UNIVERSITE LIBRE DE BRUXELLES
Faculté des Sciences appliquées

Année académique 2003-2004

REALISATION D'UN EDITEUR DE SCHEMAS POUR BASES DE DONNEES SPATIO - TEMPORELLES SOUS ECLIPSE

DIRECTEUR DE MEMOIRE : Esteban Zimanyi

MEMOIRE DE FIN D'ETUDES
PRESENTE PAR Régis Lemaigre
EN VUE DE L'OBTENTION DU
GRADE D'INGENIEUR CIVIL
INFORMATICIEN

1 Contexte du travail

Ce travail s'inscrit dans le cadre du projet MurMur, dont l'ULB est partenaire. Il s'intéresse plus spécifiquement à l'éditeur de schémas qui est un composant logiciel de ce projet.

1.1 Projet MurMur

MurMur est le diminutif de « Multi-representations and Multiple Resolutions in Geographic Databases ».

L'**objectif** général de ce projet est d'étendre les fonctionnalités actuellement fournies par les logiciels commerciaux de gestion de données pour supporter des schémas de représentation plus flexibles, comme la coexistence de représentations multiples du même phénomène du monde réel (route ou rivière représentée simultanément de plusieurs manières différentes suivant le niveau de détail par exemple).

Un ensemble de composants **logiciels** permettant d'atteindre cet objectif a déjà été réalisé en java à l'aide des librairies graphiques disponibles au moment où le projet a été réalisé, c'est-à-dire Swing et AWT.

1.2 Schémas MADS

Un **schéma MADS** constitue une description formelle de la structure d'une base de données MurMur particulière. Il décrit les différents types de données (types d'objets ou type de relations) qui interviennent dans cette base de données.

Il existe des conventions graphiques permettant de représenter un schéma MADS sous forme d'un **diagramme** similaire à un diagramme entité association enrichi des nouveaux éléments introduits par MurMur : représentations, aspect spatiaux et temporels des données, etc.

1.3 Editeur de schémas

Le composant logiciel permettant d'éditer, de sauvegarder, d'ouvrir et de valider des schémas MADS sous forme de diagrammes est l'**éditeur de schémas**. Les caractéristiques des éléments du schéma sont stockées dans un modèle orienté objet, visualisées et éditées sous forme d'un diagramme, et sauvegardées dans un fichier XML zippé.

1.4 Problèmes de l'implémentation actuelle du projet MurMur

Pour le projet en général :

- Les différents composants logiciel du projet occupent chacun une application distincte, ce qui oblige l'utilisateur à passer régulièrement de l'une à l'autre.
- Quand bien même il serait possible de regrouper ces composants dans une application unique, le désir de pouvoir vendre ces composants séparément obligerait à mettre en place une plateforme permettant l'ajout de plugins séparés, ce qui n'est pas évident.
- Au sein de la fenêtre de certains de ces composants, de nombreux cadres sont nécessaires et leur gestion est fastidieuse.

Pour l'éditeur de schémas en particulier :

- L'interface graphique de celui-ci a été réalisée à partir de rien et dans un intervalle de temps assez limité. Elle est donc très lente, peu pratique à utiliser et buggée.
- L'architecture de l'éditeur de schéma n'est pas très propre et ne respecte pas les recommandations habituelles pour une architecture modèle – vue – contrôleur.

1.5 Nouvelles technologies susceptibles de solutionner ces problèmes

1.5.1 Eclipse

Il s'agit d'une plateforme destinée à faciliter la réalisation de n'importe quel type d'IDE. A l'état brut, cette plateforme ne fait strictement rien de concret. On y introduit des fonctionnalités utiles en y contribuant à l'aide de **plugins**.

Cette plateforme est écrite en java, tout comme ses plugins, mais la librairie graphique employée n'est pas courante, il s'agit de SWT et non pas de AWT ou Swing comme c'est le cas dans l'implémentation actuelle du projet MurMur.

Eclipse permet de :

- intégrer dans une même plateforme les différents composants du projet MurMur,
- délivrer séparément ces différents composants sous forme de plugins,
- régler les problèmes de fenêtrage interne peu pratique de certains composants du projet.

En revanche :

- Eclipse utilise la librairie graphique SWT qui n'est pas compatible avec AWT et Swing, ce qui oblige à réimplémenter toute l'interface graphique du projet à l'aide d'une librairie peu familière.
- Eclipse est un monde extrêmement vaste qui nécessite une longue période d'apprentissage pour en maîtriser ne fût-ce qu'une petite partie des possibilités.

1.5.2 Draw2D

Draw2D est une **librairie graphique** permettant de construire par-dessus un composite SWT des réalisations graphiques complexes et efficaces de manière « légère », c'est-à-dire que leur peinture et la gestion des événements qui les concernent sont entièrement réalisées en java, sans faire appel à des éléments de l'OS sous-jacents (même philosophie que Swing).

Cette librairie va permettre de réaliser des représentations graphiques efficaces des schémas MADS de MurMur dans Eclipse. Ceci devrait solutionner les problèmes rencontrés à ce point de vue dans l'implémentation actuelle du projet MurMur.

Cette librairie est par contre complexe et très peu documentée.

1.5.3 GEF (Graphical Editing Framework)

Il s'agit d'une librairie conçue pour Eclipse et offrant un **framework pour la réalisation d'éditeurs graphiques suivant une architecture modèle – vue – contrôleur**. GEF délègue la partie affichage graphique à Draw2D.

Ceci solutionnerait le problème de l'architecture assez chaotique de l'éditeur de schéma sous sa forme actuelle.

Cette librairie est par contre très complexe et peu documentée.

2 Objectifs et contraintes du travail

2.1 Objectifs et stratégie

- Réaliser une étude des outils susceptibles de solutionner les problèmes actuels de l'éditeur de schémas du projet MurMur (Eclipse, Draw2D, GEF).
- Réaliser une étude de l'implémentation actuelle de l'éditeur de schémas.
- Evaluer l'impact de ces nouveaux outils sur l'implémentation de l'éditeur de schémas.
- Réaliser une application de ces nouvelles techniques sur un cas semblable à l'éditeur de schémas mais plus simple, dans le but de me familiariser avec ces techniques et de montrer qu'il est possible de réaliser l'éditeur de schémas à l'aide de celles-ci,
- Aller le plus loin possible dans les spécifications, l'analyse et la réimplémentation de l'éditeur de schémas du projet MurMur sous la forme d'un plugin pour Eclipse.

2.2 Objectif secondaire

Documenter les outils utilisés.

Il s'est avéré au cours de la réalisation de ce projet que celui-ci nécessitait la compréhension d'une quantité très importante de nouvelles techniques (librairies, plateforme) dont l'emploi est peu familier au programmeur java traditionnel et pour lesquelles il existait peu voire pas de documentation autre que la javadoc (bien souvent très insuffisante) et le code. Le fait que ces technologies semblent devoir être réutilisées à l'avenir dans le service d'informatique et réseaux, non seulement dans le cadre du projet MurMur mais aussi dans le cadre d'autres travaux, m'a poussé à rédiger une documentation aussi détaillée que possible des technologies pour lesquelles la documentation officielle manquait ou était peu claire.

3 Etude des outils disponibles

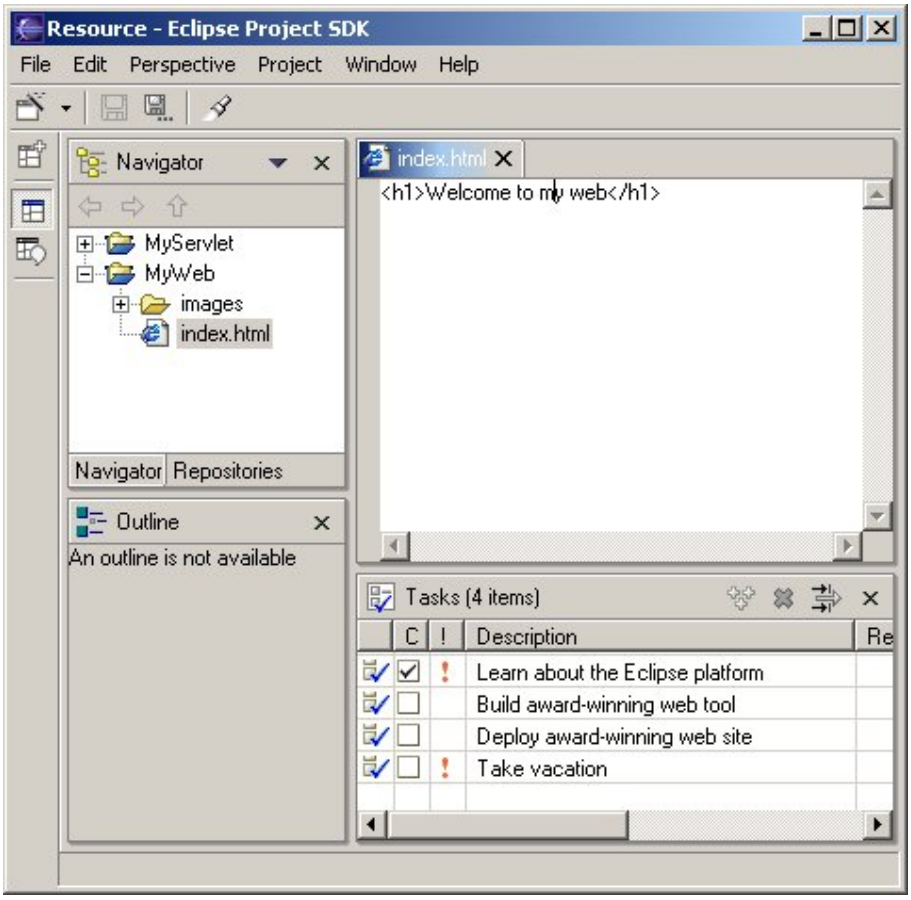
Etude des outils disponibles susceptibles de solutionner les problèmes actuels de l'éditeur de schémas du projet MurMur.

3.1 Eclipse

Cette plateforme commence à être largement documentée et ne nécessite donc pas de rédiger une documentation supplémentaire particulière. J'en ai déjà donné une brève description plus haut, je la complète un peu ici pour fixer les idées du lecteur qui ne serait pas familier avec cette plateforme et faciliter ainsi la compréhension de la suite du travail.

Une très bonne description peut être trouvée sur le site d'Eclipse : http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/guide/int_eclipse.htm. Elle prend +- 3 écrans et est illustrée. Ce document est une partie d'une documentation beaucoup plus complète sur les aspects de la plateforme Eclipse intéressants pour le développeur de plugins. Cela se trouve ici mais c'est beaucoup plus long: http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/topics_Guide.html

Très brièvement, et pour reprendre les propres termes des développeurs du projet, Eclipse est un environnement de développement intégré (IDE) pour n'importe quoi et rien en particulier. On introduit dans cet environnement des fonctionnalités de développement appropriées par l'intermédiaire de plugins. Un des plugins faisant partie de la base d'Eclipse est le workbench. Il constitue l'interface graphique de la plateforme. Voilà un screenshot de cette interface :



Dans cette fenêtre apparaissent plusieurs éléments :

- des vues : les fenêtres internes « navigator », « outline » et « tasks » sont des vues.
- un éditeur : ouvert sur le fichier index.html.
- une barre d'outils
- une barre de menus
- ...

Tous ces éléments sont définis dans des plugins écrits en java qui contribuent ainsi au workbench. Le workbench définit toute une série de « points d'extension » par lesquels les plugins peuvent contribuer à l'interface de la plateforme. Les vues et les éditeurs contribuent à des points d'extension qui leurs sont propres.

L'éditeur de schéma de MurMur sera principalement composé de vues et d'un éditeur.

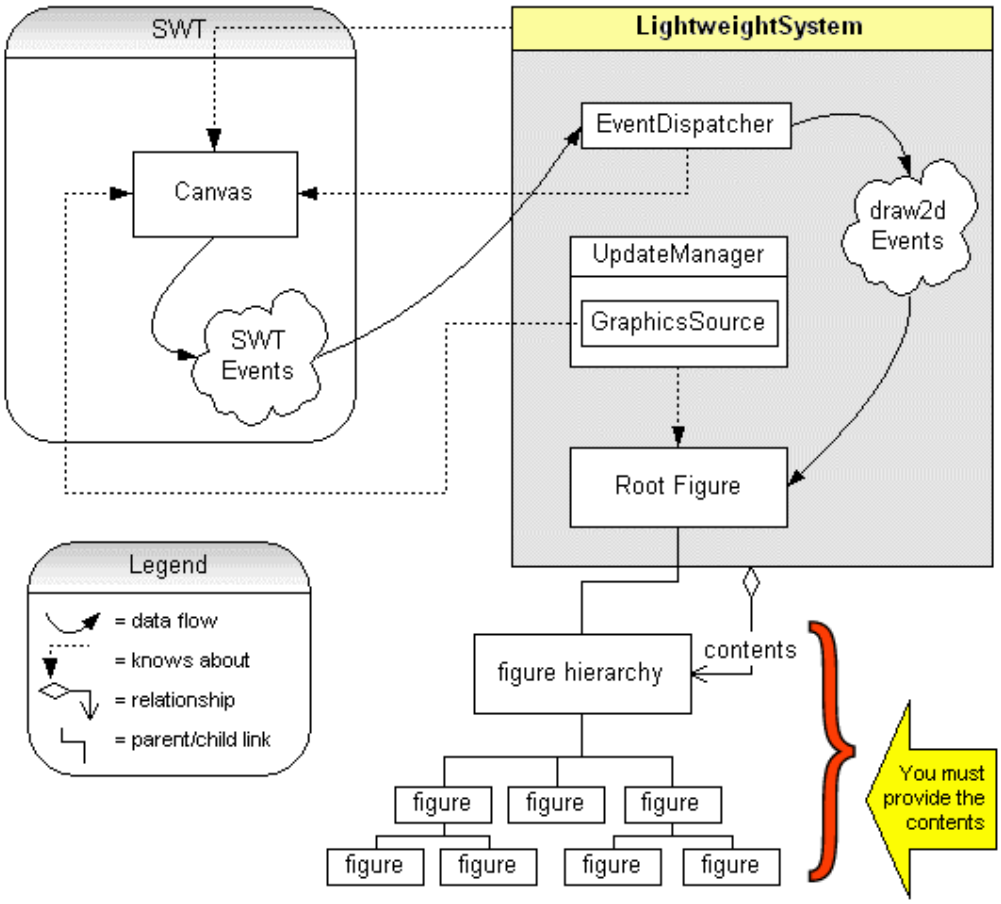
Un découpage particulier de la fenêtre du workbench en certaines vues et éditeurs est appelé une perspective.

3.2 Draw2D

Il s'agit d'une librairie développée dans le cadre d'Eclipse mais dont l'usage peut être étendu à n'importe quelle application basée sur la librairie graphique SWT, pas uniquement Eclipse.

3.2.1 Système

Draw2D est un système de composants légers bâti sur un Composite SWT (Composite = une sorte de panneau propre à la librairie graphique SWT). Les composants légers en question dérivent de la classe Figure. La javadoc fournit une vue schématique du système :



On y distingue plusieurs choses intéressantes :

- les Figures forment une arborescence,
- il existe un système de distribution des événements qui arrivent sur le Composite SWT aux Figures appropriées (EventDispatcher),
- il existe un système de mise à jour efficace de la visualisation graphique de l'arborescence de figures (UpdateManager), ceci comprenant la repeinture et le relayout de l'arborescence.

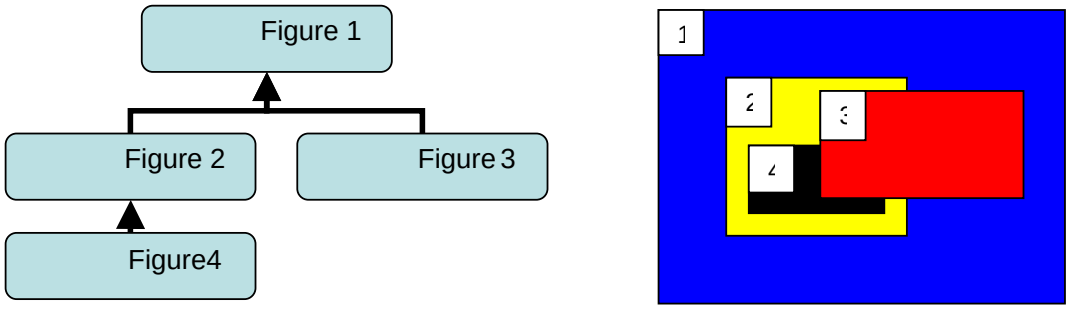
3.2.2 Figures

Une Figure est un objet possédant entre autres la faculté de se peindre, « se peindre » signifiant dans l'ordre:

1. peindre son bord,
2. se peindre lui-même,
3. envoyer à ses enfants l'ordre de se peindre à leur tour dans l'ordre où ils ont été ajoutés au parent.

La peinture de l'arborescence des figures est réalisée récursivement en demandant à la Root Figure de se peindre.

Ces deux dernières remarques définissent donc l'ordre de peinture des figures, qui détermine lui-même les avant-plans et les arrière-plans :



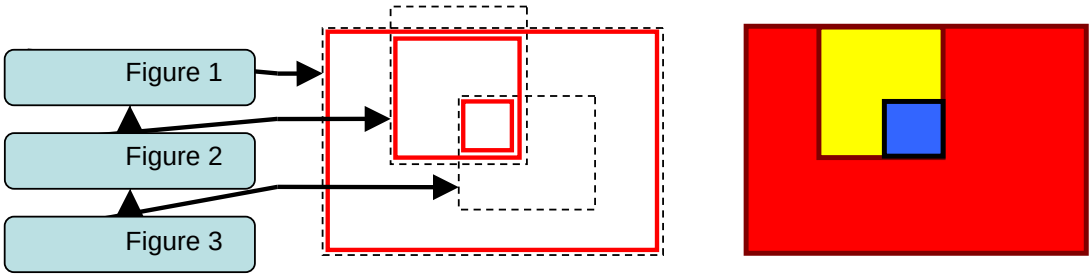
Ce schéma représente un arbre de figures et sa représentation graphique si chaque figure est peinte suivant un rectangle plein.

La relation parent/enfant dans l'arbre des figures correspond donc graphiquement à une relation arrière-plan / avant-plan.

3.2.3 Bornes des figures, repeinture de l'arborescence des figures

Chaque figure possède des bornes. Les bornes constituent une région rectangulaire hors de laquelle le système de clipping de Draw2D interdit toute peinture de la figure ainsi que de ses enfants. La figure est censée exploiter au mieux cet espace alloué pour introduire ses graphismes. La limitation de l'espace alloué à une figure permet d'améliorer considérablement l'algorithme de repeinture de l'arbre des figures.

Les fenêtres de clipping associées à différentes figures d'une arborescence sont illustrées ici :



Les limites de chaque figure sont représentées en pointillés, chaque figure introduit un graphisme constitué d'un bord noir placé sur ses bornes et un fond coloré, la fenêtre de clipping associée à chaque figure est symbolisée par un rectangle rouge.

Comme les enfants d'une figure ne peuvent introduire des graphismes qu'à l'intérieur des bornes de leur parent, la relation parent/enfant dans l'arbre des figures correspond aussi graphiquement à une relation contenant/contenu.

3.2.4 Layout managers

Les Figures peuvent être munies de layout managers. Les layout managers sont des objets responsables d'organiser la disposition spatiale des enfants de la figure à laquelle ils sont associés. L'emplacement spatial qu'un enfant devrait occuper lui est communiqué par l'intermédiaire de ses bornes. Il existe de nombreux layout managers fournis par Draw2D.

3.2.5 Algorithmes de « Hit testing »

Des algorithmes inclus dans Draw2D permettent de déterminer efficacement quelle est la figure la plus à l'avant plan susceptible d'avoir introduit des graphismes visibles en un point donné de l'interface graphique. Ceci se fait par défaut sur base des bornes des figures puisqu'il s'agit de la seule indication dont le système dispose pour déterminer où se situent les graphismes d'une figure.

3.2.6 Evénements

Il est possible d'enregistrer toute une série de types de listeners à une figure, notamment des mouse listeners et mouse motion listeners. Draw2D s'occupe de traduire les événements reçus par le Composite SWT sur lequel vit le système de figures en événements Draw2D et de router ces événements vers la figure appropriée sur base du hit testing.

3.2.7 Types de figures prédéfinis

Il en existe un grand nombre, mais il ne faut pas perdre de vue que Draw2D est une librairie à vocation graphique, malgré son système de gestion des événements, et n'a pas l'ambition de ressembler à Swing, c'est-à-dire qu'aucun modèle n'est associé aux figures et aucun comportement spécifique en réponse à des événements souris ou clavier n'est implémenté (sauf dans les cas les plus simples : bouton, checkbox).

Il existe des panneaux, des scroll panes, des labels, des figures permettant de représenter des images, des boutons, des checkbox, quelques formes géométriques simples, des lignes brisées...

3.2.8 Types de bords prédéfinis

Il en existe aussi un grand nombre, plus ou moins équivalents à ceux de swing, et qui peuvent être composés entre eux.

3.2.9 Connexions

Il existe des classes permettant de supporter la connexion de figures entre elles via une ligne brisée. Les décorations de cette ligne brisée, les points d'encrage de cette ligne brisée sur les figures extrémité, les bendpoints de cette ligne brisée, etc., évoluent dynamiquement si les figures extrémité de la connexion se déplacent. Draw2D fournit un support important pour ce genre de choses.

3.2.10 Curseurs et tooltips

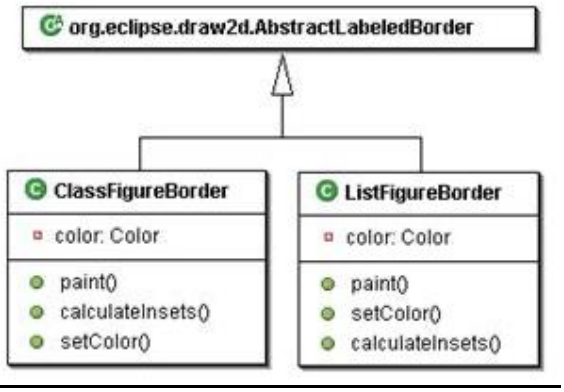
Il est possible d'associer à chaque figure un curseur et un tooltip. Lors des déplacements de la souris au-dessus du Composite SWT, le hit testing est utilisé pour déterminer quelle est la figure sous la souris et pour déterminer quel curseur afficher. Idem pour le tooltip.

3.2.11 Layers

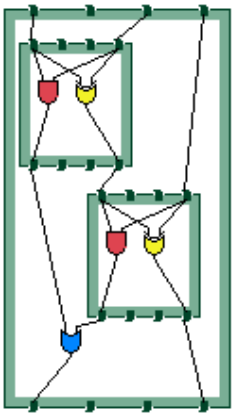
Il existe des figures transparentes aux événements (transparentes vis-à-vis du hit testing en fait) et qui peuvent être utilisées pour structurer le système de figures en une superposition de couches. Ce sont les layers.

3.2.12 Exemples de systèmes de figures Draw2D

- Diagramme de classes fait de figures Draw2D :



- Circuit électrique:



3.2.13 Description plus complète

J'ai rédigé une description plus complète mais elle n'est pas reprise ici faute de place.

3.3 GEF

3.3.1 Description

GEF constitue un framework pour la construction d'éditeurs graphiques intégrés à Eclipse. Couplé à Draw2D, son usage assure que tous les éditeurs graphiques d'Eclipse se comportent de la même manière et se ressemblent.

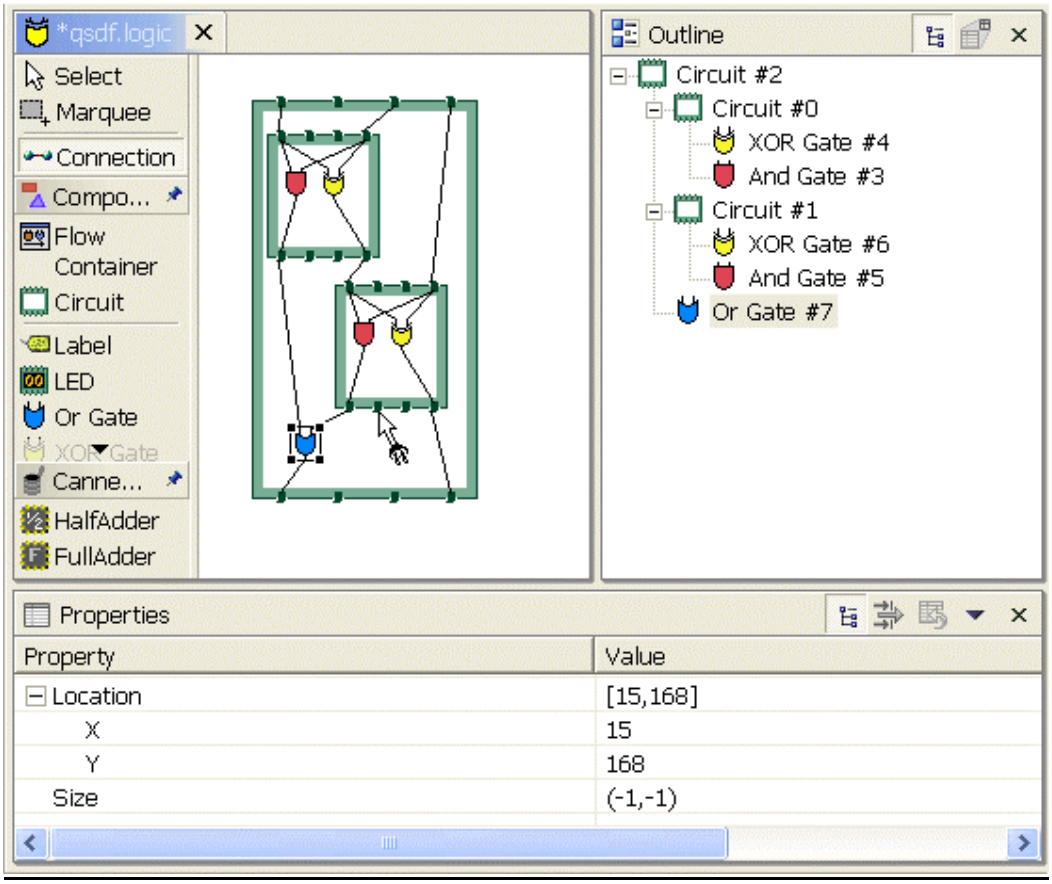
3.3.2 Caractéristiques de GEF

Les termes employés dans les caractéristiques et inscrits en italique seront définis plus clairement plus loin. Pour l'instant je suppose que leurs noms sont suffisamment évocateurs pour être plus ou moins compris.

- architecture *modèle – vue – contrôleur*.

- gestion de l'affichage du modèle suivant deux types de *viewers* : l'un basé sur les figures Draw2D (graphe), l'autre basé sur un arbre swt (arbre).
- gestion de différents *outils* standard (sélection -manipulation, création d'objets, création de connexions...).
- affichage d'une *palette* d'outils.
- affichage de *handles* sur l'objet sélectionné permettant d'obtenir au vol des outils permettant de redimensionner et déplacer les objets et de déplacer les bendpoints des connexions.
- gestion de différentes *actions* standard (suppression, undo, redo, print, etc.).
- les outils et les actions interagissent avec les contrôleurs par l'intermédiaire de *requêtes* prédéfinies.
- les contrôleurs réagissent aux requêtes en fabriquant des *commandes* encapsulant une modification particulière du modèle. Les commandes sont exécutées par une *command stack* et peuvent être faites et défaites (support du undo – redo).
- gestion du feedback : lors de l'exécution d'une opération sur l'interface graphique, les outils génèrent les requêtes définissant l'opération en cours et les envoient aux EditParts via des méthodes spécifiques pour leur donner une chance d'afficher un feedback donnant à l'utilisateur un pré - aperçu du résultat de l'opération en cours.
- possibilités de direct editing (édition des propriétés d'un élément par double clic sur sa représentation graphique).
- .....

3.3.3 Aperçu graphique d'une application GEF au sein d'Eclipse



Le modèle est ici la connectique d'un schéma électrique qu'il est possible de modifier graphiquement. On voit qu'il y a deux vues distinctes du modèle, l'une sous forme de graphe et l'autre arborescente. La palette apparaît à gauche.

3.3.4 Architecture modèle – vue - contrôleur

Les éditeurs graphiques affichent une certaine représentation graphique d'un modèle et permettent à l'utilisateur de modifier ce modèle par action sur sa représentation graphique. Selon l'application, il existe certains liens entre le modèle et sa représentation graphique qui doivent être maintenus au cours de l'édition.

L'architecture logicielle conseillée pour réaliser ce genre de chose est l'architecture modèle – vue – contrôleur :

- le modèle peut être constitué de n'importe quels objets informatiques. Cela peut être un réseau d'objets java comme cela peut être un système de fichiers. Le modèle ne devrait rien connaître de sa vue, ceci pour faciliter la réutilisation de son code dans un autre cadre et éviter de tout mélanger par souci de clarté et de facilité de modification ultérieure.
- la vue est une représentation graphique du modèle présentée à l'utilisateur. La vue ne devrait rien connaître de son modèle pour les mêmes raisons que précédemment.
- les contrôleurs sont les objets chargés de synchroniser les propriétés de la vue avec celles du modèle et vice versa, c'est-à-dire que si le modèle est modifié, le contrôleur devrait détecter cette modification et modifier la vue pour refléter le changement, et si l'utilisateur agit sur la vue, le contrôleur devrait interpréter cette action pour modifier le modèle.

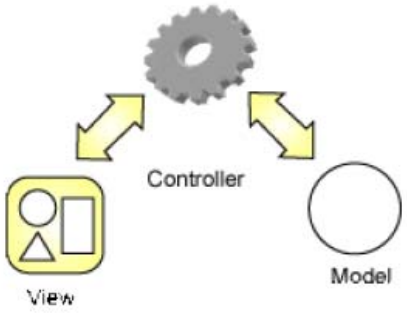


image tirée de la javadoc de GEF

GEF est construit sur cette architecture. Plus précisément :

- si c'est possible, le modèle devrait implémenter un mécanisme de notification des changements (événements, écouteurs),
- les contrôleurs devraient écouter le modèle et mettre à jour la vue en réponse aux notifications du modèle,
- lors d'une action sur la vue, les contrôleurs sont sollicités pour interpréter cette action, décider des modifications du modèle à faire, et effectuer ces modifications par l'intermédiaire de « commandes ». Les commandes devraient alors modifier le modèle uniquement. Celui-ci devrait alors notifier tous ses écouteurs du changement qui vient de se produire et notamment les contrôleurs qui mettent alors à jour la vue en conséquence.

Il y a beaucoup de conditionnel dans l'explication parce GEF n'est qu'un framework et n'impose que peu de choses à l'utilisateur. Les choses que j'ai mises au conditionnel doivent être comprises comme des règles de bonne pratique que l'utilisateur est censé respecter.

3.3.5 Contrôleurs : les EditParts

En gros :

Les contrôleurs dans GEF sont des objets de classes implémentant l'interface EditPart. Chaque EditPart lie un objet du modèle à sa vue.

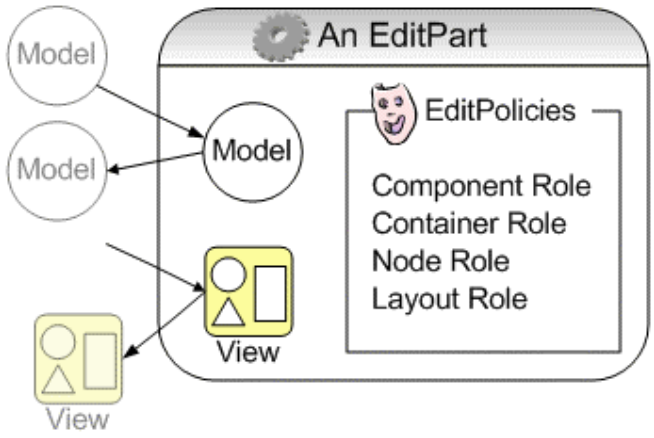


Image tirée de la javadoc de GEF

Un EditPart a la responsabilité de :

- créer la vue correspondant à l'objet du modèle auquel il est associé,
- écouter l'objet du modèle en tant que listener et modifier la vue en conséquence,
- fournir les « Commandes » encapsulant les modifications du modèle qui seront exécutées par la CommandStack,
- déterminer lui-même quels sont les objets qui sont les enfants de son modèle.

Plus en détails

Pour chaque représentation graphique de tout le modèle, il existe un groupe d'EditParts qui lient les objets du modèle à leurs vues dans la représentation graphique.

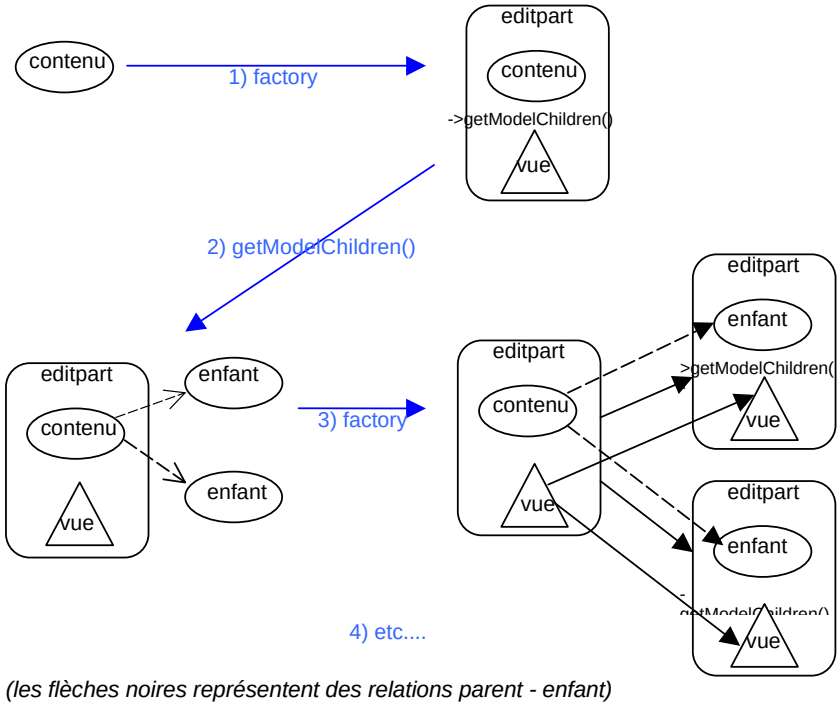
Structure de la représentation graphique

Chaque représentation graphique du modèle est formée d'un groupe d'EditParts organisés en une arborescence construite automatiquement par GEF sur base du modèle en utilisant :

- le fait qu'il existe un objet du modèle particulier appelé « contenu » et servant de base à la construction de l'arborescence,
- le fait qu'il existe un objet implémentant l'interface EditPartFactory, fourni par le programmeur, et qui permet à GEF d'associer à tout objet du modèle un EditPart de la classe adéquate (et notamment au « contenu »),
- le fait que chaque EditPart connaît les objets du modèle qui doivent servir de modèles à ses propres enfants grâce à la méthode getModelChildren() implémentée par le programmeur.

Lorsqu'un EditPart est créé par GEF et inséré dans l'arborescence, il crée sa figure et ajoute cette figure aux enfants de la figure de son propre parent. L'arborescence des EditParts ainsi construite définit donc une arborescence de figures symétrique à l'arborescence des EditParts eux-mêmes.

Ce processus est illustré à la figure suivante :



(les flèches noires représentent des relations parent - enfant)

Typiquement les retours des méthodes getModelChildren() ne dépendent que de l'état du modèle de sorte que les retours de ces méthodes définissent la structure de la vue que le programmeur souhaite obtenir pour un état du modèle donné.

Au fur et à mesure que le modèle se modifie ou que certaines options sont modifiées, il est possible que les retours des méthodes getModelChildren() varient. Dans ces conditions, la structure actuelle de la vue n'est plus celle souhaitée par le programmeur et donc un rafraîchissement est nécessaire. Ceci peut se faire par appel à la méthode EditPart.**refreshChildren**(). Cette méthode demande à un EditPart de rafraîchir sa descendance en comparant la liste des modèles de ses enfants avec les objets fournis par la méthode getModelChildren().

Cette méthode devrait être appelée par l'EditPart lui-même lorsqu'il remarque grâce au mécanisme de notification que son modèle a été modifié.

Propriétés de la représentation graphique

De même que les propriétés structurelles de la vue sont définies par la méthode getModelChildren(), les propriétés graphiques de chaque figure sont définies par la méthode refreshVisuals().

Par le processus expliqué précédemment, certains objets du modèle sont associés à une figure par un EditPart. En général, il existe un lien que le programmeur souhaite maintenir entre les propriétés de l'objet du modèle et les propriétés de la figure à laquelle il est associé. Ce lien doit être implémenté dans la méthode EditPart.**refreshVisuals**() qui doit mettre à jour les propriétés de la figure associée à un EditPart sur base de son modèle. Cette méthode peut être appelée pour rafraîchir les propriétés de la figure si le modèle change.

Cette méthode devrait être appelée par l'EditPart lui-même lorsqu'il remarque grâce au mécanisme de notification que son modèle a été modifié.

En résumé...

GEF assure la construction automatique de la partie vue et contrôleurs de l'application sur base du modèle. Le programmeur est capable de maintenir le synchronisme entre la partie modèle et la partie vue et contrôleurs en appelant des méthodes de rafraîchissement adéquates suite aux notifications de modification du modèle.

3.3.6 Viewers

Les viewers installent une vue du modèle sur un Composite SWT. Ils sont responsables entre autres de maintenir le mapping des figures vers les EditParts (dans ce sens-là, l'autre sens est direct), le mapping des objets du modèle vers les EditParts (dans ce sens-là, l'autre sens est direct), et d'effectuer à la demande un hit-testing pour savoir quel EditPart a introduit ses graphismes en tel ou tel point.

3.3.7 Outils, actions

Les outils et les actions sont des générateurs de requêtes. Ils captent les actions de l'utilisateur et génèrent les requêtes appropriées qu'ils envoient alors aux EditParts.

Les outils reçoivent les événements souris et clavier survenant sur l'interface et utilisent les fonctionnalités de hit testing du viewer pour les interpréter (par exemple pour déterminer quel EditPart possède des graphismes qui sont actuellement sous la souris).

Les actions portent en général sur l'EditPart couramment sélectionné.

3.3.8 Requêtes

Les requêtes sont des objets encapsulant toutes les données nécessaires à la spécification d'une certaine action souhaitée par l'utilisateur. Les requêtes sont envoyées aux EditParts qui ont alors la responsabilité de les traduire en Commandes spécifiques au modèle manipulé.

3.3.9 EditPolicies

Les EditParts délèguent la fabrication des commandes pour chaque type de requêtes qu'ils reçoivent à des objets particuliers de classes dérivées de EditPolicy. Les EditPolicies sont installées pour différents « rôles ». Les avantages à déléguer la fabrication des commandes à ces objets particuliers sont multiples :

- possibilité d'installer la même EditPolicy dans plusieurs EditParts différents qui nécessitent le même comportement en réponse à certaines requêtes => meilleure réutilisation du code,
- possibilité de changer dynamiquement le comportement d'un EditPart en cours d'exécution en installant d'autres EditPolicies,
- possibilité pour certaines classes d'EditParts de réinstaller des EditPolicies différentes de celles de leurs classes parentes pour certains rôles de manière à modifier les comportements d'édition hérités (mécanisme similaire à la redéfinition de méthodes).

Les EditPolicies ont aussi la responsabilité d'afficher les feedbacks fournissant une prévisualisation de l'exécution des Commandes issues des requêtes qu'elles comprennent.

3.3.10 Commandes

Les Commandes sont des objets encapsulant le code et les informations nécessaires à l'exécution d'une modification particulière du modèle. Plusieurs méthodes doivent être implémentées notamment execute(), redo(), et undo() dont les sens sont évidents.

3.3.11 CommandStack

Une implémentation d'une pile de redo-undo. Les commandes précédemment exécutées sont empilées de manière à pouvoir être défaites via leur méthode undo() et refaites par la suite. C'est la CommandStack qui est responsable d'exécuter les Commandes.

3.3.12 Description plus complète

J'ai rédigé une description beaucoup plus complète en anglais disponible aux adresses : http://eclipse-wiki.info/GEFDescription et http://eclipse-wiki.info/GEFDescription2 .

4 Implémentation actuelle de l'éditeur de schémas

4.1 Introduction

Cette section décrit l'implémentation actuelle de l'éditeur de schémas. Ses lacunes et la façon de les combler par l'emploi des technologies décrites précédemment seront expliquées par la suite.

4.2 Modèle

4.2.1 Introduction

Le modèle est une structure de données qui stocke les propriétés du schéma MurMur en cours d'édition.

Un schéma MurMur est constitué de différents types d'éléments. Principalement :

- des types d'objets,
- des types de relations,
- des représentations,
- des liens (héritage, association, etc.),
- des attributs, composés d'une liste de définitions distinctes pour chaque représentation,
- des méthodes, composées d'une liste de définitions distinctes pour chaque représentation,
- des identificateurs (éléments similaires aux clés dans les tables de dbs relationnelles) ...

Le modèle de l'éditeur de schémas comprend une classe pour représenter chacun de ces éléments. J'en présente ici un sous-ensemble simplifié qui ne comprend que les concepts que j'ai eu le temps de comprendre et qui sont proches de ceux des schémas entités - associations.

4.2.2 Présentation des Classes

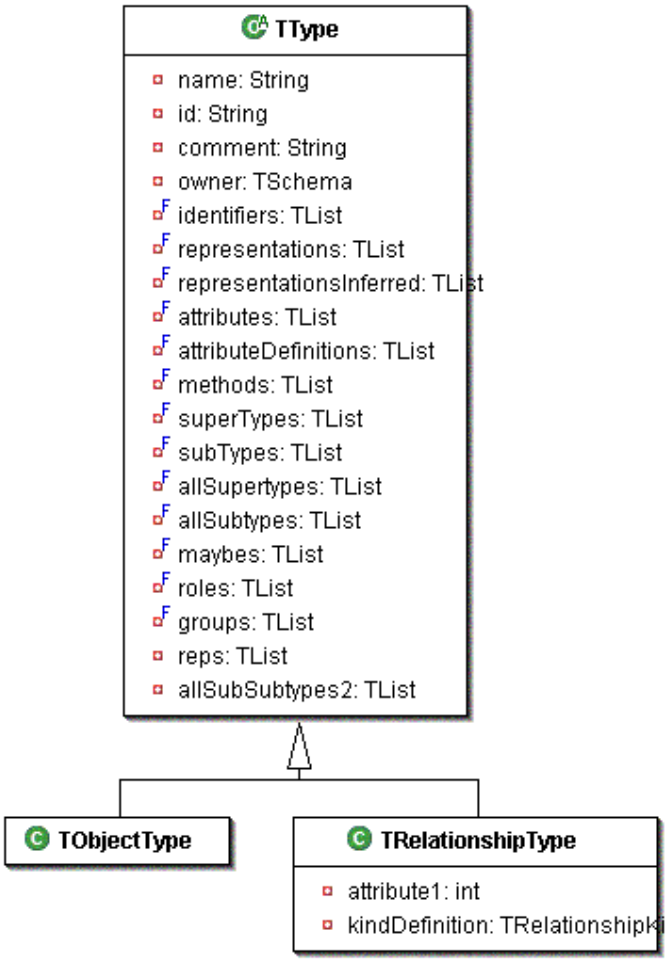
Remarque : les méthodes ne sont pas indiquées dans les diagrammes de classes faute de place.

Types

Les types d'objet et de relation que l'utilisateur peut définir sont représentés par deux classes :

- TObjectType : +- équivalent aux entités des schémas entités – relations.
- TRelationShipType : +- équivalent aux relations des schémas entités – associations.

Les deux dérivent de la super-classe TType qui en définit les propriétés communes, notamment un nom, des attributs et des méthodes.



L'attribut kindDefinition, de la classe TRelationshipKind est intéressant. La classe TRelationKind possède de très nombreuses sous-classes que je n'ai pas représentées ici.

Il existe plusieurs types de relationship. On peut se demander pourquoi ne pas en avoir fait des classes dérivées de TRelationshigType. Cependant le type d'une relation doit probablement pouvoir être changé dynamiquement après que la relation ait déjà été créée. C'est pourquoi on en fait un attribut : ainsi on évite d'avoir à supprimer la relation et puis en recréer une nouvelle si le type de la relation doit être changé.

Cette astuce revient souvent dans le code de l'éditeur de schémas.

Liens

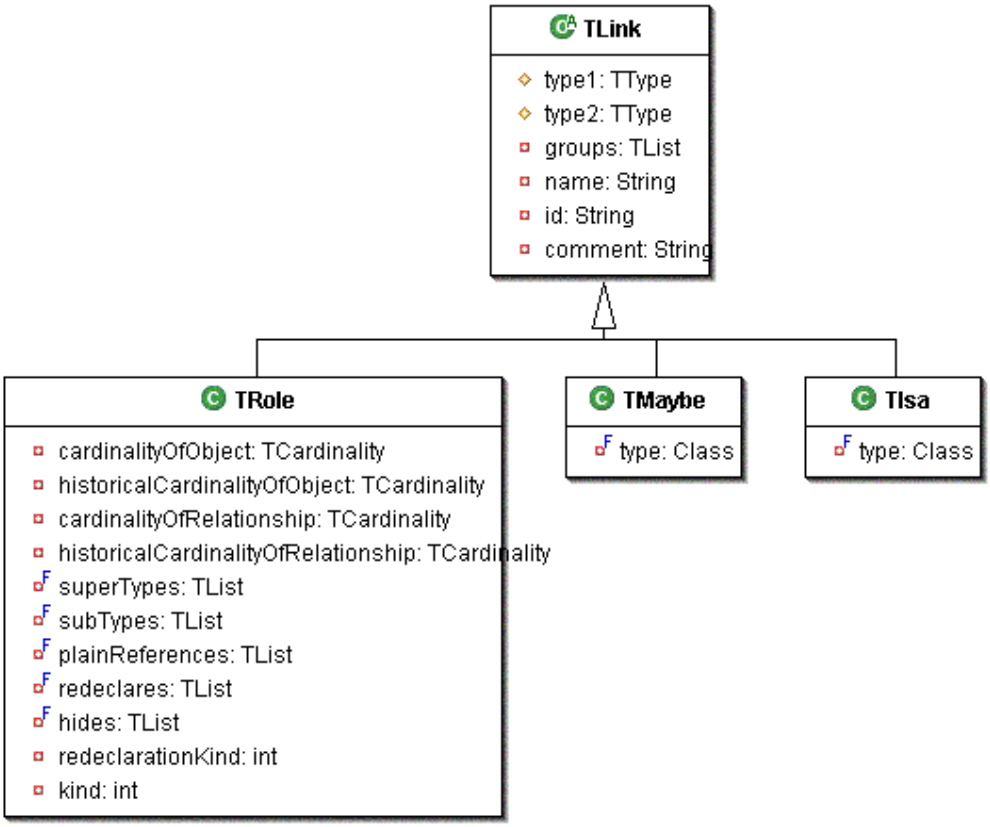
Il en existe de trois types : héritage, lien relation - entité (rôle), « maybe ».

Ils sont modélisés par les classes :

- TRole,
- TMaybe,

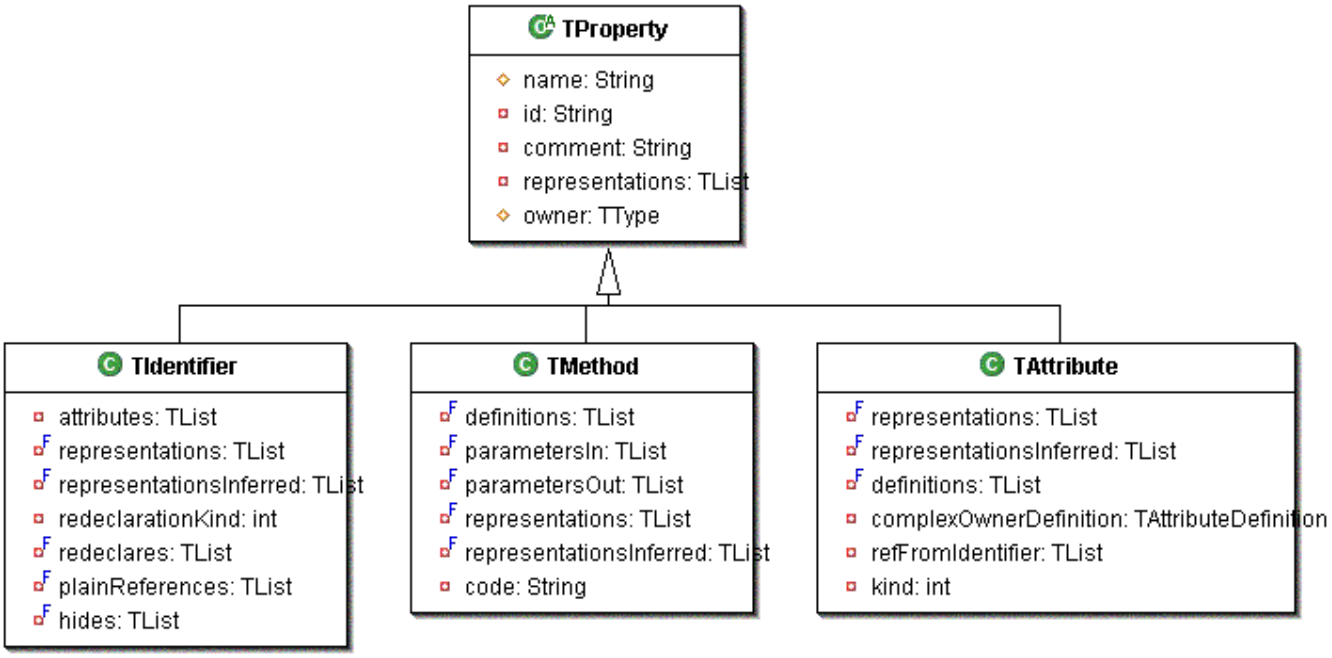
- TIsa

qui dérivent de la classe TLink qui en définit les propriétés communes, notamment les deux éléments qui partagent le lien.



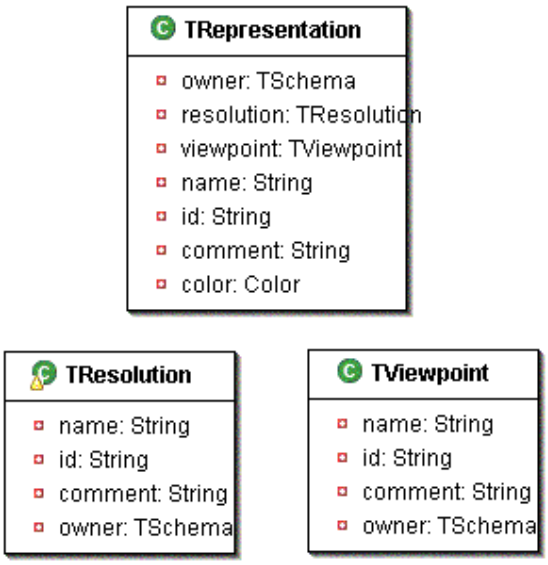
Propriétés

Dans les schémas MurMur les types de données ont des propriétés. Par exemple des attributs et des méthodes. Celles-ci sont représentées par les classes suivantes :



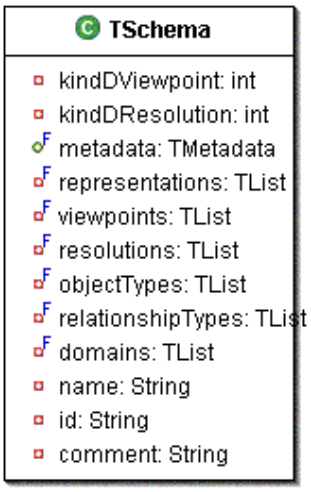
Représentation

Il s'agit d'un concept clé de MurMur (multi-representations...). Les différents éléments précédents possédaient pour la plupart une liste de représentations. Les propriétés d'un élément peuvent varier suivant la représentation. Sans entrer dans les détails, **c'est pour cette raison que par exemple les méthodes et les attributs possèdent une liste de « définitons » (les classes représentant ces définitions ne sont pas illustrées ici) et non pas une seule définition.** Une représentation est composée d'un point de vue et d'une résolution.



Schéma

La classe modélisant le schéma comprend les listes des modèles de tous les éléments qu'il contient :



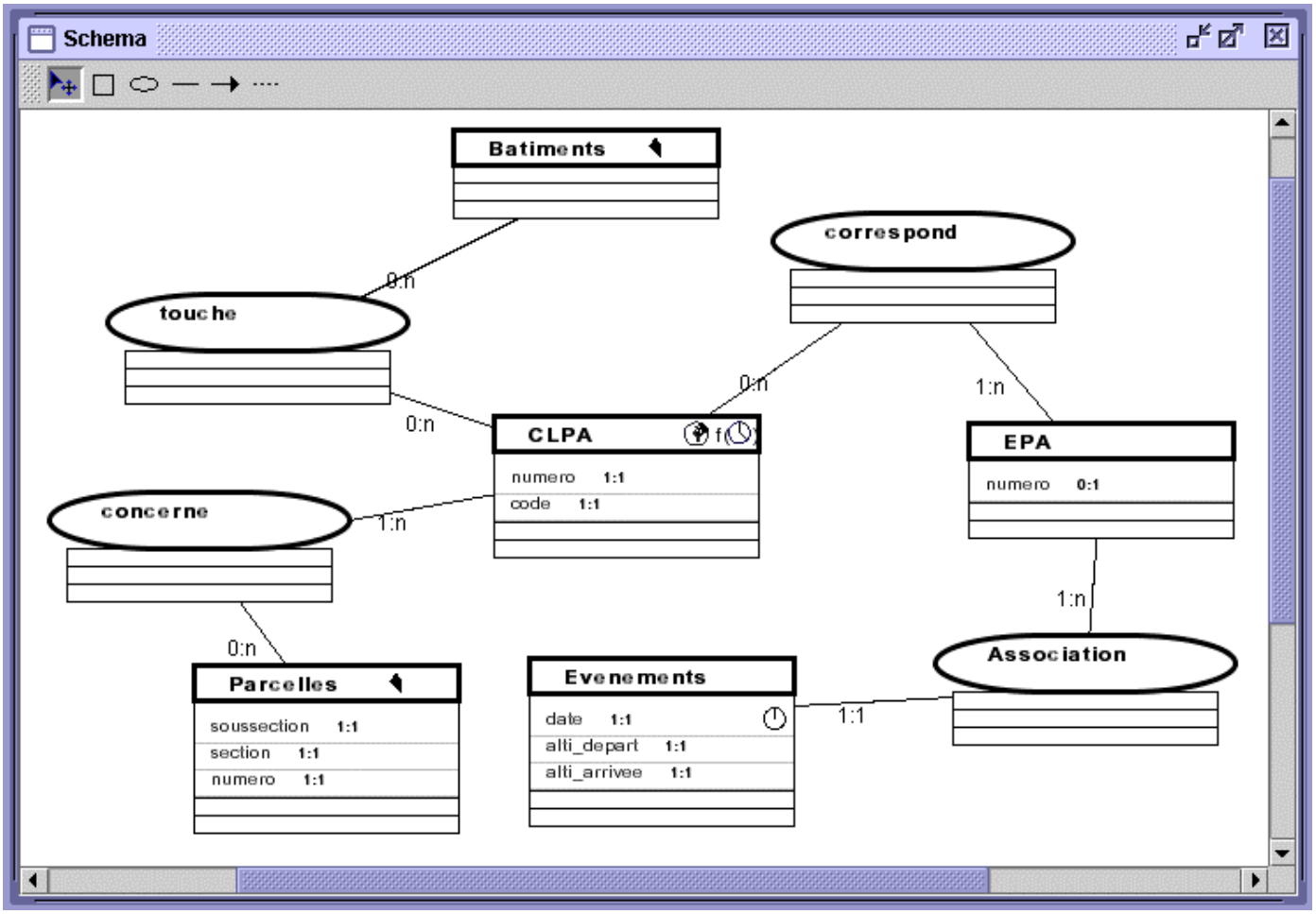
4.2.3 Lien entre le modèle et sa vue : listeners - événements ?

Les objets du modèle ne possèdent pas de liens vers leurs visualisations et ne notifient pas les changements qu'ils subissent. Ils se contentent de modéliser un schéma et c'est tout.

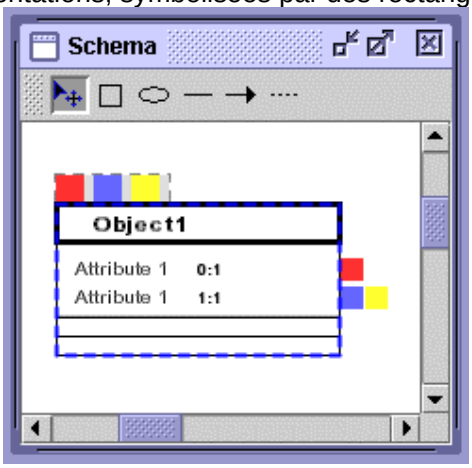
4.3 Vue principale

4.3.1 Exemple de vue du schéma

Ceci est un screenshot de l'éditeur de schémas (du moins d'une de ses internal frames) dans sa forme actuelle. On y voit représentés des liens de type rôle, des entités, des relations et des attributs ainsi que certaines propriétés spatio-temporelles des différents éléments.

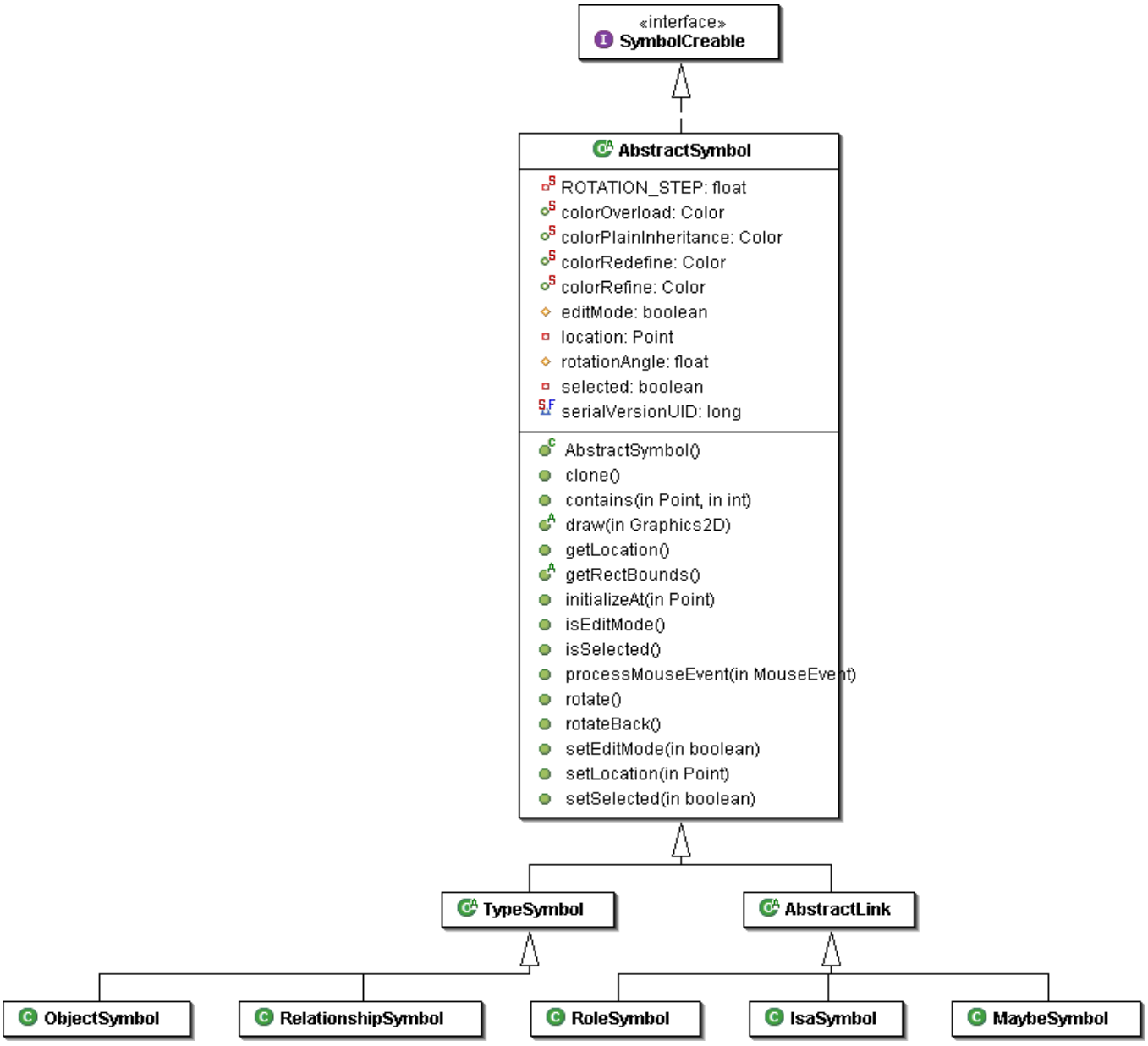


Un autre screenshot qui montre les visualisations d'éléments qui possèdent plusieurs représentations, symbolisées par des rectangles colorés :



4.3.2 Représentation graphique des éléments du schéma : les Symbols

Diagramme de classes



Peinture

La peinture de la représentation graphique du schéma se fait par la peinture successive de toutes les représentations graphiques de ses éléments sur un JPanel Swing. Dans le code du programme de telles représentations graphiques sont appelées Symbols. Ces représentations dérivent de la classe AbstractSymbol dont la particularité principale est de pouvoir se peindre, c'est-à-dire qu'elle possède une méthode draw() qui prend en paramètre un objet de la classe Graphics de Swing, cet objet donnant accès à des fonctionnalités de peinture à l'écran.

Sélection

Les représentations graphiques des éléments du schéma peuvent être sélectionnées. Le Symbol sélectionné est celui sur lequel seront focalisées les actions de l'utilisateur sur l'interface.

On voit que tous les éléments du schéma n'ont pas leur classe de représentation graphique propre, seuls les éléments de type container et les liens en possèdent une. Les attributs, les méthodes et les représentations MurMur par exemple n'en possèdent pas, leurs représentations sont simplement des propriétés graphiques des représentations du type d'objet ou de relation auquel ils appartiennent. Cela a une importance parce que les Symbols constituent l'unité de sélection dans l'interface graphique. *Il est donc impossible de sélectionner à la souris un attribut ou une méthode et de lui appliquer des opérations particulières. Ceci constitue une limitation importante, je trouve.*

Redimensionnement

Un Symbol peut être en mode d'édition ou pas (flag editMode). Si un Symbol est en cours d'édition, il affiche des points de contrôle à chaque coin qui permettent de le redimensionner.

Déplacement

Les Symbols peuvent être déplacés par dragging, ils reçoivent pour cela les événements mouse dragged qui les concernent via la méthode processMouseEvent (c'est aussi cette méthode qui gère le redimensionnement).

Lien avec le modèle

Chaque Symbol possède une référence à son modèle et dialogue avec lui lors de sa repeinture notamment. Ceci est une entorse à l'architecture modèle – vue – contrôleur dans laquelle la vue ne devrait pas connaître son modèle et vice versa.

4.3.3 Canvas de peinture du diagramme : la classe Draw

La vue est implémentée en redéfinissant la méthode paint(Graphics) d'une classe dérivée d'un JPanel Swing. Cette classe est la classe Draw. La méthode paint est appelée par Swing lorsque le JPanel doit être (re)peint en tout ou en partie.

La classe Draw écoute aussi les événements souris survenant sur le panneau.

Peinture

La classe Draw possède une liste des représentations graphiques des éléments du schéma présentées au point précédent. A quelques détails près, peindre le schéma complet revient à peindre un par un les éléments de cette liste de Symbols en appelant leur méthode draw. Tous les Symbols sont peints à chaque repeinture du schéma, même partielle.

Optimisation importante possible

Lors de l'exécution de requêtes de peinture, l'objet Graphics passé aux méthodes paint appropriées possède une fenêtre de clipping ajustée par Swing. Ceci donne lieu à une optimisation possible : si la fenêtre de clipping n'a pas d'intersection avec les bornes de l'élément à peindre, il est inutile de repeindre cet élément. Cette optimisation de la méthode paint de la classe Draw prendrait la forme :

```
public paint ( Graphics g )
```

```
{
        ...
        while ( il reste un symbole )
        {
                if ( la clipping area de g possède une intersection avec le symbole )
                        symbol.draw(g);
        }
        ...
}
```

Je n'ai pas vu de telle optimisation. Peut-être que ceci est en partie responsable du peu de vivacité de l'interface graphique de l'éditeur.

Evénements souris

La classe Draw capture tous les événements souris arrivant sur le JPanel dont elle est une sous-classe.

En gros :

- les événements mouse pressed provoquent la capture de la sélection courante, c'est-à-dire le Symbol qui est actuellement sous la souris,
- les événements mouse released provoquent l'affichage du menu contextuel,
- les événements mouse dragged sont forwardés à la méthode processMouseEvent() du Symbol sélectionné,
- les événements mouse clicked provoquent éventuellement la création de nouveaux éléments dans le schéma, et les doubles clics mettent le Symbol double-cliqué dans l'état d'édition.

Une chose est intéressante dans cela : l'unité de sélection dans l'interface graphique est le Symbol. Il n'est donc pas possible d'interagir avec des éléments internes à un Symbol par action sur l'interface graphique, comme dit précédemment.

Repeinture lors d'une modification des caractéristiques d'un Symbol

Le traitement d'un événement mouse dragged par la méthode processMouseEvent du Symbol sélectionné provoque un changement dans la position ou les dimensions de ce Symbol. Ceci occasionne dans l'implémentation actuelle de l'éditeur une repeinture de la totalité de la représentation graphique du schéma. C'est très peu efficace.

Optimisation possible

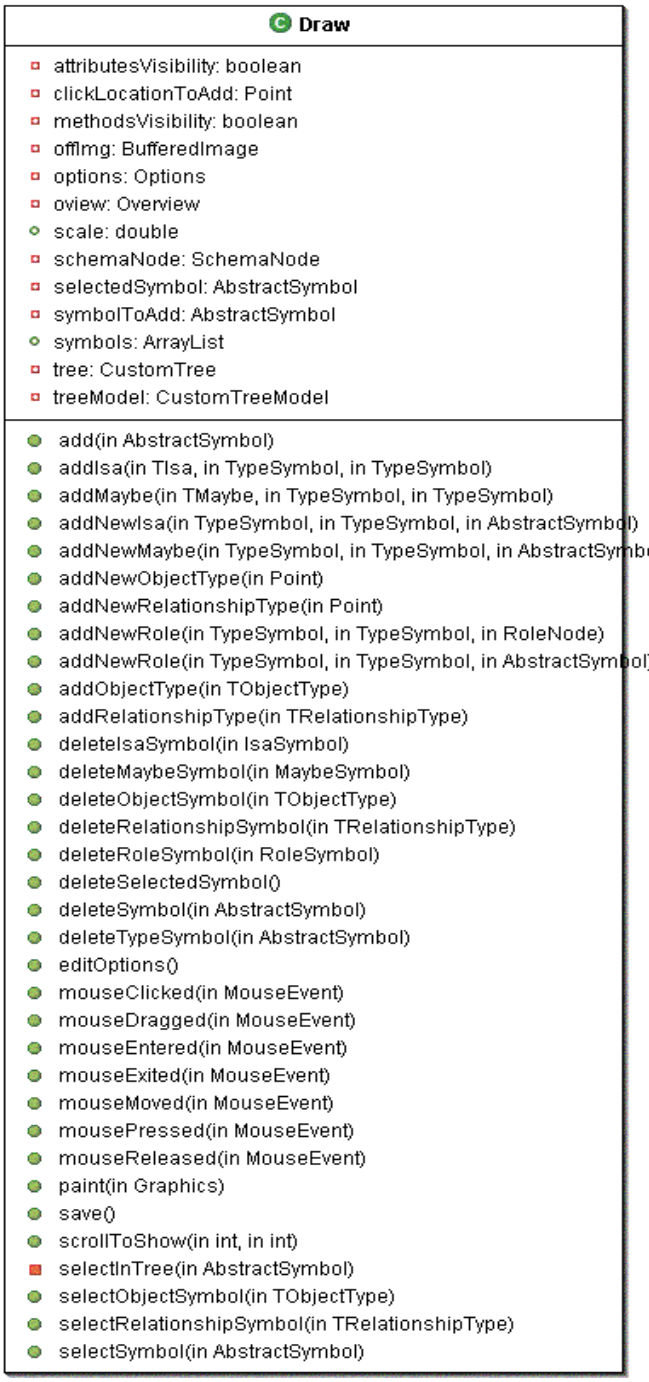
Lors du déplacement d'un Symbol, il ne faut repeindre (en différé pour permettre la « concaténation » des requêtes de peinture, c'est-à-dire à l'aide de repaint) que deux choses : la position précédemment occupée par le Symbol et sa nouvelle position. Le code devrait ressembler à cela :

```
//panel : le panneau sur lequel est dessinée la repr. graph. du schéma.
//symbol : le Symbol à repeindre.

panel.repaint ( symbol.getBounds() ) ; // repeinture différée de la région du JPanel
                                        //précédemment occupée par le Symbol.
symbol.processMouseEvent(ev) ; // déplacement du Symbol (les bornes changent).
panel.repaint ( symbol.getBounds() ) ; // repeinture différée de la région du JPanel
        //nouvellement occupée par le Symbol.
```

En effet la méthode JPanel.repaint (Rectangle) de Swing permet de programmer une repeinture partielle de l'interface. Le mot « partiel » signifie que la fenêtre de clipping de l'objet Graphics utilisé pour repeindre toute l'interface est réduite au strict nécessaire. C'est dans un tel cas que la première optimisation proposée prend tout son sens.

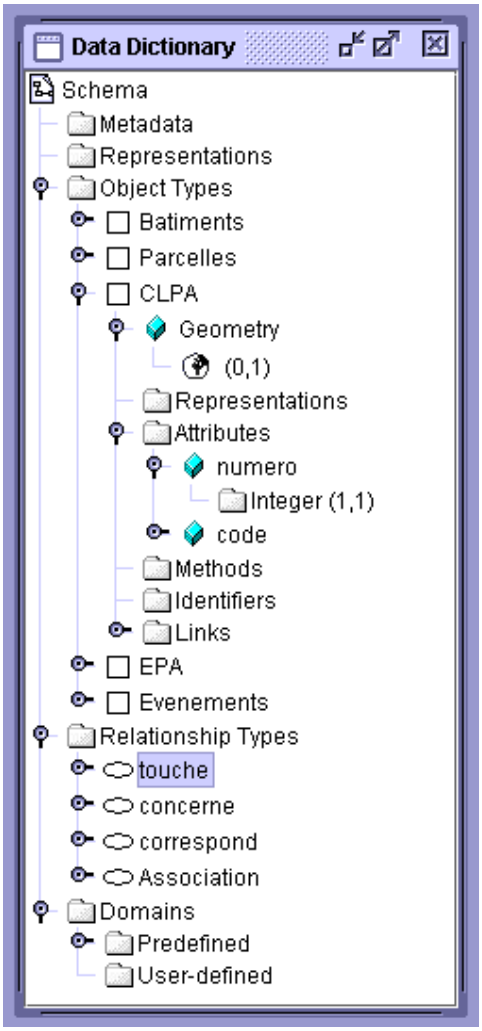
Ceci n'est pas fait dans l'implémentation actuelle de l'éditeur de schémas.



(diagramme UML simplifié pour prendre moins de place...)

4.4 Data dictionary (outline view)

4.4.1 Exemple

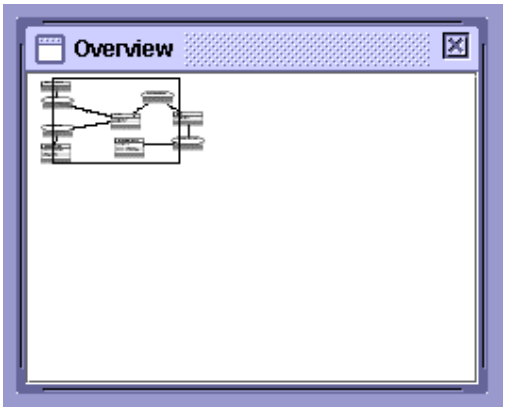


4.4.2 Description

Il s'agit d'un JTree personnalisé. De nouveau, les nœuds du JTree possèdent une référence à l'élément du schéma qu'ils représentent.

4.5 Vue miniature

4.5.1 Exemple



4.6 Commentaires sur l'architecture

Le modèle et la vue sont séparés, ce qui est une bonne chose. Cependant les données concernant l'arrangement spatial des éléments du schéma (layout, dimensions) sont stockées dans la vue, ce qui demande de sérialiser aussi la vue lors de la sauvegarde du schéma sous forme binaire sérialisée (il existe aussi un mode de sauvegarde XML).

La vue possède des références vers le modèle, ce qui est une entorse à l'architecture modèle – vue – contrôleur.

Le modèle ne possède pas de références à sa vue, ce qui est une très bonne chose et va faciliter la réimplémentation de l'éditeur de schémas en permettant son utilisation avec une autre vue et dans un autre contexte. Cependant le modèle ne notifie pas les changements qu'il subit, ce qui est très ennuyeux pour sa réutilisation avec GEF.

Il n'existe pas à proprement parler de contrôleur ou du moins celui-ci est « dilué » dans tout le code de l'application. C'est-à-dire que le code de médiation entre la vue et le modèle n'est pas centralisé dans des classes spécifiques.

Cette architecture pas très propre rend **très difficile la programmation d'un système de redo - undo** et celui-ci n'est donc pas présent.

5 Impact des nouvelles technologies sur le projet MurMur et sur l'éditeur de schémas en particulier

5.1 Eclipse

Voir travail de Sabri Skhiri Dit Gabouje.

5.2 Draw2D

Apports :

- efficacité de la repeinture du diagramme, lors du redimensionnement ou du déplacement d'éléments par exemple (les optimisations possibles signalées sont faites automatiquement si on utilise Draw2D),

- réduction importante de la quantité de code à écrire pour arriver à un résultat similaire,
- possibilité de faire en sorte que même les éléments du schéma de moindre niveau hiérarchique soient sélectionnables et manipulables indépendamment,
- hit – testing déjà implémenté efficacement,
- possibilité de profiter d'une infrastructure classique (genre swing) en matière de layout managers,
- infrastructure importante de gestion des connexions entre éléments et de leurs décorations (flèches, labels...),
- possibilité de profiter d'un grand nombre de figures toutes faites,
- support du zooming, de l'impression, de l'exportation du diagramme vers des fichiers images et des vues miniatures déjà implémenté,
- efficacité, fiabilité (Draw2D a déjà été assez largement testée).

5.3 GEF

Apports :

- Architecture modèle – vue – contrôleur donnant au code une organisation très claire,
- support des undos redos,
- possibilité de profiter d'un grand nombre de choses déjà implémentées, ceci réduisant la quantité de code à produire,
- support des feedbacks visuels durant les actions de l'utilisateur,
- fonctionnalités d' « accessibilité » (pilotage de l'interface par le clavier notamment),
- architecture rendant possible le partage de la tâche de programmation entre différents programmeurs une fois que le modèle est fixé,
- support de toutes les possibilités habituelles d'interaction de l'utilisateur avec l'éditeur graphique (redimensionnement d'éléments, etc.),
- aspect « professionnel » de l'interface,
- efficacité, fiabilité (GEF a déjà été assez largement testée).

5.4 Bémol

Les trois technologies précédentes ne sont pas familières du tout à la large majorité des programmeurs Java. La plate-forme Eclipse est un monde à elle toute seule et force à utiliser la libraire peu commune SWT (et JFace) pour la construction des interfaces graphiques. GEF et Draw2D sont des librairies complexes et encore très peu documentées en regard de la quantité de documentation qui existe sur les technologies java classiques (Swing par exemple).

D'après mon expérience personnelle, je pense que la période d'apprentissage permettant de maîtriser suffisamment ces techniques pour pouvoir réaliser une application vraiment professionnelle qui en exploite toutes les possibilités est probablement de plusieurs années. Il faut donc réfléchir à deux fois avant de s'y lancer et voir si le jeu en vaut vraiment la chandelle.

6 Programmation d'un éditeur graphique utilisant Eclipse, GEF et Draw2D

6.1 Introduction

Ce travail a été réalisé dans le but de :

- me familiariser avec Eclipse, GEF et Draw2D : en effet réaliser le design d'une grosse application sans avoir utilisé au moins une fois chacune des technologies employées est voué à l'échec,

- constituer un exemple très documenté d'éditeur graphique susceptible de servir de point de départ à l'implémentation de l'éditeur de schémas,
- servir de modèle miniature pour le design de l'éditeur de schémas.

6.2 Description

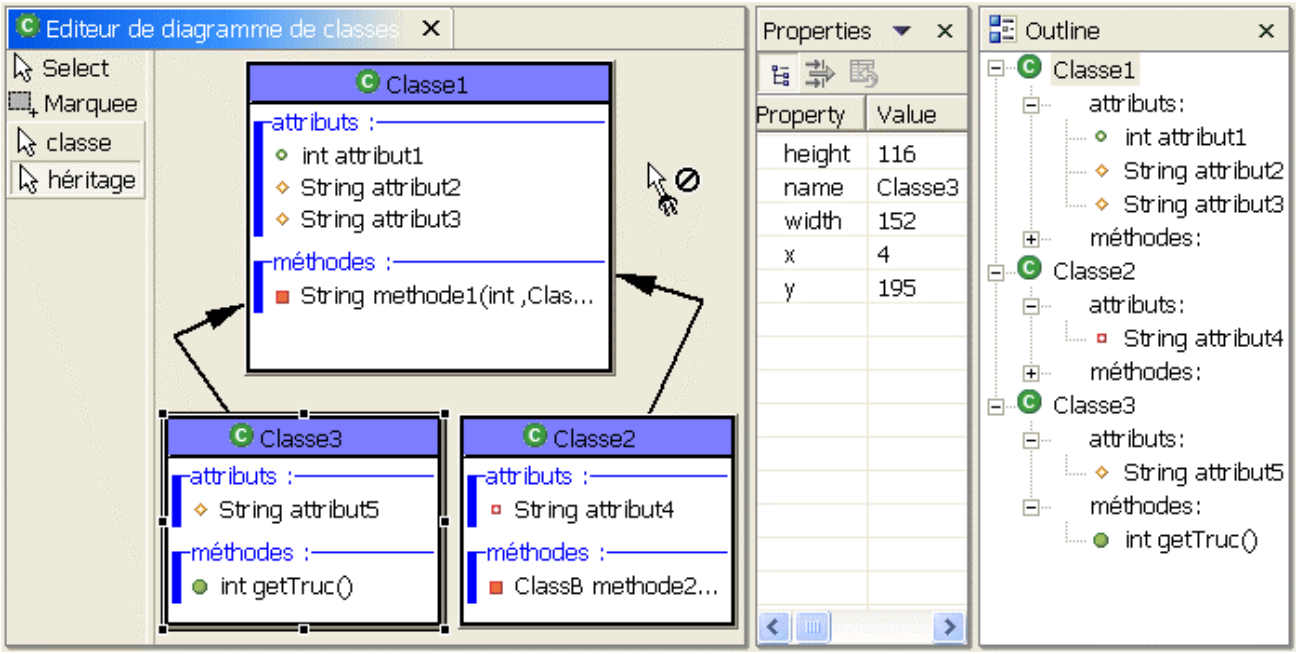
Il s'agit d'un éditeur de diagrammes de classes UML très simplifié. Le schéma créé est représenté par un modèle orienté objet, comme c'est le cas pour l'éditeur de schémas du projet MurMur.

Fonctionnalités

- création / suppression des différents éléments,
- déplacement, redimensionnement des représentations graphiques de ces éléments,
- « reparenting » par drag & drop de ces éléments,
- modifications des propriétés de ces éléments,
- présence d'une représentation arborescente du modèle,
- présence de menus contextuels sur le diagramme de classes et sur la représentation arborescente.
- support des undos – redos,
- barres de scrolling apparaissant dans la représentation graphique d'une classe si la place est insuffisante pour montrer la totalité des attributs et des méthodes de cette classe,
- possibilité de sélectionner et de manipuler plusieurs éléments à la fois,
- support des raccourcis clavier habituels (undo : ctrl+z, redo : ctrl+y et delete : del).

Tout en le codant j'ai rédigé un document détaillé sur les différentes étapes suivies et sur l'analyse orientée objet du programme. Il n'est pas présenté ici faute de place.

6.3 Aperçu du résultat



7 Réimplémentation de l'éditeur de schémas

7.1 Introduction

Cette partie devrait comporter :

- une analyse fonctionnelle de l'éditeur de schémas et ses spécifications,
- une analyse orientée objet du futur programme,
- un prototype reprenant les fonctionnalités de base.

L'analyse fonctionnelle de l'éditeur de schémas et ses spécifications sont inchangées par rapport à l'ancien éditeur et font l'objet d'un document faisant partie de la documentation du projet MurMur. Il n'y a rien à ajouter à ce sujet, le document en question est très complet.

Pour ce qui est de l'analyse orientée objet, l'architecture de toutes les applications GEF est semblable, de sorte que cette analyse orientée objet n'a pas à être faite explicitement si on connaît GEF. GEF ne laisse aucune marge de manœuvre de ce point de vue. Une fois que le modèle et les fonctionnalités d'éditions souhaitées sont définis (ce qui est le cas ici) le reste (EditParts, Figures, Commands, EditPolicies) suit directement.

7.2 Description du prototype et de ses fonctionnalités actuelles

7.2.1 Ouverture des fichiers .mur2 et affichage du diagramme

Ceci est entièrement supporté par le prototype.
Tous les fichiers au format .mur2 créés à l'aide de l'ancien éditeur de schémas peuvent être ouverts avec le nouveau et un diagramme équivalent est affiché.

7.2.2 Sauvegarde du schéma

Ceci est entièrement implémenté.

7.2.3 Possibilités d'édition

Bien qu'un schéma d'une complexité quelconque puisse être ouvert à l'aide de l'éditeur, pour l'instant les possibilités d'éditions sont très limitées. Elles se limitent à la création de nouveaux éléments (type d'objets, de relation, héritages, rôles...), à leur suppression, à la reconnexion de rôles ou de relation d'héritage, et au déplacement des éléments dans le diagramme.

En particulier les propriétés de chaque élément ne peuvent pas encore être modifiées. Cependant cette partie devrait être simple à réaliser et ne nécessite pas une connaissance approfondie de GEF, Draw2D ou Eclipse.

7.2.4 Validation du schéma

Ceci est entièrement implémenté.

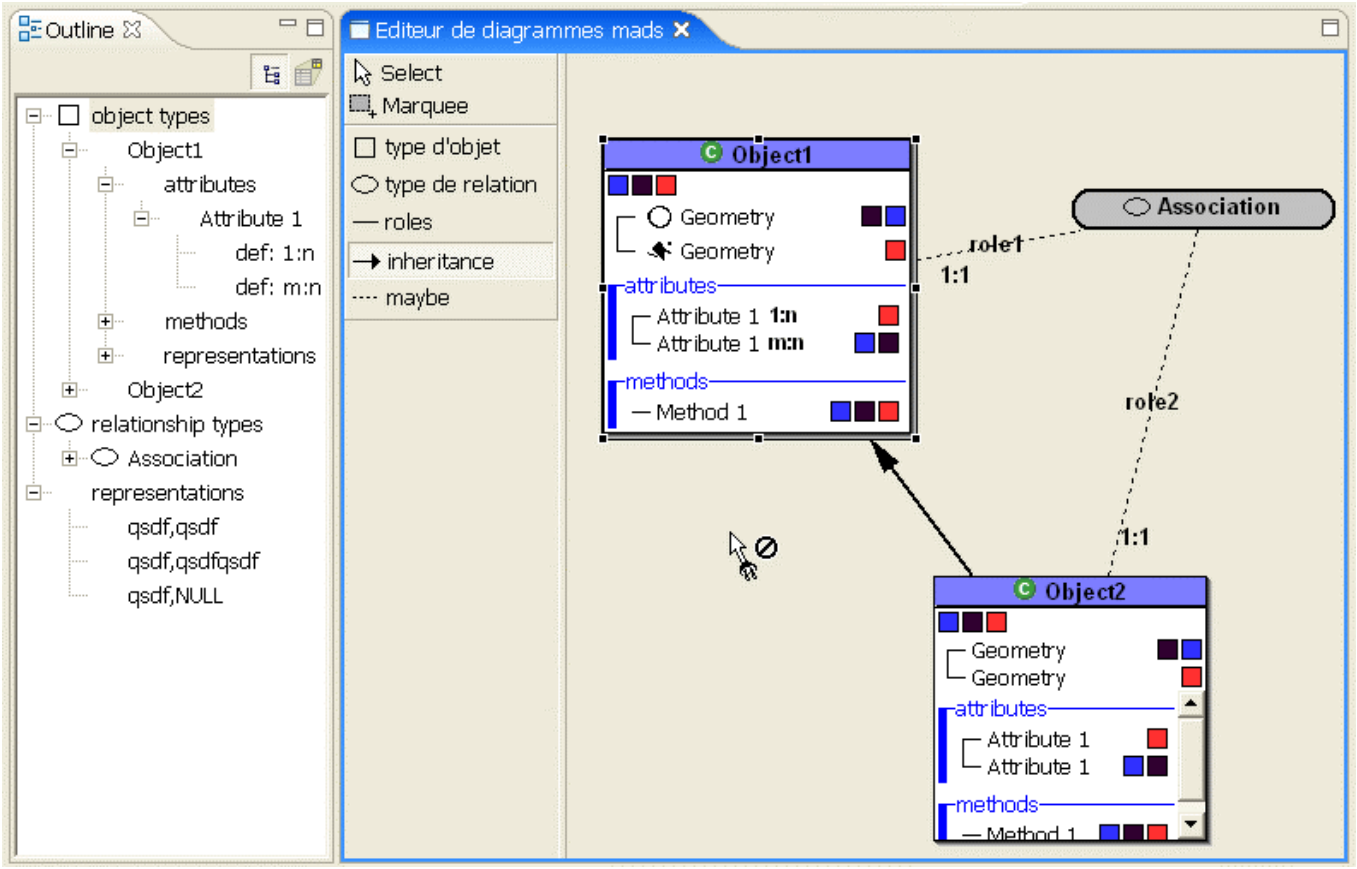
7.2.5 Représentation graphique de l'arborescence du modèle

Ceci est entièrement implémenté mais devra être optimisé si les performances se révèlent insuffisantes.

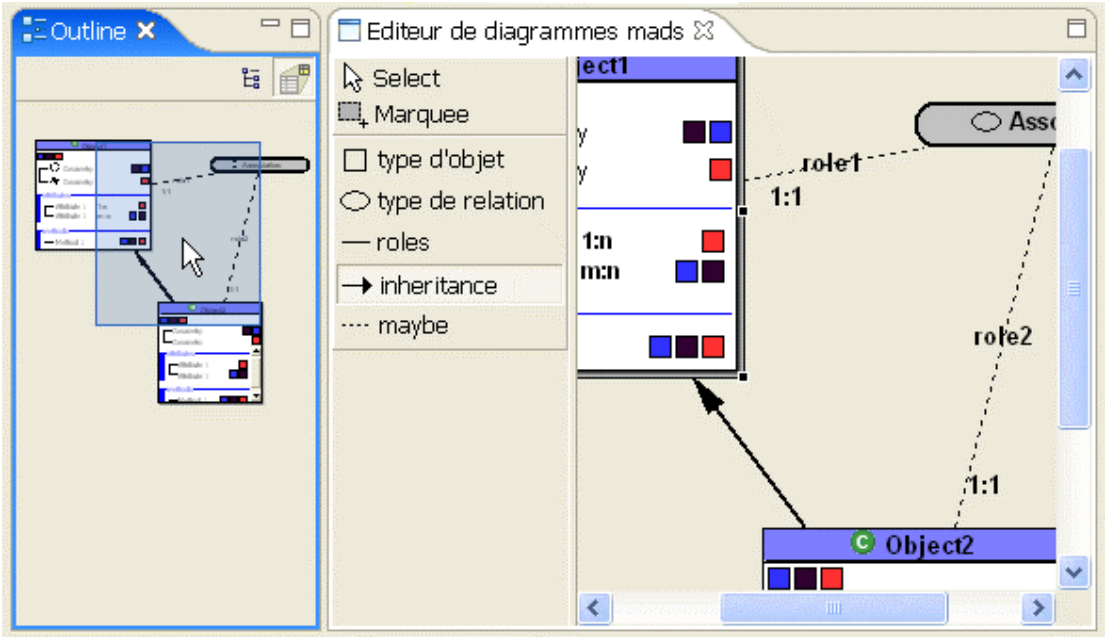
7.2.6 Représentation miniature du diagramme avec possibilité de navigation, zooming

Ceci est entièrement implémenté.

7.2.7 Screenshots



Avec la vue arborescente du modèle sur la gauche.



Avec la vue miniature sur la gauche.

Ces screenshots n'illustrent pas la barre d'outils supérieure qui comprend les boutons pour le zoom, la validation, delete et undo/redo (non implémenté mais les boutons et leurs effets sont mis en place).

7.3 Implémentation

Cette section décrit l'implémentation de l'éditeur de schémas et concerne le développeur qui souhaiterait y apporter des modifications ou l'enrichir de nouveaux éléments. La connaissance de Draw2D, GEF et une connaissance de base du projet MurMur est préférable pour comprendre ce qui suit.

7.3.1 Modèle

(paquet mads.tstructure.core)

Il s'agit presque intégralement du modèle de l'ancien éditeur de schémas.

Deux choses ont dû être ajoutées :

- pour les classes du modèle qui le nécessitent, des données membres représentant les propriétés des représentations graphiques de leurs instances ont été ajoutées (il y en a peu, si bien que l'augmentation de complexité due au stockage séparé des propriétés graphiques ne me semble pas justifiée),
- un mécanisme de notification, pour pouvoir utiliser le modèle avec GEF.

Le mécanisme de notification fournit la possibilité d'enregistrer sur les différents objets du modèle des listeners (les contrôleurs : EditParts) qui seront avertis des changements du modèle par des événements. Seules les modifications des propriétés qui possèdent une représentation graphique et les modifications des propriétés structurelles (relations parent – enfant, connections) font l'objet d'événements.

Le mécanisme de notification retenu est celui des javabeans fourni dans le paquet java.beans : PropertyChangeSupport, PropertyChangeListener et PropertyChangeEvent.

Chaque classe du modèle est munie :

- d’un objet PropertyChangeSupport qui maintient la liste des listeners ajoutés,
- de méthodes d’ajout et de retrait de listeners,
- de constantes publiques statiques constituant les identificateurs des différentes propriétés écoutées,
- d’appels à propertyChangeSupport.firePropertyChange(PropertyChangeEvent) chaque fois qu’une propriété qui fait l’objet d’une écoute est modifiée.

Voir code et documentation de l’ancien éditeur de schémas pour les détails à propos du modèle. Il suffit de regarder les constantes publiques statiques des différentes classes du modèle pour savoir quelles sont les propriétés qui peuvent être écoutées.

7.3.2 Vue

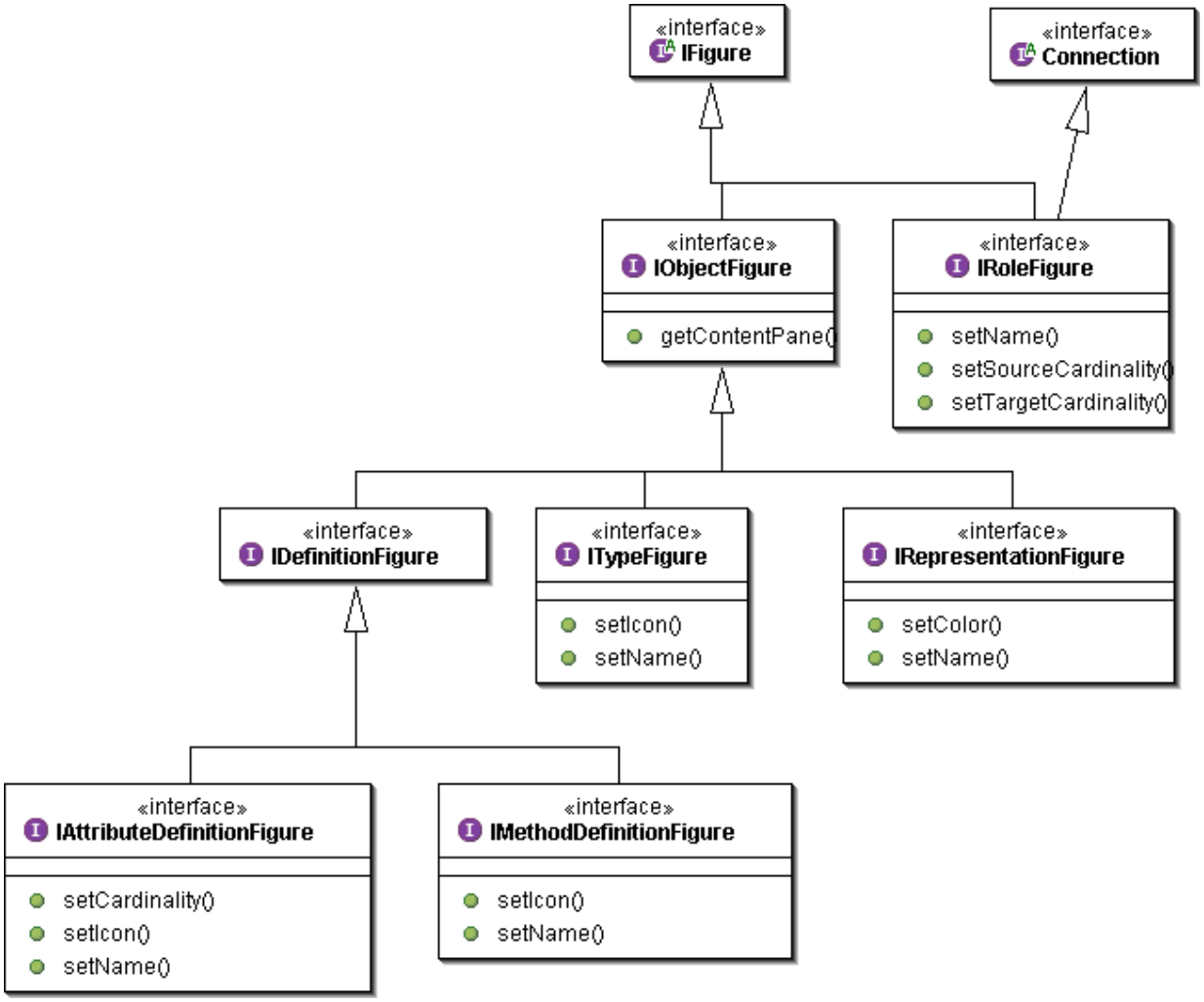
(paquet mads.editor.figures)

Cette section contient des informations sur les classes constituant la représentation graphique du modèle.

7.3.2.1 Interfaces

Les contrôleurs accèdent aux propriétés graphiques des différents éléments de la vue uniquement par des interfaces de façon entre autres à rendre le plus facile le remplacement de certains éléments de la vue par d’autres implémentant les mêmes interfaces.

7.3.2.1.1 Diagramme de classes



7.3.2.1.2 Explications

IFigure est l'interface définie par Draw2D pour accéder aux propriétés génériques des Figures.

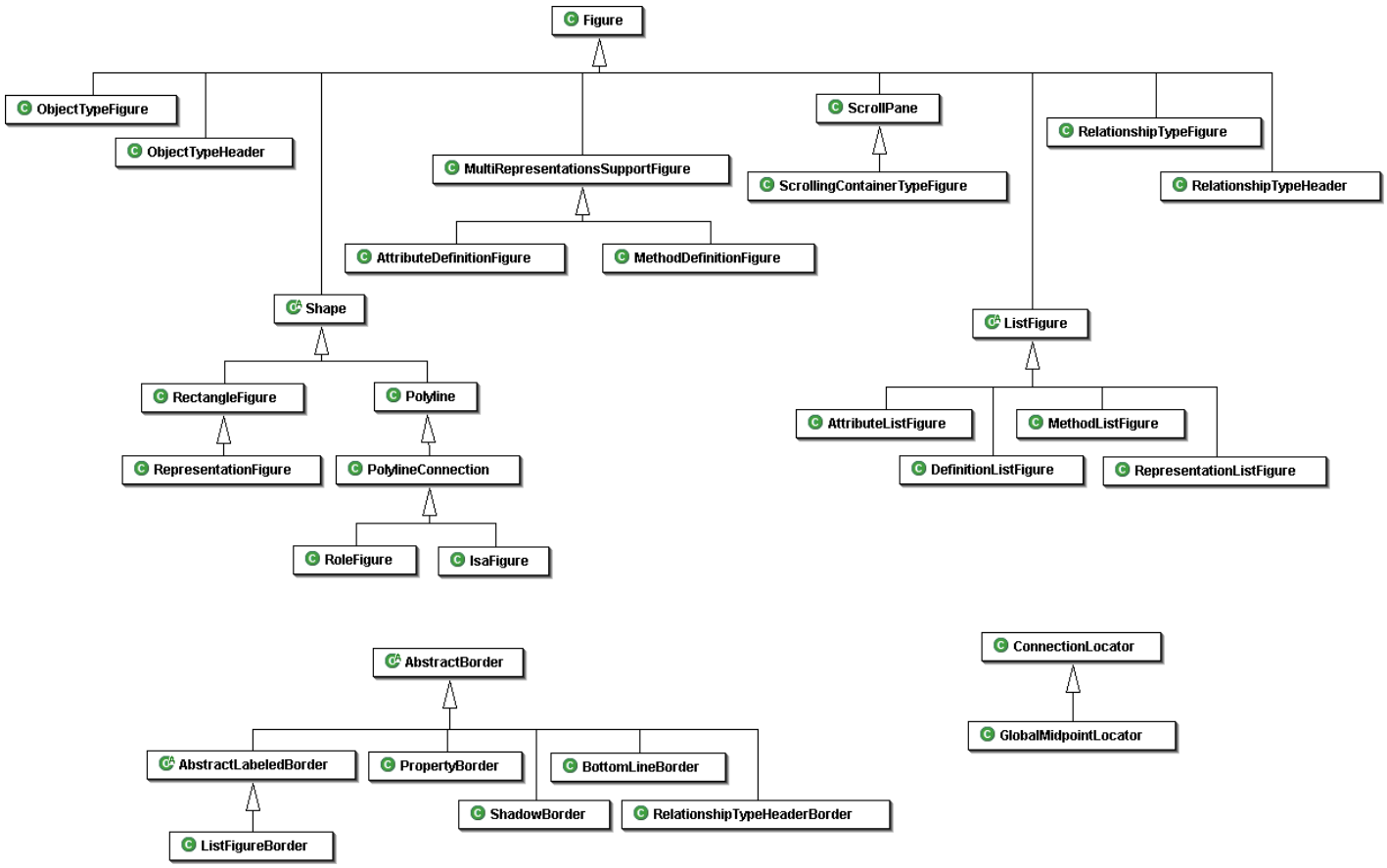
IObjectFigure est l'interface qui doit être implémentée par toute IFigure liée à un objet du modèle de l'éditeur par un EditPart. La méthode *getContentPane()* doit renvoyer une IFigure : soit la figure courante, soit une de ses descendantes. Cette méthode sera appelée par l'EditPart liant la figure à son modèle pour obtenir le content pane qui sera utilisé par GEF comme parent pour les représentations des enfants du modèle.

Les noms des autres interfaces les rendent « self-descriptive ». Elles constituent les interfaces qui doivent être implémentées par les figures représentatives des différents éléments du modèle pour permettre l'accès à leurs propriétés graphiques spécifiques.

7.3.2.2 Figures

Cette section détaille les classes dérivées de Figure qui servent à représenter les différents éléments du modèle. Ce sont principalement les implémentations des interfaces définies ci-dessus.

Diagramme de classes (une partie des classes illustrées ici fait partie de Draw2D) :



7.3.2.2.1 class RepresentationFigure (extends RectangleFigure)

Les représentations sont des éléments du modèle qui possèdent un nom (point de vue + résolution) et une couleur et sont représentés par un petit carré coloré, avec un tooltip affichant le nom de la représentation lorsque le curseur s'arrête au-dessus de la figure.

Exemple :

Draw2D fournit une classe pour afficher de tels rectangles colorés : RectangleFigure.
RepresentationFigure étend cette classe.

Les représentations sont conçues pour être affichées dans un container muni d'un ToolbarLayout. ToolbarLayout dispose les enfants sur base de leur preferred size et de leur minimum size. Pour fixer la taille d'un enfant il suffit de lui attribuer un preferred size et un minimum size identiques et de la dimension voulue. C'est de cette manière que la taille des petits carrés colorés est définie : en redéfinissant les méthodes getPreferredSize(...) et getMinimumSize(...) pour qu'elles retournent la valeur de taille adéquate (peut-être aurait-il suffi d'appeler setPreferredSize et setMinimumSize...).

L'interface publique d'accès aux propriétés de la figure est :

- **setName(String)** : pour ajuster le texte du tooltip,

- **setColor(Color)** : pour ajuster la couleur de la représentation.

7.3.2.2.2 class MultiRepresentationSupportFigure (extends Figure)

De nombreux éléments du modèle possèdent plusieurs représentations (au sens du projet MurMur, pas au sens « représentation graphique ») qui doivent être illustrées avec les autres propriétés de l'élément. Le rôle de cette classe est de fournir un support de base pour les vues de tels éléments.

Cette figure possède deux containers enfants appelés representationsContainer et principalContainer. Ces deux containers sont munis de ToolbarLayouts horizontaux.

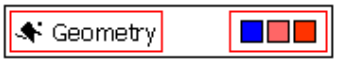
Le layout automatique est défini de manière à ce que ces deux containers :

- aient pour hauteur leur preferred size et soient centrés en hauteur sur la médiane horizontale de la figure,
- se partagent la largeur de la manière suivante : le representationContainer a pour largeur sa preferred size et est aligné à droite, le principalContainer est aligné à gauche et occupe le reste de la largeur disponible (mais jamais plus de sa preferred size).

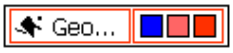
La preferred size est calculée de manière à renvoyer en hauteur la preferred size du plus haut des deux containers et en largeur (à peu près...) la somme des preferred size de chaque container.
La minimum size possède une largeur nulle.

Voilà un exemple d'une classe dérivée de MultiRepresentationSupportFigure (les deux containers sont ici bordés de rouge et la figure elle-même est bordée de noir, pour montrer les limites des différents éléments) :

- lorsqu'il y a suffisamment de place pour afficher les deux containers avec leur preferred size :



- lorsqu'il n'y a pas suffisamment de place (le representationContainer écrase l'autre) :



Les méthodes :

- **IFigure getRepresentationContainer()**
- **IFigure getPrincipalContainer()**

permettent d'obtenir les références aux deux containers enfants et d'y ajouter des éléments.

L'utilisation classique est d'étendre cette classe pour introduire systématiquement un certain nombre d'éléments dans le principalContainer et de réserver le representationContainer pour servir de content pane à un EditPart dont les enfants (getModelChildren()) sont des représentations. La méthode **getContentPane()** renvoie donc le representationContainer.

7.3.2.2.3 class AttributeDefinitionFigure (extends MultirepresentationSupportFigure)

Cette classe définit la vue d'un attribut. Elle étend MultiRepresentationSupportFigure et ajoute au principalContainer :

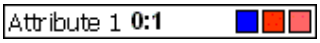
- un Label pour représenter le nom de l'attribut,

- un Label en gras pour représenter la cardinalité de l'attribut.

Les méthodes d'accès aux propriétés de la figure sont :

- **void setName(String)** : pour ajuster le nom de l'attribut,
- **void setIcon(Image)** : pour ajouter éventuellement une petite icône avant le nom de l'attribut.
- **void setCardinality(String min, String max)** : pour spécifier la cardinalité de l'attribut.

ex :



7.3.2.2.4 class MethodDefinitionFigure (extends MultiRepresentationSupportFigure)

Cette classe définit la vue d'une méthode. Elle étend MultiRepresentationSupportFigure et ajoute un Label au principalContainer pour représenter le nom de la méthode.

Les méthodes d'accès aux propriétés de la figure sont :

- **void setName(String)** : pour ajuster le nom de la méthode,
- **void setIcon(Image)** : pour ajouter éventuellement une petite icône avant le nom de la méthode.

7.3.2.2.5 abstract class ListFigure (extends Figure)

Beaucoup d'éléments du modèle doivent être représentés sous forme de listes : listes de représentations, d'attributs, de méthodes, de définitions d'attribut, de définitions de méthode, etc. La classe ListFigure fournit un support de base pour représenter ce genre de listes.

Dans le cadre de cet éditeur, les représentations des listes doivent avoir certaines propriétés communes :

- notifier leurs écouteurs lorsque des Figures enfants sont ajoutées ou supprimées,
- le bord de la liste doit être masqué lorsqu'il n'y a pas d'enfant dans la liste.

La classe ListFigure possède ces deux propriétés.

ListFigure est munie d'un ToolbarLayout.

La méthode abstraite **Border createBorder()** doit être redéfinie dans les sous-classes pour fournir le bord. Celui-ci *ne* peut *pas* être ajusté par appel à setBorder(Border).

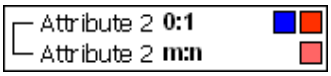
La méthode **getContentPane()** renvoie *this*.

7.3.2.2.6 class DefinitionListFigure (extends ListFigure)

Figure utilisée pour représenter un attribut ou une méthode. Les représentations de ces deux éléments sont formées de la liste des vues de leurs « définitions » (au sens de MurMur). La classe DefinitionListFigure sert à représenter une telle liste.

La méthode createBorder() de ListFigure est redéfinie pour retourner un PropertyBorder (voir code pour la classe TreeBorder, le résultat est assez parlant pour comprendre ce dont il s'agit). Le ToolbarLayout est ajusté pour être vertical.

Ci-dessous on voit une DefinitionListFigure utilisée pour représenter un attribut. Les deux définitions de l'attribut sont représentées par des AttributeDefinitionFigure. Le PropertyBorder lie les deux définitions, en noir, à gauche :

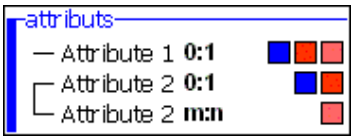


7.3.2.2.7 class AttributeListFigure (extends ListFigure)

Figure utilisée pour représenter la liste des attributs d'un type d'objet ou de relation.

La méthode createBorder() de ListFigure est redéfinie pour retourner un ListFigureBorder (voir code pour la classe ListFigureBorder) bleu et avec pour titre « attributs ». Le ToolbarLayout est ajusté pour être vertical.

Exemple :



7.3.2.2.8 class MethodListFigure (extends ListFigure)

Idem AttributeListFigure, mais pour une liste de méthodes. Seul le titre du bord diffère.

7.3.2.2.9 class RepresentationListFigure (extends ListFigure)

Figure utilisée pour représenter la liste des représentations d'un type d'objet ou d'un type de relation.

Le ToolbarLayout est ajusté pour être horizontal et la méthode createBorder() est redéfinie pour renvoyer un MarginBorder de largeur 2.

Exemple :

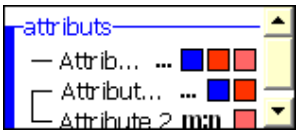
7.3.2.2.10 class ScrollingContainerTypeFigure (extends ScrollPane)

La liste d'attributs et la liste de méthodes d'un type d'objet ou d'un type de relation doivent être représentées l'une au-dessus de l'autre dans un ScrollPane particulier. La classe ScrollingContainerTypeFigure constitue un tel ScrollPane.

Le ScrollPane est ajusté pour que le contenu ne soit muni d'une ScrollBar que sur sa hauteur, la largeur du contenu étant toujours la même que celle du ScrollPane (meth. ScrollPane.getViewPort().setContentTracksWidth(true)).

Un content pane est ajouté au ViewPort du ScrollPane. Il est muni d'un ToolbarLayout vertical. La liste d'attributs et la liste de méthodes sont ajoutés à ce content pane. La méthode **getContentPane()** renvoie ce content pane.

Exemple :



7.3.2.2.11 class ObjectTypeHeader (extends Figure)

Les types d'objets (~= entités) définis dans le modèle possèdent chacun un nom. Ce nom doit être affiché dans l'en-tête de la vue du type d'objet avec une petite image à côté. La classe ObjectTypeHeader définit cet en-tête.

Cette Figure est munie d'un ToolbarLayout bien qu'elle n'ait pour enfant qu'un seul Label affichant le nom du type d'objet. L'utilisation d'un ToolbarLayout malgré le fait qu'il n'y ait qu'un seul enfant permet le calcul automatique de la preferred size de la figure en fonction de celle du Label et permet de centrer le label dans la Figure. (peut-être qu'un layout manager plus simple existe pour remplir le même rôle)

Cette Figure possède un BottomLineBorder (voir code pour cette classe) pour afficher une ligne noire à sa base.

Exemple :

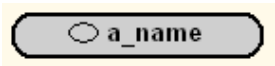
Il est possible d'ajuster le nom et l'image du label par les méthodes publiques :

- **void setName(String)**
- **void setIcon(Image)**

7.3.2.2.12 class RelationshipTypeHeader (extends Figure)

Le principe est le même que pour ObjectTypeHeader mais pour les relations le bord utilisé est différent (RelationshipTypeHeaderBorder).

Exemple :



7.3.2.2.13 class ObjectTypeFigure (extends Figure)

Figure utilisée pour représenter les types d'objets définis dans le modèle.

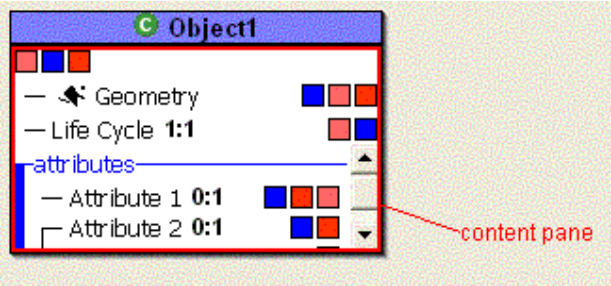
Elle est munie d'un ToolbarLayout vertical auquel sont ajoutés :

- le header,
- le content pane.

Le content pane est une Figure munie d'un ToolbarLayout vertical. La méthode **getContentPane()** renvoie ce content pane.

La figure est munie d'un ShadowBorder (voir code pour cette classe).

Exemple :



Il est possible d'ajuster le nom et l'icône du type d'objet grâce aux méthodes publiques :

- **void setName(String)**
- **void setIcon(Image)**

7.3.2.2.14 class RelationshipTypeFigure (extends Figure)

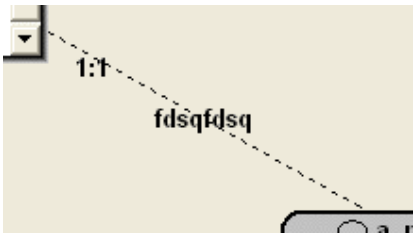
Similaire a ObjectTypeFigure mais avec un RelationshipTypeHeader.

7.3.2.2.15 class RoleFigure (extends PolylineConnection)

Figure utilisée pour représenter les rôles, c'est-à-dire les connections entre un type de relation et un type d'objet.

Il s'agit d'une PolylineConnection ayant pour enfants trois Labels positionnés automatiquement par des Locators. Ces trois labels permettent d'ajuster les cardinalités des « extrémités » du rôle et son nom.

Exemple :

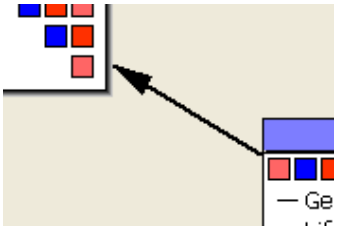


7.3.2.2.16 class IsaFigure (extends PolylineConnection)

Figure utilisée pour représenter les relations d'héritage.

Il s'agit d'une PolylineConnection munie d'une décoration en forme de flèche.

Exemple :



7.3.2.3 Contrôleurs

(paquet mads.editor.editparts)

Cette section contient des informations concernant les EditParts.

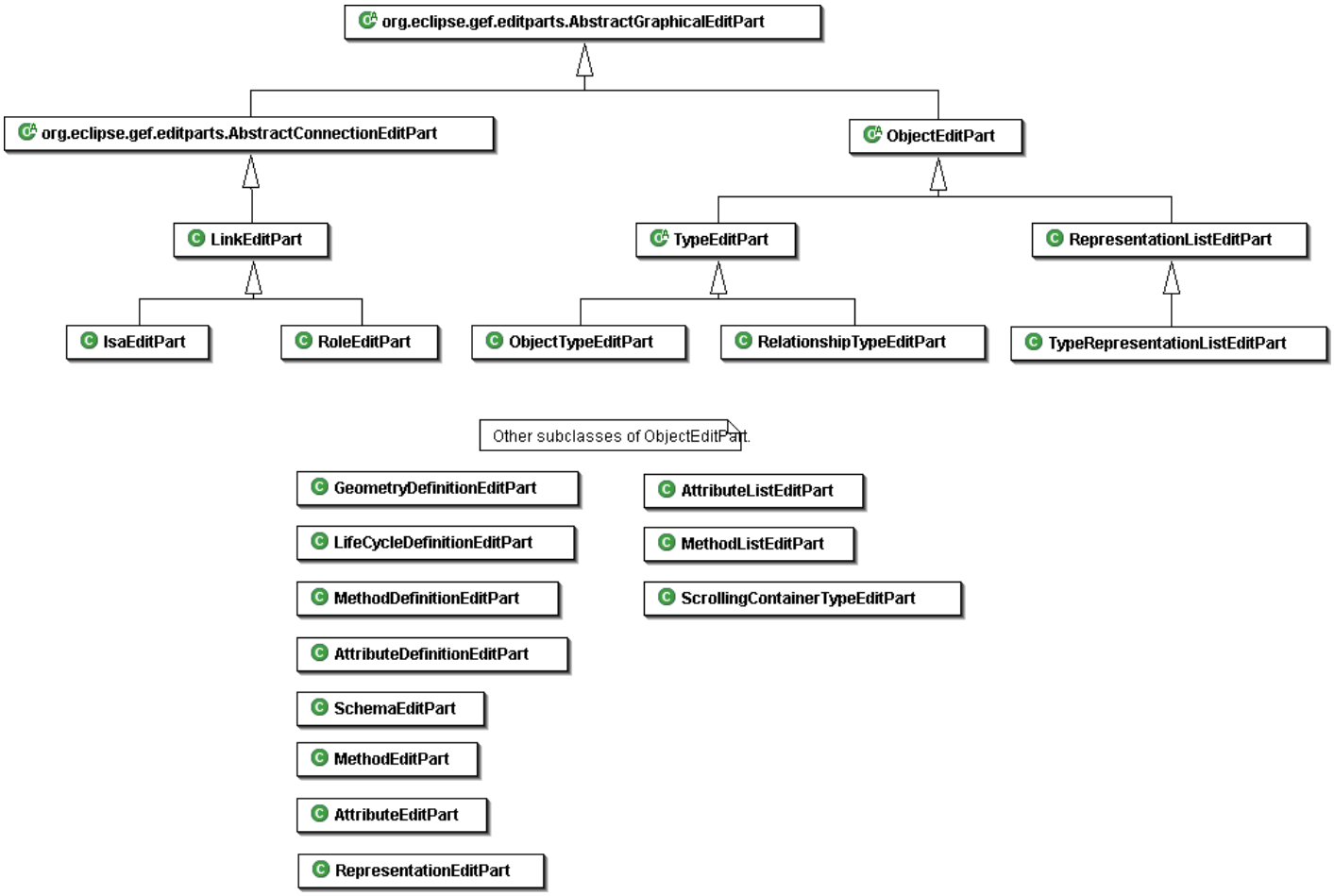
Il existe théoriquement une sous-classe d'EditPart et une sous-classe de Figure par classe du modèle dont les instances doivent être représentées et manipulées individuellement (sélectionnées, déplacées, etc.).

Pour chaque EditPart, il est important de savoir :

- à quel objet du modèle il va être associé (déterminé par l'appel à EditPart.setModel(..) lors de la création de l'EditPart),
- à quelle Figure il va être associé (déterminé par la définition de la méthode EditPart.createFigure()),
- quels sont les objets du modèle qui vont être représentés par GEF en tant qu'enfants du content pane de sa figure (déterminé par la définition de la méthode List getModelChildren()), et quels EditParts seront créés pour eux (déterminé par la définition de la méthode EditPart createChild(Object) ou par l'EditPartFactory),
- quelles fonctionnalités d'édition il supporte (déterminé par la définition de la méthode createEditPolicies()).

7.3.2.3.1 Hiérarchie

Diagramme de classes des EditParts.



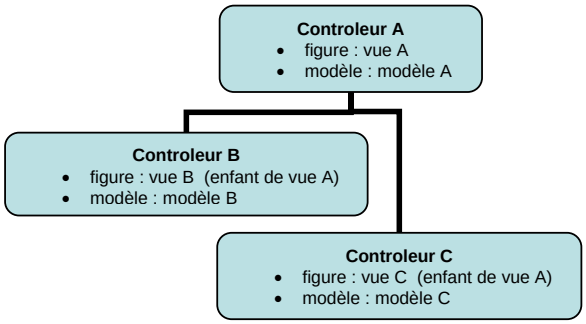
Les noms des différentes classes d'EditPart indiquent de quel type d'objets du modèle ils sont les contrôleurs. Les méthodes sont toujours les mêmes puisque ce sont simplement les méthodes abstraites de la classe AbstractGraphicalEditPart qui doivent être définies.

7.3.2.3.2 Arborescence

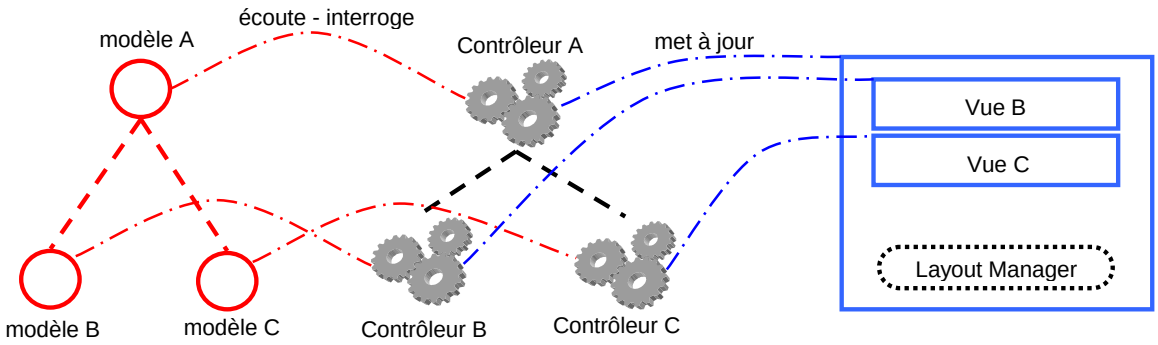
Pendant l'exécution de l'éditeur, les instances des classes d'EditParts forment une arborescence (voir intro sur GEF). GEF construit automatiquement l'arborescence des Figures qui sont associées aux EditParts de manière symétrique à l'arborescence des EditParts (relation parent – enfant dans l'arborescence des EditParts => relation parent – enfant et donc contenant – contenu pour les figures associées).

Les schémas suivants illustrent cette arborescence. Pour chaque EditPart est indiqué l'objet du modèle et la figure entre lesquels il joue le rôle de contrôleur. Puisque l'arborescence des Figures est symétrique à celle des EditParts, les schémas suivants illustrent aussi la structure de la vue.

7.3.2.3.2.1 Exemple

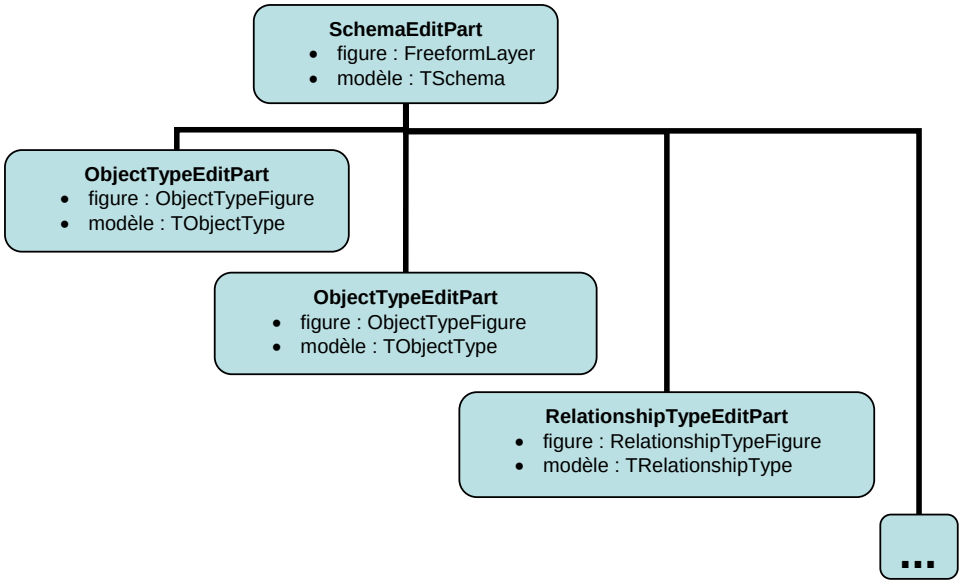


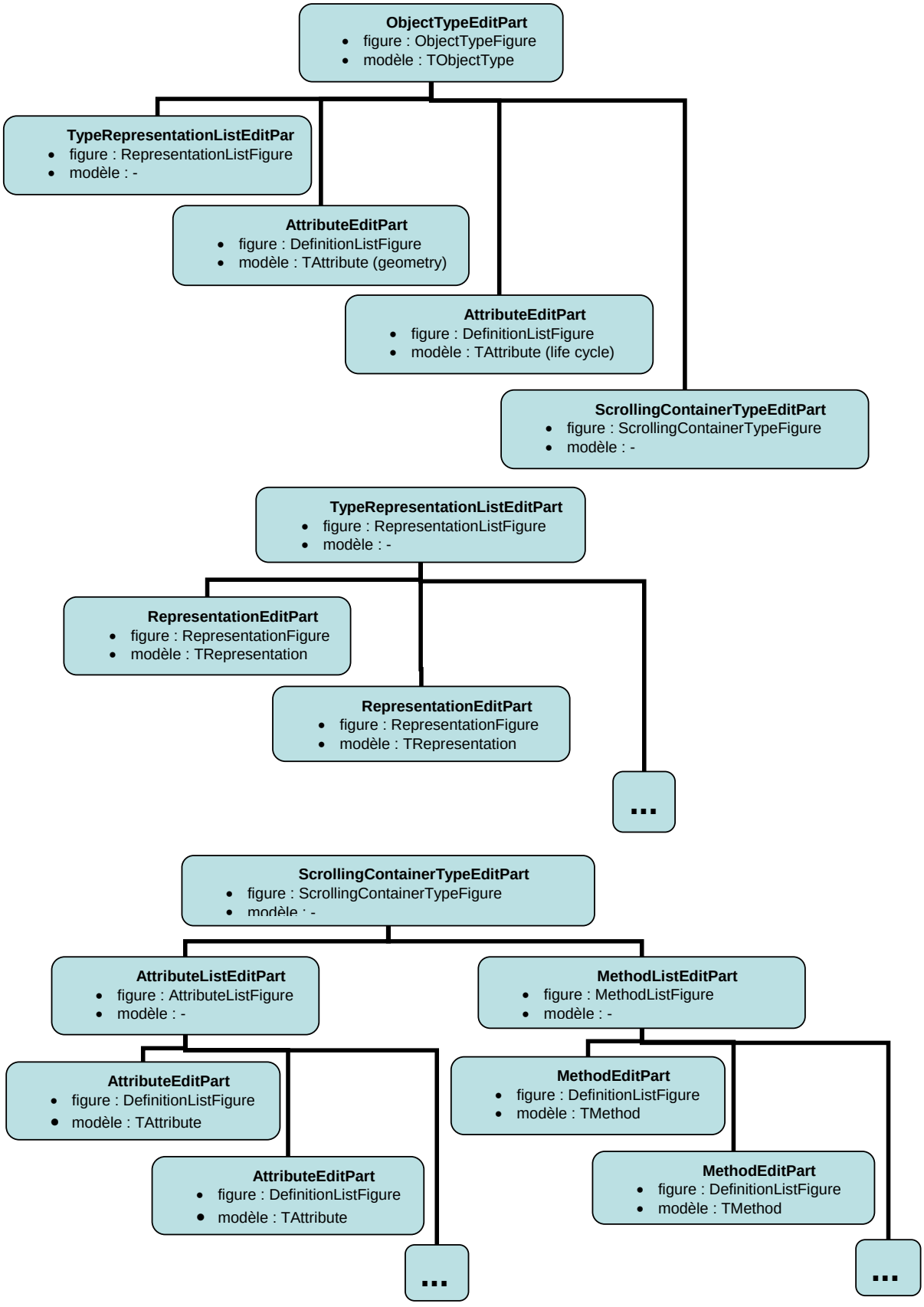
Signifie **:**

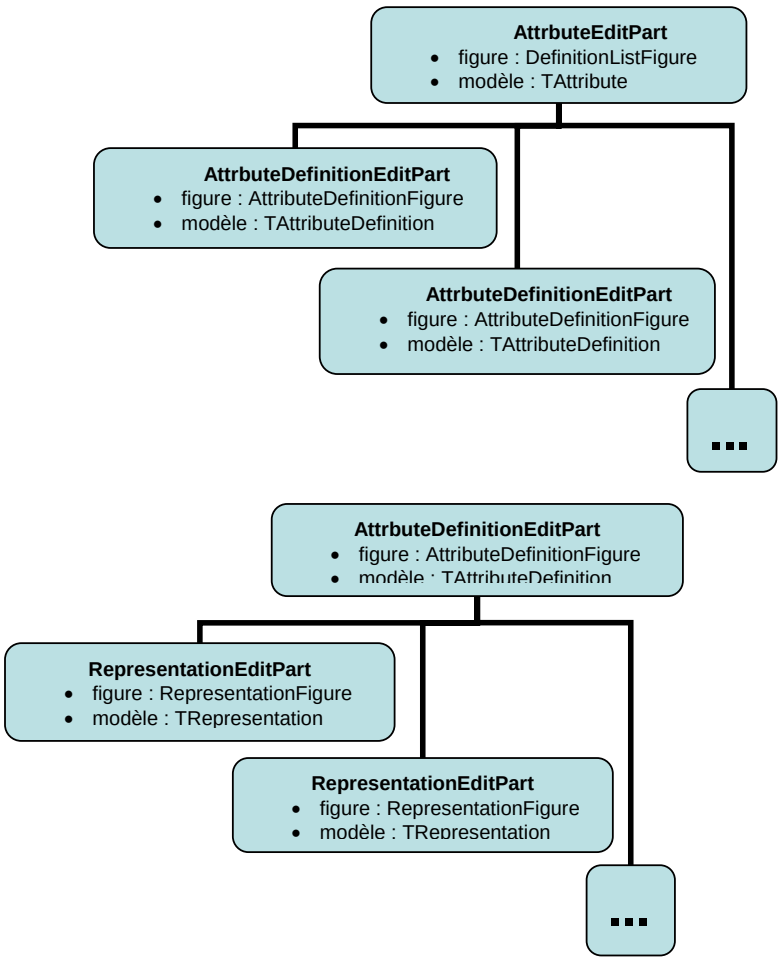


7.3.2.3.2.2 Diagrammes

(l'arbre est trop grand pour être représenté en entier, il est donc découpé en sous-arbres)




ObjectTypeEditPart
figure : ObjectTypeFigure
modèle : TObjectType
TypeRepresentationListEditPar
figure : RepresentationListFigure
modèle : -
AttributeEditPart
figure : DefinitionListFigure
modèle : TAttribute (geometry)
AttributeEditPart
figure : DefinitionListFigure
modèle : TAttribute (life cycle)
ScrollingContainerTypeEditPart
figure : ScrollingContainerTypeFigure
modèle : -
TypeRepresentationListEditPart
figure : RepresentationListFigure
modèle : -
RepresentationEditPart
figure : RepresentationFigure
modèle : TRepresentation
RepresentationEditPart
figure : RepresentationFigure
modèle : TRepresentation
...
ScrollingContainerTypeEditPart
figure : ScrollingContainerTypeFigure
AttributeListEditPart
figure : AttributeListFigure
modèle : -
MethodListEditPart
figure : MethodListFigure
modèle : -
AttributeEditPart
figure : DefinitionListFigure
modèle : TAttribute
AttributeEditPart
figure : DefinitionListFigure
modèle : TAttribute
...
MethodEditPart
figure : DefinitionListFigure
modèle : TMethod
MethodEditPart
figure : DefinitionListFigure
modèle : TMethod
...



7.3.2.4 Sauvegarde / ouverture

7.3.2.4.1 Ancien format de sauvegarde (.mur2)

Les fichiers .mur2 sont composés de deux fichiers XML zippés, un pour le schéma lui-même et un pour les propriétés graphiques du diagramme. Le fichier XML de sauvegarde des propriétés graphiques est trop pauvre pour pouvoir être conservé tel quel. En particulier, il ne permet pas la sauvegarde des tailles des différents éléments du diagramme mais uniquement de leurs positions.

Tous les schémas sauvegardés au format .mur2 par l'ancien éditeur peuvent être ouverts avec le nouveau.

7.3.2.4.2 Nouveau format de sauvegarde (.mur3)

Les fichiers .mur3 permettent de sauvegarder les propriétés graphiques supplémentaires propres au nouvel éditeur. Pour l'instant seul le stockage de la taille des éléments a été rajouté. La DTD a été changée en ce sens.

Il est possible de sauvegarder un fichier au format mur2 ou mur3 avec le nouvel éditeur. Si on sauve au format mur2, les indications de taille des éléments du diagramme sont perdues.

7.3.2.4.3 Classes impliquées dans la sauvegarde / ouverture

(paquet mads.editor.importexport)

Sauvegarde

La classe **ExportSchema** possède deux méthodes statiques publiques pour sauvegarder le schéma. On passe en paramètre un fichier ou un chemin d'accès et la racine du modèle (le TSchema). Le schéma est alors sauvegardé au format mur2 ou mur3 selon l'extension du fichier passé en paramètre. C'est la seule chose à savoir pour l'utilisation.

La classe ExportSchema s'aide d'une instance de la classe **SchemaStructureToMurx** pour créer le fichier XML comprenant la structure du schéma et d'une instance des classes **GraphicalPropertiesToMur2** ou **GraphicalPropertiesToMur3** pour créer le fichier XML comprenant les propriétés graphiques du diagramme.

Ouverture

La classe **ImportSchema** possède une méthode statique publique qui prend en paramètre un fichier .mur2 ou .mur3, construit le TSchema correspondant et le retourne.

La classe ImportSchema s'aide d'une instance de la classe **MurxToSchemaStructure** pour créer le TSchema à partir du fichier XML décrivant sa structure et d'une instance des classes **Mur2ToGraphicalProperties** ou **Mur3ToGraphicalProperties** pour compléter le TSchema avec les propriétés graphiques des éléments.

7.4 Améliorations à apporter dans l'avenir

7.4.1 Format de sauvegarde

Il faudrait inclure dans les fichiers .mur3 (et dans le modèle) la possibilité de stocker bien d'autres propriétés graphiques correspondants à des choses non encore implémentées dans l'éditeur .

Par exemple :

- les bendpoints des connections, et éventuellement le type de routeur auquel les connections sont associées et le type de ConnectionAnchor à utiliser à chaque extrémité,
- le fait que certains éléments puissent être masqués dans le schéma pour faciliter la lisibilité ou pour réaliser une impression partielle,
- certaines préférences de l'utilisateur, comme des couleurs particulières, etc...

7.4.2 Edition

Il faudrait implémenter ce que GEF appelle le direct editing pour permettre de modifier les propriétés des différents éléments du schéma par double clic sur leurs représentations graphiques. GEF fournit un support important pour cela, ce ne devrait donc pas être difficile. Cela devrait être encore simplifié par le fait qu'il suffit de modifier le modèle sans se soucier de mettre à jour la vue et les contrôleurs parce que grâce au système d'événements – listeners, ceci se fera automatiquement.

7.4.3 Gestion de ce que MurMur appelle les « domaines »

Ceci a été mis de côté pour l'implémentation du prototype.

7.4.4 Intégration plus étroite à Eclipse

Contributions à l’aide de la plateforme, wizards pour créer de nouveaux schémas, contributions aux préférences, toolbars, menus, etc...

Sources

GEF & Draw2D

- “Create an Eclipse-based Application Using the Graphical Editing Framework” (GEF Tutorial) :
http://www-106.ibm.com/developerworks/opensource/library/os-gef/

- “Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework” (IBM Redbook) :
http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246302.html

- “GEF Examples” :
http://eclipse-wiki.info/GEFExamples

- “Displaying a UML diagram using Draw2D” :
http://eclipse.org/articles/Article-GEF-Draw2D/GEF-Draw2D.html

- javadoc et code source de GEF

- javadoc et code source de Draw2D

- newsgoup de GEF :
news://news.eclipse.org/eclipse.tools.gef

Eclipse

- « The Java Developer’s Guide to Eclipse » :
Shavor, D’Anjou, Fairbrother, Kehn, Kellerman, McCarthy, éditions Addison-Wesley

- aide d’Eclipse.

MurMur

- Documents « Deliverables » D1, D2,...faisant partie de la documentation du projet.

- code source de l’ancien éditeur de schéma.