

HAZOP UML

Architecture logicielle

Version 0.4

Table des matières

1. Introduction 5

2. Prérequis techniques 6

2.1. Eclipse 6

2.2. Développer un plugin 7

2.2.1. EMF 7

2.2.2. GMF / GEF 7

2.2.3. Les vues 9

2.2.4. La librairie Jface 10

2.2.5. Lancer un plugin depuis Eclipse 10

2.2.6. Apporter des modifications aux plugins générés 10

2.2.7. Utiliser le logiciel de subversion 10

3. HAZOPUML 12

3.1. Modèle métier 12

4. Architecture générale 15

4.1. La structuration en plugins 15

5. Architecture détaillée 16

5.1. Dépendances entre les plugins 16

5.2. Détails des projets, dossiers et packages 16

5.2.1. HAZOPUML 16

5.2.2. HAZOPUML.edit 17

5.2.3. HAZOPUML.diagram.\* 18

5.2.4. HAZOPUML.views 19

6. Décisions, contraintes et justifications 20

6.1. Base de données – fichiers XML 20

6.2. Modèle métier 20

6.3. GMF – Jgraph 20

6.4. Génération du modèle ecore 21

6.5. Projet – fichier 21

6.6. Partager les même ressources entre les éditeurs 22

6.7. Modification du code source généré 22

7. Fonctionnement architectural 24

7.1. Créer/modifier/supprimer un objet du modèle 24

7.2. Fonctionnement d’un objet viewer 24

7.3. Editer plusieurs diagrammes d’un fichier : “diagram partitionning” 25

7.4. Réutiliser un élément déjà défini: activer les raccourcis 25

7.5. Créer une figure à partir d’une image 25

7.6. Ajouter un onglet dans la vue Properties 25

7.7. Ajouter une action dans la vue Project Explorer 26

7.8. Changer l’attribut d’un élément : fonctionnement des enum 26

7.9. Partager le même modèle entre les editeurs 26

7.10. Texte affiché : messages 27

8. Configuration de GMF 28

8.1. Plugin HazopModel 28

8.2. Plugin UseCaseDiagram 28

8.2.1. Configurer le dossier où sera généré le code 29

8.2.2. Ajouter les onglets dans la vue Propriétés (pour l’élément « Cas d’utilisation ») 29

8.2.3. Configurer les shorcuts 29

8.3. Plugin SequenceDiagram 29

8.3.1. Configurer le dossier où sera généré le code 30

8.3.2. Configurer les shorcuts 30

9. Glossaire 31

10. Versions des composants utilisés 32

Historique des révisions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date** | **Version document** | **Version d’HazopUml** | **Modifications apportées** | **Auteur** |
| Juin 2009 | 0.1 | 0.1 | Création de ce document | Timothée Mieulet |
| 08 juillet 2009 | 0.2 | 0.2 | Correction | Mathilde Bes |
| 15 juillet 2009 | 0.3 | 0.2 | Ajout de précisions sur Eclipse et la configuration de GMF | Timothée Mieulet |
| 22 juillet 2009 | 0.4 | 0.2 | Corrections, mise en forme | Mathilde Bes |

1. Introduction

Ce document présente l'architecture du logiciel HazopUml. Il est destiné à toutes les personnes impliquées dans le développement.

Il présente les diverses technologies utilisées, il décrit les composants et la façon dont ils communiquent.

1. Prérequis techniques
   1. Eclipse

Le logiciel HazopUml repose sur le logiciel Eclipse.

La spécificité d'Eclipse IDE vient du fait de son architecture totalement développée autour de la notion de plugin : toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in. En rajoutant un plugin dans le dossier «plugin» d’Eclipse, on ajoute une nouvelle fonctionnalité au logiciel Eclipse.

Eclipse (voir Figure 1) est constitué de vues et d’éditeurs qui sont redéfinis lors de la création d’un nouveau plugin.

Les éditeurs ne peuvent pas être tirés/sortis de la page/window alors que les vues peuvent l’être !

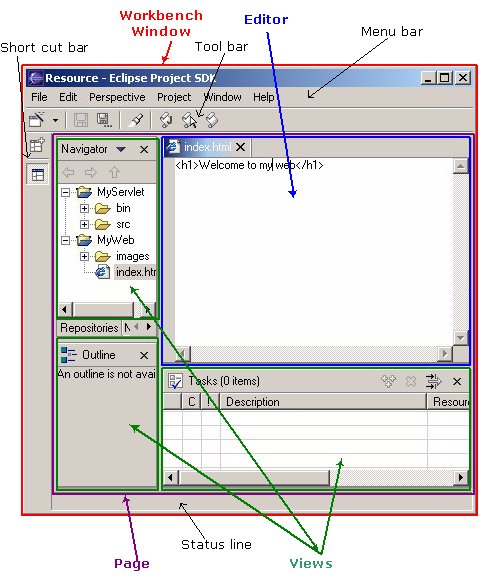


Figure - Représentation des différents éléments graphiques présents dans Eclipse

* 1. Développer un plugin

Le logiciel HazopUml fournit une «zone de dessin» pour permettre à l’utilisateur de modéliser des diagrammes. Ces interfaces graphiques ont été réalisées avec le framework GMF[[1]](#footnote-0) qui s’appuie sur les frameworks EMF[[2]](#footnote-1) et GEF[[3]](#footnote-2) pour générer le code source automatiquement.

* EMF propose un outillage qui permet de passer du modèle métier au code Java automatiquement.
* GEF permet de définir la représentation graphique des éléments contenus dans le modèle EMF.
* GMF met en place les outils permettant à l’utilisateur une meilleure ergonomie.

Ces frameworks s’appuient sur un fichier Domain Model qui représente le diagramme métier de l’application. Ce fichier est obtenu automatiquement après exportation du diagramme UML sous le format .ecore.

* + 1. EMF

1. Configuration

Jusqu’à présent, il n’a pas été nécessaire de configurer ce fichier.

1. Utilisation

On crée un fichier Domain Gen Model à partir du fichier Domain Model afin de générer le code de quatre plugins :

|  |  |
| --- | --- |
| <Project> | plug-in contenant le modèle métier sous forme de classes et permettant la persistance des données. |
| <Project>.edit | plug-in affichant la vue properties. |
| <Project>.editor | plug-in permettant de modéliser un projet HazopUml sous forme d’arbre d’un fichier \*.hazopUml |
| <Project>.tests | plug-in permettant de tester le modèle métier. |

* + 1. GMF / GEF

Ce Framework permet de créer un éditeur graphique à partir de quatre fichiers de configurations et du modèle métier utilisé par le framework EMF.

1. Configuration

|  |  |  |  |
| --- | --- | --- | --- |
|  | Domain Generation Model (.genmodel) | : | se base sur le fichier ecore (model) pour générer les classes Java. |
| gmfgraph | Graphical Definition Model (.gmfgraph) | : | se base sur le fichier ecore (model) pour définir l'aspect visuel du futur éditeur. |
|  | C'est à dire, comment sont représentés les éléments du model :   * les noeuds, * les liens * les libellés | | | |
| gmftool | Tooling Model (.gmftool) | : | se base sur le fichier ecore (model) pour définir la palette d'outils du futur éditeur. |
|  | Ce fichier permet de définir :   * les objets présents dans la vue Palette. * le “Context menu actions” affiché lorsque l’utilisateur clique avec le bouton droit sur une figure de l’éditeur. * la vue “Properties view”. | | | |
| gmfmap | Mapping Model (.gmfmap) | : | définit les liens entre le model (ecore) et sa représentation graphique (graph + tool). |
|  | C’est à dire, les relations entre les fichiers :   * Domain elements (\*.ecore) * Graphical elements (\*.gmfgraph) * Tooling elements (\*.gmftool) | | | |
| gmfgen | Diagram Generation Model (.gmfgen) | : | se base sur le Mapping model pour définir le model du diagramme et va permettre de générer notre éditeur de diagrammes. |
|  | Ce fichier permet de paramétrer «en profondeur» le nouveau plugin. Entre autres : partager des éléments entre diagrammes, comme les acteurs et de lancer un nouvel éditeur pour éditer un diagramme de cas d’utilisation ou de séquence. | | | |

1. Utilisation

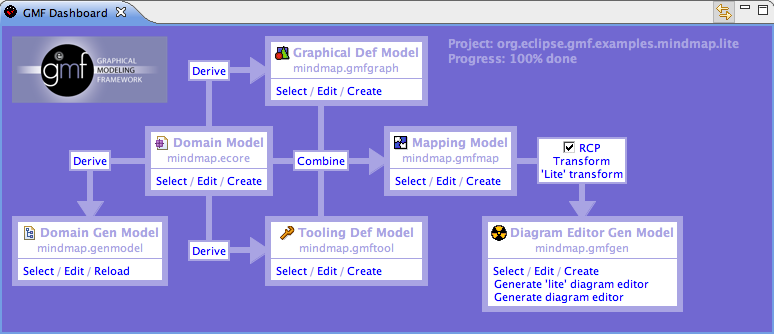


Figure - Dashboard de GMF

Dans un premier temps, on crée les fichiers «Graphical Def Model» et le fichier «Tooling Def Model» à partir du fichier «Domain Model» (ecore[[4]](#footnote-3)) qui a permis de générer le fichier «Domain Gen Model».

On crée ensuite le fichier «Mapping Model» déterminant, pour chaque classe du modèle métier, sa représentation graphique sur l’éditeur et sa représentation dans la palette.

A partir de ce fichier, on crée le fichier Diagram Editor Gen Model qui génère un nouveau plugin : <Project>.diagram.

* + 1. Les vues

La table HAZOP, les structures comme les mots guides sont affichés dans des vues différentes.

L’implémentation de ces vues se trouve dans le projet HAZOP.views.

1. Configuration

Pour ajouter une nouvelle vue au projet, on ajoute une dépendance vers le plugin «org.eclipse.ui.views.x» dans l’onglet «Dependencies» du fichier plugin.xml.

On déclare ensuite la nouvelle vue dans l’onglet «Extensions» de l’élément : «org.eclipse.ui.views» pour les vues, «org.eclipse.ui.views.properties» pour la vue Propriétés.

1. Utilisation

Le champ class de cette nouvelle déclaration doit référencer une classe héritant de la classe ViewPart pour les vues.

Pour réaliser l’interface graphique des vues, on utilise la bibliothèque JFace.

Se reporter à la section bibliographie pour plus d’informations sur les vues.

* + 1. La librairie Jface

JFace est un jeu de composants graphiques construisant sur les fondations d'une bibliothèque graphique proche du système.

JFace est basé sur SWT, et donc comme ce dernier dépend d'Eclipse, cela se retrouve dans ses paquetages de type org.eclipse.jface.\* . JFace apporte à SWT certaines fonctionnalités de Swing, à commencer par une architecture MVC (Modèle-Vue-Contrôleur), ce qui autorise les développeurs à mieux séparer leurs données et interfaces.

JFace étend donc SWT, mais n'empêche nullement l'application de se servir directement de SWT.

Cette librairie est utilisée dans le système pour créer les tableaux.

Se reporter à la section bibliographie pour plus d’informations.

* + 1. Lancer un plugin depuis Eclipse

Pour exécuter un plugin depuis Eclipse, il faut ouvrir le fichier plugin.xml et appuyer sur le bouton «Exécuter» de l’onglet «Overview».

* + 1. Apporter des modifications aux plugins générés

Lors de modifications des fichiers générés, il faut compléter le champ, situé au-dessus de la déclaration de la méthode, @generated par @generated NOT.

Cela permet de ne pas regénérer la méthode.

Après avoir regénéré le plugin, il est conseillé de rechercher avec l’outil Search d’Eclipse tous les fichiers contenant le texte @generated NOT.

* + 1. Utiliser le logiciel de subversion

Pour installer un logiciel de subversion sur Eclipse, ajoutez un nouveau site distant du nom de Subclipse et avec l'URL : <http://subclipse.tigris.org/update_1.6.x>

Pour chaque plugin (6 en tout), allez dans le menu File / New / SVN et sélectionnez «Checkout Projects from SVN».

Le projet se trouve à l’URL :

|  |  |
| --- | --- |
| Depuis le LAAS : | svn+ssh://gershwin/local/svn/hazop/trunk/ |
| Depuis l’extérieur : | svn+ssh://<user>@gershwin.laas.fr/local/svn/  hazop/trunk/HAZOPUML |

Créez un projet :

* de type «GMF project» pour le projet HAZOPUML
* de type «Plug-in project» pour les autres projets.

Le serveur SVN ne doit pas contenir les fichiers propres à la machine du programmeur : ceux commençant par «.\*», comme le dossier .settings ou le fichier .classpath.

1. HAZOPUML
   1. Modèle métier

Voici le modèle métier de l’application qui est utilisé par l’application HazopUml, sous forme de diagramme de classes UML.

Il représente à la fois les classes mises en œuvre à l’exécution de l’application, et les données stockées.

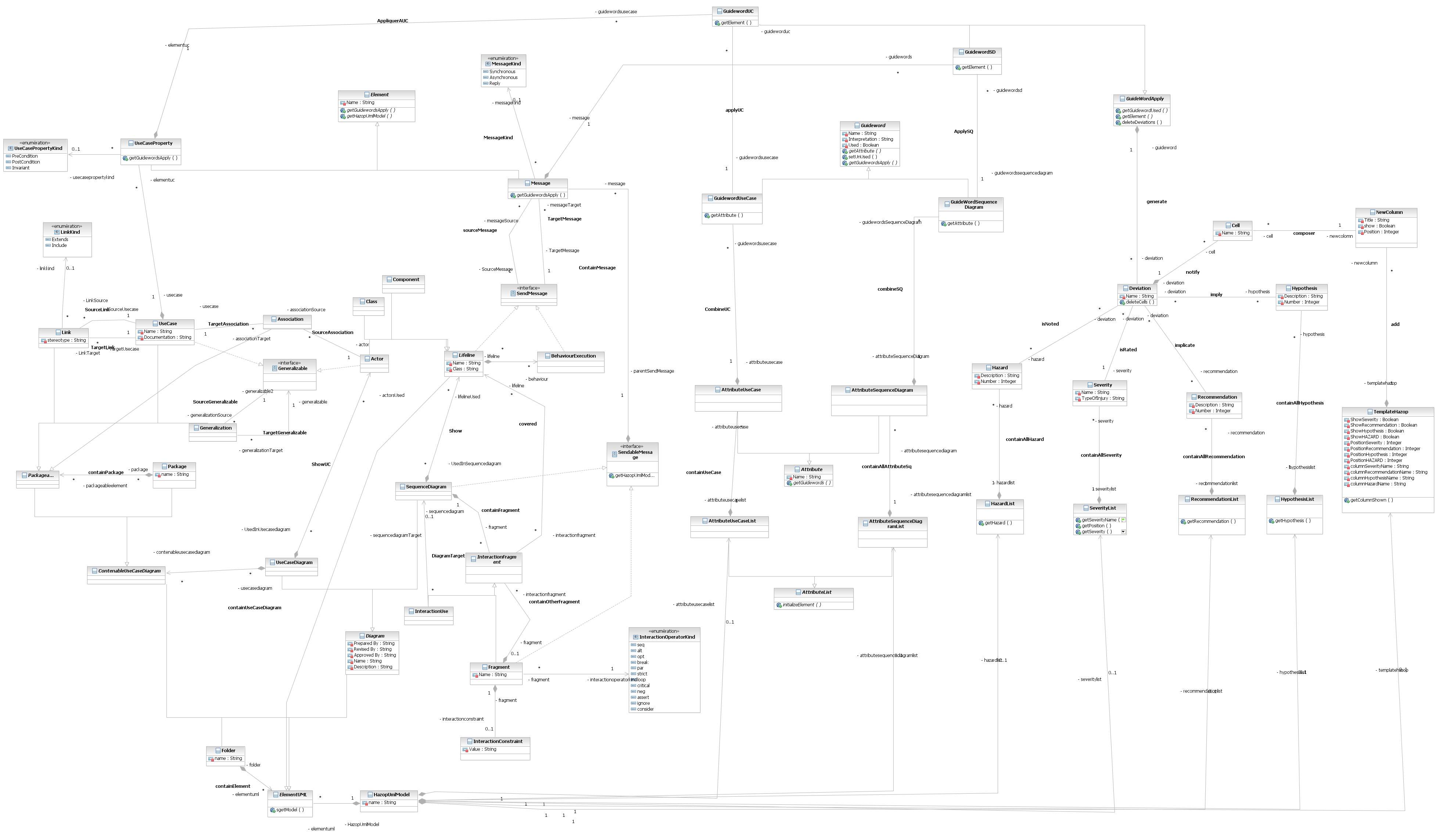
****

Figure - Diagramme de classes complet

Le fonctionnement de ce modèle métier est illustré à l’aide d’une version simplifiée (voir Figure 4, page suivante).

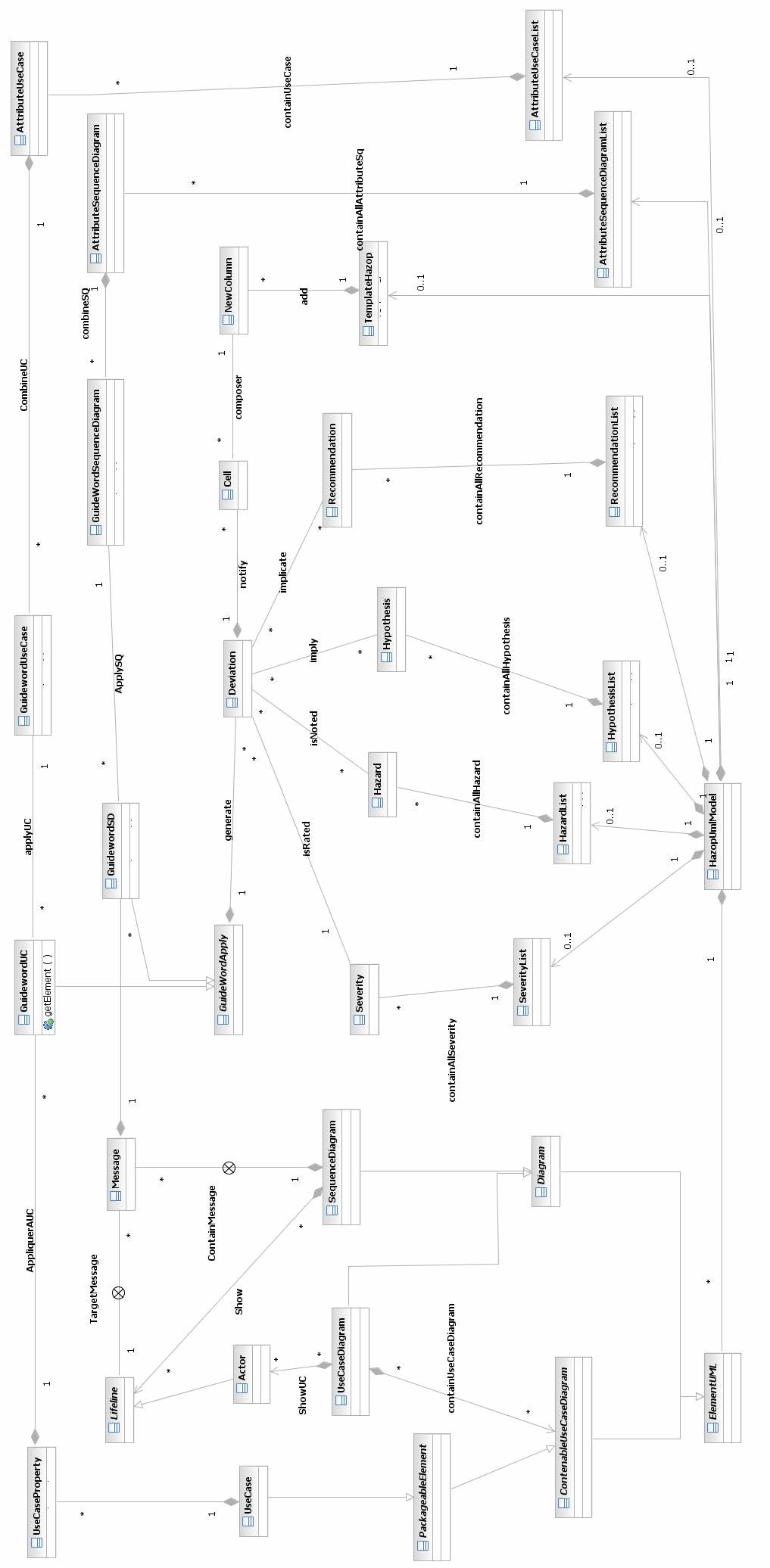


Figure - Diagramme de classes simplifié

La classe principale du logiciel HazopUml est la classe HazopUmlModel. Elle contient des diagrammes et les données communes à plusieurs diagrammes comme les cas d’utilisation, les acteurs et la table HAZOP :

* sa structure,
* les mots guides des cas d’utilisation,
* ceux des mesages ,
* les listes de gravités,
* les listes d’hazard,
* les listes de recommandations,
* les listes d’hypothèses employés lors de l’analyse.

Il existe deux types de diagrammes : les diagrammes de cas d’utilisations et ceux de séquences.

Un diagramme de cas d’utilisation regroupe plusieurs cas d’utilisation et d’acteurs communs à plusieurs diagrammes.

Un cas d’utilisation est décrit par des éléments de types Près-conditions, Post-condition ou Invariant auxquels on applique la liste de mots guides (GuidewordUseCase) prédéfinie.

Un diagramme de séquence décrit sous forme de messages les interactions entre les différents acteurs.

A chaque message est appliqué une liste de mots guides (GuideWordSequenceDiagram).

L’utilisateur peut indiquer plusieurs déviations pour chaque mot guide.

Une déviation peut référencer plusieurs Hazard, gravités, recommandations ou hypothèses prédéfinies. L’utilisateur peut aussi spécifier la valeur d’une cellule d’une colonne donnée.

1. Architecture générale
   1. La structuration en plugins

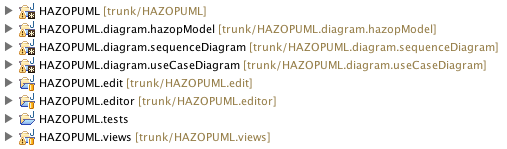


Figure - Vue du projet dans Eclipse

L’application HAZOPUML est divisée en huit plugins :

Les quatre premiers plugins sont générés à partir du fichier Domain Gen Model.

|  |  |
| --- | --- |
| HAZOP | plug-in contenant le modèle métier sous forme d’interface et permettant la persistance des données |
| HAZOP.edit | plug-in affichant les données affichées dans la vue propriétés |
| HAZOP.editor | plug-in permettant de modéliser un projet HazopUml sous forme d’arbre d’un fichier \*.hazopUml : ouverture de l’éditeur non graphique |
| HAZOP.tests | plug-in permettant de tester le modèle métier |
| HAZOP.diagram.model | plug-in permettant de créer et manipuler les dossiers, les diagrammes, les acteurs et les cas d’utilisations |
| HAZOP.diagram.sequence | plug-in permettant d’éditer graphiquement un diagramme de séquence |
| HAZOP.diagram.useCase | plug-in permettant d’éditer graphiquement un diagramme de cas d’utilisation |
| HAZOP.diagram.views | plug-in permettant d’afficher la table Hazop d’un élément ou de modifier les éléments de type rapport (report) ou structure (template) |

1. Architecture détaillée
   1. Dépendances entre les plugins

Lorsque l’utilisateur ouvre un fichier d’extension .hazopuml\_diagram, Eclipse charge le plugin HAZOPUML.diagram.model qui dépend des plugins HAZOPUML et HAZOPUML.edit.

Lorsque l’utilisateur ouvre un diagramme de séquence ou un diagramme de cas d’utilisation, le plugin HAZOPUML.diagram.model fait appel respectivement :

* au plugin HAZOPUML.diagram.sequenceDiagram
* ou au plugin HAZOPUML.diagram.useCaseDiagram.

Ces trois plugins HAZOPUML.diagram.\* appellent le plugin HAZOPUML.views afin d’afficher la vue «Hazop-view».

* 1. Détails des projets, dossiers et packages
     1. HAZOPUML

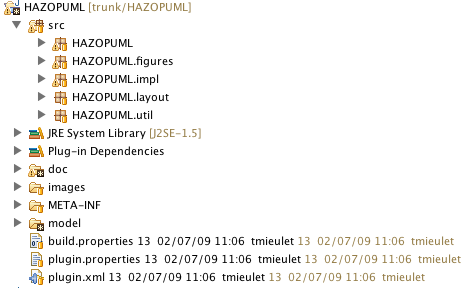


Figure - Arboresence du plugin principal

**Dossier src**

Les fichiers sources contenus dans ce dossier sont générés à partir du fichier HAZOPUML.genmodel.

Package HAZOPUML

Chaque classe présente dans le modèle métier est déclarée comme Interface.

Package HAZOPUML.impl

Chaque Interface du précédent package est implémentée par une classe qui héritent de la classe EObjectImpl.

Package HAZOPUML.util

Contient :

- la classe AdapterFactory qui attribue un adapteur à chaque classe du modèle

- la classe Switch (générique) qui supporte l’invocation de la méthode doSwitch() pour toutes les classes du modèle.

Dossier doc

Ce dossier contient la documentation relative au projet HazopUml. Actuellement, des diagrammes RSA ainsi que le présent document d’architecture.

Dossier model

Ce dossier contient l’ensemble des fichiers de configuration du framework GMF générant les plugins :

* HAZOPUML,
* HAZOPUML.edit,
* HAZOPUML.editor,
* HAZOPUML.tests,
* HAZOPUML.diagram.model,
* HAZOPUML.diagram.sequence,
* HAZOPUML.diagram.useCase.

Dossier src-figure

Si le modèle métier change, il est recommandé de supprimer le plugin généré.

Ce dossier contient les fichiers sources, déclarés dans le fichier d’extension .gmfgen, qui appartiennent aux autres plugins. Cela permet de pouvoir les supprimer sans perdre les modifications.

Package HAZOPUML.diagram.figures

Ce package contient certaines déclarations des figures plugins .diagram.\*, ainsi que les îcones.

Pour l’instant, il contient les déclarations des figures des diagrammes de séquence et de cas d’utilisation.

* + 1. HAZOPUML.edit

Package HAZOPUML.provider

Chaque classe définie dans le modèle métier hérite de la classe ItemProviderAdapter.

HAZOPUML.editor

Ce plugin permet de créer un éditer non graphique.

Packages HAZOPUML.presentation

Quatre classes y sont déclarées :

* une pour créer une vue,
* une autre pour déterminer les actions possibles,
* une autre pour créer un éditeur
* une denière permettant d’afficher une boîte de dialogue afin de créer un fichier.
  + 1. HAZOPUML.diagram.\*

La structure des plugins HAZOPUML.diagram.model ou .sequenceDiagram ou .useCaseDiagram est identique.

Ces plugins sont générés automatiquement par GMF par les fichiers de configurations Generator Model (fichiers d’extention gmfgen).

Packages HAZOPUML.diagram. x. edit. commands

Regroupe l’ensemble des classes permettant de créer un élément présent dans la palette.

Packages HAZOPUML.diagram. x. edit. parts

Ensemble des classes définissant la représentation graphique d’un élément.

C’est l’un des packages les plus intéressants à modifier manuellement, puisqu’il coordonne la vue éditeur avec le modèle métier.

Packages HAZOPUML.diagram. x. edit.policies

Les classes de ce package gèrent les requêtes de la vue et lui retournent l’objet correspondant.

On peut les modifier manuellement afin de customiser le comportement de l’éditeur de diagrammes.

Packages HAZOPUML.diagram. x. navigator

Classes déterminant le fonctionnement : objets affichés, actions, tries, ... de la vue «Project Explorer».

Packages HAZOPUML.diagram. x. part

Déclare l’ensemble des classes permettant de créer un nouveau fichier, de créer un éditeur, la vue palette,...

Packages HAZOPUML.diagram. x. preferences

Classes permettant d’éditer les préférences de l’application dans la vue préférence d’eclipse.

Packages HAZOPUML.diagram. x. providers

Classes permettant de créer des instances à partir d’un factory et d’un notifier.

Packages HAZOPUML.diagram. x. sheet

Classes permettant d’afficher la vue Properties personnalisée sur Eclipse.

* + 1. HAZOPUML.views

Packages HAZOPUML.views.activator

Ce package contient un fichier généré automatiquement permettant de gérer la «vie» du plugin.

Packages HAZOPUML.views.hazopTable

Regroupe l’ensemble des classes permettant de créer la vue HazopTable.

Packages HAZOPUML.views.severity

Regroupe l’ensemble des classes permettant de créer la vue Severity-List.

Packages HAZOPUML.views.util

Regroupe l’ensemble des classes partagées entre plusieurs vues.

1. Décisions, contraintes et justifications
   1. Base de données – fichiers XML

L’avantage d’une base de données est de pouvoir accéder rapidement aux données stockées sur le disque dur grâce aux index. Les index permettent de trouver les valeurs de ne pas lire toutes les données contenu dans un fichier. Cependant, la table HAZOP affiche toutes les données

Les frameworks utilisés enregistrent naturellement les données dans des fichiers XML.

Le fonctionnement du logiciel fait qu’il faut charger toutes les données d’une table.

L’utilisateur peut appliquer un filtre sur les déviations, mais ce filtre s’applique sur des données chargées en mémoire.

L’intérêt d’une base de données est donc limité au cas d’utilisation « Statistiques », qui n’est exécuté que rarement.

* 1. Modèle métier

Le modèle métier peut :

* s’inspirer ou reprendre celui d’un outils existant tel que UML2Tools[[5]](#footnote-4), qui est un logiciel libre utilisant GMF pour généré une partie de son code source.
* s’inspirer de projets étudiants, dont le client est Jérémie Guiochet.
* être défini.

Celui-ci a été redéfini en s’inspirant du modèle métier du logiciel UML2Tools, qui reprend celui donné par l’OMG . Ce modèle métier n’a pas été repris entièrement car il est trop complexe et il reprend tous les éléments des diagrammes UML.

* 1. GMF – Jgraph

Le logiciel HazopUml fournit une «zone de dessin» pour permettre à l’utilisateur de modéliser des diagrammes UML.

* 1. Pour chaque objet présent dans la zone d’édition (acteur, cas d’utilisation, relations...) il faut définir :
* sa représentation graphique,
* son modèle (défini par le diagramme de classes),
* et son contrôleur : dans un diagramme de cas d’utilisation, un acteur peut hériter d’un autre acteur mais pas de lui-même.
  1. L’utilisateur peut exécuter un ensemble d’actions : suppression, DéFaire/Re-faire une action, ...
  2. Plusieurs de ces objets peuvent être présent dans plusieurs diagrammes.

JGraph est une bibliothèque contenant une solution basique de création de graphiques sous Java. Il est possible de créer une fenêtre autour permettant d'employer les fonctions de base de l'API de façon transparente.

Le nombre de fonctionnalités déjà présentes est relativement élevé ce qui permet de créer sans problème les fonctionnalités N°1 & 2.

Le framework GMF s’appuie sur les framework EMF et GEF. Il a la particularité de générer le code source de l’application automatiquement grâce à cinq fichiers de configuration.

Le projet libre UML2Tools est un ensemble de plugins, généré par le framework GMF, permettant de créer des diagrammes UML.

Dans les deux cas, l’utilisation de ces deux frameworks demande un temps d’apprentissage et l’application.

Au final, ces interfaces graphiques ont été réalisées avec le framework GMF pour générer le code source automatiquement.

Au total, plus de quarante mille lignes, réparties dans plus de cinq cent classes, ont été générées automatiquement.

* 1. Génération du modèle ecore

Suite à des problèmes avec l’éditeur ecore\_diagram d’Eclipse, le modèle métier de l’application est réalisé avec le logiciel Rational Software Architect (RSA), puis exporté sous format Ecore. Cette solution permet aussi d’utiliser la notation de modélisation objet standard : UML.

* 1. Projet – fichier

Plusieurs diagrammes partagent les mêmes informations : structures (template), rapports (reports), ... L’utilisateur devra donc créer “un projet” qui contiendra ces informations.

Il existe deux solutions :

* Soit toutes les données d’un projet sont stockées dans un seul fichier. Comme avec le logiciel Rational Software Architect, les données des différents diagrammes sont présentes dans le fichier MIRAS/Diagrams/Bank Model.
* Soit l’utilisateur crée :
* Un fichier d’extension .HAZOPprj (Projet) qui contient les structures (template), les rapports (reports) et les données commun à plusieurs diagrammes comme les acteurs
* D’autres fichiers (diagrammes) d’extension .HAZOPuc ou .HAZOPsq qui contiennent respectivement les données d’un diagramme de cas d’utilisation ou celles d’un diagramme de séquence.

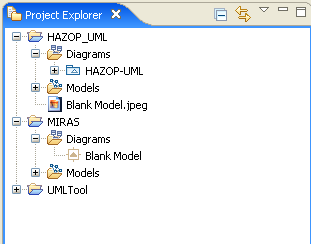


Figure Représentation des éléments du logiciel RSA

La première solution (représentée Figure 1) a été adoptée par le client car elle permet de s’échanger des données plus facilement. Cette fonctionnalité de GMF est appelé « Diagram Partitioning » sur internet et «partition de diagrammes» dans ce rapport.

* 1. Partager les même ressources entre les éditeurs

La «partition de diagrammes» provoque un bug car chaque diagramme possède ses propres Resources.

Lorsque plusieurs diagrammes sont ouverts, ils ont tous la même représentation des données, cependant lorsque l’utilisateur modifie un diagramme celui-ci a une représentation des données qui lui est propre.

Lorsque l’utilisateur sauvegarde ces modifications, les autres diagrammes mettent à jour leur représentation des données.

Mais si l’utilisateur a modifié un autre diagramme, ces modifications sont perdues lors de cette mise à jour.

Le problème est résolu lorsque tous les éditeurs utilisent la même ressource.

* 1. Modification du code source généré

Il existe deux solutions :

* soit créer un nouveau plugin modifiant le fonctionnement de certaines fonctions du plugin généré
* soit précéder les fonctions modifiées qui sont générées automatiquement du terme “@generated NOT”. Ainsi, la fonction n’est pas modifiée lors de la génération de code suivante.

Pour conserver les modifications du fichier plugin.xml, il faut remplacer «true» par «false» dans l’entête de l’option généré automatiquement :

<?gmfgen generated="true"?>

Cette dernière solution a été adoptée même si elle peut provoquer des problèmes lors de la re-génération automatique de code.

1. Fonctionnement architectural
   1. Créer/modifier/supprimer un objet du modèle

Pour modifier un objet, element de type EObject, de n’importe quelle vue, il faut créer une commande qui peut être de type :

> org.eclipse.emf.edit.command

Objets nécessaires :

* attribut de type EStructuralFeature, qui désigne l’attribut de l’objet element qui va être modifé. Par exemple, pour modifier l’attribut «name» d’un objet Deviation, on a attribut =

HAZOPUMLPackage.eINSTANCE.getDeviation\_Name()

* editeur de type TransactionalEditingDomain
* value de type Object = Nouvelle valeur.

AddCommand add = new AddCommand(editeur, element, attribut, value);

editeur.getCommandStack().execute(add);

> org.eclipse.gmf.runtime.emf.type.core.commands.SetValueCommand

Objet nécessaire : domain de type IDiagramEditDomain, objet récupéré sur l’éditeur graphique.

SetRequest reqSet = new SetRequest(editeur, element, attribut, value);

SetValueCommand operation = new SetValueCommand(reqSet);

domain.getDiagramCommandStack().execute(new ICommandProxy(operation));

* 1. Fonctionnement d’un objet viewer

Cet objet appartient à la librairie JFace et sert à créer soit une table soit un arbre.

Les nombreuses méthodes de cet objet permettent de le configurer facilement.

Voici les principales :

|  |  |  |
| --- | --- | --- |
| setInput(Object e) | : | insère des objets dans le modèle de la vue. *Attention, le ContentProvider doit être défini.* |
| addFilter(ViewerFilter filter) | : | cache facilement les lignes ne respectant pas une condition. |
| setSorter(ViewerSorter sorter) | : | permet de trier les éléments. |

La classe TableViewerColumn permet d’insérer une colonne dans une table.

Les méthodes setLabelProvider et setEditingSupport permettent de définir le texte à afficher pour chaque objet et le type d’éditeur (combo, checkbox ...) ainsi que son fonctionnement.

Les classes TableViewerFocusCellManager, ColumnViewerEditorActivationStrategy et ColumnViewerEditorActivationListener redéfinissent l’ergonomie pour éditer la table.

* 1. Editer plusieurs diagrammes d’un fichier : “diagram partitionning”

Le partitionning diagram permet d’éditer certains éléments avec un éditeur précis : lorsque l’utilisateur double-clique sur un diagramme de séquence, le système permet d’éditer ce diagramme avec le plugin «HAZOPUML.diagram.sequenceDiagram».

Cette fonctionnalité est réalisée par les classes OpenUseCaseDiagramEditPolicy et OpenSequenceDiagramEditPolicy du projet HAZOPUML.diagram.hazopModel et sont générées automatiquement par GMF.

*La configuration est donnée dans la partie* Configuration de GMF *page 26.*

* 1. Réutiliser un élément déjà défini: activer les raccourcis

Un élément donné peut être présent dans plusieurs diagrammes, comme les acteurs et les cas d’utilisation.

Cette fonctionnalité est réalisée par GMF en configurant la création de raccourcis de l’élément Diagram du fichier .gmfgen.

* 1. Créer une figure à partir d’une image

Il faut modifier la représentation graphique de l’élément dans le fichier .gmfgraph, puis générer le fichier .gmfgen à partir du fichier .gmfmap.

Générer ensuite le code du projet à partir du fichier .gmfgen.

Les déclarations des classes de ces figures se trouvent dans le package figures du projet HAZOPUML.

* 1. Ajouter un onglet dans la vue Properties

L’édition des propriétés d’un cas d’utilisation est réalisée dans des onglets personnalisés de la vue Propriétés.

En modifiant le fichier Generator Model (.gmfgen), GMF génère automatiquement une classe contenant les éléments affichés dans ce nouvel onglet, ainsi que la configuration du fichier plugin.xml.

On peut ensuite personnaliser cette classe.

Par exemple, l’onglet Invariant, affiché lors de la sélection d’un cas d’utilisation, fait référence à la classe InvariantUseCasePropertySection.

Le code généré automatiquement a été supprimé et la classe hérite maintenant de la classe AbstractUseCasePropertySection.

* 1. Ajouter une action dans la vue Project Explorer

L’ensemble des actions non générées automatiquement sont déclarées dans la classe HazopModelNavigatorActionProvider du package navigator.

* 1. Changer l’attribut d’un élément : fonctionnement des enum

Certains éléments comme les messages peuvent être de type synchrone ou asynchrone : lorsque l’utilisateur change le type d’un élément, sa représentation graphique doit s’adapter. Il faut :

* changer l’icône :

L’icône affichée est déterminée par la fonction getLabelIcon() de la

classe <ElementAModifier>EditPart.

* changer la figure :

La figure affichée est déterminée par la fonction createNodeShape() de la classe <ElementAModifier>EditPart.

* 1. Partager le même modèle entre les editeurs

Pour charger leurs ressources, les diagrammes utilisent une classe TransactionalEditingDomain (TED) qui gère aussi les modifications apportées aux ressources.

Nous avons donc créé dans le package HAZOPUML.diagram.hazopModel.part la classe MyEditingManager afin de ne charger qu’une seule fois les ressources et qu’elles soient communes à toutes les vues dans Eclipse.

En partageant la même ressource entre les éditeurs, les actions DéFaire/Refaire ne sont plus propres au diagramme/à l’éditeur.

Cependant, cela permet de modifier plusieurs diagrammes en même temps sans problème.

Pour mettre en place cette solution, nous nous sommes inspirés du tutorial “*Sharing single EditingDomain instance across several diagrams*” à l’URL :

[http://wiki.eclipse.org/GMF\_Tips](http://wiki.eclipse.org/GMF_Tips#Sharing_single_EditingDomain_instance_across_several_diagrams)

[#Sharing\_single\_EditingDomain\_instance\_across\_several\_diagrams](http://wiki.eclipse.org/GMF_Tips#Sharing_single_EditingDomain_instance_across_several_diagrams)

* 1. Texte affiché : messages

Les textes affichés sont enregistrés dans le fichier plugin.properties et sont accessibles grâce à la classe Messages ou sont directement utilisés par le fichier plugin.xml.

1. Configuration de GMF

Après modification d’un des fichiers d’extension .gmfgraph, .gmftool et .gmfmap, il faut re-générer le fichier d’extension .gmfgen. Voici les modifications à apporter à ce fichier, suivant le diagramme à générer.

Le fonctionnement est expliqué dans les paragraphes précédents.

* 1. Plugin HazopModel

Configurer le dossier où sera généré le code

Modifier les objets :

Gen Editor Generator

Package Name Prefix HAZOPUML.diagram.hazopModel

Gen Plugin

ID HAZOPUML.diagram.hazopModel

Changer le fonctionnement de l’élément Dossier

Dans les objets de l’élément Gen Editor Generator/Gen Diagram :

* Supprimer l’élément Gen Compartment2EditPart.
* Dans la vue propriétés de l’élément Gen Child Node Folder2EditPart, changer la valeur de l’attribut Diagram Element/Compartments par Gem Compartment FolderFolderCompartmentEditPart.

Configurer le partitionning diagram

Dans les objets de l’élément Gen Editor Generator/Gen Diagram :

SequenceDiagramEditPart/Open Diagram Behaviour

Diagram Kind

SequenceDiagram

Editor ID HAZOPUML.diagram.sequenceDiagram.part.HazopModelDiagramEditorID

Edit Policy Class Name

OpenSequenceDiagramEditPolicy

UseCaseDiagramEditPart/Open Diagram Behaviour

Diagram Kind

UseCaseDiagram

Editor ID HAZOPUML.diagram.useCaseDiagram.part.HazopModelDiagramEditorID

Edit Policy Class Name

OpenUseCaseDiagramEditPolicy

* 1. Plugin UseCaseDiagram
     1. Configurer le dossier où sera généré le code

Modifier les objets :

Gen Editor Generator

Domain File Extension hazopuml\_UseCaseDiagram

Model ID UseCaseDiagram

Package Name Prefix HAZOPUML.diagram.useCaseDiagram

Gen Editor Generator /Gen Plugin

ID HAZOPUML.diagram.useCaseDiagram

Name UseCaseDiagram Plugin

* + 1. Ajouter les onglets dans la vue Propriétés (pour l’élément « Cas d’utilisation »)

Ajouter trois «New Child Custom Property Tab» à l’élément Gen Editor Generator/Property Sheet :

Configurer les attributs du premier Custom Property Tab :

Human readable label Invariant

Identifier useCase-Invariant

Implementation class InvariantUseCasePropertySection

Configurer les attributs du second Custom Property Tab :

Human readable label Postcondition

Identifier useCase-Postcondition

Implementation class PostConditionUseCasePropertySection

Configurer les attributs du dernier Custom Property Tab :

Human readable label Precondition

Identifier useCase-Precondition

Implementation class PreConditionUseCasePropertySection

A chacun de ces «Custom Property Tab», ajouter un nouveau « Child Typed selection filter» ayant l’attribut «Types in selection» égal à :

HAZOPUML.diagram.useCaseDiagram.edit.parts.UseCaseEditPart

* + 1. Configurer les shorcuts

Modifier les attributs de l’élément Gen Editor Generator/Gen Diagram

Contains Shortcuts To hazopuml

Shortcuts Provided For HazopModel

* 1. Plugin SequenceDiagram
     1. Configurer le dossier où sera généré le code

Modifier les objets :

Gen Editor Generator

Domain File Extension hazopuml\_sequenceDiagram

Model ID SequenceDiagram

Package Name Prefix HAZOPUML.diagram.sequenceDiagram

Gen Editor Generator /Gen Plugin

ID HAZOPUML.diagram.sequenceDiagram

Name SequenceDiagram Plugin

* + 1. Configurer les shorcuts

Modifier les attributs de l’élément Gen Editor Generator/Gen Diagram :

Contains Shortcuts To hazopuml

Shortcuts Provided For HazopModel

1. Glossaire

**Framework :**

Ensemble de composants logiciels (classes, fichiers …) cohérent et dont le but est de proposer des services aux programmeurs, afin de leurs simplifier le développement de certaines parties logicielles.

Par exemple, EMF permet de générer une interface graphique et de sauvegarder les données simplement en décrivant le modèle métier de l’application.

1. Versions des composants utilisés

|  |  |
| --- | --- |
| **Composant** | **Version actuelle** |
| JDK | 1.5 |
| EMF | 2.4.0 |
| GEF | 3.4.2 |
| GMF | 2.1.3 |
| JFace |  |

1. Graphical Modeling Framework [↑](#footnote-ref-0)
2. Eclipse Modeling Framework [↑](#footnote-ref-1)
3. Graphical Editing Framework [↑](#footnote-ref-2)
4. ECore : meta model du domaine, représentation du schéma UML permettant de décrire sous forme arborescente n'importe quel modèle [↑](#footnote-ref-3)
5. <http://wiki.eclipse.org/MDT-UML2Tools> [↑](#footnote-ref-4)