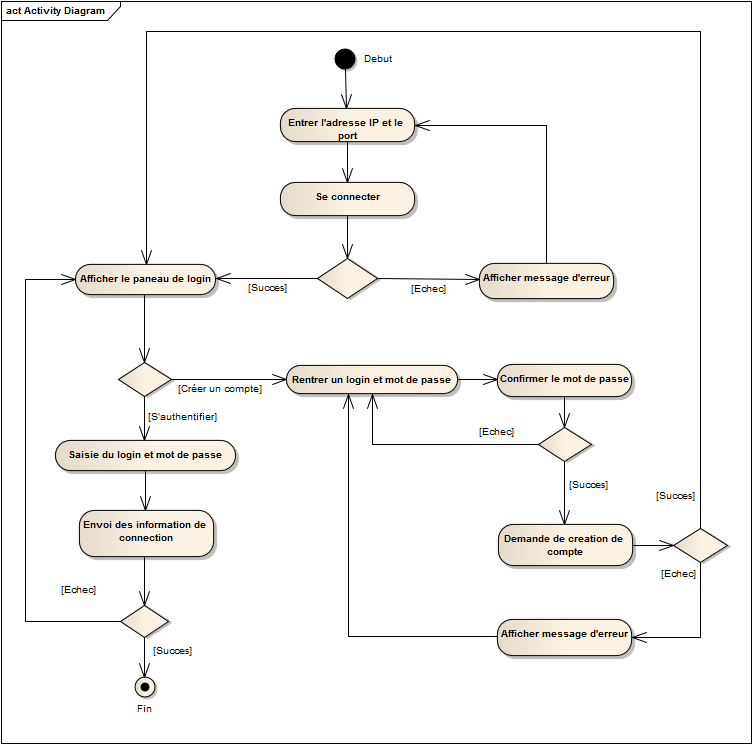
# Diagramme d’activité

Ce diagramme d’activité permet de représenter les différents scénarios depuis le lancement du programme jusqu'à ce que l’ont soit connecté et authentifié (logué) sur le serveur.



# Diagramme de séquence

Le diagramme suivant est un diagramme de séquence. Il représente les différents messages concrets entre le client et le serveur. (cf chapitre sur le diagramme de classe du serveur et du client pour plus d’information sur les classes)

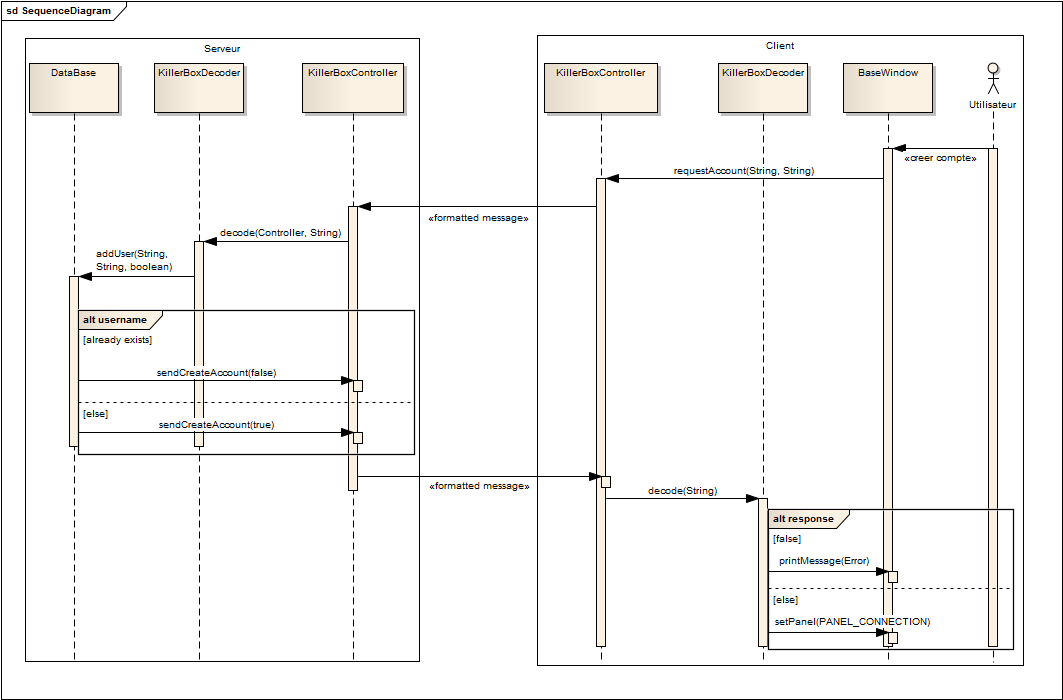
A gauche se trouve le serveur et à droite le client.

Les messages échangés représentent la demande de création d’un compte sur le serveur, par un utilisateur quelconque. Les stéréotypes « *formatted  message* » indiquent un protocole d’échange bien défini entre les deux applications.

L’utilisateur (via l’interface) graphique, décide de créer un compte sur le serveur. L’action sur le bouton « créer compte » fait appel au contrôleur qui envoie les informations pour créer le nouveau compte. Le message formaté circule via le réseau pour enfin arriver sur le contrôleur du coté serveur. Le contrôleur va ensuite transférer l’intégralité du message au décodeur (protocole). En lui passant une référence sur le contrôleur de la connexion qui a générée le message.

Le décodeur va ensuite décomposer le message et reconnaître sa signification. Celui-ci va comprendre qu’il faut ajouter un utilisateur (*user*) avec le mot-de-passe (*pass*) dans la base de données à l’aide de la classe *DataBase*. De là, deux situations peuvent se produire : L’utilisateur existe déjà dans la base de données, dans l’autre cas, l’ajout du compte à réussi. Dans ces deux cas, le contrôleur envoi un message au client associé à ce contrôleur. Un booléen est utilisé pour indiquer le succès ou l’échec de l’opération.

Retour côté client. Le contrôleur reçoit le message et le passe au décodeur. En cas d’une réponse négative, le décodeur indique à l’interface d’afficher un message d’erreur, sinon l’utilisateur est redirigé vers la page de création de compte.



# Diagramme de classe du serveur

La partie serveur est décomposée en 2 packages.

* « network »
* « killerbox »

## Package network

Le package network permet de fournir une structure de classe la plus réutilisable possible. Elle est décomposée des classes suivantes : *Server, Controller, Decoder, AbstractServerStatus, MessageServerStatus, EnumServerStatus*.

### Server

Cette classe permet représenter un serveur basique. Elle permet à des clients de se connecter. Un thread principal est en attente de connexion. A chaque nouvelle demande, un thread est crée ainsi qu’un nouveau contrôleur pour cette connexion. La gestion complète du client est confiée au thread.

Cette mise en attente de connexion est effectuée à l’aide de la classe Java *ServerSocket.* Celle-ci associe à chaque nouvelle connexion un socket pour communiquer avec le client.

Les différentes connexions sont indexées à l’aide d’un ID unique. Afin de faire le lien entre le contrôleur et l’ID de connexion, une table de hachage (*HashMap*) est utilisée. L’ID est utilisé comme clé et l’instance du contrôleur comme donnée.

La classe est considérée comme un « modèle ». Ce modèle est susceptible d’émettre des messages de type « AbstractServerStatus ».

### Controller

C’est le contrôleur pour chaque connexion. Celui-ci (par le biais d’un thread) permet de récupérer les messages du client. Cette classe fournit également une méthode permettant d’envoyer de simples chaînes de caractère au client.

### Decoder

Est une classe abstraite permettant de décoder un message. Le protocole doit être défini par les classes utilisatrices. Concrètement, la méthode *decode(string)* doit être implémentée.

### AbstractServerStatus

Il est possible d’utiliser cette classe afin d’ajouter différents messages du serveur. (En cas d’ajout de fonctionnalités par exemple). Celui-ci est composé d’un *EnumServerStatus* afin d’indiquer le type de message. Cette classe permet en outre d’implémenter une architecture MVC sur le serveur. C'est-à-dire afficher les messages sur une console, un panneau graphique, etc.

## Package killerbox

La classe principale de se package est *KillerBoxServer.* Les différentes classes se contentes entre autre d’utiliser les services du package *network* et d’implémenter des méthodes supplémentaires.

### KillerBoxServer

Cette classe permet de gérer les fonctionnalités supplémentaires de la classe *Server.* Comme par exemple la gestion des noms d’utilisateur et la gestion des parties ou encore de la gestion de la base de données.

Elle permet par exemple de broadcaster uniquement certains joueurs d’une partie, ou j’ajouter une connexion en tant que connexion authentifiée. (Apte à participer à un jeu, accéder à son compte, etc.). Alors que la classe *Server* se contentait d’associer de simples ID à une connexion, le serveur de KillerBox ajoute une couche supplémentaire en y greffant la notion de nom d’utilisateur. Une table de hachage est utilisée pour associer les noms d’utilisateur au contrôleur.

La responsabilité de la gestion des parties est confiée à la classe *GameList* présentée un peu plus loin.

Les méthodes permettant d’accéder à la base de données sont implémentées dans la classe *DataBase*.

On pourrait se poser la question « Pourquoi une composition entre *KillerBoxServer* et *Server* et pas une généralisation ? » Ce choix tout à fait discutable du moment qu’on peut affirmer d’un serveur KillerBox EST UN serveur. La composition est été finalement été retenue étant donné que la classe *Server*  est un thread à part entière effectuant des actions bien déterminées tout en générant des messages pour informer par exemple des observateurs de l’objet. Dans le cas d’un héritage notre classe *KillerBoxServer* aurait hérité du type *Runnable* ce que nous ne voulions pas. L’application du MVC aurait également posé problème dans le sens ou un classe dérivée aurait dû observer les messages générés par sa super-classe, c'est-à-dire lui-même. Certains architectes software conseillent aussi de préférer la composition à la généralisation lorsque la question se pose.

### KillerBoxController

Représente une connexion entre un client et le serveur KillerBox. Cette classe dérive la classe Controller du package network et ajoute divers message prédéfinis à envoyer au client.

### KillerBoxDecoder

Implémente le protocole d’échange et implémentant la méthode *decode(String)* de la classe abstraite *Decoder*.

### GameList

Cette classe permet de représenter une liste de parties. Les parties (classe *Game*) peuvent se trouver dans deux listes différentes. En cours ou en attente. Les seules parties en attentes sont diffusées aux clients, ainsi il n’est pas possible de rejoindre une partie en cours.

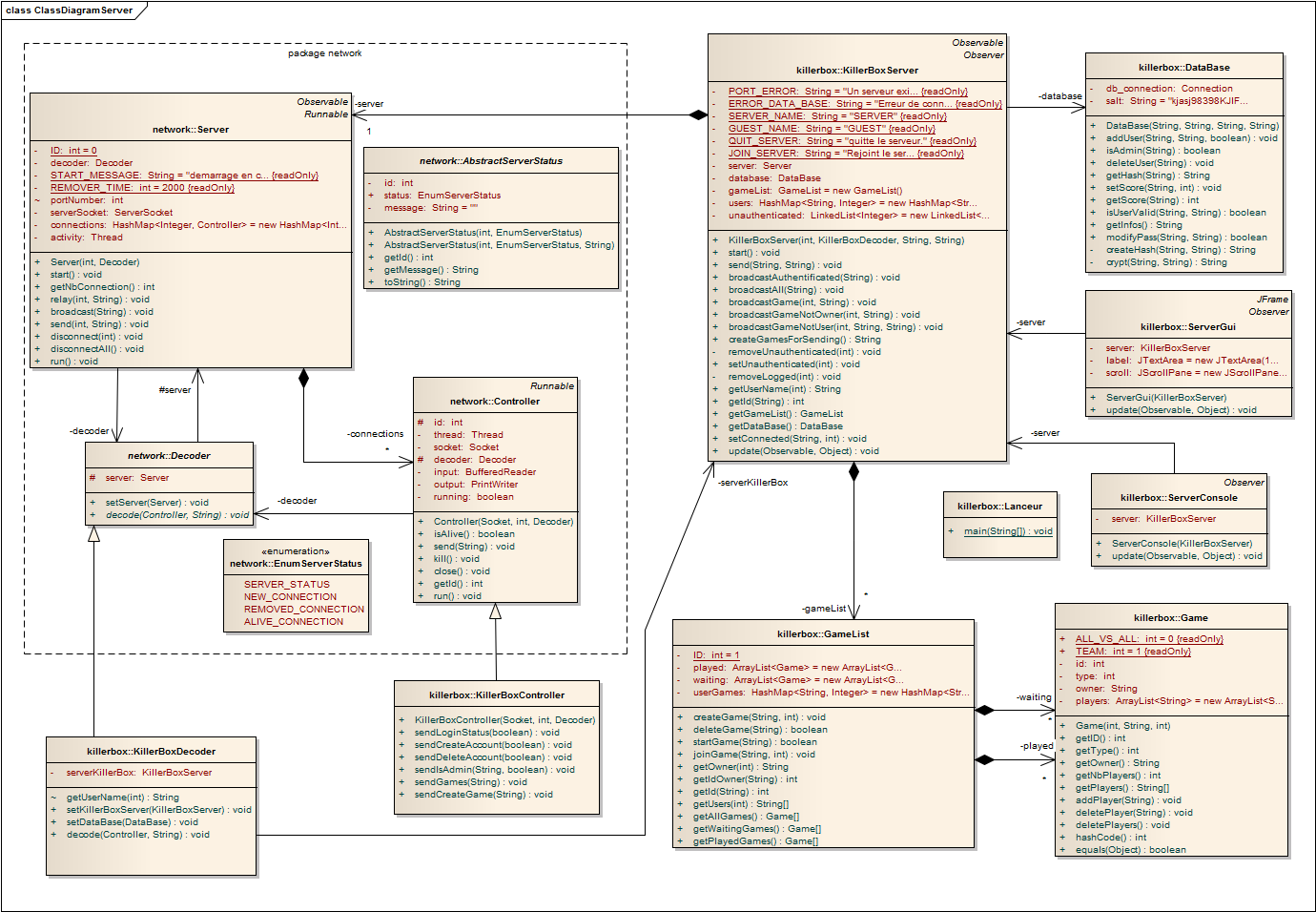
La classe fourni différentes méthodes pour créer, démarrer ou supprimer une partie. Elle permet également de retourner différentes informations sur tel ou tel partie. A chaque partie créée, celle-ci se voit attribuée un ID unique ainsi qu’un créateur (Nom d’utilisateur du créateur de la partie).

### Game

Représente une partie. Une partie est caractérisée par un numéro unique ainsi qu’un créateur. Le type de la partie est également indiqué (Tous VS Tous ou par équipe). Il est possible d’ajouter, supprimer divers joueurs dans la partie.

### DataBase

C’est classe permet de faire le passage entre le serveur KillerBox et la base de donnée MySQL. Celle-ci permet de masquer l’utilisateur de JDBC et de proposer des méthodes de bases comme ajouter des utilisateurs, changer un mot-de-passe, mettre à jour un score d’un joueur, etc. Le mot-de-passe n’est pas stocké en clair dans la base de données, mais est crypté en MD5.



# Diagramme de classe du client (hors moteur du jeu)

Le diagramme de classe suivant montre la partie réseau coté client ainsi que l’interface graphique, sans les différents panels et le moteur de jeu. Un chapitre sur le moteur de jeu y est consacré.

## Package network

Comme le serveur, le client utilise les services d’un package réseau. Ce package à été développé dans une optique de réutilisation.

### Client

Représente un modèle de client. Il est représenté par un status (connecté, non connecté), une adresse et un port d’un serveur s’il est connecté. Le client est un modèle observable en vertu de son changement de status possible. Cette classe est donc parfaitement réutilisable et permet d’utiliser ce client dans une optique MVC.

### Controller

Joue le rôle d’un contrôleur. Récupère les messages serveur puis les passent à un décodeur qui effectuera les actions adéquates. Le contrôleur observe les changements de status du client et quand celui-ci passe en status déconnecté, le contrôleur appelle la méthode *setDeconnected()* qui est abstraite et doit être implémenté dans les contrôleurs client concrets.

Lorsque le client repasse en mode connecté, le contrôleur démarre un nouveau thread et remets à jours ses références sur les flux d’input et d’output.

### Decoder

Classe abstraite dont le protocole doit être entièrement implémenté par les classes dérivées.

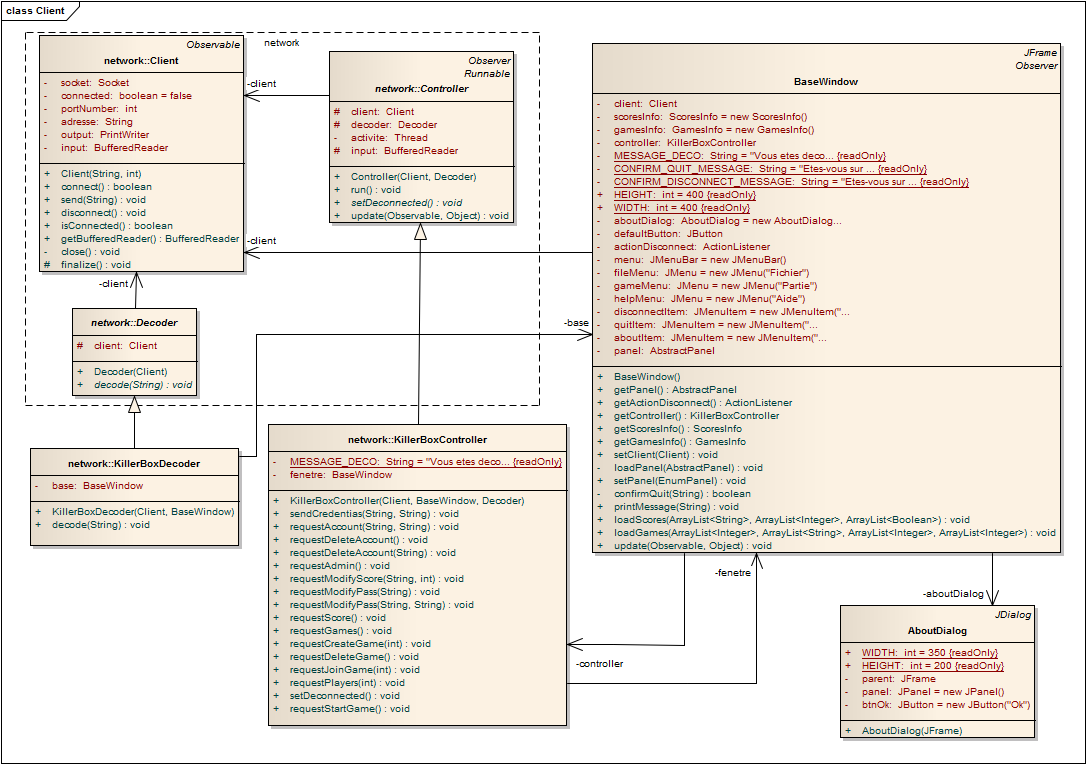
## Package killerbox.network

Ce package comprends le contrôleur (*KillerBoxController*) et le décodeur (*KillerBoxDecoder*) concrets pour le client. *KillerBoxController* met à disposition différentes méthodes pour envoyer effectuer des requêtes au serveur.

## Package killerbox.gui

Ce package permet de regrouper les éléments graphique du client (hors jeu).

Celui-ci est composé d’une fenêtre graphique principale *BaseWindow* contenant les différents panels. La boite de dialogue « about » fait également partie de se package.



# Mise à jour asynchrone des données

La mise à jour des données comme les scores, les utilisateurs connectés à une partie, etc. Sont réactualisé sans que l’utilisateur ne s’en rende compte. Un bouton actualisé est quand même présents dans la plupart des panels affichant des données serveur.

Un thread permet de collecter et rafraichir l’affichage tout le x secondes. Le thread utilise la notion de moniteur (*wait()* et *notify()*) pour garantir une cohérence entre le moment ou une requête à été effectué et la réception des données.

Les données sont regroupées sous une classe commune *Data* qui change de status lors de l’arrivée de nouvelles données.

On peut représenter le scénario de mise à jour du tableau des scores par le diagramme de séquence suivant :

