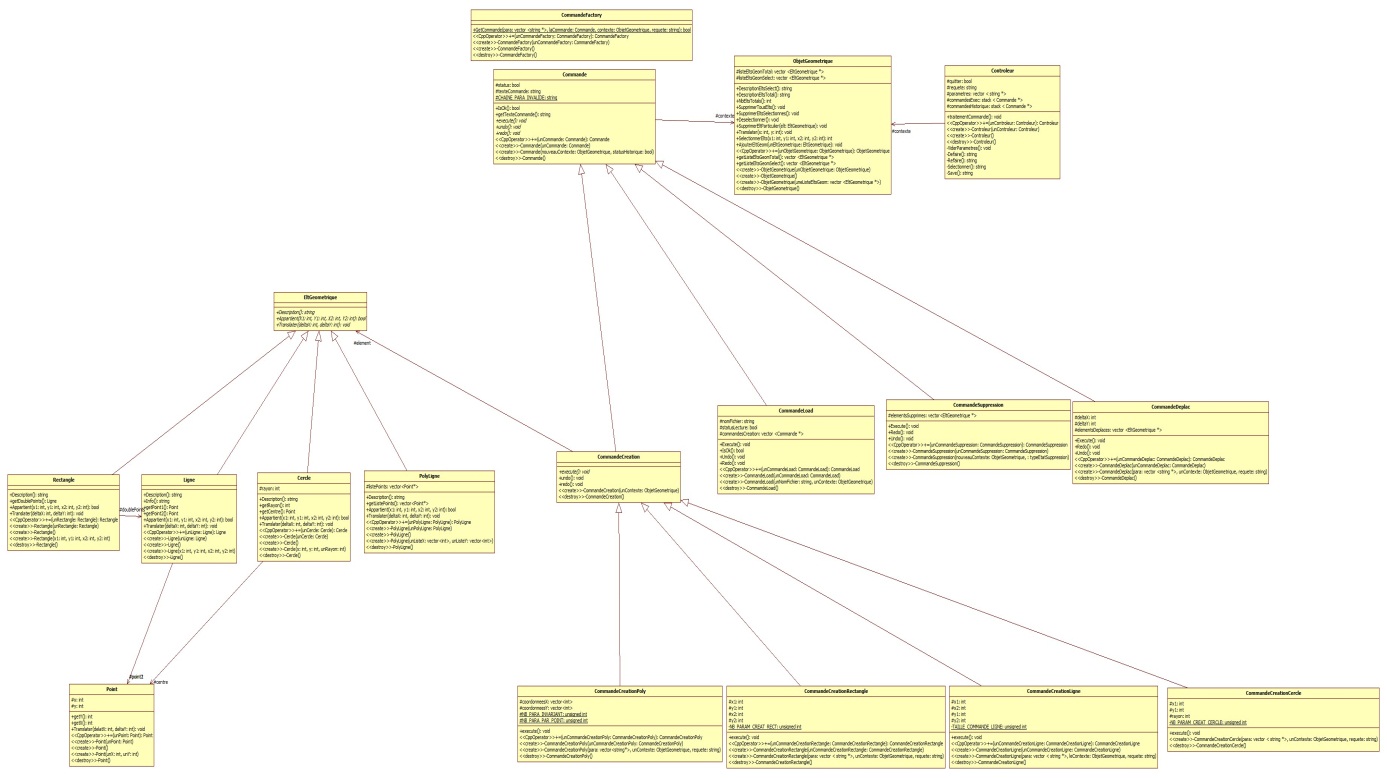
|  |
| --- |
|  |
| TP3 : Héritage, polymorphisme en C++ |
| Document de conception |
| **Binôme B3336** |
| Pierre Fouchet  Arnaud Mery de Montigny |
| **Le 16 février 2012** |

# Diagramme UML



# Explications du diagramme

Nous avons pris au départ le design pattern MVC. En remarquant que la partie vue était très petite et que nous travaillions sur la console, il n’était pas adapté de faire une partie vue. Nous avons à la place intégrer tout ce qui est IHM dans la partie Contrôleur.

Le code source de notre programme est donc séparé en deux parties : La partie modèle et la partie contrôleur. Nous avons un fichier Service.cpp et Service.h en dehors de ces deux parties qui regroupent des méthodes et des énumérations générales susceptibles d’être utilisées par toutes les classes.

## Modèle

##### ObjetGeométrique

Nous avons une classe principale **ObjetGeométrique** qui possède deux listes d’instances de la classe EltGéométrique. La première liste correspond à tous les éléments géométriques présents dans le modèle. La deuxième correspond à tous les éléments présents dans la première liste et qui sont sélectionnées. Cette classe comporte différentes méthodes qui interagissent avec cette liste ou avec les éléments des deux listes. C’est par le bais de cette classe que la partie contrôleur aura accès aux modèles.

La classe peut renvoyer une description des éléments totaux et sélectionnés qui correspond aux commandes de création des chaque élément.

La classe peut renvoyer le nombre d’éléments de la liste des éléments totaux.

Elle peut supprimer les éléments sélectionnés et totaux. La suppression d’un élément géométrique n’est pas définitive. Il s’agit juste du lien entre l’élément géométrique et la classe qui est supprimé. Cela permet de retrouver les éléments si le client tape la commande UNDO.

Elle peut sélectionner des éléments et désélectionner les éléments sélectionnés.

Elle peut translater les éléments sélectionnés.

Elle peut ajouter un ou plusieurs éléments géométriques

##### EltGeometrque

Cette classe est une interface, elle ne comporte que des méthodes virtuelles non implémentées.

Elle comporte la méthode Translater qui translate un élément géométrique d’un certain vecteur, la méthode Appartient qui permet de dire si un élément géométrique est inclus dans un rectangle défini par deux points, la méthode Description qui renvoie la description d’un élément dont la forme a déjà été définie (cf ObjetGeometrique).

L’utilité de cette classe est de permettre à la classe de posséder une liste d’éléments géométriques généraux. Ces éléments seront définis dynamiquement. C’est-à-dire que dynamiquement, un élément géométrique pourra être un rectangle ou alors un cercle. Cela dépend du contexte. Cela évité d’avoir une liste de rectangles, une liste de cercles, …

##### Ligne, PolyLigne, Cercle, Rectangle

Ces classes héritent directement de la classe EltGeometrque. Elles implémentent chacune les méthodes de la classe mère différemment. Le rectangle est identique à la ligne dans notre cas présent pour deux méthodes. Une ligne est en effet définie par deux points ainsi que le rectangle. L’inclusion à un autre rectangle est la méthode pour les deux. On a préféré différencier car cela ne rajoute beaucoup de complexité d’abord. Cela donne une plus grande cohérence aux modèles car du point de vue de la réalité, une ligne n’est pas un rectangle.

##### Point

Cette classe est utilisée par les quatre classes définies au-dessus. Un élément géométrique est défini par un ou plusieurs points. Lorsque l’on translate un élément géométrique, on translate en réalité tous les points qui le définissent. Lorsque l’on regarde l’inclusion d’un élément géométrique à un rectangle, on regarde si tous les points qui le définissent sont inclus dans ce rectangle.

## Contrôleur

##### Contrôleur

Cette classe est la classe principale de cette partie et de l’application. C’est cette classe qui interagit avec le client. Elle possède une instance de l’objet Géométrique et c’est par cette instance qu’elle va modifier le modèle, envoyer des requêtes au modèles. Cette classe récupère les commandes du client. Elle regarde pour savoir si la commande est une commande qui ne modifie pas le modèle. Si c’est le cas elle interprète elle-même la commande et l’exécute. Ces commandes qui ne modifient pas le modèle sont :

* S X1 Y1 X2 Y2
* LIST
* SAVE
* COUNT

La classe interprète trois commandes qui modifient le modèle mais de façon non générale.

Il s’agit de :

* UNDO
* REDO
* EXIT

UNDO et REDO sont des commandes qui modifient le modèle sur des commandes qui ont déjà été exécutés. UNDO exécute la dernière commande changeant le modèle mais inversement (Par exemple : Si la commande crée un rectangle, la commande UNDO la supprime). La commande REDO exécute la dernière commande qui vient de subir un la commande REDO.

EXIT est une commande spéciale car il s’agit de la dernière commande exécutée par un client. Elle ne peut subir un UNDO ou un REDO. Elle modifie le modèle car elle le supprime réellement.

Si la commande ne correspond pas à ceux ci-dessus, la classe demande à la classe CommandeFactory d’interpréter la commande et de renvoyer une commande exécutable à la classe.

Toutes les commandes renvoyées par cette classe sont empilés dans une pile.

Lorsque l’on reçoit la commande UNDO, on enlève la commande de la pile des commandes qui ont été exécuté. On exécute le undo de la commande en haut de la pile et on insère la commande dans la pile des commandes qui ont subis un UNDO.

Lorsque l’on reçoit la commande REDO, on enlève la commande en haut de la pile des commandes qui ont subis un UNDO. On exécute cette commande et on l’empile sur la pile des commandes qui ont été exécuté.

##### CommandeFactory

Cette classe interprète les commandes qui modifient le modèle à part UNDO, REDO et EXIT (voir précédemment) et renvoient une commande à la classe Contrôleur qui peut être exécutée. Ces différentes instances implémentent toutes le design pattern *Command*. Ce patron de conception sera détaillé au prochain paragraphe.

Les commandes interprétées par cette classe sont les suivantes :

* C X1 Y1 R
* R X1 Y1 X2 Y2
* L X1 Y1 X2 Y2
* PL X1 Y1 … Xn Yn
* DELETE
* MOVE X1 Y1
* LOAD
* CLEAR

##### Commande

Cette classe est une classe abstraite implémentant le design pattern *Command* qui permet d’uniformiser les traitements et de pouvoir choisir le moment exact de l’exécution de la dite commande. Elle permet le choix du type d’un objet dynamiquement. C’est le même cas que pour EltGeometrique.

Cette classe possède deux attributs : un string qui est la réponse de la commande et une instance d’ObjetGeometrique.

Cette classe possède trois méthodes matérialisation cette uniformisation de traitements :

* Execute
* Undo
* Redo

Ces méthodes ne sont pas implémentées par cette classe.

Undo exécutent la commande inversement et redo exécutent de nouveau la commande.

##### CommandeCreation

Cette classe hérite de la classe Commande mais elle est aussi une classe abstraite.

En réalité, seule la méthode execute n’est pas définie. La méthode undo et redo sont définis.

La classe possède une instance d’un élément géométrique.

undo suppriment l’élément géométrique de l’instance de l’ObjetGeometrique et redo ajoute l’élément géométrique à l’instance de l’ObjetGeometrique.

##### CommandeCreationCecle, CommandeCreationRectangle, CommandeCreationLigne, CommandeCreationPolyLigne

Ces classes héritent de la classe CommandeCreation. Elles implémentent la méthode execute et leur constructeur.

La méthode execute est différente de redo car elle définit dynamiquement le type de l’instance de l’EltGeometrique.

##### CommandeDeplac

Cette classe hérite de la classe Comande. Elle permet de translater les éléments sélectionnés par le biais de la méthode Translater de l’instance de l’ObjetGeometrique.

##### CommandeLoad

Cette classe hérite de la classe Comande.

Cette classe lit un fichier texte passé par le client. Ce fichier ne doit comporter que des commandes de création. Cette classe recrée des commandes de créations.

Elle exécute toutes ces commandes.

##### CommandeSuppression

Cette classe hérite de la classe Comande.

Elle supprime tous les éléments sélectionnés

Cette hiérarchie d’héritage permet de factoriser le code via les notions d’héritage et permet de mettre en place un polymorphisme efficace dans l’application. La contre-parti est le fractionnement des classes en de nombreuses sous-classes, qui rendent un peu plus complexe le suivi de l’application.

# Structure de données

Comme l’indique le diagramme de classes de l’application, le programme utilise de nombreuses listes. Les *stack* ont étés retenues pour stocker les listes de commandes car les fonctionnalités undo et redo ont seulement besoin d’accéder à la dernière commande exécutée (ou annulée, selon le cas). Insertion, accès et suppression se faisant en O(1).

Les *vector* ont été retenus pour stocker les autres ensembles tels que les ensembles d’éléments géométriques. En effet l’insertion se fait en O(1) si l’on fait en tête (ce qui est le cas dans notre application, car il n’y a pas d’ordre d’éléments géométriques) et l’accès peut se faire en O(1). Ce dernier point a soulevé une erreur de conception, trouvée très tardivement. En effet, dans les cas où le programme a besoin de vérifier l’existence d’un élément présent dans un *vector,* celui-ci est obligé à parcourir l’ensemble du contour.