TP4 : Héritage, Polymorphisme

Document de conception

Table des matières

Introduction

Dans ce 4ème TP de C++ il nous était demandé d'implémenter un éditeur de formes géométriques : *Cercle, Rectangle, Ligne, Polyligne* et certaines opérations sur ces formes telles que la suppression, le déplacement, la sélection. Les objectifs du TP étaient principalement de (re)voir des notions telles que :

— l'héritage
— le polymorphisme
— l'utilisation des design pattern (ou patrons de conception)
— l'évaluation des performances d'un programme

Nous avons ainsi pu mettre en pratique les notions vues en cours. Dans ce document vous trouverez une explication détaillée de notre programme et des choix effectués.

1 Conception

1.1 Design Pattern

Pour la réalisation du TP, nous avons combiné deux design patterns :

— **Pattern Singleton :** qui permet de restreindre l'instanciation d'une classe à un seul objet. Ainsi il n'y aura qu'une seule instance de la classe principale qui sera accessible par les autres classes qui en auront besoin directement notamment les commandes.
— **Pattern Commande :** qui permet une séparation optimale du code initiateur de l'action du code de l'action elle-même. A Chaque commande, doit correspondre une

classe qui possède deux méthodes publiques : une permettant d'exécuter la commande, l'autre d'annuler l'effet de la commande.

De l'utilisation de ces deux commandes, a découlé une séparation de l'application en deux grosses parties :

— le **cœur** de l'application appelée **Modele** (cf. figure 1) avec une seule instance (geoEdit) composée de classes représentant les différentes formes géométriques.

— la classe **Commande** (cf. figure 2) avec ses descendants permettant de gérer les entrées et sorties. Cette modélisation permet dans le futur de réutiliser le cœur de l'application avec une interface graphique par exemple

1.2 Diagramme de Classes

Une version d'une meilleur résolution du diagramme de classe est fournie dans le répertoire ./doc. Il s'agit du fichier *B3425Diagramme.pdf*.

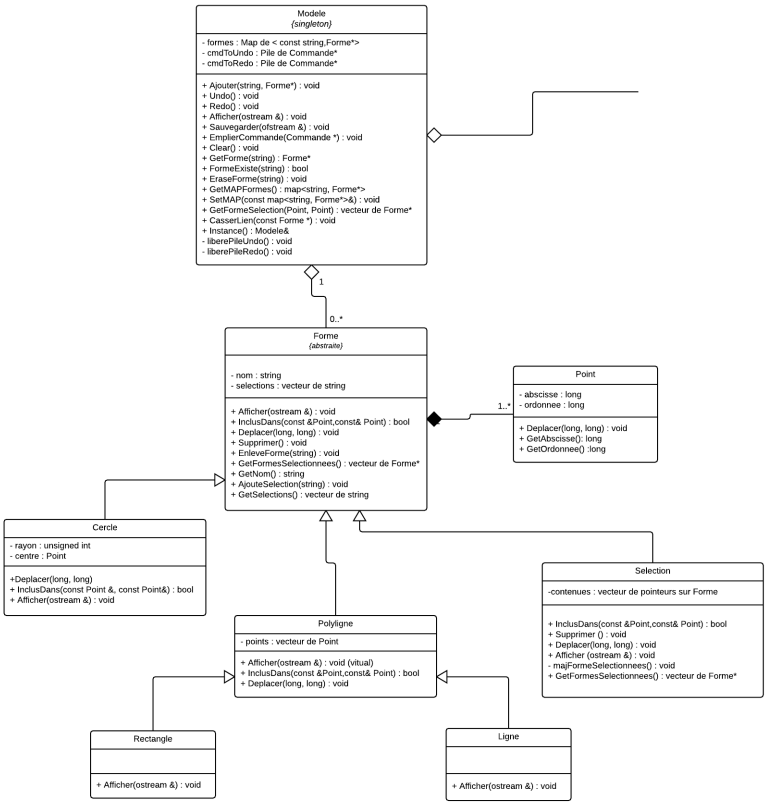

Figure 1 – Cœur de l'application

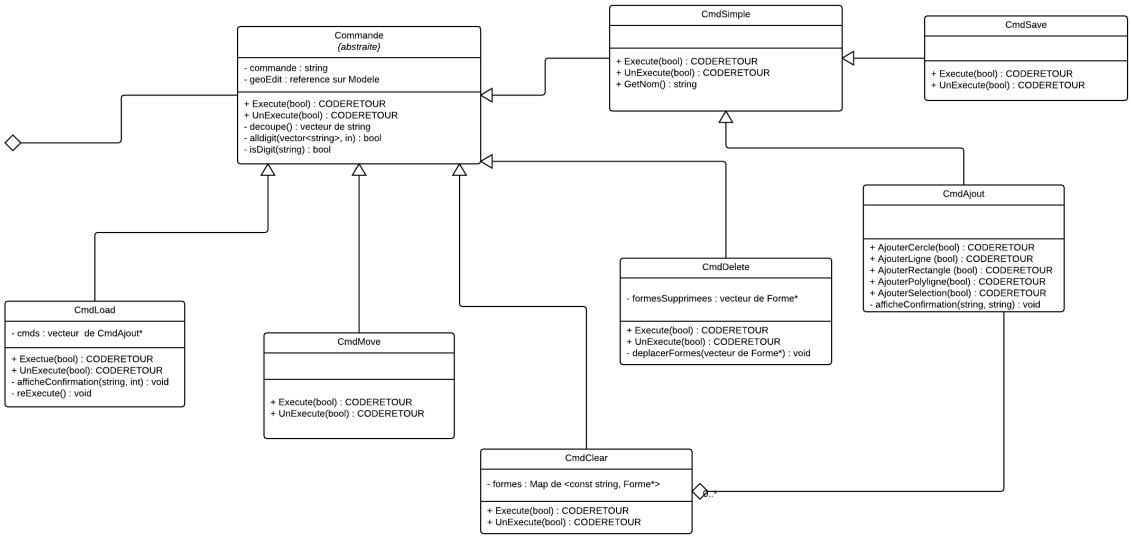

Figure 2 – Commande

2 Spécifications

Ici seront présentées les principales classes et leur méthodes et l'interaction qui se fait entre les classes.

2.1 Modele et Commande

La classe Modele, en tant que classe principale, peut être vue comme *une base de données* qui se charge de stocker toutes les formes et les commandes et fournis un certain nombre *d'opérations ou méthodes* aux commandes qui interagiront avec elles.

Les formes sont stockées dans une MAP de type : *map<const string, Forme*>* . Ainsi une forme peut être retrouvée grâce à son nom.
A coté de cela, nous avons deux piles de pointeurs sur Commande permettant de gérer le UNDO et le REDO des commandes exécutées :

- **void Undo(bool)**
  Cette méthode de la classe principale vérifie si la pile *cmdToUndo* est vide. Si ce n'est pas le cas, annule la dernière commande ayant modifiée le modèle grâce à la méthode **UnExecute** de la classe commande, empile cette commande sur la pile *cmdToRedo* et dépile *cmdToUndo*.
- **void Redo(bool)**
  Exécute les mêmes actions mais en dépilant *cmdToRedo* pour empiler sur *cmdToUndo*

et appelle la méthode **Execute**.
Les opérations d'empilement sont effectuées grâce à la méthode
void Empiler(Commande *) qui videra la pile *cmdToRedo* à chaque empilement. Il est à noter que toutes les commandes ne sont pas *undoable* comme par exemple la commande LIST, SAVE et la commande de création d'une sélection. Notre architecture gère un nombre indéterminé de UNDO et REDO contrairement à la limite de 20 qui avait été fixée.

Tous les objets de la classe Commande partagent un **attribut static geoEdit** qui référence la classe principale Modele. Ainsi chaque objet Commande pourra modifier directement le Modele et ils partageront tous le même état de *la base de données.* Parmi les autres principales méthodes fournies par le Modele, on peut citer :

- **void Ajouter(string name, Forme *)**
  Cette méthode reçoit une Forme crée par une Commande et l'ajoute dans la map. Les vérifications sont effectuées par la commande qui s'assure qu'une autre forme ayant le même nom ne soit pas déjà présente à travers une méthode fournie par la classe principale.
- **void Clear()**
  Supprime toutes les formes actuellement présentes dans la map. Mais avant de supprimer les formes, on réalise une copie au préalable. Ce mécanisme permet d'optimiser l'annulation de la commande CLEAR qui n'aura pas à récréer toutes les formes qui étaient déjà présentes dans la map.
- **void CasserLien(const Forme*)**
  Cette méthode, lors d'une commande DELETE, se chargera de signaler à toutes les selections qui contiennent l'objet passé en paramètre qu'elle n'en fait plus partie.

Une autre optimisation a été au niveau de la commande LOAD plus précisément la classe *cmdLoad* qui permet de charger un ensemble de Forme à partir d'un fichier. Cette classe est composée d'autres commandes appelées *cmdAjout* qui se chargeront de vérifier la syntaxe de toutes les lignes du fichier. Après avoir lu un fichier, on garde une indépendance totale vis à vis de celui-ci ce qui permet de pouvoir UNDO et REDO un LOAD sans lire encore une fois le fichier.

2.2 Forme

Cette **classe abstraite** constitue la classe de base dont va hériter toutes les formes géométrique. Elle engendre directement les classe **Cercle**, **Polyligne** et **Selection**. La classe sélection est considérée comme une forme car cela rend plus facile la recherche lors d'un l'appel pouvant concerner une "vraie" forme ou une sélection (MOVE, DELETE...) Les classes **Ligne** et **Rectangle** héritent elles de Polyligne. Toutes ces classes ont en commun les méthodes

- **void Afficher (ostream &)**
  Cette méthode redéfinie l'affichage sur la sortie standard pour chacun des types de forme. Elle ne renvoie rien pour une sélection qui ne doit pas apparaître lors de l'affichage des formes. Elle permet aussi de sauvegarder les Forme dans un fichier en recevant le flux vers celui-ci.
- **bool InclusDans(const & Point, const & Point)**

Cette méthode permet de définir si la forme est contenue entièrement dans le rectangle formé par les deux points passés en paramètre. Comme il n'est pas possible de sélectionner une sélection, la méthode renvoie systématiquement **false** dans le cas où elle est appelée sur une selection qui un *genre particulier* de Forme.

— **void Deplacer (long, long)**
Cette méthode déplace point par point, la forme pour laquelle elle est appelée, ou dans le cas d'une sélection, appelle la méthode Deplacer pour chacune des formes la constituant.

3 Tests

3.1 Présentation

L'application que nous avons réalisé a subi une batterie de Test décomposée en deux catégories :

— Les tests des fonctions "de base" sans conflit possible ou erreur de syntaxe : ces tests modélisent une utilisation sans erreur de l'application. Ils reprennent les fonctions telles qu'ajouter des formes ou des sélections, les déplacer, les supprimer, ou encore tester l'annulation d'une commande, sa reprise, l'enregistrement de la figure courante ainsi que le chargement d'une figure existante.

— Les tests des fonctions "avancées" de l'application : ces tests sont là pour vérifier que le programme ne plante pas en cas d'une utilisation différente de la syntaxe définie par le cahier des charges. Ce sont aussi des tests plus avancés cherchant à vérifier le bon déroulement de l'exécution de l'application même quand un enchaînement de commande est plus compliqué ou piégeux.

Pour la mémoire, on a utilisé *valgrind* pour détecter les fuites de mémoire et l'utilisation de variables non initialisées.

3.2 Exécution

Pour effectuer notre panel de test, nous avons réutilisé un script shell fourni lors d'un précédant TP. Ce script se trouve dans le répertoire ./tests et se nomme mktest.sh. Il exécute à la suite tous les textes définis dans la figure 3 et affiche sur la sortie standard le résultat unitaire de chaque test ainsi qu'un bilan répertoriant le nombre de tests passés, échoués ou mal formés. Lors de cette exécution, tous les tests que nous avons définis se sont exécutés avec succès.

4 Performances

Nous avons aussi testé les performances de notre programme grâce à un script permettant à la fois de générer un fichier de commande qui sera passé en entrée de notre application. Nous appelons ensuite notre application plusieurs fois pour réaliser la moyenne des temps d'ajout nécessaires. Les tests ont été réalisés sur des machines du département, en enlevant les traces de la sortie standard (OK ...) qui ont été pensées en compilation conditionnelle.

n°du test	Description du test
Test des fonctions "de base"	
01	Ajout Multiple et Lister
02	Ajout d'un objet puis suppression
03	Déplacement d'objet
04	Déplacement d'une sélection
05	Ajout de Forme puis UNDO
06	Ajout de Forme puis UNDO puis REDO
07	Suppression d'une sélection
08	SAVE un fichier
09	Test de la commande LOAD pour un fichier existant
10	Test de la comande CLEAR
11	Commande CLEAR puis UNDO
12	Commande LOAD puis UNDO
13	LOAD dans une application sans conflit de nom
14	Enchainement LOAD UNDO REDO
Test des fonctions "avancées"	
15	Ajout de deux Formes ayant le même nom (conflit de nom)
16	LOAD dans une application avec conflit de nom
17	MOVE un objet inexistant
18	MOVE un objet supprimé
19	DELETE un objet inexistant
20	DELETE un objet supprime
21	DELETE selection, UNDO, MOVE sélection
22	DELETE une forme d'une sélection puis UNDO puis MOVE la sélection
23	Appel de UNDO plus de fois qu'il n'y a d'événement
24	Appel de REDO plus de fois qu'il n'y a eu de UNDO
25	Ajout avec mauvais parametres puis UNDO
26	LOAD d'un fichier corrompu
27	Ajout d'une forme ayant le nom d'une forme supprimee puis UNDOx2
28	LOAD d'un fichier contenant des selections et test sur ces selections
29	Enchainement de commandes non définies par le cahier des charges

Figure 3 – Liste des Tests

Le document perfs_B3425.pdf situé dans le répertoire ./doc consigne les résultats des tests effectués.