Benassai Marino
Picco Bastien

RAPPORT POO

Application de Clavardage

Sommaire

I. Introduction

Le but était de développer une application de clavardage (“chat”) dans son intégralité, de l’authentification à l'envoi et au stockage de messages.

Le système doit pouvoir gérer la détection et la communication entre les utilisateurs internes de manière décentralisée.

Enfin, en addition, le système doit pouvoir prendre en compte des utilisateurs extérieurs, à l’aide d’un serveur.

II. Notice

Préalable :

Avant toute utilisation du système, le servlet (.war) doit être installé sur un serveur pourvu à cet effet. Ce serveur doit posséder tomcat version 9.0.16 au moins, et permettre l’utilisation de servlet. Tout utilisateur doit pouvoir accéder à ce serveur.

De plus, tout utilisateur doit être connecté au réseau de l’INSA, soit directement, soit via l’utilisation d’un vpn afin de pouvoir accéder aux bases de données.

Pour utiliser le système, l'utilisateur devra “créer un compte” pour cela, il doit contacter un administrateur qui lui demandera de créer un “login” et un “password” afin de les ajouter à la table de la base de données directement. Pour l’instant, les seuls utilisateurs ont pour identifiant “1”, “2”, “3” et “4”, et pour mot de passe admin.

L’identifiant ne doit pas contenir les chaînes de caractères suivantes : “$$$” ni “___”.

L’utilisateur peut utiliser n’importe quel ordinateur et même changer d'ordinateur entre 2 connexions.

Lancement :

Au lancement de l’application l’utilisateur est présenté à un premier menu. Celui-ci lui servira à s'authentifier, à l’aide de son identifiant, unique, et mot de passe personnel. Il devra, de plus, signaler s’il est extérieur au réseau de l’entreprise ou non.

Après avoir validé ses choix, il sera présenté à un second menu où il pourra choisir son “Pseudo” unique. Afin de passer à l’étape suivante, l'utilisateur doit donc rentrer un “pseudo” qui n’a pas encore été pris par un autre utilisateur actuellement connecté.

Enfin, l’utilisateur arrive sur le “Menu Principal”. A partir de celui-ci l’utilisateur à accès à une liste en couleur des différents utilisateurs (si le servlet est opérationnel). Les couleurs correspondent à l’état de chaque utilisateur : vert → “En Ligne”, rouge → “Occupé”, gris → “invisible”.

A noter que “invisible” signifie que l'utilisateur est bien connecté au système mais ne veut pas (et ne peut pas) être dérangé par une nouvelle communication entrante.

De plus, l'utilisateur pourra, lui-même, choisir son état à l’aide du menu déroulant prévu à cet effet. Il pourra de plus se renommer (ce qui le renvoie à l’étape précédente. (A noter : entrer un champ vide à ce moment revient à annuler l’action).

Enfin, en sélectionnant un utilisateur de la liste et en cliquant sur le bouton “Se Connecter” l’utilisateur entrera en communication avec l’utilisateur sélectionné.

Discussion :

Lorsqu’une discussion est établie, dans un sens ou dans l’autre, la fenêtre de discussion apparaît sur les 2 appareils.

A partir de celle-ci l’utilisateur peut entrer un message et l'envoyer. Il peut aussi lire les messages envoyés et reçus lors de la communication présente et celles passées.

Enfin, si l’utilisateur ferme cette fenêtre, la discussion est dès lors rompu.

A Noter :

L’utilisateur peut établir une discussion avec plusieurs utilisateurs distincts en même temps.

Si l’un des 2 utilisateurs en communication coupe la communication, l’autre utilisateur reçoit une alerte et tous messages envoyés par la suite ne seront ni reçus ni stockés.

III. Architecture Générale

L'application marche en local ou en 3-tiers si l'on vient de l'extérieur. Dans ce cas, elle utilise un servlet. L'authentification se fait via un serveur SQL.

IV. Technologie Choisie

L'ensemble du code a été fait en Java à l'aide d'Eclipse JEE. Ceci nous a permis d'organiser le code, d'utiliser les plugins, ainsi que les fonctionnalités "built-in".

Afin de pouvoir travailler à distance et chacun de son côté nous avons créé un répertoire git sur le GitEtude INSA. Toutes les actions étaient effectuées directement à partir d'Eclipse.

Pour la partie UI, nous somme partis sur la technologie SWING, et avons été aidé par un plugin qui permettait de les faire graphiquement (ce qui permettait d'avoir la structure et le code de base déjà tapés).

Pour la partie historique et authentification nous avons utilisé une base de données du GEI et donc le langage MySQL.

Enfin le servlet est implémenté dans un serveur TomCat de l'INSA.

V. Choix d'implémentation

Pour chaque catégorie (ci-dessous) nous utilisons des Gestionnaires, implémentés sous la forme de Singleton, qui centralisent les actions et données spécifiques à une certaine catégorie. Les catégories identifiées se retrouvent dans le code sous la forme des "packages".

Les choix décrits, concernent le fonctionnement "en interne" du programme. Les modifications liées à l'architecture 3-tiers seront évoquées dans la catégorie : "Servlet".

APPLICATION :

Détection des utilisateurs : (package = liste)

Dès le lancement de l'application, celle-ci va chercher à recréer la liste des utilisateurs actuellement connectés. Pour cela, nous passons par des "pseudo-commandes" transmises sur le réseau local, en broadcast grâce à UDP. Celles-ci sont aussi directement transmises à la servlet via HTTP.

Les 2 classes principales sont le Gestionnaire (GestionnaireListeUtilisateur) qui permet l'attribution d'un port d'écoute ainsi que de port d'envoie de messages utilisant UDP. C'est aussi là que sera stockée la liste des utilisateurs. Ainsi que la classe de traitement de pseudo-commande (TraitementCmdListe). Qui va décomposer la commande (représentée par un String) en mots clés et lancer la bonne action (méthodes) en fonction de ceux-ci.

Afin de ne pas perdre de messages dans les buffers, tous les messages reçus sur le port d'écoute en UDP sont transmis à un thread qui s'occupe de les traiter. De plus, puisque la liste des utilisateurs est une donnée partagée entre les threads nous utilisons un unique sémaphore pour faire un traitement à la fois et dans l'ordre. Cela assure que toutes les commandes seront traitées et sans créer de problèmes.

Ainsi, dès le lancement de l'application une première commande sera transmise en broadcast demandant à tous ceux qui la reçoivent (donc tous les connectés) de transmettre leur nom. Ce qui sera aussi fait en broadcast. Cela sert de sécurité afin de rafraîchir la liste de tous les utilisateurs, des fois qu'un message se soit perdu.

Une pseudo-commande ressemble à ceci : *ordre$$$id$$$nom$$$ip$$$statut*.
Il y a donc 5 mots clés séparés par "*$$$*" qui ne doit donc pas être présent dans les mots clés. Les messages étant broadcaster toutes les infos concernent l'envoyeur.

Les ordres possibles sont : "*add*" pour ajouter/renommer (en fonction de l'id) l'utilisateur, "*delete*" pour supprimer un utilisateur, "*listRequest*" pour demander à tout le monde d'envoyer son nom et "*statut*" pour mettre à jour le statut d'une personne.

A noter : Les 3 premières actions sont envoyées par d'autre utilisateur alors que la dernière ne peut être envoyée que par le servlet.

Enfin, un Type personnalisé à été créé : "TypeListeUtilisateur" qui correspond au type de chaque élément de la liste des utilisateurs (qui décrit donc un utilisateur).

Choix du nom utilisateur : (package = nom)

Ce package ne comporte qu'une seule classe, son gestionnaire (GestionnaireNom), qui fonctionne de pair avec le gestionnaire précédent.

Celui-ci centralise toutes les données liées à l'utilisateur actuel de l'application (nom/id/ip/ …). Ainsi que les méthodes permettant de retrouver l'id d'une personne à partir de ses autres informations. Et les méthodes de plus haut niveau gérant les actions "add" et "delete" concernant l'utilisateur courant.

Il contient de plus la méthode qui va permettre de récupérer la liste des utilisateurs actuels. Celle-ci permettra de vérifier l'unicité du nom choisi. De plus, le nom ne peut pas contenir la chaîne de caractère suivante : "$$$".

Authentification et Historique : (package = bdd)

Ce package est constitué de la classe "Liaison" permettant de se connecter à notre base de données du GEI. Ainsi, que du gestionnaire de l'historique (GestionnaireHistorique).

L'authentification s'effectue à l'aide de la table "*user*" qui contient tous les identifiants uniques ainsi que les mots de passe associés. Le programme se contente donc de regarder si le couple fourni par l'utilisateur correspond à une entrée valide dans la table.

L'historique est géré ainsi : à chaque couple d'utilisateur correspond une table d'historique nommée "*id1___id2*" avec id1 précédent alphabétiquement id2, créer dynamiquement au besoin. Chaque message est ensuite ajouté dans la table sous la forme d'une nouvelle ligne. Le gestionnaire sert à centraliser ces méthodes.

Représentation des données (package = data)

Le package data contient des représentations de données utilisées dans les autres classes.

data.Message

La classe data.Message est utilisée pour l'échange d'information entre deux clients, ou entre le client et le servlet. La communication entre ces différentes entités prend des formes diverses (TCP, http directement, http avec utilisation du servlet comme relais). L'utilisation d'un objet simple et unique dans ces communications permet de faciliter les échanges. La classe Message contient simplement 5 champs, ainsi que les getters et setters associés. Le premier est le champ type, qui représente le type du message. Il existe deux types de Messages. Les commandes

ont pour but d'agir sur un autre client ou sur le servlet (par exemple, demander au servlet de renvoyer la liste des messages qu'il a reçu pour un utilisateur, ou à un client de démarrer une session). Les messages textuels sont une représentation des messages échangés par les utilisateurs du logiciel. Les deux champs suivants représentent respectivement le mandataire et le destinataire du message, sous la forme de leur identifiant. Le quatrième message contient le corps du message, sous forme d'une chaîne de caractères. Il s'agit du message ou de la commande en tant que tel. Enfin, le dernier champ contient le moment auquel le message a été créé.

data.ServletResponse

La classe data.ServletResponse sert à stocker la réponse du servlet lorsqu'un client lui envoie un message. Elle contient le code de retour de la requête, et la liste éventuelle des Messages retournés par le servlet.

Connexions TCP (package = tcp)

Ce package contient les classes utilisés pour mettre en place et utiliser TCP

tcp.TCPClient

Un client qui peut se connecter à un serveur via une adresse et un port. Permet de créer et d'arrêter une connexion, ainsi que d'envoyer des chaines de caractère et des objets Java sérialisables.

tcp.TCPServer

Un serveur qui écoute sur un port particulier. Lorsqu'un client tente de se connecter, le serveur crée un TCPserverThread, qui possède son propre thread et s'occupe de la communication. Le serveur se remet immédiatement en attente d'une connexion.

tcp.tcpServerThread

Un thread lancé par TCPServer à chaque nouvelle connexion. Une fois lancé, se met en attente de réception d'un objet Message. Lorsqu'un objet est reçu, le stocke dans une file, puis retourne dans son état attente. Cet objet est un observable, et signal aux observateurs qui l'observent chaque nouveau message reçu.

Communication : (package = clavardage)

Les sessions de clavardage font principalement intervenir les classes du package clavardage.

clavardage.GestionnaireMessagesDistants

Cette classe récupère périodiquement les messages adressés à l'utilisateur stockés sur le servlet, et les traite.

clavardage.GestionnaireSessionClavardage

Cette classe a pour but de gérer la création, l'accès à et la terminaison des différentes sessions auxquelles participe le client.

clavardage.SessionClavardage

Les classes qui en héritent représentent une session de clavardage. Elles contiennent les informations sur cette session, les messages non encore traités par l'interface utilisateur, et permettent certaines actions entre sessions, tel que la fermeture ou l'envoi d'un message. SessionClavardageDistante est de plus un observable, et signale aux observateurs qui l'observent chaque nouveau message reçu.

Il existe deux types de sessions de clavardage, locale et distante. Les sessions locales font intervenir deux clients présents sur le réseau local, et utilisent TCP.

Création d'une session de clavardage sur le réseau local

L'ouverture d'une session de clavardage par le client A, entre le client A et le client B, se fait de la manière suivante :

- Le client a créé une connexion TCP vers la machine du client B, sur un port bien connu. Il commence aussi à créer un objet session. Une fois la connexion établie, il envoie son identifiant sous la forme d'une chaîne de caractères. Il crée aussi la fenêtre de discussion.

- Le client B reçoit cette chaîne, et vérifie alors si une session existe déjà entre lui et le client A. Si ce n'est pas le cas, il crée à son tour une connexion vers le client A, sur le port bien connu, ainsi qu'un objet session et une fenêtre de discussion.

- Le client A, après ouverture de la seconde connexion, et après avoir vérifié qu'il était en train d'ouvrir une session avec B, finit de créer son objet session.

Echange des messages pour une session locale

La présence pour chaque session de deux connexions, de A vers B et de B vers A, permet d'avoir un canal dédié à l'envoi, et un autre à la réception. Chaque connexion fait intervenir un TCPClient pour l'émission des messages, et un TCP

Détails d'implémentation

Lors du lancement de l'application, un TCPServer est lancé sur un nouveau thread via la classe GestionnaireSessionsLocales et reste actif tout au long de l'exécution du programme. Le but de ce thread est de recevoir les demandes de création de connexion TCP utilisés pour les sessions, de créer les connexions TCP nécessaires en retour, ainsi que d'effectuer les diverses vérifications citées précédemment. Les connexions TCP sont assurés par des clients tcp de la classe réseau

Echange de message lors d'une session contenant au moins un client extérieur

Les sessions distantes contiennent au moins un client situé à l'extérieur du réseau local, qui ne peut donc pas communiquer directement via TCP avec les autres clients. L'échange de message depuis et vers l'extérieur du réseau local se fait grâce à un servlet java. Les clients communiquent avec le servlet en lui transmettant des objets Message via des requêtes http POST. Le servlet traite alors le message reçu. S'il s'agit d'une commande qui lui est destinée, il l'exécute. S'il s'agit d'un message ou d'une commande destinée à un client, il le stocke jusqu'à ce que ce client le demande. Deux commandes peuvent être adressés au serveur : getMessages et initialize. GetMessages permet à un client de récupérer les messages et commandes stockés par le servlet qui lui sont destinés. Le servlet répond à la requête en transmettant la liste des messages et commandes en question. Initialize permet à un client d'indiquer au servlet qu'il souhaite recevoir des messages par son intermédiaire. Une fois cette commande reçue, le servlet stockera les messages destinés à ce client, et il pourra les récupérer à tout moment.

Création d'une session par ou vers un utilisateur à distance

Le démarrage et l'arrêt de ces sessions passent par des objets Messages spéciaux. Pour lancer une session avec B, A crée un objet session, puis transmet une commande « startSession » au servlet. Cette commande est récupérée par B, qui le traite et crée la session de son côté. Pour mettre fin à la session, A envoie à B, toujours par l'intermédiaire du servlet, une commande « stopSession ».

Détails d'implémentation

Un thread, lancé dès l'ouverture de l'application, est dédié à la récupération des messages, ainsi qu'à leur traitement, via la classe GestionnaireMessageDistants. Lors de son lancement, le thread indique au servlet qu'il veut recevoir des messages, puis récupère périodiquement ces messages. Il les traite ou les distribue entre les différentes sessions en fonction de leur contenu. Il récupère aussi les commandes qu'il envoie à TraitementCmdListe.

L’interface utilisateur est avertie de l'arrivée d’un nouveau message pour la session en cours par un mécanisme d’observateur. La donnée observée est la file contenant les messages dans la classe représentant la session.

Le servlet stocke les messages dans une HashMap, dont les clés sont les identifiants des utilisateurs ayant utilisé initialize. Le but est de pouvoir efficacement accéder aux messages d’un utilisateur en particulier, ce qui représente la majorité des opérations réalisées par le servlet (qu’il s’agisse d’ajouter ou de récupérer des messages). Les messages eux-mêmes sont stockés dans une file, qui se prête bien à l’ajout et au retrait successifs de données. L’implémentation de file utilisée met aussi en place des mécanismes de protection contre les accès concurrents, nécessaires ici (Les insertions et retraits d’éléments sont atomiques). Des précautions supplémentaires ont été prises pour s’assurer que l’opération de vidage de la file ne puisse être effectuée que par un thread à la fois.

UI : (package = ui)

Ce package contient toutes les interfaces graphiques (authentification, nommage, liste et discussion). Pour chacune d’entre elles les actions liées aux différents objets ont été mises dans des méthodes en dessous du constructeur.

Login : contient les champs d’authentification que l’utilisateur doit remplir et vérifie leur concordance avec la base de données. Contrôle, de plus, le booléen indiquant si l’utilisateur se connecte depuis l'extérieur ou non.

Nom : Permet uniquement de se nommer ou renommer suivant les cas.

List : C’est le menu principal. Il permet de voir la liste des utilisateurs actuellement connectés et de rentrer en relation avec eux. Permet, de plus, de lancer un “renommage” ou de changer son statut.
La liste est mise à jour graphiquement après chaque pseudo-commande la modifiant, depuis TraitementCmdListe.

Discussion : Permet d’envoyer les messages et de les lire ainsi que ceux des sessions précédentes. La détection des nouveaux messages se fait en observant la source de ces messages, à savoir l’objet session pour les sessions à distance et le serveur TCP recevant les messages pour les sessions locales. Les messages envoyés et reçus sont affichés dans le cadre

supérieur de la fenêtre de discussion. Chaque message est précédé du moment auquel il a été envoyé, ainsi que du nom de l'utilisateur qui l'a envoyé.

Le moment indiqué correspond à l'heure qu'indique la machine depuis laquelle le message a été envoyé au moment où il l'a été. La validité de cette information dépend donc de la validité de l'horloge de la machine ayant envoyé le message, et elle peut s'avérer fausse en cas de dérèglement de l'horloge. Il aurait été possible d'éviter ce problème en récupérant l'heure depuis un serveur sur internet, mais cela aurait rendu l'horodatage dépendant de la connexion à internet, ce que nous voulions éviter. Les messages sont toujours affichés dans l'ordre de leur réception, ou d'archivage dans la base de données pour les messages de l'historique. Cela permet d'éviter un mélange des messages si deux machines dont les horloges sont différentes communiquent.

Le nom de l'utilisateur affiché avant chaque message est le nom que portait l'utilisateur ayant envoyé le message au moment de la réception de ce dernier, ou au moment du lancement de la session pour les messages de l'historique. Pour cela, les identifiants des destinataires Ce nom n'est donc pas dynamiquement mis à jour pour les messages déjà affichés lorsqu'un des participant à la conversation change de nom.
L'enregistrement des messages dans l'historique se fait à chaque envoi de message. Cela permet de ne pas perdre les messages si les deux clients s'arrêtent de manière anormale pendant la session. Cela permet aussi aux messages qui n'atteignent pas leur destinataire d'apparaître dans son historique lors de la prochaine session.

Enfin, fermer n'importe quelle fenêtre, mise à part celle de "chat", revient à éteindre l'application. Lors de la fermeture du menu principal, une pseudo-commande est envoyée avant de fermer la fenêtre. La fermeture d'une fenêtre de chat permet la réouverture de celle-ci (on ne peut pas ouvrir 2 fois, en même temps, la même conversation, et envoie un message au correspondant pour l'informer de notre départ.

Servlet : (package = servlet)

Permet d'envoyer une requête HTTP Post ou Get au servlet. La requête post ne transmet pas d'information et permet juste de se déclarer auprès de servlet (subscribe). En revanche, la requête post transmet des informations (pseudo-commande) à l'aide de son inputStream, comme nous le verrons dans la partie concernant le code propre de la Servlet (ci-dessous).

SERVLET :

Le servlet permet la liaison des individus extérieurs au réseau ainsi que de gérer le statut de tout le monde.

Il est composé de 2 listes principales : une liste d'IP auxquels sont envoyés les changements de statuts ainsi qu'une liste d'utilisateurs, le type "UserType" étant le même que TypeListeUtilisateur précédent, permettant de stocker les informations de chaque personnes actuellement connecté afin de les transmettre aux bonne endroit. on ne transmet pas tout à tout le monde systématiquement puisque le réseau interne doit continuer à fonctionner en décentralisé.

Le servlet est composé de 3 packages. "defaut" contient le Gestionnaire et la déclaration des constantes. "server" la déclaration des protocoles de communication afin de pouvoir les utiliser dans le Notify. Enfin, "servlet" contient la classe servlet, les 3 méthodes principales, séparées sous forme de classe pour des raisons de lisibilité : Subscribe, Publish, Notif(y). Ainsi que la classe permettant le traitement de la pseudo-commande.

Subscribe : exécuté, systématiquement, après réception d'une requête HTTP GET. Elle permet d'ajouter l'émetteur de la requête à la liste des adresses IP à notifier.

Publish : exécuté, systématiquement, après réception d'une requête HTTP POST. De la même manière et pour les mêmes raisons que les messages UDP (pseudo-commande) en local, on récupère la commande dans l'inputStream de la requête, et celle-ci est transmise à un autre thread qui va la décomposer et agir selon elle.

Les différentes actions possibles sont : "new" pour ajouter une nouvelle utilisateur dans la liste (id), "name" pour changer le nom d'un utilisateur existant, "statut" pour changer le statut d'un utilisateur existant et "delete" pour supprimer un utilisateur des 2 listes.

La pseudo-commande est similaire à celle de l'application, à la différence qu'elle prend un argument de plus : un booléen indiquant si la personne appartient au réseau interne de l'entreprise ou non.

Notif(y) : correspond à l'action "Notify" mais appelé "Notif" afin de ne pas overwrite une fonction built-in. Elle est appelée après un publish, et sert à envoyer des pseudo-commandes en UDP sur le port d'écoute de

l'application. L'action dépend de la commande reçue par le servlet. La commande est envoyée en unicast, et non pas en broadcast, puisque cela n'est pas possible pour les personnes extérieurs, à toutes les personnes nécessaire. Les destinataires dépendent donc de l'ordre (statut est envoyé à tout le monde) ainsi que de la localisation de l'émetteur et des récepteurs potentiels. (Un changement de nom venant de l'extérieur est transmis à tout le monde alors qu'un venant de l'intérieur n'est transmis qu'à l'extérieur).

VI. Test - Debug

Une constante booléenne, se trouvant dans liste.Constante, nommé "debug", permet de choisir si l'on se trouve sur un réseau (false) ou si l'on a accès qu'à un seul ordinateur (true).

En effet, afin de tester le programme sur un seul ordinateur les fonctions d'envoie/réception ont été dédoublées, sous cette forme : méthode() → méthode2(). Afin de fonctionner, non plus avec une adresse IP, mais avec différent port (de 2001 à 2009).

De plus, nous avons enregistré, manuellement, 5 utilisateurs dans la table "user" de la base de donnée (cf : annexe) afin de tester et passer l'étape d'authentification avec différents "user", puisqu'on ne peut pas se parler à sois-même dans la suite du programme. En mode "debug" pour lancer plusieurs utilisateurs il faut changer le port dans le "main" (entre 2001 et 2009) et en mettre 1 différents pour chaque utilisateur.

Grâce à tout ceci nous avons pu tester en **local** (1 seul ordinateur + serveur local) toutes les fonctionnalités.

L'authentification a bien lieu : l'application accepte lorsque le couple id + psd est bon et rejette dans tous les autres cas.

Le choix du pseudo est correct : il est impossible de prendre le pseudo d'une personne déjà connectée ainsi qu'un pseudo comprenant la chaîne de caractère "$$$". Idem lors d'un renommage.

Les communications fonctionnent : ouvrir une communication ouvre bien la fenêtre de chat pour les 2 utilisateurs concernés et les messages sont bien reçus. De plus, l'historique est bien chargé.

Le servlet fonctionne pour la partie statut : un changement de statut d'un utilisateur est bien marqué sur le menu principal des autres utilisateurs.

Pour les tests en réel, suite au déploiement du .war sur le serveur TomCat nous avons pu tester avec plusieurs machines (nos 2 machines personnelles) où nous avons pu constater le bon fonctionnement de la mise à jour de la liste, du statut, de l'historique et des fins de sessions.

VII. Piste d'amélioration

Cela n'était pas l'objectif mais il aurait été intéressant de consacrer de l'attention à l'interface graphique qui reste minimaliste. De plus, la fenêtre de chat n'est pas dynamiquement scrollé et le texte ne s'adapte pas au dimension (il faut scroller horizontalement). Enfin, les caractères spéciaux posant problèmes en SQL, toute ligne en contenant ne peut pas être insérée dans l'historique.

Une connaissance non parfaite du java a mené à du "mauvais code" (nom, façon de faire, localisation) en début de projet qui a pénalisé l'ensemble lors de la complexification de celui-ci. Par exemple, la classe TypeListeUtilisateur porte mal son nom, et n'a pas de méthode "intégré" ni de constructeur et afficheur. De plus nous voulions partir du français (comme nom de variables et autre) afin de ne pas renommer des méthodes built-in. Toutefois, certaines méthodes étant toujours en anglais getter/setter et les fonction built-in l'étant aussi. Nous avons donc au final du franglais.

Enfin, une meilleure connaissance du java et compréhension du sujet aurait permis de partir avec une meilleur COO qui a donc dû être revue, au moins pour le digramme de classe.

VIII. Annexe

Table d'authentification des utilisateurs

Id	pwd
0	admin
1	admin
2	admin
3	admin
4	admin