Mai 2020

Projet Chathack
Manuel développeur

Network programming

AGULLO Vincent
CRETE Jonathan

Filière Informatique

Sommaire

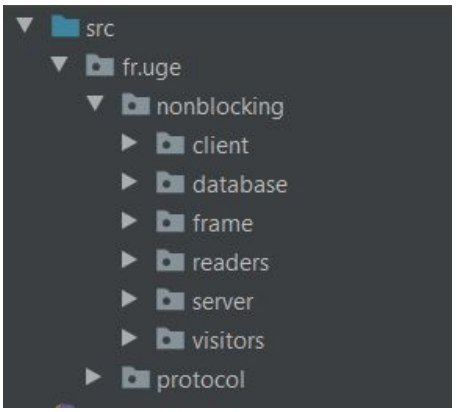
Introduction

Le but du projet ChatHack est de réaliser un service de discussions et d'échanges de fichiers. Le principe de l'application est qu'un client puisse s'authentifier auprès d'un serveur avec un login et un mot de passe ou qu'il puisse s'authentifier anonymement uniquement avec un login. De plus ce projet doit permettre les échanges de messages privées ainsi que des fichiers.
Ce document décrit comment le projet a été réalisé dans son ensemble.

I. Elaboration et choix d'architecture

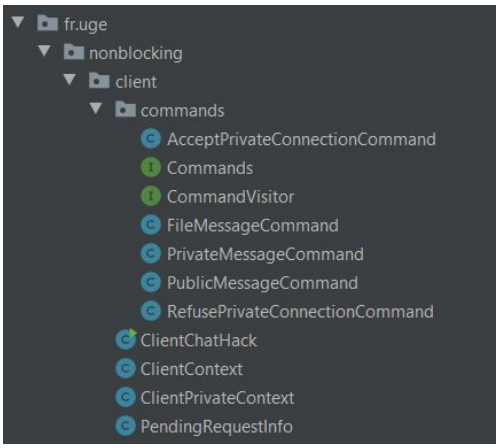
Tout d'abord voici un bref aperçu du projet.
Le projet a été architecturé en 7 packages différents :

- Le package client contient toutes les informations concernant le client (le client, les contexts public privées du client ainsi que les commandes)
- Le package database contient les informations de la base de données (les requêtes et les réponses a envoyé à la base de donnée)
- Le package frame contient toutes frames qui ont lieux lors des échanges.
- Le package readers contient les readers dit "basic" (pour certains types comme des long, des integer etc) et un "Séquentiel reader" pour pouvoir lire de manière séquentielle n'importe quel type de trame.
- Le package server contient toutes les informations concernant le serveur (le serveur, le contexte avec le client et le contexte avec la base de donnée)
- Le package visitors implémentant les différents visiteurs du projet (Design pattern visitor)
- Un package protocole répertoriant les différentes opérandes qui sont utilisaient lors des échanges de trames dans le programme.



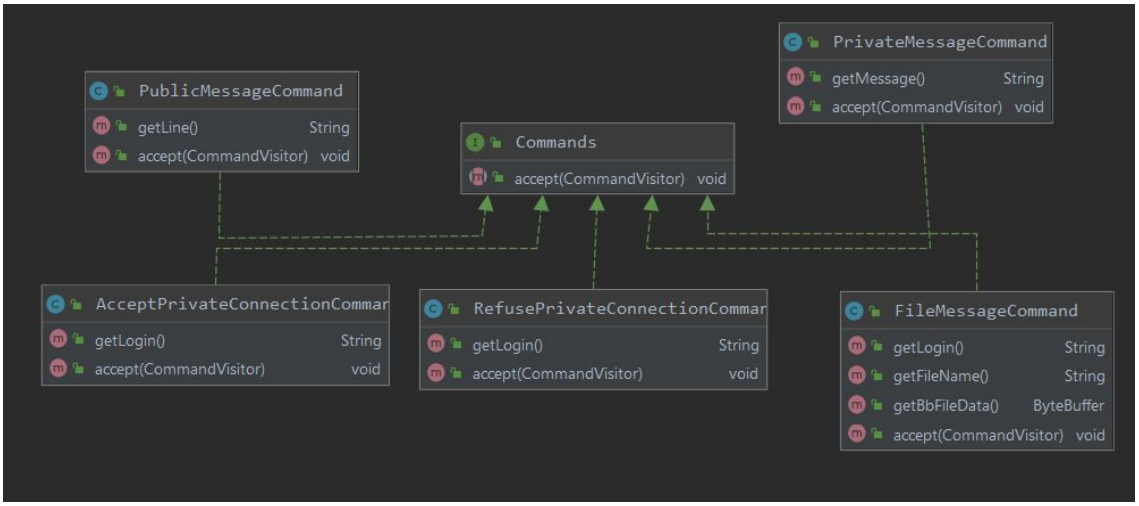
a. Client

Dans le package client nous retrouvons :



- Les commandes que peut réaliser le client. Les commandes implémente le design pattern des Visitor pour savoir quelles commandes le client est en train d'effectuer.
- En fonction de la commande souhaité le client 'visit' (appelle) la bonne commande : par exemple si le client souhaite envoyer un message public, le visiteur passera dans la classe "PublicMessageCommand".
- Depuis cette classe on peut récupérer un certain nombre d'informations comme le message inscrit en ligne de commande

Voici un schéma plus complet détaillant les différentes commandes :

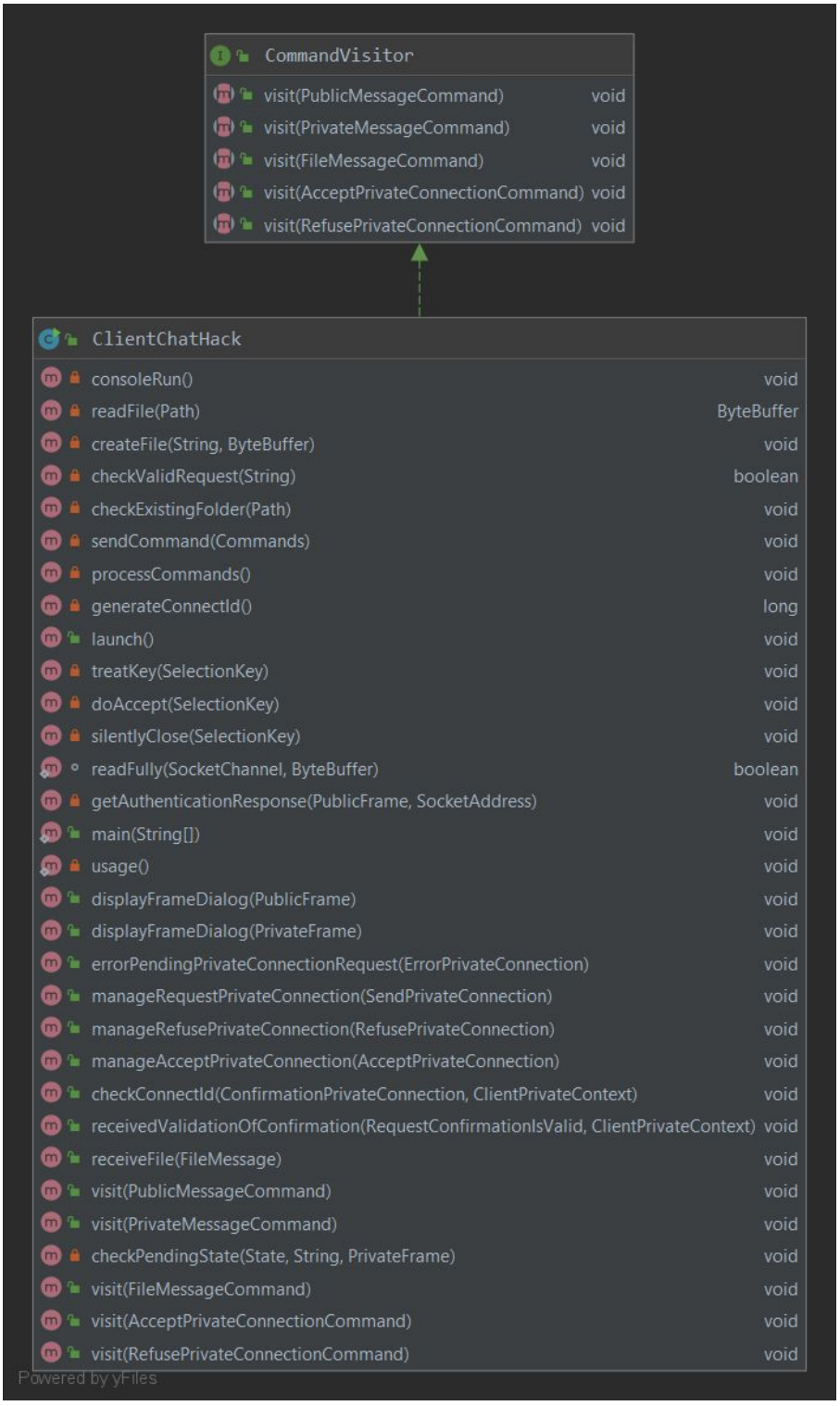


Le client possède aussi deux contextes :

- Un context dit "public" qui va permettre via ce context d'envoyer des trames auprès du serveur. Tous les échanges clients publics sont fait par ce contexte.
- Un context dit "privée" qui va permettre via ce context d'envoyer des messages privées à un autre client ainsi que des fichiers. Tous les échanges clients privées sont fait par ce contexte.

ClientChatHack

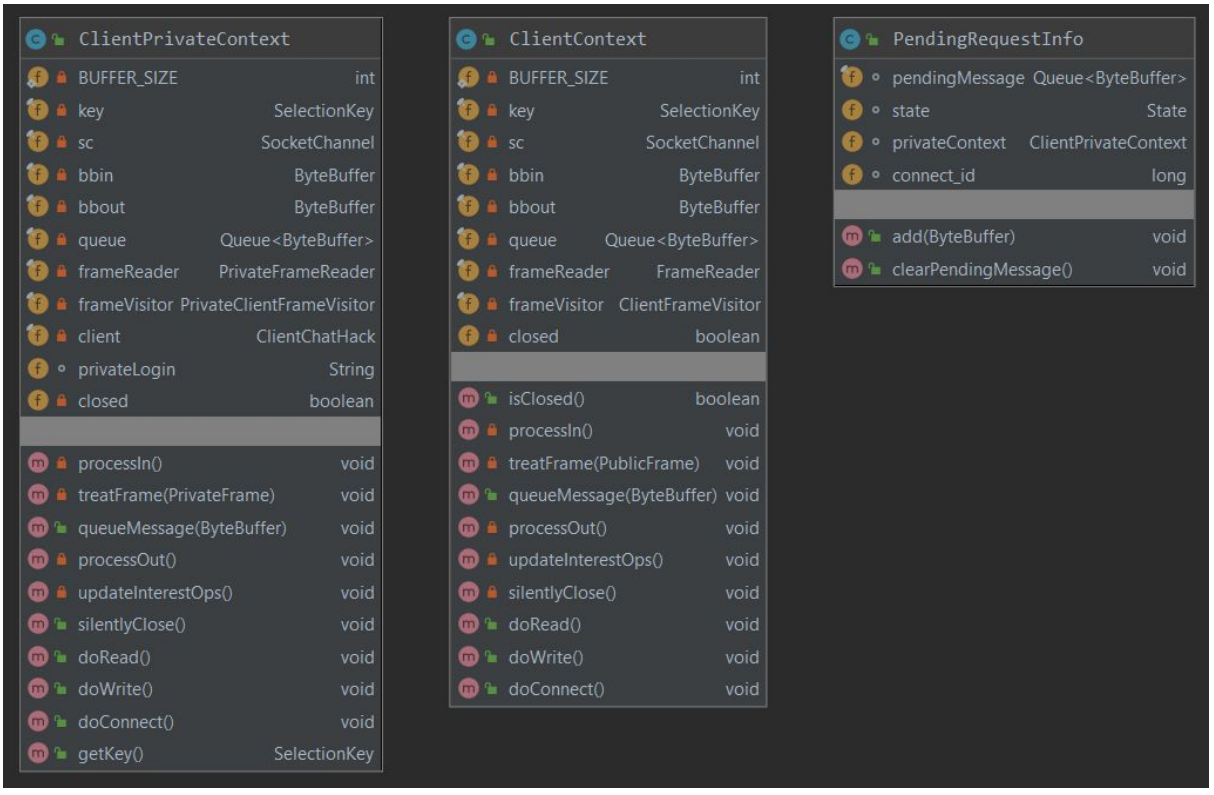
Le client est présentée ci-dessous.Celui-ci implémente l'interface **CommandVisitor** afin de visiter tous les visiteurs de commandes.



Comme dit plus haut le client possède deux contextes et une class info sur les connexion en attentes d'acceptation:

- Un contexte **ClientPrivateContext** pour les interactions privées client-client
- Un autre context **ClientContext** pour les interactions de base avec le serveur
- Une classe PendingInfo, qui est attaché à un login pour savoir l'état de la connexion avec un autre client. Chaque client a une map de clients avec lesquels on associe un objet PendingInfo.

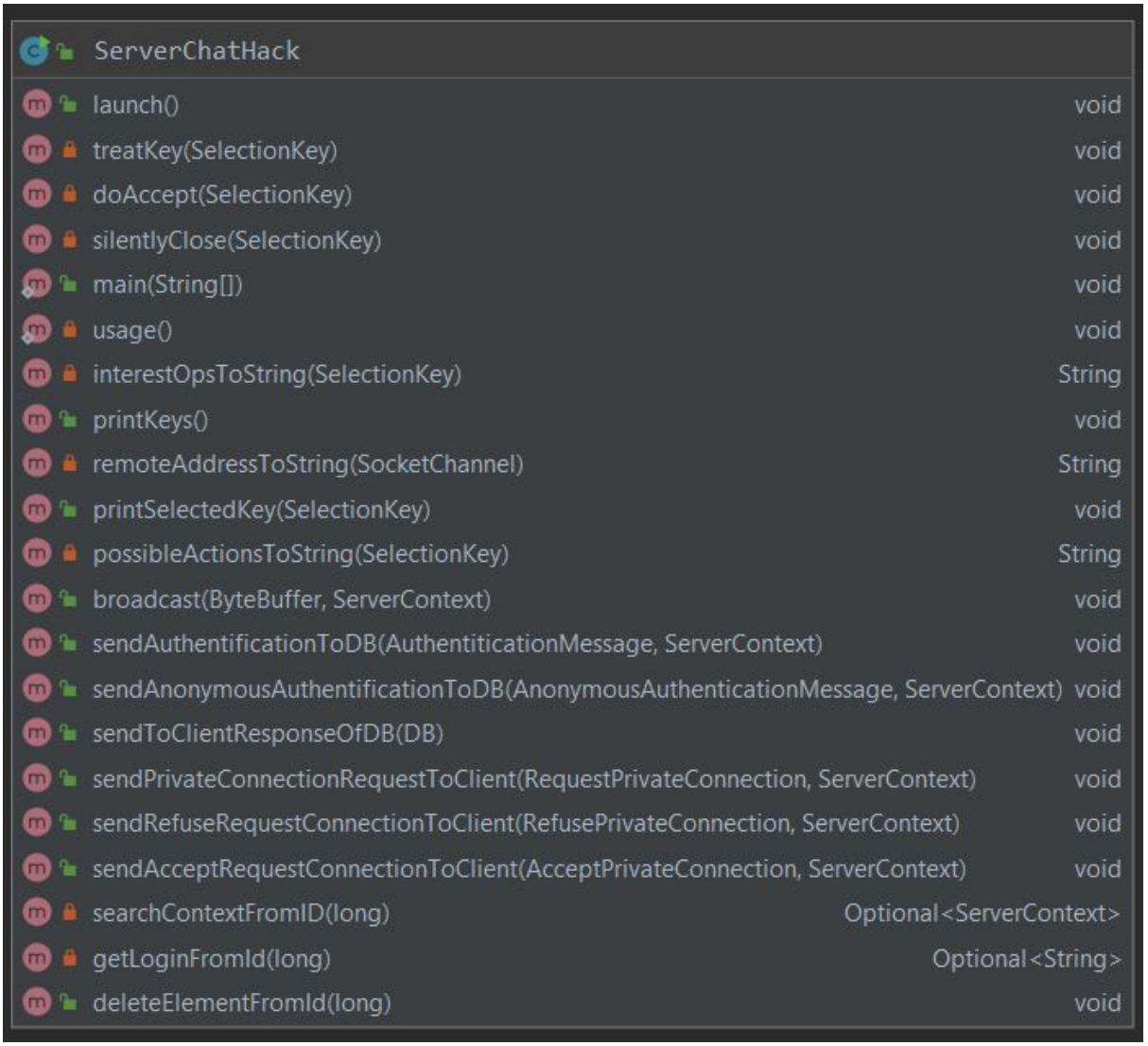
Voici comment se présente ces classes :



b. Serveur

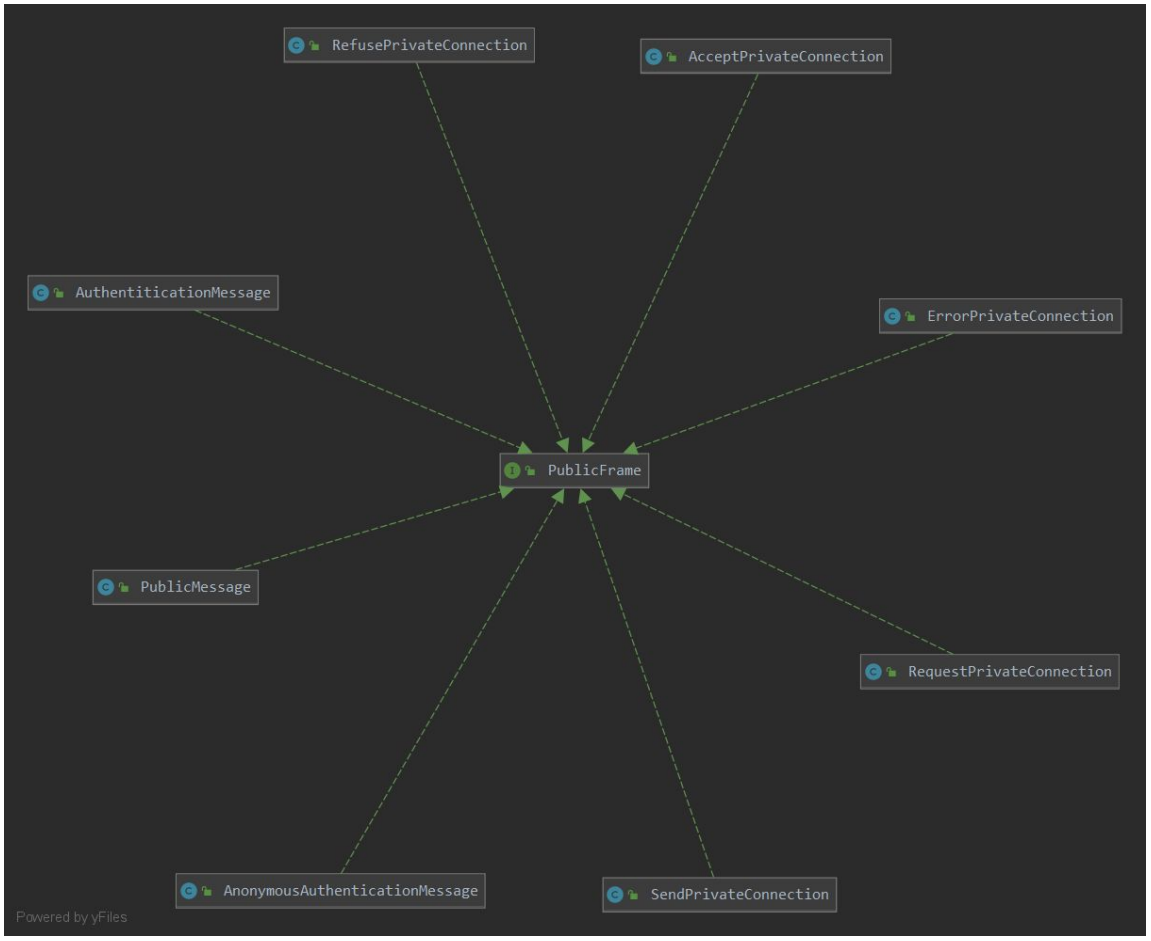
Pour le serveur ChatHack nous avons deux contextes :
→ La classe **DBContext** est le context qui sert pour l'interaction avec la base de données
→ La classe **ServerContext** est le context qui sert pour l'interaction avec les clients. Plus précisément, ce context permet de gérer l'authentification des clients, l'envoie des messages publiques, les échanges entre les demandes de connexions privées entre deux.

Le serveur est représentée par la classe ServerChatHack ci-dessous :



c. Trames

Comme nous le verrons pour les visiteurs, tout en suivant notre RFC, nous avons deux types de trames, à savoir les trames publiques représentées par l'interface **PublicFrame** et privées représentées par l'interface **PrivateFrame.** Toutes les classes de trames implémentent leurs types communs.



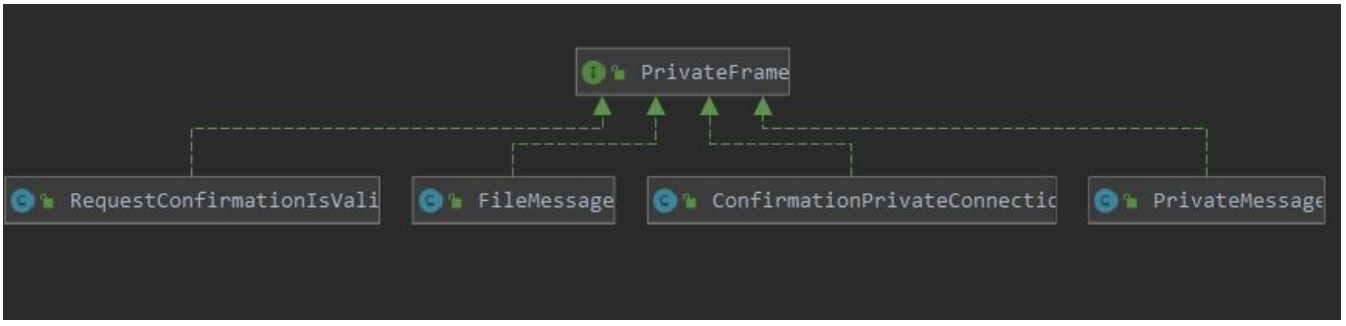
Détaillons les trames publiques :

→ La classe **PublicMessage** représente la trame d'envoi d'un message public entre tous les clients connectés , de code opérande **14**.
→ La classe **AuthentificationMessage** représente la trame d'authentification envoyé par le client avec mot de passe, de code opérande **10**.
→ La classe **AuthentificationMessage** représente la trame d'authentification envoyé par le client, sans mot de passe de code opérande **12.**
→ La classe **RequestPrivateConnection** représente la trame de demande de connexion privée envoyé par une client, avec le code opérande **15.**
→ La classe **SendPrivateConnection** représente la trame d'envoi de la demande de connexion privée, avec le code opérande **16.**

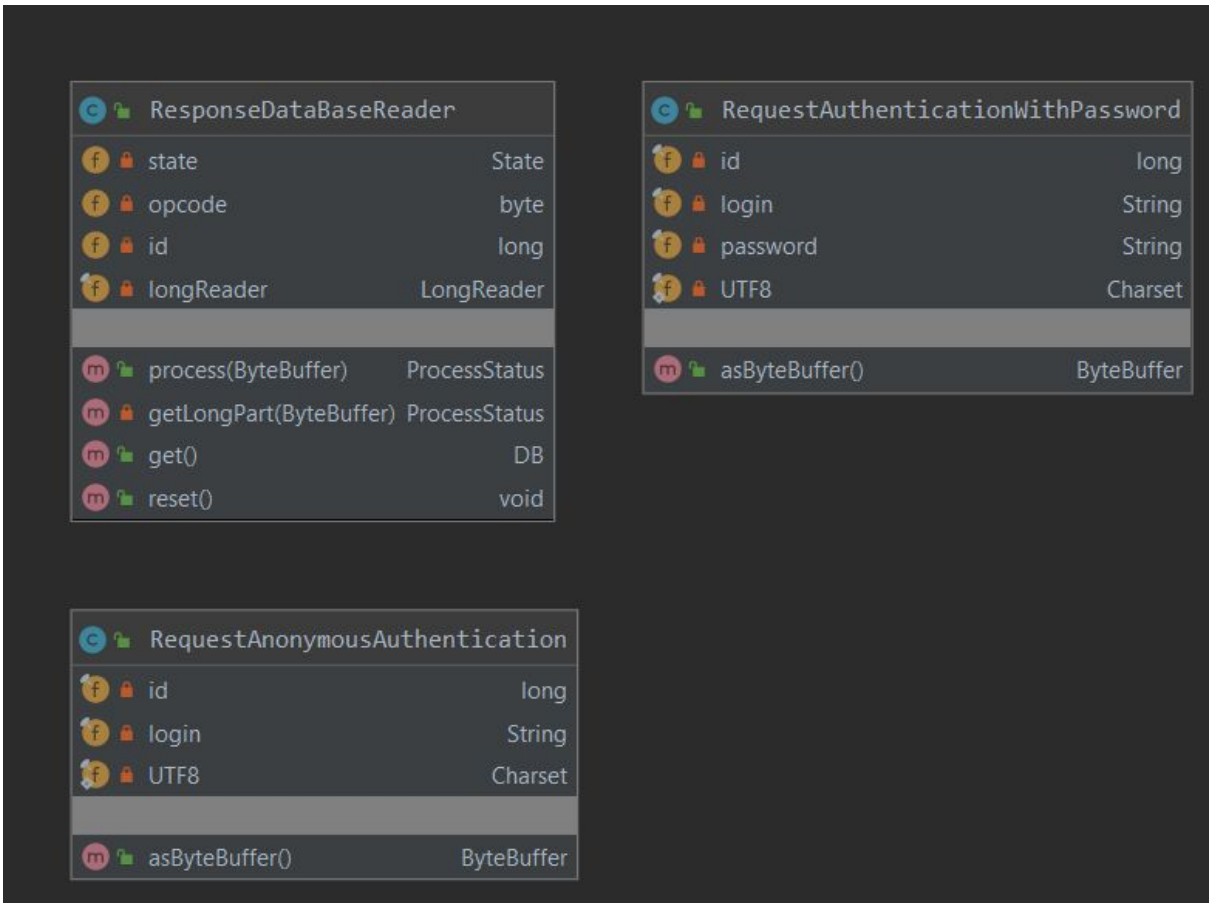
→ La classe **AcceptPrivateConnection** représente la trame d'acceptation de la demande de connexion privée, renvoyée par le client, avec le code opérande **18.**
→ La classe **RefusePrivateConnection** représente la trame de refus de la demande de connexion privée, renvoyée par le client, avec le code opérande **19.**
→ La classe **ErrorPrivateConnection** représente la trame d'erreur de la demande de connexion privée client-client

Détaillons les trames privées :

→ La classe **RequestConfirmationIsValid** représente la trame de vérification de l'ID de connexion avec le code opérande **21.**
→ La classe **FileMessage** représente la trame de d'envoi d'un fichier privé entre deux client connectés, avec le code opérande **22.**
→ La classe **ConfirmationPrivateConnection** représente la trame de confirmation de l'établissement de la connexion privée, avec le code opérande **20**.
→ La classe **PrivateMessage** représente la trame d'envoi d'un message privé entre deux clients connectés, avec le code opérande **23**



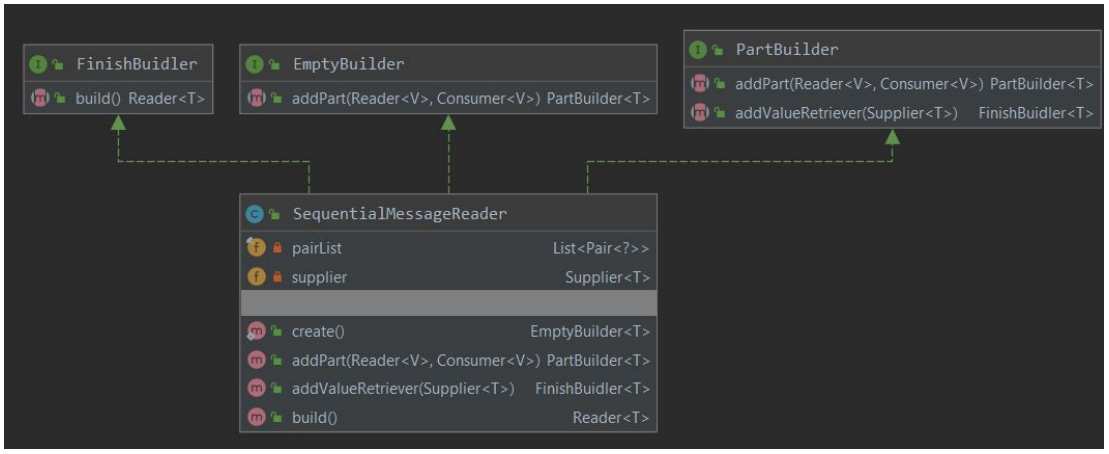
d. Base de données

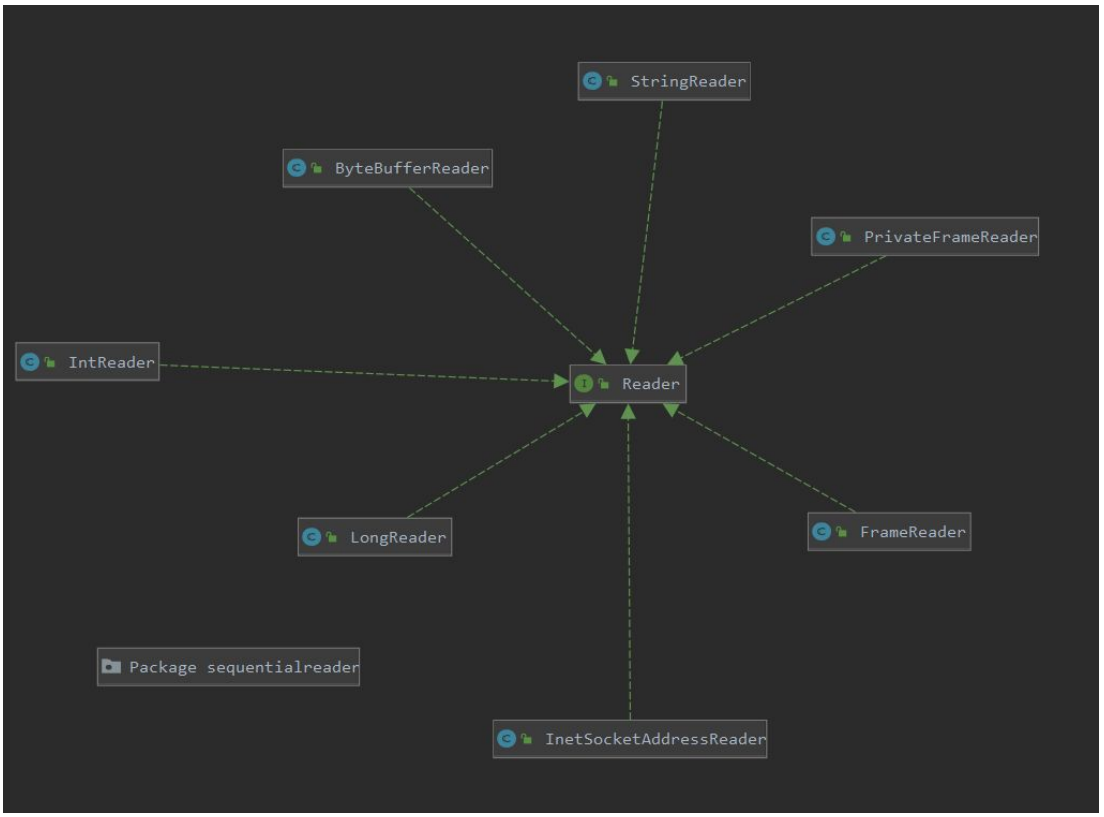


Pour la base de donnée, nous avons développé un reader nommé **ResponseDataBaseReader** ayant pour but d'interpréter la réponse à la requête envoyé à la base de donnée. La classe **RequestAuthentificationWithPassword** représente la trame envoyée par le serveur (requête) à la base de donnée afin de vérifier l'identité d'un client qui cherche à se connecter avec un *login + mot de passe.* L'autre classe **RequestAnonymousAuthentification** représente la trame envoyée par le serveur pour tenter une authentification anonyme d'un client.

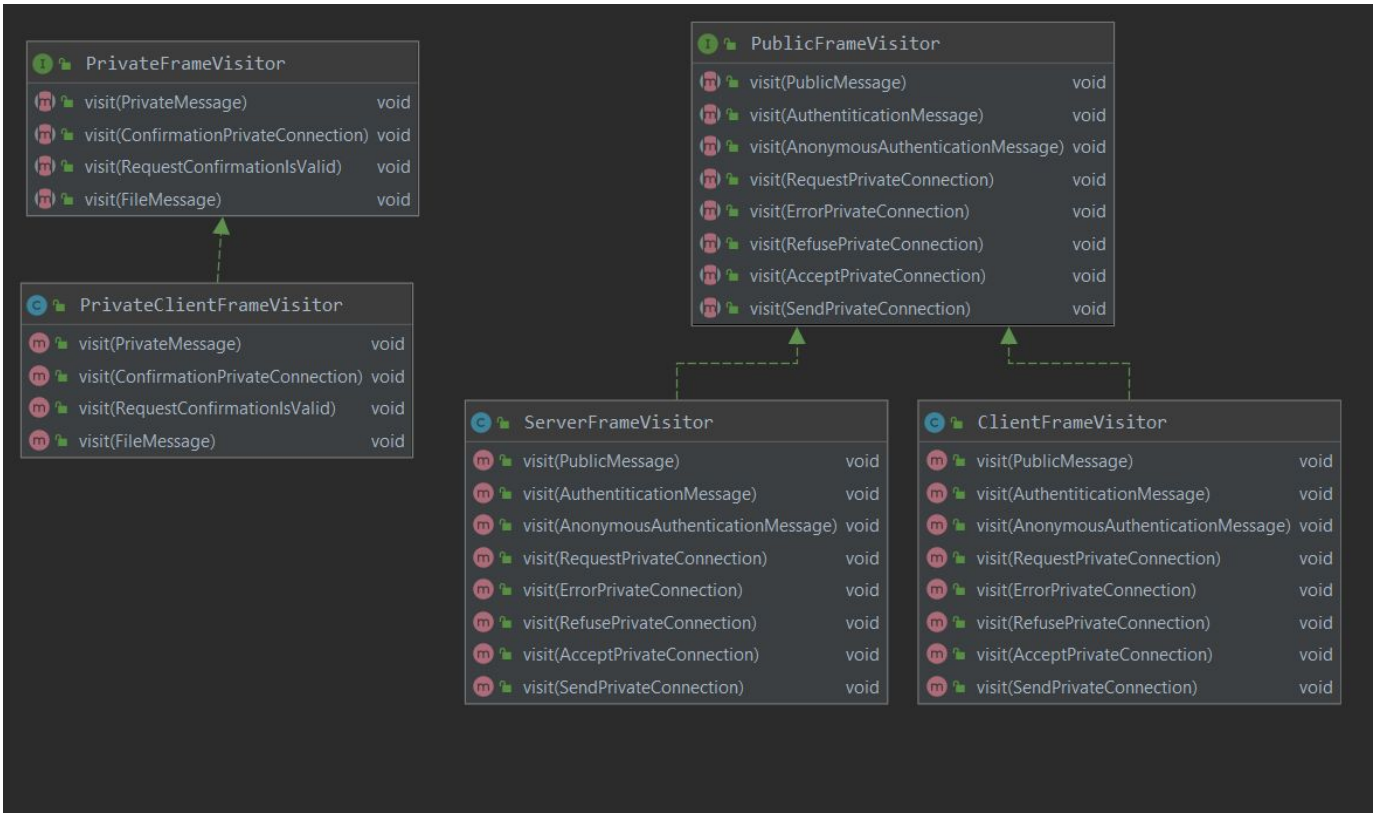
e. Readers

Lors de nos premières versions du programme nous avions beaucoup de classe reader pour lire les trames. Nous avons changé ce procédé pour avoir un reader séquentielle generic , l'idée est qu'on alimente le sequential reader pour créer la trame voulu. Le reader va lire séquentiellement les readers ajouté pour construire la trame. Ceci nous a permis d'enlever une dizaine de fichiers et d'avoir un code un plus lisible.





f. Visiteurs



Nous avons deux types de visiteurs. Le/les visiteurs pour des interactions client-client privés sont représentées par l'interface **PrivateFrameVisitor** et des visiteurs pour des interactions publiques client/serveur sont représentées par l'interface **PublicFrameVisitor.** Les visiteurs sont tous hiérarchisés : Ainsi nous avons un **ServerFrameVisitor** pour le serveur afin d'assurer l'envoi d'un message public à tous les clients, demande d'authentification avec mot de passe ou anonyme… Nous avons de même un **ClientFrameVisitor** pour gérer l'envoi de messages privés, demande de connexion privée, etc...

II. Etat des fonctionnalités

a. Ce qui fonctionne

- Authentification d'un client avec un pseudonyme et un mot de passe
  - Acceptation d'une demande de connexion privée
  - Refus d'une demande de connexion privée
- Authentification d'un client avec un pseudonyme uniquement
- Envoi de messages publiques entre tous les clients connectés
- Réception d'un message public
- Négociation d'une connexion privée côté client et serveur
  - Acceptation d'une demande de connexion privée
  - Refus d'une demande de connexion privée
- Envoi de messages privés entre deux clients connectés
- Réception de messages privés entre deux clients connectés
- Envoi de fichiers privés entre deux clients connectés
- Réception de fichiers privés entre deux clients connectés

b. Ce qui ne fonctionne pas

Toutes les fonctionnalités demandées ont été réalisées.

IV. Difficultés rencontrées au cours du développement

- La négociation de connexion privée nous as posé soucis :

Nous nous embrouillons assez facilement si on était pas assez attentif.
Par exemple nous avons eu du mal à identifier de quel client on parle. Est-ce que c'est le client qui a fait la demande ou est-ce le client qui reçoit la demande.

- Le Séquentiel Reader Generic :

L'idée avait bien été comprise mais on a eu des difficultées lorsqu'il fallait savoir les types généric de nos méthodes.

V. Évolutions après soutenance bêta

- Implémentation du design pattern visitors pour la gestion des types de commandes dans le client. Suppression du switch/case dans la méthode *processCommand()*
- Une classe avec un reader séquentiel générique permettant de créer des readers séquentiellement et plus efficacement
- Vérification que les clients sont bien connectées avant de pouvoir faire des requêtes
- L'authentification est faite en mode bloquant, ce qui fait qu'on attend la réponse du serveur avant de pouvoir faire des commandes, une fois la connexion établie on passe en non bloquant.
- Vérification de l'identité d'un client lors d'une demande de connexion privée.
- Les échanges de messages privées entre deux clients sont fonctionnelles (*ce qui ne l'était pas avant la soutenance bêta)*
- Les échanges de fichiers entre deux clients sont fonctionnelles (*ce qui ne l'était pas avant la soutenance bêta)*
- Il a deux interfaces frame : FramePublic pour les trames publiques et FramePrivates pour gérer les messages privées.