

A decorative graphic on the right side of the page. It features three blue circles of different sizes, each composed of concentric rings of varying shades of blue. Two thin blue lines intersect at a point between the top and middle circles, extending towards the top-left and bottom-right corners of the page. A third blue circle is partially visible at the bottom right corner.

Description du Plugin: fr.inria.aoste.behavior

Lengellé Denis

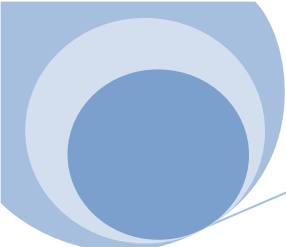
Version 1.10

02/09/2009



Sommaire :

1. Introduction.....	4
2. Spécifications	5
2.1. Architecture de TimeSquare avant l'ajout du plugin de gestion des comportements.....	5
2.2. Les plugins de « références »	7
2.2.1. Le VCD	7
2.2.2. Le papyrus animator	8
2.2.3. Le Sequence Diagram	9
2.2.4. Le Code Execution.....	10
2.3. Liste des spécifications	11
2.4. Architecture actuelle de TimeSquare	12
3. Architecture du plugin	14
4. Un MVC de base	15
4.1. Le contrôleur	16
4.2. Le modèle.....	18
4.2.1. Les classes Entity	19
4.2.2. Les « conteneurs » d'Entity	19
4.2.3. La classe DataStructureManager.....	20
4.2.4. Les classes RelationActivationState et ClockActivationState.....	20
4.3. La vue	21
4.3.1. L'interface BehaviorView.....	22
4.3.2 Les classes RelationBehaviorView et ClockBehaviorView.....	23
4.3.4. La classe BehaviorManagerDialog.....	23
5. La liaison avec les plugins de sorties.....	23
5.1. La package visible	23
5.2. Le package extensionpoint.....	24
5.3. Le package helpers	25
5.4. Interactions avec les plugins de sortie.....	26



6. La sérialisation et le run configuration.....	28
6.1. La sérialisation	28
6.1.1. Les classes PersistentEntity.....	29
6.1.1. Les classes OptionsSerializer	30
6.1.2. La classe XMLStringMaker	31
6.1.3. Une phase de sérialisation.....	33
6.2. La dé-sérialisation	36
6.2.1. Les classes optionsDealer	36
6.2.2. La classe XMLStringParser.....	36
6.2.3. Une phase de dé-sérialisation.....	37
6.3. La liaison avec le simulateur de TimeSquare.....	39
6.3.1. Le run configuration	39
6.3.2. La phase d'exécution de TimeSquare.....	40
6.3.3. La classe OutputOption.....	41
7. Liaison avec les modèles CCSL.....	41
8. Améliorations possibles	43
8.1. Meilleure synchronisation de la phase de configuration.....	43
8.2. Modifications des comportements.....	43
8.3. Utiliser l'ID du point d'extension	44
8.4. Le mode Debug.....	44
8.5. Nouveau helper pour <i>aNextStep()</i>	44
8.6. Ajout de comportements.....	44
8.7. D'un point de vue programmation	44



1. Introduction

Ce rapport explique en détail le fonctionnement du plugin de gestion de comportements du MDK TimeSquare : **fr.inria.aoste.behavior**.

Ce plugin à été développé afin d'organiser et d'harmoniser le développement de plugins Eclipse liés au MDK en leur permettant de définir des comportements sur des événements durant une exécution de TimeSquare.



2. Spécifications

2.1. Architecture de TimeSquare avant l'ajout du plugin de gestion des comportements

L'architecture de TimeSquare avant l'intégration du plugin de gestion des comportements est représentée illustration 1. On peut distinguer deux parties bien distinctes :

- le noyau de TimeSquare qui se charge de créer une exécution. Celle-ci est modélisée dans la Trace. Le simulateur offre un point d'extension qui permet à des plugins d'utiliser cette Trace.
- les plugins « de sortie » qui effectuent leurs opérations pendant l'exécution via le point d'extension du simulateur qui donne accès à la Trace.

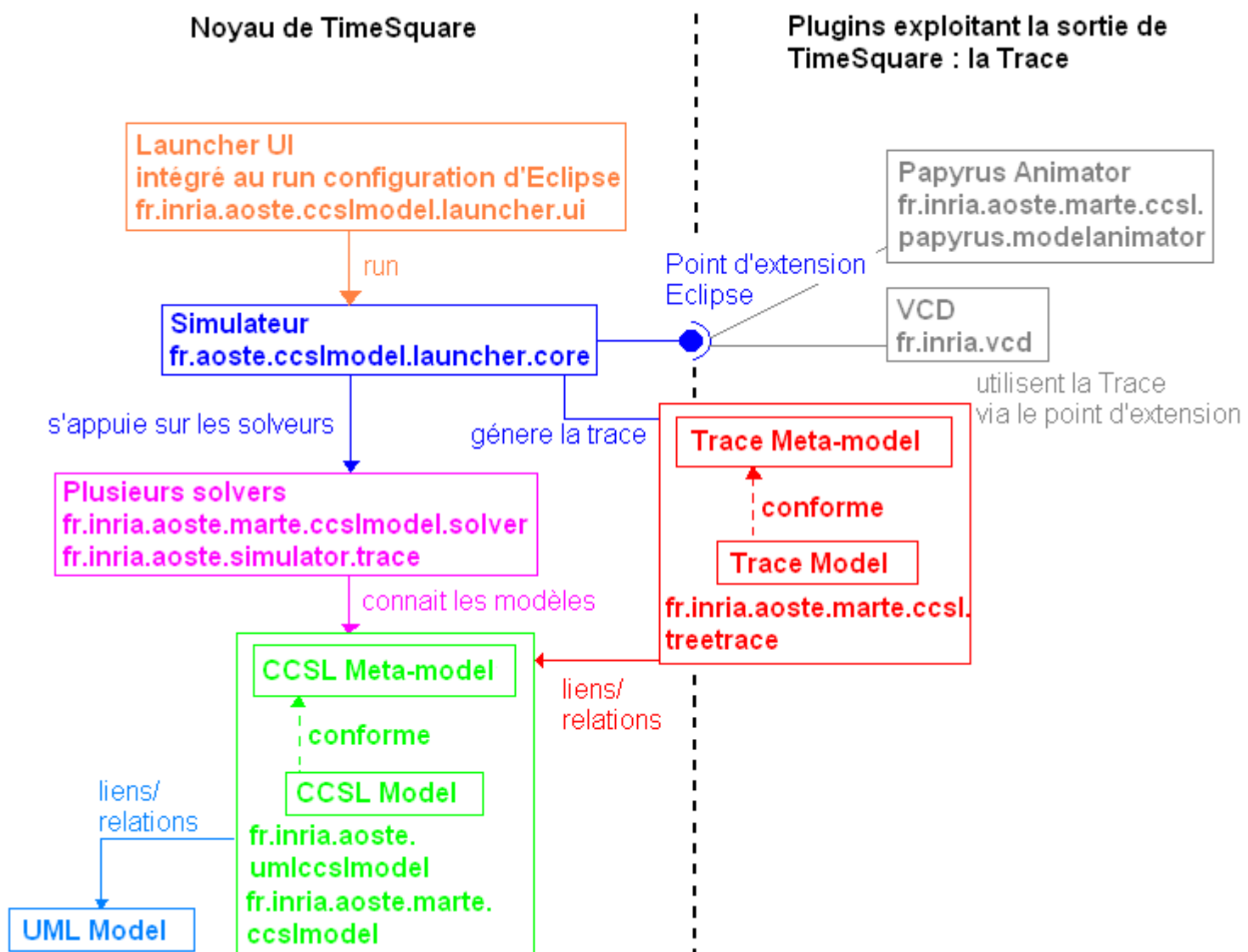
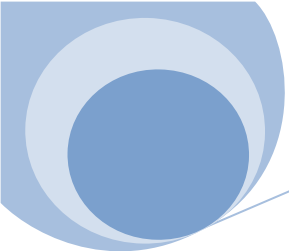


Illustration 1 : Architecture de TimeSquare avec l'intégration du plugin de gestion des comportements.

Dans cette architecture, le simulateur a un rôle très important car en plus de générer une simulation et la Trace correspondante, il doit permettre aux plugins de sortie l'accès à toutes les informations dont ils peuvent avoir besoin en plus de la Trace.

Ensuite, par le point d'extension, il informe le Launcher UI des plugins présents en sortie afin d'impacter l'UI du run configuration d'Eclipse. Tout un protocole concernant les options des plugins est mis en place afin de les afficher dans le Launcher UI et de les faire prendre en compte par le Simulateur qui les donnera enfin aux plugins eux même.



Durant l'exécution, la Trace est donnée telle quelle à chaque plugin de sortie. C'est au plugin d'en parcourir le modèle, d'y extraire les informations puis d'effectuer leurs tâches si nécessaire.

Nous rappelons que la trace contient entre autre les états de toutes les horloges du modèle CCSL à un moment donné. Les plugins de sortie effectuent tous la même opération concernant la Trace : détecter si les états des horloges activent certaines actions qui leur sont propres.

A l'image du modèle de Trace, un modèle de Relations entre instants est développé et allait être intégré à TimeSquare. Ce modèle détecte les relations liant les instants durant une exécution (cf.

<svn://pentan/AOSTE/DEVEL/openembedd2/fr.inria.aoste.umlrelationmodel/rapport>).

Il s'est avéré qu'il fallait à terme un système permettant d'ajouter des comportements sur ces relations de la même manière que pour les états d'horloge.

Des helpers sur les modèles de Trace et de Relation entre instants étaient nécessaires afin de faciliter les opérations effectuées par les plugins sur ces modèles. Ce travail était jusqu'à présent fait par tous les plugins en même temps.

Enfin, la création et l'intégration d'un plugin de sortie à TimeSquare étaient des tâches compliquées et non standardisées. Les services offerts par le point d'extension du Simulateur n'étaient pas suffisants et la volonté d'exécuter des comportements sur un état d'horloge n'y était pas bien représentée. Il fallait un système général permettant de spécifier finement quand un comportement doit être déclenché.

2.2. Les plugins de « références »

Les spécifications ont été obtenues en se basant sur des plugins de sortie vraiment différents les uns des autres. Le but étant de répondre à leurs besoins à tous et ceci de manière générale.

2.2.1. Le VCD

fr.inria.vcd

Ce plugin a pour rôle d'afficher à l'écran la simulation en cours sous forme de chronogramme. Cette affichage est incorporé dans une vue Eclipse. Le VCD demande à l'utilisateur des options générales via un panel graphique lors de la



phase de configuration. Durant l'exécution, il doit représenter l'état de chaque horloge sur le chronogramme grâce au modèle de Trace. Il a aussi besoin du modèle de relations pour les afficher lorsque l'utilisateur clique sur un instant du chronogramme.

Ce type de plugin de sortie ajoute des comportements sur tous les états de toutes les horloges ainsi que sur toutes les relations entre instants détectées. Les comportements ajoutés sont transparents aux yeux de l'utilisateur, celui-ci ne fait que choisir si oui ou non, il souhaite voir le VCD s'afficher.

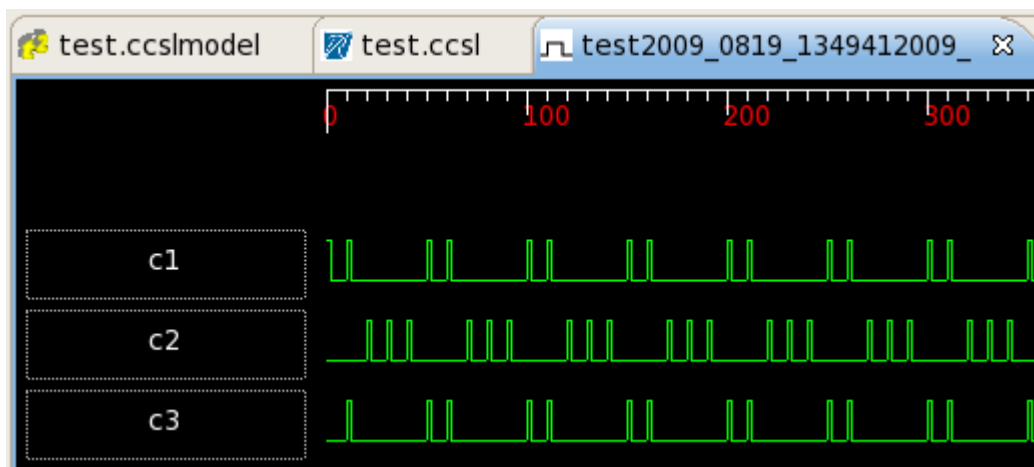


Illustration 2 : Le VCD.

2.2.2. Le papyrus animator


fr.inria.aoste.marte.ccs1.papyrus.modelanimator

Le Papyrus Animator repose sur un plugin Eclipse permettant de manipuler graphiquement des modèles : *Papyrus*.

<http://www.papyrusuml.org/>

Avant la création du plugin de gestion des comportements, le Papyrus Animator permettait de colorier en rouge la classe du diagramme UML (fichier Di2) associée à une horloge du modèle CCSL lorsque pendant l'exécution, cette horloge tick.

Grace au plugin de gestion des comportements, le Papyrus Animator peut être amélioré afin de demander à l'utilisateur de choisir la classe associée à l'horloge qu'il souhaite voir animée, avec la couleur de son choix et surtout sur un état d'horloge précis.



Ce plugin à besoin d'options générales telles que le nom du fichier Di2 à animer. De plus, l'utilisateur doit pouvoir configurer lui-même les comportements qu'il souhaite par l'intermédiaire d'une interface graphique propre à ce plugin.

2.2.3. Le Sequence Diagram

fr.inria.aoste.sequencediagram

Le Sequence Diagram fourni en fin d'exécution une représentation des modèles de Trace et de Relations entre instants sous forme de diagramme de séquence. Il utilise le plugin de manipulation graphique des modèles : *Papyrus*.

Ce plugin est du même type que le VCD. Les comportements sont définis par le plugin et l'utilisateur choisi en phase de configuration s'il souhaite la génération du diagramme de séquence.

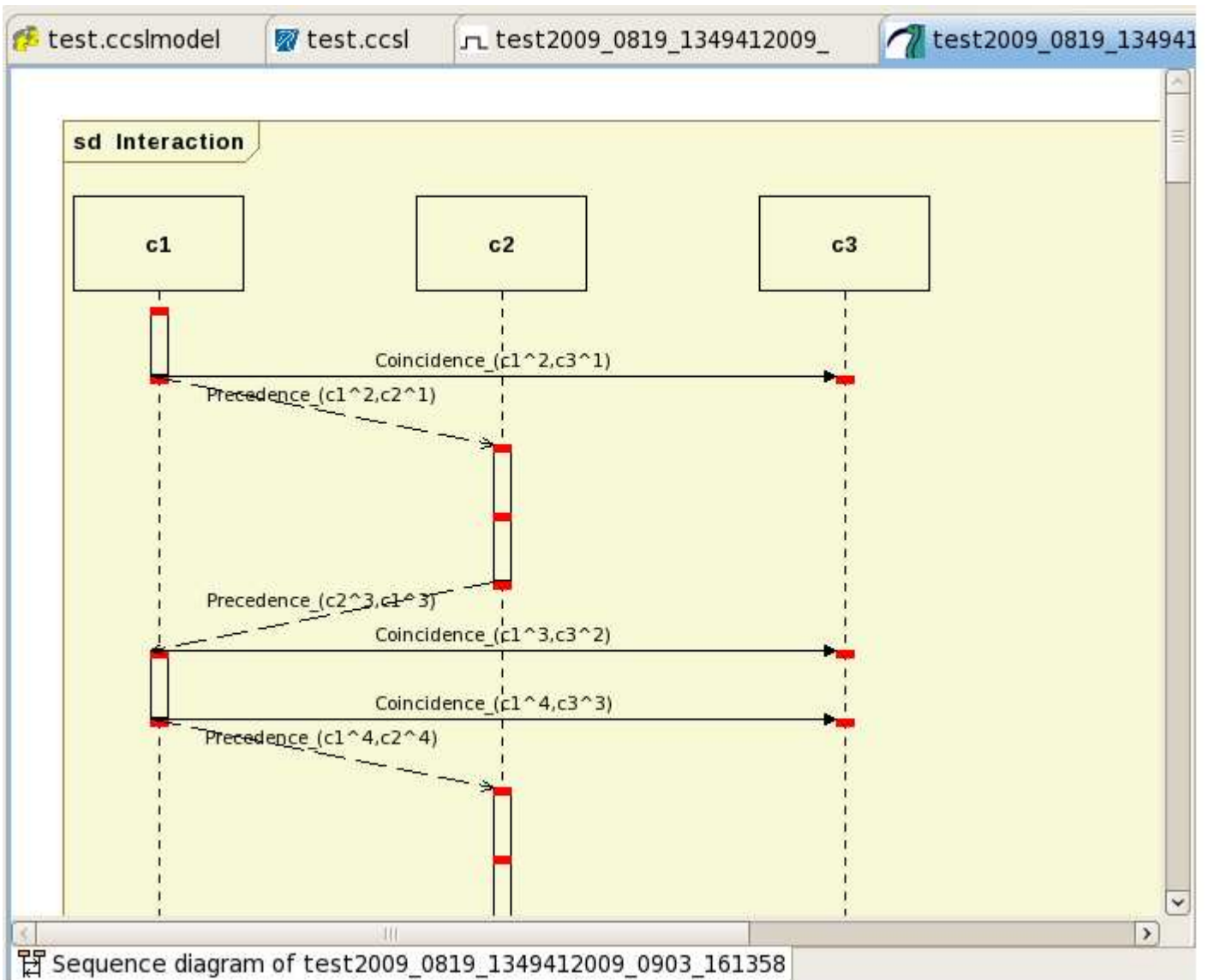


Illustration 3 : Un diagramme de séquence.

2.2.4. Le Code Execution

fr.inria.aoste.codeexecution

Ce plugin permet d'exécuter du code Java que l'utilisateur a préalablement sélectionné dans la phase de configuration. Ce plugin est du même type que le Papyrus Animator, les comportements sont personnalisables par l'utilisateur qui doit choisir le code qu'il souhaite voir exécuter sur un état d'horloge ou une relation. L'interface graphique d'un tel plugin lui est forcément spécifique.



2.3. Liste des spécifications

L'étude de ces différents plugins à permis de construire la liste de spécifications ci-dessous pour le module **fr.inria.aoste.behavior** :

- le module doit permettre la gestion de comportements sur des états d'horloge du modèle de Trace et des relations du modèle de Relations entre instants ;
- le module doit simplifier au maximum le développement et l'intégration de plugins de sortie ;
- le module doit fournir des helpers sur les modèles ;
- le module ne doit pas être dépendant d'un plugin de sortie en particulier ;
- le module doit respecter les différences des plugins de sortie concernant la phase de configuration des comportements ;
- le module doit permettre la configuration de comportements par un moyen ergonomiquement intégré à Eclipse, pour cela le run configuration est utilisé ;
- afin d'être intégré au run configuration d'Eclipse, le module doit être sérialisable.

2.4. Architecture actuelle de TimeSquare

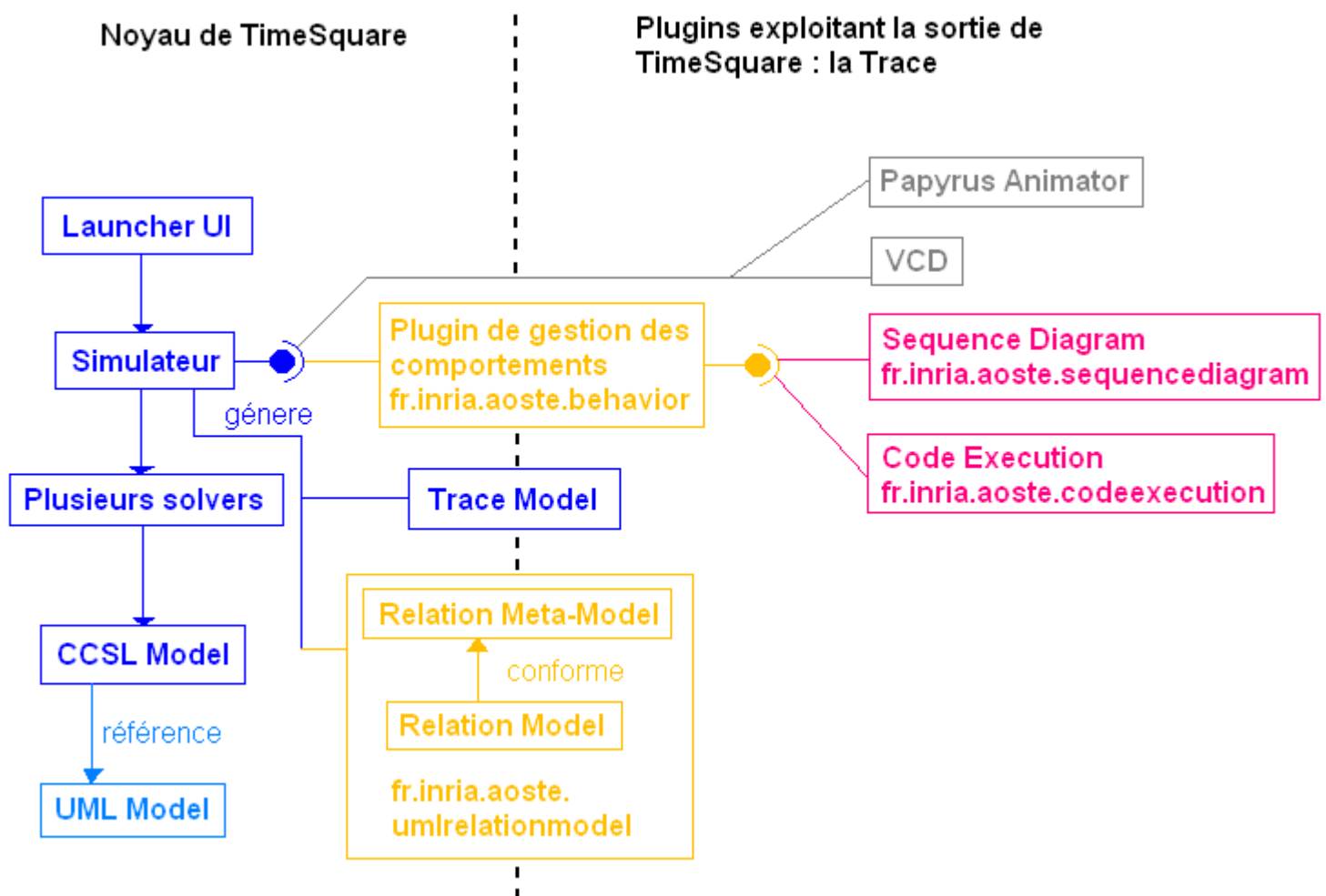



Illustration 4 : Architecture actuelle de TimeSquare.

La solution actuelle automatise et gère la notion de comportements. En phase de configuration, il récupère les comportements définis par les plugins de sortie et l'utilisateur. En phase d'exécution de TimeSquare, il récupère les informations du Simulateur par le point d'extension et se charge d'activer les comportements définis.

Le module **fr.inria.aoste.behavior** est intégré au run configuration d'Eclipse par l'intermédiaire du Simulateur et du Launcher GUI.

Le run configuration d'Eclipse englobe la notion de sérialisation de par sa capacité à enregistrer les options des utilisateurs, le module de gestion des comportements y est conforme.

Les plugins de sortie étant différents les uns des autres lors de la configuration des comportements, le plugin de gestions des comportements propose un protocole



d'ajout, suppression de comportements mais laisse le plugin de sortie complètement libre concernant la définition de ceux-ci (interface graphique etc.).

Dans l'architecture actuelle de TimeSquare qui est représentée Illustration 2, des changements sont notables :

- les informations en sortie de TimeSquare se sont vues enrichies du nouveau modèle de Relations en plus de la Trace ;
- les plugins de sorties Sequence Diagram et Code Execution ont été ajoutés ;
- enfin, le plugin de gestion de comportements est intégré. Il permet d'ajouter des comportements sur des états d'horloge et sur des relations entre instants. Il généralise la manière de distribuer les informations pendant une exécution de TimeSquare aux plugins de sortie. Ce plugin est détaillé dans ce rapport.

On peut remarquer que les nouveaux plugins de sorties sont connectés au plugin de gestion de comportements alors le papyrus animator et le VCD sont encore connectés sur le simulateur. Ceux-ci doivent migrer sur le plugin **fr.inria.aoste.behavior** dans une future étape de développement.

Actuellement, les plugins de sortie peuvent accéder à la Trace ainsi qu'au modèle de Relations via le plugin de gestion des comportements qui leur fournit des helpers.

Enfin, le plugin de gestion des comportements a été conçu dans l'optique d'alléger le travail que représente l'intégration d'un plugin de sortie à TimeSquare. Cette intégration n'est pas détaillée dans ce rapport et est traitée dans un autre document à l'adresse :

<svn://pentan/AOSTE/DEVEL/openembedd2/fr.inria.aoste.behavior/rapports/InstallationPlugin.pdf>

3. Architecture du plugin

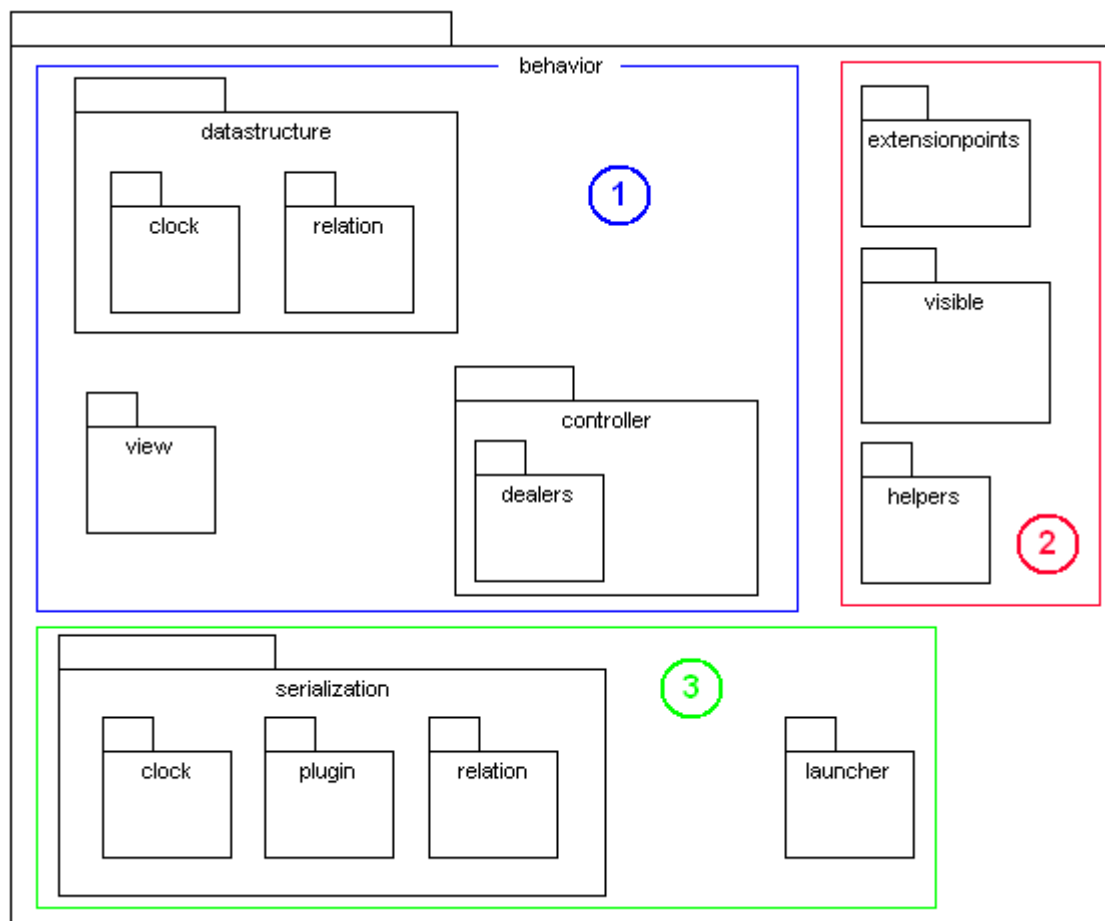


Illustration 5 : Les packages du module de gestion des comportements.

L'illustration 5 montre la décomposition en package du module **fr.inria.aoste.behavior**.

Le contenu peut se décomposer en 3 parties :

- 1 Ceci est le modèle - vue - contrôleur qui est à la base de l'architecture du plugin. Le contrôleur en est le chef d'orchestre, c'est lui qui coordonne toutes les opérations que ce soit en phase de configuration ou d'exécution. Ce MVC est détaillé chapitre [4. Un MVC de base](#) de ce document.



2

Ces packages assurent la liaison entre ce plugin et les plugins de sortie. La gestion du point d'extension y est prise en charge. Ils contiennent toutes les interfaces et les classes qui interagissent avec ces plugins. Le chapitre [5. La liaison avec les plugins de sorties](#) détail ces packages.

3

Tout ce qui concerne la liaison à TimeSquare via le point d'extension du Simulateur ainsi que la sérialisation des comportements pour le run configuration d'Eclipse est contenu dans ces packages. Ceci est expliqué chapitre [6. La sérialisation et le run configuration](#) .

4. Un MVC de base

L'architecture du plugin **fr.aoste.inria.behavior** repose sur un Modèle-Vue-Contrôleur.

Rappel sur le MVC :

<http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

4.1. Le contrôleur

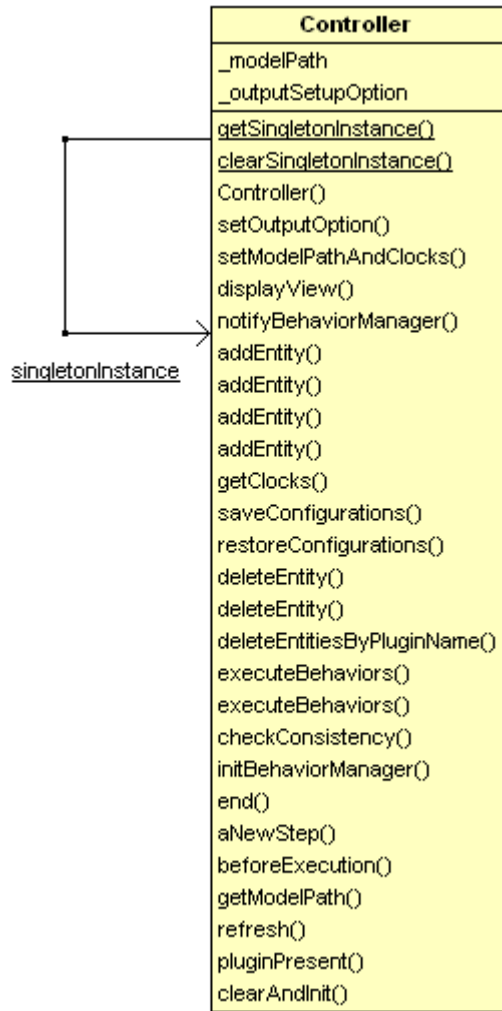


Illustration 6 : La classe Controller.

La classe `Controller` du package `fr.inria.aoste.behavior.controller` est la classe la plus importante de ce plugin. C'est le chef d'orchestre qui organise et contribue à toutes les fonctionnalités du plugin. En plus de gérer son modèle et sa vue, il se charge entre autre de la liaison avec les plugins de sortie ainsi que la sérialisation pour le run configuration.

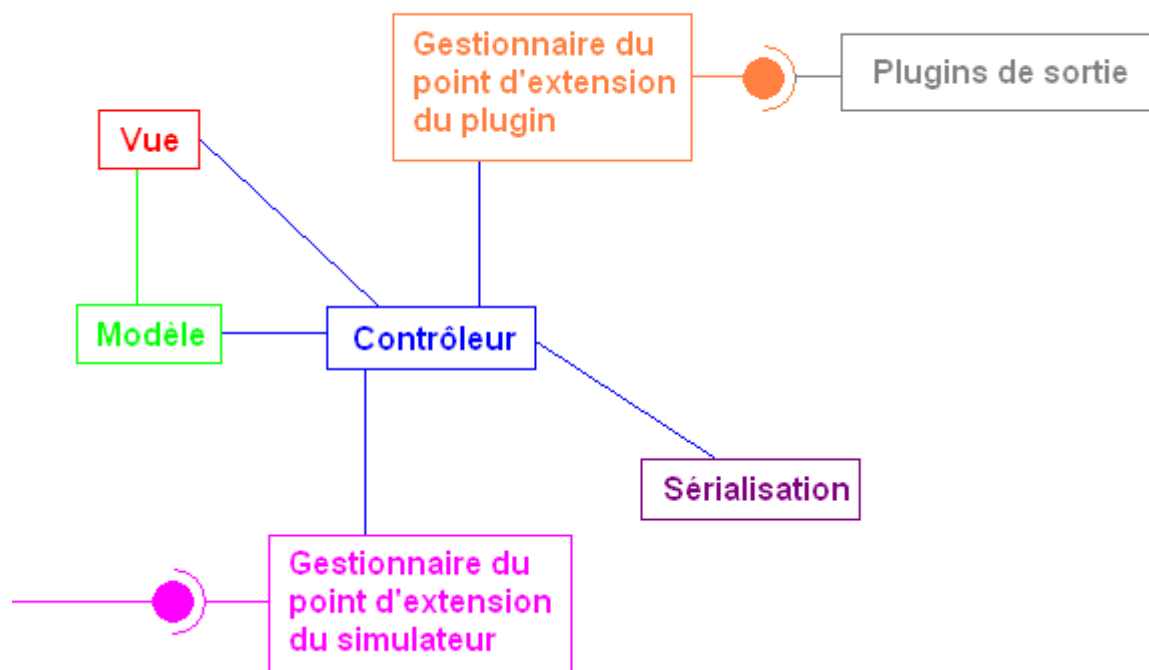


Illustration 7 : Le Controller au centre des différents composants du module.

Cette classe est un singleton et est utilisée durant la phase de configuration des comportements et la phase d'exécution de TimeSquare. C'est le composant central du module.

Les classes du package **fr.inria.aoste.behavior.controller** sont les seules à interagir avec les plugins de sortie. On peut remarquer la présence d'un sous-package **fr.inria.aoste.controller.dealers**. Ce package est utilisé durant la phase de dé-sérialisation des comportements détaillée chapitre [6.2. La dé-sérialisation](#). Sa présence dans ce package provient du fait qu'il interagit lui aussi avec les plugins de sortie durant la phase de dé-sérialisation.

4.2. Le modèle

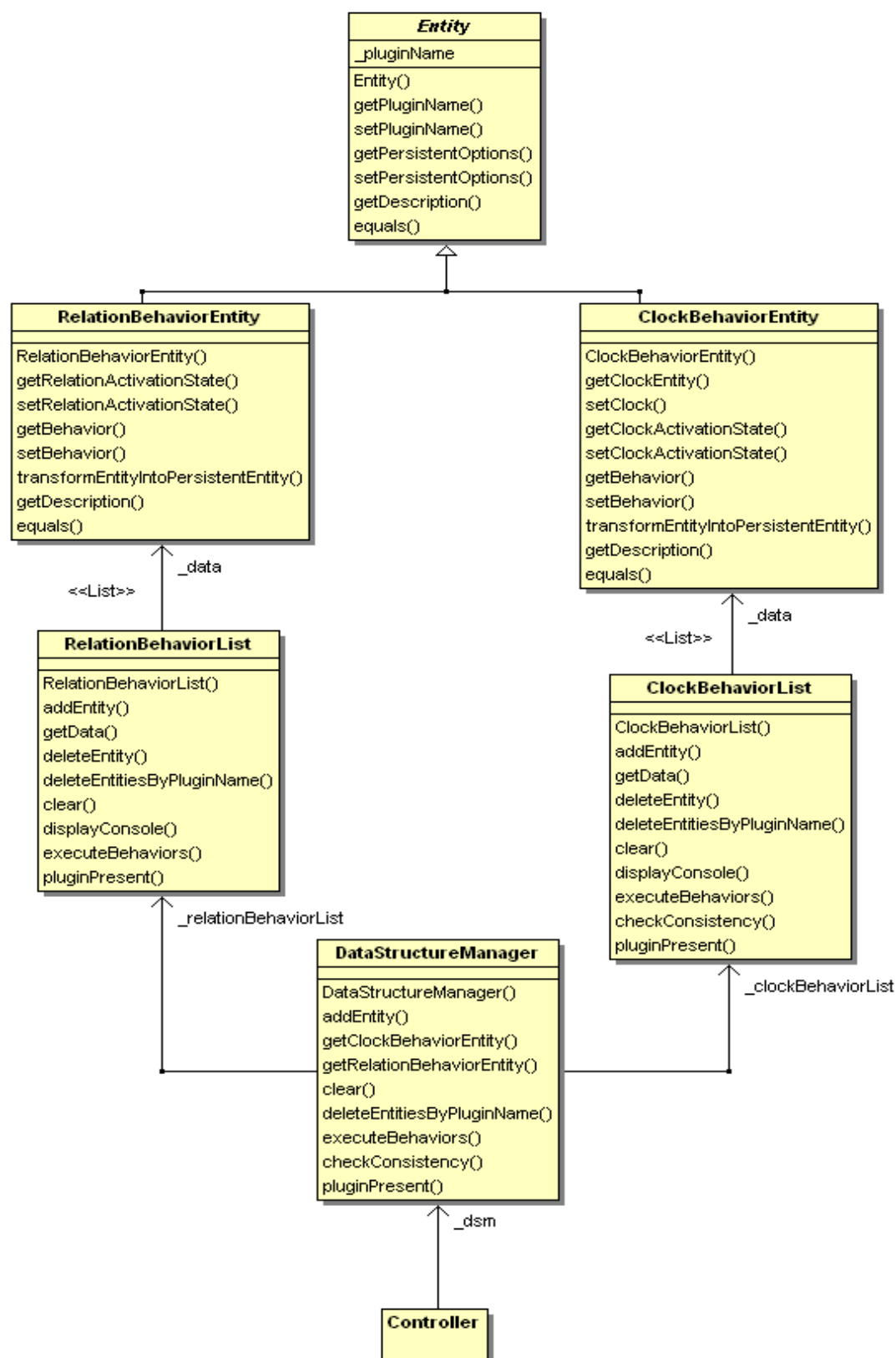
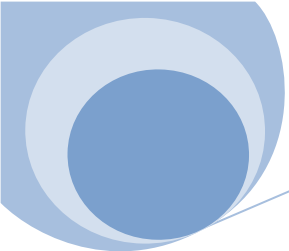


Illustration 8 : Diagramme de classes du modèle.



Le diagramme de classes de l'illustration 8 représente le modèle de ce MVC.

4.2.1. Les classes *Entity*

Le but du modèle est d'enregistrer les comportements qui ont été ajoutés par un plugin ou par un utilisateur lors de la phase de configuration. La classe abstraite *Entity* et ses classes filles contiennent toutes les informations concernant un comportement. Ce sont les briques de base du plugin et elles sont conservées dans ce modèle.

Les informations qui sont regroupées sont :

- le comportement ;
- l'état d'activation ;
- le nom du plugin qui a défini ce comportement ;
- une version sérialisable du comportement ;
- une horloge dans le cas d'un comportement sur état d'horloge.

4.2.2. Les « conteneurs » d'*Entity*

Les classes *RelationBehaviorList* et *ClockBehaviorList* sont ce que j'appelle ici des conteneurs d'*Entity*. Ils possèdent une chacun une liste d'entités et offrent des méthodes pour les manipuler.

On peut voir que de nombreuses méthodes sont similaires, et qu'une classe mère *BehaviorList* pourrait être utilisée.

En regardant plus en détail, on comprend que les entités requièrent en fait des conteneurs qui leur sont propres. De plus, on ne peut pas prévoir dès maintenant quels autres comportements et donc entités seront ajoutés au plugin, ni quelles seront les méthodes nécessaires pour les manipuler. Pour cela, dans la version actuelle de ce plugin, les conteneurs sont laissés « libres » et ne dépendent d'aucune autre classe si ce n'est leur entité.

4.2.3. La classe `DataStructureManager`

Cette classe fait office de grosse interface entre le contrôleur et les différents conteneurs d'entités. On aurait pu imaginer le contrôleur manipulant lui-même les conteneurs mais cette gestion « lourde » l'aurait complexifiée encore plus.

Lorsque le contrôleur a besoin d'agir sur le modèle, il appelle une méthode du `DataStructureManager`. C'est ensuite au `DataStructureManager` d'appliquer la modification sur les conteneurs concernés.



Si de nouveaux types de comportements devaient être développés et intégrés au plugin, il faut dériver la classe `Entity`, créer un conteneur pour ces entités et enfin modifier la classe `DataStructure` pour y intégrer le nouveau conteneur afin de lui informer des modifications venant du `Controller`.

Pour ajouter une fonctionnalité permettant de modifier des comportements, cela revient à modifier ces entités dans le modèle.

4.2.4. Les classes `RelationActivationState` et `ClockActivationState`

ClockActivationState	RelationActivationState
<u>stateNumber</u>	<u>relationNumber</u>
<u>stateList</u>	<u>relationList</u>
<u>_state</u>	<u>_relation</u>
ClockActivationState()	RelationActivationState()
ClockActivationState()	RelationActivationState()
getState()	isValidRelation()
isValidClockState()	getRelation()
getDescription()	getDescription()
equals()	equals()
<u>stateActivationOK()</u>	<u>relationActivationOK()</u>
<u>and()</u>	<u>and()</u>

Illustration 9 : Les classes définissant les états d'activation d'un comportement.

Ces classes définissent les états d'activation des comportements sur les états d'horloge ainsi que les relations. Cette notion est détaillée dans le rapport décrivant l'intégration d'un plugin de sortie (cf.

<svn://pentan/AOSTE/DEVEL/openembedd2/fr.inria.aoste.behavior/rapports/InstallationPlugin.pdf>).

De plus, ces classes sont utilisées durant la phase d'exécution au chapitre [6.3.2. La phase d'exécution de TimeSquare](#).

4.3. La vue

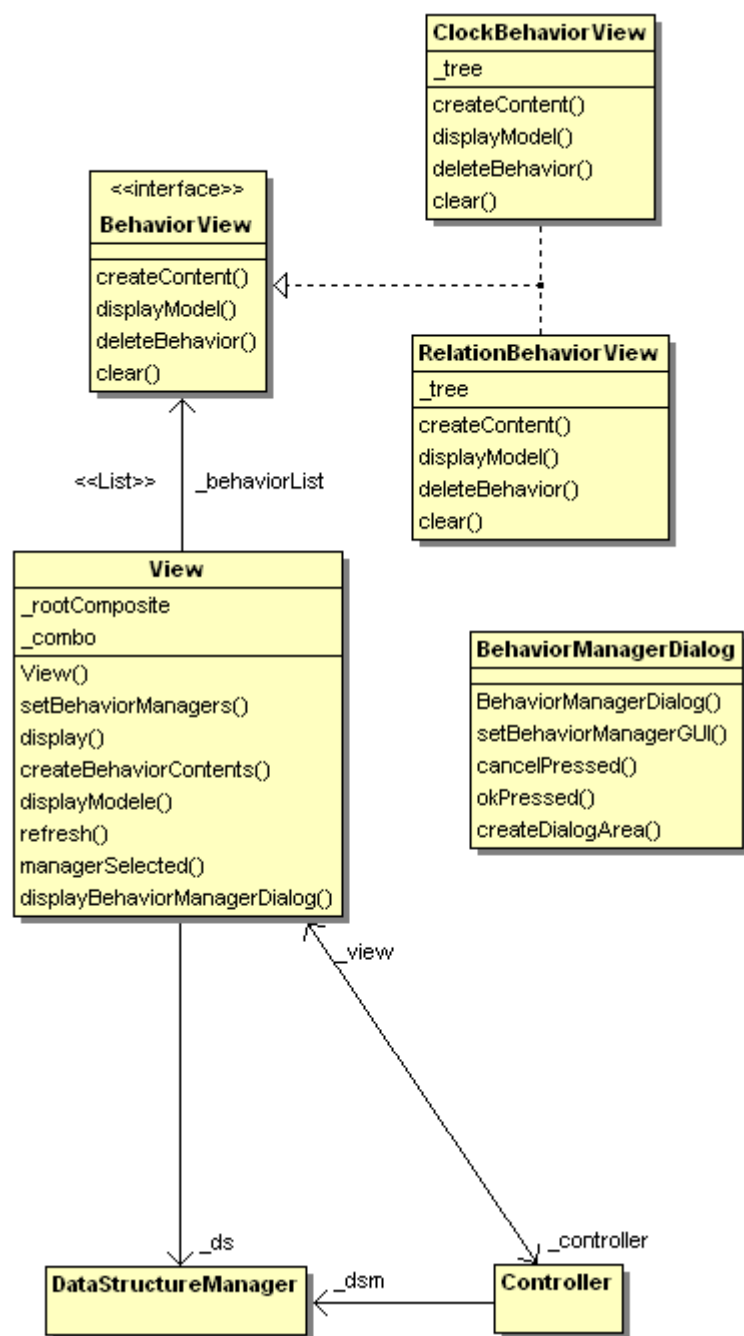
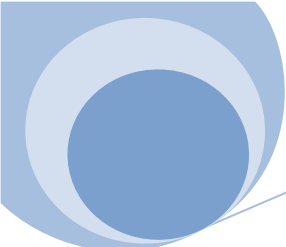


Illustration 10 : Diagramme de classes de la vue.



La vue de se MVC s'affiche directement dans le run configuration d'Eclipse. Afin de s'abstraire de cette notion à ce niveau, la méthode principale d'affichage de la vue *display()* placera ses éléments dans un Composite SWT qui lui est donné en paramètre du constructeur de la classe `View`. La manière de récupérer ce Composite qui fait partie du run configuration est indiquée dans le chapitre [6.3.1. Le run configuration](#).

Les bibliothèques graphiques utilisées dans tout ce plugin sont :

- SWT : <http://www.eclipse.org/swt/>
- JFace : <http://wiki.eclipse.org/index.php/JFace>

Comme cela est montré dans l'illustration 10, la classe `View` possède une référence sur le modèle (`DataStructureManager`) et sur le contrôleur (`Controller`). Elle répercute les choix des utilisateurs sur le contrôleur qui traite l'information, modifie le modèle en fonction et demande à la vue de se réafficher.



Dans ce MVC, ce n'est pas le modèle qui notifie la vue d'un changement mais le contrôleur.

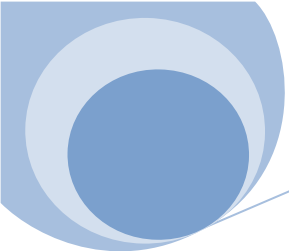
4.3.1. L'interface `BehaviorView`

Cette interface a été créée afin de permettre la visualisation de nouveaux types de comportements de manière simple.

La classe `View` possède une liste d'objets `BehaviorView` et interagit avec eux par l'intermédiaire des méthodes de l'interface. La manière dont sont implémentées les méthodes est propre à la classe implémentant l'interface.



Si de nouveaux comportements sont développés, pour les intégrer à la vue, il suffit de créer une classe implémentant l'interface `BehaviorView` puis de l'ajouter à la liste de l'objet `View`.



4.3.2 Les classes `RelationBehaviorView` et `ClockBehaviorView`

Ces classes implémentent l'interface `BehaviorView` et servent d'affichage pour les comportements sur les états d'horloges ainsi que pour les relations.

4.3.4. La classe `BehaviorManagerDialog`

La classe `BehaviorManagerDialog` est un dialog `JFace` qui est utilisé lorsqu'un utilisateur sélectionne dans la vue principale un plugin de sortie qui a choisi d'utiliser le service de base d'affichage graphique offert par le plugin **fr.inria.aoste.behavior**.

C'est par cette boîte de dialogue qu'un plugin de sortie peut faire choisir des options à un utilisateur pour configurer les comportements à ajouter.

Lorsqu'un utilisateur clique sur le nom d'un plugin dans le run configuration qui a défini un `BehaviorManagerGUI` (cf.

<svn://pentan/AOSTE/DEVEL/openembedd2/fr.inria.aoste.behavior/rapports/InstallationPlugin.pdf>), celui-ci sera affiché dans un `BehaviorManagerDialog`.

5. La liaison avec les plugins de sorties

5.1. La package visible

Ce package est le seul à être visible aux yeux des plugins dépendant de **fr.inria.aoste.behavior** tels que les plugins de sortie. Il concerne la connexion d'un plugin de sortie sur le plugin de gestion de comportements. Tout ceci est détaillé dans un autre rapport :

<svn://pentan/AOSTE/DEVEL/openembedd2/fr.inria.aoste.behavior/rapports/InstallationPlugin.pdf>.

La notion la plus importante qui nous concerne ici est qu'un plugin de sortie est vu au sein du plugin de gestion des comportements par l'intermédiaire d'une interface que le plugin doit implémenter : le `BehaviorManager`.

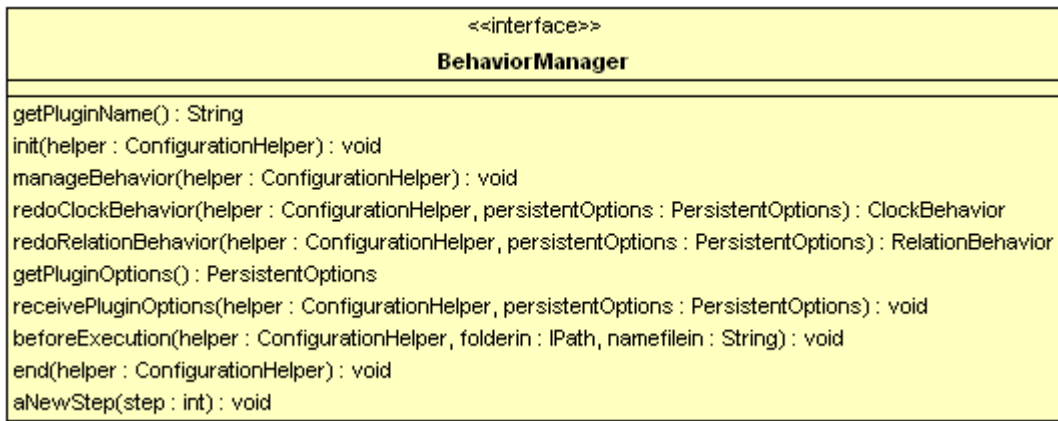


Illustration 11 : La classe BehaviorManager.

5.2. Le package extensionpoint

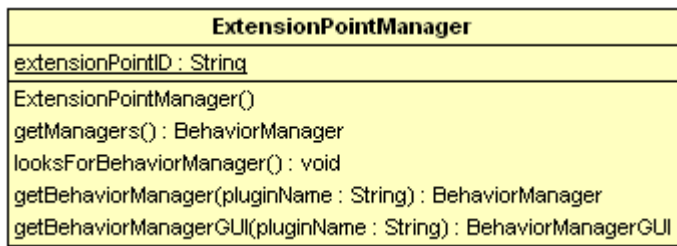


Illustration 12 : La classe ExtensionPointManager.

Le plugin de gestion des comportements offre un point d'extension aux plugins de sortie : **fr.inria.aoste.behavior.behaviormanager**.

Ce package ne contient qu'une seule classe (cf. illustration 12). Elle se charge de récupérer les plugins connectés au point d'extension, d'en instancier les classes BehaviorManager et BehaviorManagerGUI (seulement si le plugin veut utiliser le service d'affichage de base).

C'est le contrôleur qui instancie la classe ExtensionPointManager et lui demande de récupérer les BehaviorManager dans son constructeur. Ceci est fait une seule fois, à l'initialisation du Controller. Les BehaviorManager et BehaviorManagerGUI récupérés ainsi sont ensuite stockés dans une liste, le point d'extension n'est plus utilisé.

5.3. Le package helpers

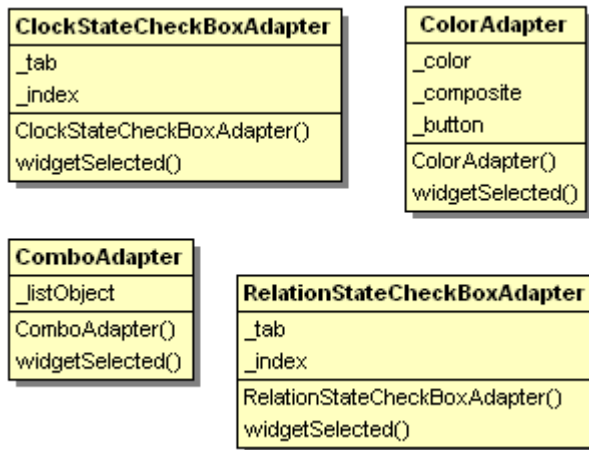


Illustration 13 : Les classes du package `fr.inria.aoste.behavior.helpers`.

Le plugin de gestion des comportements détaillé dans ce document offre des helpers aux plugins de sortie. Actuellement quatre helpers sont donnés :

En phase de configuration :

- ConfigurationHelper
- GUIHelper

En phase d'exécution :

- TraceHelper
- RelationHelper

Ceux-ci sont présent dans le package visible par les plugins de sortie :

fr.inria.aoste.behavior.visible.

Ces helpers peuvent utiliser des classes annexes qui leur sont propres et qui ne doivent pas forcément être visibles aux plugins de sortie. Ces classes se trouvent dans le package **fr.inria.aoste.behavior.helpers**.

Pour le moment, les classes présentes dans ce package sont des adaptors graphiques pour le `GUIHelper` qui aide au développement d'une interface graphique d'un plugin de sortie.

5.4. Interactions avec les plugins de sortie

Les diagrammes de séquence suivants illustrent le protocole entre le plugin de gestion des comportements avec deux plugins de sortie dont un (Code Execution) définissant une interface graphique (BehaviorManagerGUI).

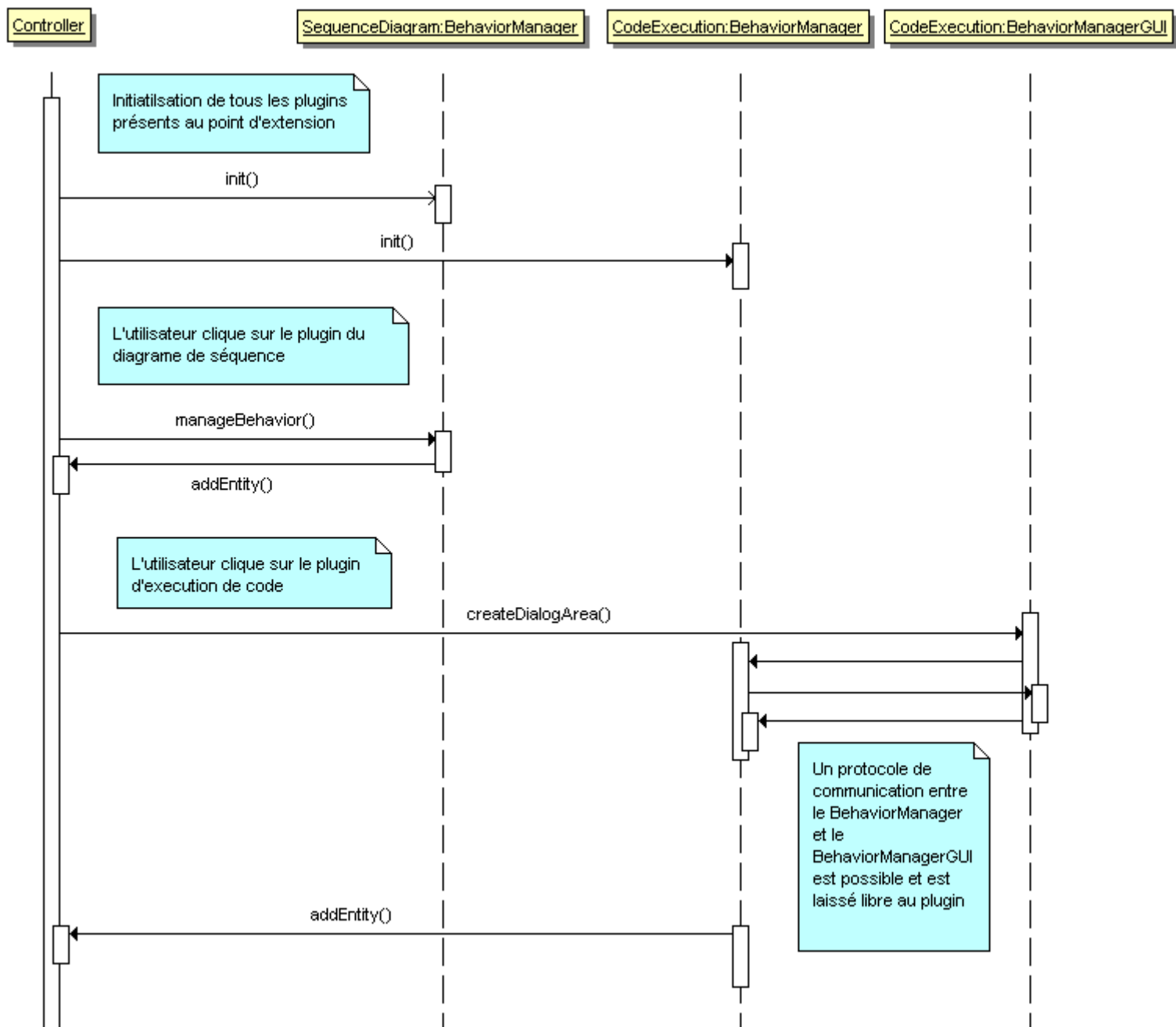


Illustration 14 : Phase de configuration.

L'illustration 14 représente la communication entre Controller et BehaviorManager lors de la phase de configuration.



La fonction *init()* peut être appelée plusieurs fois :

- Une fois de base lorsque l'utilisateur lance un run configuration dans Eclipse.
- A chaque fois que l'utilisateur change le nom du modèle et/ou les horloges qui y sont définies.

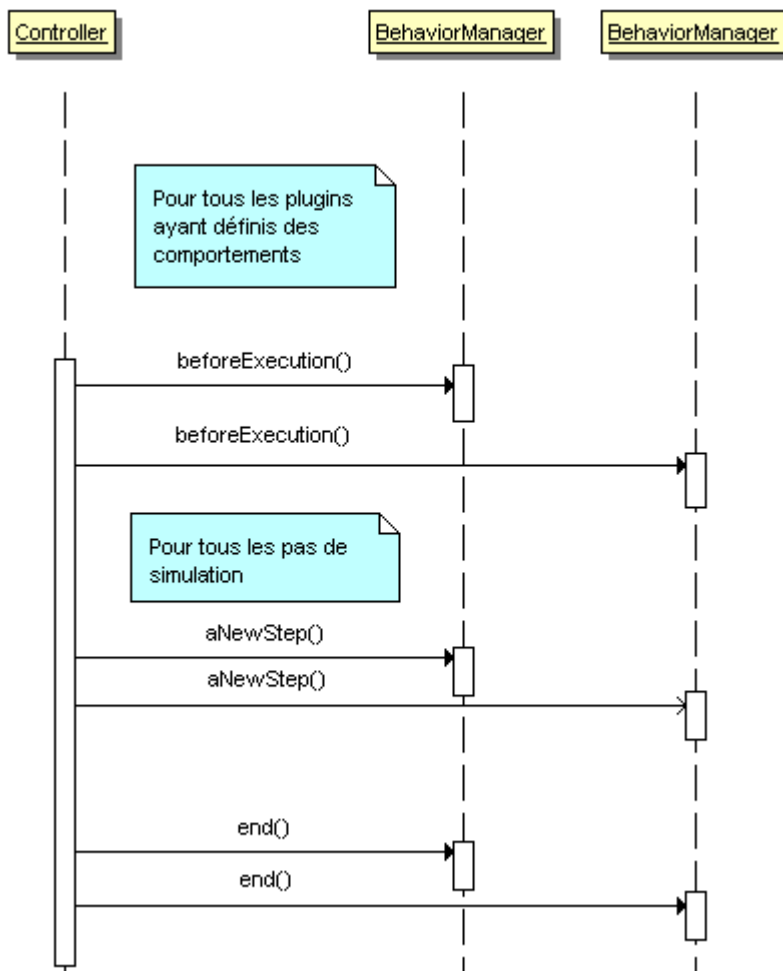


Illustration 15 : Phase d'exécution.

L'illustration 15 représente la communication entre Controller et BehaviorManager lors de la phase d'exécution.

6. La sérialisation et le run configuration

L'intégration du plugin au run configuration et la sérialisation des comportements sont fortement liés. En effet, le run configuration d'Eclipse impose une sérialisation sous forme de String.

Tout le plugin a donc la capacité de se sérialiser en une seule String.

6.1. La sérialisation

La question à se poser ici est : que faut-il sérialiser ?

Il est clair que le modèle du MVC contenant tous les comportements doit être sérialisé.

Mais pas seulement, les plugins de sortie ont aussi des options qui leur sont propres, que l'utilisateur a pu définir dans leur interface graphique. Ces options générales sont aussi à sérialiser.

Ces options ainsi que les comportements étant des classes propres aux plugins de sortie, on ne peut pas leur imposer une contrainte telle que le fait que ces classes doivent être sérialisables.

Pour pallier à ce problème, le plugin de gestion des comportements offre une interface `PersistentOptions` dans le package **fr.inria.aoste.behavior.visible** qui à pour unique rôle d'implémenter elle-même l'interface `Serializable` de Java.

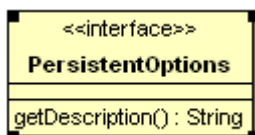


Illustration 16 : La classe `PersistentOptions`.

Tout ce qu'un plugin doit ou veut sérialiser doit se trouver dans des classes implémentant cette interface.

Lors de l'ajout d'un comportement, le plugin peut associer à ce comportement un objet `PersistentOptions` à partir duquel il sait reconstruire ce comportement.

C'est le BehaviorManager qui reçoit les PersistentOptions et qui doit en échange recréer et retourner les objets correspondants.

Si un plugin a des options générales à sérialiser (exemple le path d'un fichier di2 pour un plugin utilisant Papyrus), ces options doivent être placés dans une classe implémentant l'interface PersistentOptions et données au plugin grâce aux méthodes du BehaviorManager.

C'est le run configuration qui demande et rythme la sérialisation/dé-sérialisation du module.

6.1.1. Les classes PersistentEntity

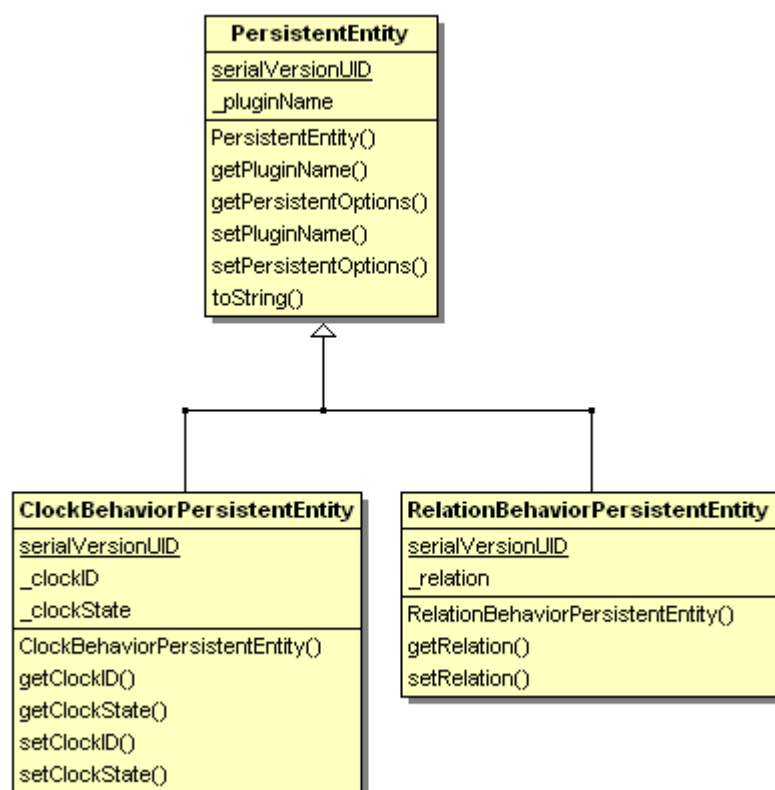


Illustration 17 : Les classes PersistentEntity.

Contrairement aux options générales des plugins de sortie sous forme de PersistentOptions, les entités du modèle ne peuvent être sérialisées telles quelles.

Je rappelle que les entités sont les briques de base du modèle, celles-ci contiennent toutes les informations concernant un comportement. Outre les options du

comportement qui sont fournies par le plugin sous forme de `PersistentOptions` lors de l'ajout d'un comportement, le reste des informations doit aussi pouvoir être sérialisé (l'horloge, l'état d'activation défini ...).

C'est le rôle du `PersistentEntity` qui est une représentation sérialisable d'une `Entity` du modèle. Cette classe est obtenue à partir des classes filles dérivant de `Entity`. En effet, les classes `ClockBehaviorEntity` et `RelationBehaviorEntity` définissent toutes les deux une fonction *`transformEntityIntoPersistentEntity()`* qui crée et retourne un objet de type `PersistentEntity`.



Si de nouveaux comportements apparaissent, en plus de la création de classes filles `Entity`, il faut aussi leur associer des `PersistentEntity` ainsi que la manière de passer de l'un à l'autre.

6.1.1. Les classes `OptionsSerializer`

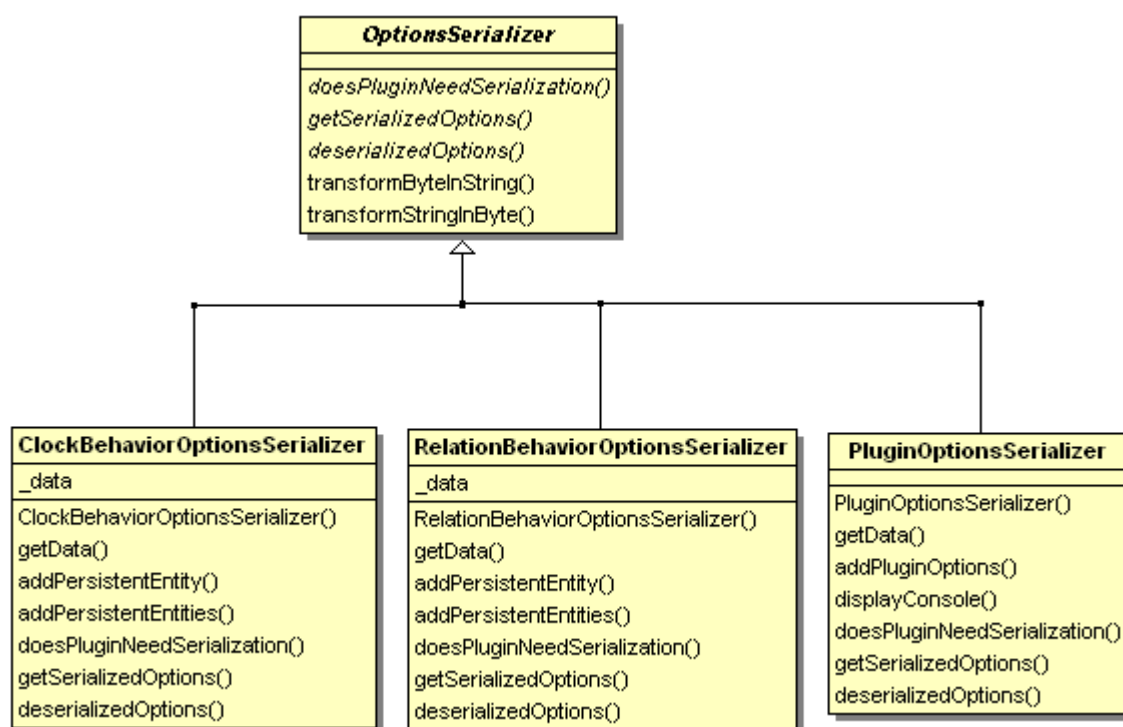
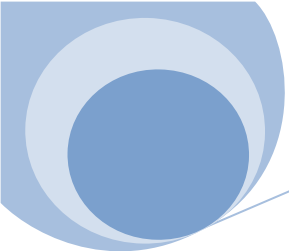


Illustration 18 : Les classes `OptionsSerializer`.



Tous comme des conteneurs ont été définis pour les entités dans le chapitre [4.2.2. Les « conteneurs » d'Entity](#), les `PersistentEntity` ont aussi des conteneurs. Ceux-ci sont les `OptionsSerializer`.

Ces objets contiennent des objets sérialisables dans une structure de données et se chargent de la sérialisation et la dé-sérialisation de cette structure en `String`.

On notera que le `ClockBehaviorOptionsSerializer` ainsi que le `RelationBehaviorOptionsSerializer` sont des conteneurs de `PersistentEntity` alors que le `PluginOptionsSerializer` est un conteneur de `PersistentOptions` qui sont les options générales des plugins de sortie.



En cas d'ajouts de comportement qui entraînent l'ajout de `PersistentEntity`, il faut aussi créer une structure de sérialisation/dé-sérialisation dérivant de `OptionsSerializer`.



La sérialisation utilisée est binaire puis le résultat est encodé en Base64 afin d'obtenir une `String`.

6.1.2. La classe `XMLStringMaker`

Pour intégrer le système de gestion de comportements au run configuration d'Eclipse, nous devons assurer une sérialisation sous forme de `String`.

Les classes `OptionsSerializer` nous rendent déjà des objets sérialisés sous forme de `String`. Il faut maintenant les assembler et les structurer.

Pour cela, le format que nous avons choisi est une `String` contenant du XML.

```
<root>
  <pluginName name = « Nom du plugin 1 » >
    <pluginOptions>
      String retournée par le PluginOptionsSerializer ;
    </pluginOptions>
    <clockBehaviorOptions>
      String retournée par le
      ClockBehaviorOptionsSerializer ;
```



```

        </clockBehaviorOptions>
        <relationBehaviorOptions>
            String retournée par le
            RelationBehaviorOptionsSerializer ;
        </relationBehaviorOptions>
    </pluginName>

    <pluginName name = « Nom du plugin 2 » >
        ...
    </pluginName>
    ...
</root>

```

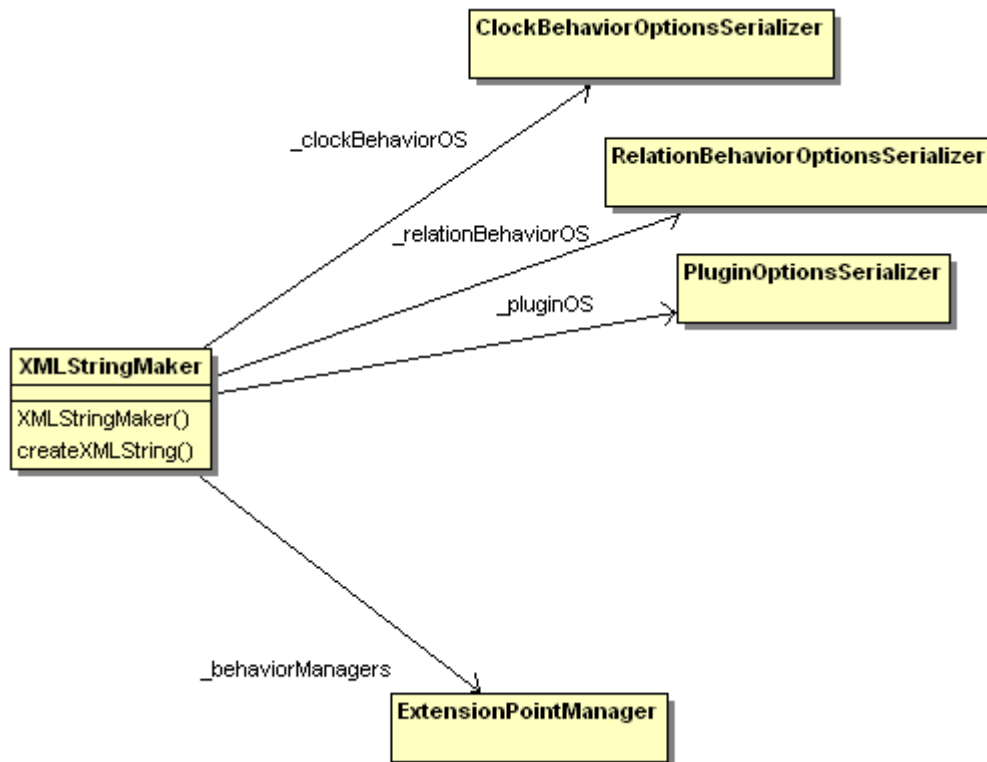


Illustration 19 : La classe `XMLStringMaker`.

Cette String contenant du XML est obtenue grâce à l'objet `XMLStringMaker`. Celui-ci s'aide des différents `OptionsSerializer` afin d'obtenir la sérialisation des objets sous forme de String ainsi que la classe qui gère le point d'extension `ExtensionPointManager` pour avoir la liste des plugins actuellement présents.

6.1.3. Une phase de sérialisation

Les illustrations suivantes détaillent une phase de sérialisation complète sous forme de diagrammes de séquence :

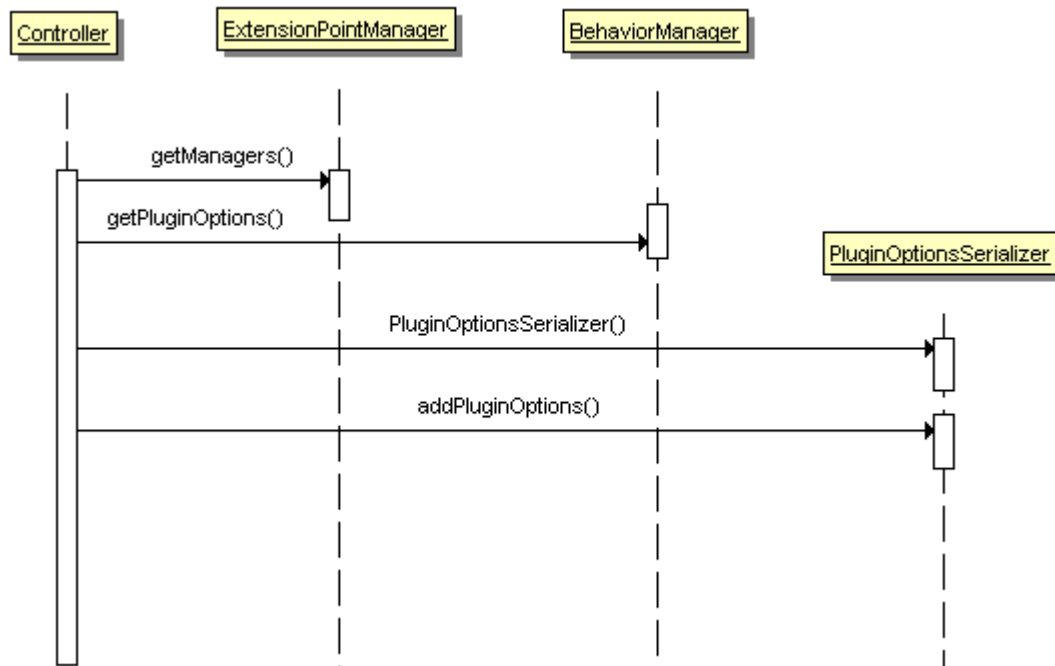


Illustration 20 : Diagramme de séquence d'une phase de sérialisation 1.

Le Controller récupère la liste des BehaviorManager connectés au point d'extension. Puis il appelle leur méthode *getPluginOptions()* qui retourne un PersistentOptions. Ces PersistentOptions sont ensuite donnés à un PluginOptionsSerializer qui est créé.

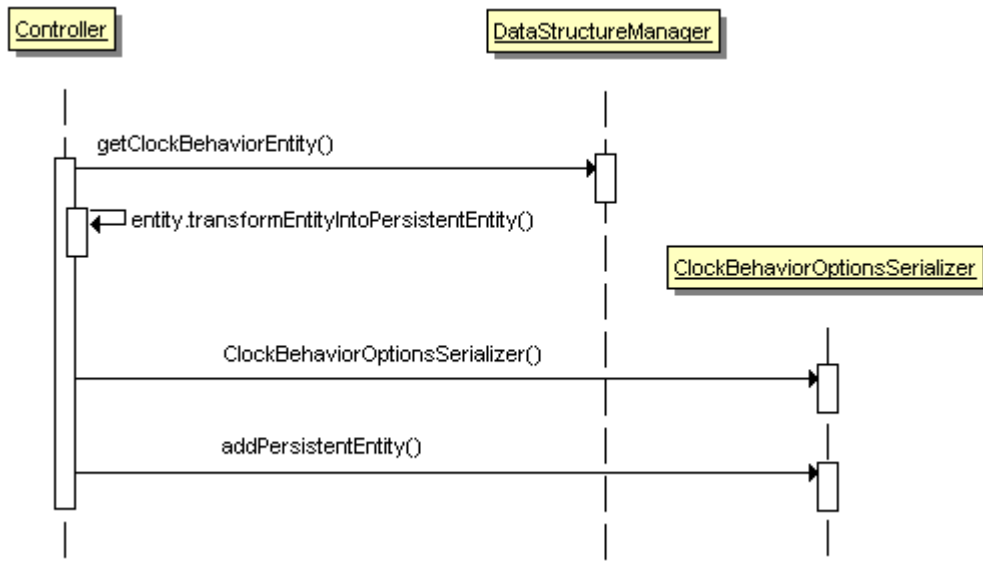


Illustration 21 : Diagramme de séquence d'une phase de sérialisation 2.

Le Controller va maintenant récupérer les ClockBehaviorPersistentEntity correspondant aux ClockBehaviorEntity. Le modèle (DataStructureManager) retourne au Controller la liste de ClockBehaviorEntity qu'il connaît. Le Controller les transforme en ClockBehaviorPersistentEntity puis les enregistre auprès d'un ClockBehaviorOptionsSerializer qui est créé.

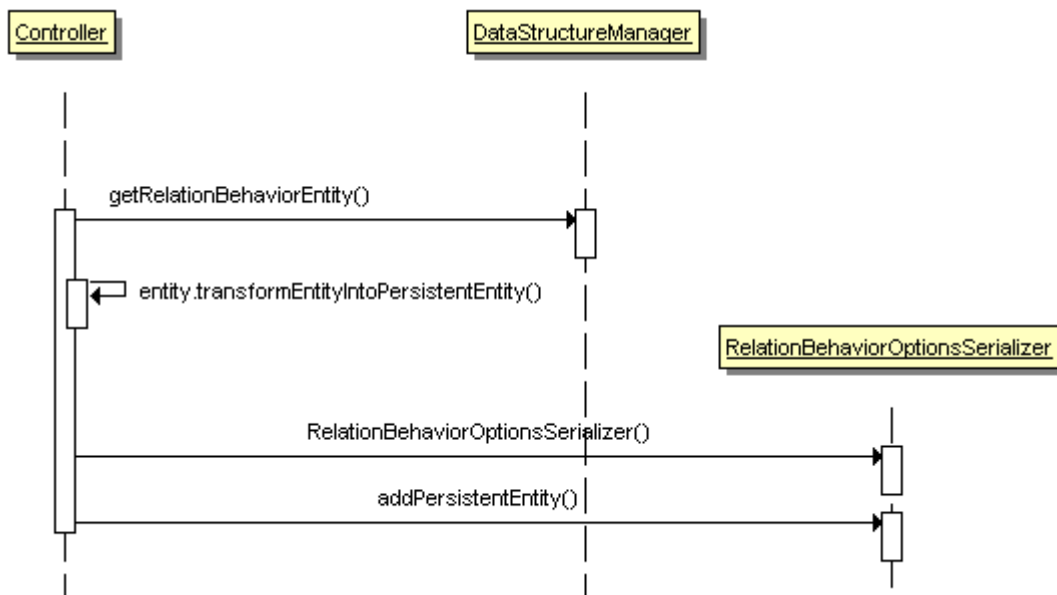


Illustration 22 : Diagramme de séquence d'une phase de sérialisation 3.

Il en est de même pour les `RelationBehaviorEntity`.

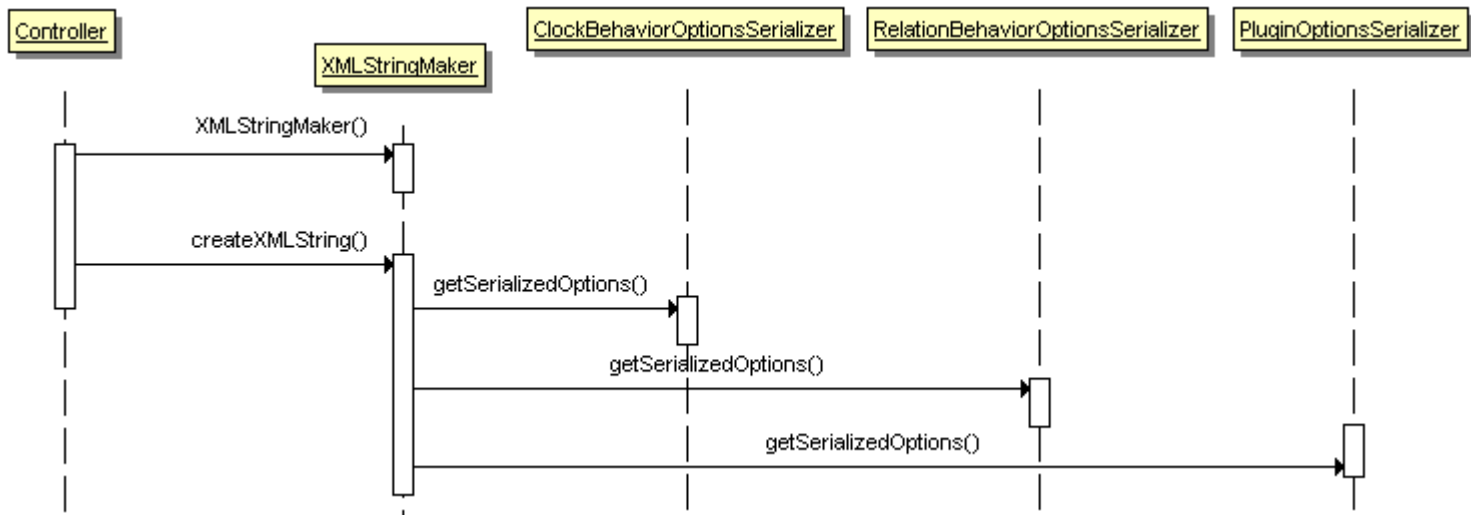


Illustration 23 : Diagramme de séquence d'une phase de sérialisation 4.

Enfin, le `Controller` crée un `XMLStringMaker` en lui donnant en argument les trois objets `OptionsSérializer` ainsi que les plugins présents au point d'extension (n'est pas représenté ici par manque de place). Le `XMLStringMaker` se charge ensuite de créer la `String` de type XML en utilisant les fonctions de sérialisation des `OptionsSerializer`.

6.2. La dé-sérialisation

6.2.1. Les classes optionsDealer

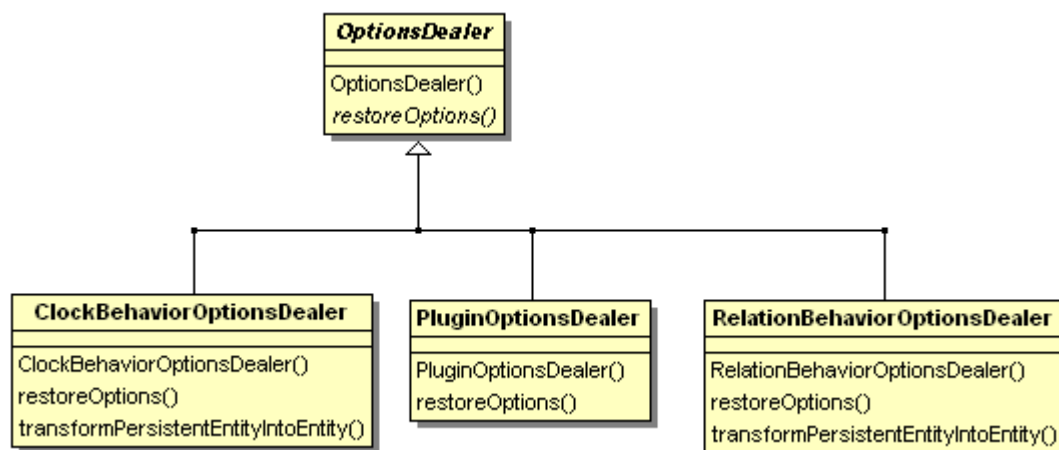


Illustration 24 : Les classes OptionsDealer.

Revenons au package **fr.inria.aoste.behavior.controller**. A l'intérieur de celui-ci se trouve un sous package **fr.inria.aoste.behavior.controller.dealers**. Je rappelle que les classes contenues dans ces packages sont les seules à interagir directement avec les BehaviorManager des plugins de sortie.

Les OptionsDealer sont les objets qui recréent des Entity à partir de PersistentEntity. Ils ont donc besoin d'interagir avec les BehaviorManager afin de leur rendre les objets PersistentOptions précédemment sérialisés.

Les OptionsDealer récupèrent les options et entités sérialisés à partir des OptionsSerializer. Ils se servent ensuite du Controller pour ajouter les nouveaux objets Entity.

6.2.2. La classe XMLStringParser

En phase de dé-sérialisation, le but est d'analyser la String contenant du XML et l'objet XMLStringParser se charge de cette tâche.

Le XMLStringParser prend les mêmes arguments en constructeur que le XMLStringMaker (les OptionsSerializer et l'ExtensionPointManager qui donne les BehaviorManager présents au point d'extension).

Le XMLStringParser parcourt la String XML et se sert des OptionsSerializer pour dé-sérialiser les options qui se recréent et se placent dans les structures de données des OptionsSerializer.

6.2.3. Une phase de dé-sérialisation

Les illustrations suivantes détaillent une phase de dé-sérialisation complète sous forme de diagrammes de séquence :

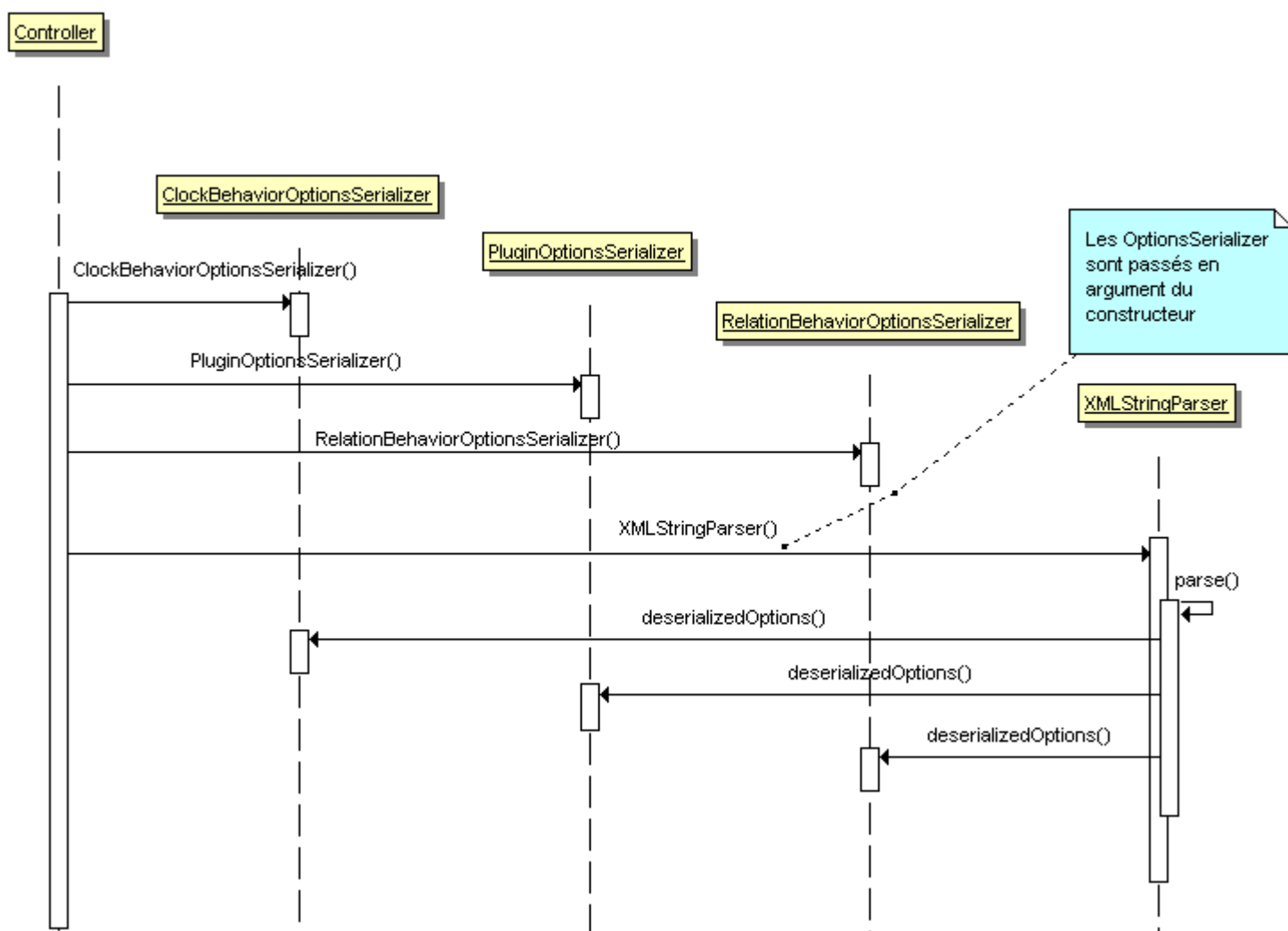


Illustration 25 : Diagramme de séquence d'une phase de dé-sérialisation 1.

Le Controller instancie les OptionsSerializer, puis le XMLStringParser en lui donnant les OptionsSerializer et l'ExtensionPointManager (n'est pas

représenté dans le schéma par manque de place). Le XMLStringParser traite la String XML et remplit les OptionsSerializer.

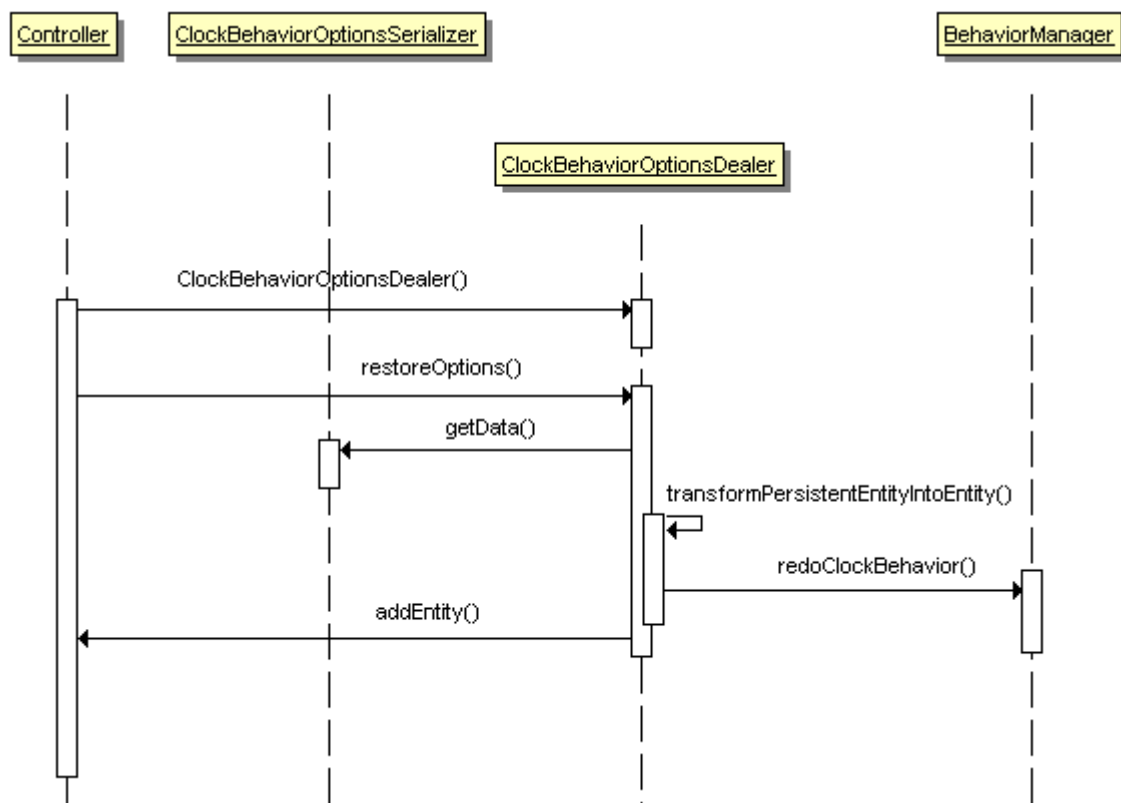


Illustration 26 : Diagramme de séquence d'une phase de dé-sérialisation 2.

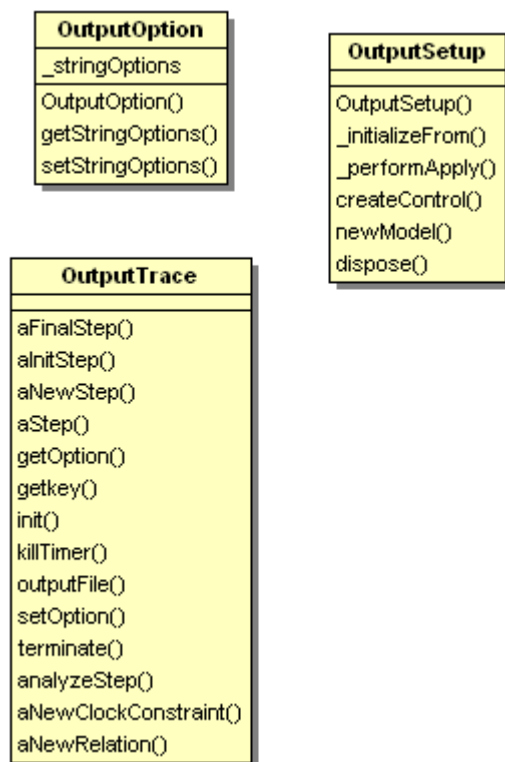
L'illustration 26 montre l'instanciation d'un ClockBehaviorOptionsDealer et la recréation des entités pour des comportements sur des états d'horloge.

La manière de faire pour les options générales des plugins et les comportements sur les relations comportent peu de différence avec ce scénario.

6.3. La liaison avec le simulateur de TimeSquare

Le plugin de gestion des comportements est connecté au simulateur **fr.inria.aoste.ccsImodel.launcher.core** par l'intermédiaire de son point d'extension **fr.inria.aoste.ccsImodel.launcher.core.outputformat**.

Les classes assurant cette liaison sont contenues dans le package : **fr.inria.aoste.behavior.launcher.extensionpoint**.




*Illustration 27 : Les classes du package
fr.inria.aoste.behavior.launcher.extensionpoint*

6.3.1. Le run configuration

La classe `OutputSetup` assure la liaison avec le run configuration d'Eclipse. Elle implémente l'interface `OutputSetupOption` donnée par le simulateur.

L'interface fournit le Composite SWT dans lequel la vue du MVC va pouvoir s'afficher.

La sérialisation et dé-sérialisation sous forme de String est prise en compte dans cette classe.



Le simulateur procure au plugin de gestion des comportements le modèle CCSL que l'utilisateur a choisi dans le run configuration ainsi que les horloges définies dans ce modèle.



Le plugin ne peut pas fonctionner si le simulateur ne lui fournit pas le modèle CCSL et ses horloges.

Enfin, le plugin de gestion des comportements doit être capable d'avertir le simulateur qu'un changement s'est produit afin que le Launcher UI soit averti et que les boutons Apply et Revert du run configuration se dégrisent.

Pour cela, le `Controller` prend aussi une référence sur la classe `OutputSetup` qu'il va notifier dès qu'un changement est effectué.

Le `Controller` étant un singleton, il est récupéré dans cette classe et ces options lui sont données : le modèle CCSL sélectionné, les horloges du modèle et une référence sur l'objet `OutputSetup`.

6.3.2. La phase d'exécution de TimeSquare

Le simulateur fournit l'interface `IOutputTrace` qui est utilisée durant la phase d'exécution de TimeSquare.

La classe implémentant cette interface est `OutputTrace`. L'interface donne au plugin le modèle de Trace et le modèle de Relations.

Le `Controller` est récupéré en tant que singleton dans cette classe. Lorsqu'un nouveau pas de simulation est généré, la classe `OutputTrace` récupère le modèle de Trace, parcourt toutes les horloges, transforme leur état en `ClockActivationState` et informe le `Controller`.

Il en est de même pour le modèle de Relations qui est récupéré par la classe `OutputTrace` lorsqu'une relation entre instants est détectée.

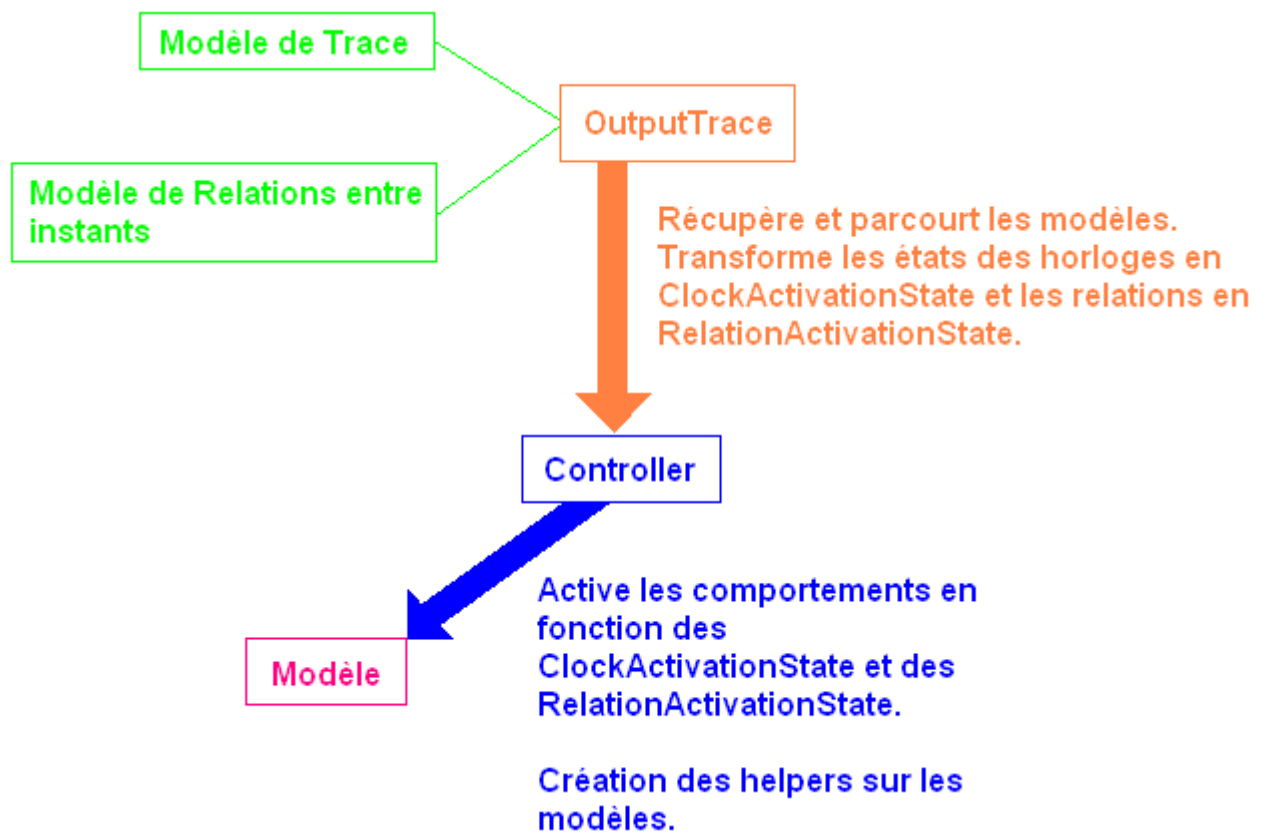


Illustration 28 : Schéma d'exécution de TimeSquare.

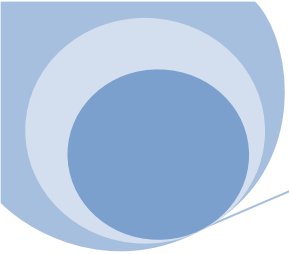
6.3.3. La classe `OutputOption`

La classe `OutputOption` implémente l'interface `IOutputOption` donnée par le simulateur. Cette classe est une classe de stockage d'options utilisée en interne par le simulateur. Elle sert de liaison entre l'affichage du run configuration et l'exécution d'une configuration déjà prédéfinie.

Pour ce plugin, nous enregistrons dans cette classe la String XML ainsi que le modèle CCSL utilisé et ses horloges.

7. Liaison avec les modèles CCSL

Ce chapitre traite les liens entre le plugin de gestion des comportements avec les modèles CCSL.



Nous rappelons qu'il existe actuellement deux modèles :

- **fr.inria.aoste.marte.ccslmodel**
- **fr.inria.aoste.umlccslmodel**

La distinction entre ces deux modèles est nécessaire à un seul endroit dans le plugin de gestion des comportements :

La classe `ClockEntity` du package **fr.inria.aoste.behavior.visible**. Cette classe représente une horloge du modèle CCSL.

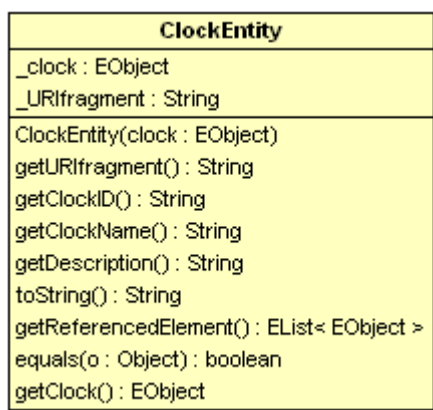


Illustration 29 : La classe ClockEntity.

Nous rappelons que les horloges du modèle CCSL sont fournies par le simulateur dans les classes `OutputOptions` et `OutputSetup`. Ces horloges sont données dans une liste de `EObject` `Ecore`.

Afin d'obtenir le nom des horloges, il faut que la classe `ClockEntity` connaisse le type exacte de l'horloge.

Il est à noter que la sérialisation d'une classe `ClockEntity` utilise les xmi ID des éléments des modèles `Ecore`.



8. Améliorations possibles

8.1. Meilleure synchronisation de la phase de configuration

Actuellement lors de la phase de configuration, le singleton `Controller` est créé ou récupéré avec son modèle. Les plugins de sortie lui donnent ensuite des comportements et leur version sérialisable (`PersistentOptions`).

A la fin de la phase de configuration, lorsque l'utilisateur clique sur le bouton Run du run configuration d'Eclipse, le singleton est détruit (cf. la classe `OutputSetup`).

En phase d'exécution (cf. la classe `OutputTrace`), le `Controller` est donc recréé pour deux raisons :

- il a été détruit en fin de phase de configuration ;
- on a lancé une exécution sans passer par le run configuration graphique.

Les comportements et les options générales des plugins sont donc dé-sérialisés et l'exécution se poursuivra.

Il résulte de cela que **les comportements ne sont pas utiles en phase de configuration** (puisque toujours recrées après). Le plugin peut ne donner que des `PersistentOptions`. Ceci devraient s'inscrire seuls et devraient être affichés à la place des `Behaviors` dans la vue.

8.2. Modifications des comportements

Après le premier changement, une modification des comportements du modèle implique en fait une modification des `PersistentOptions`. Ces modifications ne peuvent se faire qu'avec l'aide du plugin, en effet le module de gestions des comportements ne connaît un `PersistentOptions` que par son interface, seul le plugin sait ce qu'il y a dedans et la manière de le changer.

- Les `Behaviors` doivent avoir un booléen indiquant si il est modifiable ou non ;
- Une méthode doit être rajoutée au `BehaviorManager` pour la modification d'un comportement ;
- Cette méthode doit être accompagnée d'un helper ainsi qu'une aide à l'interface graphique (comme pour le `BehaviorManagerGUI`).



8.3. Utiliser l'ID du point d'extension

Dans la version actuelle du module, L'ID qui est utilisé pour gérer les `BehaviorManager` est la String retournée par la méthode `getPluginName()`. Ceci doit être remplacé à terme par l'ID demandé dans le point d'extension : **fr.inria.aoste.behavior.behaviormanager**.

8.4. Le mode Debug

Pour faire tourner le module de gestions des comportements en mode Debug d'Eclipse, il faut implémenter plusieurs méthodes de la classe `OutputTrace` et les répercuter sur les `BehaviorManager` des plugins. Ceux-ci auront donc le choix de se voir exécuter en mode pas à pas du Debug ou non.

8.5. Nouveau helper pour *aNextStep()*

Lors de l'exécution, le `Controller` indique aux `BehaviorManager` la présence d'un nouveau pas de simulation via la méthode `aNextStep()`. Il faut rajouter en argument de cette méthode un nouveau helper sur le Trace.

8.6. Ajout de comportements

Au même titre que pour les états d'horloge et les relations entre instants, il peut être intéressant de pouvoir ajouter du comportement sur une phase d'initialisation `beforeExecution()`, un nouveau pas de simulation `aNewStep()` et la à la fin `end()`.

8.7. D'un point de vue programmation

Quelques points qui peuvent être améliorés :

- La vue a actuellement connaissance du modèle
- Le design pattern Visitor peut être utile lors de l'exécution des comportements sur la structure de données (modèle du MVCs).