Rapport premiers pas vers l’ingénierie du logiciel

Table des matières

[I/ UML 1](#_Toc66619110)

[II/ Les formes géométriques 2](#_Toc66619111)

[1/ Vecteur 2](#_Toc66619112)

[2/ Forme 3](#_Toc66619113)

[3/ Forme particulière 3](#_Toc66619114)

[4/ Groupe de formes 4](#_Toc66619115)

[III/ Les transformations géométriques 4](#_Toc66619116)

[IV/ Dessiner une forme 5](#_Toc66619117)

[1/ Client TCP/IP 5](#_Toc66619118)

[2/ Librairie réseau C++ 5](#_Toc66619119)

[3/ Protocole 6](#_Toc66619120)

[4/ Dessin d’un groupe 7](#_Toc66619121)

[V/ Sauvegarde/Chargement sur un fichier disque 8](#_Toc66619122)

[1/ La sauvegarde 8](#_Toc66619123)

[2/ La lecture 8](#_Toc66619124)

[VI/ Fonctionnalités diverses 9](#_Toc66619125)

[VII/ Difficultés rencontrées 9](#_Toc66619126)

[VIII/ Eléments à améliorer 9](#_Toc66619127)

[IX/ Pédagogie de l’application 10](#_Toc66619128)

# I/ UML

Nous avons réalisé un UML, celui-ci est disponible dans son intégralité sur le site visual paradigm online. Nous l’avons découpé en plusieurs partie, tout se trouve dans uml/uml\_split/

Il suffit d’importer les fichiers. Vous devez vous rendre sur le site https://online.visual-paradigm.com/fr/

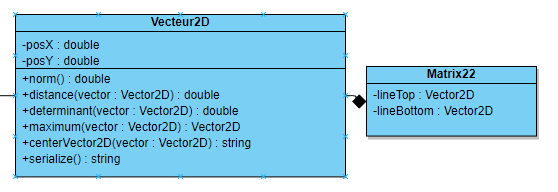
Crée un compte puis rendez-vous « Mon espace de travail » en haut à droite.

Sélectionné My Documents -> from Devise

Vous pouvez également choisir de voir

# II/ Les formes géométriques

## 1/ Vecteur



On va commencer par la classe vecteur2D, celle-ci va être indispensable tout le long du projet. Elle nous permettra d’effectuer des calculs géométriques beaucoup plus simplement.

Elle a deux attributs, qui sont les coordonnées d’un vecteur en deux dimensions.

Beaucoup de surcharge d’opérateur sont implémentées afin d’être le plus efficace et rapide possible (+, -, \*, +=, etc).

Tous les modules seront expliqués au fur et à mesure. La classe Matrix22 sera aussi expliqué plus tard.

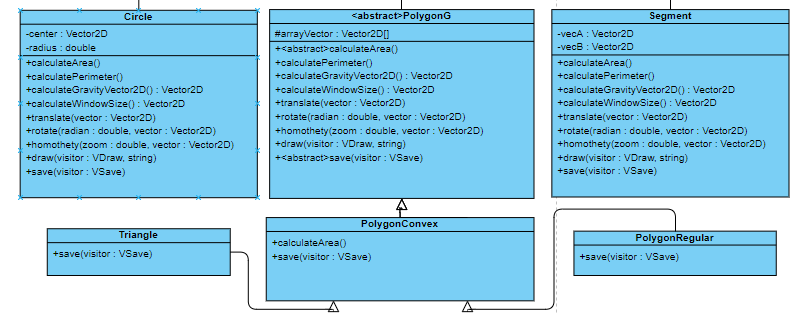
## 2/ Forme

On continue avec la classe abstraite Form qui est la classe principale. Celle-ci permettra de créer n’importe quelles formes géométriques en 2D. Elle est composée de la classe Vecteur vu précédemment et de Color qui comme son nom l’indique gère la couleur de la forme.

On a choisi de créer la classe Color afin de séparer les codes couleurs statiques. Form étant le noyau de l’application, il est préférable de pouvoir ajouter de nouvelle couleur facilement.

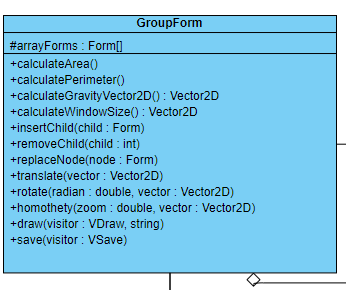
La classe Color permet également de prendre en paramètre n’importe quel code couleur en hexadécimal sous forme de String.

## 3/ Forme particulière



Les formes particulières dérivent toutes de la classe Form. On a créé une autre classe abstraite PolygonG qui dérive de Form. Cela permet de centraliser toutes les méthodes communes dedans, quasiment tous les calculs sont faits par cette classe. À côté on a les classes Circle et Segment qui sont des cas à part et qui nécessité d’être géré individuellement.

## 4/ Groupe de formes



La dernière classe qui dérive de Form est GroupForm. Un GroupFom a un ou plusieurs fils qui sont des formes et à un père qui est une forme.

GroupForm représente un arbre, on peut ajouter ou retirer des fils, toutes les formes sont des pointeurs afin de faciliter la mise en œuvre de la classe.

# III/ Les transformations géométriques

Il y’a 3 types de transformations :

* La translation qui consiste à déplacer tous les vecteurs qui composent la forme. Par exemple, pour faire une translation d’un segment, il suffit d’additionner chacun des points du segment au vecteur de translation.
* La rotation qui consiste à effectuer la rotation d’une forme avec un radiant et un vecteur de rotation. Là nous avons créé la classe Matrix22 pour éviter d’avoir des calculs trop complexes à lire et permettant d’utiliser les surcharges d’opérations des vecteurs.
* L’homothétie qui permet de zoomer ou de dézoomer une forme à partir d’un coefficient de zoom et d’un vecteur de zoom.

Pour les transformations géométriques, on a deux types de transformations. La première consiste à transformer la forme actuelle sans valeur de retour, cette méthode est abstraite dans Form et est redéfini dans les classes qui en dérivent. La deuxième méthode retourne un clone de la forme qui est transformé, Elle consiste à cloner la forme actuelle et à effectuer la première méthode de transformations dessus. On a implémenté ces deux méthodes afin de pouvoir choisir si la forme actuelle veut être modifié ou non.

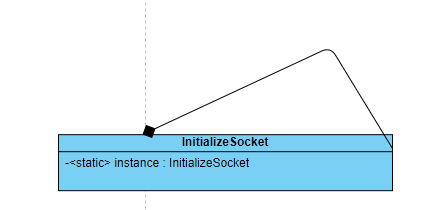
# IV/ Dessiner une forme

## 1/ Client TCP/IP

Pour dessiner nos formes, nous utilisons javaFX qui est une bibliothèque de java. Pour ce faire nous devons envoyer les informations des formes en C++ via une méthode d’envoie de donnée. On va avoir le client qui souhaite dessiner ces formes qui va transmettre des instructions au serveur java.

## 2/ Librairie réseau C++

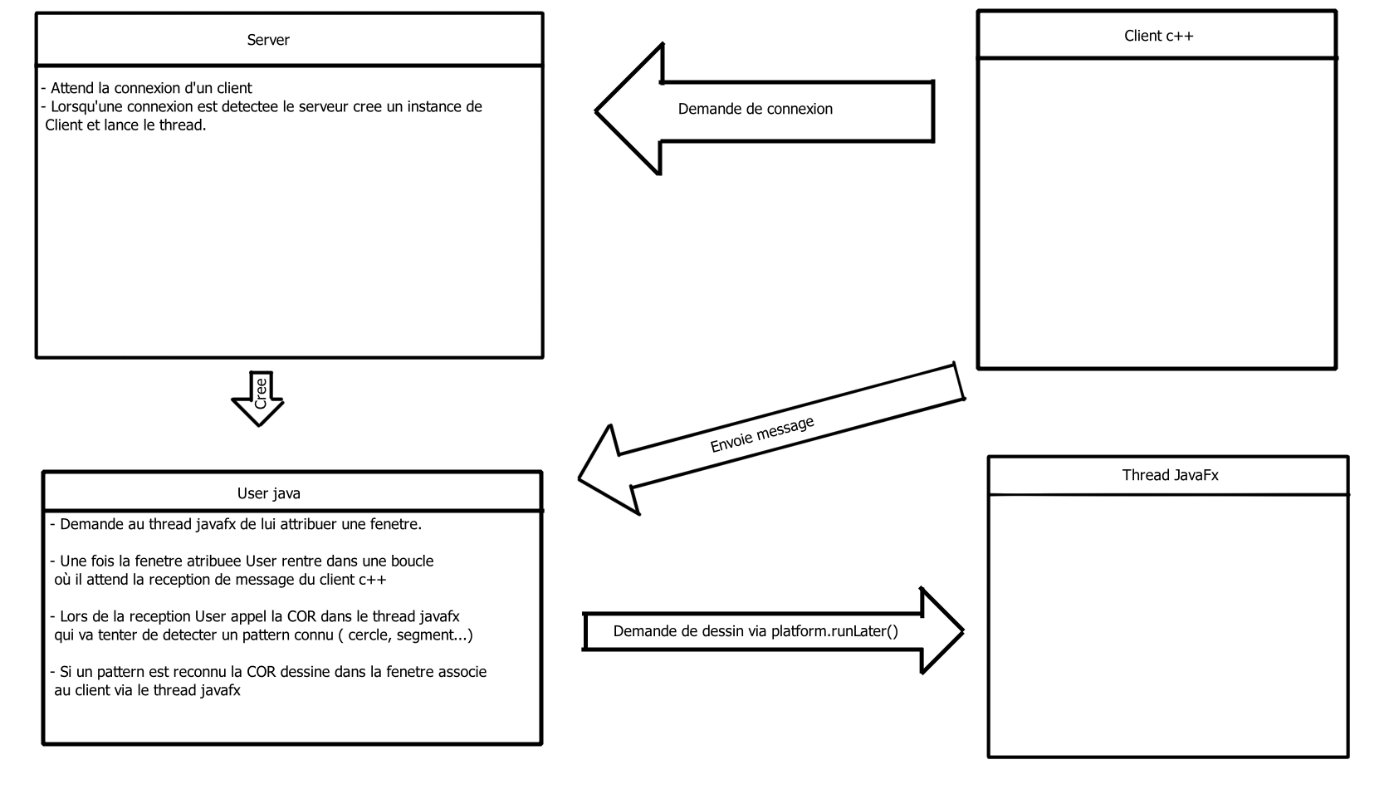
On a utilisé la bibliothèque Winsock qui permet se connecter au serveur java et d’envoyer les instructions.



On a implémenté un singleton qui permet d’initialiser la bibliothèque Winsock, elle doit être utiliser une seule fois.

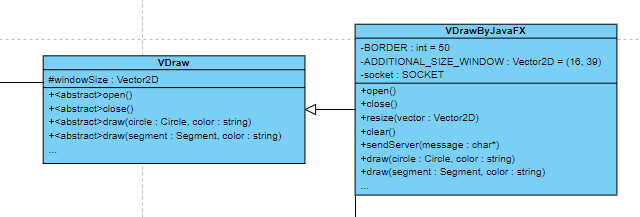
Il faut ensuite établir la connexion au serveur, c’est là qu’intervient le design pattern visitor qui permettra de définir de nouveau outil de dessin sans avoir connaissance du cœur de l’application.

## 3/ Protocole



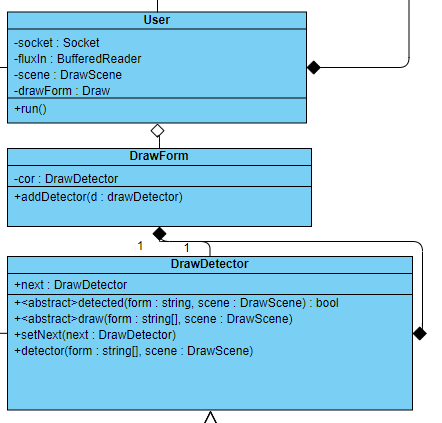
Lors du lancement de l’application java, le serveur qui est un singleton s’instancie. Dans le main le thread du serveur est lancé puis le thread JavaFX est démarré. Le serveur attend la connexion d’un client. Une fois qu’une connexion est demandée le serveur crée une instance d’une classe User qui demande au thread JavaFX de lui attribuer une fenêtre de dessein puis attend que le client C++ envoie une demande de dessein.

Lorsqu’un message est reçu, User récupère le message et tente de le traiter de façon à afficher la forme demandée dans la fenêtre qui lui est attribué.



En ce qui concerne le coté client, le visitor permet de redéfinir l’envoie des formes en fonction de l’outil de dessin. Pour javaFX on doit définir un vecteur additionnel qui est fixe et qui s’ajoutera à la taille de la fenêtre sinon la forme dessinée dépassera un peu du cadre.

## 4/ Dessin d’un groupe

Lorsque le client décide de dessiner un groupe de forme, il va envoyer successivement au serveur les différentes formes composant ce groupe. Chaque message sera récupéré par la classe User et sera envoyé dans la chaîne de responsabilité qui traitera le message et déduira la forme ainsi que les informations qui lui sont associées.

Une fois la forme ou la commande détectée, elle est traitée (affichage d’une forme, actualisation de la fenêtre …). Finalement lorsque le client a dessiné il peut à nouveau demander de dessiner des formes ou il peut fermer la connexion ce qui entraînera la fermeture de la fenêtre.

La commande de Resize est utilisée pour faire en sorte que la fenêtre s’adapte et affiche toutes les formes qui lui sont envoyé.

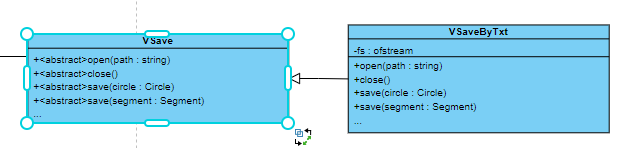
La commande de Clear est utilisée pour nettoyer la fenêtre. Elle permet de créer des animations ou bien de nettoyer la fenêtre avant d’envoyer de nouvelles formes.

Lorsqu’on envoie la demande de dessin une couleur est associe au message. Cette couleur est celle du groupe de forme au quelle appartient la forme envoyée. Puisque les formes sont envoyées individuellement, JavaFX n’a pas besoin d’avoir un expert gérant les groupes de formes. Ainsi pour vérifier que la couleur du groupe de forme est bien respectée il faut envoyer l’information au serveur.

# 

# V/ Sauvegarde/Chargement sur un fichier disque

## 1/ La sauvegarde



Pour la sauvegarde on a de nouveau utilisé le design pattern visitor, cela permettra de sauvegarder sur d’autre type de fichier en redéfinissant l’écriture en fonction de l’extension du fichier.

Pour sauvegarder les formes dans un fichier texte, on a décidé d’écrire une forme par ligne. Pour les groupes de formes, ils sont sauvegardés dans un bloc de ligne délimité par des séparateurs.

## 2/ La lecture

La lecture est un peu plus complexe, pour lire un fichier texte, nous devons lire ligne par ligne afin de récupérer chaque forme jusqu’à la fin du fichier. Le problème c’est que on ne connait pas par avance le type de forme de la ligne. Il faut qu’on puisse déterminer quel type de forme on est en train de lire c’est pour ça qu’on a utilisé le design pattern chain of responsability.

Toutes les lignes dans le fichier sont construite de la manière suivante :

Nom de la forme | X Vecteurs | Paramètre individuel | Couleur

Nous avons implémenté quatre fonctions permettant de récupérer facilement les données de chaque ligne. Pour chaque détection de forme on récupère en priorité le nom de la forme puis on vérifie s’il s’agit bien de la forme voulue.

La difficulté est dans la lecture des groupes de forme. Il faut qu’on appelle récursivement la fonction de lecture pour ajouter les futurs formes dans le groupe.

Le problème c’est que le groupe de forme à besoin d’un autre paramètre qui est le fichier de lecture afin de pouvoir synchroniser le même curseur sinon on ne peut pas lire le fichier (boucle infini). On a opté pour la solution de la variable de fichier en statique car cela permet d’éviter le paramètre supplémentaire.

Notre fonction de lecture retourne un tableau(vector) de forme.

# VI/ Fonctionnalités diverses

L’application peut également calculer l’aire et le périmètre de n’importe quelle forme.

Une conversion en string existe pour chacune des formes.

On peut redimensionner la taille de la fenêtre en fonction de la taille du dessin.

# VII/ Difficultés rencontrées

L’une des difficultés majeures de l’application était de faire fonctionner javaFX en multithreading.

Le problème étant que JavaFX fonctionne sur un seul Thread et que les méthodes de javaFX ne peuvent être appelées uniquement au sein de son thread.

Une solution est donnée par JavaFX : Platform.runLater(runnable)

Cette méthode permet d'ajouter des bouts de code au fil d'exécution du thread javaFX.

Seulement nous ne savons pas quand seront exécutés ces instructions ce qui peut mener à la création d'erreur. Pour résoudre ce dernier problème nous avons décidé d'utilise notifyAll() et wait() pour bloquer notre thread User tant que des instructions vitales à son fonctionnement n'étaient pas exécutées par le thread javaFX.

Le second problème était la lecture du fichier texte. Trouver une solution optimale pour le problème évoqué précédemment nous a pris beaucoup de temps. Finalement on s’est contenté d’avoir une fonctionnalité qui marche.

Ce n’est pas vraiment une difficulté mais au début on n’utilisait pas les const ce qui nous a fait perdre beaucoup de temps dans la réécriture.

# VIII/ Eléments à améliorer

En termes de temps de développement, on a perdu beaucoup de temps à réécrire des fonctions, des classes, … On changeait souvent d’avis sur la façon pour implémenter l’application.

Actuellement, la classe GroupForm pourrait être un peu améliorée, on pourrait rajouter plus de fonctions permettant la maniabilité des arbres.

On pourrait également ajouter de nouvelles formes.

Une interface graphique via terminal ou encore même directement intégré dans l’application pourrait être un élément important à développer.

L’organisation des fichiers du projet serait améliorable également.

# IX/ Pédagogie de l’application

On a appris beaucoup de choses lors du développement de cette application :

* Les design pattern sont vraiment intéressants. Que ce soit la cor, le visitor ou le singleton, ils permettent d’avoir une application efficace et lisible.
* La notion d’héritage était un peu floue auparavant en C++, elle ne l’ai plus maintenant.
* Le principe des const était également flou avant, maintenant c’est parfaitement acquis.
* On a découvert comment faire un code source de qualité, quand bien même qu’il reste toujours améliorable.
* La notion TCP/IP était intéressante à mettre en place