PAN Yilei

SIMIER Quentin

UNIVERSITÉ DE LORRAINE

RAPPORT PROJET DE SYNTHÈSE

TABLE DES MATIERES

PAN Yilei

SIMIER Quentin

LE SUJET DANS SA GLOBALITE :

Le projet a pour but de développer une application gérant différentes formes géométriques 2D. Cette application se compose d'un côté client programmé en C++ et d'une partie serveur en JAVA. Il doit être possible d'appliquer diverses opérations sur ces formes, en particulier une homothétie, une translation ainsi qu'une rotation. Ces formes devront pouvoir être dessinées par le serveur et on pourra également les sauvegarder et charger depuis un fichier. Une ressource est disponible aussi bien pour le côté C++ avec Doxygen que pour la partie Java avec Javadoc.

FONCTIONNEMENT GLOBAL DE L'APPLICATION

L'application fonctionne de cette manière : L'utilisateur définit la forme qu'il veut dessiner ainsi que les données la concernant (exemple : le centre et le rayon si un cercle doit être dessiné). Une fois ceci fait, le client va aller se connecter sur le serveur JAVA. Le serveur est toujours en écoute et est multi-thread. C'est-à-dire qu'il peut gérer plusieurs clients à la fois. Le client va donc se connecter et envoyer une requête au serveur avec les informations sur la forme pour que celui-ci puisse correctement la dessiner. Si la forme ne peut pas être dessinée pour certaines raisons alors l'utilisateur en est averti et la connexion se coupe entre le client et le serveur. Si les données sont bonnes alors une fenêtre s'ouvre (le titre ainsi que la taille seront au préalable choisis par l'utilisateur) et la forme est dessinée à l'intérieur. Le client se déconnecte.

DIAGRAMME UML :

La classe principale de l'application est **Forme**. En effet c'est à partir d'elle qu'on va pouvoir créer toutes sortes de formes pour ensuite faire plusieurs opérations dessus.

Nous avons donc plusieurs formes particulières (**Cercle**, **Segment**, **Triangle**, **Polygone**, **Groupe**) qui héritent de la classe abstraite Forme.

Une classe **Exception** est également implémentée pour gérer toutes les erreurs qui peuvent potentiellement intervenir pendant le fonctionnement de l'application. Les erreurs peuvent intervenir soit au moment de la connexion entre le client et le serveur, si l'un des protagonistes est par exemple inaccessible, ou bien lors de l'enregistrement d'une forme si le fichier ne dispose pas assez de droits.

Une classe **SingletonConnexion** est utilisée pour créer une nouvelle instance de connexion une seule fois dans tout le programme. Lorsque le client veut communiquer avec le serveur, une nouvelle connexion sera établie entre les deux, les requêtes seront envoyées, puis la connexion sera fermée.

Chaque forme particulière dispose de méthodes pour effectuer diverses opérations telles qu'une *homothétie*, une *translation* ou encore une *rotation*.

Divers design patterns sont mis en place pour rendre l'application la plus indépendante possible, pour par la suite ajouter de nouvelles fonctionnalités sans avoir besoin de modifier tout le code précédemment écrit.

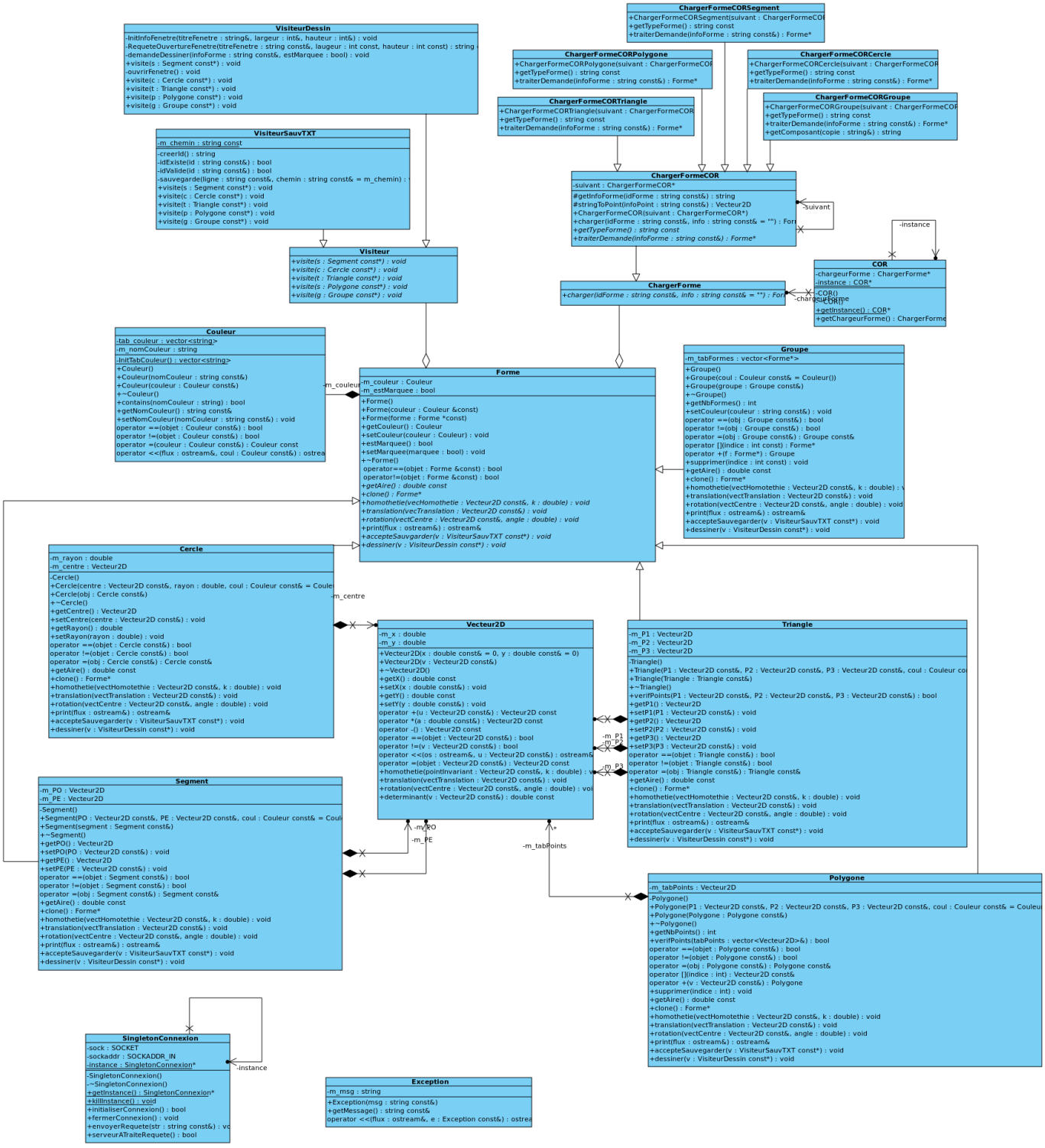
On verra par la suite plus en détail ces design patterns.

Les design patterns ont également été employés dans la partie Java de l'application.

PAN Yilei

SIMIER Quentin

Schéma UML :



PAN Yilei

SIMIER Quentin

GESTION D'UNE FORME GEOMETRIQUE

Comme présenté rapidement avant, une forme est représentée par une classe abstraite. Toutes les formes sont composées d'une couleur ainsi que de différentes données qui peuvent varier en fonction du type de forme. Elles possèdent également un booléen (*estMarquee ()*) qui correspond à leur présence dans un groupe ou non. De ce fait on peut savoir si elles sont déjà utilisées quelque part.

Une forme peut être simple ou bien composée. Dans ce dernier cas, elle sera alors composée d'autres formes et le principe reste le même. Une forme composée est représentée en tant que groupe qui hérite de la classe forme et qui contient un tableau de toutes les formes qui la composent. Une forme ne peut pas appartenir à plusieurs groupes en même temps.

On considère qu'il n'y a jamais d'intersection entre les formes.

On peut appliquer 3 transformations (énoncées précédemment) différentes sur chaque forme. On suppose que la forme est modifiée directement sans avoir besoin de faire une copie de celle-ci. En effet, dans notre cas nous n'avons pas besoin de garder la forme de départ intacte.

La plupart des formes disposent de divers surcharges d'opérateur comme == qui teste si deux formes sont les mêmes, l'opérateur d'affectation. Toutes les formes peuvent être construites par recopie grâce au constructeur par recopie.

Les formes disposent aussi d'une méthode de clonage qui va retourner un pointeur vers une nouvelle forme qui aura les mêmes données et également une méthode de conversion d'une forme en une chaine de caractères. Il est possible de calculer l'aire de chaque forme (sauf pour le segment).

CLIENT C++

Le client est programmé en C++. C'est lui qui va créer une instance et établir une connexion vers le serveur pour envoyer une requête de demande de dessin.

Chaque requête est envoyée de la même manière. A chaque forme à dessiner, la méthode *getInstance()* est appelée. Cela va donc réinitialiser le socket qui est un singleton et rétablir la connexion avec la méthode *initialiserConnexion()*. Le client va donc établir une connexion vers le serveur Java, envoyer une première requête de demande d'ouverture d'une fenêtre graphique pour pouvoir dessiner à l'intérieur, puis une requête de demande de dessin pour dessiner la forme. Une fois la requête envoyée, on peut savoir si elle a bien été traitée avec la méthode *serveurATraiteRequete()* qui nous renvoie un booléen vrai si la requête a effectivement bien été traitée ou un booléen faux dans le cas contraire.

Toutes ces méthodes sont implémentées avec le design pattern « Singleton » que l'on verra par la suite.

PAN Yilei

SIMIER Quentin

SERVEUR JAVA

Dans cette application, le côté JAVA correspond au serveur qui va récupérer et traiter les requêtes envoyées par le client pour dessiner des formes.

De son côté JAVA est donc multithreading donc peut gérer plusieurs clients en même temps. Le serveur est toujours en écoute par qu'un client puisse se connecter quand il le souhaite. Lors de la réception d'une requête, le serveur va traiter celle-ci en application différentes actions sur la chaine (on verra plus en détail ce point lorsque l'on parlera des design patterns) puis utiliser une librairie graphique pour dessiner la forme en question.

Le serveur va donner à chaque nouveau client un numéro pour que ce soit plus simple par la suite pour debugger ou bien avoir des données sur le nombre de clients qui se sont connectés.

A chaque fois que le serveur est sollicité par une requête il renvoie « 1 » au si tout s'est bien passé et « 0 » s'il y a eu un problème. C'est la méthode *serveurATraiteRequete(…)* du client C++.

La classe **ObjetAwtFenetre** permet de dessiner une forme dans une fenêtre grâce aux méthodes *creerFenetre(…)* qui permet de créer une fenêtre dans laquelle dessiner ainsi qu'une méthode pour chaque type de forme que l'on doit dessiner (dessinerCercle, dessinerSegment …).

La plus grosse partie du serveur concerne le design pattern « Chain Of Responsability » qui nous verrons plus tard.

SAUVEGARDE DES FORMES DANS UN FICHIER

L'une des fonctionnalités que doit avoir l'application est la sauvegarde de formes. Nous avons décidé d'utilise le format de fichier texte pour stocker les données car c'est intuitif et facile d'utilisation, de manipulation.

Comme chaque forme ne possède pas les mêmes données en commun nous avons dû utiliser un visiteur. Il va nous permettre d'appeler la bonne méthode liée à la forme que nous devons stocker. De ce fait, si un jour une nouvelle forme venait à être ajoutée, alors il suffirait d'ajouter une nouvelle méthode de visite dans le visiteur pour sauvegarder cette forme.

Une forme est sous la forme : idForme-TypeForme:CouleurForme;données1;données2;…;

Un groupe est sous la forme : idGroupeTypeForme:CouleurForme;InfoForme1(sans identifiant mais avec «-»)&InfoForme2&InfoForme3&

Cette syntaxe est propre à toutes les formes sauf en ce qui concerne les champs de données.

Ainsi la classe **VisiteurSauvTXT** possède plusieurs méthodes pour sauvegarder une forme. La méthode *creerId(…)* qui va demander à l'utilisateur un nom qui correspondra à l'identifiant de la forme. Cet identifiant doit être unique et ne pas contenir certains caractères qui peuvent être confondus avec les séparateurs pour extraire les données de la forme. Pour se faire deux méthodes *idValide(…)* et *idExiste(…)* sont utilisées.

La méthode *sauvegarde(…)* va juste aller ajouter une ligne de texte qu'on passe en paramètre dans un fichier.

PAN Yilei

SIMIER Quentin

Il existe une méthode visite pour chaque type de forme que l'on veut stocker. Pour chaque méthode on va d'abord vérifier si la forme fait parti d'un groupe ou non, puis récupérer les données propres à la forme et pour finir enregistrer ces informations dans le fichier. Juste avant d'enregistrer les informations on prend soin de supprimer les potentiels espaces dans l'identifiant de la forme pour éviter tout problème de syntaxe par la suite.

Concernant le groupe c'est légèrement différent puisque l'on va juste récupérer la couleur du groupe puis boucler sur chaque forme qui le compose et appeler alors la méthode correspondante à la bonne forme.

CHARGEMENT DES FORMES DEPUIS UN FICHIER

Bien évidemment puisque l'on peut sauvegarder des formes dans un fichier, il faut ensuite pouvoir les récupérer. Pour ce faire nous avons utilisé le design pattern COR (« Chain of Responsibility »). Il est représenté par la classe abstraite **COR**. Elle possède un constructeur par défaut qui va aller chainer entre eux tous les experts. Tout comme pour la classe **SingletonConnexion**, notre **COR** ne sera instancié qu'une seule fois grâce à la méthode getInstance qui retournera un pointeur sur l'instance si celle-ci est déjà créée ou bien la créera.

La méthode *getChargerForme(…)* permet de retourner un pointeur sur un expert.

Un expert est donc représenté par la classe abstraite **ChargerFormeCOR**. La classe qui hérite directement de celle-ci (**ChargerFormeCOR**) possède diverses méthodes comme *getInfoForme(…)* qui permet de récupérer les informations d'une forme dans un fichier. La méthode *stringToPoint(…)* permet de convertir une chaine de caractères représentant un point en un **Vecteur2D**.

Et la méthode principale qui est charger. C'est elle qui va aller demander aux experts de résoudre le problème. Si l'expert n'est pas capable de résoudre le problème et qu'il reste au moins un expert après lui alors on passe au suivant, sinon on retourne NULL.

Pour savoir si un expert peut résoudre le problème on a implémenté une méthode (*getTypeForme(…)*) dans chaque expert (classes **ChargerFormeCORCercle**, **ChargerFormeCORTriangle** …) qui retourne une chaine de caractère correspondante au type de forme que l'expert peut traiter. Par exemple la méthode *getTypeForme(…)* de la classe **ChargerFormeCORCercle** va retourner la chaine de caractères « Cercle ». Si l'expert en question peut traiter la demande alors on va appeler une autre de ses méthodes : *traiterDemande(…)*. Elle prend en paramètre une chaine de caractères qui contient toutes les données de la forme (couleur, Vecteur2D …). L'expert va ensuite extraire ces données pour enfin créer un nouvel objet contenant toutes les informations qui étaient stockées.

REQUETES DE DESSIN

Pour que le serveur puisse dessiner des formes et surtout quoi savoir dessiner, il faut bien que le client lui envoie les bonnes informations avec le bon format et la bonne syntaxe. Il faut que les deux côtés utilisent les mêmes règles pour pouvoir se comprendre. C'est pourquoi nous avons implémenté côté client, une classe **VisiteurDessin** qui hérite de la classe **Visiteur** (tout comme **VisiteurSauvTXT** vu précédemment).

Dans cette classe nous avons défini la syntaxe que le client et serveur devaient comprendre pour que l'application fonctionne correctement.

Pour ce faire, la classe dispose d'une méthode appelée *InitInfoFenetre(…)* qui permet de demander à l'utilisateur un nom pour la fenêtre de dessin qui sera créé côté serveur, ainsi que le choix de la hauteur et de la largeur de la fenêtre. Si l'utilisateur rentre 0 alors il y a déjà des constantes définies et on prendra donc leur valeur. On considère ici que l'utilisateur rentre toujours un nombre. Nous n'avons pas fait de vérifications si celui-ci rentre un caractère par exemple. Mais cela peut facilement être implémenté par la suite pour augmenter la sécurité de l'application.

La méthode *RequeteOuvertureFenetre(…)* va convertir les informations concernant la fenêtre en une chaine de caractères avec une certaine syntaxe pour que le serveur puisse reconnaître le type de requête et extraire les données pour par créer la fenêtre correspondante aux données qui ont été envoyées par le client.

La requête est sous cette forme : F$TitreFenetre;Largeur;Hauteur;

Le F$ signifie fenêtre, c'est-à-dire que du côté serveur nous avons implémenté un design pattern COR (voir plus tard) qui regardera quel type de requête est envoyé. Dans cette requête on indique donc que l'on veut créer une nouvelle fenêtre avec le titre *TitreFentre* et de hauteur *Hauteur* avec une largeur *Largeur*.

La méthode *ouvrirFenetre(…)* va donc utiliser les deux méthodes décrites juste avant et envoyer la requête de demande de création de fenêtre au serveur.

Une autre méthode du même type est *demandeDessiner(…)* qui va être appelée par toutes les méthodes visite pour demander au serveur de dessiner la forme. Dans cette méthode tout au début on test si la forme est marquée ou non pour ouvrir une fenêtre. Cette vérification est nécessaire puisque si nous sommes dans un groupe et que l'on veut dessiner par exemple la 3ème forme qui compose se groupe alors il faut dessiner dans la fenêtre actuelle et non pas en créer une nouvelle.

Au contraire, si une nouvelle forme qui n'a rien à voir avec la précédente doit être dessinée alors il faut créer une autre fenêtre.

Il existe bien évidemment autant de méthodes visite de que formes puisque qu'il s'agit d'un visiteur et que l'on doit envoyer des requêtes différentes en fonction du type de forme car elles ne possèdent pas toutes les mêmes données.

Dans chaque méthode visite, on envoie les données sous cette forme au serveur :

D$Forme:Couleur;donnée1;donnée2;…;

D$ signifie dessin pour que du côté serveur la COR sache quoi faire. On indique ensuite la forme à dessiner (Cercle, Triangle, Segment …), la couleur de la forme et ensuite les données propres à chaque type de forme.

On va tester si la forme est marquée ou non. En effet, si elle est marquée (donc fait partie d'un groupe) alors on ne doit pas initialiser une nouvelle connexion. On envoie ensuite les requêtes au serveur (ouverture de la fenêtre + dessin de la forme) et on ferme la connexion ainsi que l'instance.

On considère dans ce projet qu'un utilisateur se connecte au serveur uniquement pour demander de dessiner une forme simple ou composée. C'est-à-dire qu'il va renseigner les informations concernant la fenêtre, la forme, tout envoyer au serveur, attendre une réponse puis directement se déconnecter. Si il veut de nouveau envoyer une forme, il devrait répéter cette opération autant de fois que nécessaire.

Du côté serveur, nous avons utilisé le design pattern COR pour interpréter les couleurs, les requêtes ainsi que les formes.

La classe COR sert à chainer tous les experts entre eux.

La classe InterpreteurRequete sert à interpréter toutes les requêtes qui arrivent. En effet, lorsqu'une requête arrive au serveur, on extrait le premier caractère pour regarder si c'est un D ou un F. Dans notre protocole la première requête s'agit d'un F. Notre COR va alors appeler la méthode *traiterRequete(…)* de la classe InterpreteurRequeteCOROuvertureFenetre pour la création de la fenêtre avec les données contenues dans cette requête.

Il va ensuite recevoir une ou plusieurs requêtes (si on veut dessiner un groupe) concernant les formes à dessiner. C'est alors la classe InterpreteurRequeteCORDessinerForme qui va prendre le relais et aller interroger la COR au niveau de la classe InterpreteurForme pour savoir quelle forme dessiner. Dans InterpreteurFormeCOR, grâce à la méthode dessinable, on va pouvoir choisir la bonne ferme à dessiner. Si l'expert ne peut pas la dessiner alors on passe à l'expert suivant.

Le dessin se fait dans la classe appropriée à la forme à dessiner. Si par exemple on veut dessiner un cercle, on va alors appeler la méthode *genererDessin(…)* de la classe InterpreteurFormeCORCercle.

Cette méthode va nous permettre d'extraire toutes les données de la forme, donc la couleur et les données propres à celle-ci.

Entre temps la couleur va aussi utiliser une COR pour définir quelle est la couleur à appliquer à la forme (ou au groupe). Cette COR fonctionne exactement comme les deux précédentes donc on ne va pas s'attarder dessus.

Une fois que toutes les données ont été stockées dans des variables on peut utiliser ces variables pour dessiner la forme grâce à l'objet awt que l'on aura passé en paramètre de cette méthode.

Une chaine de caractère égale à 1 est ensuite renvoyée au serveur pour indiquer que la forme a bien été dessinée ou égale à 0 dans le cas contraire.

CONCLUSION

Le cœur de ce projet sont les design patterns, en effet ils permettent d'implémenter du code qui est totalement indépendant, ce qui permet par la suite d'ajouter de nouvelles fonctionnalités, de faire des modifications de code de manière plus simple et plus rapide.