Projet de synthèse
Premier Pas vers l'Ingénierie du Logiciel

Présentation du projet :

Ce projet de synthèse a consisté à créer un logiciel capable de gèrer des formes géométriques en 2D. Plus précisément, nous pouvons créer des formes, les manipuler et les afficher à l'aide d'une bibliothèque graphique de notre choix.

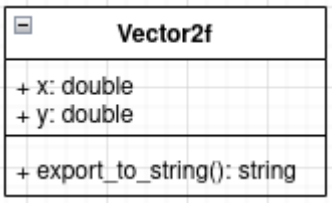
Pour le moment nous avons choisi d'implémenter les formes suivantes : segment, triangle, cercle et polygone, mais rien ne nous empêchera d'en rajouter dans le futur. Nous pouvont aussi manipuler des groupes de formes, et leur attribuer une couleur qui sera utilisée lors de l'affichage.

Ce logiciel comprend aussi un système de sauvegarde et de chargement des différentes formes.

Gestion des formes :

- Gestion des vecteurs :

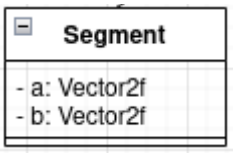
Comme une forme geométrique 2D n'est rien d'autre qu'une succession de points sur un plan, nous avons décidé de créer une classe permettant de créer et manipuler ces points (pouvant également être appelé vecteurs), pour faciliter la compréhension de notre code, nous avons donc implémenté la classe suivante :



Un vecteur 2D est composé de deux coordonnées permettant de le placer sur un plan (x, y).
Nous avons choisi des coordonnées réelles pour améliorer la précision de nos formes.

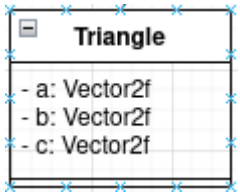
Maintenant que l'on peut représenter des points dans notre code, nous allons pouvoir les utiliser pour créer nos formes.

- Gestion des segments :



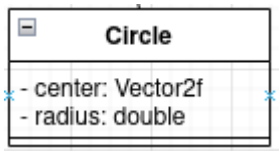
Un segment est une droite qui part d'un point et s'arrête à un autre.
Il nous suffit alors de stocker deux vecteurs A et B qui représenteront ces points.

- Gestion des triangles :



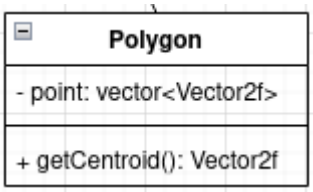
Un triangle est composé de trois sommets A, B et C. Comme pour un segment, nous pouvont représenter ces points par des vecteurs dans notre code.

- Gestion des cercles :



Un cercle est très simple simple à retranscrire. Il nous faut juste son centre, qui encore une fois sera représenté par un vecteur, et son rayon. Le rayon est stocké en tant que réel pour être avoir plus de précision.

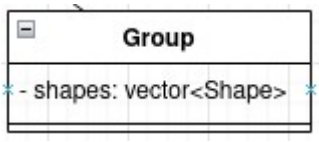
- Gestion des polygones :



Un polygone peut sembler très complexe à modéliser à première vue, mais en pratique il faut simplement connaitre tous ses sommets et leur ordre pour pouvoir retracer la figure complète.
C'est pour cela que nous avons un tableau de vecteurs qui sert à stocker tous ces points dans l'ordre que l'on a choisi.

- Gestion des groupes de forme :

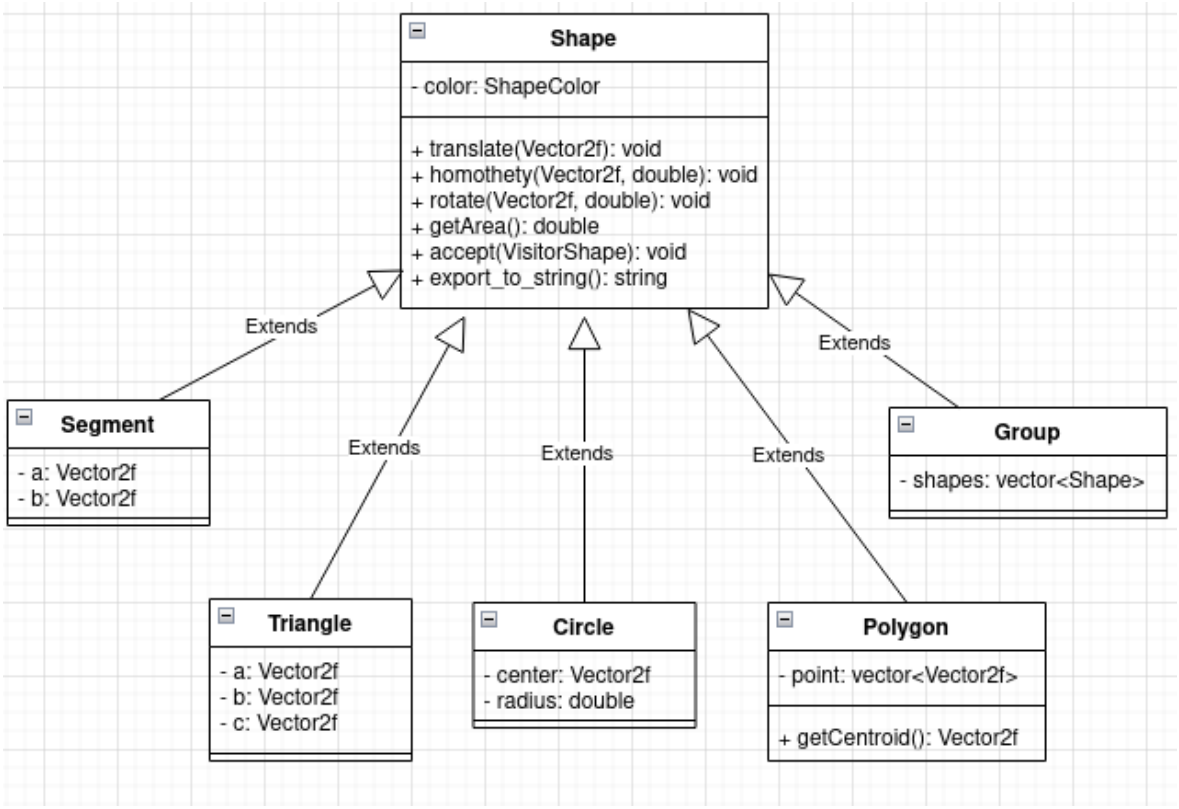
Un groupe de forme peut être vu comme une forme complexe que l'on peut découper en plusieurs formes plus simples.
L'avantage d'avoir un groupe se montre lors de la manipulation d'un grand nombre de formes, il devient alors plus simple de les regrouper pour manipuler qu'un seul bloc de données.



Il nous faut donc un tableau stockant toutes les formes dont on a besoin.

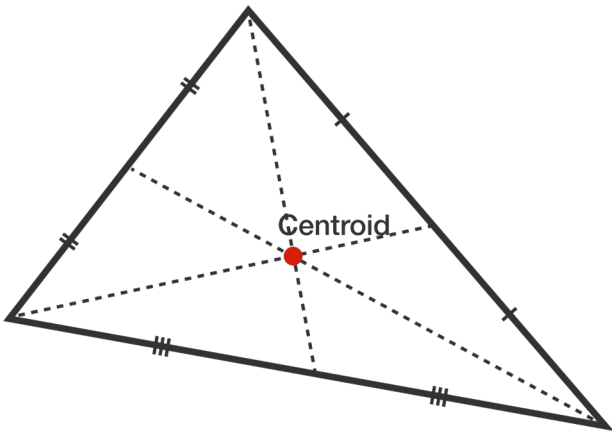
Finalement nous nous retrouvons avec ces différentes classes qui ont le point commun de toutes représenter des formes, de plus elles doivent posséder une couleur qui nous servira lors de l'affichage.
Il parraît alors logique de vouloir les lier entre elles, et pour cela nous avons donc utilisé un aspect essentiel de la programmation objet : l'héritage.

Finalement nous nous retrouvons avec cette hiérarchie ci-dessous avec une classe abstraite définissant une forme quelconque et contenant sa couleur :



On remarque qu'il y a une méthode de calcul de l'aire d'une forme qui devra être définie pour chaque forme concrète.

Pour le calcul de l'aire du segment, du triangle et du cercle on utilise les formules classiques connues, pour le polygone nous avons decider de "découper" la forme en plusieurs triangles (dont on sait déjà calculer l'aire).
Pour cela nous cherchons d'abord le centroïde du polygone, puis nous formons des triangles avec 2 points successifs du polygone et le centroïde.



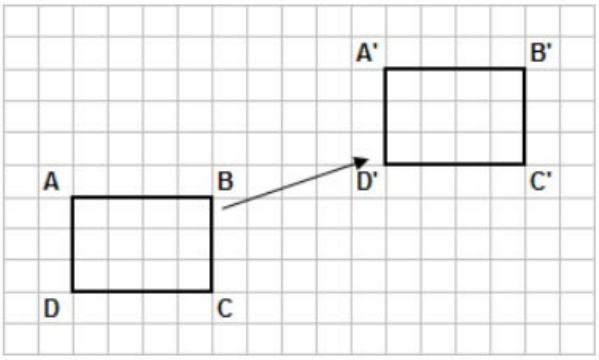
Il ne reste ensuite plus qu'à faire l'additionner de l'aire de tous ces triangles pour obtenir celle du polygone.

En plus de ce calcul, il y a 3 méthodes importantes qui nous permettent d'appliquer une transformation sur ces formes, nous allons donc les détailler ci-dessous.

Transformation sur les formes :

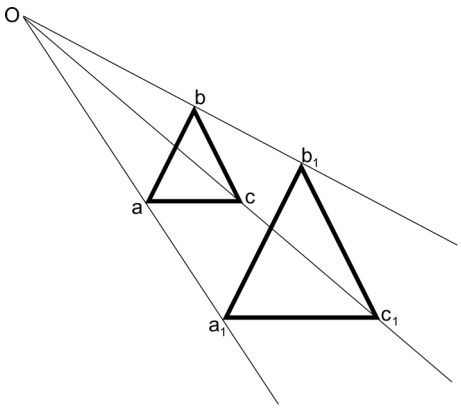
- Translation :

La translation d'une forme permet son déplacement selon un vecteur donner en paramètre.
Cette opération est très simple, il suffit d'additionner le vecteur de translation à chaque point de la forme concernée.



- Homothétie :

Appliquer une homothétie sur une forme revient à appliquer un zoom (ou dézoom). Il nous faut pour cela un point fixe O et un rapport d'homothétie.



La transformation est un peu plus complexe, mais comme pour la translation il faut appliquer la formule suivante sur chaque point des formes.

On a un point (vecteur) P (a, b), un vecteur d'homothétie V(x, y) et un rapport k :

$a = (k * (a - x)) + x$

$b = (k * (b - y)) + y$

- Rotation :

La rotation d'une forme peut servir a la faire pivoter sur elle-même en fonction d'un angle en radians, mais nous pouvont aussi la faire tourner autour d'un point passer en paramètre. Pour réaliser cette opération nous devont encore une fois appliquer une formule sur chaque point de la forme en question :

On a un point P (a, b), un point de rotation V(x, y) et un angle r (en radian) :

$$a = \cos(r) * (a - x) - \sin(r) * (b - y) + x$$
$$b = \sin(r) * (a - x) + \cos(r) * (b - y) + y$$

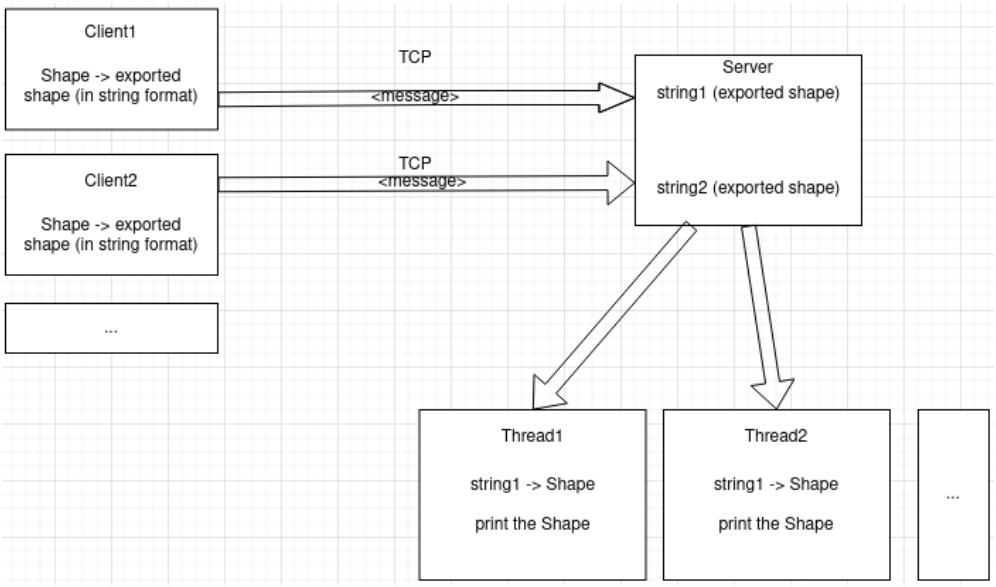
Rendu graphique des formes :

- Utilisation d'un serveur :

Maintenant que l'on peut créer des formes et les manipuler, on aimerait pouvoir les afficher sur notre écran. Pour cela nous allons nous servir d'un serveur de dessin qui s'occupera de cette tâche à notre place. Ce serveur permettra d'alléger notre programme en terme de ressource à l'exécution, ce qui est un avantage si l'on veut garder notre logiciel portatif sur de plus petites machines moins puissantes.

Pour que le serveur sache quoi dessiner, nous allons communiquer avec lui via le protocole TCP, qui permet d'envoyer des requêtes sur le réseau, donc à distance de notre machine faisant tourner l'application C++.

Le principe de cette communication est de pouvoir envoyer un message du client vers le serveur grace au TCP, donc il va falloir transformer nos formes en chaine de caractères contenant leurs informations essentielles.
Une fois la connexion établie avec un client, le serveur va créer un thread on l'on pourra charger nos formes et les afficher. Ceci veut dire que plusieurs clients pourront se connecter en meme temps au serveur ce qui est un autre avantage de cette solution.



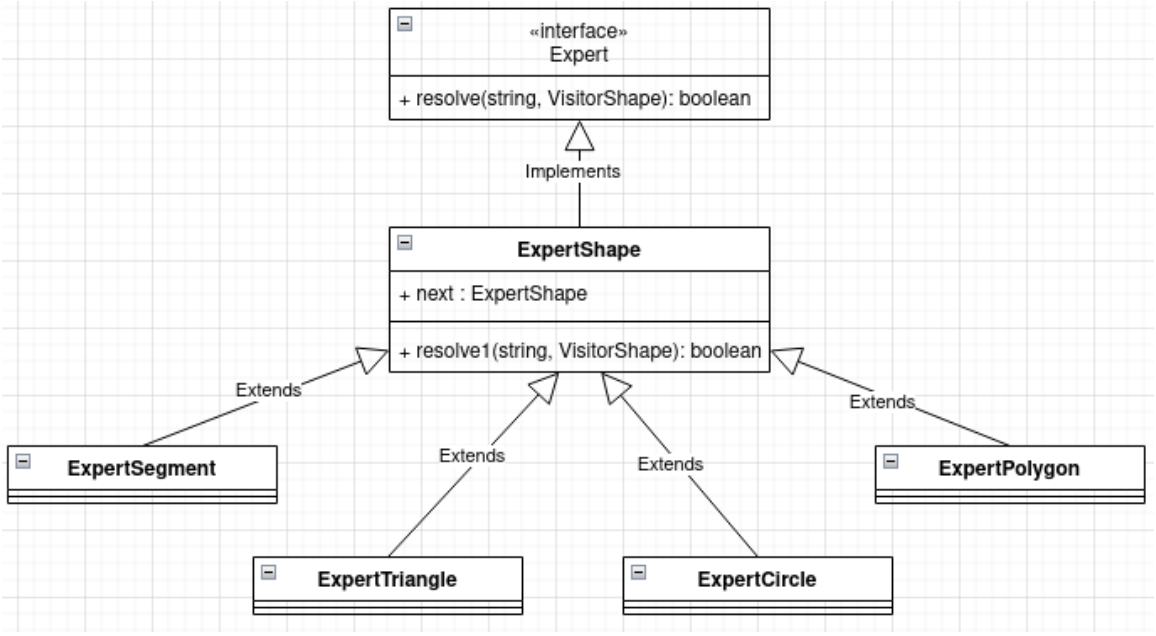
Ce serveur utilisera un programme Java, qui possède beaucoup de choix de gestion graphique directement intégré comparé au C++ qui n'en a aucun de base.

Pour le moment les requêtes représentant nos formes sont de la forme suivante :

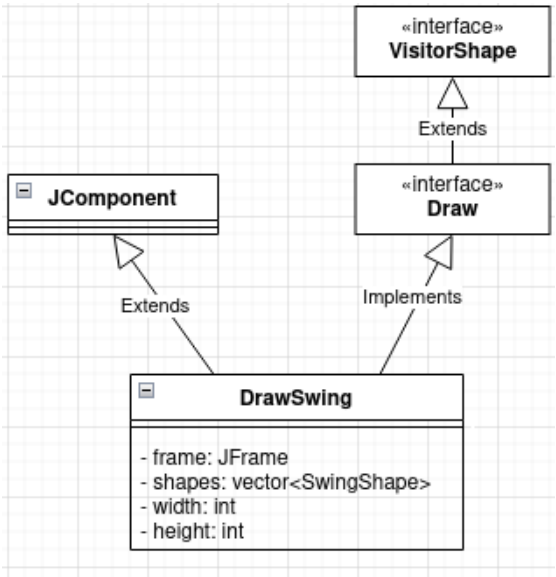
NOM_DE_FORME;COULEUR;LISTE_DES_POINTS

Après réception d'une requête, le serveur va pouvoir la parser grace à un design pattern appelé Chain Of Responsibility. Il permettra d'extraire les données et de passer d'une simple chaine de caractères à une forme prêt a être dessiné.

Cette chaine est couplée à un autre design pattern qui lui, permettra à la forme d'être dessiné.

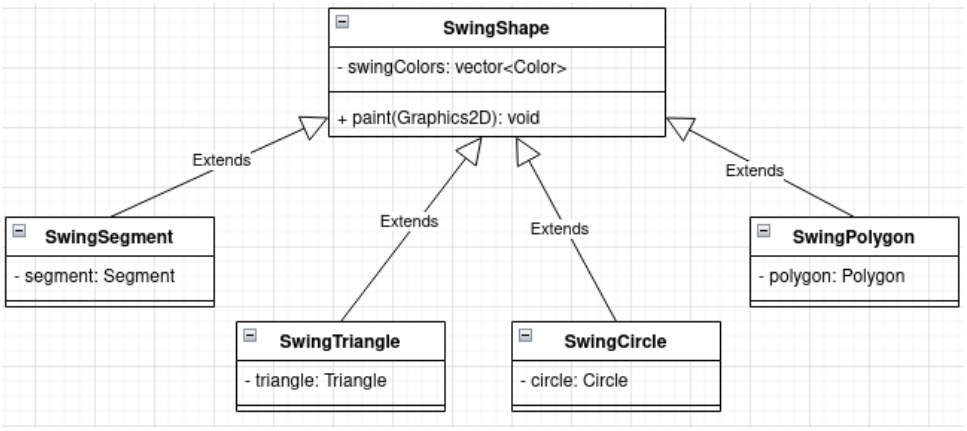


Ce dernier est nommé Visiteur et il aide à garder une indépendance vie à vie de la bibliothèque graphique utilisée pour afficher les formes. En effet avec cela nous pourrons visiter chaque forme et injecter notre code de dessin sans avoir besoin de modifier la classe de la forme en question.



Pour ce projet nous avons choisi la bibliothèque Swing qui est légère et qui est déjà intégrée à Java.

Nous avons aussi ajouté des classes pour nous permettre de dessiner toutes les formes en même temps sur une fenêtre. Pour cela elles prendront en paramètre la partie graphique utilisé par la fenêtre Swing pour y ajouter une forme à dessiner.



- Rendu en C++ dans le logiciel :

L'utilisation d'un visiteur pour dessiner nos formes, vu juste au-dessus, nous permet facilement de changer notre mode d'affichage. Ainsi nous pouvons implémenter une bibliotheque graphique directement dans notre programme C++ et gérer le dessin au même endroit que la création des formes.
Avoir un logiciel qui permet de tous faire de lui-même est un gros avantages car nous enlevons le besoin de mobiliser en permanence un serveur de dessin.

Sauvegarde et chargement des formes :

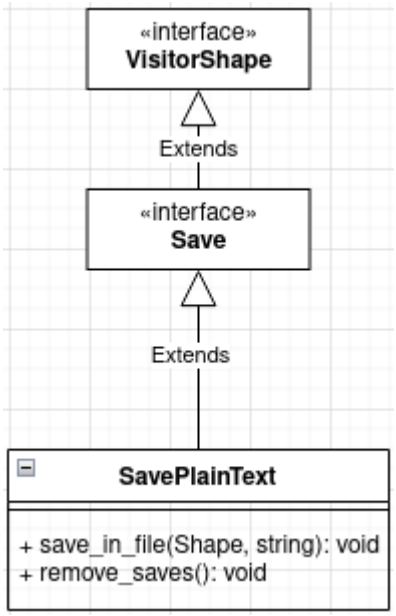
- Sauvegarde :

Lors de manipulations sur des formes il peut s'avérer utile de pouvoir garder une trace de celles-ci, alors pour accomplir cela nous avons mis en place un système de sauvegarde.

Pour le moment nous sauvegardons ces objets dans des fichiers textes, sous la même forme que lorsque l'on communique avec le serveur de dessin, et les groupes sont émulés par des dossiers contenant leurs formes.

Mais pour prévoir de futurs moyens de sauvegarde (XML, …) nous avons réutilisé un design pattern Visiteur comme pour l'affichage graphique.

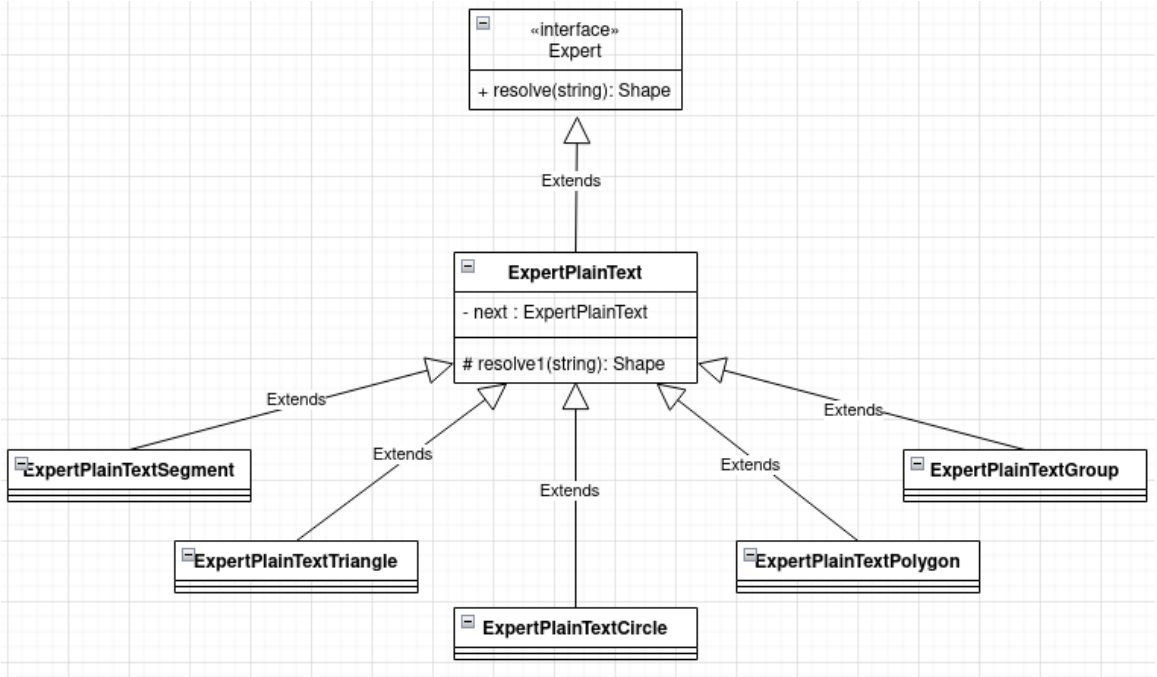
Dans le même esprit que pour la structure de dessin nous obtenons cette hiérarchie de classes :



Cette hiérarchie nous permettra facilement de modifier ou de rajouter de nouveau moyens pour la sauvegarde de nos formes.
Pour mettre en place le code gérant la sauvegarde il suffit de redéfinir les méthodes de visite contenues dans l'interface VisitorShape.

- Chargement :

Le chargement d'une forme est très similaire au dessin sur le serveur. En effet nous réutilisons une Chain Of Responsibility pour traiter chaque cas particulier.



On ajoute cependant un cas pour les groupes de formes.
Ici nous chargeons les formes préalablement sauvegarder sous la même forme que les requêtes envoyées au serveur, mais comme pour la sauvegarde il sera possible d'étendre le chargement à d'autres types de fichiers.

Futures améliorations:

Comme nous avons pu le décrire lors de ce rapport, notre projet utilise une multitude de designs patterns facilitant la modification de notre code mais surtout l'ajout de nouvelles fonctionnalités qui n'altéreront pas le code existant.

Ainsi nous pouvons imaginer une mise en place d'un affichage directement intégré au logiciel en cas de panne du serveur du dessin ou si nous n'avons pas la possibilité de communiquer avec lui.

En suivant cette logique nous pourrions aussi modifier la bibliotèque de dessin utiliser sur le serveur si elle vient à être abandonnée, ou qu'une nouvelle plus puissante sorte entre-temps.

Le système de sauvegarde et de chargement de forme actuelle reste rudimentaire et nous pourrions le modifier par un système utilisant de l'XML par exemple, ce qui simplifierait la création de fichiers de tests.

De plus, comme ces designs patterns utilisent l'héritage sur des classes meres abstraites, nous pouvons implementer plusieurs solutions dans le projet sans qu'elles ne rentrent en concurrence. Il faudra juste instancier la méthode voulue lors de l'exécution.

Enfin, le point essentiel de ce programme étant de manipuler des formes, nous avons tout fait pour faciliter l'ajout de nouvelles formes, et les seules classes à modifier dans ce projet pour permettre leur addition seront les visiteurs. Il faudra bien sûr rajouter un cas aux chaines de responsabilité pour traiter les nouveaux cas mais nous n'aurons pas besoin de toucher à d'autres classes ce qui rend leurs ajout au code existant très rapide.