Compte rendu du TP n°4 de C++

Adrien Lepic et Quentin Vecchio (B3323)

INSA Lyon - 3IF

Vendredi 5 Février 2016

I - Spécifications du TP

Notre logiciel permet la modélisation et la gestion de différentes figures, tels que des points, des segments, des rectangles, des polynômes convexes et des ensembles de figure. On peut également interagir avec les figures, comme par exemple les déplacer, les supprimer, les réunir ou encore tester si un point fait partie d'une figure. Le logiciel gère également la sauvegarde et le chargement de figures grâce à des fichiers textes. Les fonctions Undo et Redo ont aussi été implémentées.

II - Structures de données retenues

Nous avons fait le choix d'identifier chaque figure par un nom unique. De ce fait, nous avons pu stocker les figures dans un map avec comme index le nom de la figure. Cette structure permet d'accéder rapidement à un élément, on peut donc facilement gérer de grande quantité de données. Concernant la classe *Polygone*, nous avons choisi de stocker chaque points qui le compose dans un vector.

III - Choix de développement et tests

Afin de développer plus vite mais aussi de ne pas réinventer ce qui existait déjà, nous avons fait le choix d'utiliser des designs pattern comme le pattern *Command* ou le pattern *Composite*. Nous avons également fait le choix d'utiliser le guide de style mis en place par l'INSA afin d'avoir un code propre et cohérent. Nous avons aussi utiliser le framework de test introduit dans le TP3 de C++. Ce framework nous a permit de gérer la non-regression de notre code en mettant en place des tests fonctionnels et unitaires pour chaque classe. Nous avons mis en place des test de rapidité sur de gros volumes de données, vous trouverez les résultats de ces expériences dans le rapport de performance.

IV - Spécifications des classes principales

Vous trouverez un diagramme de classe UML complet à la fin de ce document. Ce diagramme permet d'avoir une vue globale des attributs et méthodes de chaque classe.

A - La classe *Vect*

La classe *Vect* permet de modéliser un vecteur grâce à ses coordonnées x et y. La classe permet certaines opérations, comme l'addition et la soustraction entres vecteurs. On surcharge également l'opérateur d'affichage et d'assignation.

B - La classe *Figure*

La classe *Figure* est une classe abstraite dont hérite toutes les autres classes (*Segment*, *Rectangle*, *Polygone*, …). Elle est constitué d'un nom qui permet de l'identifier. Dans notre logiciel le nom d'une figure est unique. De ce fait il n'existe jamais deux figures avec le même nom. Nous avons également mis en place les fonctions *Print*, *Move*, *Copy* et *IsIn* qui sont des méthodes virtuelles qui seront obligatoirement redéfini dans les classes filles.

C - La classe *Segment*

La classe *Segment* modélise un segment constitué de deux points (classe *Vect*). Comme dit précédemment, la classe Segment implémente les méthodes *Print*, *Move*, *Copy* et *IsIn*. On a aussi surchargé l'opérateur d'affichage et d'affectation.

D - La classe *Rectangle*

La classe *Rectangle* modélise un rectangle avec deux points (classe Vect). Les deux points sont le point en haut à gauche et le point en bas à droite. Comme dit précédemment, la classe *Rectangle* implémente les méthodes *Print*, *Move*, *Copy* et *IsIn*. On a aussi surchargé l'opérateur d'affichage et d'affectation.

E - La classe *Polygone*

La classe *Polygone* modélise un polygone avec un ensemble de points (classe Vect). Une méthode nous permet de tester si le polygone est convexe ou non. Comme dit précédemment, la classe *Polygone* implémente les méthodes *Print*, *Move*, *Copy* et *IsIn*. On a aussi surchargé l'opérateur d'affichage et d'affectation. Le polygone est ajouté dans le dessin seulement si il est convexe.

F - La classe *SetOfFigures*

La classe *SetOfFigures* modélise un ensemble de figures. Cette classe hérite aussi de *Figure*, elle implémente donc aussi les méthodes *Print*, *Move*, *Copy* et *IsIn*. Ces dernières restent virtuelle afin d'être redéfinis dans les classes *Union*, *Intersection* et *Dessin*.

G - La classe *Union*

La classe *Union* modélise un ensemble de figures. Cette classe hérite de *SetOfFigures*, elle implémente donc les méthodes *Print*, *Move*, *Copy* et *IsIn*.

H - La classe *Intersection*

La classe *Intersection* modélise un ensemble de figures. Cette classe hérite de *SetOfFigures*, elle implémente donc les méthodes *Print*, *Move*, *Copy* et *IsIn*.

I - La classe *Dessin*

La classe *Dessin* modélise un ensemble de figures, c'est le conteneur principale du programme. Cette classe hérite de *SetOfFigures*, elle implémente donc les méthodes *Print*, *Move*, *Copy* et *IsIn*. C'est dans cette classe que nous avons mis en place les méthodes *Save* et *Load* qui permettent de sauvegarder ou charger un dessin depuis un fichier texte.

J - La classe *Command*

La classe *Command* modélise une paire de commandes. Pour chaque commande on a son inverse. De ce fait on sait quelle commande appeler pour annuler l'autre commande. Nous sommes basés sur le patron de conception *Command* pour concevoir la classe *Command* et *UndoRedo*.

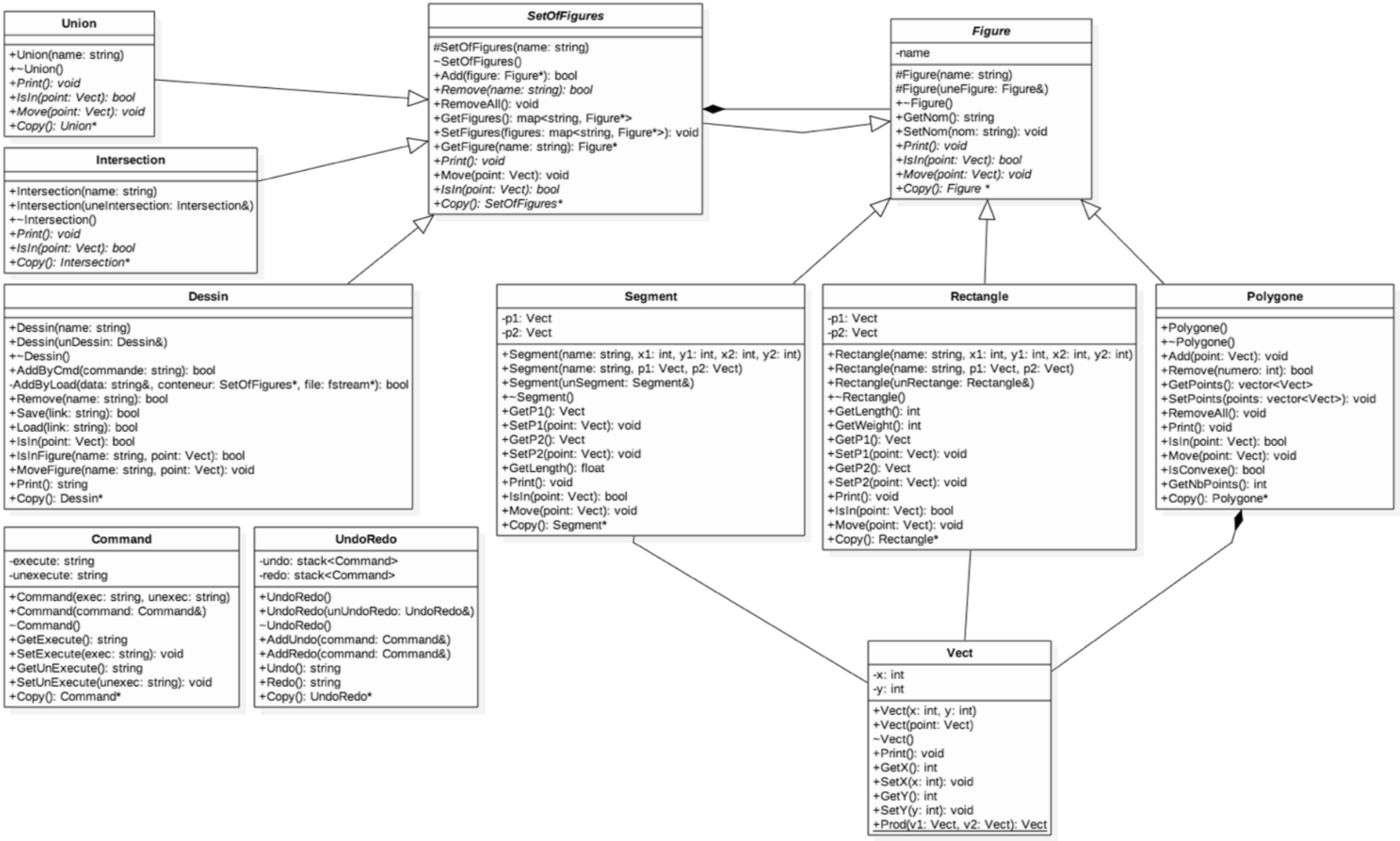
K - La classe *UndoRedo*

La classe *UndoRedo* modélise le système Undo/Redo de notre programme. La classe est constitué de deux piles, une pour le Undo et l'autre pour le Redo. Chaque piles contient des commandes (*Command*) et à chaque fois qu'on appelle *Undo*, on dépile la pile Undo et on réempile la commande dans la pile Redo. Et on fait exactement l'inverse lors de l'appel à *Redo*.

V - Fonctions

Nous avons mis en place dans les fichiers *Fonctions.h* et *Fonctions.cpp*, des fonctionnalités qui nous permettent d'interagir avec l'utilisateur. C'est à travers ces fonctions que l'on va pouvoir interpréter les commandes de l'utilisateur. Nous avons implémenté le système *UndoRedo* directement dans le main, cependant c'est la fonction *InterpretreCommande* qui gérer les undos et redos.

VI - Diagramme de classes


Union
+Union(name: string)
+~Union()
+Print(): void
+IsIn(point: Vect): bool
+Move(point: Vect): void
+Copy(): Union*
SetOfFigures
#SetOfFigures(name: string)
~SetOfFigures()
+Add(figure: Figure*): bool
+Remove(name: string): bool
+RemoveAll(): void
+GetFigures(): map<string, Figure*>
+SetFigures(figures: map<string, Figure*>): void
+GetFigure(name: string): Figure*
+Print(): void
+Move(point: Vect): void
+IsIn(point: Vect): bool
+Copy(): SetOfFigures*
Figure
-name
#Figure(name: string)
#Figure(uneFigure: Figure&)
+~Figure()
+GetNom(): string
+SetNom(nom: string): void
+Print(): void
+IsIn(point: Vect): bool
+Move(point: Vect): void
+Copy(): Figure *
Intersection
+Intersection(name: string)
+Intersection(uneIntersection: Intersection&)
+~Intersection()
+Print(): void
+IsIn(point: Vect): bool
+Copy(): Intersection*
Dessin
+Dessin(name: string)
+Dessin(unDessin: Dessin&)
+~Dessin()
+AddByCmd(commande: string): bool
-AddByLoad(data: string&, conteneur: SetOfFigures*, file: fstream*): bool
+Remove(name: string): bool
+Save(link: string): bool
+Load(link: string): bool
+IsIn(point: Vect): bool
+IsInFigure(name: string, point: Vect): bool
+MoveFigure(name: string, point: Vect): void
+Print(): string
+Copy(): Dessin*
Segment
-p1: Vect
-p2: Vect
+Segment(name: string, x1: int, y1: int, x2: int, y2: int)
+Segment(name: string, p1: Vect, p2: Vect)
+Segment(unSegment: Segment&)
+~Segment()
+GetP1(): Vect
+SetP1(point: Vect): void
+GetP2(): Vect
+SetP2(point: Vect): void
+GetLength(): float
+Print(): void
+IsIn(point: Vect): bool
+Move(point: Vect): void
+Copy(): Segment*
Rectangle
-p1: Vect
-p2: Vect
+Rectangle(name: string, x1: int, y1: int, x2: int, y2: int)
+Rectangle(name: string, p1: Vect, p2: Vect)
+Rectangle(unRectange: Rectangle&)
+~Rectangle()
+GetLength(): int
+GetWeight(): int
+GetP1(): Vect
+SetP1(point: Vect): void
+GetP2(): Vect
+SetP2(point: Vect): void
+Print(): void
+IsIn(point: Vect): bool
+Move(point: Vect): void
+Copy(): Rectangle*
Polygone
+Polygone()
+~Polygone()
+Add(point: Vect): void
+Remove(numero: int): bool
+GetPoints(): vector<Vect>
+SetPoints(points: vector<Vect>): void
+RemoveAll(): void
+Print(): void
+IsIn(point: Vect): bool
+Move(point: Vect): void
+IsConvexe(): bool
+GetNbPoints(): int
+Copy(): Polygone*
Command
-execute: string
-unexecute: string
+Command(exec: string, unexec: string)
+Command(command: Command&)
~Command()
+GetExecute(): string
+SetExecute(exec: string): void
+GetUnExecute(): string
+SetUnExecute(unexec: string): void
+Copy(): Command*
UndoRedo
-undo: stack<Command>
-redo: stack<Command>
+UndoRedo()
+UndoRedo(unUndoRedo: UndoRedo&)
~UndoRedo()
+AddUndo(command: Command&)
+AddRedo(command: Command&)
+Undo(): string
+Redo(): string
+Copy(): UndoRedo*
Vect
-x: int
-y: int
+Vect(x: int, y: int)
+Vect(point: Vect)
~Vect()
+Print(): void
+GetX(): int
+SetX(x: int): void
+GetY(): int
+SetY(y: int): void
+Prod(v1: Vect, v2: Vect): Vect