Arnaud Dupeyrat
Antoine Breton
3IF - TP4 - CDC

SPÉCIFICATION

TABLE DES MATIERES

Arnaud Dupeyrat
Antoine Breton
3IF - TP4 - CDC

I. INTRODUCTION

Le rôle de notre application est de gérer le traitement d'une suite d'actions prédéfinies par l'utilisateur.

Ces actions permettent d'agir sur des figures géométriques (Rectangle, Segment, Polygone convexe). L'utilisateur peut aussi modifier ou vérifier le contenu de notre modèle en accédant a l'ensemble de figures disponibles pour ces futures requêtes.

D'autre part, nous avons voulu que notre application réponde à certains critères fortement incités par le cahier des charges initial. L'objectif principal est d'augmenter l'efficacité et la réutilisabilité de notre application :

Ainsi la conception de nos composants :

- Offre une grande réutilisabilité et évolutivité grâce au polymorphisme et a la mise en place de design pattern)
- Répond à des contraintes de performance importante afin d'être le plus efficace possible.

II. DESCRIPTION GÉNÉRALE ET PRINCIPALES DÉFINITIONS

1) PATTERN DESIGN UTILISÉ

Dans l'objectif de rendre notre application le plus réutilisable possible et de répondre aux demandes du TP nous avons donc utilisé des designs pattern.

Premièrement, nous avons utilisé le pattern **MVC** (Modèle-Vue-Controleur) qui permet une séparation des données (ensemble des figures) et l'interaction avec l'utilisateur. Le lien s'effectue à l'aide d'un contrôleur qui se charge de relier les différents le modèle et la Vue.

Ce choix est principalement motive par le support des futurs évolution du programme. En effet, les interactions utilisateur\applications se faisant à l'aide des lignes de commandes l'intérêt d'un pattern MVC est clairement limité dans le seul cadre du TP.

Nous avons alors essayé de séparer un maximum les interactions entre la **Vue** et le contrôleur permettant une évolution facilitée vers une interface graphique.

Deuxièmement, nous avons utilisé le pattern **composé\composites** qui nous permet de traiter un ensemble de figure de la même manière qu'une figure classique (Rectangle, Segment, Polygone convexe).

Cela permet de faire abstraction du type de l'instance, et ainsi traiter l'ensemble des requêtes de la même manière quelque soit le type de l'objet.

Troisièmement, nous avons utilisé le pattern design **singleton**. Cela nous permet de nous assurer l'unicité du contrôleur partagé entre la vue et le modèle.

L'utilité du design pattern s'explique par le fait que le contrôleur doit nécessairement être unique car celui-ci contient des variables propres qui sont nécessaires au bon fonctionnement du programme.

2) POLYMORPHISME

Pour rendre notre application complètement réutilisable, nous avons employé le **polymorphisme** dans les cas suivants :

- Le traitement des actions descendent d'une classe abstraite mère **IAction** nous permettant de codifié les action REDO et UNDO pour chaque action différentes.

- Le traitement des différentes figures, nous permettant de codifié les actions partagé entre l'ensemble des figures et spécifier les caractéristique propres a chacune.

Ce choix s'est décidé pour deux raisons :

- Premièrement le polymorphisme facilite amplement l'implémentation de notre programme, en effet nous traitons seulement des pointeurs de la classe mère dans l'ensemble de l'application.

- Deuxièmement l'héritage facilite la réutilisabilité de notre application permettant en codifiant les méthodes nécessaires, l'ajout de nouvelle Action ou Figure.

Arnaud Dupeyrat
Antoine Breton
3IF - TP4 - CDC

III. MODÈLE

Le modèle a pour but de stocker l'ensemble des figures qui composent notre application.

Son rôle est aussi de traiter les différentes requêtes du contrôleur qui cherche à modifier le modèle. C'est pour ces raisons que la classe modèle est centrale à notre application :

- Elle est charge de transmettre correctement les informations au controleur à travers différentes méthodes :

  - bool EstDedant(Point p) : permettant de vérifier l'appartenance d'un point à une figure

  - bool CreerComposite (bool intersection,const string nom,vector<string> & ensembleDePoint): permettant la création d'une nouvelle figure

  - bool SupprimerFigures(vector<std::string> & noms): permettant la suppression d'un ensemble de figures

  - bool Deplacer(const string nom, int dx, int dy): permettant de dépasser la figure selon un déplacement horizontale dx et verticale dy

  - bool RestaurerFigures(vector<string> & noms): permettant des restaurer les figures supprimées et ainsi les rendre visible de nouveau pour l'utilisateur.

Le modèle est composé d'autres méthodes (cf diagramme de classe) mais celle-ci étant privé n'interagit pas avec le contrôleur.

3) INTERFACES UTILISÉES POUR LE MODÈLE

INTERFACE IFIGURE

Classe abstraite permettant de codifier certaines méthodes nécessaires au bon fonctionnement de notre application:

- **EstDedans**(Point pointATester) : Permet de tester l'appartenance du point tester à la figure.
- **Deplacer**(double dx, double dy): Permet de déplacer la figure selon un déplacement horizontale dx et verticale dy.
- **Afficher**() : Permet l'affichage de la figure traité.

Dans l'ensemble du programme nous traiterons des pointeurs de type IFigure afin de facilité le traitement des figures.

De plus la classe IFigure possède un paramètre « type » nous permettant de remonter au type de la figure instanciée. De cette manière nous facilitons la sauvegarde et le Load de notre modèle.

Cette string peut contenir plusieurs valeurs en fonction de leur signification :

- R = Objet de classe Rectangle.
- S = Objet de classe Segment.
- PC = Objet de classe Polygone.
- C = Objet de classe Composite.

Interface IComposant

La classe IComposant est une classe fille de Figure.

Dans l'état actuel du programme l'interface IComposant n'apporte pas forcément de fonctionnalité supplémentaire notoire.

La classe Icomposant est abstraite et permet de codifié certaines méthodes nécessaire au bon fonctionnement de notre application, la méthodes suivantes sont virtuelle pure :

- IFigure * Copie(), permettant une copie profonde de la figure « classique » traité.

La classe abstraite permet alors de spécifier un comportement commun à l'ensemble des figures classiques (Rectangle-Segment-Polygone) sans modifier l'architecture profondément.

4) Classes utilisées pour le modèle

Classe Rectangle

Classe fille de IComposant, celle-ci permet de modeliser le comportement d'un rectangle.

Un rectangle est défini par le biais de deux points qui spécifient le point supérieur et le point inférieur du rectangle.

Arnaud Dupeyrat
Antoine Breton
3IF - TP4 - CDC

Classe Segment

Classe fille de IComposant, celle-ci permet de modeliser le comportement d'un Segment.

Un segment est défini par le biais de deux points qui spécifient les deux extrémités du segment.

Classe Polygone

Classe fille de IComposant, celle-ci permet de modeliser le comportement d'un polygone strictement convexe.

La construction d'un polygone s'effectue à partir d'un ensemble de points, ensemble de points dont l'odre du traitement est identique à l'ordre des points placés en paramètres.

5) Classe Vecteur

Classe permettant de modéliser le comportement d'un vecteur.

La construction d'un vecteur s'effectue à partir de deux points.

La classe Vecteur est composé de plusieurs méthodes:

- double **Angle** (Vecteur & vecteurATester) : Permet de retourner l'angle formé par deux vecteurs (Nous

l'avons utilisé dans le test du polygone convexe)

- double **Norme** () : Permet de calculer la norme du vecteur (Nous l'avons utilisé dans la plupart des

calcul d'appartenance).

6) Classe Composite

Classe fille de IFigure, permettant de modéliser une intersection ou une réunion de figure « classique » qui compose notre programme.

Un composite est défini par le biais d'un ensemble de figures à un instant t. La construction d'un nouveau composite effectue une copie profonde des figures placées en paramètres, permettant de conserver les valeurs respectives des figures à l'instant t.

Arnaud Dupeyrat
Antoine Breton
3IF - TP4 - CDC

7) Classe Point

Classe permettant de modeliser le comportement d'un Point.

La classe permet de situer un point dans l'espace à l'aide d'une abscisse et d'un ordonné.

L'ensemble des figures « classiques » repose sur la classe Point, permettant leur construction.

La classe Point posséde plusieurs méthodes afin de répondre entiérement aux demandes du TP :

- bool DeplacerPoint(double differenceX,double differenceY) : Permettant de déplacer le point selon un déplacement horizontale differenceX et un déplacement verticale differenceY.
- void afficherPoint(ostream & fichierSortie) const : Permettant l'affichage de d'un point sous la forme [abscisse, ordonne].

IV. Vue

8) Classe Vue

Classe nous permettant d'interagir avec l'utilisateur.

Nous somme conscient que nous dans le cadre du TP, nous aurions pu nous passer de la classe Vue.

En effet l'ensemble des interactions se déroulant en ligne de commandes, le plus simple aurait été interagir à l'aide du contrôleur. our répondre au problème posé le plus proprement possible, nous avons choisis d'implémenter une classe Vue et une classe Parseur pour traiter les interactions avec l'utilisateur nous permettant de séparer correctement les différents modules en charge des interactions avec l'utilisateur.

Ce choix nous permet d'imaginer des futures évolutions pour notre application :

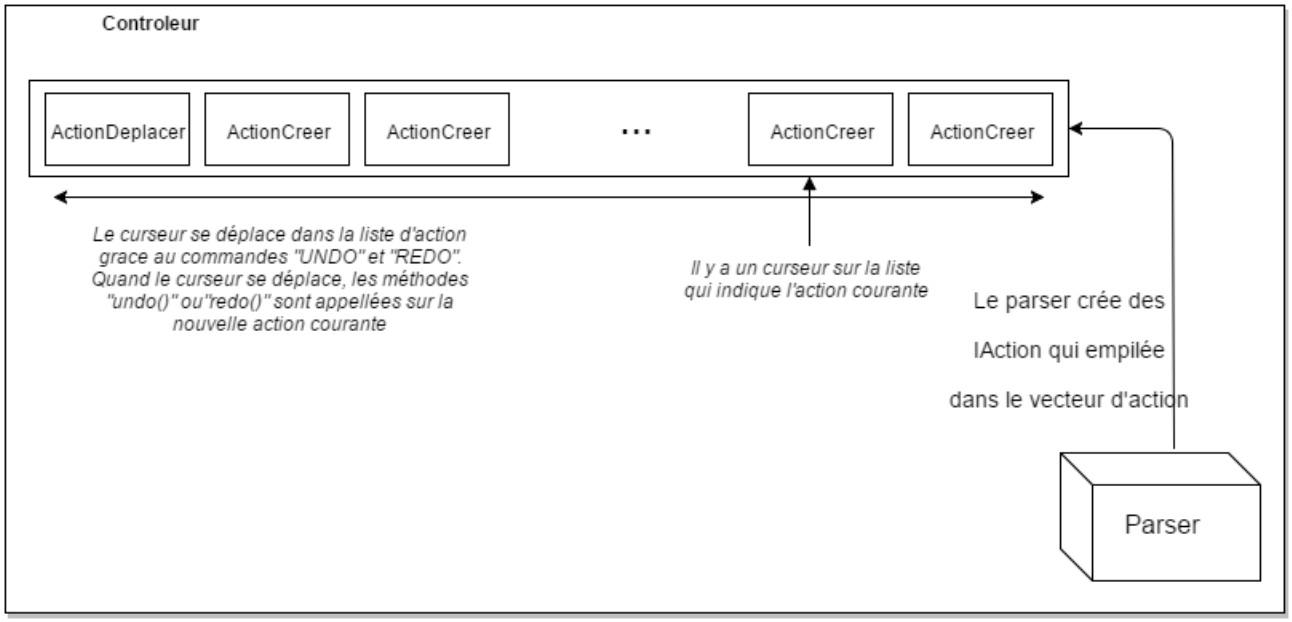
- L'implémentation d'une interface graphique.
- L'ajout de nouvelles commandes traités par le programme.
- permettre l'ajout de nouvelle fonctionnalités offertes par le vue.

Arnaud Dupeyrat
Antoine Breton
3IF - TP4 - CDC

V. CONTROLEUR

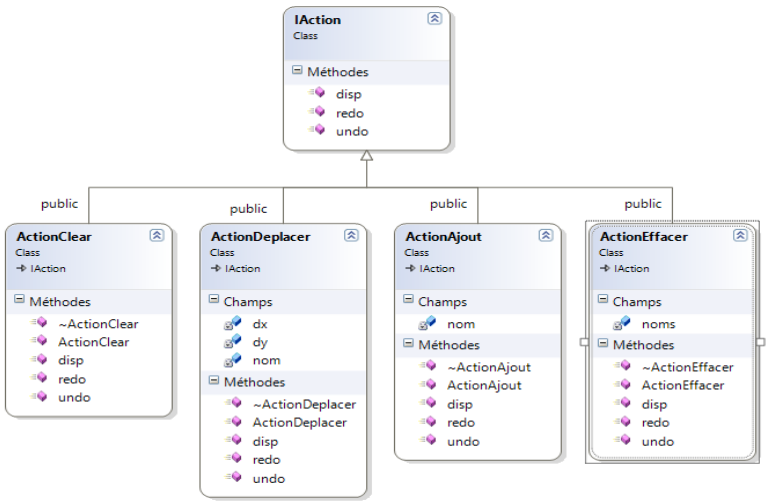
Le contrôleur joue le rôle d'intermédiaire entre la vue et le modèle. Ainsi, il gère les fonctionnalités suivantes :

- contient un parser d'entrée, qui gère la lecture des commandes de l'utilisateur,
- contient un vecteur d'action permettant de gérer les fonctionnalités "undo" et "redo". Une action est représentée par une interface **IAction** qui est implémentée par les différents types d'actions.



9) INTERFACES UTILISÉES POUR LE CONTRÔLEUR

IACTION



Cette interface impose les méthode que chaque actions doit proposer. Chaque action doit proposer une méthode undo() qui défait l'action, et redo() qui refait l'action.

10) Classes utilisées pour le contrôleur

ActionClear:

- undo() : déplace toutes les figures du modèle dans la liste corbeille,
- redo() : déplace toutes les figures du modèle de la corbeille vers le vecteur de figure.

ActionDeplacer:

- undo() : déplace la figure de -dx et -dy,
- redo() : déplace la figure de dx et dy.

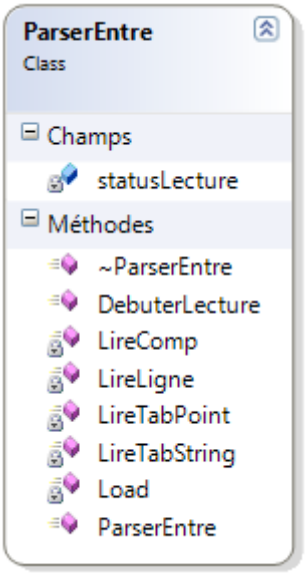
ActionEffacer:

- undo() : restaure la figure supprimée vers le vecteur de figure,
- redo() : déplace la figure vers la corbeille.

ActionAjout:

- undo() : met la figure crée dans la corbeille,
- redo() : déplace la figure depuis la corbeille vers le vecteur de figure.

Le parser



Le parser contient une boucle infinie que analyse la saisie d'un flux d'entrée, par défaut **cin**.

Il lit et gère toutes les commandes définies dans le cahier des charges et tiens un vecteur de commandes entrées. Lorsque l'utilisateur entre la commande SAVE <nomFic>, le parser crée un fichier portant le nom entré en paramètres et y écrit la liste de commandes entrées. Quand l'utilisateur entre la commande LOAD <nomFic> les commandes contenues dans le "nomFic" sont réinjectées dans le parser puis retraitées.