l'ordonnanceur « classique » décrit dans l'algorithme 6 page 125, sous réserve qu'aucune interaction, et qu'aucune primitive abstraite d'une entité ne manipule la valeur courante du pas de temps (voir section 5.2.2).

La figure 5.14 représente le diagramme UML des classes présentées ici.

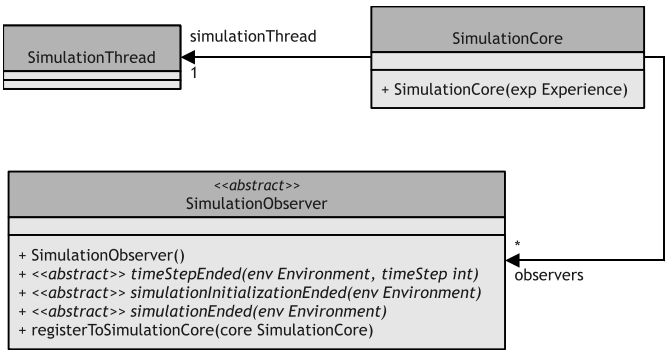


FIGURE 5.14 – Diagramme UML des classes permettant de définir des observateurs dans JEDI.

### 5.3.3 Construction d'une expérience

Une fois le modèle implémenté, il est possible de construire une expérience reposant sur le simulateur ainsi défini. Cette part n'est pas propre à la méthodologie IODA, et n'a donc pas été décrite dans notre approche formelle. Pour des raisons d'implémentation, nous proposons des classes permettant de créer une expériences. Il s'agit là d'un choix ayant permis de restreindre l'étude menée dans cette thèse.

A terme, la phase de spécification d'une expérience devrait disposer de son propre modèle, pouvant être spécifié et implémenté avec une approche transversale. En effet, comme le mentionnent Shannon *et*

*al.* ainsi que Grimm *et al.*[Sha98, GBB+06], un modèle validé ne prend son sens que si les résultats des simulations sont communiqués à la communauté, et vérifiables par cette communauté. Cela implique la description précise des expériences ayant permis d'obtenir les résultats communiqués (d'où la nécessité d'un modèle), mais aussi la possibilité de reproduire ces expériences (d'où la nécessité d'un processus garantissant une transition valide entre modèle et implémentation, et donc une approche transversale). Certains travaux, tels que [GBB+06, DDG09] vont en ce sens, et étudient comment modéliser l'initialisation de l'environnement d'une simulation. Ces spécifications sont toutefois en l'état insuffisantes, car peu formelles (par exemple le protocole ODD [GBB+06] ne fournit qu'une description verbale de ce qui doit être contenu dans un modèle), ou très proche d'un langage informatique (par exemple XELOC [DDG09], qui est une extension de XML nécessitant de définir ses propres balises afin de décrire une simulation). De plus, ces modèles ne sont actuellement pas intégrés à une approche de conception transversale. Il s'agit là d'une perspective que nous ne décrirons pas plus dans ce chapitre.

Dans JEDI, pour qu'une expérience puisse être exécutée, on doit avoir la connaissance :

- des dimensions de l'environnement, ainsi que de sa nature torique ou non selon l'axe des abscisses et l'axe des ordonnées ;
- de comment initialiser l'environnement de la simulation, ce qui implique de savoir quelles entités initialement instancier, comment initialiser leurs divers attributs, et en particulier leur surface dans l'environnement, et enfin savoir à quelle position les placer dans l'environnement ;
- des conditions sous lesquelles la simulation doit se terminer ;
- du moyen utilisé pour initialiser et démarrer une simulation ;
- de comment les résultats de la simulation sont observés, et donc de définir les observateurs de la simulation (voir section 5.3.2) ;
- de comment initialiser les divers paramètres du moteur de simulation (voir section 5.2).

Dans JEDI, une unique classe abstraite intitulée *Experiment* est étendue pour réaliser toutes ces tâches. Le diagramme UML spécifiant cette classe est représenté sur la figure 5.15.

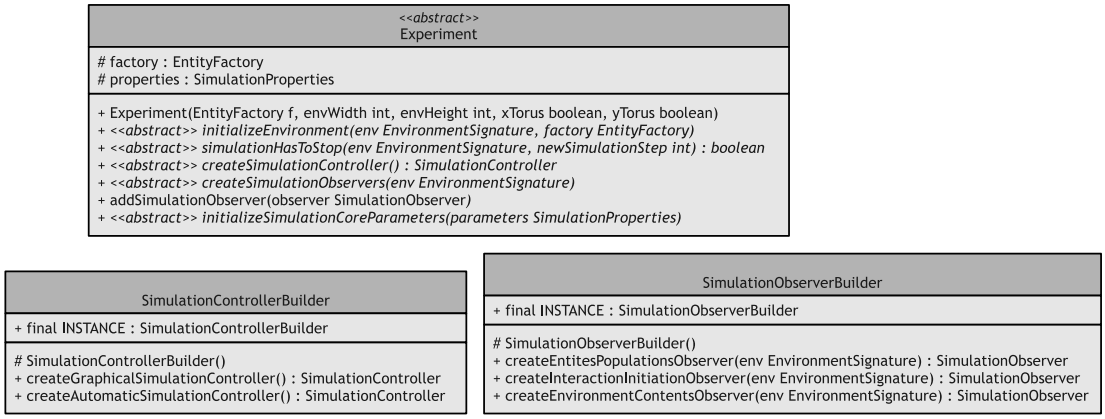


FIGURE 5.15 – Diagramme UML décrivant les classes et interfaces de la plateforme JEDI permettant de créer des expériences reposant sur l'implémentation d'un modèle IODA.

**Création d'une expérience** La création d'une expérience passe par la création d'une classe qui étend la classe abstraite *Experiment*. Cette classe doit disposer d'un constructeur dont l'unique paramètre est une instance de la classe *EntityFactory*, dont l'implémentation consiste à appeler le constructeur *Experiment(EntityFactory,double,double,boolean,boolean)* de la classe *Experiment*, en lui fournissant en paramètres la description de l'environnement utilisé pour cette expérience. Une telle description comprend la spécification de la largeur et de la hauteur de l'environnement, ainsi que la nature torique ou non de l'environnement selon l'axe des abscisses et l'axe des ordonnées.

***Exemple. Création d'une expérience pour le modèle « HelloWorld »***

*Les lignes 1 à 5 de la figure 5.16 illustrent l'implémentation de la création d'une expérience, et de la spécification des dimensions de l'environnement avec* JEDI.

FIGURE 5.16 – Implémentation avec la plateforme JEDI d'une expérience utilisant le modèle « HelloWorld » décrit au long de la section 5.3.1.

```
01:  public class ExperienceAvecHelloWorld extends Experiment {
02:    public ExperienceAvecHelloWorld(){
03:      // L'expérience a lieu dans un environnement 10 x 10 non torique.
04:      super(10,10,false,false);
05:    }
06:    public void initializeEnvironment(EnvironmentSignature env, EntityFactory factory){
07:      // Création d'une entité dont la surface dans l'environnement est de 1 x 1
08:      ExempleDeFamille entite = factory.createEntity(ExempleDeFamille.class, 1, 1);
09:      // Placement de l'entité dans l'environnement, à la position (1,1)
10:      env.add(entite,1,1);
11:      // Création d'une autre entité dont la surface dans l'environnement est de 1 x 1
12:      entite = factory.createEntity(ExempleDeFamille.class, 1, 1);
13:      // Placement de l'autre entité dans l'environnement, à la position (1,2)
14:      env.add(entite,1.5,2);
15:    }
16:    public boolean simulationHasToStop(EnvironmentSignature env, int newSimulationStep){
17:      // La simulation se termine à la fin du premier pas de temps.
18:      return newSimulationStep == 2;
19:    }
20:    public SimulationController createSimulationController(){
21:      // L'initialisation de la simulation, et son démarrage sont gérés par des boutons.
22:      return SimulationControllerBuilder.INSTANCE.createGraphicalSimulationController();
23:    }
24:    public void createSimulationObservers(){
25:      // Dans cette simulation, nous observons les interactions étant déclenchées.
26:      SimulationObserver obs = SimulationObserverBuilder.INSTANCE
           .createInteractionInitiationObserver();
27:      this.addSimulationObserver(obs);
28:    }
29:    public void initializeSimulationCoreParameters(SimulationProperties parameters){
30:      // Dans cette simulation, nous utilisons "2000" comme graine dans le générateur
31:      // de nombres aléatoires.
32:      parameters.setRandomizerSeed(2000);
33:      // Nous utilisons la politique d'ordonnancement de l'activité des entités par
34:      défaut.
35:      // Nous utilisons la politique de mise à jour des entités par défaut.
36:    }
37:  }
```

**Initialisation de l'environnement** Le contenu de l'environnement est initialisé par la méthode abstraite *initializeEnvironment(EnvironmentSignature,EntitesFactory)* de la classe *Experiment*, qui prend en paramètres l'environnement initialisé, ainsi que la fabrique abstraite permettant de créer des instances de familles d'entités. L'implémentation de cette méthode consiste à créer des instances de familles d'entités à l'aide de la fabrique abstraite d'entités, d'initialiser les entités ainsi créées, puis de les ajouter à l'environnement.

***Exemple. Initialisation de l'environnement pour le modèle « HelloWorld »***
*Les lignes 6 à 15 de la figure 5.16 illustrent l'implémentation de l'initialisation de l'environnement avec* JEDI.

**Critère d'arrêt d'une expérience** Le critère déterminant quand stopper l'algorithme de simulation est défini dans la classe *Experiment* par une méthode abstraite appelée *simulationHasToStop(EnvironmentSignature,int)*, qui prend en paramètres deux valeurs, et retourne un booléen. Le premier paramètre est l'environnement dans lequel a lieu la simulation. Le second paramètre est le numéro du pas de temps sur le point de commencer. Si jamais cette méthode retourne « faux », le pas de temps sur le point de commencer est passé, et la simulation s'arrête. Les implémentation de ce critère sont très diversifiées, et peuvent par exemple consister à vérifier qu'un pas de temps maximal est atteint, ou encore que le nombre d'entités dans la simulation n'est pas nul, *etc.*

***Exemple. Critère d'arrêt d'une expérience pour le modèle « HelloWorld »***
*Les lignes 16 à 19 de la figure 5.16 illustrent l'implémentation du critère d'arrêt d'une simulation avec* JEDI.

**Mode de contrôle de la simulation** Selon les simulations effectuées, le contrôle du démarrage ou de l'initialisation d'une expérience peut se faire de manières différentes. Les deux plus courantes consistent soit à fournir une interface graphique, où des boutons permettent d'initialiser, de démarrer, ou encore de mettre en pause une simulation, soit à initialiser automatiquement une simulation, puis démarrer automatiquement la simulation, afin d'effectuer du traitement intensif de données.

Dans JEDI, la façon de contrôler une simulation est définie par une instance de la classe abstraite *SimulationController*, dont ne fournissons pas le détail dans ce chapitre. Deux implémentations par défaut de cette classe sont fournies, et donnent accès aux deux façons de contrôler une simulation présentées ci-avant. Une fabrique abstraite appelée *SimulationControllerBuilder* a été créée dans le but d'instancier simplement ces classes, à l'aide des deux méthodes *createGraphicalSimulationController()* et *createAutomaticSimulationController()*.

Le contrôleur utilisé dans une expérience est défini par la méthode abstraite *createSimulationController()*. Son implémentation consiste à appeler l'une des deux méthodes de la fabrique abstraite *SimulationControllerBuilder*, ou d'instancier directement un autre type de contrôleur.

***Exemple. Mode de contrôle de la simulation pour le modèle « HelloWorld »***
*Les lignes 20 à 23 de la figure 5.16 illustrent l'implémentation du mode de contrôle de la simulation avec* JEDI.

**Observation de la simulation** L'observation d'une simulation se fait dans JEDI à l'aide d'instances de la classe abstraite *SimulationObserver*, dont le principe de fonctionnement est exposé dans la section 5.3.2. La spécification des observateurs associés à une expérience se fait dans la méthode abstraite *createSimulationObservers(EnvironmentSignature)* de la classe *Experiment*. Son implémentation consiste à instancier les différents observateurs de l'expérience, et de les mémoriser en appelant la méthode *addSimulationObserver(SimulationObserver)* de la classe *Experiment*.

L'instanciation des observateurs définis par défaut dans JEDI se fait via une fabrique abstraite intitulée *SimulationObserverBuilder*, qui définit trois méthodes. La première méthode, appelée *createEntitiesPopulationsObserver(EnvironmentSignature)*, permet d'instancier un observateur affichant dans une interface graphique le nombre courant d'instances de chaque famille d'entités présentes dans l'environnement. La deuxième, appelée *createInteractionInitiationObserver(EnvironmentSignature)*, permet d'instancier un observateur affichant dans une interface graphique l'historique des interactions initiées et subies par chaque entité de l'environnement. La dernière méthode, appelée *createEnvironmentContentsObserver(EnvironmentSignature)*, permet d'instancier un observateur affichant dans une interface graphique le contenu de l'environnement, sous la forme d'une grille où les entités sont représentées par des formes géométriques colorées, ou par des images. Nous ne décrivons pas comment créer le dernier type d'observateur mentionné dans la section 5.3.2, qui consiste à mémoriser dans un fichier des données lues à chaque pas de temps.

***Exemple. Observation de la simulation pour le modèle « HelloWorld »***
*Les lignes 24 à 28 de la figure 5.16 illustrent l'implémentation de la création des observateurs de la simulation avec* JEDI.

**Initialisation des paramètres du moteur de simulation** La façon de paramétrer le moteur de simulation est déjà décrite dans la section 5.2. Cette étape de la construction d'une expérience consiste à mettre en pratique cette spécification dans la méthode abstraite *initializeSimulationCoreParameters(SimulationProperties)* de la classe *Experiment*.

***Exemple. Paramètres du moteur de simulation pour le modèle « HelloWorld »***
*Les lignes 29 à 36 de la figure 5.16 illustrent l'implémentation de l'initialisation des paramètres du moteur de simulation avec* JEDI.

**Expérimentation** Une fois la classe spécifiant une expérience définie, il est possible de créer la classe qui exécutera la simulation. Cette classe dispose d'une unique méthode statique intitulée *main(String[])*, qui consiste à exécuter l'expérience. Son implémentation consiste à instancier une fabrique d'entités telle que spécifiée dans la section 5.3.1, puis à instancier une expérience, dont la spécification est décrite dans la section 5.3.3, et enfin à créer une instance de la classe *SimulationCore* à l'aide de son constructeur *SimulationCore(Experiment)*, et à appeler la méthode *start* de cette instance.

***Exemple. Classe principale de l'expérience utilisant le modèle « HelloWorld »***
*La figure 5.17 illustre l'implémentation de la classe principale utilisant le modèle « HelloWorld » décrit dans cette section, pour une expérience intitulée* ExperienceAvecHelloWorld.

FIGURE 5.17 – Classe principale permettant d'exécuter l'expérience décrite dans la section 5.3.3, qui repose sur le modèle décrit dans la section 5.3.1

```
public class HelloWorldMain {
   public static void main(String[] args){
     EntityFactory fabrique = new ExempleDeFabrique();
     Experiment experience = new ExperienceAvecHelloWorld(fabrique);
     SimulationCore moteurDeSimulation = new SimulationCore(experience);
     moteurDeSimulation.start();
   }
}
```

## 5.4 L'environnement de développement intégré JEDI-Builder

Comme nous venons de le voir, écrire une simulation avec JEDI nécessite la création d'un grand nombre de classes pour implémenter fidèlement chaque concept de IODA. L'implémentation de tels modèle est facilitée s'il est possible de transformer automatiquement un modèle IODA en une implémentation.

Les principes énoncés textuellement dans la section 4.1 page 97 du chapitre 4 peuvent être automatisés afin que la construction d'un modèle IODA ne se fasse que de manière graphique, dans un environnement de développement intégré (IDE). Il devient alors possible de tirer parti de la description informatisée du modèle, et des algorithmes de simulation afin de générer automatiquement un simulateur dans un langage de programmation cible.

Afin d'éprouver ces principes nous avons développé un IDE appelé JEDI-BUILDER, qui permet de construire un modèle IODA, et d'en générer automatiquement l'implémentation pour la plateforme de simulation JEDI. La seconde vocation de ce prototype a été d'expérimenter comment adapter le concept d'héritage au modèle de IODA et à JEDI. Nous en décrivons le résultat dans le chapitre 8 page 221. Cet IDE a été implémenté en JAVA, et est composé de 239 classes et interfaces, cumulant en tout 21440 lignes de code (commentaires compris), équivalentes à 12770 lignes de code logique [18].

18. comptabilisées selon la métrique SLOC de la commande unix `sloccount`

**Étapes de la méthodologie couverte par JEDI-BUILDER** JEDI-BUILDER constitue la première étape d'une preuve de la possibilité d'implémenter la méthodologie IODA et de générer à partir des description du modèle une implémentation sur une plateforme reposant sur les principes de IODA. Nous avons pour cela émis deux hypothèses simplificatrices, que nous résumons sur la figure 5.18.

La première a consisté à restreindre à JEDI les plateformes pour lesquelles JEDI-BUILDER permettait de générer du code. Puisque dans JEDI, l'environnement est fixé et ne peut être modifié, les étapes de la méthodologie IODA permettant de définir la sémantique de l'environnement ne sont pas intégrées à JEDI-BUILDER (voir rectangle vert dans la figure 5.18). À l'avenir, cet IDE devra être étendu à la génération de code pour d'autres plateformes, afin de confirmer que les algorithmes décrits dans le chapitre précédent permettent non seulement de conserver la structure du modèle lors de l'implémentation, mais de plus que ces algorithmes sont valides pour tout type de langage de programmation.

La seconde a consisté à restreindre l'ensemble des étapes de la méthodologie couvertes par l'édition graphique dans JEDI-BUILDER aux étapes ne nécessitant pas de descriptions algorithmiques. Cette hypothèse simplificatrice est émise afin de maximiser le rapport entre effort d'implémentation de JEDI-BUILDER et gain en termes de facilité de conception d'un modèle IODA. En effet, les étapes reposant sur des algorithmes sont les plus spécifiques de la méthodologie IODA, et nécessitent un certain degré d'expertise en informatique pour être menées. Il est donc moins gênant que ces étapes soient réalisées directement par une implémentation selon le langage de programmation JAVA, plutôt que par des outils graphiques.

JEDI-BUILDER se focalise actuellement sur un sous-ensemble de sept étapes de la méthodologie IODA, résumées par le rectangle rouge dans la figure 5.18.

**Identification des entités.** L'étape d'identification des entités, notée B sur la figure 5.18, est effectuée dans JEDI-BUILDER par l'édition d'une liste de chaînes de caractères, où chaque chaîne représente l'identifiant d'une famille d'entités. Un identifiant de famille d'entités peut être ajouté, supprimé ou modifié à l'aide d'un menu contextuel apparaissant par un clic droit de la souris.

**Identification des interactions.** L'étape d'identification des interactions, notée C sur la figure 5.18, est effectuée dans JEDI-BUILDER de la même manière que l'étape d'identification des entités, par l'édition d'une liste de chaînes de caractères.

**Spécification d'une matrice d'interaction brute.** L'étape de spécification d'une matrice d'interaction brute, notée E sur la figure 5.18, consiste à faire glisser des identifiants situés dans la liste des interactions vers des cellules de la matrice d'interaction. Ces opérations mènent à l'ouverture d'un menu, dans lequel la garde de distance des éléments d'assignations ainsi créés peut être éditée.

**Spécification d'une matrice d'interaction raffinée.** L'étape de spécification d'une matrice d'interaction raffinée, notée G sur la figure 5.18, consiste à définir la priorité de chaque élément d'assignation, à partir du menu ayant permis d'éditer la garde de distance.

**Spécification d'une matrice de mise à jour.** L'étape de spécification de la matrice de mise à jour, notée H sur la figure 5.18, repose sur des principes identiques à l'édition de la matrice d'interaction brute. Elle a toutefois lieu dans une fenêtre graphique différente de celle permettant l'édition de la matrice d'interaction brute.

**Spécification d'une matrice de mise à jour ordonnée.** L'étape de spécification d'une matrice de mise à jour ordonnée, notée J sur la figure 5.18, repose sur des principes identiques à l'édition de la matrice d'interaction raffinée, et permet de modifier via un menu la priorité de chaque élément d'assignation figurant dans la matrice de mise à jour ordonnée.

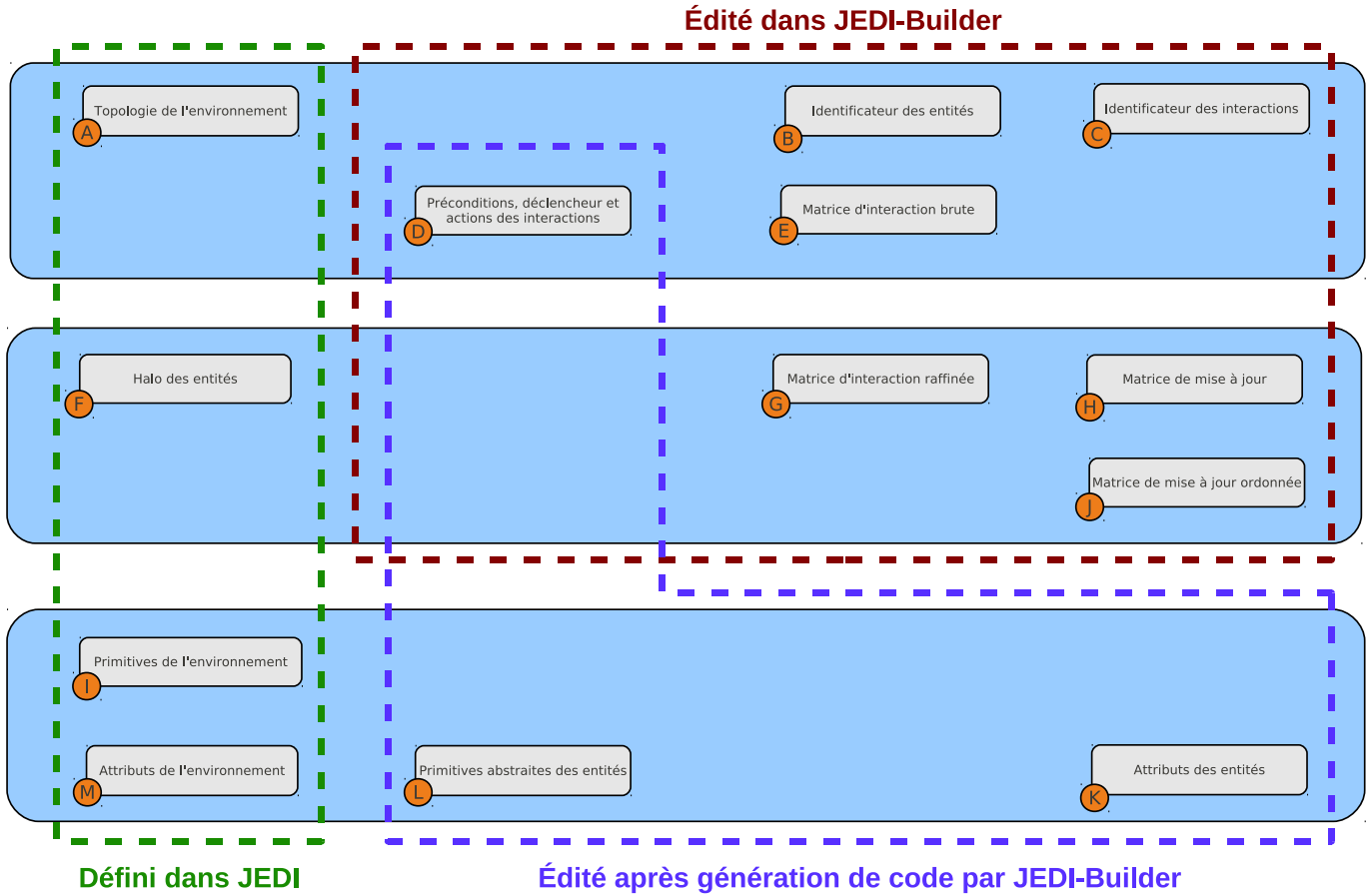


FIGURE 5.18 – Résumé des étapes de la méthodologie IODA couvertes par JEDI-BUILDER. Le rectangle vert représente la part de la méthodologie IODA établissant la sémantique de l'environnement, fixée par la palteforme JEDI, et ne devant pas conséquent pas être éditée dans JEDI-BUILDER. Le rectangle rouge représente la part de la méthodologie IODA pouvant être éditée graphiquement dans l'IDE JEDI-BUILDER. Le rectangle violet représente la part de la méthodologie IODA ne pouvant être édité qu'après la génération de code par JEDI-BUILDER.

**Spécification des préconditions, du déclencheur et des actions des interaction.** Dans la méthodologie de l'approche IODA, l'étape de spécification des préconditions, du déclencheur et des actions des interactions, notée D sur la figure 5.18, consiste à décrire les préconditions, le déclencheur et les actions d'une interaction, et d'en déduire les primitives de perception et d'action figurant dans la signature des source et dans la signature des cibles de cette interaction. Puisque nous avons émis l'hypothèse simplificatrice de spécifier les algorithmes après la génération de code, cette étape de la méthodologie consiste à établir la liste des primitives abstraites des sources et des cibles de chaque interaction. Ces primitives peuvent alors être éditées afin d'en définir l'identifiant, le type de retour, la description de sa fonction en langage naturel, ou les paramètres. La figure 5.19 illustre comment cette édition est effectuée par une capture d'écran de l'interface graphique de JEDI-BUILDER.

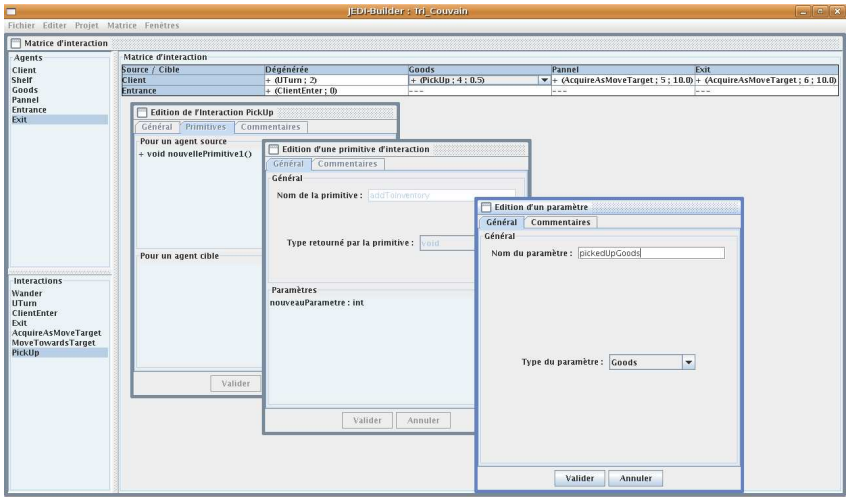


FIGURE 5.19 – Interface permettant d'éditer les primitives d'une entité dans JEDI-BUILDER.

**Génération de code.** A partir des informations de finies graphiquement, JEDI-BUILDER peut générer un ensemble de classes implémentant fidèlement le modèle décrit pour la plateforme JEDI. Les classes générées incluent l'implémentation :

- du squelette de chaque interaction, sous la forme d'une classe. Dans ce squelette, seules restent à définir l'implémentation des préconditions, du déclencheur et des actions ;
- de chaque signature d'entité dans une de ces interactions, sous la forme d'une interface ;
- du squelette de chaque famille d'entités, sous la forme d'une classe. Dans ce squelette, seules restent à définir l'implémentation des primitives abstraites, ainsi que la surface utilisée comme halo, et la primitive d'initialisation de l'entité ;
- d'une fabrique, sous la forme d'une classe, permettant de créer des instances de chaque famille d'agents ;
- de la matrice d'interaction brute et la matrice d'interaction raffinée, sous la forme d'un attribut de la fabrique ;
- du squelette d'une expérience, sous la forme d'une classe. Dans ce squelette, restent à définir le critère d'arrêt de la simulation, et l'initialisation de l'environnement.

Le code généré fait automatiquement le lien entre familles d'entités et signature d'entités, et génère à cette fin une méthode vide pour chaque primitive devant être spécifiée par une famille d'entités. Ainsi, le prototype de JEDI-BUILDER permet actuellement de générer automatiquement un squelette d'implémentation pour JEDI, dans lequel restent à implémenter les préconditions, le déclencheur et les actions des interactions, les primitives des entités, et enfin l'initialisation de la simulation et des entités.

Nous montrons ainsi qu'il est possible d'automatiser la première partie de la méthodologie IODA, et de passer automatiquement à son implémentation sur la plateforme JEDI.

| Relation dans IODA | Implémentation dans JEDI |
|---|---|
| L'entité $e$ a pour famille $\mathcal{F}$, *i.e.* $e \prec \mathcal{F}$ | L'entité $e$ est une instance (au sens de java) de la classe $\mathcal{F}$ |
| Les instances de la famille d'entités $\mathcal{F}$ ont pour signature $S$ dans une interaction $\mathcal{I}$ | $\mathcal{F}$ implémente l'interface $S$ |
| Les entités d'une simulation ont pour signature $S$ dans l'environnement | Toutes les familles d'entités implémentent l'interface $S$ |
| L'environnement a pour signature $S$ dans une interaction | L'environnement implémente l'interface $S$ |

TABLE 5.1 – Tableau récapitulant comment les relations entre les concepts du modèle formel IODA sont implémentées dans la plateforme JEDI

## 5.5 Synthèse du chapitre

Dans ce chapitre, nous avons caractérisé la plateforme de simulation JEDI qui implémente IODA pour les environnements les plus couramment rencontrés en simulation : les environnements euclidiens à deux dimensions. Les tableaux 5.1 et 5.2 montrent qu'il y a une correspondance directe entre chaque concept de IODA et son implémentation dans JEDI sous la forme d'une classe, d'une interface, d'une méthode ou d'un attribut. JEDI est donc une implémentation fidèle de IODA qui conserve la structure du modèle. Elle permet ainsi :

- d'automatiser l'implémentation d'un modèle ;
- de réviser les modèles aisément grâce à la structure modulaire de l'implémentation ;
- de constituer des bibliothèques d'agents et des bibliothèques d'interaction qui facilitent la construction de modèles.

Nous prouvons ainsi que les avantages de IODA ne se limitent pas à la phase d'analyse d'un modèle.

La plateforme JEDI offre un ensemble de paramètres assurant la reproductibilité des simulations, et fournissant un contrôle total sur la prise de parole des entités et sur le déclenchement de leur mise à jour. Ce réglage fin est une nécessité, puisque les algorithmes utilisés pour spécifier une simulation particulière ne sont pas forcément adaptés à d'autres simulations. Les paramètres de IODA permettent de régler finement les algorithmes de simulation, et de les adapter aux besoins des simulations.

Une expérience nécessite plus que l'implémentation du modèle du phénomène. En réponse à cette nécessité, nous avons donc décrit différentes classes nécessaires à l'implémentation d'une expérience. Elles permettent entre autres de :

- définir le critère d'arrêt d'une simulation ;
- définir l'initialisation de l'environnement au début de l'expérience ;
- décrire comment observer la simulation et surtout comment obtenir les données correspondant aux résultats des expériences réalisées.

Enfin, nous avons conçu un prototype d'environnement de développement intégré (IDE) appelé JEDI-BUILDER, que nous utilisons pour confirmer la transversalité de l'approche IODA. En effet, cet IDE permet de spécifier un modèle IODA dans une interface graphique, et d'en générer automatiquement un squelette d'implémentation prêt à remplir pour la plateforme JEDI.

Le changement de perspective induit par notre approche centrée sur les interactions permet d'étudier les problèmes de simulation sous un angle nouveau. Dans la partie suivante, nous nous sommes appuyés sur JEDI et JEDI-BUILDER pour explorer des problématiques inhérentes à la simulation informatique, mais n'apparaissant pas explicitement dans les approches existantes, ou y étant traitées de manière non satisfaisante.

| Notion dans IODA | Nature dans JEDI | Détails |
|---|---|---|
| Attribut d'une famille d'entités | Attribut | Un attribut d'une famille d'entités est représentée par un attribut au sens java dans la classe représentant la famille d'entités. |
| Cellule de la matrice de mise à jour | Classe | Une cellule de la matrice de mise à jour est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. UpdateMatrixCell* |
| Cellule de la matrice de mise à jour ordonnée | Classe | Une cellule de la matrice de mise à jour ordonnée est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. UpdateMatrixCell* |
| Entité | Instance | Une entité est une instance d'une famille d'entités, *i.e.* une instance d'une classe étendant la classe abstraite *fr.lifl.jedi.model.Agent* |
| Environnement | Classe | Un environnement est une instance de la classe *fr.lifl.jedi.model.Environment* |
| Famille d'entités | Classe | Une famille d'entités est une classe étendant la classe abstraite *fr.lifl.jedi.model.Agent* |
| Halo | Classe | Le halo est une instance de la classe *fr.lifl.jedi. model.halo.PhysicalHalo*, générée à l'aide de la classe *fr.lifl.jedi.model.halo.HaloBuilder* (une fabrique abstraite) |
| Interaction Dégénérée | Classe | Une interaction dégénérée est une classe (en général singleton) étendant la classe abstraite *fr.lifl.jedi.model.interactionDeclaration. DegenerateInteraction* |
| Interaction Individuelle | Classe | Une interaction individuelle est une classe (en général singleton) étendant la classe abstraite *fr.lifl.jedi.model.interactionDeclaration. SingleTargetInteraction* |
| Ligne de la matrice d'interaction brute | Classe | Une ligne de la matrice d'interaction brute est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. InteractionMatrixLine* |
| Modèle de sélection d'interaction (*i.e.* ligne de la matrice d'interaction raffinée) | Classe | Le modèle de sélection d'interaction réactif est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. InteractionMatrixLine*, où une priorité est associée à chaque élément d'assignation |
| Matrice d'interaction brute | Classe | La matrice d'interaction brute est une instance de la classe *fr.lifl.jedi.model.InteractionMatrix* |
| Matrice d'interaction raffinée | Classe | La matrice d'interaction raffinée est une instance de la classe *fr.lifl.jedi.model.InteractionMatrix* |
| Matrice de mise à jour | Classe | La matrice de mise à jour est une instance de la classe *fr.lifl.jedi.model.UpdateMatrix* |
| Matrice de mise à jour ordonnée | Classe | La matrice de mise à jour ordonnée est une instance de la classe *fr.lifl.jedi.model.UpdateMatrix* |
| Primitive abstraite | Méthode | Une primitive abstraite est une méthode |
| Signature d'une entité dans une interaction | Interface | La signature d'une entité dans une interaction est une interface étendant l'interface *fr.lifl.jedi.model.interactionDeclaration. EntitySignature* |
| Signature de l'environnement dans une interaction | Interface | La signature de l'environnement dans une interaction est une interface étendant l'interface *fr.lifl.jedi.model.interactionDeclaration. EnvironmentSignature* |
| Signature d'une entité dans l'environnement | Interface | La signature d'une entité dans l'environnement est une interface étendant l'interface *fr.lifl.jedi.model.interactionDeclaration. EntitySignature* |

TABLE 5.2 – Tableau récapitulant comment chaque concept du modèle formel IODA est implémenté dans JEDI. Certains éléments n'y apparaissent pas car JEDI est une plateforme spécifiée pour un cas particulier des modèles IODA : les environnements euclidiens continus en deux dimensions, où les interactions sont de cardinalité $(1, 0)$ ou $(1, 1)$, où le halo est défini par la spécification d'une surface de perception, ou les entités ont un modèle de sélection d'interaction réactif axé sur les priorités.