Rapport d'activité
Java Clavardage

Maud Pennetier et Damien Molina

14 février 2021

Table des matières

Introduction

Le présent document est un rapport d'activité concernant le projet d'application bureau de Clavardage développée en Java dans le cadre de l'UF COO/POO de quatrième année à l'INSA Toulouse. Ce document a été réalisé par Maud Pennetier (4IR - SC) et Damien Molina (4IR - SI).

Contexte Un client contacte notre équipe de développement pour développer une application de clavardage pour son entreprise. Certains de ses employés sont connectés sur un réseau interne à l'entreprise, d'autres en dehors de ce réseau (depuis leur domicile par exemple). L'application sera développée en Java.

Mots clés Dans le présent rapport, nous utiliserons l'appellation *système* pour définir l'ensemble des applications développées pour le client : l'application bureau et le serveur Tomcat.

L'ensemble du code développé est disponible sur un répertoire Git hébergé sur la plateforme GitHub accessible à l'adresse https://github.com/Pythagus/java-messenger. Les diagrammes UML présentés dans ce rapport sont également accessibles dans le dossier **/docs/modeling/**.

Chapitre 1

Préparation du travail

Dans cette partie, nous nous intéressons à la préparation du travail de développement de l'application. Vous retrouverez ainsi une liste de **contraintes additionnelles** avec une liste des **livrables**, le travail de **conception** (sous forme de diagrammes d'objet UML majoritairement), des réflexions autour de l'**architecture** du projet et des **technologies utilisées**, ainsi que l'**organisation du travail** au sein de l'équipe de développement.

1.1 Contraintes

Vous retrouverez dans cette section les contraintes majeures du projet ainsi que des contraintes additionnelles fixées par l'équipe de développement. Ces nouvelles contraintes sont en accord strict avec le cahier des charges fourni par le client, que vous pouvez retrouver sur le répertoire Gît `docs/specifications.pdf`.

1.1.1 Contraintes majeures

Vous trouverez ci-dessous les contraintes majeures du système reformulées afin de mieux cerner les problèmes que l'on va rencontrer :

Décentralisation des pseudonymes L'une des plus importantes contraintes formulées dans le cahier des charges concerne la décentralisation du système pour le choix du pseudonyme de l'utilisateur. L'utilisateur souhaitant se connecter doit donc découvrir en temps réel quels utilisateurs sont connectés avec leurs pseudonymes.

Pseudonyme volatile Le choix du pseudonyme de l'utilisateur se fait indépendamment d'un système identifiant-mot de passe. A chaque connexion, l'utilisateur doit redéfinir un pseudonyme en veillant à ne pas en choisir un déjà utilisé par quelqu'un d'autre, sans système centralisé. Entre deux sessions de clavardage, un utilisateur peut donc avoir deux pseudonymes différents.

Gestion de l'historique L'historique de la conversation entre deux utilisateurs doit être automatiquement chargé lorsqu'une personne démarre (dans une nouvelle session) la conversation.

1.1.2 Contraintes additionnelles

Avant de débuter le développement, nous souhaitons ajouter de nouvelles contraintes au cahier des charges. Ces contraintes ne sont et ne doivent pas être en désaccord avec ce dernier.

Faciliter la maintenance Le travail de développement doit simplifier au mieux la maintenance du code par le client. Le code doit être correctement commenté et structuré.

Déploiement simplifié Le déploiement du système doit être documenté afin de faciliter le travail des développeur-administrateurs du client.

Configuration La configuration du système pour adapter l'application aux caractéristiques de l'infrastructure réseau du client.

1.1.3 Technologies

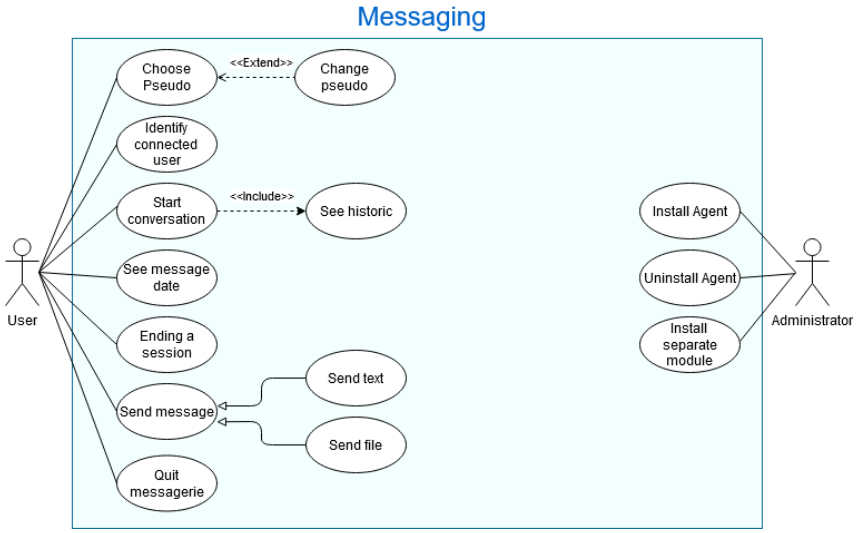
Le système a été développé en Java, langage imposé. *Java Swing* a été utilisé pour l'interface graphique du fait de sa simplicité et de sa présence native dans le langage Java. Pour communiquer avec la base de données, nous choisissons d'utiliser *MySQL*, langage que nous connaissons et simple à utiliser en Java.

1.2 Conception

Afin de préparer au mieux le développement de l'application, nous avons réalisé une étude de conception en utilisant des représentations UML.

1.2.1 Diagramme des cas d'utilisations

Le projet est représenté par le diagramme des cas d'utilisation ci-suivant. On distingue deux utilisateurs : User, la personne qui utilise le client, et Administrator, celui qui installera le client. Les messages envoyés peuvent être de deux types : du texte ou bien un fichier (image, png, pdf, etc).



1.2.2 Interactions globales

Le développement s'est fait en utilisant de très nombreuses classes, rendant plus simple leur lecture, mais réduisant la visibilité du graphe des interactions totales. Nous ne présentons donc pas un diagramme de classe général dans ce rapport (un exemplaire est tout de même présent sur le Git). En revanche, un diagramme de séquence en boite noire permet une meilleure compréhension. Ici **Actor** représente un utilisateur, **System** décrit le client d'application de ce dernier, **Network** les autres clients, **TomcatServer** le serveur de présence et **MessageServer** la base de données des messages.

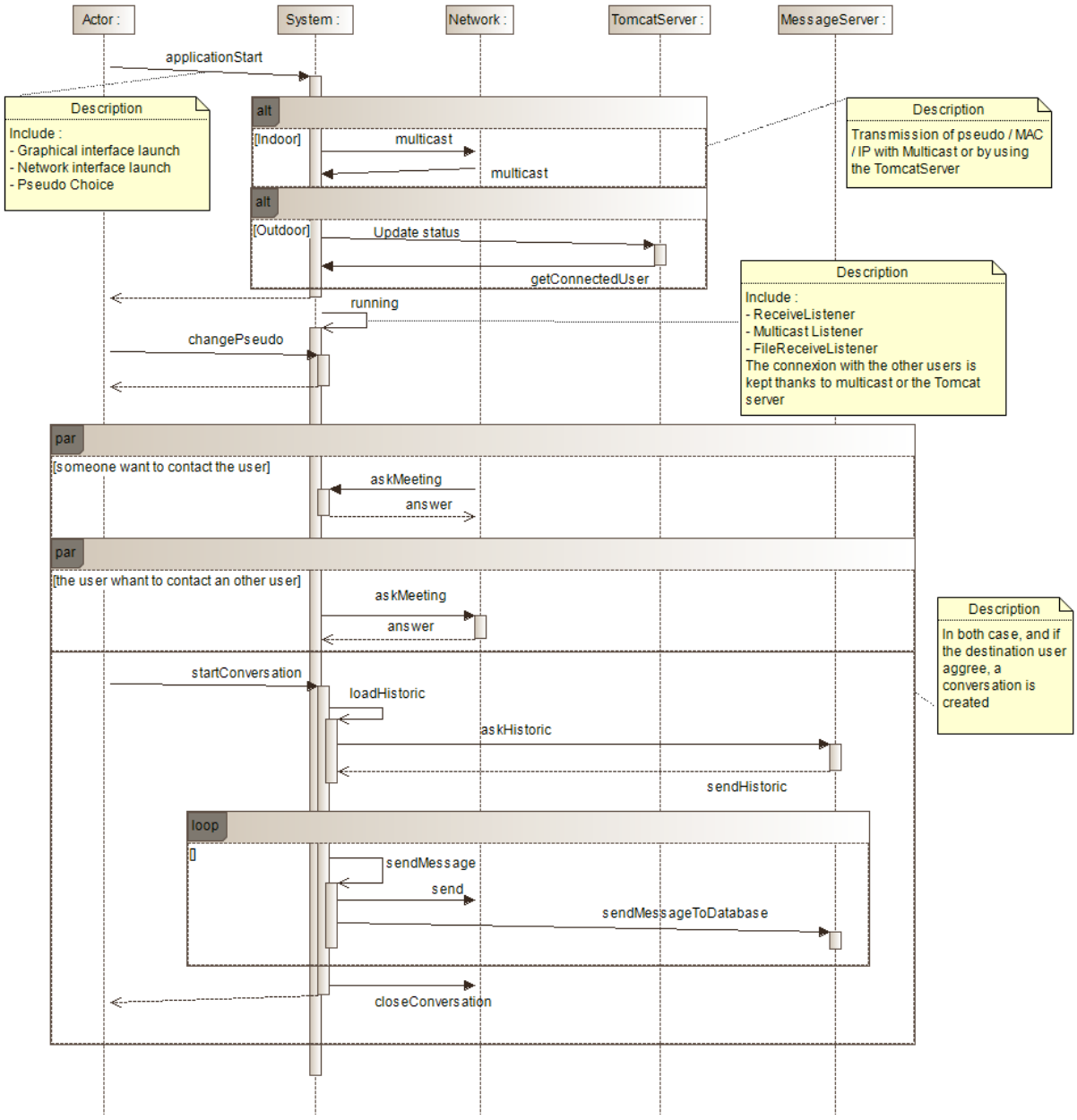


FIGURE 1.1 – Diagramme de séquence global

1.2.3 Diagramme de séquence

Les diagrammes de séquence suivant représentent le déroulement précis de certaines actions notables de l'application.

— **Lancement du client - Figure 1.2** Ce diagramme présente la séquence suivant le démarrage de la fonction *start()*, comprise dans la classe Launcher. Le but de cette fonction est de démarrer les instances principales du client (l'interface graphique et l'interface réseau notamment).

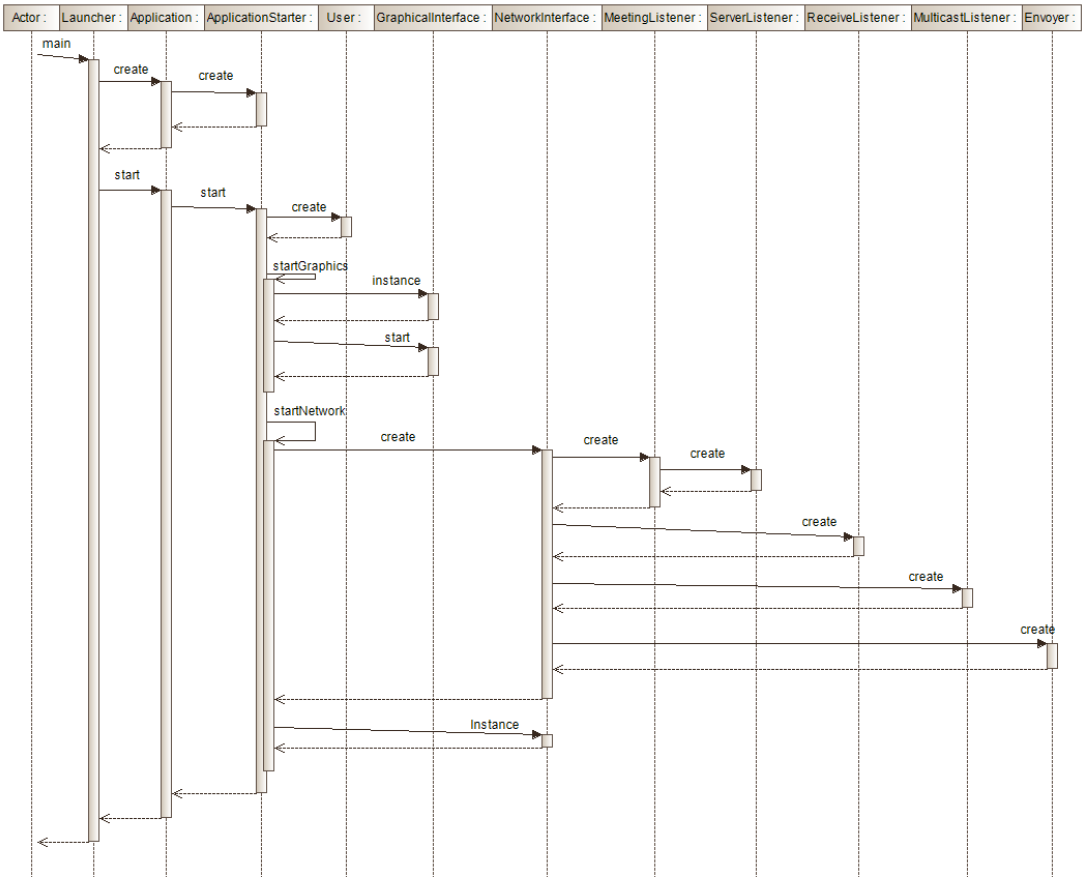


FIGURE 1.2 – Diagramme de séquence de la fonction start()

— **Changement de pseudo - Figure 1.3** Le choix du pseudo est plus contraignant que le changement, le diagramme suivant permet donc d'expliquer les deux actions. La première partie de la séquence concerne la validation du pseudo (i.e. personne d'autre n'utilise le même). Après cette phase, le client charge le reste de l'interface graphique et envoie le pseudo aux autres utilisateurs. Finalement l'utilisateur est connecté au serveur de présence.

— **Envoi d'un message - Figure 1.4** L'envoi d'un message avec la fonction send() (présente dans le package MessageEnvoyer et implémentée dans l'interface graphique) est divisée en deux parties successives. Premièrement la création du *packet* qui est lié à un *User*). Secondement, le packet est transmis à l'*Envoyer* qui gère le Stream vers l'autre *User*.

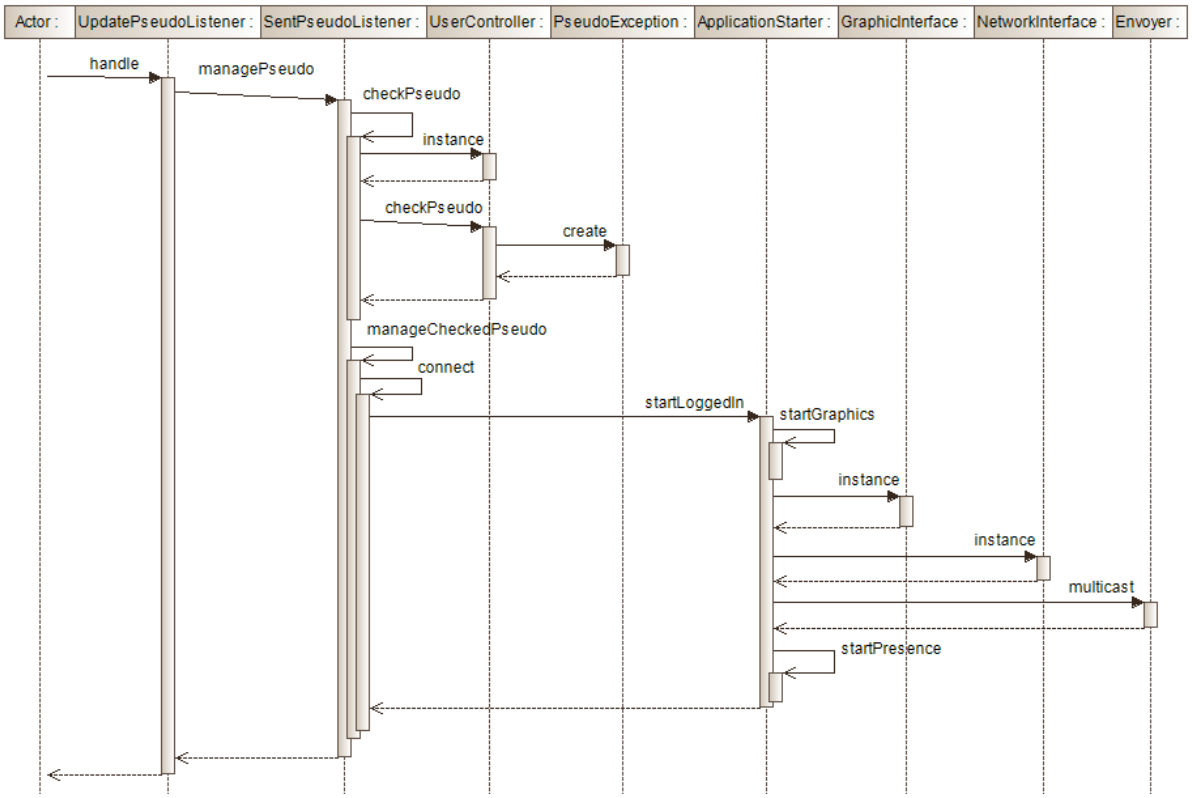


FIGURE 1.3 – Diagramme de séquence du choix de pseudo

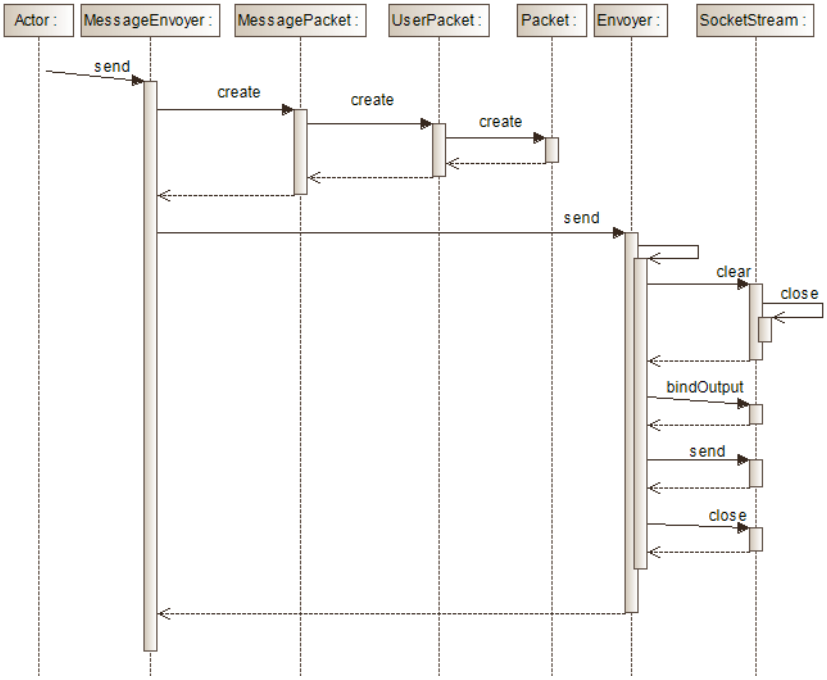


FIGURE 1.4 – Diagramme de séquence de la fonction send()

1.3 Architecture logicielle

Dans cette partie, nous allons présenter l'architecture choisie pour les différentes applications composant notre système. Vous retrouverez également les réflexions liées aux choix des technologies utilisées. Nous avons choisi de positionner l'ensemble du système sur un seul et même répertoire Git pour faciliter l'envoi du code source au client et notre organisation interne. A la racine du projet, vous retrouverez donc les dossiers :

— **bin** : dossier contenant l'ensemble des exécutables du projet. Vous retrouverez `client.jar` pour exécuter l'application bureau et `presence.war` pour démarrer le serveur de présence.
— **docs** : contient l'ensemble des diagrammes UML (présentés ci-dessus) et le cahier des charges fournis.
— **lib** : contient les librairies (sous forme d'exécutable) utilisées.

1.3.1 Application bureau

Le code est disponible sur le Git dans le dossier `messenger-client`. Les principaux packages du dossier `fr/insa/messenger/client` sont listés ci-après. Les diagrammes de classes de http, controllers, models et observers sont disponibles dans l'annexe B.

— **network** : package regroupant toute l'interaction de l'application bureau avec le réseau. Le package regroupe les *envoyers* et les *listeners* qui permettent respectivement d'envoyer et de recevoir des paquets envoyés par d'autres utilisateurs. La `NetworkInterface` permet de centraliser le démarrage des *threads* réseaux et d'obtenir l'instance d'`Envoyer` permettant de gérer l'envoi de paquets pour un type de message spécifique.
— **http** : package utilisé pour communiquer avec le serveur de présence via requêtes HTTP. La classe `PresenceInterface` contient les deux seules méthodes accessibles par l'utilisateur : *subscribe* et *publish*.
— **controllers** : ensemble des contrôleurs. Ceux-ci sont responsables du contrôle de la validité des données ainsi que du traitement de celles-ci. Vous retrouverez donc les classes `UserController` pour gérer le choix du pseudo et le statut de l'utilisateur, et `ConversationController` pour la gestion des conversations.
— **ui** : contient l'ensemble de l'interface graphique utilisateur. La *GraphicInterface* permet de démarrer l'ensemble des composants graphiques dans un même *thread*.
— **system** : package contenant l'ensemble des outils liés au système. Entre autre, vous retrouverez la classe `Env` contient les variables d'environnement de l'application (telle que l'instance de l'utilisateur connecté), des utilitaires pour la gestion des *assets* et un gestionnaire de console pour afficher des messages personnalisés en fonction de l'état de l'application.
— **models** : contient les classes définissant les concepts de bases du système utilisés dans le reste des classes. On y trouve les classes `User`, `Conversation` et `Message`.

La classe `Launcher` permet de lancer l'application en créant une instance de la classe `Application`.

1.3.2 Serveur de présence

Le code est disponible sur le Git dans le dossier `messenger-tomcat-server`. Le package est relativement léger. Le serveur de présence fonctionne via requêtes HTTP POST. Les routes disponibles sont présentes dans le package `routes`.

Stockage des status

Le statut des utilisateurs est stocké en base de données. Ceci est un choix de l'équipe de développement, vu qu'aucune contrainte ne limitait le stockage à une solution donnée. Dans le cas où le serveur de présence doit redémarrer (erreur interne, problème de serveur, etc.), les statuts doivent être disponibles sans pour autant demander de nouveau à chaque utilisateur son statut.

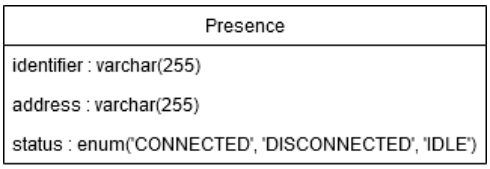


FIGURE 1.5 – Base de données du serveur de présence

Tests graphiques

Le package `routes/tests` contient la définition de routes GET permettant de tester graphiquement le fonctionnement des routes. Par exemple, la route `/publish-get` définie dans la classe `PublishGetServlet` permet d'accéder à un *input* pour renseigner votre pseudonyme, tel qu'il est possible de le faire en POST.

Voici la liste des routes POST avec les paramètres attendus :

Route	Paramètres	
	Type	Nom
/subscribe	String	identifier
/publish	String	identifier
	UserStatus	status

TABLE 1.1 – Liste des paramètres des routes

1.3.3 Package tools

Le package *tools* est accessible dans le dossier `messenger-tools` du répertoir Git. Il regroupe l'ensemble des outils communs à l'application bureau et au serveur de présence. Il contient entre autre la gestion de la base de données et la configuration des applications. Un fichier *jar* a été généré dans le dossier `lib` afin de faciliter l'import de la librairie.

La configuration des applications se fait via le fichier `.properties` présent à la racine des applications. Celui-ci permet de configurer les clients applicatifs pour s'adapter à l'environnement réseau. Le fichier contient, par exemple pour l'application bureau, les accès à la base de données et l'adresse du serveur de présence.

Chapitre 2

Développement du projet

Dans cette partie, nous allons nous focaliser sur le développement des principales fonctionnalités de l'application bureau : **authentification** des utilisateurs, **découvertes des *indoor users***, **découverte des *outdoor users***, **envoi** et **réception** d'un message puis la gestion de l'**historique**.

2.1 Authentification des utilisateurs

Comme spécifié dans le cahier des charges, l'authentification d'un utilisateur ne doit pas se faire de façon centralisée. Nous pouvons donc exclure les systèmes d'identifiant-mot de passe. De plus, l'utilisateur ne doit pas être identifié de façon unique par un identifiant, puisque l'utilisateur peut le changer d'une session de clavardage à une autre. Néanmoins, afin de pouvoir retrouver les messages envoyés, il faut convenir d'un identifiant remarquable et définissant de façon sûr un utilisateur.

2.1.1 Identifiant de l'utilisateur

Nous avons ainsi opté pour utiliser un identifiant réseau pour authentifier l'utilisateur. Ainsi, dans les communications, l'utilisateur sera identifié par ce numéro réseau (adresse IP, adresse MAC, etc.) au lieu d'un identifiant volatile.

Dans les communications, les utilisateurs seront identifiés par leur adresse MAC. L'adresse IP est trop volatile, du fait de nombreuses installations réseaux fonctionnant avec un serveur DHCP [1] allouant dynamiquement des adresses IP aux machines du réseau. L'adresse MAC est assurée unique au monde par les constructeurs de cartes réseaux. Ainsi, elle nous permettra d'identifier à coup sûr un utilisateur de l'application.

2.1.2 Choix du pseudonyme

Au niveau utilisateur, il est impensable de commencer une discussion avec une personne identifiée par son adresse MAC. Dès sa connexion à l'application, l'utilisateur devra

1. *Dynamic Host Configuration Protocol (DHCP)* : protocole réseau permettant d'allouer dynamiquement des adresses IP à des machines d'un réseau.

donc renseigner un pseudonyme unique pour utiliser l'application. Une contrainte ici s'impose : comment assurer l'unicité des pseudonymes ?

Lors du lancement de l'application, l'utilisateur indique à toutes les personnes connectées qu'il vient de démarrer son application. Ces personnes répondent en ajoutant leur pseudonyme. Ainsi, l'utilisateur sera capable de choisir un pseudo non utilisé sur le réseau.

Une fois le pseudonyme choisi, l'utilisateur envoie un message à tous les connectés pour leur indiquer qu'il est disponible pour démarrer une discussion.

2.2 Découverte des *indoor users*

Pour identifier les utilisateurs présents sur le réseau d'entreprise, nous allons utiliser une adresse de multicast [2]. Pour un réseau local, il est impératif d'utiliser une adresse de multicast comprise entre 224.0.0.1 et 224.0.0.255, auquel cas le message ne sera jamais reçu par les récepteurs.

La découverte des *indoor users* se fait dès le lancement de l'application. Dès qu'un nouvel utilisateur se connecte, il demande à son tour la liste des personnes connectées (voir la section Choix du pseudonyme).

2.3 Découverte des *outdoor users*

Pour des personnes connectées en dehors du réseau d'entreprise (réseau local), il n'est pas possible d'utiliser une adresse de multicast, car celles-ci sont bloquées sur la majorité des routeurs de l'Internet. Pour palier à ce problème, nous mettons en place un serveur de présence.

Le serveur de présence est développé en *Java Servlet* et requiert donc un serveur *Apache Tomcat* pour fonctionner. Le serveur doit être accessible depuis l'Internet et dispenser les méthodes suivantes :

Subscribe Cette méthode permet de s'inscrire à un flux. L'utilisateur renseigne son adresse MAC et pourra donc être notifié dès qu'un utilisateur se connecte ou se déconnecte du réseau via paquets TCP.

Publish Cette méthode permet à un utilisateur de modifier son statut qui peut prendre, par exemple, les valeurs *Connecté* ou *Déconnecté*. L'utilisateur sera notifié des changements de statut des autres utilisateurs uniquement si son statut est différent de *Déconnecté*.

Dans notre cas, les requêtes précédentes sont implémentées en HTTP via la méthode POST.

2. *multicast* : diffusion d'un émetteur vers un groupe de récepteurs. Cette diffusion se fait par l'intermédiaire d'une adresse de multicast sous la forme d'une adresse IP.

2.4 Envoi et réception d'un message

L'envoi et la réception de messages se fait via socket. Chaque type de message a une implémentation particulière.

2.4.1 Demande de rencontre

La demande de rencontre s'effectue lorsqu'un utilisateur souhaite en contacter un autre par le biais de l'application bureau. Cette demande permet à la cible de refuser de nouvelles connexions en fonction de son statut ou de sa charge actuelle.

Ainsi, la demande de rencontre dispose de 4 états :

Request Le paquet contient une demande de connexion. Lors de la réception de ce type de message, le client peut accepter la requête en envoyant un `ACCEPTED` ou la refuser avec un `DENIED`.

Accepted La précédente demande de connexion a été validée par la cible. On peut donc graphiquement ajouter l'utilisateur à la liste des discussions courantes.

Denied La précédente requête a été refusée. L'utilisateur reste donc dans la liste des personnes joignables.

Leave L'envoyeur exige la fin de la communication. Cette exigence n'est pas une demande : de son côté, l'envoyeur a déjà clos tous les canaux concernant l'utilisateur cible. Il n'est donc déjà plus possible de communiquer avec celui-ci.

2.4.2 Messages

Lors de l'envoi graphique d'un message, le client ouvre une connexion TCP en direction de la cible sur le port `RECEIVING_PORT` de la `NetworkInterface`. Sur ce même port, la cible est en attente de connexion avec un `ServerSocket`.

2.4.3 Fichiers

L'envoi et la réception d'un fichier ne peuvent se faire de la même manière qu'un message. La taille d'un fichier, pouvant être volumineux, peut dépasser la taille maximale admissible par un paquet TCP. Il faut donc procéder à l'envoi de plusieurs paquets.

Ainsi, lorsqu'un utilisateur sélectionne graphiquement le fichier à envoyer, une demande d'envoi se fait chez le client en précisant le nom original du fichier, le nom temporaire dans le système de fichiers et sa taille. Ainsi, la cible peut se mettre en attente de réception d'un fichier sur le port `FILE_RECEIVING_PORT` de la `NetworkInterface`.

2.4.4 Multicast

Dans un premier temps, une solution `broadcast` a été développée pour l'envoi des informations ponctuelles aux utilisateurs du réseau. Une solution `multicast` a néanmoins remplacé cette forme de communication pour améliorer les performances. Il reste néanmoins possible de changer pour le `broadcast`, les méthodes ont été laissées dans le code.

Les paquets `multicast` sont envoyés via le protocole UDP. Les informations sont sérialisées (*serialize*) et désérialisées (*unserialize*) pour être plus facilement transportées et interprétées.

2.5 Gestion de l'historique

L'historique de la conversation entre deux utilisateurs est chargé à chaque début de conversation. Les utilisateurs sont identifiés par leur adresse MAC dans la base de données. Ainsi, dès le début d'une discussion, une requête MySQL permet de récupérer l'ensemble des précédents messages. Les fichiers envoyés sont visibles mais non téléchargeables du fait de la décentralisation du système.

Annexe A

Guides d'utilisation

Dans cette partie, vous retrouverez les différents guides utilisateurs et administrateurs concernant l'application bureau et le serveur de présence. Vous aurez, ainsi, des **guides d'installation**, des guides pour **démarrer** et **utiliser** les applications.

Si vous utilisez le logiciel `IntelliJ` de la suite JetBrains, les fichiers `IML` permettront de configurer le projet ainsi que les `artifacts` afin de générer facilement les fichiers `jar`.

A.1 Guide d'installation

Les deux applications du système emploie le gestionnaire de dépendances Maven [1] pour faciliter le développement et le déploiement. Le répertoire Git contient dans le dossier `bin` les exécutables pour démarrer les applications.

Dans un premier temps, il vous faut récupérer le contenu du répertoire Git via la commande :

```
git clone https://github.com/Pythagus/java-messenger
```

La commande précédente va créer un dossier `java-messenger` contenant le code du système.

Maven L'ensemble du système utilise Maven pour la gestion des dépendances telles que `MySQL` et `JSON`. Après le téléchargement du code source, il faut vous référer à la documentation Maven pour télécharger les dépendances nécessaires au fonctionnement du projet. Dans les fichiers `jar` du dossier `bin`, les dépendances sont incluses dans l'exécutable.

Fichier de configuration Le fichier `.properties` regroupe l'ensemble des paramètres modifiables de l'application, tels que les identifiants de la base de données et l'adresse HTTP du serveur de présence. Afin de créer ce fichier dans votre architecture, vous pouvez exécuter la commande : `make install`. Vous pourrez alors modifier librement le fichier `.properties` généré.

Serveur de présence Afin d'installer le serveur de présence, vous devez importer le fichier `WAR` généré à l'aide de l'interface graphique de votre serveur Tomcat.

1. *Maven* : Gestionnaire de dépendances développé par Apache dont la documentation est accessible à l'adresse https://maven.apache.org.

A.2 Exécuter l'application

Dans cette partie, vous retrouverez un guide expliquant comment démarrer l'application depuis le fichier `jar` ou depuis un IDE.

A.2.1 En utilisant le fichier jar

Vous retrouverez à la racine du projet un dossier `bin` contenant des versions compilées des différentes applications. par exemple, pour exécuter l'application bureau, il vous suffit d'exécuter la commande :

```
java -jar bin/client.jar
```

A.2.2 En utilisant le fichier IDE

Démarrer l'application depuis un IDE doit uniquement être réservé aux administrateurs. Veillez à ne pas fournir cet accès aux utilisateurs.

Application bureau Pour démarrer l'application bureau, vous devez vous rendre dans le fichier `fr/insa/messenger/client/Launcher` disposant d'une méthode `main`. C'est le point d'entrée de l'application.

Serveur de présence Le serveur de présence ne dispose pas de point d'entrée. Il fonctionne sur la base des `Java Servlet` : dès qu'une route est appelée, le servlet associé est exécuté. Il n'est donc pas possible de démarrer le serveur de présence.

A.3 Utilisation de l'application

Cette sous-partie s'intéresse à l'utilisation de l'application bureau. L'utilisation du serveur de présence a été détaillée dans la section 1.3.2.

A.3.1 Choix du pseudo

Une fois l'application bureau démarrée, une fenêtre comme en figure A.1 demandant un pseudo doit s'ouvrir. Si le pseudo choisi est invalide ou déjà utilisé, une pop-up affiche le message d'erreur à l'utilisateur.

A.3.2 Fenêtre principale

La barre de navigation est présent à gauche de la fenêtre principale (figure A.3). Dans l'ordre de haut en bas :

— **Logo de l'application** : retour à l'accueil. Ce bouton affiche de nouveau la page de démarrage de l'application.

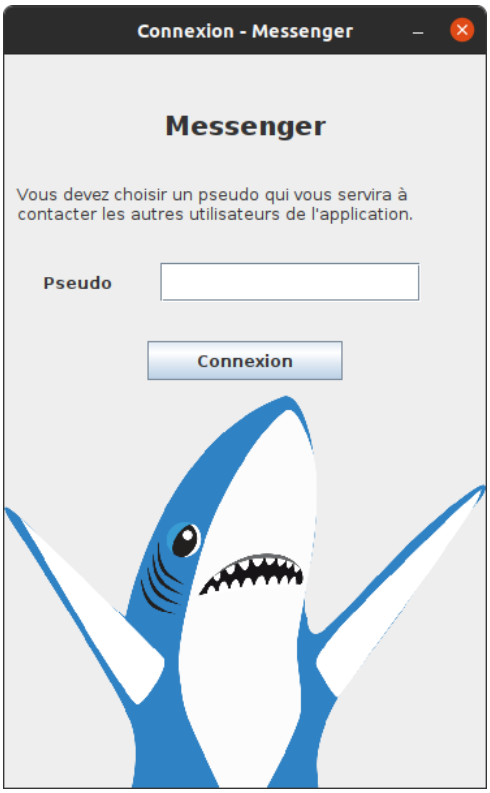


FIGURE A.1 – Fenêtre de démarrage de l'application

— **Personnes connectées** : liste des personnes connectées au système avec lesquelles l'utilisateur connecté n'a pas démarré de conversation. Il suffit de cliquer sur le nom d'une personne pour démarrer une conversation.
— **Mes discussions** : liste des conversations courantes. Dès qu'une personne ferme une conversation ou quitte le réseau, elle disparaît de cette liste.

A.3.3 Paramètres

Sur la page de paramètres présentée en figure A.2, vous retrouverez la modification du pseudo de l'utilisateur et de son statut (Connecté, Déconnecté, Absent).

A.3.4 Conversation

Dès qu'une conversation débute, la page présentée en figure A.3 s'affichera à l'écran. La conversation peut être quittée via le bouton rouge situé en haut à droite.

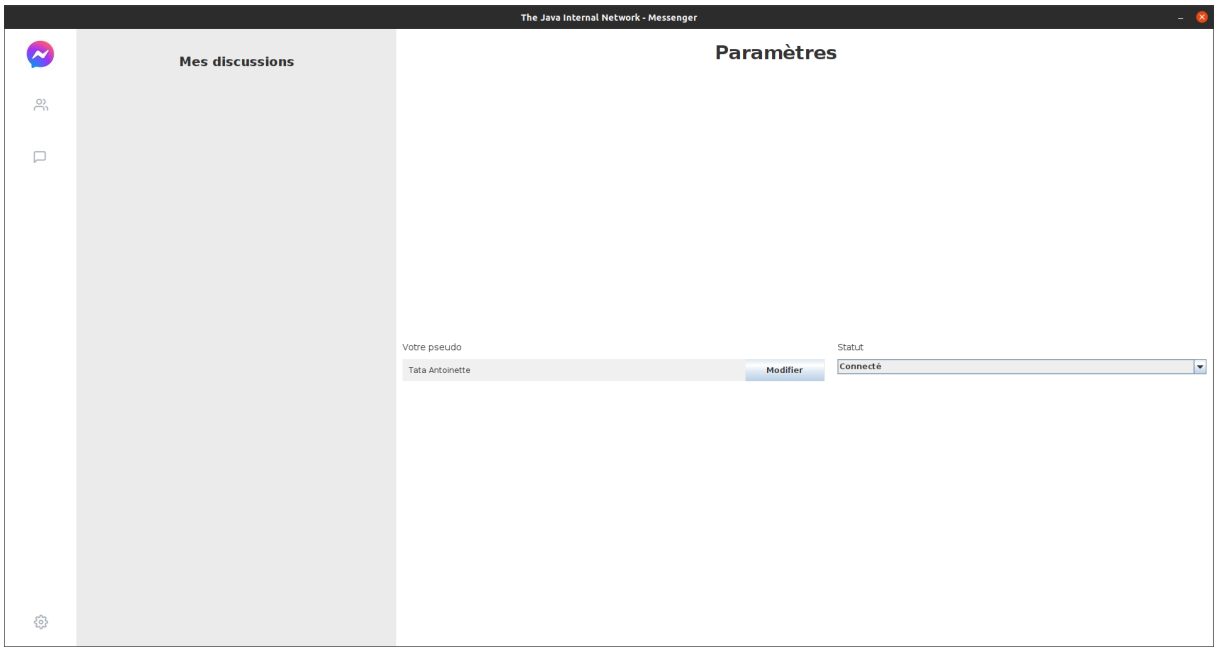


FIGURE A.2 – Page de paramètres

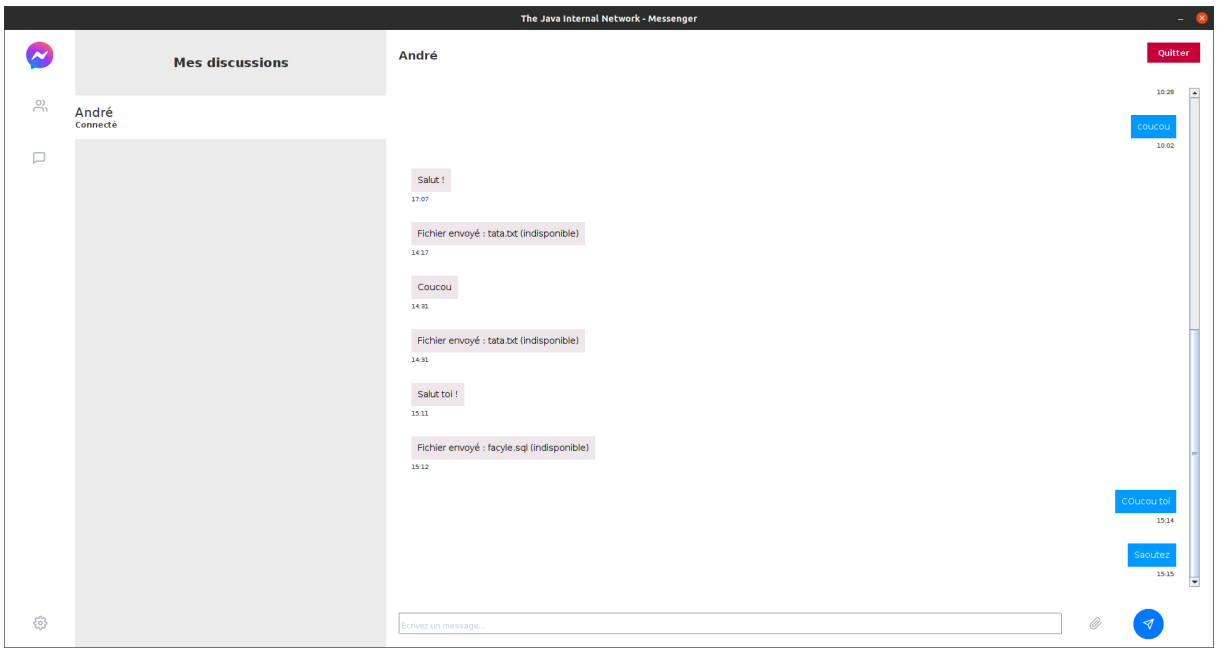


FIGURE A.3 – Affichage d'une conversation

Annexe B

Diagrammes

Cette partie les diagrammes des classes des packages les plus importants (le package network étant très complet, le diagramme est peu lisible sur ce format, un .png est disponible sur Git)..

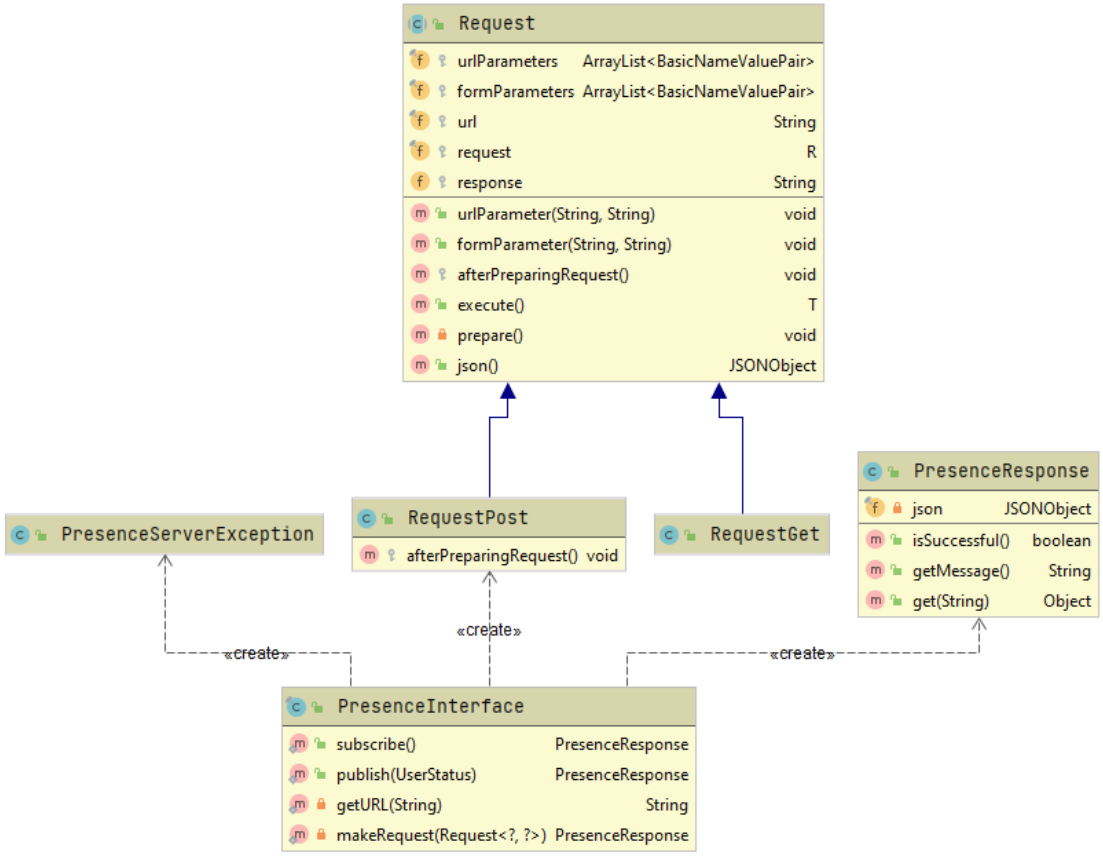


FIGURE B.1 – Package http

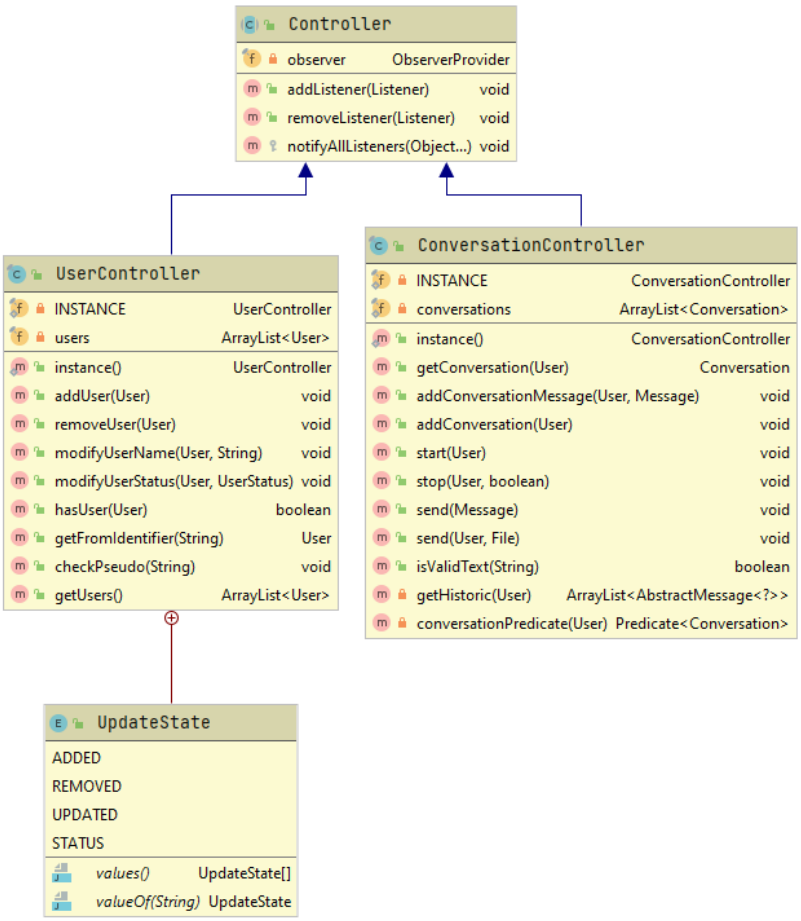


FIGURE B.2 – Package controller

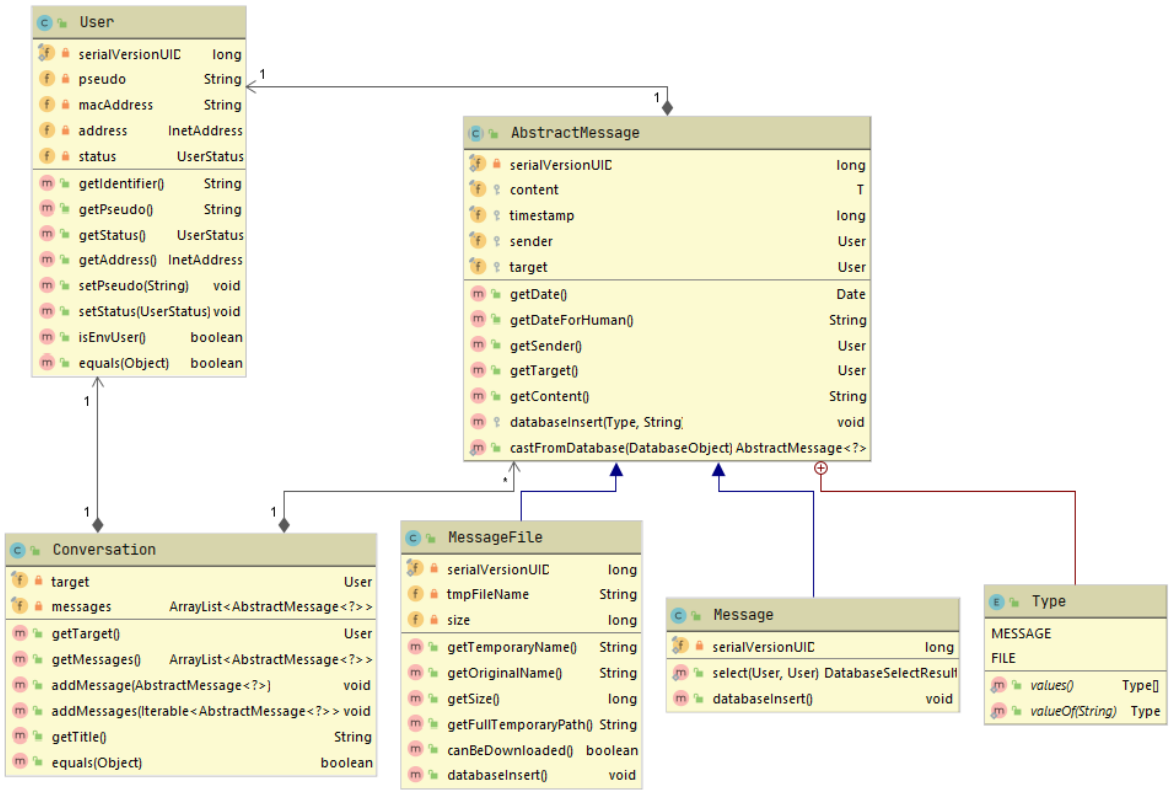


FIGURE B.3 – Package models

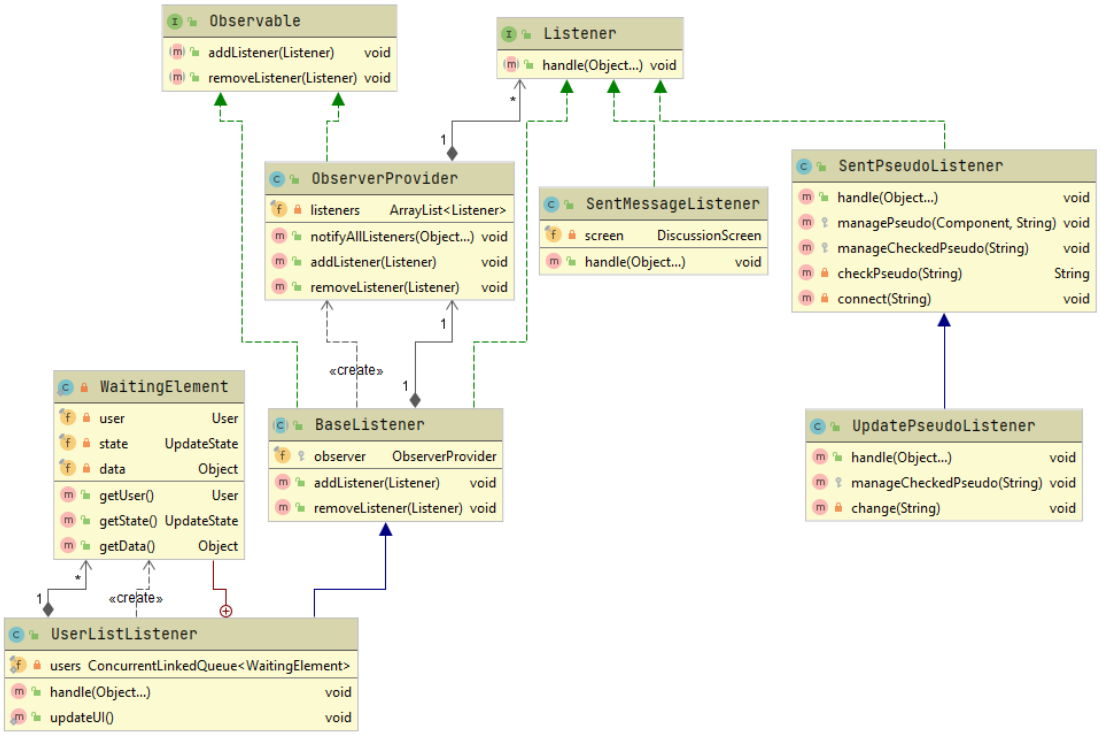


FIGURE B.4 – Package observers