Rapport Projet C++/JAVA

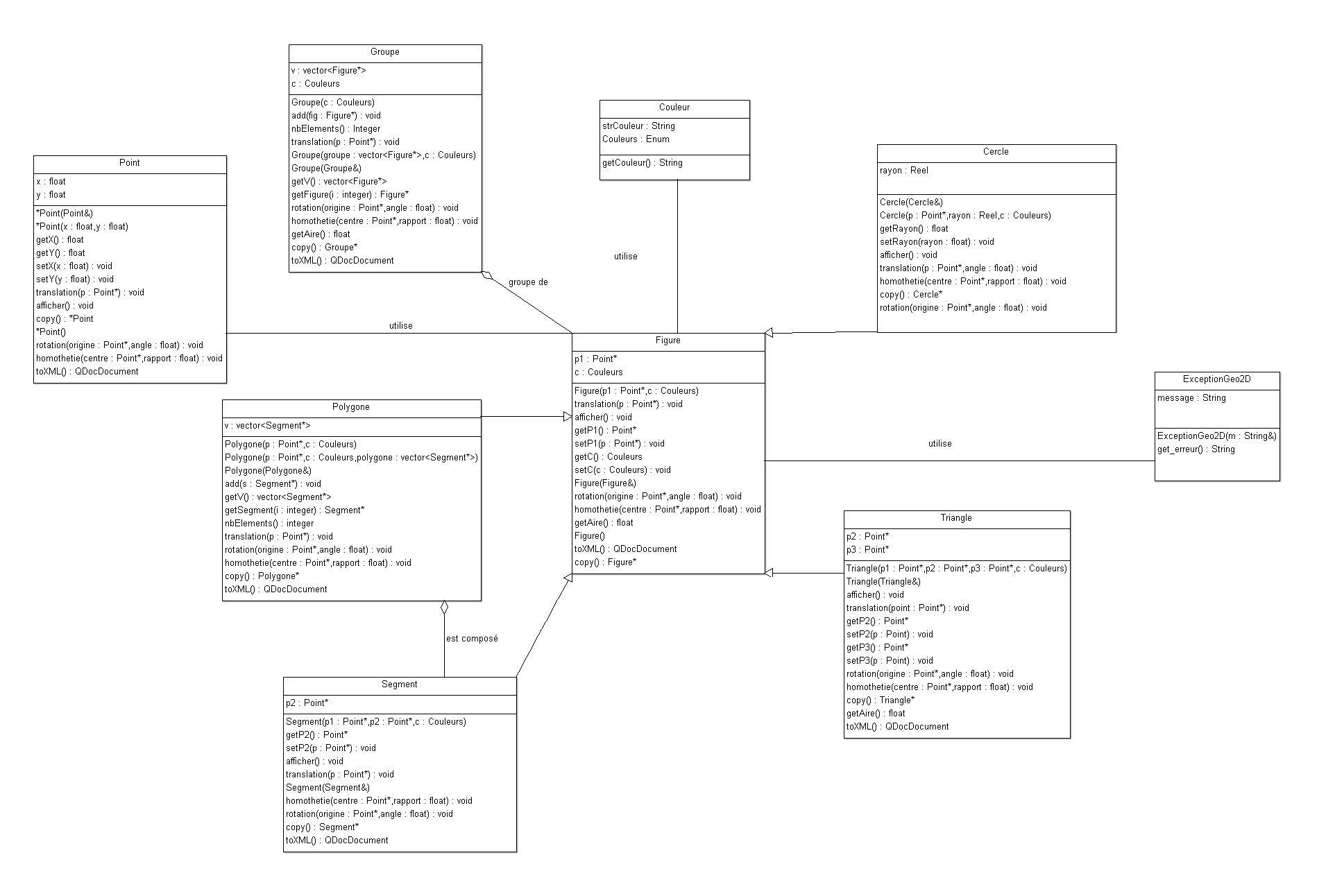
UFR MIM METZ

# Rapport présenté par Koby Dylan & Vecchio Quentin

# Analyse du sujet

## Diagramme de classes UML

Après analyse du sujet et de ses objectifs, nous avons établis le diagramme des classes ci-dessus en langage UML, que nous allons maintenant vous décrire en détail :



* Figure : cette classe est la classe mère de toutes, elle constitue le point de départ des créations de figures. Elle comprend donc un point (la classe Point sera décrite juste après celle-ci), ainsi qu’une couleur (classe également décrite plus tard).

Elle comprend trois constructeurs, le premier qui est un constructeur par défaut, le second avec le dit « point de départ » et une couleur, et le dernier qui est un constructeur par copie de figure.

Il a également été mis en place les fonctions de transformations (translation, rotation et homothétie) mais qui n’ont pas été développées dans cette classe mais dans ses classes filles ; pour en assurer une bonne gestion, des getters et des setters pour le point et la couleur ont été implémentés. Il ne reste qu’une simple fonction d’affichage de la figure qui n’a pas été implémentée non plus ici mais dans les classes filles. De plus, une fonction de clonage nommée « copy() » a été mise en place afin de cloner (copie virtuelle) un objet en une figure, elle est également présente dans toutes les classes filles.

* Point : cette classe est utilisée par figure et est donc présente dans toutes les classes de l’application, elle est simplement constituée de deux nombres réels constituant les coordonnées sur un plan tels que P,x,y ➔ P(x,y).

Comme pour toutes autres classes, des getters et setters présents afin d’ajuster les coordonnées d’un point à tout moment, une fonction d’affichage d’un point, à nouveau les fonctionnalités de transformation sauf l’homothétie d’un point qui n’a pas été jugée utile dans cette classe. Concernant les constructeurs : un par défaut, un autre par copie et un troisième avec les deux coordonnées.

* Couleur : une classe également utilisée par Figure et par conséquent présente dans toute la hiérarchie, elle contient simplement une énumération de couleurs et une fonction permettant d’y accéder.
* Groupe : cette classe est une composition de figures, définies dans le code par un objet vector. Elle comprend également trois constructeurs (un par défaut, un par copie et un dernier avec un groupe passé en paramètres. Les fonctions de transformations sont bien entendus présentes, ainsi que les fonctionnalités d’ajout d’une nouvelle figure, du calcul de nombre de figures dans le groupe, de getters permettant de récupérer une figure à un certain index et une autre pour récupérer le vector, ainsi qu’une fonction de clonage. Cette composition est symbolisée par un losange vide car lorsqu’on supprime un groupe, on ne supprime pas les figures qui y ont été insérées.
* Polygone : cette classe pourrait être renommée multi-segment car en effet, un polygone est une composition de segments. Il en découle trois constructeurs (un par copie, un autre avec un point et une couleur comme la classe Figure, et un dernier avec un point, une couleur et un vector de segments), une fonction d’affichage, des getters afin de récupérer un segment à tel index, l’objet vector lui-même et des fonctions de manipulation du vector (ajout, nombre d’éléments), de clonage, et enfin les fonctions de transformations. Cette composition est symbolisée par un losange vide car lorsqu’on supprime un polygone, on ne supprime pas les segments qui y ont été insérés.
* Triangle : cette classe se compose de trois points. Elle comporte deux constructeurs (par copie, avec trois points et une couleur), les fonctions de transformations, d’affichage, un getter et un setter pour le second point (le premier étant hérité de Figure) et une fonction de clonage.
* Segment : cette classe se compose de deux points. Elle comporte deux constructeurs (par copie, avec deux points et une couleur), les fonctions de transformations, d’affichage, un getter et un setter pour le second point (le premier étant hérité de Figure) et une fonction de clonage.
* Cercle : cette classe est composée du centre qui est le point hérité de Figure, et d’un rayon qui est un réel. Elle comporte un constructeur par copie et un prenant le centre, le rayon et une couleur. Un getter et un setter permettant de récupérer et d’ajuster le rayon du cercle, les fonctions de transformations, d’affichage et de clonage sont bien entendues présentes.
* Exception : cette classe a été construite dans le but de gérer les exceptions pour chaque objets crées. Elle a un seul paramètre qui est une chaîne de caractère correspondant au message d’erreur de l’exception. Il n’y a qu’un constructeur par copie ainsi qu’une fonction permettant de récupérer le message d’erreur.

## Détails de certaines fonctionnalités

Pour une meilleure manière d’accès, les objets vector ne contiennent pas des figures ou des segments mais des pointeurs sur ces derniers.

Tous nos objets sont manipulés via des pointeurs pour une plus nette rapidité d’exécution et de gestion.

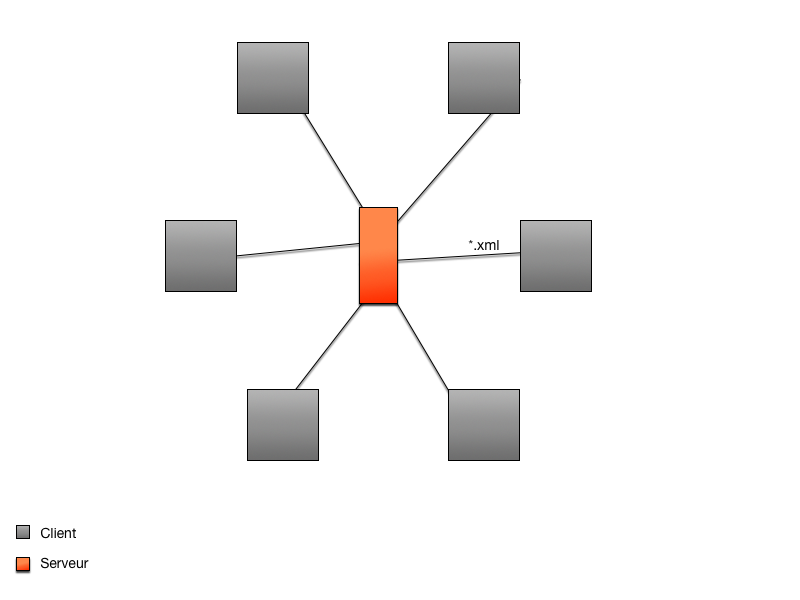
Cela n’a pas été détaillé dans toutes ces classes, mais une surcharge de l’opérateur « << » a été implémentée afin de faciliter l’affichage des objets. Comme autre surcharge d’opérateur, celle du « == » afin de pouvoir comparer directement des objets (notamment des points).

Comme demandée dans le sujet, une fonction de conversion nommée « toXML() » a été mise en place afin de créer un flux XML contenant nos objets qui sera envoyé au serveur JAVA.

Nous avons également instaurés la fonction « getAire() » qui, comme son nom l’indique, va calculer l’aire de la figure, c’est-à-dire ici l’aire du cercle, du triangle et d’un groupe.

Voici la description des fonctions de transformations qui sont :

* Translation : cette fonction prend en paramètre un objet Point, puis on ajoute simplement les coordonnées de ce point aux coordonnées de la figure, ce qui va entraîner un déplacement de cette dernière.
* Rotation : cette fonction prend en paramètre un objet Point qui sera le point d’origine et bien entendu un angle saisi en radian.
* Homothétie : cette fonction prend en paramètre un objet Point qui sera le centre d’homothétie, puis un réel pour le rapport d’homothétie.

2. Réseau

A. Principe

1. Protocole

Nous avons choisi d’utiliser l’XML pour formater nos données lors de l’envoie vers le serveur JAVA, mais également pour la sauvegarde des figures. L’XML a l’avantage d’être simple à comprendre et simple à prendre en main. Il sera donc facile d’éditer un fichier de sauvegarde et de rajouter à la main des figures. De nombreux logiciels de CAO utilisent ce format de données pour sauvegarder ou communiquer. De plus côté C++ (avec la bibliothèque Qt) et côté JAVA, de nombreux frameworks existent pour pouvoir manipuler des données XML facilement.

1. Bibliothèque réseau

Il nous était demandé d’utiliser la librairie réseau Winsock. Pour des raisons de portabilité entres les différents OS, nous avons choisi d’utiliser comme bibliothèque réseau celle de Qt. Elle se veut portable selon les OS et elle est facile à prendre en main.

3. Designs patterns

1. Chain Of Responsibility

Le design pattern Chain Of Responsibility (COR) a été mis en place aussi bien côté client que côté serveur. En effet, nous avons laissé la possibilité au client de pouvoir sauvegarder une figure, il était donc logique qu’il puisse également la rouvrir. Nous avons donc en entrée des données XML. Le programme va donc parcourir tout le document XML, et pour chaque nœud, il appellera un groupe d’experts, appelé ici constructeurs. Chaque constructeurs ira analyser le nœud (chacun leur tours) et le traitera si il le peut. La création de la solution dans ce cas est la création d’un objet et l’ajout dans l’objet Groupe. Côté serveur, le principe est exactement le même, on reçoit un flux XML et les constructeurs vont chacun leur tours essayer de résoudre le problème. Ici la solution se ramène à dessiner sur une fenêtre.

1. Singleton

Le design pattern Singleton a été mis en place côté client. Il se traduit par l’initialisation unique de la bibliothèque réseau. Nous avons créé un objet indépendant, Network, qui va gérer toute la partie réseau cliente. Une méthode permet d’envoyer des données à un serveur. L’initialisation se fait une seule fois, au moment de la création de l’objet. L’objet n’a besoin que d’une adresse IP et un port pour fonctionner.

1. Visitor

Le design pattern Visitor fut mis en place côté client afin qu’un groupe de figures puisse lui-même être contenu dans un groupe et ainsi de suite, on dit alors que « groupe hérite de figure et en est composé », or comme un groupe est une figure et est composé de figures, il peut contenir d’autres groupes. Tout ceci peut se modéliser via un arbre, la racine serait le groupe premier, les nœuds seraient les groupes qu’il contient, et les feuilles seraient les figures présentent dans les groupes.

A la page suivante de ce rapport vous trouverez un arbre illustrant un possible exemple de groupe.

Triangle

T1

Triangle

T2

4. Spécifications

Pour mieux ordonner notre projet, nous avons choisi de développer en suivant le modèle MVC, aussi bien côté serveur que côté client. Nous avons essayé de respecter au mieux ce modèle en séparant correctement les parties données et vues. Des contrôleurs ont été créés afin de pouvoir faire le lien entre les vues et les données.

Pour travailler dans de bonnes conditions, nous avons travaillé avec les outils Bitbucket et Mercurial, qui sont des gestionnaires de versions. Ils permettent de travailler sur un projet à plusieurs de façon optimisée et partagée.