TP5 – Conception

B3125 Romain BRUNAT & Merlin NIMIER-DAVID

Architecture générale

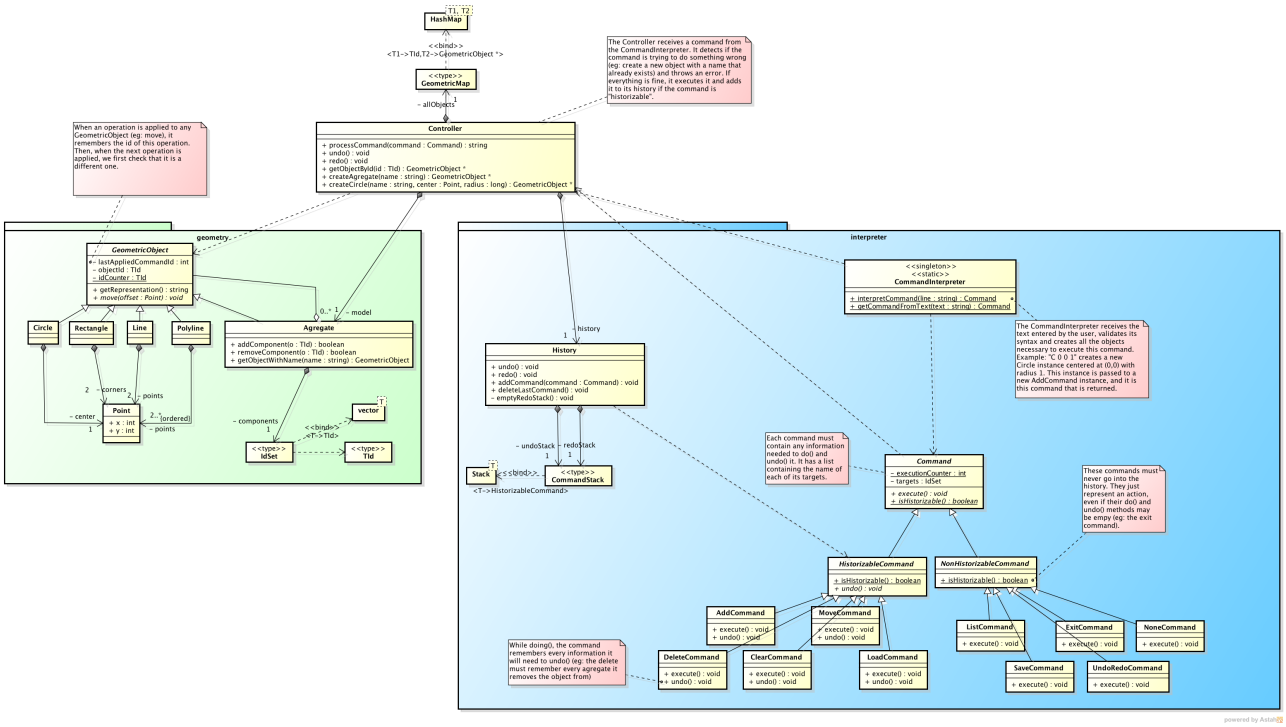

Diagram 1: Voir le fichier `class-diagram.png` *pour une vue détaillée*

L'application s'organise autour de deux « packages » principaux :

- Le package `geometricObjects`, qui comporte toutes les classes représentant un objet géométrique
- Le package `commands`, qui comporte l'ensemble des classes représentant une commande, ainsi que l'interpréteur de commandes et l'historique.

Ces deux grandes parties de l'application sont coordonnées par le contrôleur.

Le contrôleur

La classe `Controller` est la classe centrale de l'application. Comme toutes les opérations passent par cette classe, nous avons utilisé le *design pattern* `Singleton`. Ainsi, la méthode statique `Controller::GetInstance()` permet d'accéder à l'unique instance de `Controller` depuis n'importe quelle classe de l'application, notamment les classe Command.

Le contrôleur est le seul à détenir les instances d'objets géométriques manipulés dans l'application. Un pointeur vers chacune des instances de `GeometricObject` créé lors de la session est conservé dans un dictionnaire. Pour s'assurer qu'aucune instance ne soit créée en dehors du contrôleur, on propose des *méthodes* `Factory` : `Controller::CreateCircle`, `Controller::CreateAgregate`, etc.

Les identifiants

Chaque objet géométrique est désigné par son **identifiant** (de type Tid, défini comme long). Cet identifiant est utilisé par toutes les classes pour faire référence à l'objet. Ainsi, les commandes et les objets agrégés ne détiennent jamais de pointeur vers une instance, mais seulement un set d'identifiants. Pour accéder aux instances effectives, au moment de faire des traitements, on utilise la méthode `Controller::GetObjectById(TId)`.

De plus, on distingue le dictionnaire des instances (qui contient tous les objets géométriques jamais créés) du *document*, qui contient simplement les identifiants des objets actuellement contenus dans le document. Si l'utilisateur utilise une commande `DELETE`, l'identifiant de l'objet correspondant est simplement retiré du document, mais l'instance est conservée. Si l'utilisateur souhaite annuler, on insère à nouveau l'identifiant correspondant. Le document est représenté sous la forme d'un objet géométrique `Agregate`, qui s'avère être tout à fait adapté puisqu'il propose des méthodes d'ajout et de suppression.

Ce choix architectural apporte une sécurité accrue : les instances sont sous la responsabilité du contrôleur, qui les conserve jusqu'à la fermeture de l'application. Cela est particulièrement utile pour le support des opérations `UNDO` et `REDO`. Cependant, on introduit ainsi un coût supplémentaire de recherche à chaque opération.

Utilisation du polymorphisme

L'utilisation du polymorphisme a grandement aidé lors du développement de l'application. Tous les objets géométriques (`Circle`, `Rectangle`, `Agregate`, etc) héritent de la classe abstraite `GeometricObject` et peuvent donc tous être manipulés comme tels. De même, les commands (`AddCommand`, `ExitCommand`, etc) héritent de la classe abstraite `Command`.

Gestion des commands

Pour exécuter les commandes entrées par l'utilisateur, et pour découpler au mieux l'interface utilisateur de la logique métier, on utilise de *design pattern* `Command`. On crée une classe héritant de la classe abstraite Command par action réalisable dans l'application. La méthode `Command::Execute()` de chacune de ces classes effectue l'action, tandis que la méthode `Command::Undo()` est replace le document dans l'état précédent.

Le prompt

Le programme principal fonctionne selon une boucle de type prompt basique :

1. Interpréter la commande depuis l'entrée standard
2. Si une commande valide est interprétée, la passer au contrôleur qui l'exécute.
3. Si la commande est annulable, le contrôleur l'ajoute à l'historique.
4. Si l'utilisateur n'a pas demandé à quitter l'application, goto 1.

Création des commandes

La classe `CommandInterpreter` propose des méthodes statiques permettant d'interpréter des commandes à partir d'un flux d'entrée. Il effectue le maximum de vérifications et provoque une erreur utilisateur si le format d'instruction n'est pas respecté. Si les paramètres donnés sont corrects, une instance de la `Command` correspondante est crée et retournée.

Historique

L'historique est, lui aussi, entièrement détenu par le contrôleur. La classe `History` maintient deux piles d'objets de type `HistorizableCommand`. En effet, on n'accepte pas par exemple les commandes `EXIT`, `LIST`, etc, qui n'ont pas leur place dans l'historique puisqu'elles ne sont pas annulables. Ses méthodes `History::Undo()` et `History::Redo()` lance l'annulation / la ré-exécution et met à jour ces deux piles en conséquences.

Actions propres à chaque commande

Chaque classe enfant de `Command` a son implémentation propre de la méthode `Execute`. Les actions à réaliser sont très variées : ajouter ou retirer un objet du document, lister les objets du document, déplacer les objets, etc. Grâce au pattern `Singleton` utilisé pour le contrôleur, on obtient facilement une visibilité sur le document, puis sur les instances en passant par les méthodes proposées par le contrôleur. Il devient alors facile de manipuler les objets du document.