UNIVERSITÉ DE LORRAINE

Licence 3ème année
Promotion 2020-2021

Projet de synthèse
Premier Pas vers l'Ingénierie du Logiciel

Membres du Binôme:
Lorian SIZAIRE
Camille MASSON

I) Présentation du projet

Ce projet a pour but de réaliser une application distribuée avec un client C++ et un serveur JAVA permettant la création et la gestion de formes géométriques en 2D. Elles comportent différentes fonctionnalités comme des transformations de formes, des sauvegardes et chargements de formes via des fichiers et la gestion de plusieurs fenêtres de dessin en même temps.

De plus ce projet doit suivre différents patrons de conception (design pattern) tel que le design pattern Visiteur, le Singleton et la Chaîne de responsabilité pour la création de ses fonctionnalités et ainsi permettre une extensibilité de l'application.

II) Gestion du projet:

Répartition du travail:

Pour travailler efficacement en binôme nous nous sommes réparties les tâches dès le début du projet. Ainsi, même si cela ne nous a pas empêché de nous entre aider, nous avons réparti les tâches de la façon suivante:

- Conception des formes avec les paramètres et l'héritage à appliquer : Lorian et Camille
- Conception des instructions entre le serveur et le client : Lorian et Camille
- Créations des formes simples et composées en C++ : Camille
- Création des fonctions de transformation en C++ : Camille
- Singleton de connexion : Camille
- Pattern Visitor des fonctions dessiner : Camille
- Calculs d'aire des formes : Lorian et Camille
- Serveur multithreading JAVA : Lorian
- Chaîne de responsabilité des requêtes du serveur : Lorian
- Stockage des couleurs et des fenêtres côté serveur : Lorian
- Services de gestion des formes et des fenêtres du serveur : Lorian
- Conversion en string des formes : Lorian
- Sauvegarde des formes (Pattern Visitor) : Camille
- Chargement des formes (Chaîne de responsabilité) : Lorian
- Console : Camille
- Rapport et Schémas : Camille et Lorian

Outils utilisés:

Pour effectuer ce projet en coopération tout en restant à distance nous avons dû utiliser des outils de communication, de partage et de travail.

Ainsi pour la communication nous avons principalement utilisé discord pour sa facilité et ses partages d'écrans, google drive pour partager des documents comme la répartition des tâches et le rapport. Aussi pour la création du serveur JAVA et du client C++ nous nous sommes servi respectivement d'Eclipse et de Visual Studio. Et enfin pour regrouper et partager les changements du projet nous avons utilisé Git Bash et GitHub.

III) Conception et réalisation du projet:

Protocole de communication:

Pour le protocole de communication, le client utilise une socket de WinSCP via un singleton auquel on donne une adresse ip et un port et qui tente ensuite une connexion au serveur. Le serveur lui est démarré et mis en écoute sur un port défini puis va attendre des connexions. Une fois qu'un client fait une demande, le serveur l'accepte et l'affecte à un socket qui sera géré par un thread dédié.

Une fois la connexion établie et le client affecté à un thread, le thread attend en boucle des instructions du client jusqu'à la fermeture de cette connexion. Chaque instruction est testée et effectuée en fonction de la requête.

Pour distinguer à quelle requête l'instruction est destinée, nous avons mis en place un modèle simple de message à décomposer et tester. Le message est découpé à partir d'un délimiteur sous forme de "/" et comporte en première position un mot qui correspond à l'action, suivie d'un nombre défini de paramètre qui dépend de l'action voulue.

Partie JAVA:

a) Conception du serveur

La première étape de la création du serveur a consisté en la mise en place d'un serveur de la bibliothèque java et du paramétrage de celui-ci avec le port par défaut "9111" basé sur le serveur JAVA simple vu en cours *(Class ServeurBase)*. Une fois cette étape faite et la création de la boucle d'attente des clients effectuée, il a fallu

créer un gestionnaire des threads des clients *(Class ThreadClientHandler)* qui est toujours basé sur la version du cours (bien qu'il soit modifié pour être adapté au projet). Ce gestionnaire comporte lui aussi une boucle d'attente des instructions et un moyen de les récupérer sous forme de string.

Ensuite pour gérer les instructions des clients et mettre en place les fonctionnalités de l'application, il a fallu créer la chaîne de responsabilité des requêtes demandées dans le sujet et créer une classe dictionnaire permettant de parcourir celle-ci avec facilité *(Class DictionnaireRequete)*. Cette classe comporte un simple constructeur qui permet d'ajouter des classes de requêtes en une ligne et un paramètre qui repère le début de la chaîne, et une fonction de récupération de ce paramètre.

La chaîne de responsabilité a été assez rapidement mise en place grâce à la préparation des TD et comporte une classe abstraite des requêtes *(Class abstraite Requete)*, une classe abstraite de requête pour la chaîne de responsabilité *(Class RequeteCOR)* qui hérite de la classe *Requete* et plusieurs classes qui héritent de la classe *RequeteCOR* et qui définissent le test à effectuer sur l'instruction ainsi que les actions à faire en cas de réussite du test.

Nous retrouvons donc dans cette chaîne les classes descendantes de *RequeteCOR* suivantes:

- DessinCroix qui permet de dessiner une croix
- DessinRond qui permet de dessiner un rond/cercle
- DessinSegment qui permet de dessiner une simple ligne
- DessinTriangle qui permet de dessiner un triangle
- EffacerDessins qui permet d'effacer tous les dessins d'une fenêtre
- FermetFenetre qui permet de fermer une fenêtre spécifique
- OuvrirFenetre qui permet d'ouvrir une fenêtre

Suite à cette chaîne nous avons cherché comment repérer et stocker les fenêtres créées dans la classe *OuvrirFenetre* et utilisées par toutes les autres requêtes ainsi que comment désigner les couleurs et les récupérer sans recréer une liste à chaque utilisation. Le cheminement qui nous a permi de trouver la solution est développée dans la sous-partie suivante mais au final nous avons opté pour deux singletons *(Class FrameFactory et ColorFactory)* qui nous évitent de recréer des listes de couleurs et de fenêtres, et d'y accéder depuis toutes les requêtes avec facilité. Aussi pour ranger les fenêtres, la classe *InfoFrame* a été créé avec comme attribut le numéro du client, le numéro de la fenêtre et la fenêtre elle-même.

b) <u>Problèmes rencontrés</u>

Le serveur JAVA a rencontré différents problèmes mineurs au moment de la création mais qui ont généralement été réglés dans les minutes. Mais il a aussi nécessité par moments de plus de réflexion pour faire face à certains problèmes dûs par exemple à l'utilisation de la chaîne de responsabilité.

Le premier problème que nous avons eu a été la compréhension d'une petite partie du sujet sur les objectifs du serveur. Nous avons hésité au bout d'un moment entre s'il devait gérer plusieurs clients en même temps et ainsi leur permettre de tout dessiner séparément ou si nous devions considérer les fonctions dessin du C++ comme des clients à part entières. Nous avons compris que le premier raisonnement était le bon suite à des échanges par mail avec notre professeur et des discussions entre nous-mêmes.

Le deuxième problème a été la manière de stocker les fenêtres tout en permettant au client d'en créer via des instructions, donc dans des classes requêtes de la chaîne de responsabilité, tout en évitant de devoir passer des informations supplémentaires via celle-ci et de laisser accéder les autres classes de la chaîne qui ont besoin de la fenêtre pour y effectuer des actions. Nous avons d'abord exploré l'option de créer une ArrayList dans le thread du client mais nous devions faire passer cette liste dans la chaîne ce que nous voulions éviter. Ensuite nous avons réfléchi à un moyen de sauvegarder les informations de celle-ci côté client mais il fallait donc la recréer à chaque action côté serveur ce qui était impensable pour nous. Enfin nous avons pensé à un singleton regroupant les différentes frames des différents clients. Cette solution fut donc choisie. Après cela nous avons essayé de créer une HashMap des frames dans le singleton mais il est vite apparu qu'une ArrayList d'une classe créée spécialement pour le stockage des fenêtres *(Class InfoFrame)* était plus simple à gérer.

Partie C++:

a) Conception du client

Le Client doit permettre de créer et de stocker des figures mais aussi de les affecter avec différentes fonctions, par exemple: les déplacer, les faire tourner, ...
Il doit aussi permettre d'envoyer des requêtes au serveur pour qu'il affiche les figures dans les différentes fenêtres. Le client est géré par un singleton pour s'assurer que la connexion ne soit établie qu'une et une seule fois.
La partie qui gère la connexion au serveur est séparée de la gestion des figures pour laisser possible la mise en place d'un autre type de serveur, cela et gérer grâce au pattern Visitor, appris en cours. La sauvegarde des données bénéficie aussi de ce modèle pour faire en sorte de pouvoir charger d'autre type de données par la suite.
Sur chaque figure peut s'appliquer différentes fonction:

- Translation selon un vecteur
- Rotation via un angle et par rapport à un point
- Transformation en chaîne de caractères.

Pour la partie du chargement des formes nous avons créé comme pour le serveur JAVA une chaîne de responsabilité avec des classes abstraites et une classe dictionnaire

permettant de la parcourir facilement. Quant aux classes charger elles permettent de lire une forme et un format de fichier spécifique (vérifié par un test sur le nom de la forme et sur un regex). A noter qu'il y a une classe charger pour chaque forme et pouvant toutes lire un fichier texte tant qu'il respecte le modèle de sauvegarde.

Aussi chaque classe lit tout simplement la première ligne et décompose celle-ci pour retrouver les informations de créations de la forme.

b) Problèmes rencontrés

Lors de la conception du Client plusieurs problèmes ont été mis en évidence.
Dans un 1er temps il fallait choisir la manière d'un chaque figure allais être définie.
Il fallait organiser chaque classe de manière à pouvoir utiliser le Visitor pattern lors d'une tentative de dessin.
Un autre problème majeur concerne la mise en place de la connexion avec le serveur Java. Il fallait permettre la communication au serveur de requête établie selon un protocole.
Pour finir, le plus gros problème se trouvait dans les calculs de vecteur. Par exemple, pour faire tourner une figure il fallait recalculer tous les vecteurs qui la composent. Cela peut s'avérer fastidieux pour des figures comportant un nombre indéfini de vecteur.

IV) Conclusion:

Ce projet fut pour nous très intéressant car il est le premier à demander autant de polyvalence avec à la fois la création d'un serveur et d'un client, l'utilisation de C++ et de Java dans le même projet et l'utilisation de patron de conception récemment appris. Il nous a permis de travailler plus sérieusement et de nous créer plus d'expérience en informatique pour nous rapprocher toujours plus du niveau professionnel.

Aussi, bien que nous soyons fiers du travail que nous avons effectué, nous avons été ralenti par les circonstances et le contexte actuel des choses, il y a donc bien des choses que nous aurions voulu améliorer ou ajouter. Nous aurions déjà voulu ajouter la fonctionnalité des calculs d'aires demandées et une meilleure gestion des exceptions notamment du côté des chargements des formes, mais aussi apporter des améliorations au niveau l'affichage des formes sur les fenêtres qui reste trop simpliste.

Enfin nous avons fait notre maximum pour rendre le projet dans le meilleur état possible et profiter de l'expérience.

Annexe:

Version pdf et photo disponible avec le rapport.

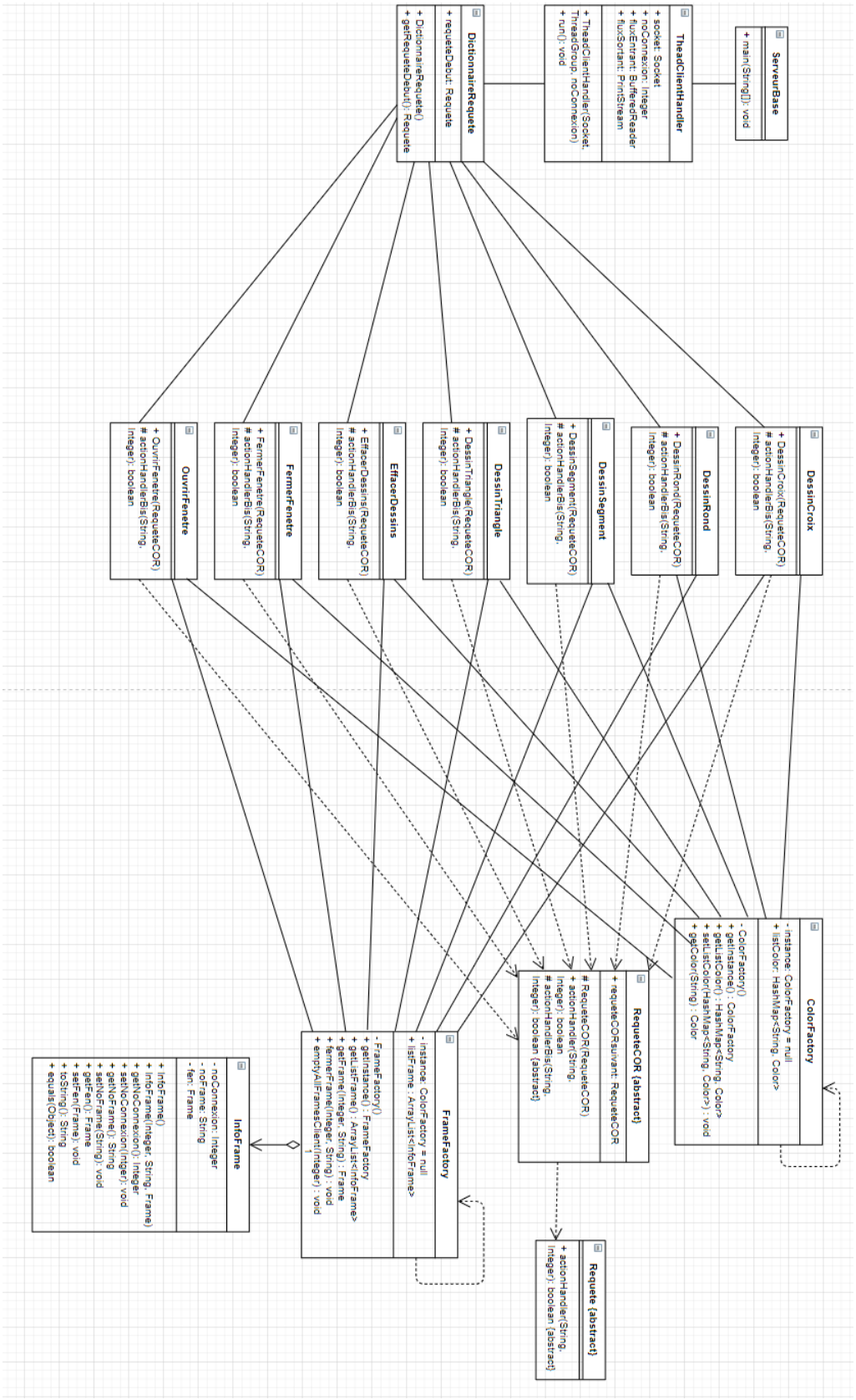

Schéma UML Figure (Client)

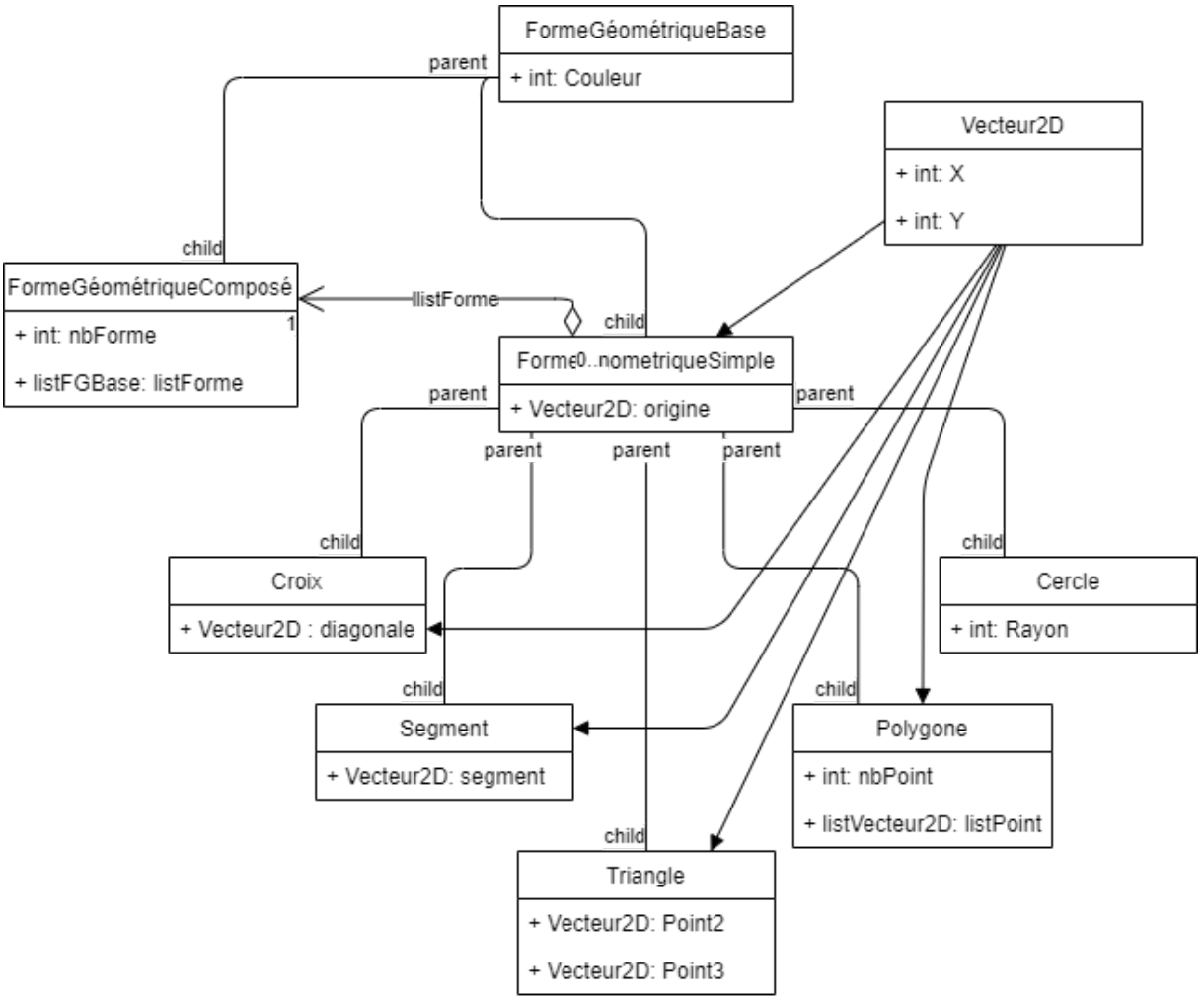

Schéma UML Fonction Dessin Pattern Visitor (Client)

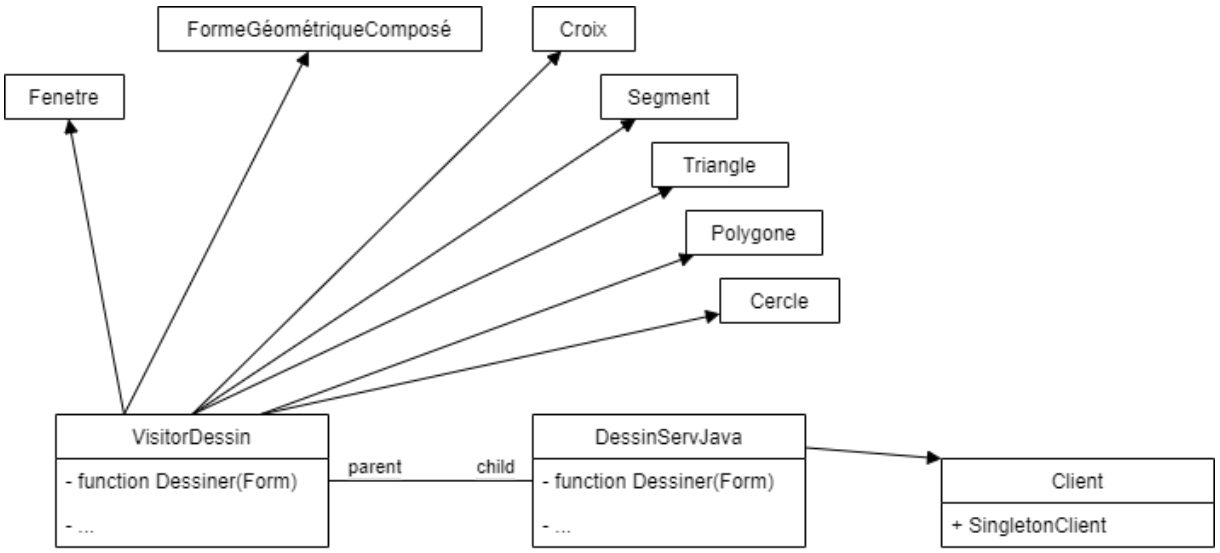

Schéma UML Fonction Sauvegarde Pattern Visitor (Client)