T. CALONNE
T. JUILLARD

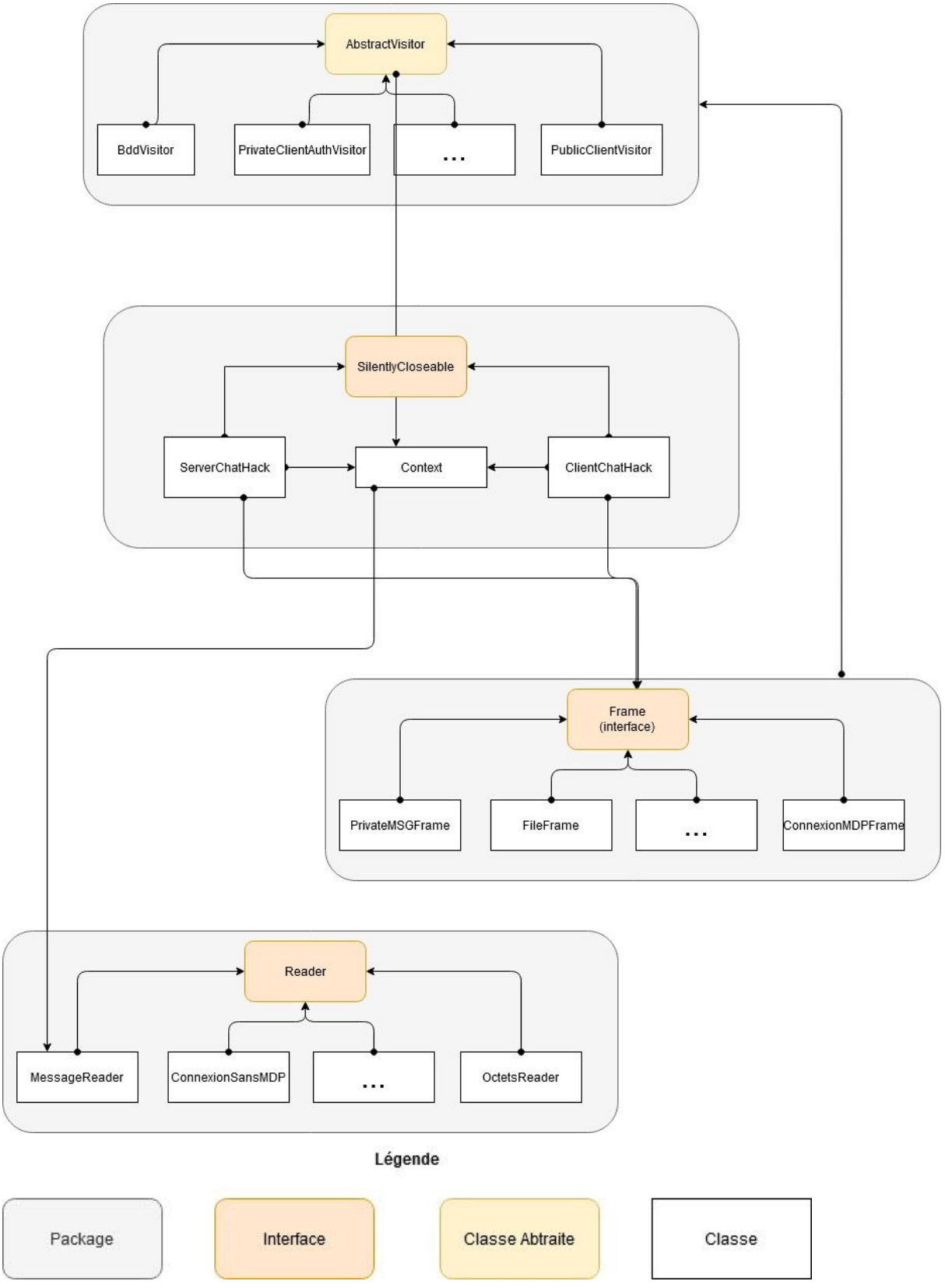
26 avril 2020

Manuel développeur

Cette documentation présente rapidement les choix d'architecture que vous avez fait, des schémas, les choses qui fonctionnent et celles qui ne fonctionnent pas dans la version livrée, les difficultés que vous avez rencontrées. Une section de ce manuel sera explicitement dédiée aux évolutions demandées lors de la soutenance bêta et à la manière dont vous les avez traitées.

I. **C**hoix d'architecture

Voici un schéma de notre architecture.



Nous voyons que nous avons quatre packages. Un package avec les *Frames*, c'est à dire avec toutes les trames que nous pouvons écrire ou décoder. Elles implémentent toutes l'interface *Frame* qui possède une méthode *accept* qui permet d'utiliser un certain visitor attaché à un context. Chaque frame possède aussi une méthode *asByteBuffer* qui permet de transformer l'objet Frame en un ByteBuffer.

Puis nous avons donc le package *Reader* qui permet de décoder un ByteBuffer. La seule classe qui est appelée depuis l'extérieur du package est *MessageReader*. Nous devons donc passer par cette classe qui s'occupera de trouver la bonne Frame. Cette classe sert donc de passerelle avec l'extérieur, elle permet de savoir où en est le décodage de la trame et aussi de récupérer l'objet Frame associé à la trame décodée.

Ensuite nous avons le package Visitor qui s'occupe de "visiter" les *Frame*. Chaque visitor possède des *visit* différent et a donc une utilisation différente. Chaque visitor correspond à un certain état du client ou du server. Il occupe plusieurs rôles dont la sécurité et permet de pouvoir faire un "switch" sur un objet, un fois implémenté il est facile de le maintenir et de la faire évoluer.

Enfin nous avons le package principal avec le Client, le Server, le Context et une interface *SilentlyCloseable* avec une seule méthode *SilentlyClose(SelectionKey)* pour le Context puisse fermer une Key de façon transparente sans qu'il sache si la Key appartient au Server ou au Client.

II. **L**es améliorations possible

Ce code est simple à maintenir et à faire évoluer grâce à ses nombreuses abstractions. Si vous voulez changer ou rajouter une trame vous pouvez le faire depuis la package Frame et Reader sans toucher aux autres package, le code est fermé. Si vous voulez rajouter un visitor pour simuler un nouvel état dans lequel se trouvera l'utilisateur vous pour le rajouter dans le package Visitor en créant juste une nouvelle classe qui étend de *AbstractVisitor* en surchargeant juste les méthodes dont vous avez besoin car les autres existent déjà et appellent par défaut *SilentlyClose*. Cette protection a été faite pour que vous ne vous trouviez pas dans un état incohérent. Le code est donc fermé ici aussi. Pour le dernier package, Vous pouvez rajouter des méthodes au context si vous en avez besoin ainsi qu'au Client et au Server.

III. **L**es possibilitées d'utilisation

Avec ces deux jars, vous avez la possibilité de converser en publique, de faire de demande connexion privée, d'envoyer des messages privés et des fichiers aux utilisateurs qui ont accepté votre demande d'amis. Lorsqu'un utilisateur se connecte sans la base de donnée, aucun autre utilisateur ne peut se connecter avec le même pseudo. Mais lorsque cet utilisateur se déconnecte ce pseudo est de nouveau libre et un autre utilisateur peut se connecter avec ce pseudo.

Aucun utilisateur peut se connecter sans mot de passe avec un pseudo utilisé dans la base de donnée et il peut y avoir que un seul utilisateur connecté en mode connexion base de donnée pour un seul couple pseudo/motDePasse.

Lorsque deux utilisateurs se parlent de manière privée et que l'un des deux se déconnecte, l'autre utilisateur sera informé de la déconnexion de son amis et ne pourra plus lui envoyer de messages privés. Mais si un autre utilisateur se connecte avec le même nom alors il pourra lui faire une demande de connexion privée et puis converser avec lui s'il accepte.

L'envoie de fichier fonctionne correctement si le fichier est plus petit de MAX_INTEGER * 512.

IV. **L**es points négatifs

Si un fichier est très volumineux, il prendra du temps à s'envoyer et pourra bloquer l'envoie des autres messages privés à cet utilisateur. Mais il ne bloquera pas la réception des messages et l'envoie d'autres messages privés ou publiques aux autres utilisateurs.

V. **L**es améliorations depuis la soutenance

Lors de la soutenance bêta, de nombreuses améliorations ont été mentionné. Voici une liste de ces remarques et comment nous avons géré ces remarques:

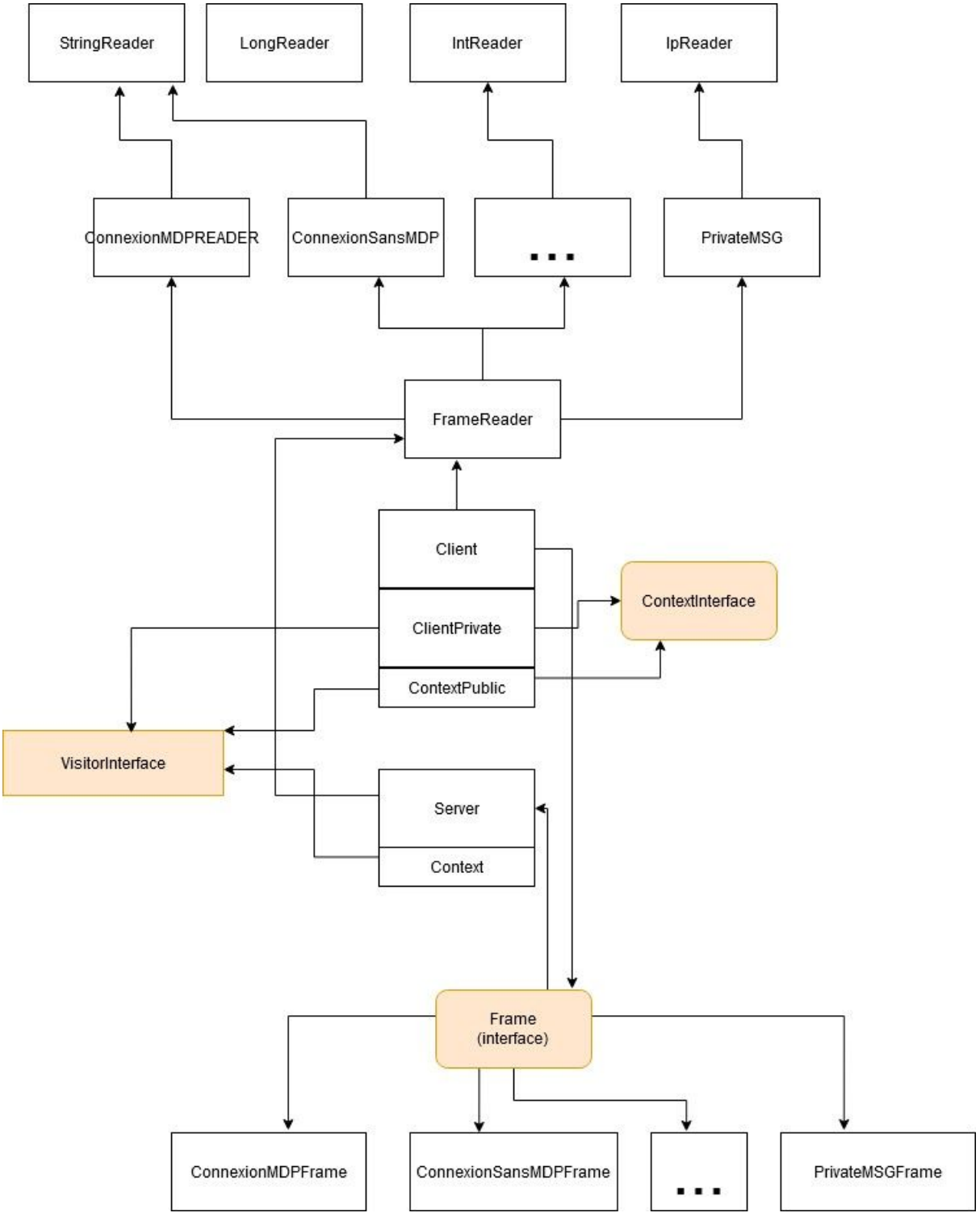
- Rendre le code plus objet
    - Nous avons créé une méthode asByteBuffer dans chaque frame.
    - Les contexts ont été mis dans des classes à part, puis nous les avons regroupé en un seul context.
    - Nous avons mis de AssertionError() lorsqu'on était dans un état incohérent
    - Nous avons encapsulé les authentifications dans les contexts
    - Nous avons mis le wakeup() dans la boucle de sélection alors qu'avant il était dans le server
    - Nous avons créé plusieurs visitor et nous les attachons au context en fonction de l'état dans lequel se trouve le context de l'utilisateur (il peut avoir plusieurs contextes et plusieurs visitors)
- Ne plus faire de instanceof() dans le broadcast
    - Nous avons sauvegardé la key qui sert à sauvegarder le contexte du ServerMdp pour que lorsque nous faisons un broadcast, nous ne mettons le message que dans queue des contextes qui correspondent aux clients. Comme nous avons sauvegardé la key qui contient le contexte qui communique avec la base de donnée, nous faisons juste une simple comparaison avec "==".

- Dans la méthode ProcessOut nous utilisons position() et non capacity()
    - Nous l'avons changé car nous n'avions pas pensé aux utilisateurs mal intentionnés qui nous envoient des trames et où position() est différent de capacity().
- Vérifier de personne ne peut envoyer de messages en se faisant passer pour quelqu'un d'autre ou juste pouvoir parler avec quelqu'un sans y être autorisé
    - Nous avons plusieurs visitor en fonction de l'état ou se trouve l'utilisateur, au début il peut juste se connecter et s'il envoie une autre trame, le serveur fermera directement la connexion.
    - A chaque fois qu'un utilisateur envoie un message par le serveur, son identité est vérifiée.
    - Lorsqu'un client se transforme en serveur, il y a aussi un système de visitor qui accepte que l'authentification, et une fois l'authentification réussi on change le visitor pour qu'il est le droit de converser, d'envoyer des messages privés, des fichiers et le droits de les décoder.
    - Si un utilisateur B envoie une trame de connexion acceptée à un utilisateur A qui attend la réponse d'un utilisateur C alors A ignore la réponse de B.
- Lorsqu'on attend la réponse de la base de donnée au serveur quand quelqu'un veut se connecter, il faut cancel la key
    - Nous avons cancel la key car elle ne pouvait être ni en OP_READ ni en OP_WRITE et on re register lorsqu'il y a la réponse de la base de donnée.
    - Nous l'enregistrons de nouveau dans le selector une fois que le serveur reçoit la réponse de la base de donnée
- Nous mettons les messages dans la queue d'un context et ne pas l'envoyer si le context n'est pas prêt à être utilisé (par exemple lorsqu'on fait une demande de connexion privée en envoyant un message ou un fichier)
    - Nous avons mis un champs boolean dans le context qui permet de passer les messages de la queue dans le ByteBuffer *bbin* si le boolean est vrai.
- Se connecter en non bloquant d'un client à l'autre
    - Avant nous nous connectons en mode bloquant mais cela peut prendre du temps et même bloquer tout notre client. Nous le faisons donc en mode non-bloquant maintenant.

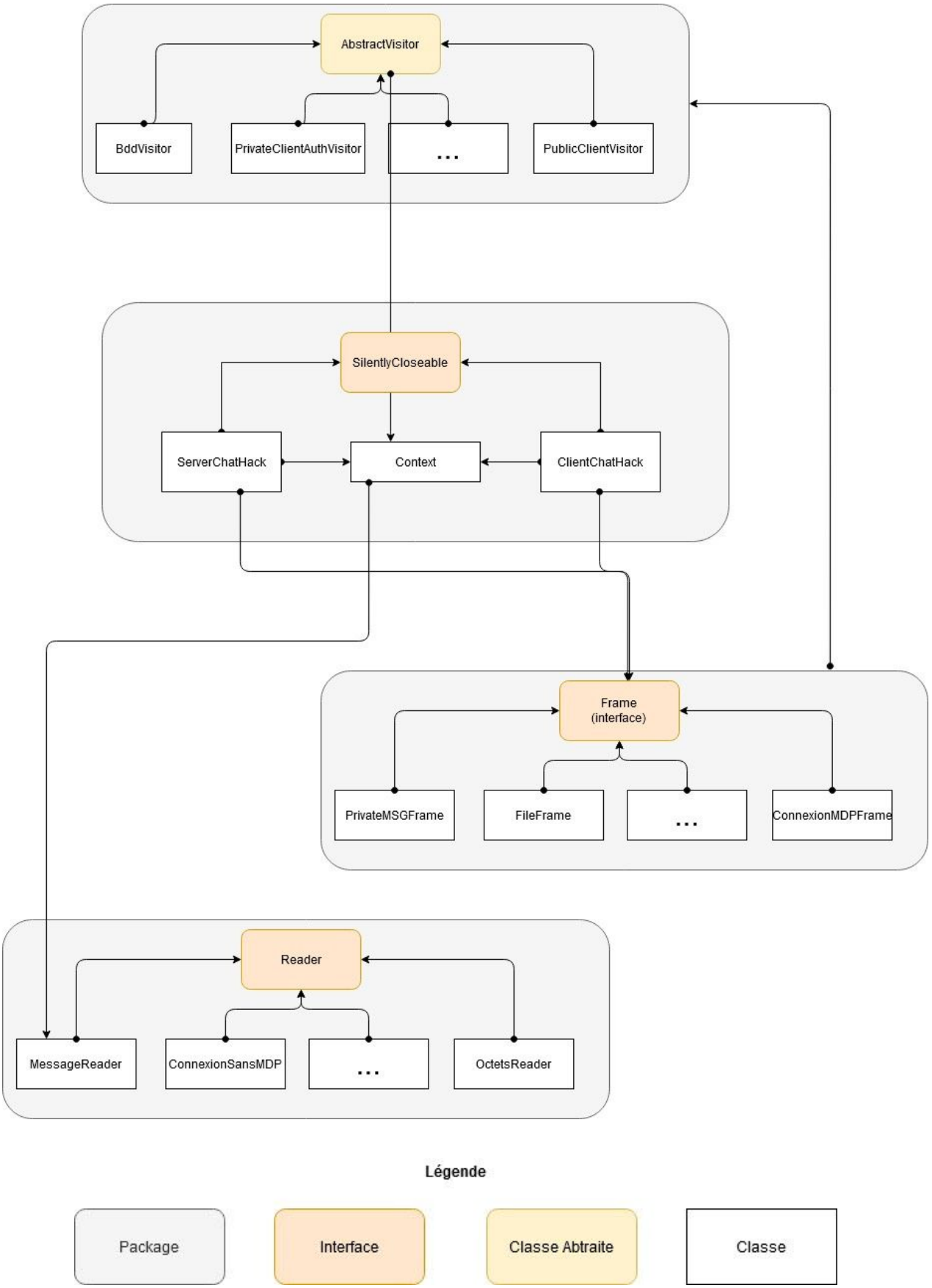
Pour résumer, les plus gros points qui étaient à améliorer étaient: Faire des contextes propres, respecter la philosophie objet et faire un open en non-bloquant pour une connexion privée entre clients. Nous pensons avoir réussi chaque tâche qui nous étaient données. Pour vous montrer le changement qu'il y a eu voici deux schémas qui sont notre ancienne et notre nouvelle architecture.

Nous voyons donc maintenant que chaque package possède une tâche spécifique et c'est donc plus facile à maintenir et faire évoluer. Nous voyons aussi que les relations entres les différents packages sont limitées. Nous privilégions l'abstraction, c'est à dire que les classes de couches supérieures dépendent pas des couches inférieures, mais des abstractions. Les classes des couches inférieures implémentent donc ces abstractions.

Ancienne Architecture



Nouvelle architecture



VI. **D**ifficultées rencontrées

Nous n'avons pas beaucoup eu de difficultées en général pour coder car nous avions bien réfléchi à la structure du projet et les différentes trames. Ce qui a été compliqué c'est de bien comprendre la structure du projet, de faire en sorte qu'elle soit évolutive. De plus, il fallait penser à un moyen de protéger le serveur des attaques extérieures, c'est à dire de que si un client ou le serveur reçoit une trame incohérente, ou un pirate qui envoi une trame au serveur alors qu'il n'a pas encore les droits.

De plus nous avons rencontrés de nombreuses incohérences entre le monde Linux et le monde Windows. Nous avons dû tester régulièrement sur ces deux environnements et adapter notre code.

Nous avons aussi eu quelques difficultées pour tester en réseau local sur deux machines différentes.

Enfin nous avons dû factoriser plusieurs classes après la soutenance bêta.