LEOST Maelan et THUILLIER Colin

RAPPORT
- Projet de synthèse -
Premiers Pas vers l'Ingénierie du logiciel
-Gestion de formes géométriques en 2D-

Table des matières

LEOST Maelan et THUILLIER Colin

INTRODUCTION

Le sujet demande la réalisation d'une application distribuée (client C++/serveur Java) pour la gestion de formes géométriques en 2D. Les formes géométriques simples (segment, cercle, triangle, polygone) et les formes géométriques composées (groupes) doivent être prises en compte. Les trois types de transformations géométriques (translation, homothétie et rotation) peuvent être appliqués aux formes, y compris aux groupes. La partie la plus complexe du projet consiste à dessiner les formes géométriques en utilisant une méthode C++ qui agit en tant que client TCP/IP vers un serveur Java de dessin, avec la possibilité d'ajouter une future extension avec une librairie graphique C++ différente.

Il est recommandé d'utiliser la librairie graphique Java pour la partie de dessin car il n'y a pas de librairie graphique standard en C++. Le serveur Java de dessin doit également être écrit, et une librairie réseau C++ doit être utilisée pour la communication client-serveur. Le protocole de communication entre le client et le serveur peut être inventé, mais il est conseillé de s'inspirer de la maquette vue en TD, basée sur des chaînes de caractères. Enfin, pour permettre une extension future avec une librairie graphique C++ différente, la méthode de dessin doit utiliser le pattern Visitor pour séparer l'algorithme du client TCP/IP.

I/ Organisation du projet

A/ Outils utilisés

Afin de réaliser le projet le plus rapidement possible, nous avons défini les logiciels que nous allions utiliser avant de nous répartir les tâches. Par chance, nous travaillons tous les deux sur Mac. C'est pourquoi nous avons décidé d'utiliser Clion pour la partie client (en C++) et IntelliJ pour la partie serveur (en Java), avec l'aide de GitHub Copilot pour certains calculs. Par la suite, nous avons décidé d'utiliser GitHub Desktop pour partager l'avancement de nos parties respectives. Enfin, nous rédigeons ce rapport sur Microsoft Word.

B/ Répartition des taches

Pour assurer une bonne répartition des tâches et travailler efficacement en binôme, nous avons réparti la réalisation des différentes classes et design patterns de la manière suivante :

LEOST Maelan et THUILLIER Colin

Répartition des classes Formes :

- *La classe Forme : Maelan*
- *La classe Segment : Maelan*
- *La classe Polygone : Colin*
- *La classe Cercle : Maelan*
- *La classe Vecteur2D : mise à jour par Colin*
- *La classe FormeComplexe : Colin*
- *La classe Erreur : réalisée par M.Michel*
- *La classe Couleur : Colin*
- *La classe Matrices2x2 : Maelan*

Répartition des design patterns :

- *Le singleton (Communication) : Colin*
- *La chaine de responsabilité (Cpp) : Maelan*
- *La chaine de responsabilité(java) : Colin*
- *Le visiteur : Colin*

De plus, la partie java a été réalisé par Colin.

Nous avons également effectué des tâches supplémentaires, telles que l'arborescence du projet, qui a été réalisée en collaboration par Maelan et Colin, ainsi que les tests qui ont été effectués conjointement par les deux membres de l'équipe.

Cette répartition claire et précise des tâches nous a permis de travailler efficacement en évitant les doublons et en tirant parti des compétences spécifiques de chacun des membres de l'équipe.

II/Conception et implémentation

Dans cette seconde partie de notre rapport, nous allons aborder la conception et l'implémentation de notre projet de dessin collaboratif. Cette partie se décompose en deux parties distinctes, la partie client et la partie serveur.

Nous allons tout d'abord nous intéresser à la partie client, qui comprend plusieurs aspects essentiels tels que la gestion des formes, la communication entre les différents utilisateurs, la chaine de responsabilité pour la gestion des événements, la gestion des erreurs et la

sauvegarde des dessins. Nous avons également implémenté un design pattern "Visiteur" pour faciliter la manipulation des différents éléments du dessin.

Ensuite, nous allons nous pencher sur la partie serveur de notre projet. Cette partie inclut également la chaine de responsabilité pour la gestion des événements, la communication entre les différents utilisateurs, ainsi que la gestion du monde et de ses interactions.

Cependant, malgré notre bonne collaboration et notre efficacité en binôme, nous avons rencontré quelques difficultés lors de la réalisation de ce projet. Nous aborderons donc dans cette partie les problèmes rencontrés, à la fois sur la partie client et sur la partie serveur, et les solutions que nous avons trouvées pour les résoudre.

A/ Partie Client

1/ Les formes

La classe Forme :

La classe Forme est la classe de base de laquelle dérivent toutes les autres formes géométriques du programme. Elle possède comme attribut une couleur codée en hexadécimal. Les méthodes de la classe Forme sont principalement virtuelles pures, ce qui signifie qu'elles doivent être implémentées dans les classes filles.

La classe Forme possède des méthodes virtuelles pures pour le calcul de l'aire, du centre de symétrie, de la translation, de l'homothétie et de la rotation, ainsi que des méthodes pour obtenir les coordonnées des points minimum et maximum de la forme.

En outre, la classe Forme possède une méthode "operator string" qui permet de convertir la forme en une chaîne de caractères représentant la forme.

Rapport sur les classes dérivées de la classe Forme :

La classe Forme est la classe de base de toutes les formes géométriques du programme. Elle est dérivée par plusieurs classes de formes géométriques, chacune ayant ses propres spécificités et méthodes.

- *La classe Segment :*

La classe Segment est une classe dérivée de la classe Forme. Elle représente un segment de droite entre deux points dans un espace 2D.

- *La classe Polygone :*

La classe Polygone est une classe dérivée de la classe Forme. Elle représente un polygone régulier dans un espace 2D.

- *La classe Cercle :*

La classe Cercle est une classe dérivée de la classe Forme. Elle représente un cercle dans un espace 2D.

- *La classe FormeComplexe :*

La classe FormeComplexe est une classe dérivée de la classe Forme. Elle représente une forme complexe composée de plusieurs formes géométriques dans un espace 2D. Cette classe possède des méthodes pour ajouter et supprimer des formes à la forme complexe, ainsi que pour obtenir la liste des formes et leur nombre. Elle possède également une méthode pour modifier la couleur de toutes les formes de la forme complexe.

Comme dit dans la classe Forme, ses classes dérivées ont toutes des méthodes de transformation géométrique, à savoir la translation, l'homothétie et la rotation.

La méthode translation permet de déplacer la forme d'un vecteur donné en ajoutant les composantes du vecteur à chaque coordonnée de la forme. Cette méthode est implémentée dans toutes les classes dérivées, où elle prend en paramètre un vecteur de translation.

La méthode homothétie permet de changer la taille de la forme selon un coefficient de dilation donné en multipliant les coordonnées de la forme par ce coefficient. Cette méthode est implémentée dans toutes les classes dérivées, où elle prend en paramètre un vecteur de dilatation et un coefficient de dilatation.

La méthode rotation permet de faire tourner la forme d'un angle donné autour d'un point donné. Cette méthode est implémentée dans toutes les classes dérivées, où elle prend en paramètre un point de rotation et un angle de rotation.

Nous avons également créé une class enum couleur. Les couleurs des formes sont exprimées en hexadécimal, cependant l'utilisation de cette class fut uniquement dans un but de simplifier l'écriture du code.

2/ La Communication

Comme stipulé dans le sujet, pour la communication j'ai utilisé le Design Pattern Singleton. En effet en utilisant ce pattern pour la classe InitCommunication, il est possible de s'assurer qu'un seul socket est créé et utilisé par toute l'application, évitant ainsi les conflits de socket.

Si plusieurs instances de la classe InitCommunication sont créées, chaque instance créera son propre socket. Cela peut entraîner des conflits de socket, en particulier lorsque les différentes instances essaient de se connecter simultanément à un même serveur. En utilisant le design pattern Singleton, une seule instance de la classe InitCommunication est créée, ce qui signifie qu'un seul socket est créé et utilisé par toute l'application, évitant ainsi les conflits de socket et assurant une connexion stable avec le serveur.

Étant sur macOS il m'a était plus simple et rapide de créer cette class.
Le plus dur dans cette class fut de trouvé les bibliothèques pour pouvoir créer le socket, il a également fallu créer des structures pour pouvoir utiliser certain type.

3/ La chaine de responsabilité

La partie concernant la chaîne de responsabilité (COR) du client est une partie importante de la conception et de l'implémentation de notre système. Cette partie utilise le Design Pattern COR pour résoudre le problème de la lecture de lignes à partir d'un fichier et de la création de formes correspondantes.

Le Design Pattern COR consiste en une chaîne d'experts chargés de traiter des requêtes ou des problèmes. Dans notre cas, chaque expert de la chaîne est chargé de résoudre un problème spécifique de la ligne lue depuis le fichier. La chaîne est organisée de telle sorte que chaque expert traite le problème s'il le peut, sinon il le passe à l'expert suivant de la chaîne.

La partie client de notre système utilise quatre types de classes pour implémenter le Design Pattern COR. Tout d'abord, nous avons la classe abstraite ExpertChargement, qui définit la méthode virtuelle pure resoudre() pour résoudre le problème de la ligne lue. Ensuite, nous avons la classe ExpertChargementCOR, qui est la classe de base pour chaque expert de la chaîne. Cette classe contient un pointeur vers l'expert suivant de la chaîne et la méthode virtuelle resoudre1() pour résoudre le problème de la ligne lue.

Ensuite, nous avons les classes ExpertChargementCercleCOR et ExpertChargementSegmentCOR, ExpertChargementPolygoneCOR et ExpertChargementFormeComplexeCOR qui héritent de la classe ExpertChargementCOR. Elles sont utilisés pour charger un cercle, un segment, un polygone, un triangle et enfin une forme complexe. Chacune de ces classes redéfinit la méthode resoudre1() pour résoudre le problème spécifique de la ligne correspondant à la forme qu'elle est chargée de traiter.

Enfin, nous avons la classe ChargeurListeForme, qui utilise la chaîne d'experts pour charger un fichier texte et créer un vecteur de formes correspondant aux lignes lues. Cette classe utilise les classes ExpertChargementCercleCOR, ExpertChargementSegmentCOR, ExpertChargementPolygoneCOR et ExpertChargementFormeComplexeCOR pour charger chaque forme du fichier.

Cette partie de notre système a été mise en œuvre avec succès, mais elle a également rencontré des difficultés lors de sa conception et de son implémentation. Nous discuterons de ces difficultés dans la section suivante de notre rapport.

4/ La gestion des Erreurs

La classe Erreur qui a été repris de celle réalisée par M. Dominique Michel permet de gérer les erreurs en lançant des exceptions avec des messages personnalisés. Les méthodes fournies permettent de la non-vacuité d'un pointeur, etc.

5/ La partie Visiteur

Le Design Pattern Visitor est une approche complexe qui nécessite une compréhension approfondie pour réussir son implémentation. Dans ce projet, il a fallu plusieurs séances de travaux pratiques pour saisir les concepts de ce design. Cependant, une fois les principes fondamentaux assimilés, il a été facile de l'appliquer avec succès dans deux cas d'utilisation différents. J'ai pris la décision de séparer les visiteurs en deux catégories distinctes : les visiteurs de librairie et les visiteurs de sauvegarde. En effet si nous voulons pouvoir envoyer des données avec les différents arguments je pense qu'il est préférable de séparer ces deux visiteurs.

a) Sauvegarder

Dans le cadre de notre projet, nous avons implémenté un visiteur qui permet de sauvegarder les formes dans un fichier .txt. Toutefois, grâce à l'utilisation du Design Pattern Visitor, il serait facile de créer un nouveau visiteur qui enregistrerait les formes dans un autre fichier avec une extension différente.

b) Dessiner

Dans le but d'envoyer les données de forme au serveur, j'ai décidé de les transmettre sous une forme spécifique, compatible avec la librairie AWT. Cependant, il est possible que dans le futur nous ayons besoin d'envoyer ces données d'une autre manière, à une autre librairie Java. Comme dit précédemment grâce à l'utilisation du Design Pattern Visitor, il serait facile de créer un nouveau visiteur qui pourrait envoyer les formes selon cette nouvelle méthode. Ainsi, nous

aurions la flexibilité nécessaire pour adapter notre application en cas de besoins futurs sans avoir à modifier le code existant de manière significative.

B/ Partie Serveur

Dans la seconde partie du projet, l'objectif était de récupérer les données envoyées par le client en C++ afin de dessiner les formes de la requête reçue. J'ai opté pour l'utilisation de la bibliothèque AWT et plus également la classe Frame pour la création de ma fenêtre.

1/ La chaine de responsabilité

Implémenté la chaine de responsabilité à l'aide du Design Pattern Chain of Responsibility. Cette implémentation a été plus aisée que celle réalisée en C++. En effet, des méthodes telles que "split" nous ont permis de parser plus rapidement une ligne de caractères, ce qui a facilité la mise en place de la chaîne. Chacune des class parsent la forme souhaitée et la dessine en appliquant les modifications nécessaires des coordonnées pour correspondre au monde.

2/ Le Monde

La gestion du monde était essentielle pour pouvoir réaliser les transformations géométriques nécessaires à l'affichage des formes. Pour ce faire, j'ai créé deux classes distinctes : une pour les opérations mathématiques telles que le changement de repère et la transformation en coordonnées écran, et une autre pour la création de la fenêtre graphique permettant de dessiner les formes avec les nouvelles coordonnées obtenues grâce à la première classe. Cette approche modulaire et bien structurée m'a permis de gérer efficacement le monde dans le cadre de ce projet.

3/ La Communication

Nous avons utilisé la même méthode de communication Java que celle vue lors des travaux pratiques. Cependant, ici nous avons implémenté notre propre expert à passer en paramètre de cette communication.

C/ Les difficultés rencontrés

1/ Sur la partie Client

Maelan :

Concernant la chaîne de responsabilité, j'ai rencontré quelques difficultés de compréhension au début, mais j'ai pu les surmonter après une analyse plus approfondie de la mise en place de la chaîne et avoir revu les corrections des anciens TP réalisés. Par la suite, j'ai été en mesure

de comprendre comment chaque classe s'intégrait dans la chaîne et comment elles communiquaient entre elles pour traiter les événements. J'ai également rencontré des difficultés au niveau des calculs nécessaires pour effectuer la rotation qui, je trouve, était la plus difficile des 3 méthodes à implémenter.

Colin :

Au cours du projet, j'ai rencontré des difficultés dans la mise en place du Design Pattern Visitor pour la partie client. Bien que j'aie bénéficié d'une séance de TP explicative, il m'a fallu un certain temps pour comprendre le concept de visiteur. Cependant, une fois la logique assimilée, l'implémentation en C++ s'est avérée beaucoup plus facile.

2/ Sur la partie Serveur

Colin :

La partie la plus difficile du projet a été la partie mathématique de la partie Java. En effet, la compréhension et l'implémentation des calculs nécessaires pour la transformation vers le monde écran ont été fastidieuses et ont demandé beaucoup de temps. En somme, les classes qui ont été nécessaires pour cette transformation ont été les plus complexes à mettre en place.

CONCLUSION

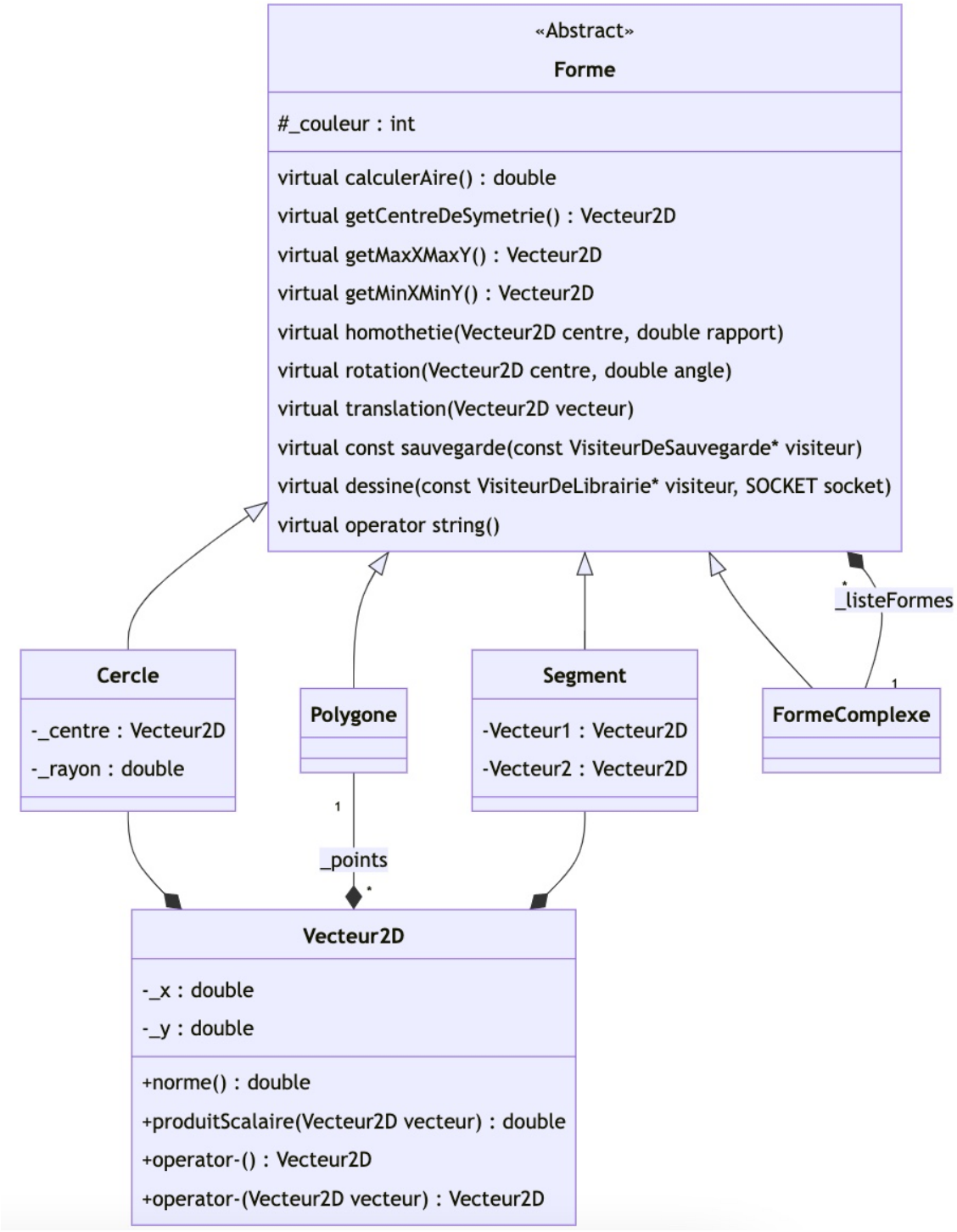
En conclusion, la réalisation de ce projet en binôme a été une expérience très enrichissante pour nous. Nous avons non seulement réussi à former un binôme efficace et autonome, mais également à bien nous répartir les tâches en fonction de nos compétences et de nos intérêts respectifs. Cette organisation nous a permis de progresser rapidement et d'obtenir un résultat satisfaisant en peu de temps.

Nous sommes fiers d'avoir pu mener à bien ce, ce qui témoigne de notre capacité à travailler en équipe et à prendre des décisions ensemble. Nous avons également apprécié la mise en pratique des différents concepts appris en matière de développement de logiciels et la mise en œuvre de trois design patterns différents.

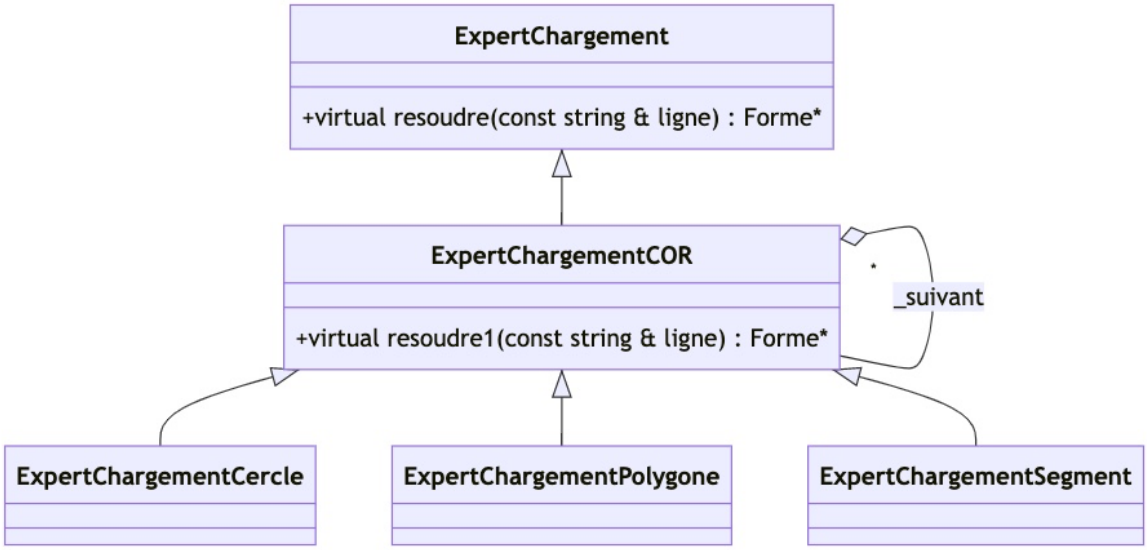
Au-delà de l'aspect technique, cette expérience nous a également permis de renforcer notre relation de travail et de mieux comprendre les compétences de chacun. Nous sommes donc convaincus que cette expérience nous sera bénéfique dans nos futurs projets professionnels.

ANNEXES

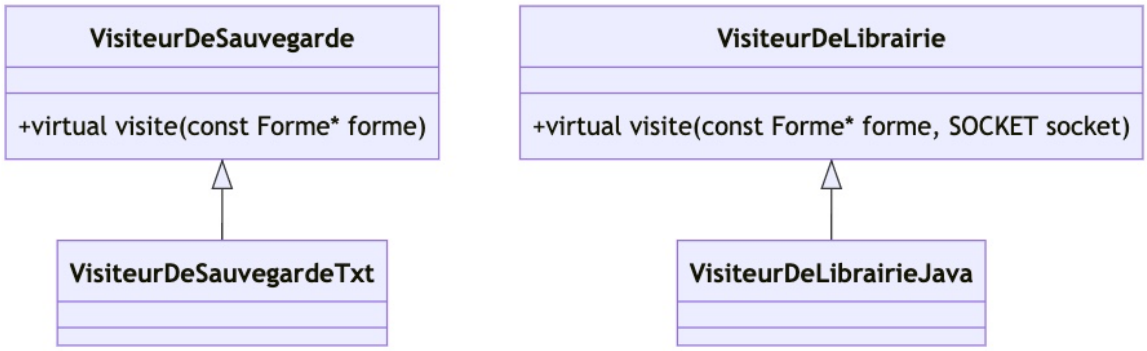
Les formes



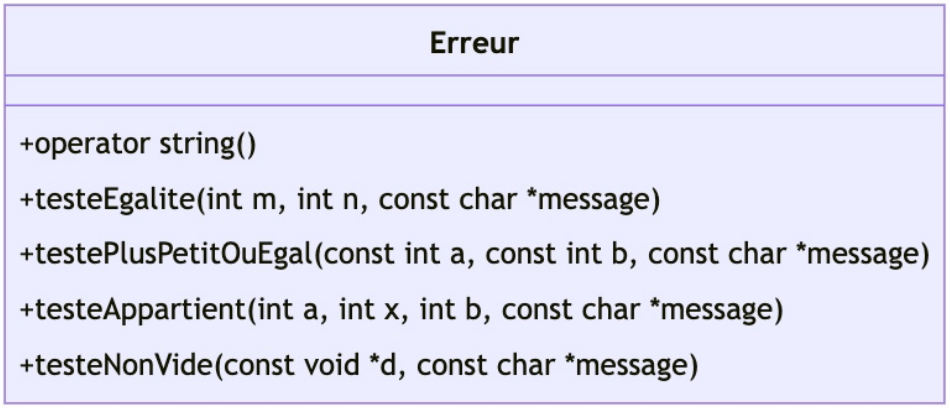
Les Experts

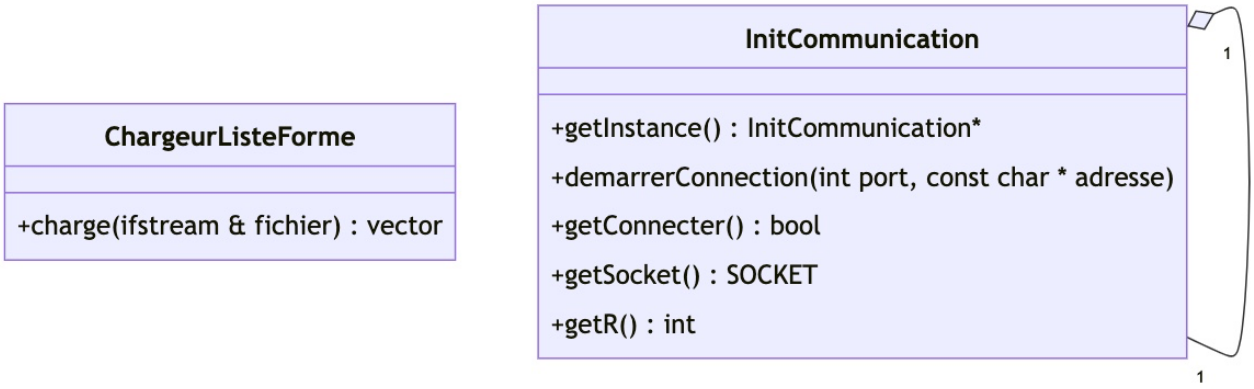


Les Visiteurs



Autres classes




ChargeurListeForme
+charge(ifstream & fichier) : vector
InitCommunication
+getInstance() : InitCommunication*
+demarrerConnection(int port, const char * adresse)
+getConnecter() : bool
+getSocket() : SOCKET
+getR() : int
1
1