Installation d'un Plugin de comportements sur Time Square

Ce document explique comment ajouter un plugin de comportements au MDK Time Square basé sur Eclipse.

Lengellé Denis

28/08/2009

Version 1.30

Table des matières

I. INTRODUCTION

Ce rapport explique comment enrichir TimeSquare d'un plugin définissant des comportements.

Actuellement, il est possible d'ajouter des comportements sur des états d'horloges ainsi que sur des relations entre instants (cf. fr/inria/aoste/umlrelationmodel/rapport/Rapport_LP_SIL_2009_Anthony_Gaudino.pdf).

Ce document explique dans l'ordre les concepts importants nécessaires à la compréhension du plugin de gestion de comportements auquel doit se connecter le plugin à ajouter.

II. ARCHITECTURE GENERALE

La possibilité d'ajouter des comportements dans TimeSquare se base sur le plugin de gestion de comportements **fr.inria.aoste.behavior**.

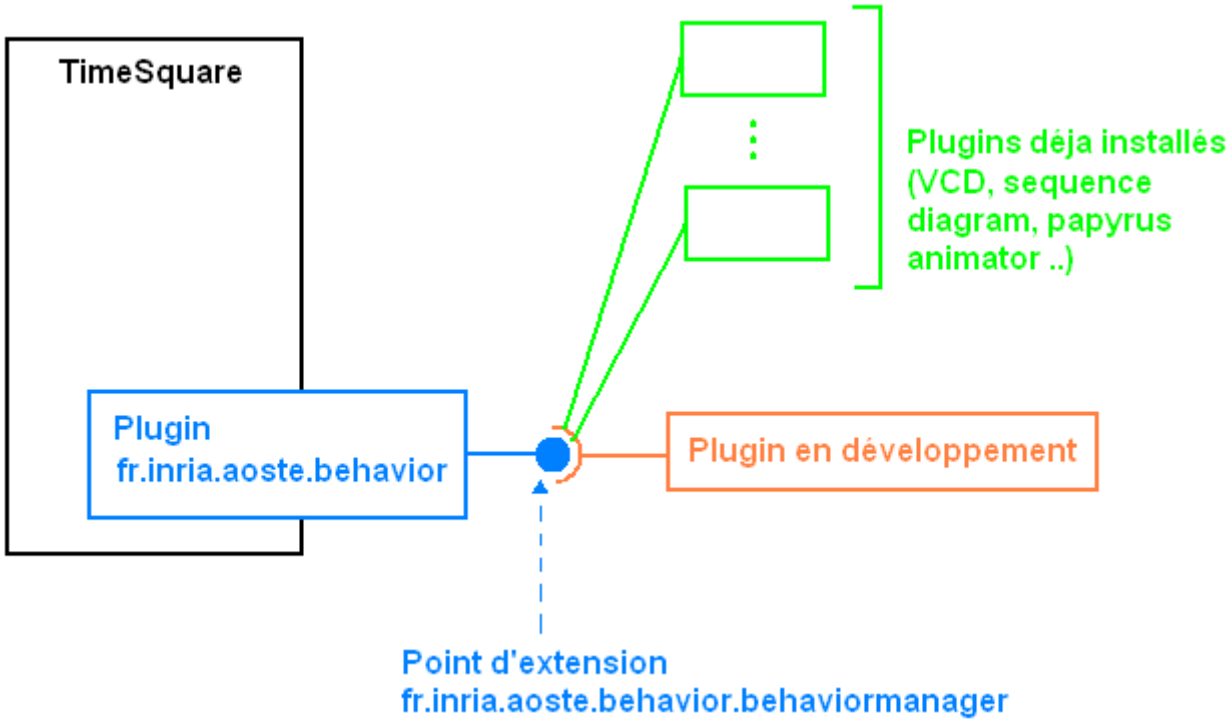

Illustration 1 : Connexion de plugins de comportements à TimeSquare.

Ce plugin offre un point d'extension et un package de classes : **fr.inria.aoste.behavior.visible**. Le protocole de communication entre le plugin de gestion de comportements **fr.inria.aoste.behavior** et le plugin que vous développez se fait par le biais de ces classes. Le but étant à terme de bien en comprendre le rôle et de les implémenter correctement.

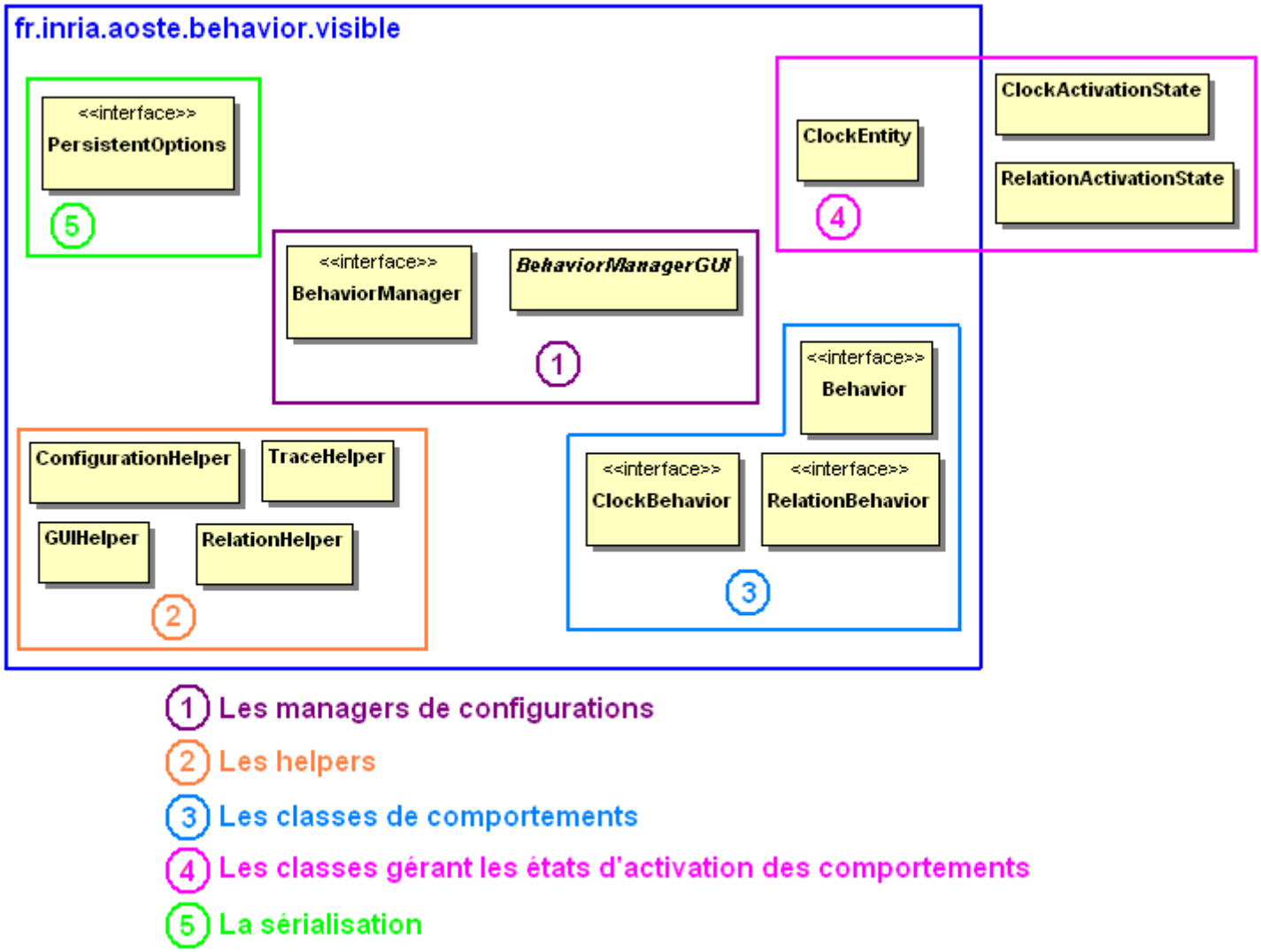

Illustration 2 : Les classes et interfaces nécessaires.

1 Les classes BehaviorManager et BehaviorManagerGUI ont un rôle central et sont demandées à la connexion du plugin sur le point d'extension. Ceci est détaillé dans le chapitre III. Mise en place du plugin.

La classe BehaviorManager est instanciée directement par **fr.inria.aoste.behavior.** C'est par l'intermédiaire de cette classe que vous pouvez configurer et enregistrer les comportements que vous développez. Toutes les opérations devant s'effectuer entre votre plugin et le plugin de configuration passent par cette classe. Dans un ordre de développement classique, le BehaviorManager étant la dernière classe à devoir être implémentée est détaillée à la fin de ce rapport au chapitre IV.4) Le BehaviorManager

(2) Afin d'interagir avec TimeSquare que ce soit pendant la phase de configuration ou bien la phase d'exécution, des objets helpers (cf. Illustration 2) vous sont donnés afin de vous aider à effectuer de nombreuses opérations. Ces helpers sont détaillés tout au long de ce rapport selon leur domaine d'application.

(3) Le plugin **fr.inria.aoste.behavior** a pour but de permettre à votre plugin d'activer des comportements sur certains événements durant la phase d'exécution de TimeSquare. La notion de comportement et les interfaces à implémenter (cf. Illustration 2) sont détaillées dans le chapitre IV.1) Les comportements.

(4) Lors de la configuration et de l'enregistrement de vos comportements, vous devez indiquer au plugin **fr.inria.aoste.behavior** quand est ce que ceux-ci doivent être actifs. Pour cela, un système permettant de spécifier finement quand un comportement doit être déclenché a été mis en place et détaillé dans le chapitre IV.2) Le système d'état d'activation d'un comportement de ce rapport.

(5) Enfin, le plugin de gestion de comportements est lié au run configuration d'Eclipse, pour cela une gestion de la sérialisation des comportements est nécessaire et est expliquée au chapitre IV.3) Les PersistentOptions.

III. MISE EN PLACE DU PLUGIN

1) Création d'un plugin de développement

Pour se connecter au plugin de comportement, il vous faut créer un projet de développement de plugin Eclipse.

- Cliquez sur File -> New -> Other…
- Sélectionnez dans la liste proposée : Plug-in Project

- Cliquez sur Next
- Entrez un nom de projet.
- Si vous souhaitez un simple plugin de développement sans utiliser des templates préconfigurés par Eclipse, cliquez sur Finish.

Note : Si vous souhaitez utiliser des options avancées de configuration de plugin de développement, referez vous aux tutoriaux fournis par Eclipse : http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html

2) Ajout de dépendances

Il faut que le plugin que vous développez ait une dépendance sur le plugin de comportements.

Pour cela :

- Sélectionnez le fichier MANIFEST.MF du répertoire META-INF du projet de développement de plugin Eclipse.
- Cliquez sur l'onglet Dependencies.
- Appuyer sur le bouton Add… puis entrez : **fr.inria.aoste.behavior**.
- Validez sur ok.

Maintenant, il existe une dépendance entre votre plugin et le plugin de comportements. Vous avez accès au package **fr.inria.aoste.behavior.visible** (cf. Illustration 2) qui contient les classes permettant de se connecter au point d'extension et de définir des comportements.

3) Connexion au point d'extension

Il faut maintenant se connecter au point d'extension offert par le plugin de comportements : **fr.inria.aoste.behavior.behaviormanager**.

- Sélectionnez le fichier MANIFEST.MF du répertoire META-INF du projet de développement de plugin.
- Cliquez sur l'onglet Extensions.
- Appuyer sur le bouton Add… puis entrez : **fr.inria.aoste.behavior.behaviormanager**.
- A ce moment, on vous demande de remplir 3 champs :

Le premier est une chaîne de caractères définissant un ID pour votre plugin. Il doit être différent de ceux des plugins déjà connectés, entrez donc par exemple le nom complet de votre plugin.

Le deuxième est une classe implémentant une interface nommée BehaviorManager et se trouvant dans le package **fr.inria.aoste.behavior.visible**. Cette classe est une des plus importantes étant donné qu'elle assurera le lien entre le plugin en développement et le plugin de comportements.

Le troisième champ est optionnel, il vous permet d'utiliser un service de base d'affichage graphique pour vôtre plugin. Si vous voulez que l'utilisateur personnalise les comportements à ajouter via un pannel graphique, il vous suffit de créer une classe implémentant l'interface BehaviorManagerGUI du package **fr.inria.aoste.behavior.visible**.

Note : Pour ces deux champs, utilisez l'aide fourni par Eclipse lors de la connexion à un point d'extension en cliquant sur le nom des champs. Une boite de création de classes s'ouvre et vous pouvez ainsi directement créer une classe qui étend la bonne interface. Les explications quand à l'implémentation de ces interfaces sont données aux chapitres IV.4) Le BehaviorManager et IV.5) Le BehaviorManagerGUI de ce rapport.

- Validez sur ok.

Arrivé à ce stade, vous êtes relié au plugin de gestion de comportements et vous disposez de :

- une classe implémentant BehaviorManager ;
- une classe implémentant BehaviorManagerGUI (optionnelle).

IV. DETAIL DES CLASSES A IMPLEMENTER

Les classes BehaviorManager et BehaviorManagerGUI permettent la relation entre votre plugin et le plugin de gestion de comportements. Par leur biais, vous pourrez associer des comportements à des événements de TimeSquare (un état d'horloge ou bien une relation entre instants).

Avant d'expliquer comment remplir ces classes, nous allons voir en détail des concepts importants comme les comportements eux même, la notion d'état d'activation, ainsi que leur sérialisation.

Nous reviendrons enfin sur le BehaviorManager et le BehaviorManagerGUI et la manière d'intégrer vôtre plugin à TimeSquare.

1) Les comportements

Dans cette section nous allons détailler ce qu'est un comportement. Actuellement il est possible d'ajouter des comportements sur l'état d'une horloge ainsi que sur une relation entre instants.

Ces deux types de comportements sont différents et deux classes sont à utiliser situées dans le package **fr.inria.aoste.behavior.visible**.

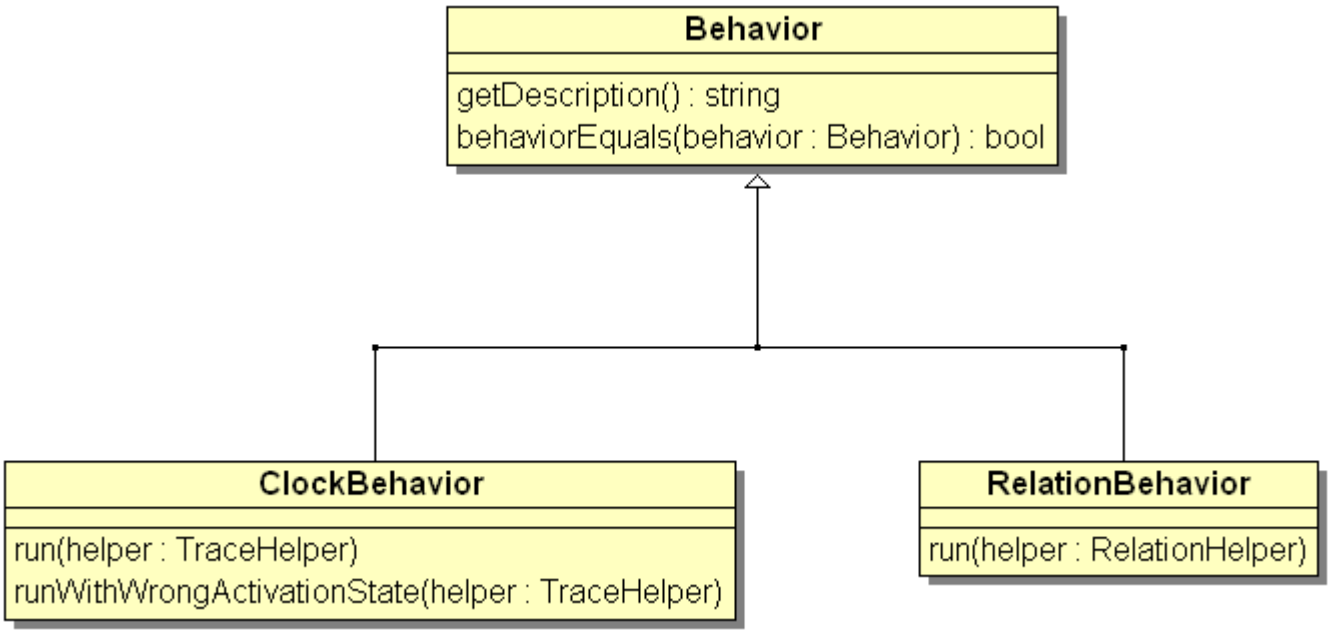

Illustration 3 : Hiérarchie des classes de comportements.

a) L'interface Behavior

Commune à tous les comportements, cette interface force la redéfinition de deux méthodes :

- String getDescription()

Doit retourner une description du comportement sous forme de texte. Ce texte sera affiché dans le run configuration d'Eclipse.

- boolean behaviorEquals(Behavior behavior)

Redéfinition de la méthode equals de Object. Cette fonction est très importante car elle est utilisée lors de l'ajout/suppression de comportements dans le run configuration. Cette méthode doit retourner la valeur « true » si l'objet de comportement sur lequel elle s'applique est équivalent à l'objet de comportement passé en paramètre. Sinon, la valeur « false » doit être retournée.

b) L'interface ClockBehavior

Cette interface est utilisée pour définir un comportement sur l'état d'une horloge.

- void run(TraceHelper helper)

Cette méthode est appelée durant la simulation lorsque le comportement est activé.

Cette méthode introduit la notion de Helper qui est utilisée à plusieurs reprises dans le plugin de comportements.

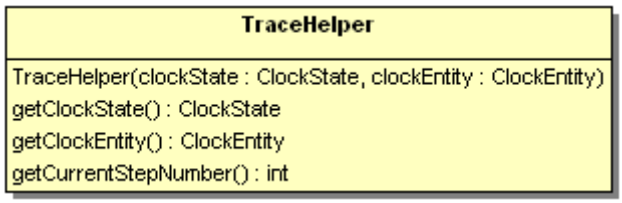

Illustration 4 : La classe TraceHelper.

Lorsque cette méthode est exécutée, un helper sur la trace (cf. fr.inria.aoste.marte.ccsl.treetrace) lui sera donné en argument. Via ce helper, vous pouvez simplement récupérer l'horloge sur laquelle ce comportement est associé ainsi que son état actuel. Vous avez de la même manière accès à l'étape courant de simulation.

- void runWithWrongActivation(TraceHelper helper)

Cette méthode sera appelée durant la simulation lorsque l'horloge associée à ce comportement n'est pas dans un état correspondant à l'état d'activation défini par le plugin ou l'utilisateur.

c) L'interface RelationBehavior

Cette interface est utilisée pour définir un comportement sur une relation d'instants.

Durant l'exécution, lorsque la relation associée au RelationBehavior par le plugin ou l'utilisateur est détectée, la méthode :

- void run(RelationHelper helper)

est exécutée et un helper sur le modèle de Relation lui sera donné (cf. fr.inria.aoste.umlrelationmodel).

Via ce helper, vous pouvez récupérer les instants sur lesquels a porté la relation.

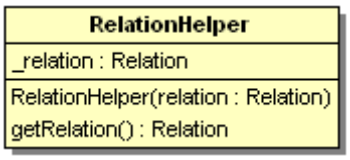

Illustration 5 : La classe RelationHelper.

2) Le système d'état d'activation d'un comportement

Ceci est un point des plus importants. Un comportement doit être associé à quelque chose qui permet de déterminer dans quel cas on veut le déclencher.

Dans le cas d'un comportement sur un état d'horloge, il faudra donc associer ce comportement à une horloge (ClockEntity) et à un ClockActivationState.

Dans le cas d'un comportement sur une relation d'instants, il faudra associer ce comportement à un RelationActivationState.

Comme on peut l'observer dans l'illustration 2 de ce document, les classes ClockActivationState et RelationActivationState ne sont pas dans le package **fr.inria.aoste.behavior.visible**.

a) La classe ClockEntity

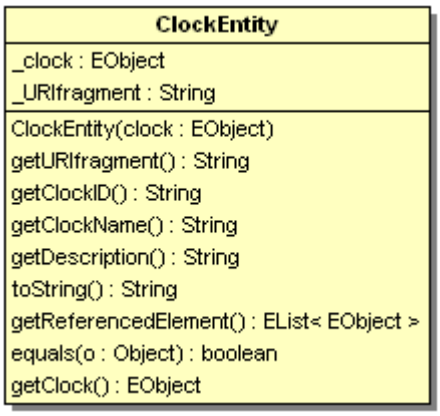

Illustration 6 : La classe ClockEntity.

Cette classe contient l'horloge des modèles CCSL. On peut la récupérer sous la forme d'un EObject grâce à la méthode getClock().

De plus, on peut récupérer les objets UML qui lui sont associés structurellement via la méthode getReferencedElement().

b) La classe ClockActivationState

Cette classe exprime sur quels états d'une horloge on souhaite que le comportement s'exécute.

Cette classe n'est pas accessible car elle n'est pas dans le package **fr.inria.aoste.behavior.visible**.

Les seules manières de récupérer des instances de ces classes sont :

- en demander une « pré construite » au ConfigurationhHelper ;
- utiliser le constructeur qui prend en paramètre un tableau de booléens à partir du ConfigurationHelper.

Nous étudierons le ConfigurationHelper plus en détail dans la partie concernant le BehaviorManager.

La classe ClockActivationState n'a pas été mise à disposition des plugins directement car il s'agit en fait d'un simple tableau de booléens dont la taille et le format peuvent être récupérés par un helper.

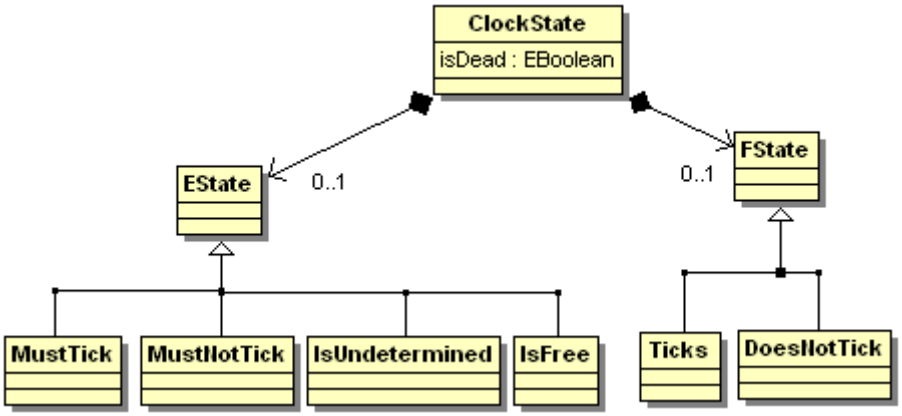

Illustration 7 : Rappel des états possibles d'une horloge dans la trace.

Les états EState et FState qui se trouvent dans la trace sont représentés par un simple tableau de booléens dans la classe ClockActivationState.

En voici le format :

```
public static String[] stateList =
        {
                "ticks",
                "doesntTick",
                "mustTick",
                "mustNotTick",
                "isFree",
                "isUndetermined",
                "isDead"
        };
```

Il suffit de mettre la valeur « true » dans les états souhaités.

Exemple :

Un comportement doit s'activer lorsqu'une horloge « tick ».

La trace est définie telle qu'une horloge qui tick peut avoir les états :

EState	FState
MustTick	Ticks
IsFree	Ticks
IsUndetermined	Ticks

Le tableau de booléens permettant « d'attraper » tous les cas est donc :

[1010111]

Le dernier booléen indique si on veut le dernier tick ou pas.

Si l'on souhaite exécuter un comportement seulement quand une clock doit « ticker », il faut alors construire le tableau suivant :

[1010001]

Le fonctionnement interne de ce système est le suivant :

Lors de la simulation, un pas est crée. Le plugin de comportements parcours celui-ci à la recherche de tous les états qu'il transforme en ClockActivationState et donc en tableau de booléens. Ce tableau est ensuite comparé aux ClockActivationState (tableaux de booléens) associés aux comportements de cette manière :

Soit CTrace le ClockActivationState créé à partir de la trace.

Soit CDefined le ClockActivationState défini par les plugins ou utilisateurs.

Soit ^ l'opérateur Logical And.

CTrace ^ CDefined = CDefined

Si le résultat de ce test est « true » alors le comportement sera exécuté.

c) La classe RelationActivationState

Le fonctionnement est identique que celui des ClockActivationState.

Le format du tableau de booléens est le suivant :

```
public static String[] relationList =
{
    "coincidence",
    "disjoint",
    "packet",
    "strictPrecedence",
    "precedence"
};
```

Cette classe n'est pas accessible car elle n'est pas dans le package **fr.inria.aoste.behavior.visible**.

3) Les PersistentOptions

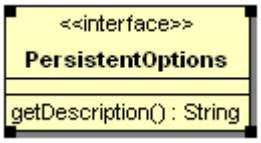

Illustration 8 : L'interface PersistentOptions.

Un des avantages du plugin de comportements et qu'il sauvegarde les comportements définis par les plugins et utilisateurs dans le run configuration d'Eclipse.

Il est important de comprendre que les classes implémentant les interfaces Behaviors devraient être persistantes mais ceci est une contrainte trop forte qui ne pourra bien entendu pas toujours être respectée. Pour pallier à ce problème, le plugin de comportements offre une interface PersistentOptions qui a pour unique rôle d'implémenter la classe Serializable de Java.

Cette interface ne possède qu'une méthode d'affichage. Le reste est laissé libre au plugin. Celui-ci doit être capable de recréer des instances de Behavior à partir d'objets PersistentOptions. C'est le BehaviorManager qui a la charge de recréer des comportements à partir de PersistentOptions.

La manière d'effectuer cette transformation est laissée libre au plugin.

De plus, si le plugin a besoin d'enregistrer des options générales, celles-ci doivent être données au plugin de comportements via cette interface dans la fonction *getPluginOptions()* du BehaviorManager. Ces options seront récupérées par la méthode *receivePluginOptions()*.

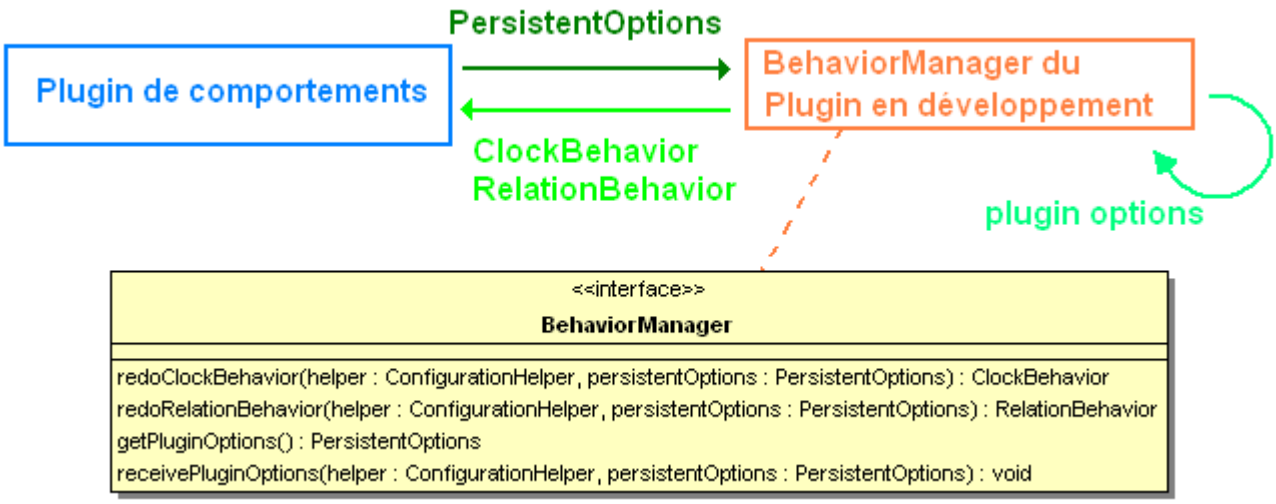

Illustration 9 : Phase de dé-sérialisation de PersistentOptions.

4) Le BehaviorManager

Nous revenons sur la classe qui va interagir directement avec le plugin de comportements. Le BehaviorManager va être utilisé pendant la phase de configuration des comportements dans le run configuration d'Eclipse.

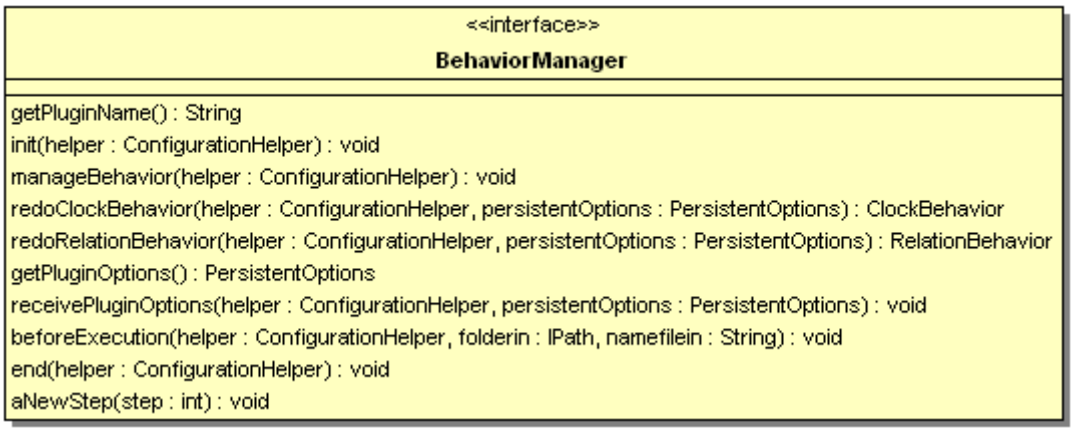

Illustration 10 : L'interface BehaviorManager.

Cette classe sera instanciée par le plugin de gestion de comportements, il est indispensable qu'elle contienne un constructeur sans paramètre. Sans ce constructeur, cette classe ne peut être instanciée via le point d'extension.

La première méthode à implémenter dans cette interface est :

- String getPluginName()

Ce String est utilisé comme un ID. Il est nécessaire qu'il soit différent de ceux des plugins déjà connectés au plugin de comportements.

Ceci fait, vous pouvez testez que le plugin est bien détecté en lançant un deuxième runtime Eclipse et sélectionnez le pannel graphique du run configuration.

Vous devriez voir apparaitre le nom de votre BehaviorManager dans la liste.

Voici l'ordre d'appel des fonctions du BehaviorManager :

<<Phase de configuration>>

- void init(ConfigurationHelper helper)

Cette méthode est appelée pendant la phase de configuration. Dans cette version du plugin de comportements, il est à noter que cette fonction est appelée plusieurs fois et pour tous les plugins présents.

- void manageBehavior(ConfigurationHelper helper)

Lorsque l'on clique sur le nom du BehaviorManager dans la liste du run configuration, cette méthode est appelée seulement si aucun BehaviorManagerGUI n'a été défini. Dans le cas contraire, c'est la classe qui étend BehaviorManagerGUI qui sera utilisée.

<<Phase d'execution>>

- void beforeExecution(ConfigurationHelper helper, IPath folderin, String namefilein)

Cette méthode est appelée lorsque l'utilisateur a cliqué sur le bouton run ou debug d'Eclipse. C'est la première méthode exécutée en début de simulation. En plus du helper, un répertoire pour placer des fichiers de sortie ainsi qu'un suffixe pour nom de fichier contenant le numéro de la simulation en cours sont fournis en argument.

- void aNewStep(int step)

Durant la simulation, dès qu'un nouveau pas est créé, cette méthode est appelée.

- void end(ConfigurationHelper helper)

Cette méthode est appelée en fin de simulation.

<<Fin>>

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).

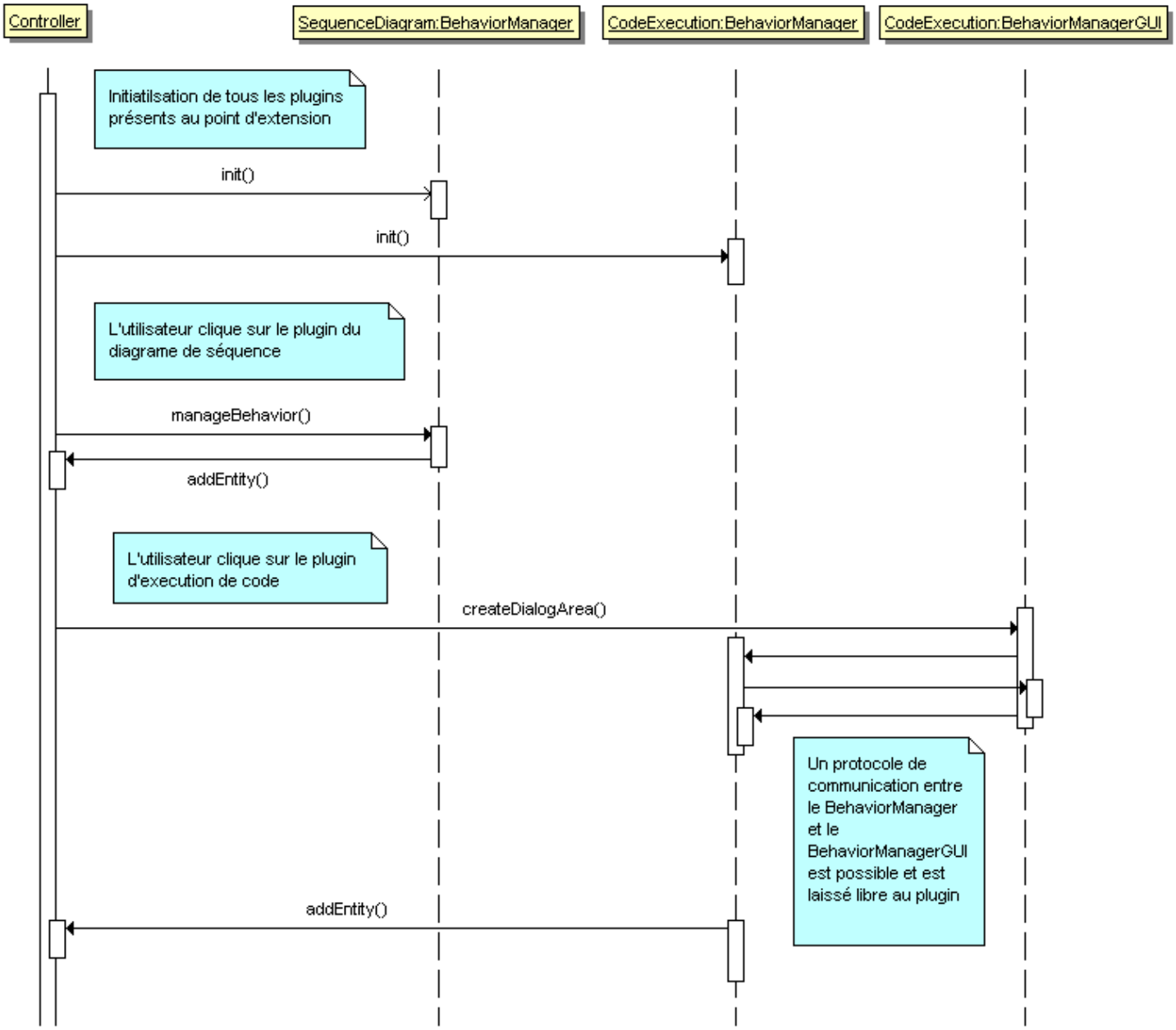

Illustration 11 : Phase de configuration.

L'illustration 11 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.

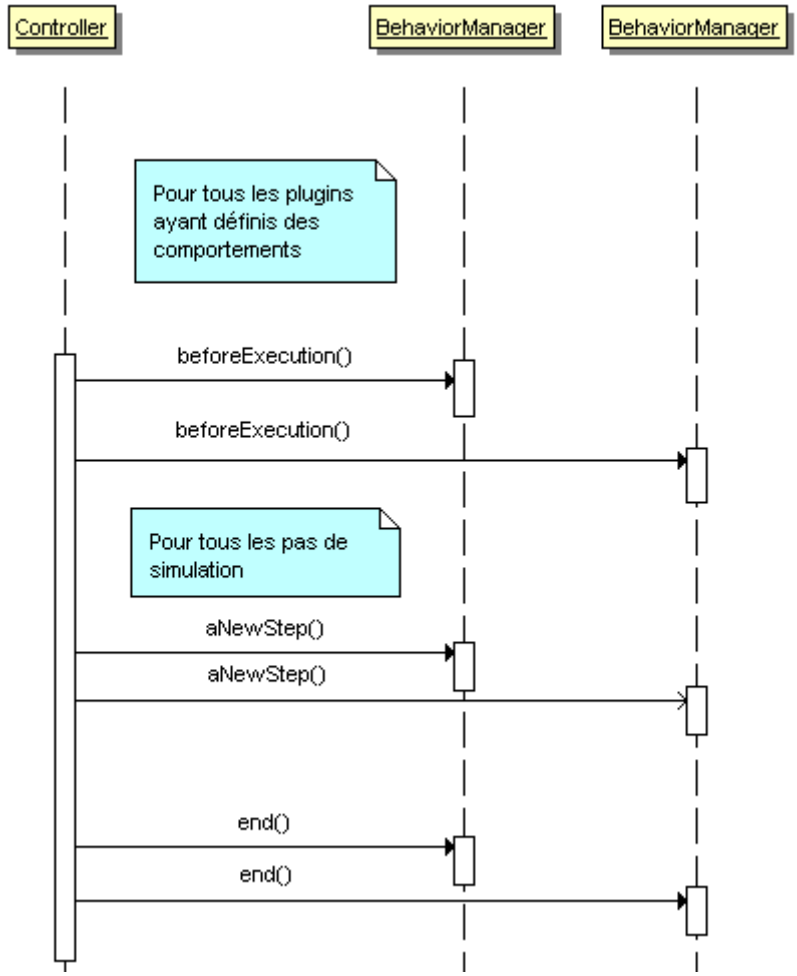

Illustration 12 : Phase d'exécution.

L'illustration 12 représente la communication entre Controller et BehaviorManager lors de la phase d'exécution.

Parallèlement à cela, des méthodes concernant la sérialisation des comportements et des options générales du plugin sont appelées. Ces fonctions utilisent toutes l'interface PersistentOptions.

La plupart des méthodes du BehaviorManager ont pour argument un helper de configuration. Celui-ci permet l'ajout, suppression, de comportements et donne des informations supplémentaires telles que le path du modèle utilisé.

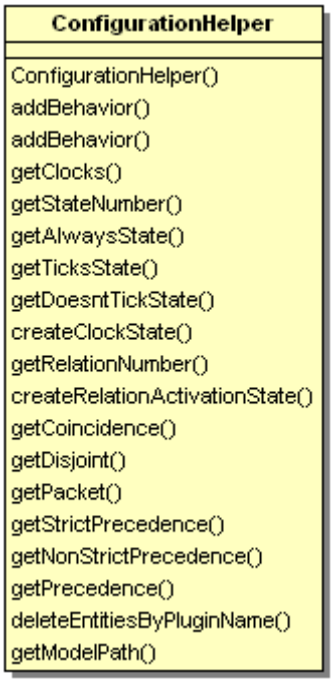

Illustration 13 : La classe ConfigurationHelper.

Les méthodes addBehavior forcent l'association entre un comportement, un état d'activation et un PersitentOptions.

Le helper peut aussi créer des RelationActivationState et des ClockActivationState.

5) Le BehaviorManagerGUI

Cette classe est optionnelle, elle peut être fournie au niveau du point d'extension. C'est une classe abstraite qui implémente un service de base d'affichage graphique offert par le plugin de comportements.

Ce service graphique repose sur l'api de SWT et JFace (cf. http://www.eclipse.org/swt/).

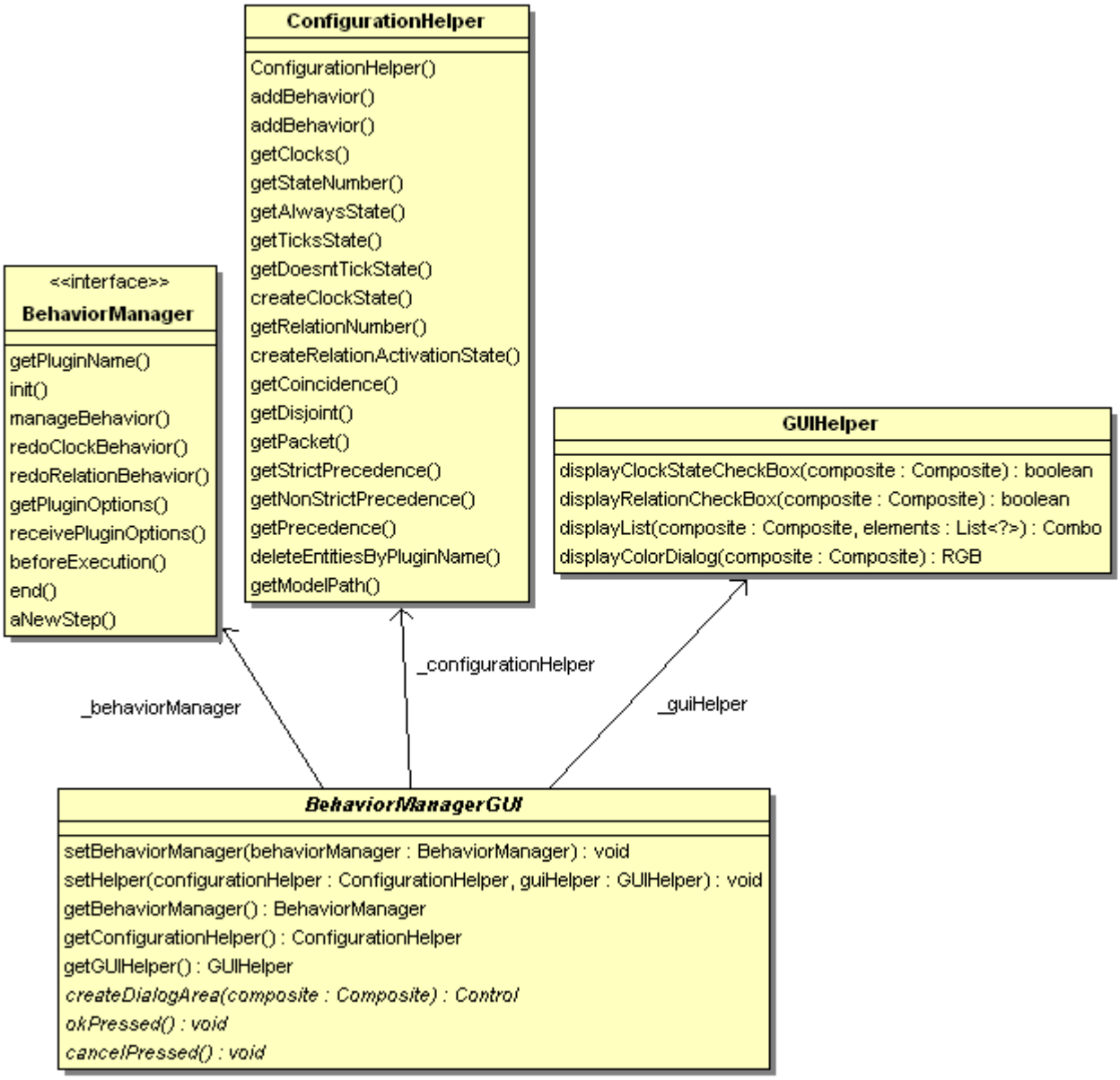

Illustration 6 : La classe BehaviorManagerGUI.

Si vous inscrivez cette classe dans le point d'extension, lorsque vous cliquez sur le nom de votre BehaviorManager dans la liste du run configuration d'Eclipse, une boite de dialogue s'ouvrira **au lieu** d'appeler la méthode manageBehavior() du BehaviorManager.

Cette classe abstraite contient :

- une référence sur le BehaviorManager du plugin ;
- une référence sur un ConfigurationHelper ;
- une référence sur un GUIHelper.

Les méthodes à implémenter seront utilisées lors de la création et manipulation de la boite de dialogue.

Le BehaviorManager et le ConfigurationHelper étant déjà détaillés plus haut dans ce rapport, nous nous intéressons ici au GUIHelper.

Cet objet a pour but de simplifier l'écriture des opérations graphiques classiques.

Par exemple pour afficher une liste d'objets dans un SWT combo box, il suffit d'utiliser la fonction displayList avec pour argument la liste des objets en question. Cette méthode utilise la fonction toString() pour l'affichage et retourne une référence sur l'item SWT Combo. A tout moment, on peut récupérer l'objet choisi par un utilisateur en utilisant la fonction getData() de Combo (cf. Javadoc).

De la même manière, ce helper offre deux méthodes qui affichent dans la boite de dialogue un ensemble de SWT Button style check box pour que l'utilisateur puisse choisir un ClockActivationState et un RelationActivationState. En retour, vous récupérez une référence sur le tableau de booléens représentant l'état d'activation souhaité.

V. Conclusion

Après avoir lu ce rapport, vous devriez être en mesure de connecter un plugin qui ajoute du comportement au MDK TimeSquare. Pour plus de détails sur le fonctionnement interne du plugin de comportements, il faut alors vous référer au rapport correspondant.