# GEF : Exemple.

Cet exemple s'inspire :

- d'exemples Gef très simples et très utiles trouvés sur http://eclipsewiki.swiki.net/ (le site a souvent des problèmes...voire est carrément mort...)
- d'un exemple de diagramme de classes réalisé avec draw2d : http://eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html

## 1. Objectif du document :

- Décrire la réalisation d'un éditeur de diagrammes de classes UML très simplifié, sous forme de plugin pour Eclipse, utilisant Gef et draw2d.
- Permettre à son auteur de se familiariser avec les techniques nécessaires à la réimplémentation de l'éditeur de schémas du projet murmur, puisque cet éditeur est proche d'un éditeur de diagrammes de classes.

  La réalisation comprend :
  - les figures draw2d représentatives des classes, méthodes, attributs et relations,
  - les modèles des classes, attributs, méthodes et relations,
  - la partie édition réalisée avec Gef (EditParts, EditPolicies, Commands, etc.)
  - l'intégration (rudimentaire) à Eclipse.

Le document présente les différentes étapes suivies plus ou moins dans l'ordre chronologique et pourrait constituer une aide pour quelqu'un qui souhaite faire le même genre de chose, d'où beaucoup de remarques très pragmatiques et pas forcément passionnantes.

## 2. Le plugin :

Réalisé à partir d'un « plug-in project » vierge (c-à-d sans code auto-généré : paradoxalement c'est bien plus simple ainsi pour commencer).

Le plus simple pour démarrer le plugin est de prendre un des exemples Gef simplifié sur http://eclipsewiki.swiki.net/, d'essayer de le faire marcher (ceci nécessitant de faire les imports nécessaires dans le workspace) et de s'en inspirer pour démarrer la fabrication du plug-in.

### 2.1) Fichier plugin.xml :

On peut partir soit d'un exemple comme ceux de http://eclipsewiki.swiki.net/ et copier-coller le ficher plugin.xml, soit de la documentation d'Eclipse sur les extension points du workbench, à la section sur l'extension point org.eclipse.ui.editors. Un exemple type pour le point d'extension org.eclipse.ui.editors est fournis. Dans les deux cas il suffit de modifier un peu les choses pour les personnaliser.

Voir code.

### 2.2) La classe de l'éditeur.

Il faut fournir une classe implémentant org.eclipse.ui.IEditorPart. Le plus simple pour commencer rapidement est de prendre une implémentation d'IEditorPart fournie par Gef, c'est-à-dire soit GraphicalEditor, soit GraphicalEditorWithPalette. J'ai choisi d'étendre GraphicalEditorWithPalette. Voir classe rlemaigr.classdiagrameditor.MyEditor.

## 3. Choix d'architecture :

### 3.1) modèle :

Le modèle doit au moins posséder une représentation des éléments suivants :

- des méthodes (propriétés : l'identificateur, le type, la visibilité, les paramètres),
- des attributs (propriétés : l'identificateur, le type, la visibilité),
- des classes (propriétés : le nom, des méthodes, des attributs, une liste de relation d'héritage dont la classe est la source et idem pour la cible),
- des relations d'héritage (propriétés : la source, la cible, les bendpoints).

Il faudra une classe pour représenter chacun de ces éléments.

Par facilité de manipulation du modèle et d'organisation des editparts, il est pratique d'ajouter une classe qui servira de « top-container » au modèle. Par exemple dans ce cas-ci une classe représentant le diagramme qui aura pour donnée membre la liste des classes du modèle.

Le modèle utilisera le mécanisme de notification propre aux java beans. C'est-à-dire :

- les classes du modèle devant notifier posséderont chacune un objet java.beans.PropertyChangeSupport (objet qui comprend la liste des listeners),
- les classes devant écouter les notifications implémenteront java.beans.PropertyChangeListener.

### 3.2) editparts :

Rappel : il faut définir une arborescence dans le modèle. Cela se fait par la méthode getModelChildren() d'EditPart. L'arborescence des editparts est automatiquement calquée par gef sur cette arborescence, et l'arborescence des représentations visuelles des EditParts est calquée elle aussi sur cette arborescence.

Ne pas oublier que dans draw2d, les relations parent - enfant correspondent à des relations graphiques contenant – contenu, et donc le choix de l'arborescence du modèle défini la manière dont les représentations graphiques des éléments seront contenues les unes dans des autres.

On voudrait pouvoir manipuler graphiquement et individuellement sur le diagramme chaque méthode/attribut pour pouvoir les faire changer de place dans la liste des méthodes/attributs de leur classe ou pour pouvoir les faire changer de classe par drag&drop. Cela nécessite de prévoir des EditParts et donc aussi des figures séparées pour les attributs et les méthodes.

Les attributs et les méthodes doivent être représentés graphiquement à l'intérieur de leur classe et se déplacer avec elle. Cela conduit à en faire des enfants de leur classe via la méthode getModelChildren() de l'EditPart associé à l'objet classe du modèle. Cela implique

que les figures des méthodes et des attributs seront ajoutés automatiquement au content pane de la figure de leur classe. **Cela pose un problème : on souhaite pouvoir séparer les méthodes des attributs et non pas les regrouper ensemble dans un même container visuel !** En effet Gef ne permet pas de définir plusieurs content panes pour un même Editpart.

Une manière de s'en sortir est de définir deux EditPart enfants de l'EditPart représentant une classe. L'un dont le content pane contiendra les attributs et l'autre les méthodes. On peut faire cela facilement en ajoutant dans le modèle deux éléments un peu artificiels : chaque classe aura pour enfant une liste de méthode représentée par une classe particulière et illustrée par un EditPart particulier (dont la figure servira de container au méthodes) et une liste d'attributs, de la même manière. Ces listes auront à leur tour pour enfant respectivement les méthodes et les attributs.

Le modèle devient donc :

- des méthodes (propriétés : l'identificateur, le type, la visibilité, les paramètres),
- des attributs (propriétés : l'identificateur, le type, la visibilité),
- des listes de méthodes (enfants : une liste de méthodes...),
- des listes d'attributs (enfants : une liste d'attributs...),
- des classes (propriétés : le nom, une liste de relation d'héritage dont la classe est la source et idem pour la cible ; enfants : un objet encapsulant une liste de méthodes et un objet encapsulant une liste d'attributs),
- un objet représentant le diagramme (enfants : la liste des classes).

Chacun de ces éléments ayant son propre EditPart et sa propre figure, la notion d'enfant étant définie par la méthode getModelChildren() de l'EditPart associé.

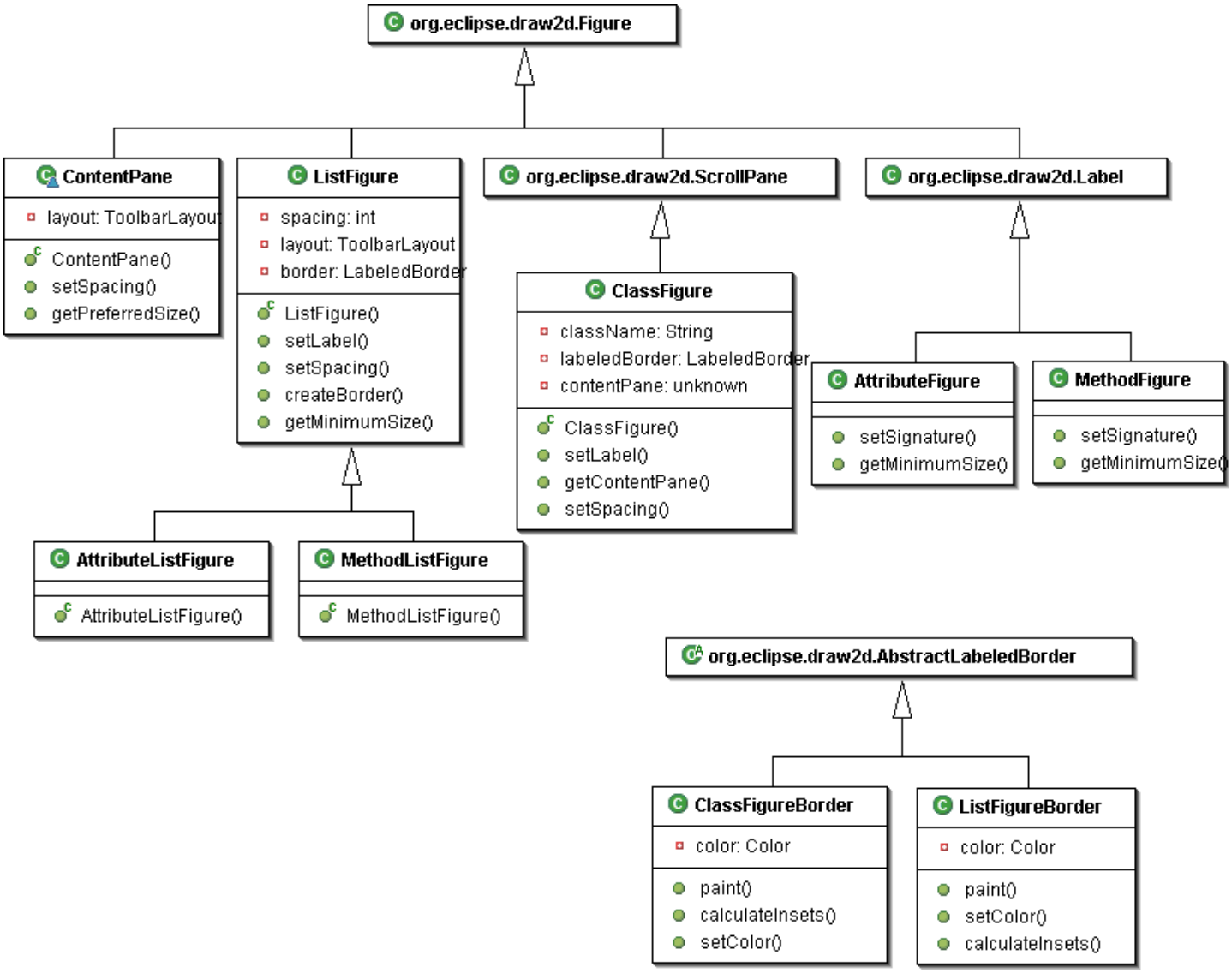
### 3.3) figures :

- Les méthodes et attributs seront représentés par des org.eclipse.draw2d.Label.
- Les listes de méthodes et d'attributs seront des Figures ordinaires mais dont les enfants seront organisés par un ToolbarLayout et avec un bord particulier mentionnant s'il s'agit de méthodes ou d'attribut.
- Les classes seront représentées par un scroll pane dont les enfants seront organisés par un ToolbarLayout.
- Les relations d'héritage seront représentées par des PolylineConnections décorées d'un triangle à leur extrémité.

## 4. Les figures.

Dans Gef, le code des figures est totalement indépendant du modèle. Une figure n'a aucune référence vers le modèle qu'elle représente. Les figures doivent donc sauver dans des données membres les différentes propriétés du modèle qui sont à représenter et fournir des méthodes pour permettre de les ajuster. Ces méthodes, lorsqu'elles sont appelées, devraient provoquer elles-mêmes une revalidation et/ou une repeinture de la figure si nécessaire, de sorte que ces détails propres à draw2d soient masqués pour les clients (les editparts en l'occurrence).

Les figures se trouvent dans le package rlemaigr.classdiagrameditor.figures :




**4.1) images :**

C'est joli d'ajouter des petites images dans les figures draw2d...

On peut en trouver de très nombreuses sous format gif dans les plugin d'eclipse et notamment dans org.eclipse.jdt.ui, répertoire icons. Toutes les images utilisées dans le java developement toolkit se trouvent là.

Voilà comment s'en servir :

- on place les images intéressantes dans le répertoire source du package qui doit y accéder via l'explorateur windows.
- On fait un refresh du projet de plugin dans Eclipse. Normalement les images doivent apparaître dans le package et seront placées automatiquement lors de la compilation dans la version compilée de package (à vérifier).
- On importe org.eclipse.swt.graphics.Image dans la classe (MaClasse) où l'image sera utilisée.
- On crée l'image comme ça :

```
Image monimage = new Image(null,
MaClasse.class.getResourceAsStream("image.gif"));
```

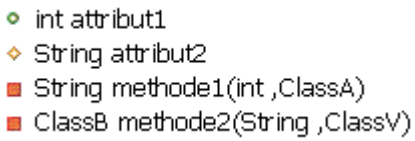
- On peut ensuite la peindre à l'aide d'un objet Graphics dans la méthode paint d'une figure ou d'un bord.

### 4.2) figures des attributs et méthodes

Classes AttributeFigure et MethodFigure.

Ce sont des labels draw2d. L'icône dépend de la visibilité de la méthode/attribut. L'icône est une image telle que décrite à la section précédente.

Ca donne ça :




Il existe une méthode pour ajuster la signature de la méthode ou la déclaration de l'attribut. Cette méthode ajuste les propriétés du Label (pas besoin de revalider/repeindre donc : draw2d le fait tout seul lorsqu'on modifie les propriété du Label).

La méthode getMinimumSize(int hint, int vint) est redéfinie pour renvoyer new Dimension(0,0). En effet, ce label est destiné à être contenu dans un container géré par ToolBarLayout, et ce layout manager n'alloue jamais au éléments une taille inférieure à leur minimumSize. Ceci peut conduire à ce que les bornes des éléments dépassent du container dans lequel ils sont contenus (mais pas leur représentation graphique because clipping). Ca pose un problème dans gef parce que ces éléments, lorsqu'ils seront sélectionnés, se verront décorés d'un rectangle de sélection qui a la taille de leur bornes mais se trouve dans un autre layer. Ce rectangle noir dépasse donc du container parent puisqu'il n'est pas clippé et c'est inesthétique.

Ces figures ne sont pas destinées à avoir d'enfants.

### 4.3) figures des listes de méthodes et d'attributs :

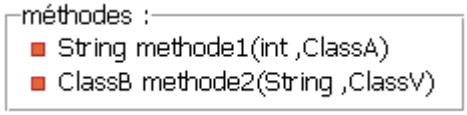
C'est deux figures ne diffèrent que par le label explicatif, c'est pourquoi les deux classes MethodListFigure et AttributeListFigure dérivent de la classe ListFigure.

ListFigure étend la classe Figure.

Son layout manager est ajusté à ToolbarLayout. Voir la javadoc pour les propriétés exactes de ce layout. En gros, il aligne les enfants en colonne en leur donnant leur preferredSize s'il y a suffisamment de place en largeur, et sinon réduit leur largeur mais jamais en dessous de leur minimumSize. Il est possible de lui demander d'allouer aux enfants des largeurs supérieures à leur preferredSize de manière à leur permettre d'occuper toute la largeur excédentaire si celle-ci existe. C'est ce qui est fait ici, de nouveau pour des raisons esthétiques lors de la sélection des enfants dans gef.

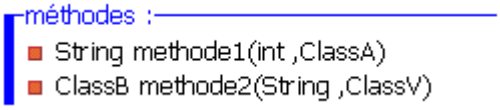
La méthode getMinimumSize est redéfinie comme ci-dessus.

ListFigure est munie d'un bord étendant la classe LabeledBorder de manière à pouvoir y écrire « méthodes : » ou « attributs : » selon le cas. Il en existe de tout fait dans draw2d. Avec GroupBoxBorder, ça donne ça :




Mais ça n'est pas très joli.

A la place, ça n'est pas très difficile d'étendre AbstractLabeledBorder pour faire ça :




Et c'est déjà plus joli (à mon goût en tout cas).

Le bord en question est défini dans la classe rlemaigr.classdiagrameditor.figures.ListFigureBorder. Les méthodes paint() et calculateInstets(IFigure f) sont redéfinies pour respectivement introduire les graphismes et définir la place prise par le bord pour que la figure hôte puisse ajuster sa client area correctement et ne pas dépasser sur le bord en se peignant et en peignant ses enfants.

Il n'y a rien de transcendant sauf l'utilisation d'une méthode statique de la classe FigureUtilities de draw2d qui permet de calculer la place que prend un texte dans une police de caractère donnée : `FigureUtilities.getTextExtents(...)`.

Ces figures sont destinées à avoir des enfants et sont leur propre content pane.

### 4.4) figures des classes :

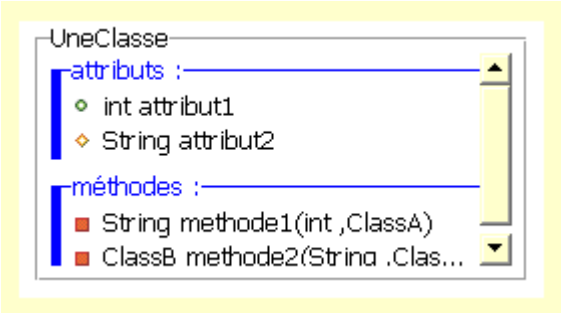
Classe ClassFigure.

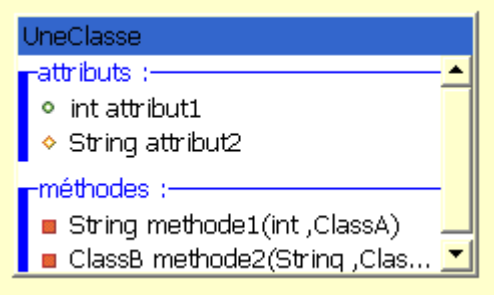
Cette classe étend ScrollPane. Les listes de méthodes et de figures sont ajoutées à une figure particulière, instance de la classe ContentPane, définie dans le même fichier que ClassFigure et accessible par la méthode getContentPane.

Un ContentPane est régi par un ToolbarLayout. La méthode getPreferredSize est redéfinie de manière à ce que la largeur préférée n'excède jamais la largeur du viewpert du scroll pane. Voir les explications à ce sujet dans le code. C'est encore lié au même problème d'esthétique que précédemment, posé par la sélection d'éléments compris dans un container.

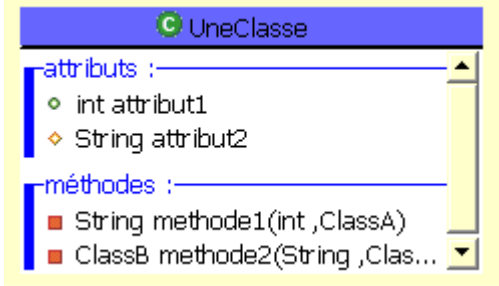
Le bord est un LabeledBorder dont le label peut être ajusté pour montrer le nom de la classe.

On peut le faire avec un GroupBoxBorder ou un FrameBorder, mais soit ça n'est pas très joli, soit ça ressemble à une fenêtre dans la fenêtre :



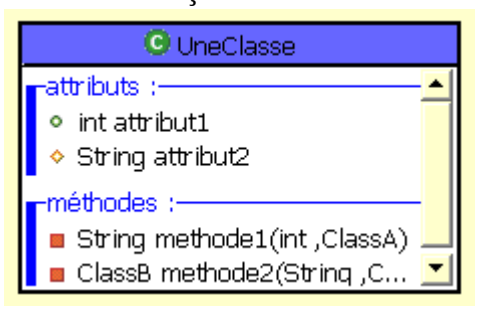




Avec un LabeledBorder personnalisé ça donne ça :

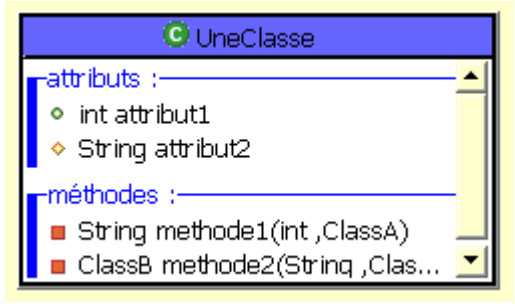



Ce bord est défini dans la classe rlemaigr.classdiagrameditor.

Ca manque encore d'une bordure. Ca peut se faire en composant ce bord avec des bords existant dans draw2d. Si on compose avec un LineBorder puis avec un SimpleRaisedBorder ça donne :




puis




Et c'est satisfaisant.

Il existe une méthode pour ajuster le nom de la classe. Elle revalide et repaint la figure puisque ça n'est pas fait automatiquement par draw2d dans ce cas.

Cette figure est destinée à avoir des enfants et sont content pane est une instance de ContentPane.

## 5. le modèle :

### 5.1) Description :

Les classes du modèle se trouvent dans rlemaigr.classdiagrameditor.model.
Il comprend les classes :

- DiagramModel
- ClassModel
- InheritanceModel

- MethodListModel
- AttributeListModel
- MethodModel
- AttributeModel
- DiagramElementModel
- ListModel
- ModelObject

Les 7 premières sont selfs descriptives.

DiagramElementModel représente un élément du diagramme qui possède une donnée membre contrainte déterminant la position dans le diagramme. ClassModel dérive donc de DiagramElementModel.

ListModel est la classe qui regroupe les comportements partagés par MethodListModel et AttributeListModel (tous en fait, pour l'instant).

ModelObject introduit un support pour la notification des événements. C'est la classe parente à toute la hiérarchie.

Il n'y a pas grand-chose à en dire de plus que ce qui a déjà été dit dans la section sur les choix d'architecture, sauf au point de vue du mécanisme de notification des changements du modèle.

### 5.2) Notification :

Chaque classe du modèle possède :

- une donnée membre de type java.beans.PropertyChangeSupport qui encapsule la liste des listeners,
- deux méthodes addPropertyChangeListener( java.beans.PropertyChangeListener) et removePropertyChangeListerner( PropertyChangeListener) pour ajouter et supprimer des listeners (en l'occurrence l'EditPart associé à l'objet),
- une liste de données membres public static final String qui définissent les différentes propriétés sujettes à notification.

Ces éléments sont inclus dans la superclasse ModelObject.

Quels changements du modèle doit-on notifier aux listeners dans le cas de gef ?

Ceux qui conduisent à une désynchronisation d'une visualisation graphique par rapport au modèle. On suppose alors que les listeners (EditParts) savent que faire pour resynchroniser les vues du modèle avec celui-ci, selon la définition des différentes vues du modèle.

Il faut donc notifier :

- les changements de propriétés des objets du modèle qui sont représentées graphiquement d'une manière ou d'une autre dans la Figure associée à cet objet du modèle,
- les changements structuraux du modèle conduisant à un changement dans l'arborescence des EditParts (et donc de la visualisation du modèle), c'est-à-dire conduisant à un changement du retour de la méthode getModelChildren dans un EditPart ou l'autre,

- les changements structuraux conduisant à une modification des connections entre les NodeEditParts.

Le premier cas est clair.
Le second conduit à notifier :

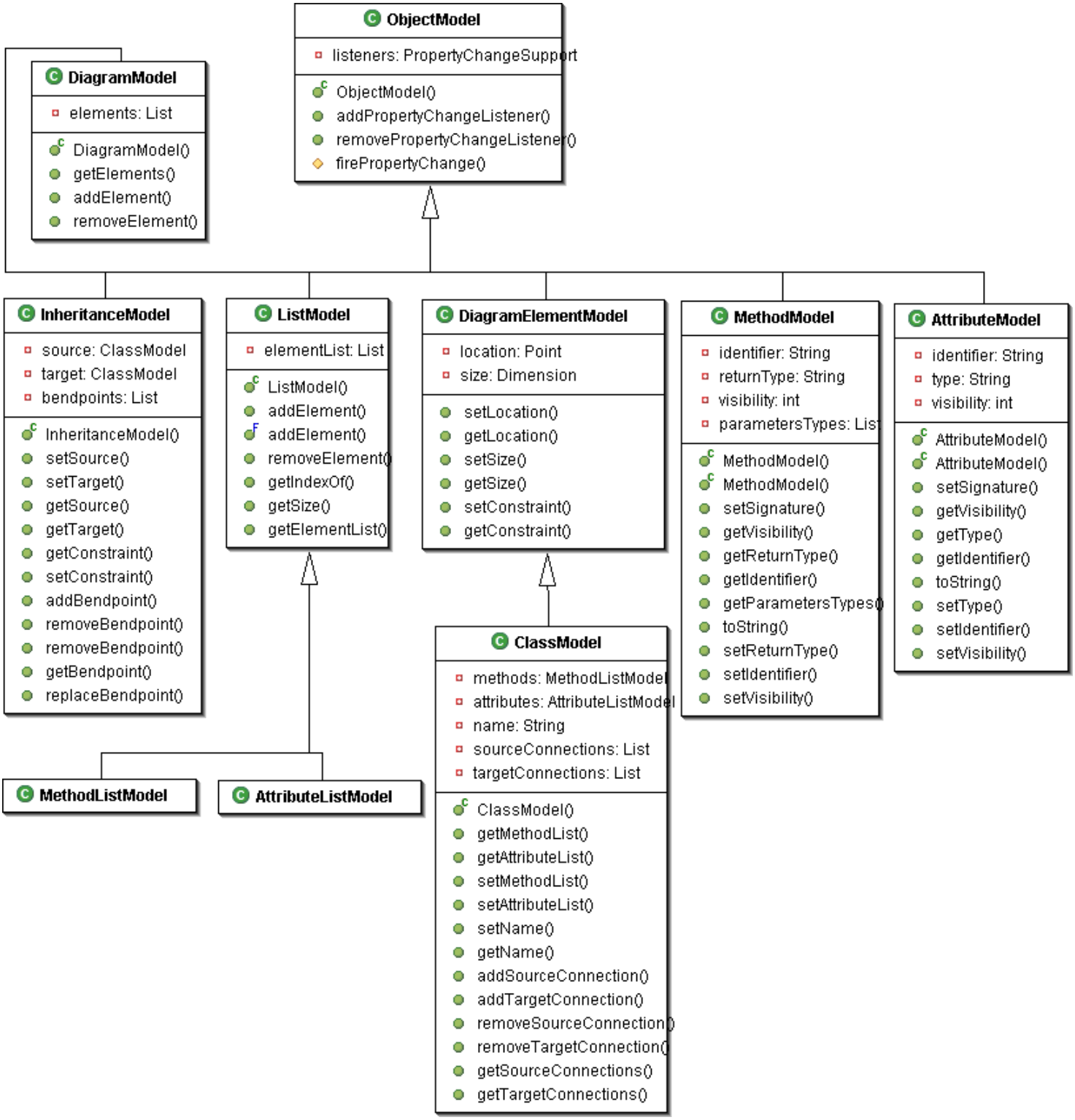
- chaque ajout/retrait de méthode à une liste de méthodes,
- chaque ajout/retrait d'attribut à une liste d'attributs,
- chaque ajout/retrait d'une liste de méthodes ou d'une liste d'attributs à une classe,
- chaque ajout/retrait d'une classe au diagramme.

Le troisième conduit à notifier :

- chaque ajout/retrait de relation d'héritage à une classe.

### 5.3) Diagramme de classes :

Certains éléments peu intéressants n'apparaissent pas pour gagner de la place...




## 6. les EditParts :

### 6.1) Description :

Ils sont définis dans les classes :

- DiagramEditPart,
- ClassEditPart,
- InheritanceEditPart,
- ListMethodEditPart,
- ListAttributeEditPart,
- AttributeEditPart,
- MethodEditPart,
- ListEditPart,
- ObjectEditPart

dans le package rlemaigr.classdiagrameditor.editparts.

Voir plus haut pour les grandes lignes. Il existe un EditPart et une figure par classe du modèle à représenter graphiquement. Les EditParts suivent la même hiérarchie que le modèle pour profiter des propriétés communes des différentes objets du modèle, notamment au sujet de l'écoute des événements et du rafraichissement de la vue.

Les EditParts étendent tous (ici) la classe AbstractGraphicalEditPart qui est particularisée pour les figures draw2d. Il y a deux méthodes abstraites à implémenter :

- **IFigure createFigure()** : doit renvoyer la figure associée au modèle de cet EditPart.
- **Void createEditPolicies()** : doit installer les différentes EditPolicies de cet EditPart.

D'autres méthodes sont à redéfinir :

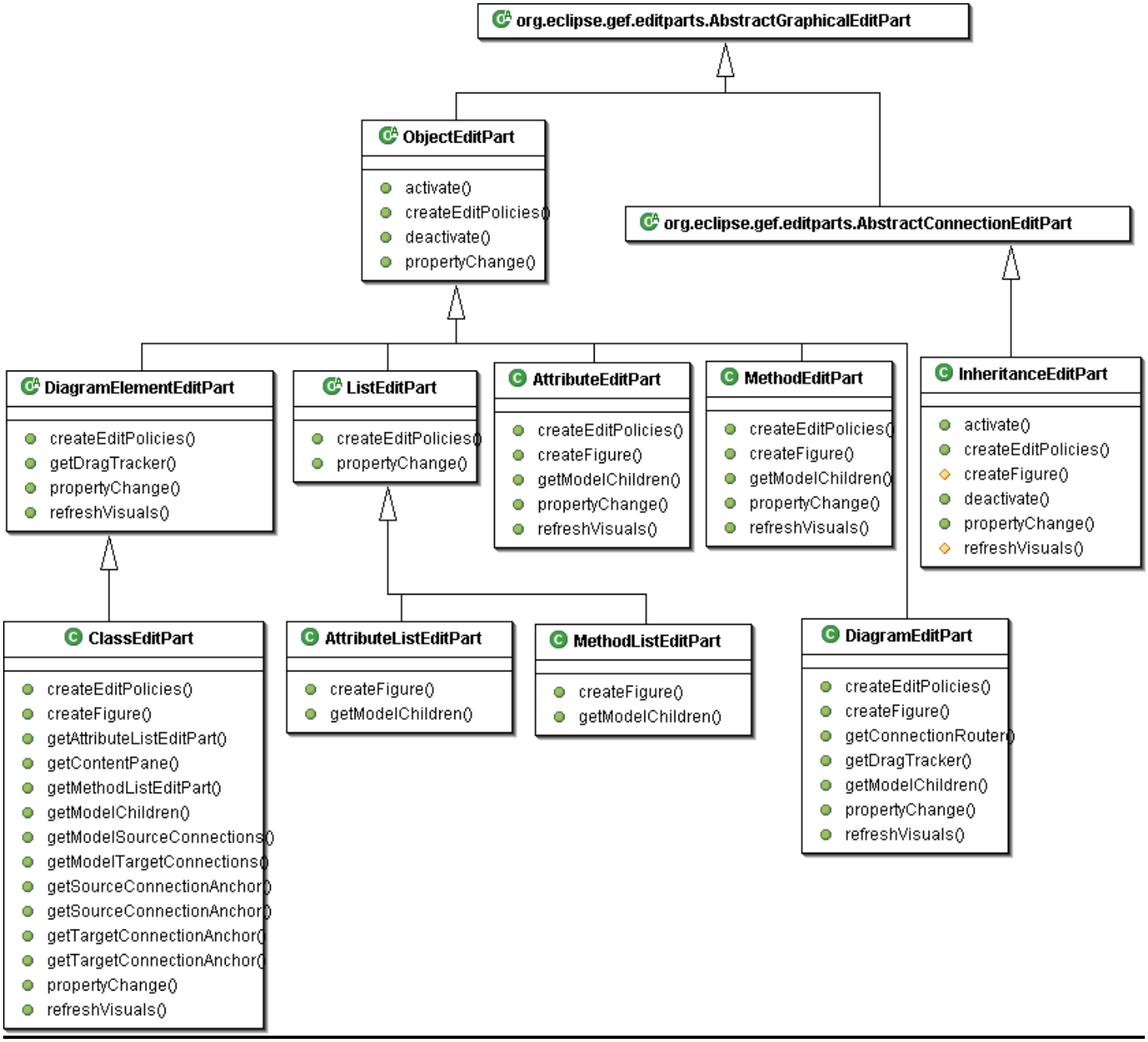
- **List getModelChildren()** : appelée par gef pour obtenir la liste des objets du modèle qui sont considérés comme les enfants du modèle de l'EditPart courant, pour la représentation graphique tout au moins. Cette méthode ne peut pas retourner null (retourner new ArrayList() à la place) !!
- **IFigure getContentPane()** : si le content pane de l'EditPart est différent de sa figure, comme c'est le cas pour les ClassEditPart, cette méthode doit être redéfine.
- **Void refreshVisuals()** : cette méthode est appelée par Gef (elle est protected et non public !) lorsqu'il est nécessaire de synchroniser les graphismes de la figure de l'EditPart avec le modèle de l'EditPart. Elle est appelée une première fois avant l'affichage de la figure de l'EditPart. C'est ici qu'il faut ajuster les propriétés de la figure en fonction du modèle et éventuellement introduire la contrainte correspondante dans le LayoutManager du content pane de l'EditPart parent.

Avec les 5 méthodes précédentes, c'est suffisant pour afficher un modèle sans connexions. C'est ce qu'il faut arriver à faire dans un premier temps. Pour les connexions voir plus loin.

Après on continue en introduisant des possibilités d'édition via les méthodes suivantes (et les EditPolicies) :

- **Void activate()** : appelée lorsqu'un EditPart fait partie d'une arborescence d'EditPart visualisée dans un viewer. C'est ici qu'il faut enregistrer l'EditPart en tant que listener de son modèle.
- **Void desactivate()** : il faut désenregistrer l'EditPart en tant que listener de son modèle.
- **Void propertyChanged(PropertyChangeEvent)** : pour réagir au notification du modèle. Cette méthode pourraient simplement appeler refreshVisuals() ou se montrer plus sélective dans la mise à jour en fonction de l'événement.
- **DragTracker getDragTracker()** : cette méthode est appelée par le SelectionTool lors d'un mouse pressed event sur un EditPart pour obtenir le DragTracker qui va gérer la suite des événements jusqu'au mouse released event. Les éléments qui doivent pouvoir être déplacés/reparentés par dragging devraient redéfinir cette méthode et retourner un DragEditPartTracker. La figure d'arrière plan devrait aussi la redéfinir pour permettre la sélection d'un groupe d'éléments via un MarqueeDragTracker.

### 6.2) diagramme de classes (final) :




## 7. étapes suivies pour rajouter les comportements d'édition :

**TODO : faire passer la section 7.1 dans le document sur gef à la section tools (et oui tu vas encore devoir changer toute la numérotation...).**

Pour comprendre la suite il est préférable de savoir comment fonctionne un DragEditPartTracker :

### 7.1) DragEditPartTracker :

Définition :

Un DragEditPartTracker est un outil permettant de déplacer des EditParts au sein de leur parent, ou de les déplacer vers un autre parent :

- Lors du relâchement du bouton de la souris, il distribue aux EditParts concernés les requêtes req_move, ou req_orphan et req_add (selon qu'il s'agit d'un déplacement au sein du parent ou d'un changement de parent) en récupère les commandes et les exécute via la CommandStack.
- Lors du dragging, il déclenche l'affichage et l'effacement des **feedbacks de source et de cible**. Source = les EditParts déplacés, feedback de source = représentation simplifiée des editparts si ils étaient déplacés à la position courante. Cible = l'EditPart de destination du déplacement, feedback de cible = représentation simplifiée de la position qu'occuperait les EditPart dans le (éventuellement nouveau) parent si ils étaient droppés à la position courante.

[ Pour comprendre le fonctionnement des feedbacks, penser au déplacement d'un fichier dans l'explorateur windows en fonction des différents layout choisis pour le dossier de destination du fichier. On retrouve les deux types de feedbacks. ]

Fonctionnement :

Operation set et target EditPart :

- Un DragEditPartTracker opère sur un certain ensemble d'EditParts appelé **operation set**. Celui-ci est formé de tous les EditParts actuellement sélectionnés dans le viewer qui comprennent (*comprennent = méth. understandRequest(Request) d'EditPart*) les requêtes de type req_move.
- Lors du dragging, à chaque déplacement du pointeur, le **target EditPart** est déterminé. Il s'agit de l'EditPart le plus à l'avant plan capable de faire office de cible (*capable de faire office de cible = méth. getTargetEditPart(Request) d'EditPart*) pour le déplacement du groupe d'EditParts.

A chaque mouse dragged event : si le target EditPart est le container parent des EditParts de l'operation set, il s'agit d'un déplacement (req_move) sinon il s'agit d'un reparenting (req_orphan – req_add). Le target EditPart est alors sollicité pour fabriquer la commande correspondant à la requête de déplacement/reparenting. Si il retourne null (c'est-à-dire qu'il peut faire office de cible mais que cette requête particulière pour ce groupe d'EditParts particulier ne lui est pas applicable) le curseur prend la forme d'un panneau d'interdiction de stationner  .

Lors du relâchement du bouton de la souris, la commande éventuellement fabriquée lors du dernier événement mouse dragged est exécutée via la command stack.

Obtention d'un DragEditPartTracker par le SelectionTool :

Les DragEditPartTrackers sont obtenu par le SelectionTool de deux manières :

- Suite à un mouse pressed sur un EditPart, la méthode getDragTracker est appelée sur l'EditPart et le DragTracker retourné prend le contrôle,
- Suite à un mouse pressed sur un handle introduit sur une figure préalablement sélectionnée, si elle est munie d'un ResizableEditPolicy ou d'un NonResizableEditPolicy. Les handles sont faits pour retourner des dragtrackers.

## 7.2) sélection et déplacement des figures des classes dans le diagramme :

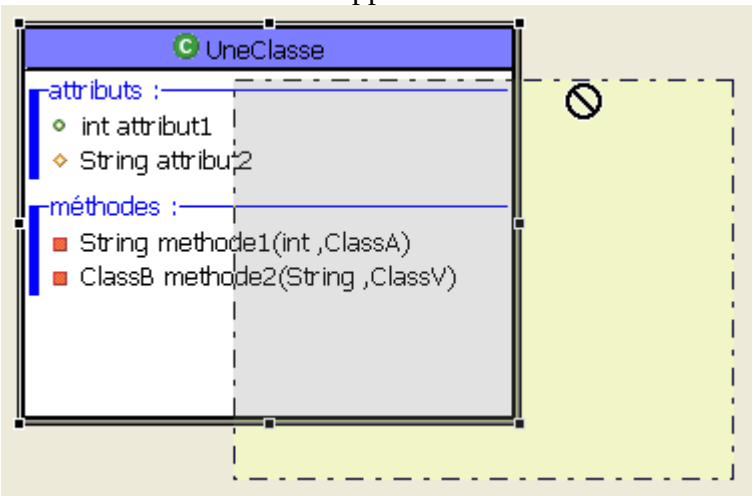
Pour permettre de déplacer les figures des classes sur le diagramme il faut permettre au SelectionTool d'obtenir les DragEditPartTracker adéquats. Il faut donc :

- Redéfinir getDragTracker dans ClassEditPart pour renvoyer un DragEditPartTracker. C'est déjà suffisant pour déplacer les classes mais pas pour les redimensionner.
- Munir ClassEditPart d'un ResizableEditPolicy qui affichera des handles lorsque la figure sera sélectionnée, et qui renverront des dragtrackers adéquats.

L'ajout d'un ResizableEditPolicy à ClassEditPart se fait automatiquement en ajoutant un XYLayoutEditPolicy à DiagramEditPart.

ResizableEditPolicy est en effet l'EditPolicy satellite par défaut de XYLayoutEditPolicy et sera installée/désinstallée automatiquement pour le primary_drag_role de ClassEditPart dès qu'un ClassEditPart enfant sera ajouté/supprimé de DiagramEditPart. Voir document sur Gef, section sur LayoutEditPolicy pour les détails à ce sujet.

A ce stade les classes peuvent être sélectionnées, et les feedbacks de déplacement/redimensionnement apparaissent :




Cependant, une fois la souris relâchée, rien ne se passe. La forme du pointeur indique que DiagramEditPart retourne null lorsqu'on lui demande de fournir une commande pour une req_move.

C'est parce que les requêtes sont émises par les dragtrackers mais aucune commande n'est encore fournie en retour. Pour cela il faut redéfinir la méthode Command createChangeConstraintCommand(EditPart child, Object constraint) dans XYLayoutEditPolicy pour qu'elle retourne une commande qui aura pour effet de changer la contrainte de positionnement associée à la figure de la classe dans le modèle (et pas dans le layoutmanager !! les commandes devraient en effet modifier uniquement le modèle qui notifie alors les editparts qui mettent alors à jour la vue). L'XYLayoutEditPolicy modifiée est rlemaigr.classdiagrameditor.XYLayoutEditPolicyImpl et la commande est rlemaigr.classdiagrameditor.DiagramElementChangeConstraintCommand.

A ce stade, ça ne fonctionne toujours pas : le modèle est effectivement modifié sous l'effet des commandes de changement de bornes des classes mais ClassEditPart n'écoute pas encore les notifications de modification des bornes. Il faut

- que ClassEditPart implémente PropertyChangeListener et mette à jour la figure de la classe (par appel à refreshVisuals() par exemple) lors des appels à propertyChanged(PropertyChangedEvent),
- redéfinir activate() et desactivate() dans ClassEditPart pour ajouter this comme listener du modèle.

Et maintenant ça marche (gloria et alléluia).

**7.3) sélection et déplacement des attributs et des méthodes dans leur propre container :**

Le problème est le même qu'au point 7.1 à ceci près que les layout des container des méthodes et attributs ne sont pas gérés par des XYLayout managers mais par des ToolbarLayout managers. Or il n'existe pas de ToolbarLayoutEditPolicy ! Il faut donc en programmer une soi-même en étendant OrderedLayoutEditPolicy.

On commence par particulariser OrderedLayoutEditPolicy pour le ToolbarLayout. Ca se fait en redéfinissant la méthode :

| | |
|---|---|
| `protected abstract EditPart` | **`getInsertionReference`**`(Request request)`: Calculates a *reference* `EditPart` using the specified `Request`. The EditPart returned is used to mark the index coming *after* that EditPart. `null` is used to indicate the index that comes after *no* EditPart, that is, it indicates the very first index. |

La requête reçue est une ChangeBoundsRequest et possède la position souris à laquelle l'EditPart doit être placé. Il faut trouver la position d'ordre parmi les enfants du content pane de l'EditPart hôte qui correspond à cette position souris en se basant sur leurs bornes, tenant compte de la manière dont le ToolbarLayout les a disposés. Voir code pour ça. La classe est rlemaigr.editpolicies.ToolbarLayoutEditPolicy. Cette classe est l'équivalent de XYLayoutEditPolicy mais pour un ordered layout.

Cette editpolicy doit encore être subclassée pour être particularisée à cette application et renvoyer les commandes adéquates. C'est fait avec la classe ToolbarLayoutEditPolicyImpl et la commande ListElementChangeConstraintCommand. Il ne faut pas oublier de faire un refresh du ListEditPart sous l'effet des notifications du modèle de manière à mettre à jour les EditParts et les Figures lorsque des enfants sont ajoutés/supprimés de ListModel. Voir code.

**7.4) target feedback lors du déplacement des méthodes et attributs :**

Par défaut aucun feedback n'est montré. Je voudrais faire apparaître une ligne noire entre les deux éléments de la liste entre lesquels l'élément se trouverait si il était déposé à la position souris courante.

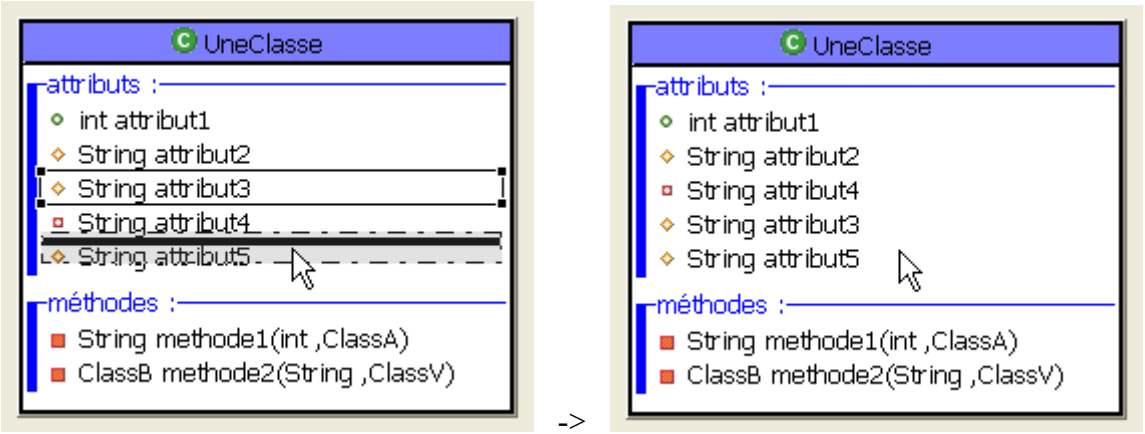
Pour ça il faut redéfinir les méthodes :

| | |
|---|---|
| `protected void` | **`showLayoutTargetFeedback`**`(Request request)`: Shows target layout feedback. During *moves*, *reparents*, and *creation*, this method is called to allow the LayoutEditPolicy to temporarily show features of its layout that will help the User understand what will happen |

| | |
|---|---|
| | if the operation is performed in the current location. |
| protected void | **eraseLayoutTargetFeedback**(Request request): Erases target layout feedback. |

La méthode showLayoutTargetFeedback peut être appelée répétitivement sans appel intermédiaire à eraseLayoutTargetFeedback. L'approche est calquée sur celle suivie par NonResizableEditPolicy pour montrer le rectangle fantôme de feedback de source pour les req_move. Voir code.

Ca donne :




Le rectangle fantôme est le source feedback affiché par défaut par NonResizableEditPolicy lors du déplacement de l'hôte, la ligne noire est le feedback de cible personnalisé montré par ToolbarLayoutEditPolicy.

**TODO : faire passer la section 7.5 dans le document sur gef.**

### 7.5) palette :

Description :

La palette permet à l'utilisateur de spécifier quel est l'outil courant dans les différents EditPartViewer connectés à l'EditDomain.

Fonctionnement :

Dans l'état actuel des choses, il ne peut exister qu'au maximum une palette par EditDomain et elle est partagée par tous les EditPartViewer. Lorsque l'utilisateur choisi un nouvel outil dans la palette, la palette notifie ses listeners de ce changement. L'EditDomain est enregistré en tant que listener de la palette. Le nouvel outil choisi devient alors l'outil courant dans l'EditDomain, c'est-à-dire celui vers lequel seront routés les événements en provenance de chaque EditPartViewer connecté à l'EditDomain.

Implémentation :

La palette est implémentée dans gef en réutilisant gef et draw2d. La palette possède donc un modèle (qui peut changer dynamiquement) représentant les outils et groupes d'outils. Ce modèle est présenté graphiquement dans un PaletteViewer, qui est une classe dérivée de EditPartViewer.

Utilisation :

Tout le code de construction de la palette devrait prendre place dans la méthode createPartControl(Composite) d'IEditorPart.

Il faut réaliser les étapes suivantes, dans l'ordre (?) :

- La construction de la palette commence par la construction du PaletteViewer par new PaletteViewer().
- La spécification de la palette utilisée par l'EditDomain et l'enregistrement de l'EditDomain en tant que listener de la palette se fait par la méthode :

| | |
|---|---|
| `void` | **`setPaletteViewer`**`(PaletteViewer palette)`: Sets the `PaletteViewer` for this EditDomain |

- La réalisation (le fait qu'elle soit incorporée à l'ui) de la palette se fait par la méthode createControl(Composite). Le Composite SWT supportant le PaletteViewer est au choix du programmeur, comme pour tout EditPartViewer.

- Le peuplement du viewer de la palette se fait par la méthode **d'EditDomain (et non par la méthode du même nom de PaletteViewer)**

| | |
|---|---|
| `void` | **`setPaletteRoot`**`(PaletteRoot root)`: Sets the PalatteRoot for this EditDomain. |

où root est la racine du modèle de la palette.

Reste à définir le modèle de la palette.

Modèle :

Les classes constituant ce modèle sont définies dans le package org.eclipse.gef.palette.

- PaletteEntry : la racine de la hiérarchie des classes du modèle (= racine statique). Défini les propriétés communes à toutes les éléments de la palette, comme un label, une icône, etc...
- PaletteContainer : un PaletteEntry muni d'une liste d'enfants et de méthodes d'accès à cette liste.
- PaletteRoot : un PaletteContainer destiné à servir de racine au modèle (=racine dynamique)
- PaletteGroup : un PaletteContrainer destiné à contenir une liste de PaletteEntry dont la visualisation graphique n'est pas collapsable.
- PaletteDrawer : un PaletteContrainer collapsable. Ne peut pas contenir d'enfant de type container.
- ToolEntry : un PaletteEntry qui constitue une factory retournant un Tool.
- SelectionToolEntry, MarqueeToolEntry, CreationToolEntry : self-descriptifs.

Il suffit de constituer une arborescence avec ce genre de chose pour réaliser le modèle de la palette.

Des ImageDescriptors pour certaines icône d'outil standard se trouvent en données membre statiques de la classe org.eclipse.gef.SharedImages.

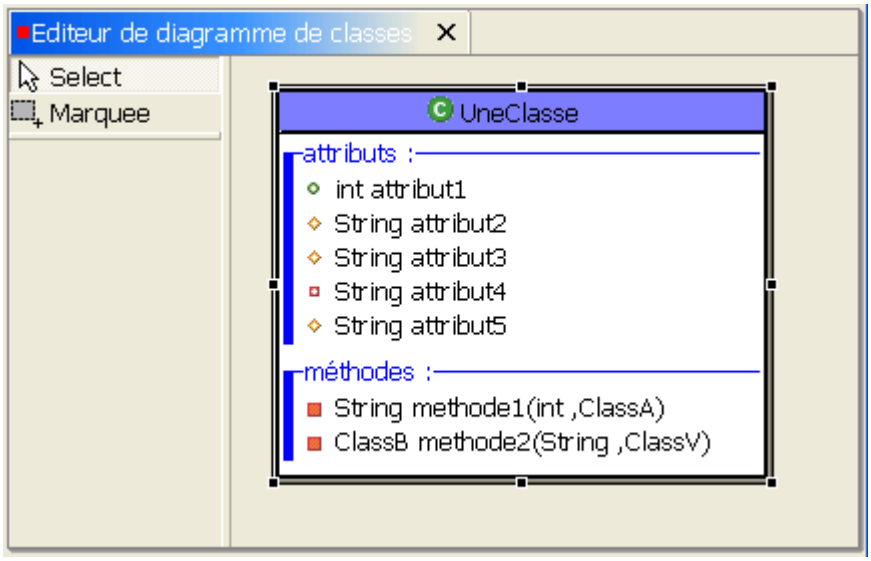
### 7.6) introduction d'une palette :

La prise en charge d'une palette est déjà incluse dans la classe de l'éditeur puisqu'elle dérive de GraphicalEditorWithPalette. Il suffit de redéfinir la méthode

| | |
|---|---|
| protected abstract PaletteRoot | **getPaletteRoot**(): Returns the PaletteRoot for the palette viewer. |

de GraphicalEditorWithPalette pour fournir la racine du modèle de la palette, dans laquelle on introduit les différentes entrées standards. Voir code.

Le résultat est :




**TODO : faire passer la section 7.8 dans le document sur gef.**

## 7.8) CreationTool :

CreationTool est un TargetingTool, tout comme DragEditPartTracker. Son foncionnement est très similaire. Il a pour donnée membre une factory particulière qui est incluse dans chaque requête de création émise. Il ne prend pas en charge lui-même la création des nouveaux objets, donc. Ceci est délégué aux EditPart container de destination des requêtes qui renverront la commande correspondant à la création du nouvel objet, ou null au cas ou la création du type d'objet spécifié par la factory n'est pas supportée.

A chaque déplacement du curseur souris sur le viewer, le CreationTool détermine quel est l'EditPart sous le curseur propre à recevoir une requête de type req_create (c'est le target EditPart). Une requête req_create est émise à destination du target EditPart. Si celui-ci retourne null, le curseur prend la forme d'un panneau d'interdiction de stationnement, et la forme d'un + dans le cas contraire. Le CreationTool provoque aussi l'apparition et l'extinction du target feedback sur le target EditPart.

La commande de création du nouvel objet peut être rendue effective de deux manières :

- Par clic : dans ce cas la requête de création n'inclut aucune indication de taille ni de position du nouvel objet à créer,
- Par dragging : dans ce cas la requête de création inclut pour position le point de début du dragging et pour taille le rectangle dont le coin supérieur gauche est le début du dragging et le coin inf. droit est la fin du dragging.

Les requêtes de type req_create sont de type CreationRequest, qui implémente l'interface DropRequest, tout comme les requêtes de déplacement et d'ajout d'enfant. L'interface DropRequest donne accès à une indication de position de curseur souris, exploitable par le target EditPart pour montrer le feedback indifféremment pour les requête de création, ajout d'enfant et déplacement d'enfant.

### 7.9) création de nouvelles classes dans le diagramme :

On introduit une CreationToolEntry dans la palette. Il faut fournir au constructeur de CreationToolEntry une factory pour la construction des nouveaux objets, dans ce cas « new SimpleFactory(ClassModel.class) ». SimpleFactory se trouve dans org.eclipse.gef.requests. Elle sera passée au CreationTool. Voir code.

Il faut implémenter la méthode

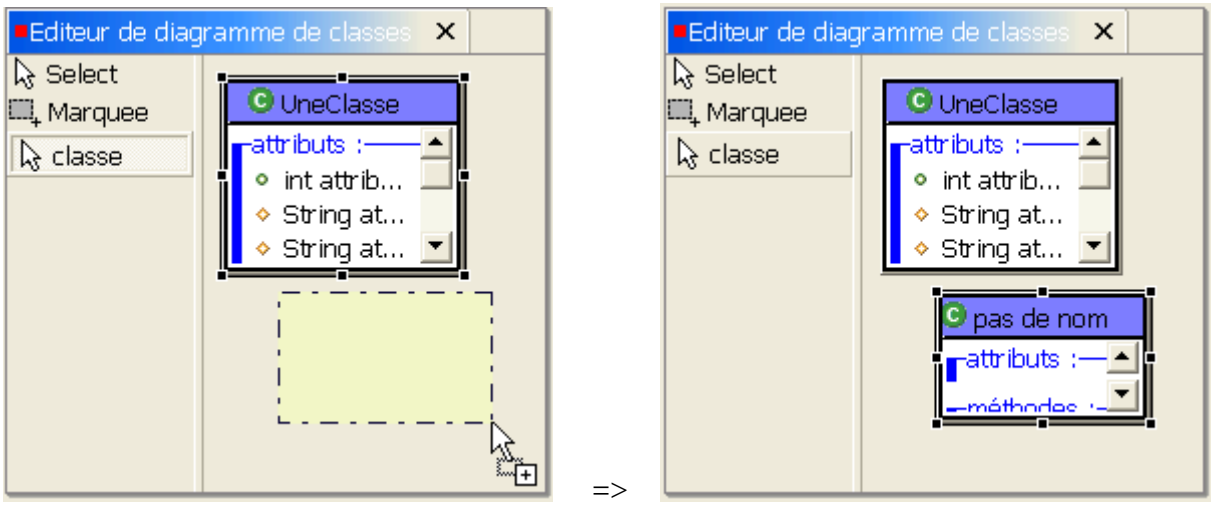
| | |
|---|---|
| protected abstract Command | **getCreateCommand**(CreateRequest request) : Returns the Command to perform a create. |

de manière à ce qu'elle renvoie une commande de création d'un nouvel enfant de type ClassModel dans toutes les LayoutEditPolicies dont le modèle de l'hôte est propre à recevoir un nouvel enfant de ce type. Dans ce cas il n'y a que XYLayoutEditPolicyImpl. Voir code.

Il faut écrire la commande d'ajout d'enfant de type ClassModel au diagramme. C'est fait dans la classe rlemaigr.classdiagrameditor.commands.CreateClassCommand. Voir code.

Il faut provoquer le rafraichissement des enfants de l'EditPart du diagramme lorsqu'un événement de type property_element_added ou property_element_removed survient. C'est fait dans la méthode propertyChange(PropertyChangeEvent) de la classe DiagramEditPart. Voir code.

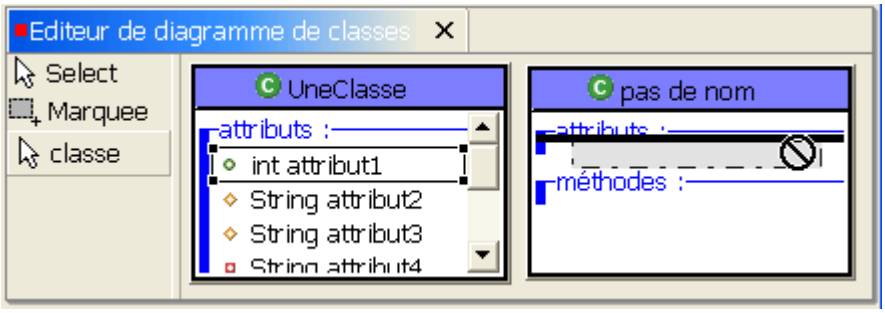
Ca donne :




### 7.10) reparenting d'attributs et de méthodes :

Maintenant qu'il est possible de créer de nouvelles classes dans le diagramme, il est possible aussi d'envisager des passages d'attributs ou de méthode d'une classe à une autre.

Pour l'instant ça n'est pas possible parce que la méthode getCommand() de ListEditPart renvoie null pour les requêtes de type req_orphan_children et req_add, comme l'atteste la forme du curseur :




Pour remédier à cela, il faut implémenter dans ToolBarLayoutEditPolicyImpl les méthodes :

| | |
|---|---|
| `protected abstract Command` | **`createAddCommand`**`(EditPart child, EditPart after)`<br>Returns the `Command` to add the specified child after a reference `EditPart`. |

de OrderedLayoutEditPolicy et

| | |
|---|---|
| `protected Command` | **`getOrphanChildrenCommand`**`(Request request)`<br>Returns the `Command` to orphan a group of children. |

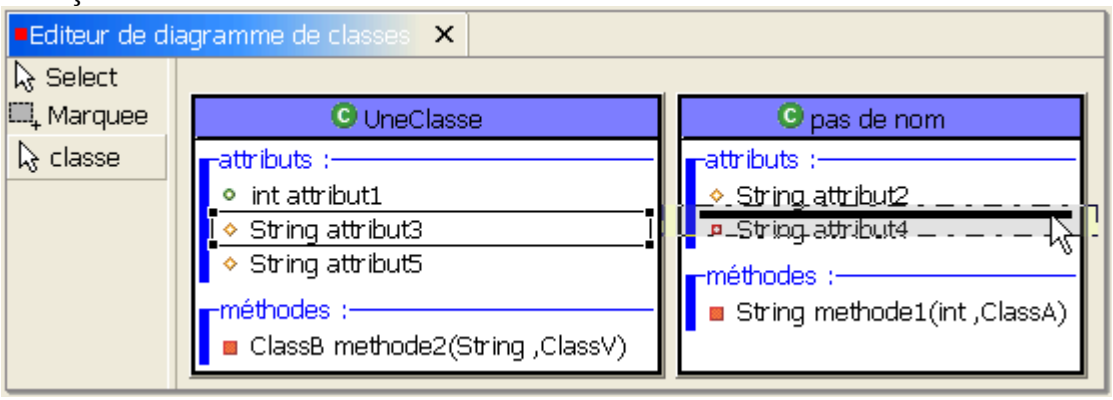
de LayoutEditPolicy.

Cependant, les requêtes de type orphan et delete sont envoyée à l'EditPart qui doit être supprimé et non à son parent. C'est l'EditPolicy ComponentEditPolicy qui se charge de forwarder au container parent ces deux types de requête (se rappeler qu'elles sont non-graphiques...). Il faut donc installer une ComponentEditPolicy pour le component_role des EditParts des attrubuts et des méthodes.

Il faut ensuite écrire les commandes correspondantes et mettre à jour les EditParts en fonction des notifications du modèle.

Pour terminer, il faut s'assurer que la méthode createAddCommand() renvoie null pour tout autre EditParts enfant que AttributeEditPart (pour le container ListAttributeEditPart) et idem pour les méthodes.

Et là ça marche :




**TODO : faire passer la section 7.10 dans le document sur gef.**

### 7.10) Actions, ActionRegistry :

But et description du principe :

Dans une interface graphique, il existe souvent de nombreux moyens différents de déclencher une même opération. Par exemple un menu item, une touche clavier et un bouton de toolbar. Les apparences de ces différents éléments dépendent d'un certain nombre de propriétés propres à l'action à exécuter. Par exemple son nom, le tooltip à afficher, l'icône, le fait que l'action soit actuellement exécutable ou non, etc.

C'est pour ces raisons qu'il est pratique de regrouper dans un même objet toutes les propriétés communes à ces différents contrôles, ainsi que le code d'exécution de l'action. C'est un tel objet qu'on appelle une action.

Les Actions servent de modèles pour tous les petits contrôles disponibles dans l'interface graphique pour exécuter une opération. Chaque fois qu'on veut pouvoir mettre à la disposition de l'utilisateur une opération particulière, de quelque moyen que ce soit (menu, toolbar, etc..) il faut commencer par écrire une Action personnalisée ou utiliser une des actions standard définies dans gef (redo, undo, delete, save, print, etc.).

Les actions possèdent un état propre qui dépend de l'état de certaines parties du workbench. Par exemple :

- les actions style delete ne devraient être enabled que quand il existe un objet sélectionné dans l'éditeur,
- les actions undo/redo ne devraient être enabled que quand il existe une commande dans la command stack qui peut être annulée/réexécutée.

L'état des action peut ou non se mettre à jour automatiquement au fur et à mesure les changements des propriétés des éléments du workbench (ou autres) qui entrent dans la définition de cet état. Ce sera par exemple le cas si l'action est un listener des éléments susceptibles d'en modifier l'état. A l'opposé, certaines actions implémentent l'interface UpdateAction. Elles sont alors munies d'une méthode update() et c'est au programmeur de veiller à ce que cette méthode soit appelée lorsqu'un événement susceptible de changer l'état de l'action survient.

Implémentation :

L'interface org.eclipse.jface.action.IAction définit l'API d'accès aux actions. La classe org.eclipse.jface.action.Action implémente IAction et doit être utilisée comme classe de base pour toute action. Gef étend cette classe Action (et d'autres classes dérivées d'Action) pour définir toute une série d'actions propres aux éditeurs Gef, définies dans le package org.eclipse.gef.ui.actions. Voir diagrammes.

Utilisation :

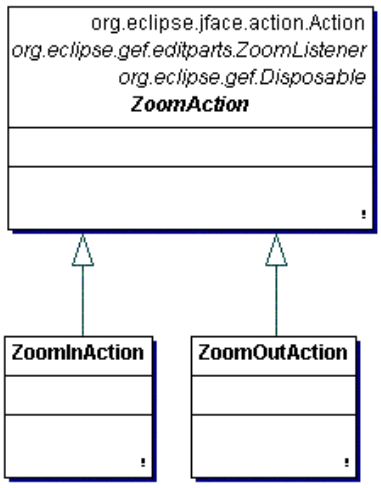
Gef fournit une classe dédiée à servir de container pour les actions : ActionRepository. Chaque éditeur Gef sur lequel sont définies certaines actions devrait posséder une donnée membre ActionRepository. ActionRepository permet d'appeler update() sur une liste d'actions définie par la liste de leurs ID (chaque action possède un identificateur).

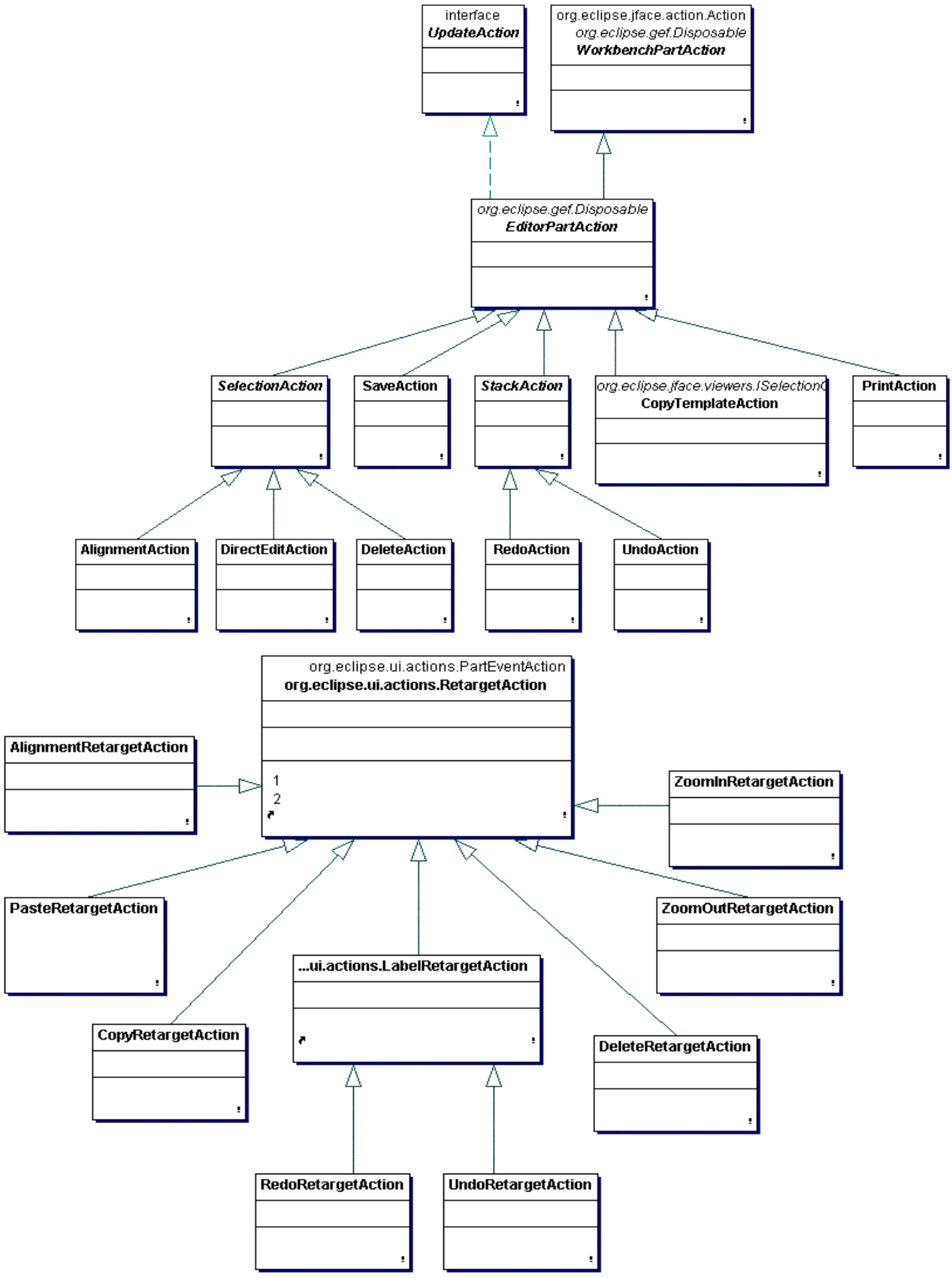
La bonne manière de faire devrait être la suivante :

- créer les actions une fois que l'éditeur est entièrement réalisé (je veux dire dans l'ordre chronologique du déroulement de l'exécution du programme...).
- les ajouter à l'ActionRepository de l'éditeur en maintenant par ailleurs : la liste des ID d'actions dont l'état dépend des changement survenant dans la CommandStack (redo, undo..), la liste des ID d'actions dont l'état dépend des changements de sélection dans l'éditeur, la liste des ID d'actions dont l'état dépend des changements des propriétés de l'éditeur,
- faire de l'éditeur un écouteur de la command stack, du service de sélection du workbench et de ses propres propriétés et réagir aux événements en appelant update() sur celle des trois listes d'actions qui convient selon le cas.

Pour des exemples, voir la classe GraphicalEditor et le logic example fournit par gef.

Diagrammes de classes du package org.eclipse.gef.ui.actions :






**TODO : faire passer la section 7.11 dans le document sur gef.**

### 7.11) réagir aux événements clavier :

Gef fournit une classe dédiée à cela : KeyHandler.

On ajoute un KeyHandler à un EditPartViewer par la méthode :

| | |
|---|---|
| `void` | **`setKeyHandler`**`(KeyHandler keyHandler)` : Sets the `KeyHandler`. |

Il faut absolument l'employer et non essayer de capturer les événements claviers d'une autre manière pour éviter certains problèmes avec l'utilisation des Tools.

On spécifie le comportement du KeyHandler lors d'une frappe clavier via la méthode :

| | |
|---|---|
| `void` | **`put`**`(KeyStroke keystroke, IAction action)`<br>Maps a specified `KeyStroke` to an `IAction`. |

La méthode run() de action est lancée lors de toute frappe clavier matchée par le keystrocke si action.getEnabled() retourne vrai (du moins si le Composite sur lequel est installé le viewer a le focus (?)).

### 7.12) permettre la suppression des classes, attributs et méthodes via la touche delete :

S'il fallait implémenter ça en partant de rien, il faudrait :

1. étendre la classe org.eclipse.gef.ui.actions.SelectionAction (qui est une classe spécialisée pour les actions dont l'état dépend de la sélection courante dans le workbench) pour faire la classe DeleteAction,
2. dans cette classe, implémenter calculateEnabled() pour vérifier que la sélection courante est bien constituée d'EditParts,
3. dans cette classe, implémenter run() pour lancer les req_delete vers chaque EditPart, recueillir les commandes correspondantes, les chaîner et les exécuter via la CommandStack,
4. dans cette classe implémenter init() pour définir l'ID de l'action, son icônes, son label, tooltip, etc....
5. créer un action registry dans l'éditeur,
6. enregistrer un écouteur du service de sélection du workbench qui appelle update() sur l'action chaque fois que la sélection change,
7. appeler dispose() sur l'action registry lorsque l'éditeur est « disposé »,
8. enregistrer l'action dans l'action registry,
9. ajouter un keyhandler au viewer sensible aux keystroke de type delete qui déclenche la DeleteAction.

Mais :

- DeleteAction est déjà fournie par gef,
- GraphicalEditor et GraphicalEditorWithPalette offrent des facilités pour la gestion des actions, et en particulier possèdent un ActionRegistry, mettent automatiquement à jour les actions dont l'état est sensible à celui de la CommandStack, à celui de l'éditeur ou à la sélection courante,

- GraphicalEditor et GraphicalEditorWithPalette ajoutent déjà à l'action registry les actions les plus courantes (undo, redo, delete, save, print).

Il ne reste donc plus qu'à :

- ajouter un keyhandler au viewer sensible aux frappes de la touche delete et déclenchant l'action delete,
- ajouter une ComponentEditPolicy à tous les EditParts susceptibles d'être supprimés de leur parent (ceci provoque **selon la javadoc** le « forwardement » des req_delete à l'EditPart parent sous forme de req_delete_dependant **mais c'est faux** il faut l'implémenter soi-même !),
- faire en sorte que le parent supprime l'enfant sous l'effet des requêtes de type req_delete_dependant (en implémentant les méthodes adéquates de LayoutEditPolicy ou ContainerEditPolicy) en revoyant les commandes correspondantes,
- ecrire les commandes de suppression des enfants,
- appeler refreshChildren lors de notifications d'événement issus de l'exécution de ces commandes.

Voir code pour tout ça...

**TODO : faire passer la section 7.13 dans le document sur gef.**

### 7.13) retargetable actions :

Ceci n'est pas un élément propre à Gef mais au workbench (voire à Eclipse en général).

Définition et buts du système :

Il existe des actions pouvant être accessoirement définies par de nombreuses vues et éditeurs dont le sens est commun bien que l'implémentation exacte diffère (les actions *nécessairement* définies par les vues et éditeurs comme save, save as, etc. ne sont pas concernées par ce paragraphe). Par exemple, undo, redo et print. Le workbench permet à toutes ces actions sémantiquement identiques de partager un même contrôle dans l'interface graphique du workbench. L'exécution de l'action est confiée à la vue ou à l'éditeur actif au moment où l'utilisateur actionne le contrôle, pour autant que la vue ou l'éditeur actif définissent une contribution à l'action. Dans le cas contraire l'action est désactivée.

De telles actions sont appelées retargetable actions ou global actions.

Utilisation :

Voir l'aide d'éclipse section « setting a global action handler », c'est très clairement expliqué. Ca se fait dans la méthode createControl de la vue ou de l'éditeur et les lignes de code à ajouter peuvent être calquées sur :

```
getEditorSite().getActionBars().setGlobalActionHandler(
                IWorkbenchActionConstants.ACTION_NAME,
                your_action);
```

L'interface IworkbenchActionConstants défini les noms des différentes retargetable actions auxquelles l'éditeur ou la vue peut contribuer, your_action est une action quelconque implémentant l'interface IAction.

### 7.14) undo et redo via touches ctrl+y et ctrl+z et le menu edit :

Undo et redo sont des retargetable actions définies par le workbench. Il suffit d'y contribuer en enregistrant les actions undo et redo de l'ActionRepository de GraphicalViewer en tant que global handlers pour ces actions.

Voir code, méthode createPartControl(Composite) redéfinie dans la classe de l'éditeur.

### 7.15) implémenter des actions personnalisées : création d'attributs et de méthodes via des raccourcis clavier

Le but est que lorsque l'utilisateur presse sur la touche 'a', un nouvel attribut (aux propriétés pour l'instant non paramétrables) soit créé dans la classe couramment sélectionnée. Idem pour la touche 'm' et les méthodes.

La manière de réagir à un événement clavier a déjà été expliquée. Reste à :

1. écrire les Actions qui vont créer l'attribut ou la méthode,
2. implémenter la méthode getCreateCommand(Request) dans la LayoutEditPolicy de ListEditPart (c'est-à-dire dans ToolbarLayoutEditPolicyImpl) pour renvoyer la commande de création d'attribut ou de méthode,
3. écrire les Commandes de création d'attributs et de méthodes,
4. s'assurer que ListEditPart effectue un RefreshChildren lorsqu'il reçoit la notification correspondant à l'ajout de méthode/attribut dans propertyChange(PropertyChangeEvent).

Les étapes 2, 3 et 4 sont routinières maintenant, reste donc l'étape 1 à expliquer.

L'état des actions à écrire (le fait qu'elles soient exécutables ou pas, ici) dépend de la sélection courante dans le workbench. Il doit s'agir d'un ClassEditPart unique. Gef fournit une super classe pour toutes les actions dont l'état est sensible à la sélection courante : org.eclipse.gef.ui.actions.SelectionAction. Le constructeur de cette classe prend en paramètre un IWorkbenchPart (le GraphicalEditor dans ce cas) et la classe introduit des méthodes facilitant l'accès à la sélection courante dans le workbench. Trois méthodes sont à redéfinir :

- run() : se base sur la sélection courante pour envoyer une requête de création d'attribut au ListAttributeEditPart enfant du ClassEditPart couramment sélectionné (idem pour les méthodes)
- calculateEnable() : se base sur la sélection courante pour déterminer si oui ou non l'action est exécutable.
- init() : pour ajuster toutes les caractéristique propre à l'ui de l'action (label, icône, tooltip, discription, etc...) et sont identificateur (indispensable !).

Voir rlemaigr.classdiagrameditor.actions.CreateAttributeAction.

Il faut encore s'assure que la méthode update() est appelées sur ces actions lorsque la sélection change dans le workbench. Pour cela, il faut ajouter les actions à l'action registry du GraphicalEditor et ensuite ajouter leurs identificateurs à la liste des actions sensible à la sélection. GraphicalEditor maintient une telle liste. Elle est accessible par la méthode getSelectionActions(). Lorsque la sélection change dans le workbench, le GraphicalEditor appelera automatiquement update() sur les actions dont les identificateurs se trouvent dans la liste getSelectionActions().

Voir méthode createActions() redéfinie dans la classe de l'éditeur.

### 7.16) Menus, MenuManagers, ContextMenuProvider :

Les Menus et les MenuItems qui les composent sont des widgets swt. Ils peuvent être utilisés seuls comme les Menu et les MenuItems de swing par exemple. Cependant JFace introduit le concept d'Action et les Actions doivent pouvoir être utilisées pour composer un Menu. Le lien entre les Actions de JFace et les MenuItems de SWT est fait par les MenuManager qui sont des éléments de JFace.

Un MenuManager est capable de construire un Menu SWT et de le peupler à partir d'Actions JFace. Des Actions peuvent lui être ajoutées et celles-ci se retrouveront dans le menu construit par le MenuManager. Un seul Menu peut être créé, si bien qu'il existe une relation 1-1 entre les MenuManagers et leurs Menus.

Lorsque le Menu est sur le point d'être affiché, il notifie son MenuManager. Celui-ci retire toutes les actions préalablement enregistrées (ce comportement est paramétrable) et ensuite notifie tous ses listeners, ce qui leur une chance d'ajouter au MenuManager les Actions nécessaires. Ensuite le Menu est peuplé sur base des actions qui sont enregistrées dans le MenuManager.

Gef fournit un MenuManager : org.eclipse.gef.ContextMenuProvider. Ce MenuManager est son propre listener et appelle la méthode buildContextMenu() lorsque le Menu est sur le point d'être affiché. Cette méthode est abstraite et est prévue pour être redéfinie de manière à y construire le menu en ajoutant des Actions aux menu manager via la méthode add(Action) par exemple.

Voir org.eclipse.examples.logicdesigner.LogicContextMenuProvider.

Chaque EditPartViewer peut être muni d'un tel ContextMenuProvider (ou de tout autre MenuManager) via la méthode

| | |
|---|---|
| `void` | **`setContextMenu`**`(MenuManager contextMenu)`: Sets the context `MenuManager` for this viewer. |

### 7.17) création d'attributs et de méthodes via un menu contextuel :

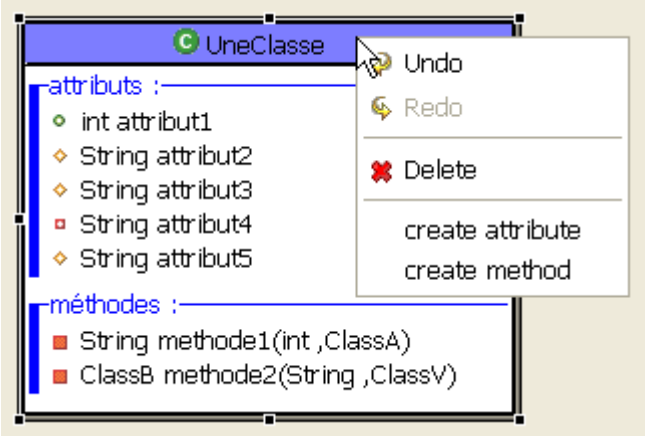
Les actions écrites dans le cadre de la section 7.15 et ajoutées à l'ActionRegistry du GraphicalEditor peuvent être réutilisées ici pour construire un menu contextuel, ainsi que toutes les autres faisant partie de l'ActionRegistry. Il suffit d'étendre la classe ContextMenuProvider et de passer au constructeur l'ActionRegistry pour permettre à la méthode buildContextMenu() de construire le menu à l'aide des actions se trouvant dans l'ActionRegistry.

On peut :

- soit ajouter toutes les actions présentes dans l'action registry au context menu, celles qui sont disabled apparaîtront alors grisées dans le menu,
- soit n'ajouter au menu que les actions enabled.

C'est la deuxième option qui est retenue ici.

L'implémentation du ContextMenuProvider est calquée sur celle du logic example. La classe est rlemaigr.classdiagrameditor.ContextMenuProviderImpl.




## 7.18) contribution à la properties view :

Si la properties view est visible, lorsque la sélection change dans le workbench, les éléments de la sélection courante sont interrogés via la méthode getAdapter(Class) en vue d'obtenir un proxy de type IPropertySource. L'interface IPropertySource comprend entre autres les méthodes :

| | |
|---|---|
| `IPropertyDescriptor[]` | **`getPropertyDescriptors`**`()`: Returns the list of property descriptors for this property source. |
| `Object` | **`getPropertyValue`**`(Object id)`: Returns the value of the property with the given id if it has one. |
| `void` | **`setPropertyValue`**`(Object id, Object value)`: Sets the property with the given id if possible. |

Pour contribuer à la properties view, il faut donc:

1. enregistrer un SelectionProvider auprès du workbench qui soit capable de fournir la sélection courante dans l'éditeur gef, à savoir les EditParts couramment sélectionnés,
2. que ces EditParts implémentent l'interface IAdaptable et retournent un proxy de type IPropertySource permettant d'accéder aux propriétés de leur modèle.

Le point 1. est fait d'office si on utilise un GraphicalEditor ou un GraphicalEditorWithPalette. En effet les GraphicalEditPartViewer sont des SelectionProvider et GraphicalEditor les enregistre automatiquement en tant que tel auprès du workbench lorsque l'éditeur est créé.

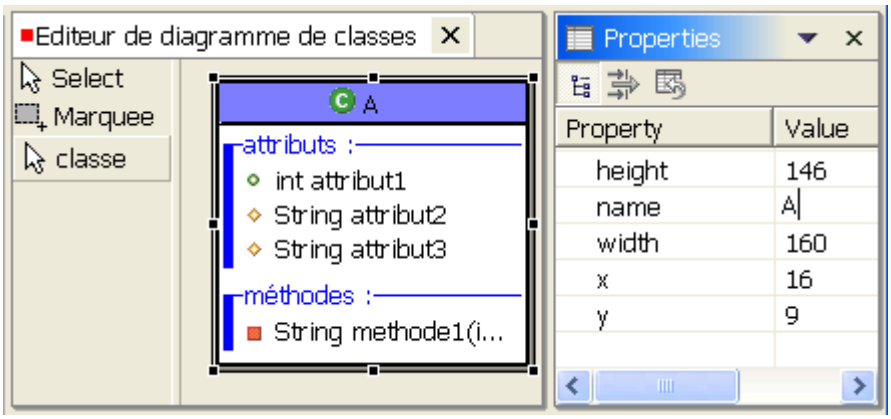
Le point 2. est déjà en partie réalisé puisque les EditParts implémentent l'interface IAdaptable et renvoient comme proxy pour la classe IPropertySource :

- soit le modèle lui-même s'il implémente cette interface,
- soit getModel().getAdapter(IPropertySource.class).

Reste donc à faire en sorte que chaque classe du modèle implémente l'interface IPropertySource. Voir code du modèle et voir logic example.

Il reste un problème à solutionner : les modifications introduite dans dans le modèle via la properties view ne sont pas incluses dans la command stack (ne sont pas « undoable ») et les modifications du modèle ne se répercutent pas sur la properties view s'il n'y a pas de changement de la sélection dans le workbench.

Gef règle ce problème en fournissant une property sheet particulière à la properties view. Chaque fois que nécessaire la properties view demande à l'éditeur actif de lui fournir une property sheet via la méthode getAdapter(Class aClass), où aClass est IPropertySheetPage.class. C'est là qu'il faut retourner une propertysheetpage qui convient. Voir code.




## 7.19) contribution à l'outline view :

[il y a des approximations dans les explications qui suivent, voir javadoc pour détails]

Si l'outline view est visible, lorsque l'éditeur actif change dans le workbench, le nouvel éditeur actif est interrogé via la méthode getAdapter(Class) en vue d'obtenir une IContentOutlinePage. Si l'éditeur retourne null, l'outline view indique « an outline is not available ». Sinon l'outline view montre le contenu de l'IContentOutlinePage et ajuste sa toolbar. Ce qui est visible dans l'outline view dépend donc de l'éditeur actif. En général on s'en sert pour afficher une certaine vue schématique du contenu de l'éditeur actif.

Une IContentOutlinePage est une IPage (un objet qui peut introduire des contrôles swt sous un Composite donné et contribuer à une ActionBar) et un ISelectionProvider. L'implémentation standard de IContentOutlinePage est org.eclipse.ui.views.outline.ContentOutlinePage et est basée sur un TreeViewer. Il faut étendre cette classe et fournir au TreeViewer un LabelProvider et un ContentProvider. Il faut aussi faire en sorte que l'arbre se mette à jour si le modèle de l'éditeur change, ça peut se faire en écoutant directement le modèle ou bien la CommandStack (ce que j'ai fait ici : plus simple mais moins efficace).

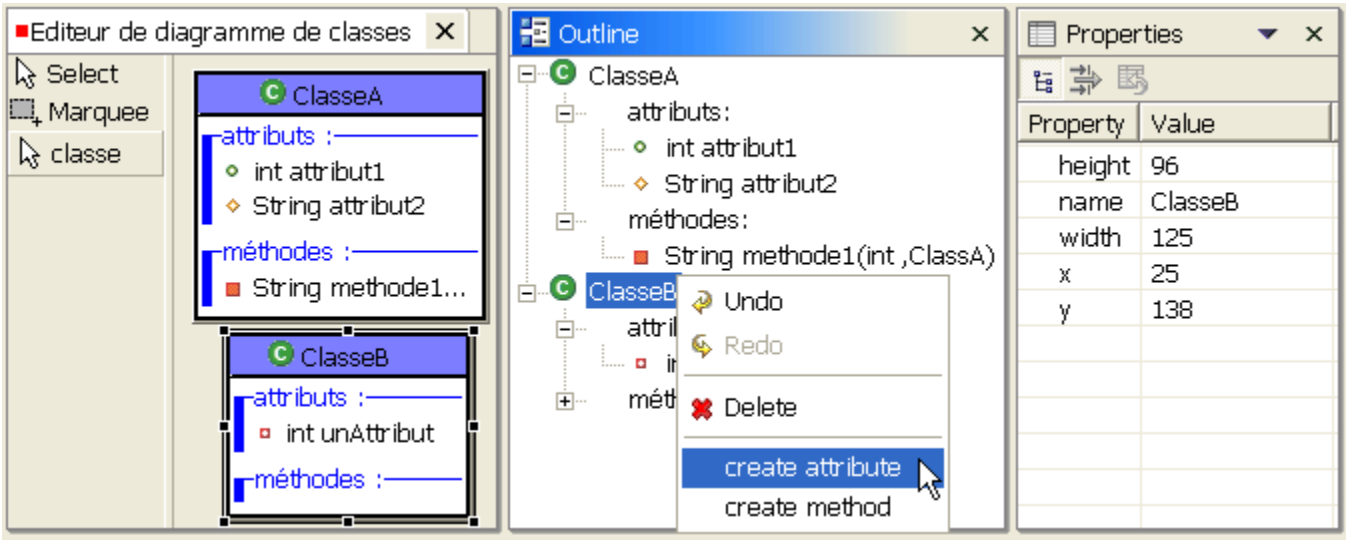
L'arbre (le content provider et l'input donc) peut être basé :

- soit sur le modèle de l'éditeur,
- soit sur les EditParts.

Ici j'ai choisi de le baser sur les EditParts, parce que d'une part c'est le plus simple puisqu'on peut profiter des méthodes getChildren() des EditParts pour construire le content provider et d'autre part, de cette manière, la sélection fournie par le TreeViewer au workbench est constituée d'EditParts, tout comme celle de l'EditPartViewer, et cela permet de réutiliser les Actions sensibles à la sélection définies pour l'éditeur. De cette manière, il est aussi possible de récupérer le MenuManager de l'EditPartViewer (ou du moins la classe) pour créer un menu contextuel pour l'outline view identique à celui disponible dans l'éditeur. Le

contenu de la properties view est aussi sensible à la sélection faite dans l'outline view puisqu'elle est de même nature que celle faite dans l'EditPartViewer.

Voir rlemaigr.classdiagrameditor.DiagramOutlinePage.




## 7.20) Draw2d : Polyline, PolylineConnection, ConnectionRouter, ConnectionAnchor...

Polyline :

Les polylines sont des figures draw2d (des Shapes en fait) qui ont pour graphisme une ligne brisée dont les points de passages ainsi que le style et l'épaisseur sont ajustables.

La méthode getBounds() est redéfinie pour retourner le plus petit rectangle englobant les positions courantes de tous les points de passage. La méthode primTranslate est redéfinie pour occasionner la translation des point de la Polyline (lorsqu'un ancêtre de la polyline se déplace, la polyline subit donc la même translation). La méthode containsPoint est redéfinie pour retourner vrai si le point se trouve sur la ligne brisée, à epsilon près.

ConnectionAnchor :

Les ConnectionAnchor sont des objets capables de renvoyer une position et qui notifient leurs listeners lorsque cette position est succeptible d'avoir changé. Il en existe plusieurs types. En général, ils sont attachés à une figure particulière dont ils écoutent les déplacements. La position retournée par le connectionAnchor est notamment calculée sur base de la position de cette figure. Lorsque la figure bouge, l'anchor notifie ses listeners.

PolylineConnection :

Extension de Polyline qui constitue une figure draw2d permettant de connecter deux objets de manière telle que lorsque les deux objets se déplacent, les extrémités de la connexion se déplacent avec eux et ses points de passage modifient leurs positions selon un algorithme de layout ajustable.

Les objets extrémités de la connexion sont des ConnectionAnchor ajustables que la PolylineConnection écoute.

L'objet responsable du layout des points intermédiaires de la connexion est un ConnectionRouter.

Il est possible de spécifier des décorations pour les extrémités de la PolylineConnection.

ConnectionRouter :

Ce sont les objets responsables de la disposition (du layout) des points de passage d'une PolylineConnection. Il en existe plusieurs types, voir javadoc.

Un des ConnectionRouter définis par draw2d est BendPointConnectionRouter. Les contraintes associées à ce ConnectionRouter sont des objets de type Bendpoint qui sont chacun capable de fournir un point de passage désiré de la connexion via la méthode getLocation(). Les bendpoints peuvent être des éléments plus complexes que de simple points. Par exemple le point de passage retourné par la méthode getLocation() d'un bendpoint peut dépendre de la position des extrémités de la connexion.

**Ne pas confondre** les Bendpoints qui sont des contraintes du BendPointConnectionRouter avec les points de passage de la polyline qui sont les positions effectives des points intermédiaires entre la source et la cible de la connexion.

Pour les détails supplémentaires, voir code de draw2d et javadoc.

### 7.21) Ajout de relations d'héritage entre les classes :

Modèle :

Modifications et ajout de nouvelles classes :

- Il faut ajouter à chaque nœud potentiel une liste de connexions sources et une liste de connexions cibles. Les seuls nœuds potentiels sont ici les classes, représentées par des objets de type ClassModel. On ajoute un List de connexions source et une List de connexion cible à la classe ClassModel. C'est nécessaire puisque ClassModelEditPart va devoir implémenter la méthode getModelSourceConnection et getModelTargetConnectionModel pour retourner les connexions dont le ClassModel est la cible et celles dont il est la source.
- Chaque connexion est représentée par un objet de la classe InheritanceModel. Cette classe a pour propriétés une source, une cible et un contrainte de type List contenant ses bendpoints.

Notification :

- Les connexions notifient les changements de leurs bendpoints (changement non-structuraux -> refreshVisuals() dans l'EditPart).
- ClassModel notifie tout changement dans ses connexions source et cible (changement structuraux -> refreshSourceConnections() ou refreshTargetConnections() dans l'EditPart).

Création, suppression et modification d'une connexion du modèle :

Tout cela se fait par accès aux seules méthodes setSource(ClassModel) et setTarget(ClassModel) de la classe InheritanceModel. Ces méthodes font les appels adéquats sur les CalssModel source et cible pour leurs ajouter/supprimer la connexion courante et leur délègue donc par la même occasion la tâche de notifier les changements. Voir code.

EditParts :

ClassEditPart doit maintenant implémenter l'interface NodeEditPart, nécessaire pour retourner les anchors de source et de cible pour les connexions qui souhaitent s'y accrocher. ClassEditPart doit aussi écouter les notifications de son modèle indiquant des changements

dans les connections dont il est la source ou la cible et appeler les méthodes de rafraîchissement structurel adéquates.

InheritanceEditPart doit étendre AbstractConnectionEditPart. La méthode refreshVisuals doit être redéfinie pour mettre à jour les bendpoints associés au connection router du connection layer pour la figure de la connection. La méthode createFigure est redéfinie pour retourner une polylineconnection décorée avec une extrémité en flèche. InheritanceEditPart écoute les notifications de son modèle indiquant une modification de la liste des bendpoints et appelle refreshVisuals si besoin.

La méthode refreshVisuals de DiagramEditPart est redéfinie pour ajouter un ConnectionRouter au connection layer du root editpart de viewer. Je sais que l'endroit pour faire ça est assez curieux mais ça semble être consacré par la pratique (selon les exemples que j'ai vus).

EditPolicies :

- Ajout d'une implémentation de BendpointEditPolicy à InheritanceEditPart pour permettre la modification des bendpoints,
- Ajout d'une implémentation de EndPointEditPolicy à InheritanceEditPart pour introduire les handles permettant l'obtention d'un DragTracker pour déplacer les extrémités de la connexion,
- Ajout d'une implémentation de ConnectionEditPolicy à InheritanceEditPart pour en permettre la suppression,
- Ajout d'une implémentation de GraphicalNodeEditPolicy à ClassEditPart pour permettre la création de nouvelle connections et la reconnexion des connexions.
- Ajout d'une implémentation de ComponentEditPolicy à ClassEditPart qui fait en sorte que toutes les connections de la classe soient supprimées avec la classe.

+ Toutes les Commandes qui vont avec.

Voir code...