Robin Nicolet
Jérôme Guidon
Ada-Maaria Hyvärinen

B3429

TP C++ - Héritage

EDITEUR DE FORMES GEOMETRIQUES

Document de conception

Spécifications supplémentaires (voir énoncé)

Rappel

Notre application permet de dessiner un dessin dans un plan 2D, contenant au choix les formes géométriques suivantes : cercle, rectangle, ligne, poly-ligne ainsi que des objets agrégés (groupes) regroupant plusieurs éléments des 4 premières catégories.

Contour du dessin

Dans notre programme, les coordonnées sont exprimées en long int. Le dessin est donc limité par la limite du type *long int* sur ses deux axes. La valeur max du *long int* dépend de l'environnement, mais les *long int* sont garantis d'être **au moins** compris entre -2 147 483 648 à 2 147 483 647. Tout entier renseigné qui dépasserait la limite génèrera une erreur (les déplacements hors limite sont aussi impossibles).

Prises d'initiatives

Le programme ne prend **aucune prise d'initiative**. On ne rajoute pas de suffixe sur le nom lorsqu'on tente d'ajouter un élément et que celui-ci est déjà pris. Aucune alerte si l'utilisateur quitte le programme sans avoir sauvegardé.

Annulation

De manière générale, toute opération qui échoue en plein milieu doit être complètement annulée. On est sur un système de **tout ou rien**. (Ex : Supprimer, charger un fichier …)

Il est possible d'annuler toutes les opérations effectuées **depuis le début** du programme (mise à part le listage des éléments du dessin bien entendu). Une fois qu'on indique une instruction au programme, les instructions annulées sont détruites.

Chargement d'un fichier

L'utilisateur peut aussi sauvegarder et charger un fichier. Le nom de fichier donné peut être un chemin absolu ou relatif. Le chargement se fait par-dessus le modèle existant. Si un nom est déjà pris, on annule le chargement.

Le programme ne gère pas les **noms de fichier avec espace**, il indiquera une erreur, car deux noms lui sont donnés.

Sauvegarde

Pour le nom des fichiers, voir le paragraphe précédent. Il est possible de sauvegarder un dessin vide, le fichier sera vide.

Objets Agrégés

On ne peut pas créer d'objet agrégé vide, mais un objet agrégé qui devient vide par la suite n'est pas supprimé. La **suppression d'un objet agrégé** n'entraîne pas la suppression des objets qu'il contient.

Architecture générale du programme

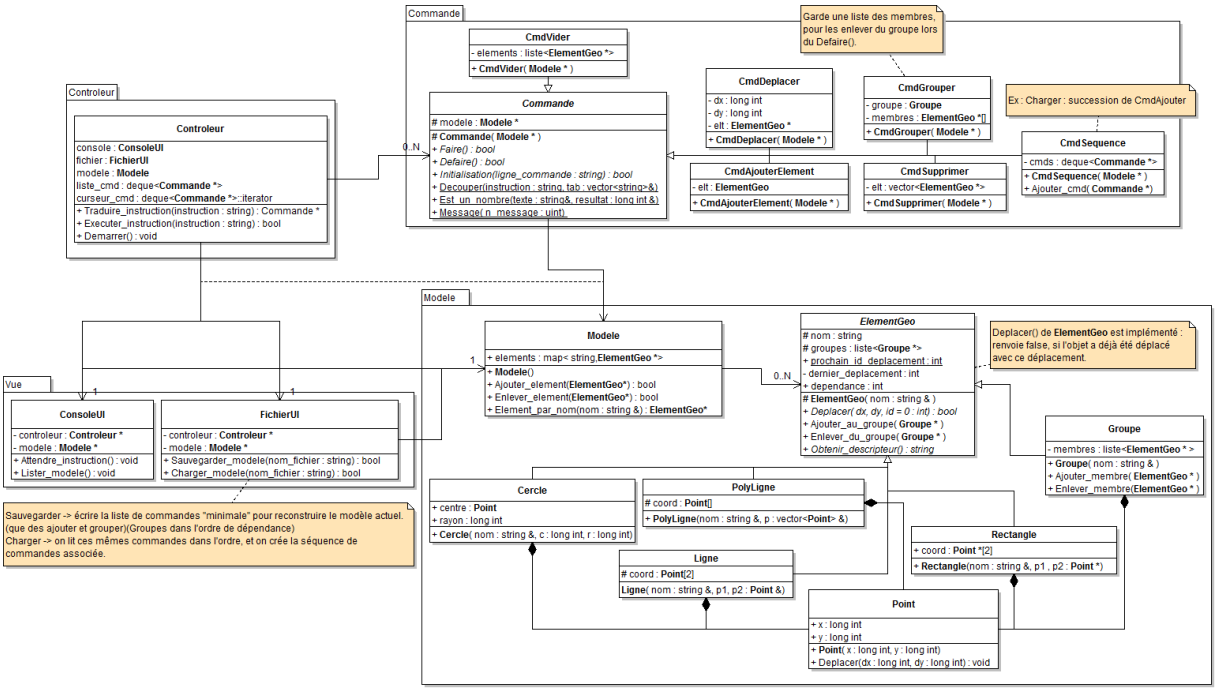

Figure 1- Diagramme UML complet

Pour répondre au cahier des charges, nous avons choisi de structurer notre programme selon le patron architectural **MVC** et le design pattern **Commande**. Ceci nous permet d'implémenter facilement les *UNDO* et *REDO*.

Nous avons donc divisé notre programme en quatre morceaux : *Modele*, *Vue*, *Controleur* et *Commande*.

Modele

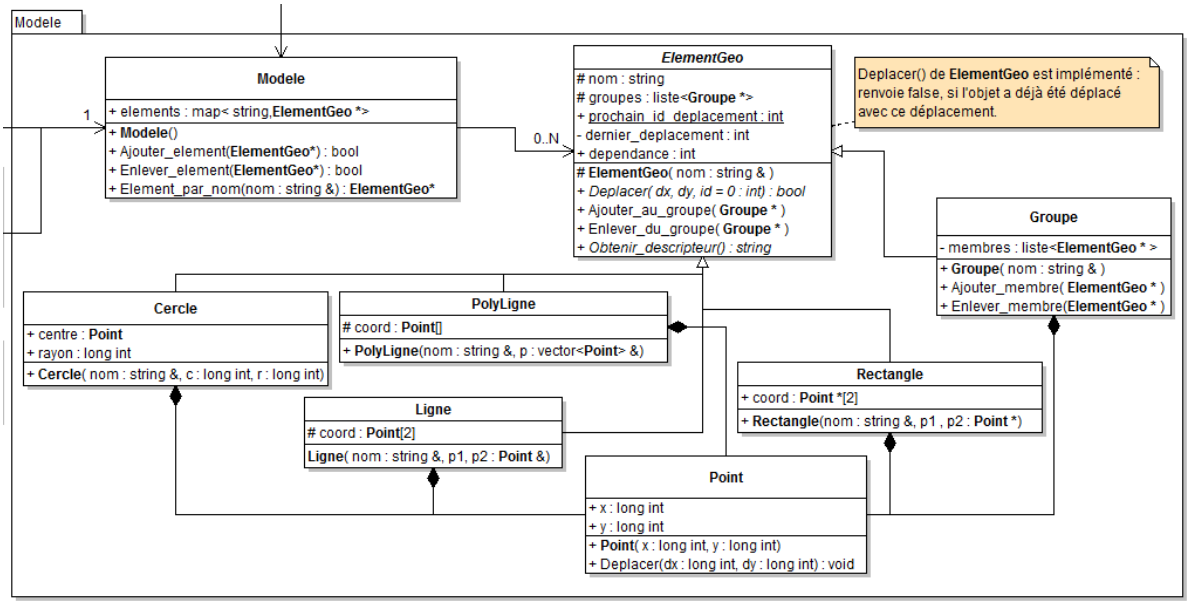

Figure 2- La package Modele

Le package *Modele* représente toutes **les données géométriques** d'un point de vue théorique, ainsi que les interactions entre chaque classe. Ce package est totalement **indépendant** des autres.

Composition du package

Le package comporte la classe *Modele*, qui contient les éléments de dessin qui héritent tous de *ElementGeo*. *ElementGeo* demande d'implémenter la méthode *Deplacer*. On détaillera plus loin cette méthode.

Commande

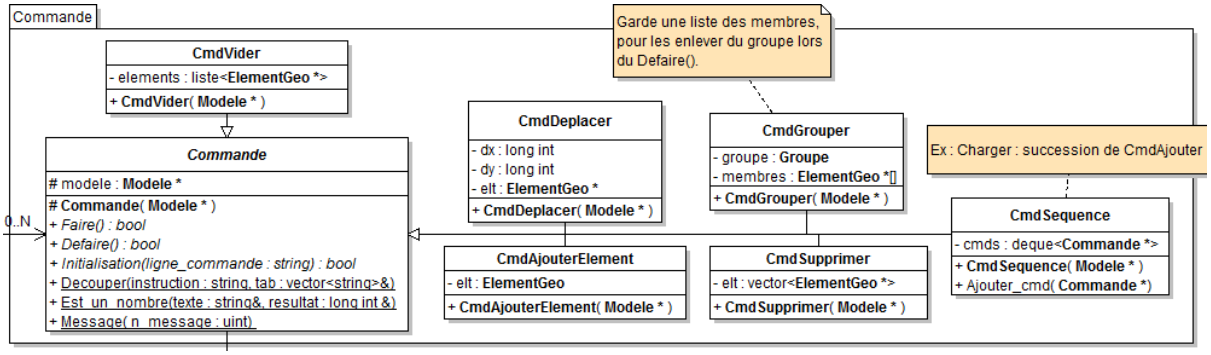

Figure 3 - Le package Commande

Le package *Commande* représente **les actions qui modifient le modèle**, et qui sont donc annulables. Il n'y a donc pas de commande pour sauver le modèle ou lister les éléments.

Les classes de ce package sont donc dépendantes du modèle.

Composition du package

Toutes les classes de ce package héritent de la classe *Commande* qui demande d'implémenter 3 méthodes : *Faire* (Do/Redo), *Defaire* (Undo) et *Initialisation*. Ces méthodes nous permettent de manipuler le modèle selon le cahier des charges.

Controleur

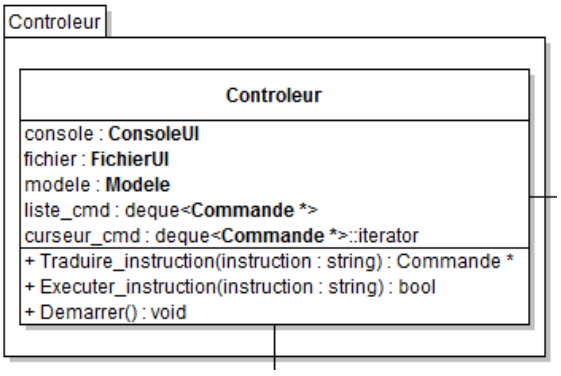

Figure 4 - Package Controleur

Le package *Controleur* se résume à une unique classe homonyme. C'est le **cerveau** du programme. C'est lui qui exploite ce que l'entrée lui donne pour modifier le modèle et fournir la bonne sortie.

Vue

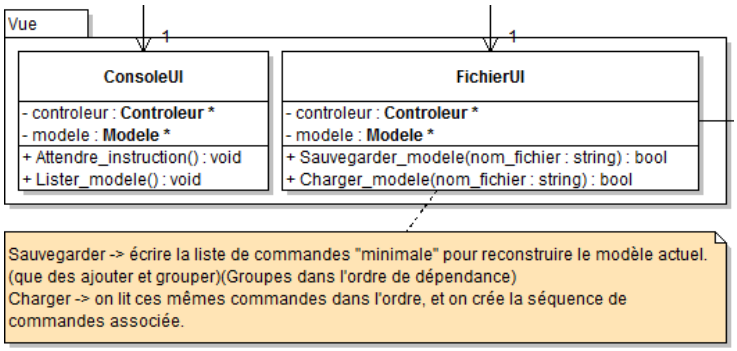

Figure 5 - Package Vue

Le package Vue contient deux classes :

- *ConsoleUI* permet de communiquer avec l'utilisateur en mode console.
- *FichierUI* permet de communiquer à travers un fichier (chargement/sauvegarde).

Les classes du package ont accès au modèle.

Architecture détaillée – Exemples de polymorphismes et d'héritage

Intéressons-nous à quelques points du programme qui méritent notre attention. Nous expliquerons le problème et verrons comment nous l'avons résolu.

Déplacer les éléments

Le problème

Lors du déplacement d'un objet agrégé, celui-ci peut contenir plusieurs références à un même objet (par l'intermédiaire d'autres objets agrégés). Il ne faut donc pas déplacer plusieurs fois cet élément.

Notre solution

La méthode *Deplacer* demande un entier qui est *0* par défaut. Quand on veut déplacer un groupe d'objet, on attribue un **identificateur** (ou **id**) au déplacement (*prochain_id_deplacement*). Lorsqu'on déplace un élément avec un id autre que *0*, on n'**autorise le déplacement** (on implémente la fonction de la classe mère pour qu'elle joue ce rôle) que si le dernier déplacement effectué sur l'élément est **différent de id**.

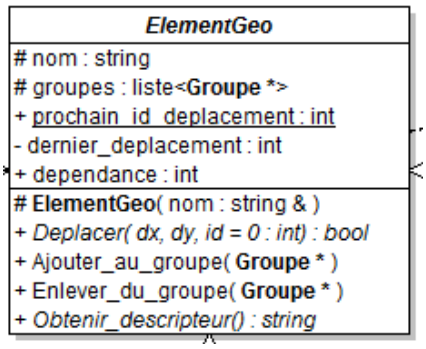

Figure 6 - Classe ElementGeo

Les méthodes Deplacer ont donc cette allure :

ElementGeo

```
Deplacer( dx, dy, id )
    Si dernier deplacement == id
    ET id != -1 alors
        renvoyer faux
    Sinon
        dernier_deplacement = id
        renvoyer vrai
```

Classes filles

```
Deplacer( dx, dy, id )
    Si ElementGeo ::Deplacer(dx,dy,id)
    alors
        {…Deplacer…}
    Sinon
       {ne rien faire}
        renvoyer vrai
```

Chargement d'un dessin

Le problème

Le chargement d'un dessin pose un problème majeur : on doit ajouter tout une série d'élément, mais on doit l'annuler en une fois.

Notre solution

Nous avons crée un type de commande que nous appelons *CmdSequence*, qui symbolise une suite (séquence) de commandes.

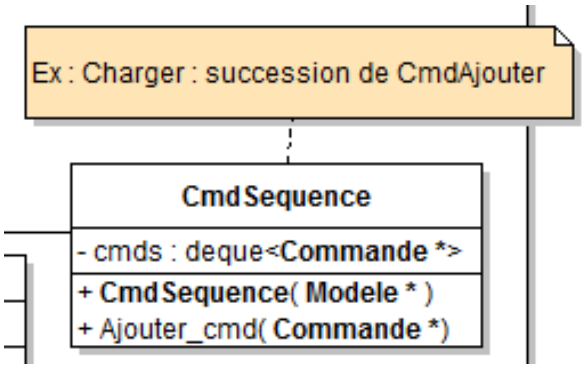

Figure 7 - Classe CmdSequence

Ainsi le problème est résolu grâce au **polymorphisme**. En effet les méthodes *Faire* et *Defaire* de la séquence vont appeler les méthodes respectives de chacune des commandes. Et du point de vue du controleur, tout cela n'est **qu'une seule et unique commande**.

Notons que les commandes d'une sequence pourraient être des séquences elles aussi, de la même manière qu'un objet agrégé peut être contenu dans un autre objet agrégé. Mais ce ne sera jamais le cas dans notre programme.

Sauvegarde du dessin

Le problème

Lorsque l'on sauvegarde un dessin, on ne peut pas simplement parcourir tous les éléments et enregistrer leur descripteur dans un fichier. Il faut respecter un certain ordre, que nous appelons **ordre de dépendance**.

En effet, si les lignes, poly-lignes, cercles et rectangles sont indépendants les uns des autres, ce n'est pas le cas des objets agrégés.

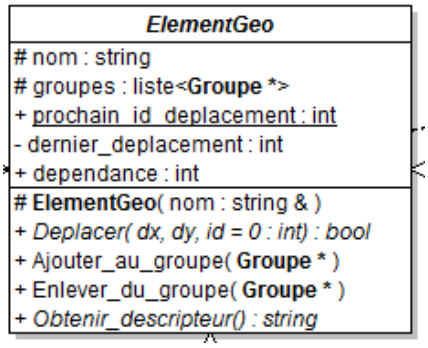

Figure 8 - Classe ElementGeo

Notre solution

Nous avons ajouté un attribut à *ElementGeo* qui permet de connaitre le **niveau de dépendance** d'un *ElementGeo* (cf. Figure 9). On peut se demander **pourquoi** mettre cet attribut dans *ElementGeo* et non dans *Groupe* puisque seuls les objets agrégés sont concernés ? Parce que lorsque nous enregistrons les éléments nous ne savons pas ce qu'ils sont, et qu'on peut imaginer que par la suite on veuille créer un nouveau type d'objet dépendant.

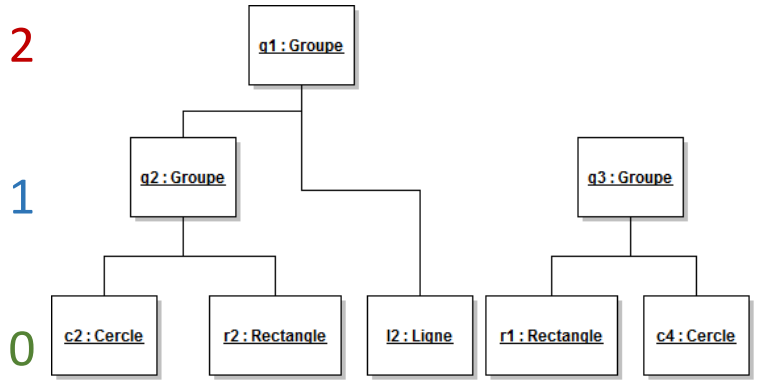

Figure 9 - Niveau de dépendance

Lors de la sauvegarde, il ne nous reste plus qu'à écrire les descripteurs dans **l'ordre croissant de dépendance**, indépendamment du type de l'élément.