INFO-F-209: Projet d'informatique 2
Software Requirements Document

Thibault Hermans, Florian Vanhee, Nicolas Potvin, Kishiro Nishio,
Denis Hoornaert, Nathan Liccardo, David Majcherczyk, Tim Renière

Février 2016

Table des matières

1 Introduction

1.1 But du projet

Le projet consiste en la réalisation d'un jeu de cartes fantastiques en réseau jouable en un contre un sous Linux.
Lors de son inscription, chaque joueur reçoit un certain nombre de cartes, lui permettant ainsi de créer ses premiers decks. La collection de cartes du joueur s'étendra au fil des victoires qu'il remportera, lui rapportant chacune une ou plusieurs cartes.
Les cartes sont divisées en deux types : les créatures et les sorts. Les decks sont composés d'exactement 20 cartes et ne peuvent contenir qu'au maximum 2 exemplaires d'une même carte (à condition que le joueur possède au moins 2 exemplaires de cette même carte).
Lors des duels, chaque joueur commence avec 20 points de vie, 5 cartes en main et 1 point d'énergie. Lors d'un nouveau tour, chaque joueur pioche une carte et voit son maximum de points d'énergie augmenter de 1 (maximum 10 points d'énergie). Le premier joueur dont les points de vie sont réduits à 0 perd le duel.

Le but de ce projet est de permettre aux étudiants de mettre en pratique les concepts vus aux cours d'analyse et méthodologie informatiques et de systèmes d'exploitation. L'analyse UML du projet, le respect du paradigme orienté-objet et la bonne gestion du client-serveur seront donc des points essentiels dans la réalisation de ce projet.

1.2 Présentation du jeu

Le nom du jeu est "StoneTrooper". Le jeu est basé sur le thème Star Wars et largement inspiré des films ainsi que des séries liés à cette saga.
Les cartes sont réparties en 3 factions : Neutre, République et Empire. Lorsqu'un joueur crée un deck, il commence par choisir son héros (qui est soit de la faction République soit de la faction Empire). Ensuite, il ne peut ajouter à ce deck que des cartes de la même faction que son héros ou des cartes neutres.
Les termes liés au jeu (tel que les effets des cartes) ont été choisis en fontion

de ce thème.
Les règles et fonctionnalités détaillées du jeu sont reprises plus loin dans ce document.

1.3 Glossaire

avatar Image choisie par le joueur pour être affichée dans son profil.

combo Enchaînement de plusieurs cartes dans un ordre précis et dont l'efficacité combinée est supérieure à leur utilisation séparée.

créature Carte qui est déployée sur le terrain lorsqu'elle est jouée. Elle dispose de points d'attaque, de points de vie, d'un coût en énergie, et éventuellement d'effets spéciaux.

deck Paquet de 20 cartes utilisé par le joueur lors d'un duel. Ne peut contenir plus de deux fois la même carte.

force Points d'actions du joueur. Ils sont incrémentés de 1 à chaque début de tour du joueur. Le nom a été choisi pour rester en adéquation avec le thème, les noms les plus courants sont "énergie" ou encore "mana".

héros Carte principale du joueur lors d'une partie. Lorsque ses points de vie tombent à 0, le joueur perd la partie. Il possède également des capacités uniques.

personnage Catégorie regroupant les créatures et les héros.

qt Librairie c++ permettant notamment la réalisation d'interface graphiques.

sort Carte appliquant un effet spécial au détriment d'un certain coût en énergie. Elle est généralement défaussée lorsqu'elle est jouée.

1.4 Historique du document

Version	Auteur	Date	Description
16	Thibault H.	21/03/16	Choix d'implémentation duel
15	Thibault H.	29/02/16	Corrections syntaxiques
14	Thibault H. et Florian V.	28/02/16	Module Duel
13	David M.	27/02/16	Diagramme matchmaking
12	Tim R.	27/02/16	Module matchmaking et chat
11	Denis H.	26/02/16	Ajouts de diagrammes
10	Denis H.	26/02/16	Module Bases de données
9	Nathan L.	25/02/16	Module Affichage
8	Thibault H.	15/12/15	Corrections syntaxiques
7	Florian V., Kishiro N.	12/12/15	Diagramme de classe
6	Tim R., David M.	11/12/15	Ajouts de diagrammes
5	Denis H., Nathan L.	10/12/15	Ajouts de diagrammes
4	Denis H., Nathan L.	10/12/15	Exigences du système
4	Thibault H.	09/12/15	Exigences fonctionnelles
3	Nicolas P., Kishiro N.	07/12/15	Exigences non fonctionnelles
2	Florian V.	03/12/15	Exigences du domaine
1	Thibault H.	01/12/15	Introduction et structure

2 Besoins de l'utilisateur

2.1 Exigences fonctionnelles

Les exigences fonctionnelles de l'utilisateur ont décrites à l'aide des différents diagramme UML ci-dessous. Nous avons décidé de séparer ces diagrammes en 3 parties représentant chacune une interface particulière de l'application. Chacun des ces 3 diagrammes est décrit par un tableau reprenant ses pré-conditions, post-conditions, cas généraux et cas particuliers. Ces tableaux sont fournis en annexe.

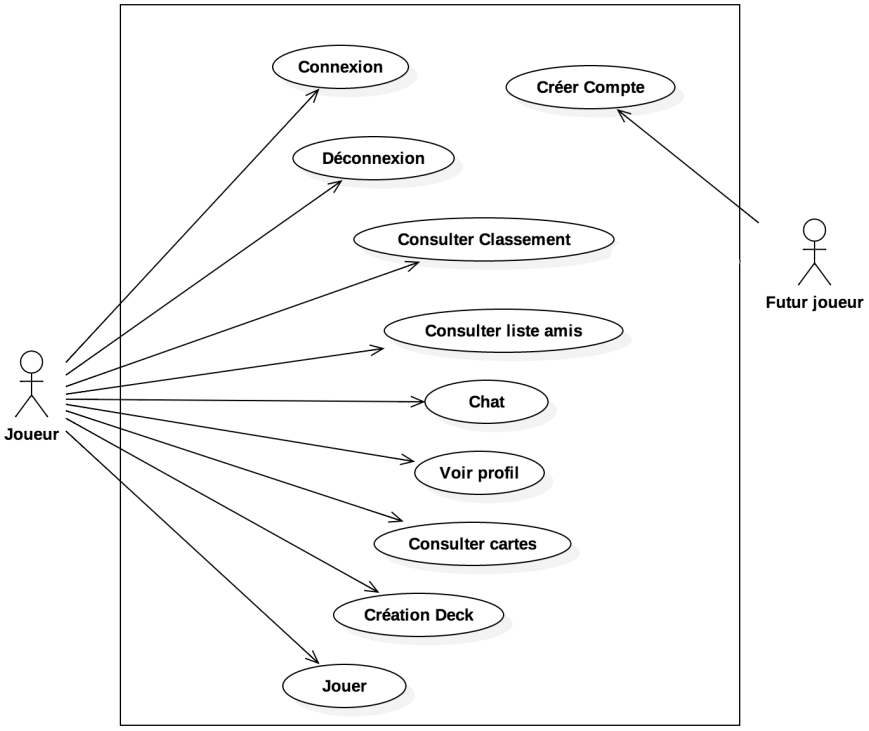


FIGURE 1 – Diagramme use case hors duel

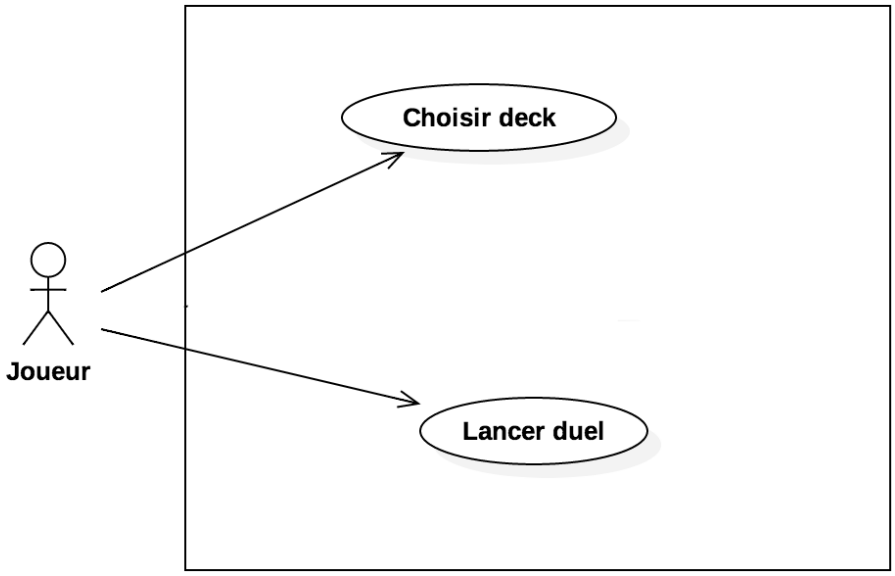


FIGURE 2 – Diagramme use case inter duel

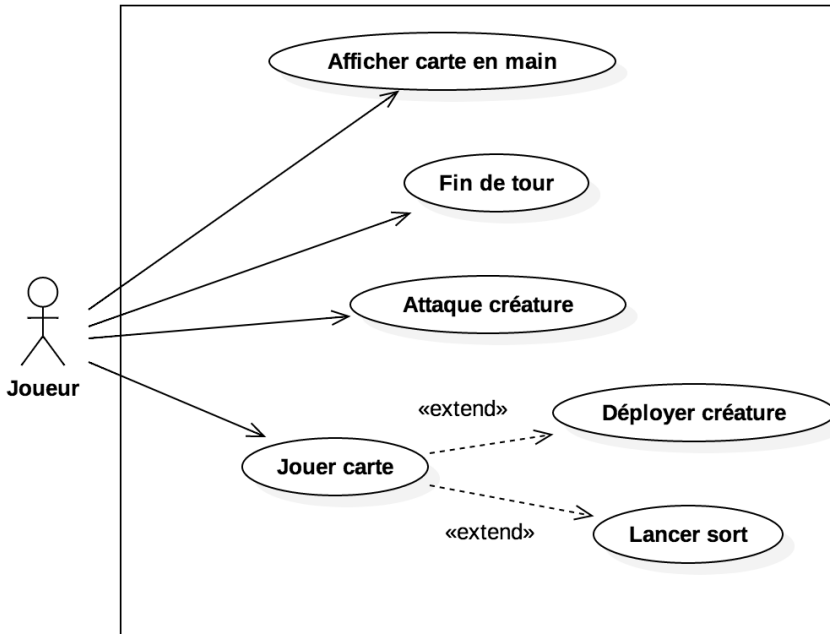


FIGURE 3 – Diagramme use case en duel

2.2 Exigences non fonctionnelles

Certains des besoins de l'utilisateurs sont en fait des obligations, comme les deadlines à respecter ou le choix du langage de programmation.
En ce qui concerne les dealines, la première est le vendredi 18 Décembre, la structure générale du programme devra avoir été décidée et présentée sous forme de diagrammes. Les deadlines suivantes seront le 26 février et le 25 mars.
Quant au langage imposé il s'agit du langage C++. La portabilité du programme n'a pas été mentionnée, mais il est évident qu'il doit pouvoir fonctionner sur les machines de l'ULB (Linux) et une compatibilité avec Windows a été évoquée, mais ne sera réellement envisagée que si elle ne demande pas trop de modifications et que les délais le permettent.

Les autres besoins non fonctionnels que nous avons relevés concernent le jeu et auront certainement une influence sur l'expérience de jeu du joueur :

- L'équilibrage des cartes :
  En effet, il est évident que le joueur voudra gagner. Si certaines cartes sont trop puissantes par rapport à d'autres, elles seront favorisées aux dépends des autres et les différents decks seront toujours les memes à quelques variations minimes près. Alors que si la puissance des cartes d'un meme coût est relativement bien équilibrée, plusieurs styles de jeu peuvent apparaitre ce qui amènera plus de diversité.

- La tolérance à la déconnexion :
  La déconnexion est un problème qui peut survenir à tout moment pour plusieurs raisons. Si le joueur est confronté à ce problème, alors il est normal que le jeu s'arrête et que l'adversaire est déclaré victorieux par forfait. Mais dans le cas où elle est brève, de l'ordre de la minute, une reprise de la partie en cours peut être envisagée. Il s'agit d'un jeu au tour par tour, il y a donc une contrainte potentiellement moins grande quant à la qualité de la connexion d'un joueur : qu'il joue durant la première minute de son tour ou la seconde, le résultat est globalement le même.

- Absence prolongée du joueur :
  Il est également possible que le joueur s'absente. Il est tout à fait normal qu'à un moment donné le joueur ait une obligation de la vie réelle à

remplir (téléphone, ...). Cependant il ne faut pas perdre de vue que de l'autre coté de l'écran se trouve un autre joueur qui attend que son adversaire inactif joue pendant toute la durée du tour (2 minutes) ce qui peut être exaspérant. C'est pour cette raison que si un tour s'écoule complètement sans que le joueur ait joué, son tour suivant est réduit à 15 secondes après lesquelles, si il n'a pas donné pas signe de vie, la partie est terminée et il est déclaré perdant.

Au cours de la réalisation de ce projet, certains besoins non fonctionnels encore non envisagés appraîtront peut être en plus de ceux-ci. Lorsque cela arrivera, il sera toujours fait en sorte de permettre le plus grand confort de jeu possible pour que l'expérience que le joueur en retire soit la plus positive.

2.3 Exigences du domaine

Le domaine du projet est à la croisée des chemins entre le jeu vidéo et le jeu de cartes. Il est important d'avoir un jeu qui soit agréable au regard et ludique. On a donc deux facettes importantes dont il faut tenir compte :

- La partie graphique qui sera traitée dans une version ultérieure de ce document.
- La partie design et règles du jeu.

2.3.1 Game design

Il faut à la fois que le jeu procure du plaisir au joueur durant ses sessions de jeu et qu'il maintienne son intérêt dans le temps. Il faut donc avoir une collection de cartes suffisante et introduire différents niveaux de rareté pour celles-ci afin que le joueur prenne du temps à la remplir et qu'il ait envie de gagner les cartes les plus rares.

En outre, les cartes doivent avoir des synergies entre elles. Dans l'idée que certaines cartes sont indispensables à des combos, récompensant ainsi le joueur d'avoir créé un deck efficace.
Pour que le joueur puisse constater sa progression de manière plus générale, il pourra consulter un classement prenant en compte le ratio de victoires/défaites de chaque joueur et un historique personnel des ses dernières parties.

De plus, le joueur pourra choisir entre 2 camps, qui comprendront chacun leurs cartes spécifiques ainsi que de nouveaux héros, que le joueur pourra débloquer en faisant monter son niveau dans chacun de ces deux camps (en jouant des parties avec ces deux camps). Le joueur pourra également débloquer des cartes en gagnant des parties.

3 Besoins du système

3.1 Exigences fonctionnelles

Les besoins fonctionnels du système regroupent les besoins cités par le client et ayant des conséquences sur l'architecture du système. Ces besoins n'ont généralement aucun lien avec l'utilisateur du système et sont donc cachés.
Après avoir analysé les besoins on retrouve donc plusieurs besoins fonctionnels qui sont les suivants :

- Trouver un duel :
  Cette fonction permet au système de trouver deux personnes disponibles aléatoirement afin de démarrer un duel.

- Initialiser un duel :
  Une fois les joueurs définis, le système doit initialiser la partie ainsi que les différentes données nécessaires. L'instanciation d'un nouveau duel implémentera directement la sélection aléatoire du premier joueur de la partie.

- Duel :
  Le duel en tant que tel est également une partie importante du programme. Ce duel sera agrémenté d'un certains nombre de fonctionnalités demandées. Ces fonctionnalités sont les suivantes : Gestion des points de vie, gestion des effets, gestion des cartes (aussi bien dans le deck que dans la main ou sur le terrain) et vérification de victoire ou de défaite ainsi que la récompense qui est associée au vainqueur.

- La mise en place d'un classement regroupant l'ensemble des joueurs et étant mis à jour après chaque partie.

Les points repris précédemment composent donc (pour nous) les besoins les plus importants exprimés par le client et concernant la partie système de l'architecture.

3.2 Exigences non fonctionnelles

Les besoins non-fonctionnels regroupent l'ensemble des besoins qui, après analyse du projet, semblent être important afin de modéliser correctement le projet. Cette section reprend donc les quelques points qui n'ont pas été exprimés de manière explicite par le client.
Etant donné que le programme sera réalisé sous la forme d'un client-serveur il est évident qu'un certains nombre d'interactions entre le client et le serveur seront mises en place. Nous avons donc regroupés trois interactions importantes :

- L'enregistrement d'un utilisateur sur le système ainsi que la connexion de ce dernier
- La mise à jour du profil enregistré sur la base de données
- La mise à jour de la liste d'amis de l'utilisateur

Nous avons également pris le parti de créer un objet dit "Héros" qui représentera le joueur durant la partie. Le duel décrit plus haut implementera donc deux Héros (représentant les deux joueurs).
Il est important de noter que les besoins repris ci dessus ne constituent que les points clés repris suite à l'analyse de la demande du client.

3.3 Design et fonctionnement du système

Dans cette section sont repris les différents diagrammes permettant de décrire le fonctionnement du système. On y retrouve le diagramme de classes, des diagrammes de séquences, etc.

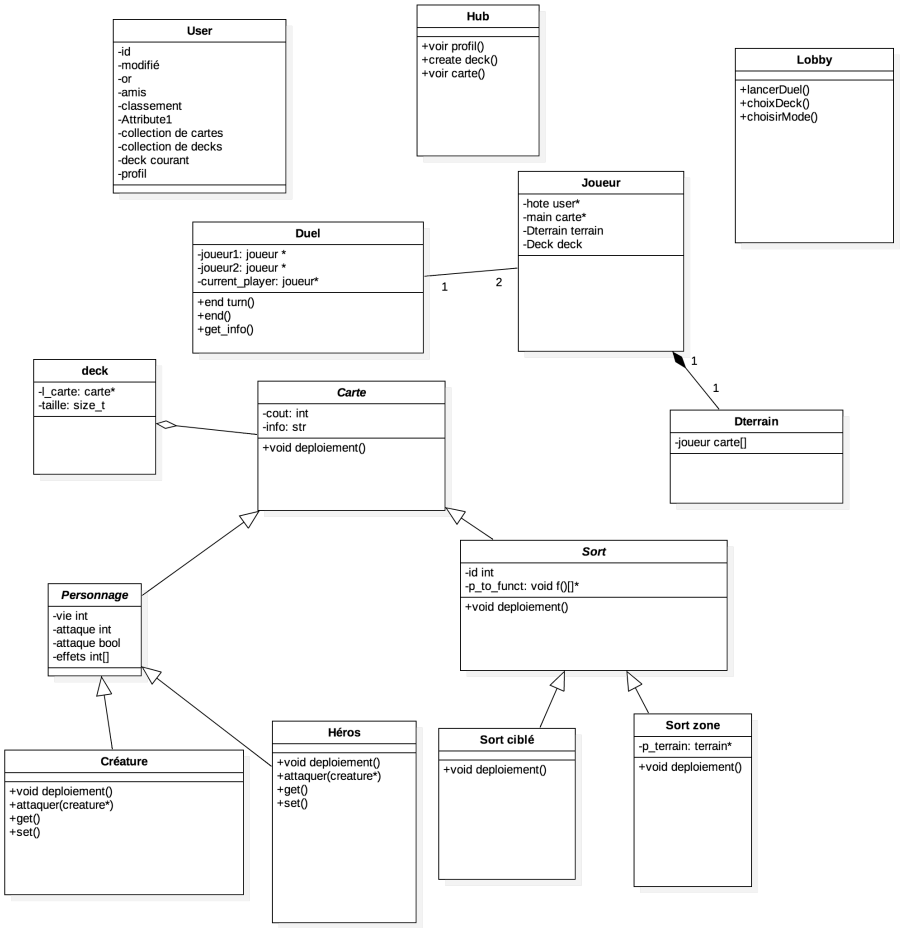


FIGURE 4 – Diagramme de classes général

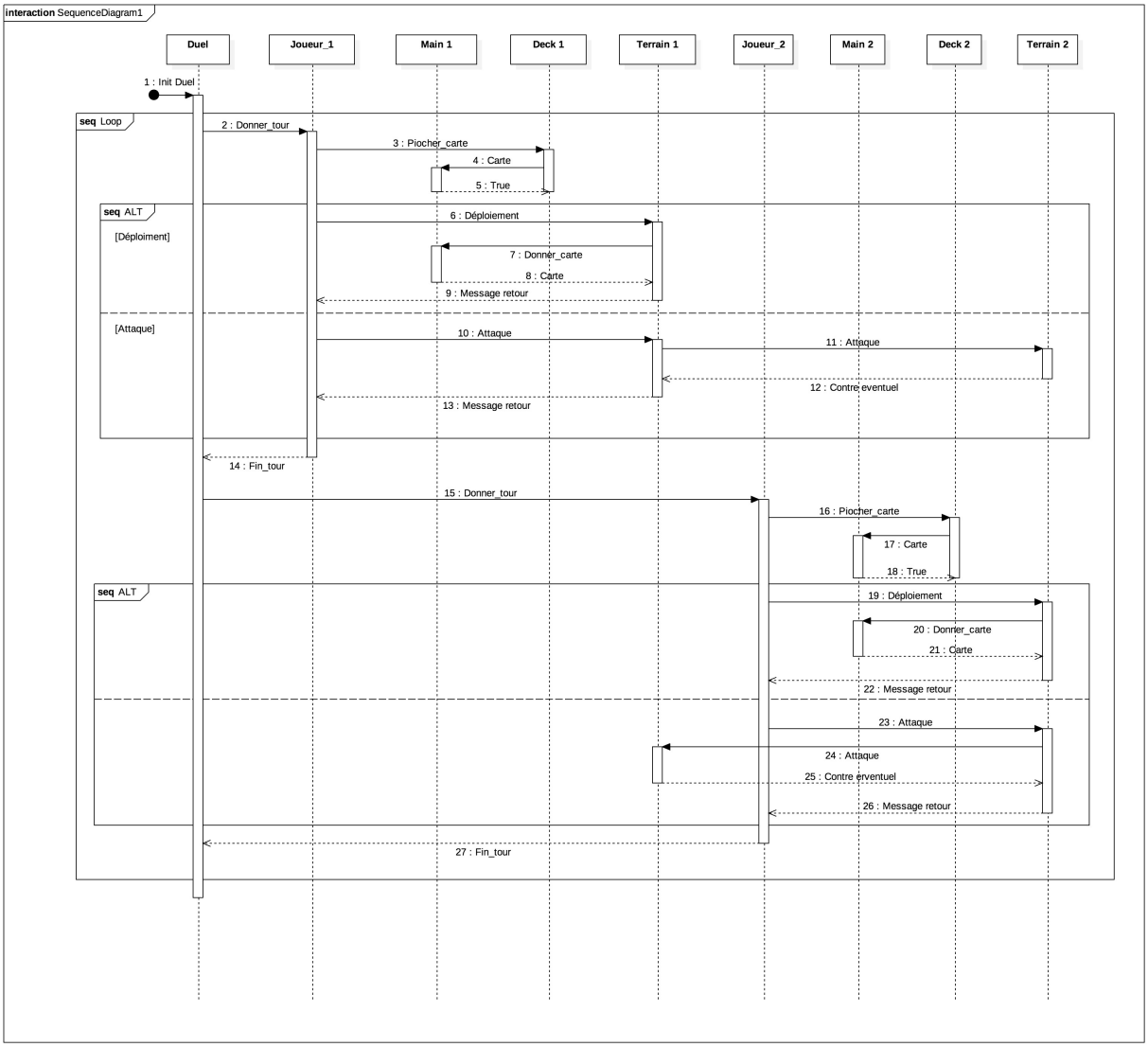


FIGURE 5 – Diagramme de séquence du duel

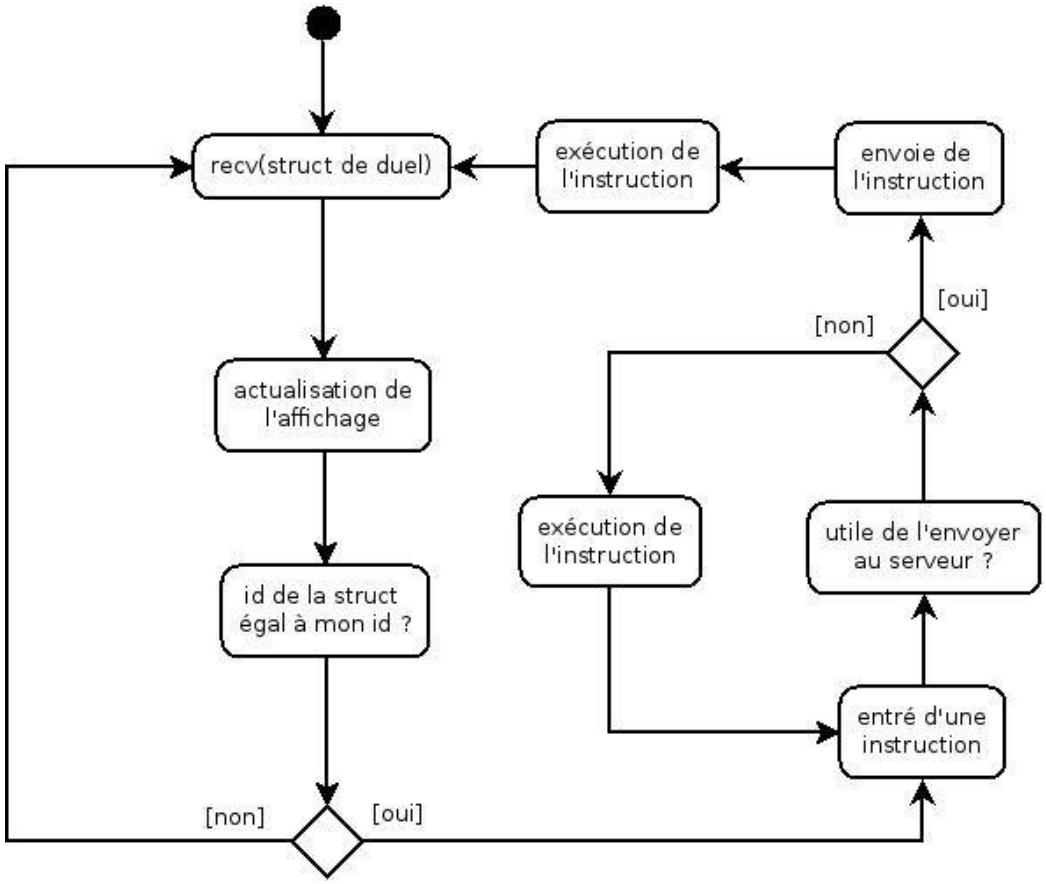


FIGURE 6 – Duel côté client

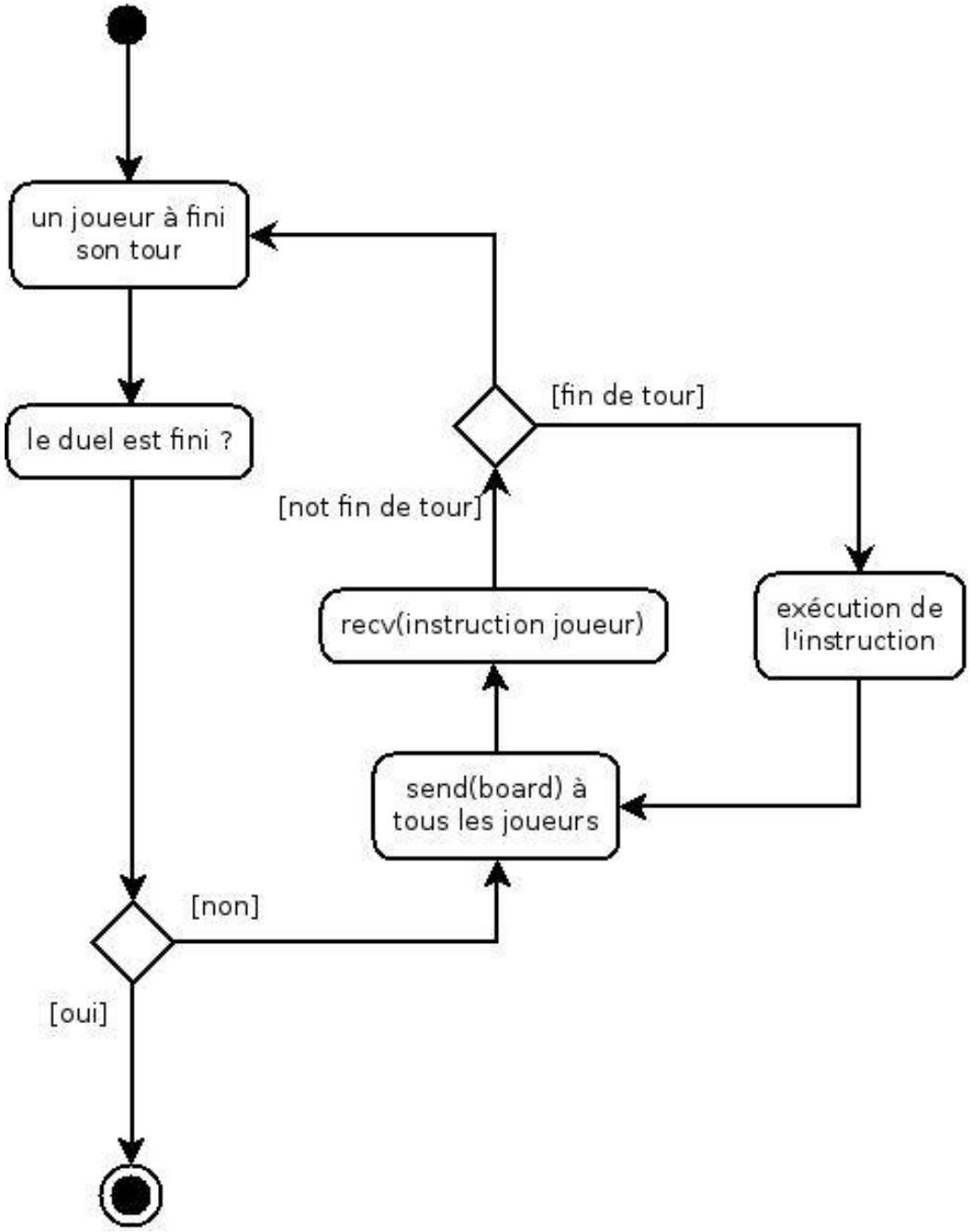


FIGURE 7 – Duel côté serveur

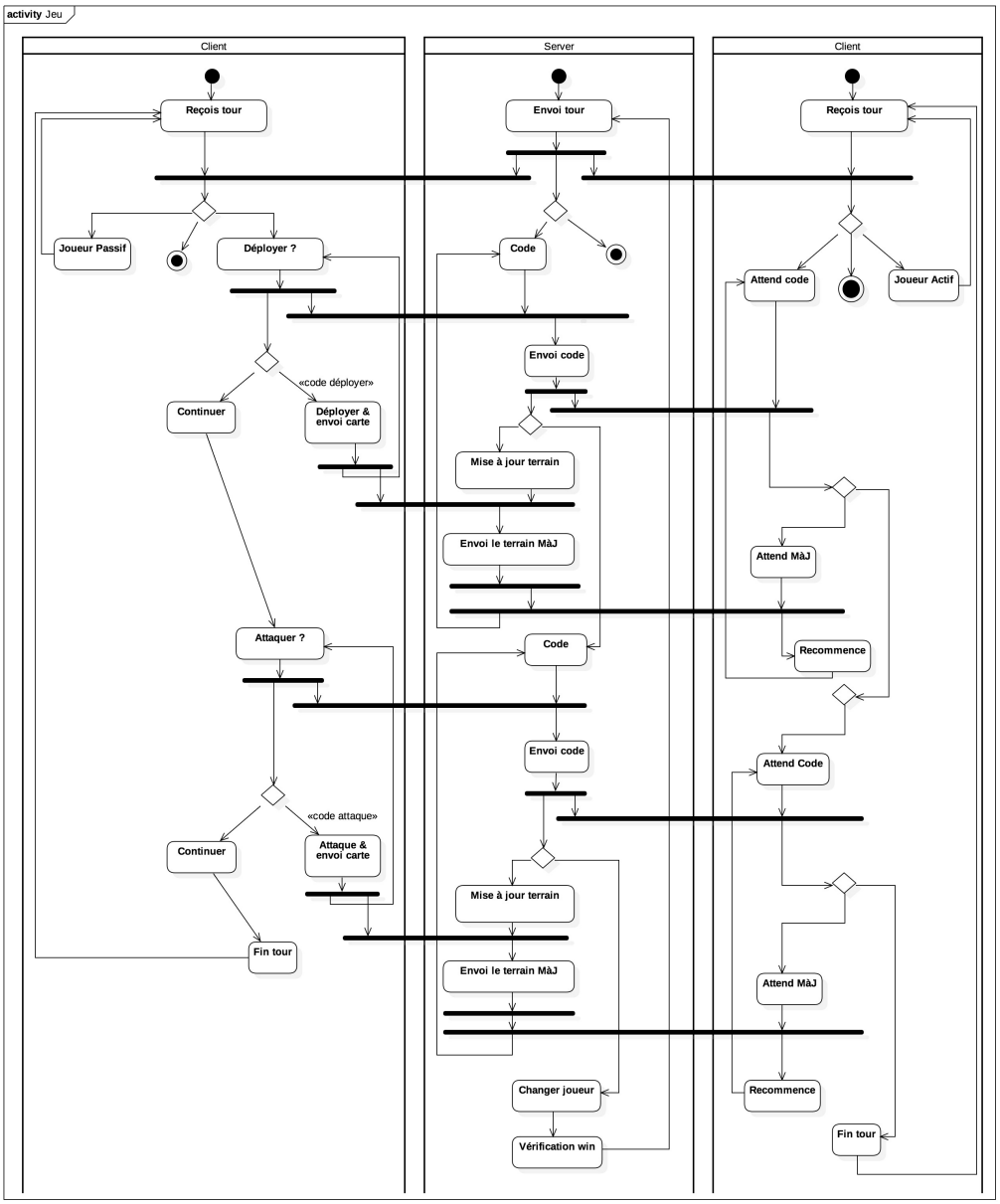


FIGURE 8 – Diagramme d'activité détaillé des interactions client-serveur lors d'un duel

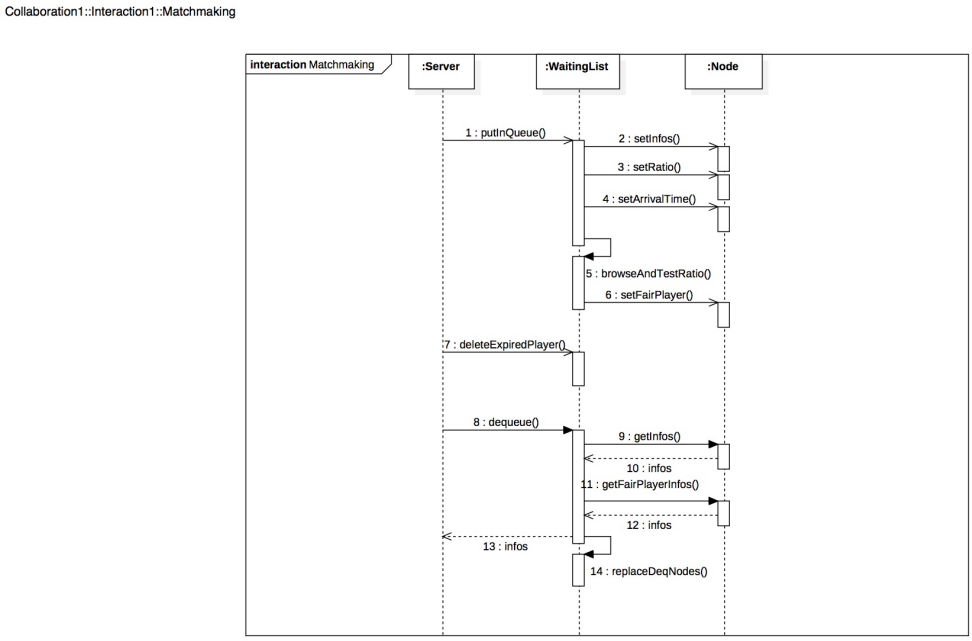


FIGURE 9 – Diagramme de séquence du matchmaking

4 Modules

Afin de faciliter le développement du projet et d'obtenir une organisation cohérente, nous avons décidé de découper le développement du programme en différents modules. Ces modules sont détaillés ci-dessous.

5 Module Duel

Cette section reprend les détails du gameplay ainsi qu'une explication générale de leur implémentation.

5.1 Gameplay

Lors de son tour, un joueur a le choix entre les différentes actions suivantes :

- Déployer une créature
- Lancer un sort
- Attaquer une créature adverse avec l'une de ses créatures
- Mettre fin à son tour

Le joueur peut effectuer autant d'actions qu'il le souhaite, dans les limites de ses cartes et de ses points de force bien sûr.

Le terrain de jeu de chaque joueur est divisé en 3 parties : Une ligne de 3 emplacements à l'avant, une ligne de 4 emplacements à l'arrière, et enfin le héros. Le héros et les créatures posées sur la ligne arrière ne peuvent être attaquée que si la ligne avant de l'adversaire est vide.

De nombreux effets pour les sorts et les créatures ont été implémentés. Les sorts peuvent notamment soigner une créature mais aussi lui faire des dégâts, la tuer, la booster, l'empêcher d'attaquer ou encore influer sur ses effets passifs (détaillés ci-après). Les sorts peuvent cibler une créature en particulier ou une créature aléatoire, avoir un effet de zone, etc.
Les créatures peuvent quant à elles avoir des effets passifs parmi cette liste :

- **Lightspeed :** peut attaquer le tour où elle est déployée.
- **Deflector shield :** les premiers dégâts subis sont réduits à 0.
- **Long aim :** peut cibler n'importe quelle créature même si la première ligne adverse n'est pas vide.
- **Armouring (n) :** lorsqu'elle est attaquée, les dégâts subis par la créature sont réduits de la valeur de ses n points d'armure.
- **Mind barrier :** Insensible au sorts.
- **Ataru (n) :** La créature peut attaquer n fois chaque tour.
- **Last breath :** La créature lance un sort à sa mort.
- **Ingage :** La créature lance un sort lorsqu'elle est déployée.
- **Begin turn effect :** La créature lance un sort au début de chaque tour.
- **Attack effect :** La créature lance un sort chaque fois qu'elle attaque.

5.2 Communication avec les clients

Lorsque le duel est lancé, le serveur communique aux clients toutes les informations dont ils ont besoin pour afficher le jeu et y jouer, à savoir :

- Les cartes contenues dans le deck du joueur adverse
- Les cartes contenues dans son deck ainsi que les cartes contenues dans sa main
- Les créatures déployées sur le terrain
- Les informations concernant les 2 joueurs
- Les informations permettant au joueur de savoir si c'est son tour.

Ces informations sont envoyées après chaque opération effectuée par le serveur.

5.3 Choix d'implémentation

Certaines parties du gameplay nécessitent de faire des choix non-explicités par le client. Ces principaux choix sont repris ci-après.

- **Créatures avec ingage :** Comme expliqué ci-dessus, les créatures possédant l'effet "ingage" lancent un sort lorsqu'elles sont déployées. Si ce sort nécessite une cible, l'utilisateur doit indiquer celle-ci. Il a été décidé que si l'utilisateur entre une mauvaise cible, une erreur lui est renvoyée et la créature lui est déployée. En revanche, si l'utilisateur n'entre pas de cible, la créature est déployée sans lancer son sort.

- **Mind Barrier :** Les créatures avec mind barrier sont insensibles au sorts. Lorqu'un joueur cible une telle créature avec un sort, une erreur lui est renvoyée et le sort n'est pas lancé. En revanche, un sort choisissant une créature aléatoire peut choisir une créature avec mind barrier, qui en absorbera alors les effets ! Que les effets soient positifs ou négatifs, ils ne peuvent affecter la créature choisie aléatoirement, le sort est donc lancé mais sans effet. Quant aux sorts de zone, ils infligent simplement leurs effets à toutes les créatures de la zone ciblée ne possédant pas mind barrier.

- **La gestion de la mort des créatures :** Dans certains cas, la gestion de la mort des créatures peut amener à faire des choix concernant l'ordre des actions à effectuer, nottament lorsque plusieurs créatures avec l'effet "last breath" meurent lors du même tour. Dans ce jeu, cette gestion est effectuée comme suit : Après chaque action susceptible d'avoir tué une ou plusieurs créatures, une "death handler" est appelé. Celui-ci parcoure le terrain à partir d'une position aléatoire et, lorsqu'il rencontre une créature morte, il lance l'exécution de son effet de "last breath" si nécessaire. Ensuite, il enlève la créature du terrain et passe à la suivante. Lorsque toutes les créatures ont été observées, le "death handler" est rappelé si l'éxécution d'un effet a eu lieu. Sinon, l'exécution s'arrête là.
  Il est néanmoins nécessaire de soulever un point important dans cette gestion de la mort des créatures. L'effet "last breath" d'une créature morte peut toucher d'autres créatures mortes qui n'ont pas encore été analysée. Cet effet pourrait potentiellement infliger des dégâts à cette créatures, auquel cas les dégâts seront perdus, ou éventuellement soigner cette créature, la ramenant ainsi à la vie (si le soin est suffisant) !

Les effets des "last breath" peuvent ainsi être assez imprévisibles, ajoutant une touche de fun et de sigularité au jeu.

5.4 Diagramme de classes

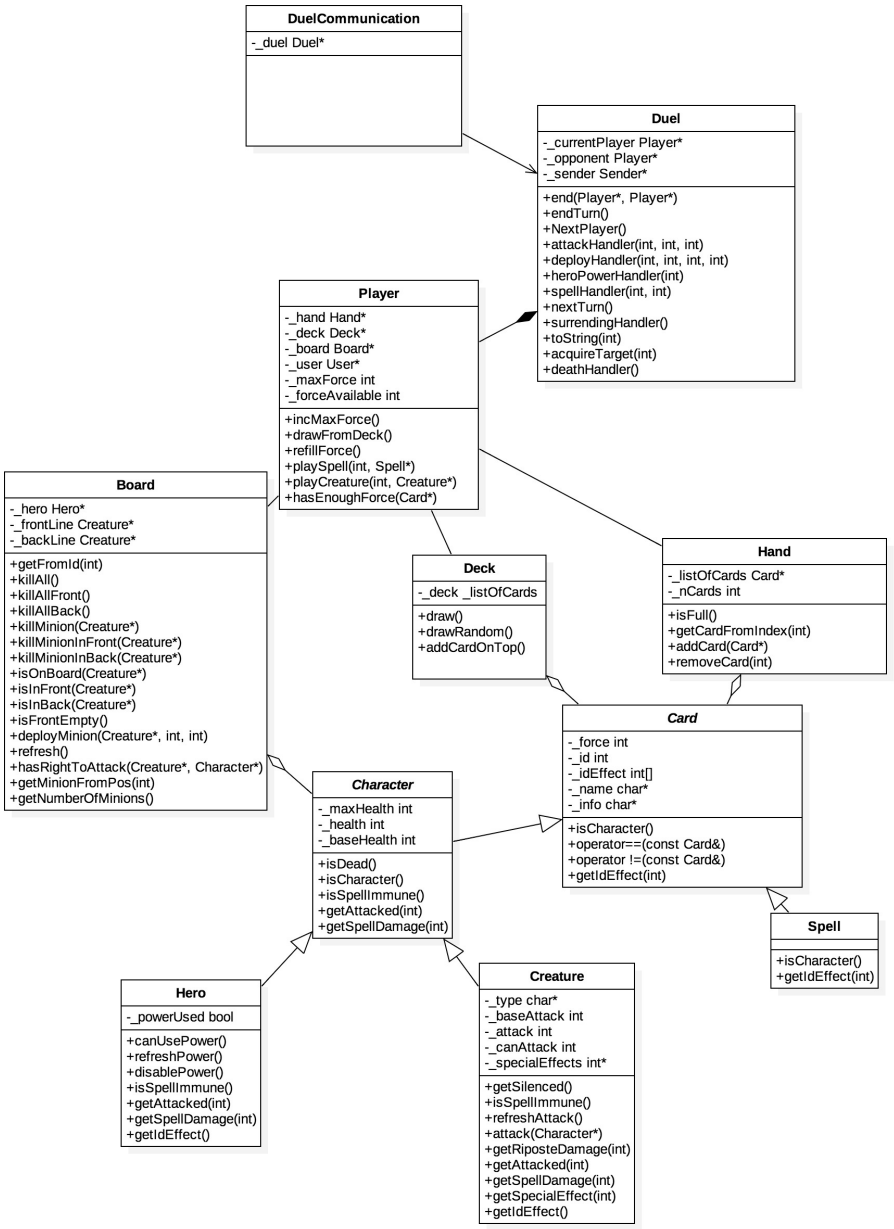


FIGURE 10 – Diagramme de classes du duel

6 Module Communication

6.1 Architecture générale

6.1.1 Mode de fonctionnement

L'idée de base était de s'affranchir completement de la contrainte des stream sockets qui consiste à faire suivre chaque send par un recv. Il peut sembler ne s'agir que d'un détail, mais le fait est qu'une très grande partie du programme doit pouvoir gérer des événements produits par un joueur, ce qui veut dire un envoi (quasi) immédiat d'informations, il est hors de question d'attendre un recv avant d'envoyer une commande qui devrait etre executée à l'instant.
Pour permettre cette communication à double sens, nous nous sommes inspirés du mode de fonctionnement des pipes bidirectionels. C'est à dire que deux ports sont utilisés, le serveur écrit sur l'un uniquement (c'est à dire que concrétement il effectue d'abord un send et fini par un recv) et écoute sur l'autre (fonctionnement inverse.) Du coté client le principe est exactement le meme, mais avec les ports inversés.
Cette façon de faire a d'énormes avantages, mais elle implique également la présence d'une boucle infinie qui attend de recevoir une information via le socket de chaque coté (client et serveur.) Cette idée peut sembler très mauvaise, d'autant plus qu'elle est utilisée à de nombreux endroits pour plusieurs raisons différentes (voir plus loin) mais un soin tout particulier a été apporté quant au caractère bloquant des instructions utilisées dans ces boucles. De cette manière le busy waiting est évité et le programme reste IO-bound dans son ensemble. (Ce qui devrait etre la norme pour un jeu video.)

6.1.2 Architecture du client

Plusieurs objets présents sur l'executable client nécessite un accés au serveur, dans un soucis de facilité, il a été décidé de cacher absolument les détails de la communication à ces objets et de leur fournir une interface "prete à l'emploi" avec laquelle la connection, les contraintes bas niveau comme la conversion des données en chaines de caractères ou les contraintes dues aux send/recv (évoquées plus haut) n'ont rien à voir. La meilleure illustration de ce choix est la DataBaseInterface, nous y reviendront plus tard.

6.1.3 Les interfaces

La stratégie adoptée pour transmettre un message d'un endroit du code client au serveur consiste en l'implémentation d'interfaces dédiées. Bien sur, avec plusieurs interfaces, plusieurs messages peuvent potentiellement etre envoyés en meme temps, il a donc fallut gérer l'accés concurrentiel aux méthodes d'envoi. Il a également fallut taguer les messages envoyés pour les traiter efficacement sur le serveur. Ces tags se présentent sous la forme de suffixes (définis dans networkUtils.h.) Chaque interface possède un suffixe qui lui est propre et si c'est pertinent, les méthodes d'une interface possèdent également un suffixe.

ClientInterface La base des communications est la ClientInterface. Elle est très particulière dans le sens où elle est quasi entièrement statique, du à son statut de "lien" entre les fonctions C du module NetworkFunctions très bas niveau et le reste du programme orienté objet et codé en C++. Cette "staticité" aurait pu etre un problème lors des constructions successives via l'instanciation de classes filles ou des destructions d'un de ces classes qui "tueraient" la base commune. Pour palier à cette contrainte, elle ne se construit qu'une seule fois (avant meme le lancement du jeu en tant que tel) et à chaque instanciation d'une classe fille, elle se contente d'incrémenter un compteur de référence. Symetriquement, ce compteur est décrémenté à chaque destruction, et lorsqu'il atteint zéro, l'objet ClientInterface est effectivement détruit.
La ClientInterface a pour role de se connecter au serveur à sa création, et assure les communications par la suite. C'est à dire qu'elle propose deux méthodes d'envoi de messages, la première se contente d'envoyer un message sans attendre de réponses, la seconde est à utiliser lorsqu'une réponse du serveur est attendue (majoritairement dans le cas de requetes en base de données.) Un thread daemon contenant une boucle d'écoute se réveille lorsqu'un message est reçu du serveur et l'envoie au bon endroit (connu grace au système de suffixe évoqué plus haut)
Lorsque la ClientInterface a été pensée, nous n'avions pas encore une idée très précise de ce qui viendrait se greffer dessus. En plus du fait qu'elle reste tout de meme d'un niveau assez bas, nous avons été amené à faire une entorse au paradigme orienté objet en utilisant des pointeurs vers les handlers (statiques) des classes filles auxquelles transmettre les messages reçus. Le dispatching pourrait donc certainement etre amélioré, mais par manque de

temps, il n'a jamais été modifié.

DataBaseInterface La dataBaseInterface se comporte exactement comme la dataBase réelle présente sur le serveur, c'est à dire que les méthodes sont exactement les memes, ce qui donne l'illusion d'avoir la database présente sur le client. Elle comporte énormément de méthodes, chacune possédant sont suffixe propre. A l'appel d'une de ces méthode, une requete est construite en (pour la transmettre sur le réseau, elle doit etre sous forme de chaine de caractères, nous y parvenons via des casts et une bonne dose d'arithmétique de pointeur) pour etre décodée sur le serveur ensuite et traitée adéquatement.
La plupart de ces méthodes sont en fait des getters, ce qui implique de renvoyer une réponse. Etant donné que l'envoi du message se fait sur le thread principal du programme et que la réception se fait sur un autre, il était essentiel de bloquer le thread principal le temps de recevoir la réponse. Ce problème est géré grace à un buffer dans lequel est stockée la réponse reçue par le daemon d'écoute et une paire de mutex (empty et full) qui permettent de faire attendre un thread ou l'autre le temps que la requete arrive (ou inversemment, le temps que la réponse à la requete précédente ait été traitée.) Pour éviter les problèmes de collisions entre les différentes réponses des différentes requetes, un mutex doit aussi etre utilisé lors de chaque envoi de message, qu'il soit de type requete ou de type "standard".

DuelistInterface La DuelistInterface est volontairement très simple, elle ne fait que servir de relai entre la ClientInterface et DuelCom (classe fille qui gère les communications duel) et se contente d'ajouter son suffixe lors d'un envoi de message et d'appeler une méthode virtuelle pure qui traite les messages qui lui sont adressés dans la classe héritière. La raison de cette simplicité tiens d'abord du fait que toutes les infos envoyées et reçues sont sous la forme de structs, il était donc plus facile de proposer une interface minimaliste et de caster le struct (POD) à envoyer comme une chaine de caractères. Ensuite, elle marque la jonction du travail de deux groupes distincts, DuelistInterface ayant été implémentée pas "l'équipe communications" et DuelCom par "l'équipe duel" ce qui explique cet héritage d'une classe abstraite.

ChatInterface ChatInterface réagit lors de la réception d'une demande de chat de la part d'un amis et permet de récupérer l'adresse Ip d'un amis à qui envoyer des messages. Le système de chat fonctionne en peer to peer, raison pour laquelle il est nécessaire d'utiliser un système de getter d'Ip. Une fois de plus, l'abstraction voulue du réseau à travers les interfaces n'a pas été poussée jusqu'au niveau souhaité pour une question de répartition du travail. ChatInterface ne cache donc pas les détails bas niveau des connections entre amis mais se contente de renvoyer une adresse Ip à une classe fille qui se charge du reste du travail.

6.1.4 Architecture du serveur

Le serveur consiste en trois types de processus différents, le processus père, à l'écoute des connections entrantes, il possède plusieurs threads et ADT (décrits plus loin) permettant de gérer notament le chat et les requetes de duel. Les processus fils se chargent d'écouter un client chacun et de transmettre les éventuels messages des autres processus vers le client. Il est également en charge de gérer une déconnection du client et les requetes en dataBase. Le dernier type de processus, est le processus Duel, un fils créé lorsqu'il y a match entre deux joueurs voulant se battre. Il communique avec deux de ses frères en charge des communications avec les clients.
La raison pour laquelle une architecture en plusieurs processus a été choisie est la sécurité et la robustesse du serveur. En effet, si pour une raison ou une autre un processus devait se crasher, mieux vaut que ce soit un processus fils plutot que le processus contenant la boucle listen-accept.

Communication inter-processus Il est crucial que les différents processus puissent communiquer entre eux, lors d'une requete de chat pour savoir si son amis est connecté, et le cas échéant, récupérer son adresse Ip. Et lors d'une requete de duel, et ensuite pendant le duel. les communications père/fils se font via des pipes anonymes, les communications via des pipes nommés. Une autre architecture avait d'abord été envisagée : l'idée était de transmettre les sockets réseau des processus fils Client/Serveur au processus duel, de telle sorte qu'un duel communique directement avec les client distant plutot que via un intermédiaire. C'était possible grace aux sockets Unix qui possède cette caractéristique de pouvoir transmettre des files descriptors ouvert d'un processus à un autre, mais les restrictions des pcs en salles machines ne nous

permettaient pas de le faire. Nous avons donc opté pour une architecture plus simple et un peu plus "telephone arabe", se servir du processus fils comme relai.

Processus Père Le processus père commence par initialiser trois ADT importantes : la waitingList, essentielle pour effectuer le matchmaking, l'IpContainer, qui se charge de stocker les adresses Ip des joueurs connectés ainsi que leur référence vers un thread d'écoute (type thread daemon à lécoute sur un pipe) et la mailbox, qui sert de "reserve" entre un thread qui se comporte comme un consommateur et les threads producteurs à l'écoute des processus fils.
En plus de ces Adt, deux threads sont créés, l'un effectuant le matchmaking, réveillé à chaque fois qu'un joueur est ajouté à la file d'attente, l'autre se réveillant lorsqu'une requete d'un fils a été placée dans la mailbox. A chaque nouvelle connection, juste avant le fork. chaque thread d'écoute d'un fils se charge d'ajouter son client dans l'IpContainer, et une fois que la deconnection est detectée, il le retire avant de se couper lui meme. De telle sorte qu'il ne reste pas de thread fantome sur le processus père. Le thread linkPlayers se réveille simplement quand un matchmaking est possible et se charge de trouver des adversaires de niveau à peu prés équivalent autant que possible, quant au thread traitant les requetes des fils, il se charge soit d'effectuer les requetes d'adresses Ip soit de placer un PlayerDescriptor dans la waitingList en attendant le meilleur match possible.

Processus Fils Le processus fils possède la dataBase, toute requete dataBase est directement gérée dans le fils. Les requetes de chat et de duel sont transmises au parent qui dispatche ensuite les messages adequats aux threads ou Adt concernés. Il sert egalement de relai entre les processus père et duel et le client lorsqu'un message doit etre envoyé au client (en cas de requete duel de la part d'un amis ou de demande de duel) Il se caractérise par un thread d'écoute du client (de type daemon) un thread d'écoute du processus duel (lancé uniquement en cas de duel) et un thread d'écoute du père (le thread principal dans ce cas.) Les tentatives de reconnection lors d'une deconnection du client sont aussi gérées dans ce processus. Si jamais le joueur n'est pas parvenu à se reconnecté dans la minute, le processus envoie un message au père pour signaler qu'il faut supprimer le thread qui se charge de l'écouter, et fini par se couper de lui meme.

Processus Duel Enfin le processus duel, il possède un sender et un receiver qui lui meme contient un objet duel. Le sender sert à envoyer des messages via les pipes nommés du processus duel aux processus fils qui relaient ensuite l'information aux clients, le receiver quant à lui écoute les pipes (via deux threads daemons) et transmet les messages reçus du client adressés au duel. Duel, Sender et Receiver interagissent entre eux pour faire jouer les duelistes l'un contre l'autre.

7 Module Bases de données

Le projet est composé d'un module faisant temporairement office de base de donnée. Ce module permet de stocker, dans le temps, l'ensemble des informations relatives aux joueurs et aux cartes dont le serveur et les clients ont besoin. Dans le cadre de ce projet, il a été décidé de mettre au point une base données pour empêcher de devoir tout stocker dynamiquement en mémoire de manière à éviter un *bloatware*. Ce choix à été fait, bien que de toute évidence, le nombre de joueur inscrits ne sera jamais assez conséquent pour pleinement justifier ce choix. Cependant, un tel système évite que des informations ne soient pas à jour suite a une interruption inopinée du programme.

7.1 Informations relatives aux joueurs

La base de données contient l'ensemble des informations demandées :

- un pseudonyme,
- un mot de passe,
- une collection de cartes,
- une collection de decks,
- une liste des autre membres avec qui il est amis.

Cet ensemble est principalement utilisé lors d'opérations telle que la visualisation du profil d'un ami, lors de la création d'un deck, lors de la visualisation de sa collection de cartes etc.

7.2 Informations relatives aux decks

Comme pour les joueurs, cet ensemble reprend principalement les informations demandées et suivant les règles établies. Celle-ci est, dans les grandes lignes, composée :

- de noms,
- d'ensembles de cartes,
- de la faction à laquelle un deck donnée.

Cet ensemble est peu utilisé par le client et est surtout utile lors de l'entrée d'un joueur en duel.

7.3 Informations relatives aux cartes

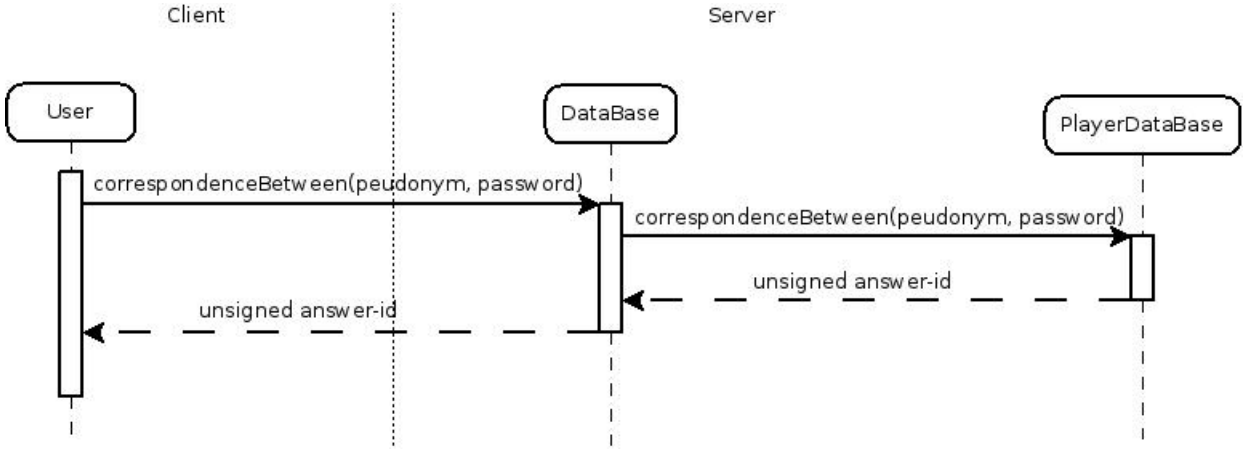
La base de données contient aussi l'ensemble des cartes disponibles sur le serveur. Celles-ci ont comme caractéristiques :

- un nom,
- un coût (en),
- un coefficient d'attaque,
- un coefficient de vie,
- une faction,
- un effet,
- un type.

Ces données sont utilisées lors de l'entrée en duel ou lors des différentes demandes de visualisation de cartes que le client pourrait faire.

7.4 Accès aux données

Typiquement, le client ne contient (presque) pas de données, il doit donc aller les chercher à chaque fois dans la base de donnée. Ce mécanisme, bien que couteux en performance (temps de transfert), permet d'être sûr de l'intégrité des données soumises au serveur. Par exemple, lors d'une tentative de connexion, le client demande de vérifier la correspondance entre le pseudonyme entré et le mot de passe entré, ce à quoi le serveur lui répond par son identifiant ou par zéro.

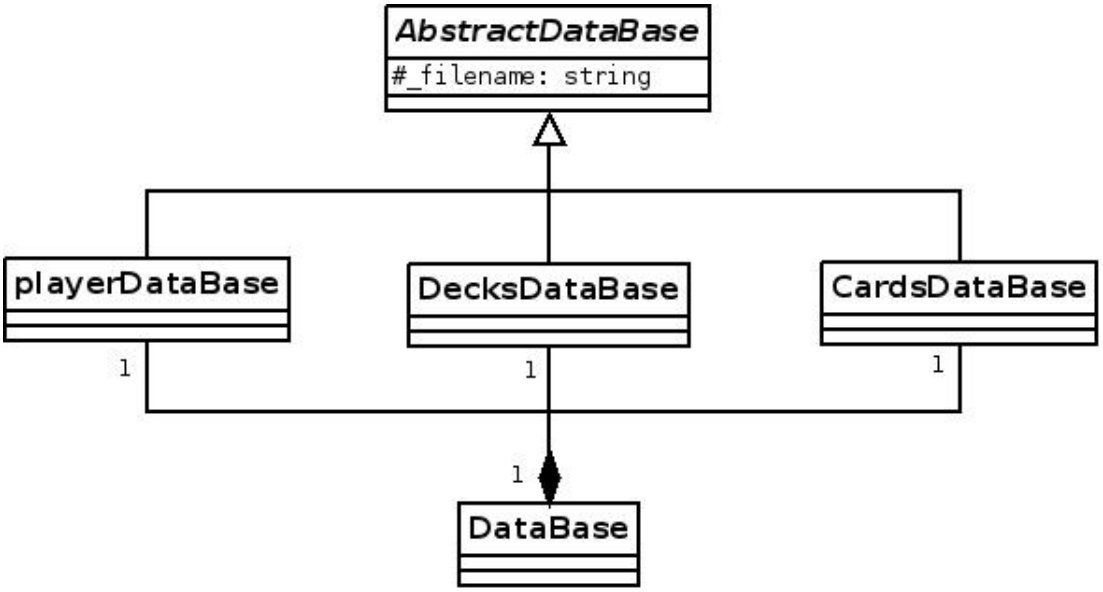


Bien que les valeurs renvoyées par le serveur soient différentes, chaque requête envoyée au serveur suit la même logique.

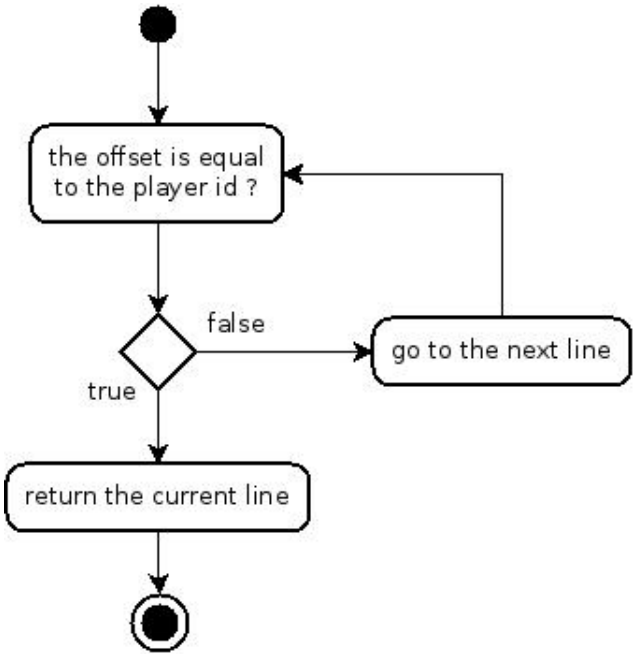
7.5 Structure de la base de données

La structure du module est composée de cinq classes :

- une classe depuis laquelle se font les requêtes (`DataBase`)
- trois classes permettant la lecture et l'écriture sur leur fichier respectif (`PlayerDataBase`, `DecksDataBase`, `CardsDataBase`)
- une classe abstraite (dont les trois si-dessus héritent) reprenant des méthodes communes aux classe de lecture et écriture.



Une telle structure est motivée par le fait que chaque classe de lecture et écriture offre une abstraction dans le sens où on peut facilement récupérer une information sans connaitre la structure interne de leur fichier respectif. Et que pour des raisons d'abstraction et de maintenance, il est plus aisé d'avoir une seul classe faisant "interface" et redirigeant les requêtes (`DataBase`) car si la manière de stocker les données change, la manière de les obtenir ne changera point. Finalement, Les trois classes de lecture/écriture héritent de la même classe car la structure de leur fichier respectif possèdent des points communs. Un exemple de point commun est le fait que l'identifiant d'un objet (joueur, deck ou carte) correspond à son décalage dans le fichier. Ainsi, l'accès à un élément d'un objet donné passe par l'obtention de la ligne lui correspondant. L'obtention de cette ligne se fait de la manière suivante :



8 Succès

Un succès est un événement qui arrive lorsque une certaine statistique est atteinte (on les dit "débloqués"). Les succès, dans ce projet, couvrent plusieurs aspects du jeu (succès concernant les factions, succès concernant les réussites sans distinction des factions, succès concernant les différents héros en jeu). voici une liste exhaustive des différents succès disponible en partie :

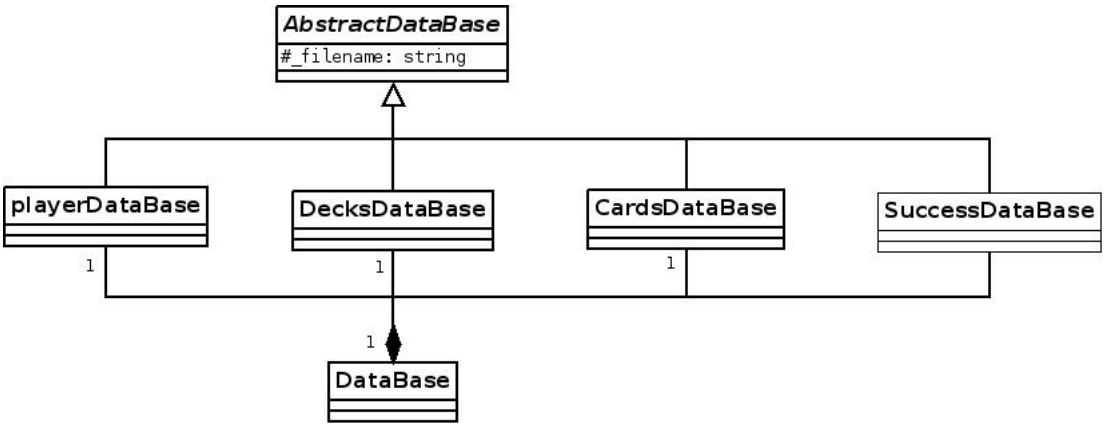
- première connexion au serveur.
- possession de toute les cartes où les répétition sont prisent en compte.
- possession d'un certain nombre de cartes (100, 150 et 200) sans prise en compte des répétitions.
- avoir gagné un certain nombre de duels (10, 50, 100).
- avoir perdu un certain nombre de duels (10, 50, 100).
- avoir gagné un certain nombre de duels d'affilé (3, 5, 10).
- avoir perdu un certain nombre de duels d'affilé (3, 5, 10).
- avoir battu un certain nombre de decks de type sith (10, 20).
- avoir battu un certain nombre de decks de type jedi (10, 20).
- avoir gagné 10 duels avec un hero (Luke Skywalker, Snoke, Liberator Star Destoyer, Imperial Star Destroyer, Yoda et darth Sidius).

- avoir joué un certain nombre de duels sur le serveur (1, 10, 50, 100, 1000).

Chaque joueur sur le serveur possède son propre avancement par rapport aux différents succès disponible. Ainsi, chaque joueurs possède des champs supplémentaire en base de données.

8.1 Structure de la base de données

La classe faisant interface avec la base de données concernant les succès s'intègre dans l'architecture déjà présente comme représenté dans le diagramme si dessous.



9 Expérience de faction

En plus d'avoir la possibilité de débloquer des succès, il est aussi possible, dans ce projet d'augmenter son expérience "in-game" grâce à des victoire (et même des défaites) avec un héro d'une faction donnée. L'expérience accumulée permet de monter de niveaux. Ces niveaux donnent accès à certain cadeaux telles que :

- de nouveaux héros de la faction dont vous avez montez le niveau. Ces cartes sont uniquement obtenable via cette manière.
- des cartes aléatoires qui viendront compléter votre collection.

10 Module Matchmaking et Chat

10.1 Le chat

Afin de proposer un jeu agréable, il convient naturellement de fournir au joueur un chat avec lequel il pourra discuter avec ses amis. Néanmoins, limité dans un premier temps par l'utilisation du terminal linux comme interface, le chat sera assez rudimentaire. En effet il ne sera possible que de discuter avec une seule personne à la fois.

Le chat est ici un programme indépendant du jeu ou plus précisément deux programmes indépendants. Dans le but de simplifier la tâche, le chat est implémenté de sorte que les deux joueurs y communiquant soit directement connectés l'un à l'autre. Le joueur ayant démarré un chat reçoit alors le rôle de "serveur" et l'autre joueur, invité dans le chat, reçoit le rôle de "client". Les termes "serveur" et "client" sont à nuancer car il est évident de constater que tant le "serveur" que le "client" sont égaux.

Deux programmes "chat", assez similaire, sont donc fournis avec le jeu. Le premier, appelé `chatHost`, est lancé chez le joueur créant une conversation et le second, nommé `chatGuest`, est exécuté chez le joueur invité dans le chat. Ce dernier est donc lancé à l'insu du joueur contrairement à `chatHost`. Afin d'accomplir une telle action, il est nécessaire que le serveur du jeu (à ne pas confondre avec le "serveur" dans le cas du chat) envoie un signal au jeu du joueur invité dans le chat pour que celui-ci lance, sans la connaissance du joueur, le programme `chatGuest` dans un tout autre terminal. Ainsi, de

l'autre côté, lorsque le joueur démarre le chat, le programme `chatHost` est lancé sur sa machine (également sur un autre terminal que celui où se déroule le jeu) et attend ensuite la connexion de son vis-à-vis. Une fois connectés, les deux joueurs peuvent alors s'envoyer des messages via le terminal.

Pour ce faire, chacun des deux programmes chat est divisé en deux processus. Le processus père s'occupe de recevoir les éventuels messages et de les afficher dans le terminal. Le processus fils quant à lui permet au joueur d'entrer un message puis se charge de l'envoyer.

En cas de déconnexion d'un des deux correspondants, l'autre sera averti d'un message écrit en rouge dans le terminal. Il lui faudra alors en relancé un autre, étant donné que chaque programme chat n'est conçu que pour accueillir une seule fois une connexion.

Un problème rencontré et dû à l'utilisation d'un terminal était le fait que lorsqu'un joueur en train de rédiger un message (et n'a donc pas encore appuyé sur enter pour l'envoyé) et qu'il reçoit à ce moment même un message de son amis, le texte qu'il était un train de rédiger et le message reçu "entrent en collision". En effet la ligne courante où le joueur est en train d'écrire est vu comme libre (puisqu'il n'a pas encore appuyé sur enter) par le processus transmettant les messages reçus et affiche donc le dit message sur cette même ligne.

Néanmoins, le texte que rédigeait le joueur lors de la réception d'un message n'est pas écrasé pour autant. Un simple saut à la ligne permet au joueur de continuer à voir ce qu'il a écrit. Le début et la fin de son texte seront simplement séparés par le message reçu alors en train d'écrire.

Il va de soit que lorsqu'une interface graphique sera implémentée, ce problème sera résolu. Le chat pourrait alors ne plus être du "peer-to-peer" ce qui permettrait, si tous les messages envoyés et reçus passent par le serveur du jeu, de stocker les conversations entre joueurs ou encore de modérer les propos des joueurs en censurant ici et là les possibles insultes potentiellement proférées par l'un ou l'autre utilisateur. A noter que cela permettrait aussi des conversations de groupes entre plusieurs joueurs.

10.2 Le matchmaking

Dans tout jeu multijoueur qui se respecte, le matchmaking, autrement dit la création des salons de matches, est un élément primordial. Il ne doit pas être laissé au hasard et doit dépendre de plusieurs facteurs comme par exemple le niveau des joueurs, leur localité etc.

Lorsqu'un joueur veut lancer un duel, le matchmaking doit s'opérer. Le joueur est alors placé dans une file d'attente. Algorithmiquement parlant, cette file d'attente est représenté par un objet (instancié une seule fois) de la classe `WaitingList`. Cette classe possède plusieurs attribut dont les pointeurs de noeuds `firstNode, lastNode` et `lastInQueue` qui pointent respectivement vers le premier noeud de la liste (son prédécesseur est nul), le dernier noeud de la liste (son successeur est nul) et vers le noeud contenant les informations sur le joueur ayant été ajouté en dernier dans la file.. Un noeud est un objet de la classe `Node` et possède également plusieurs attribut dont un pointeur vers le noeud suivant, un vers le noeud précédent et l'information contenue dans le noeud (par exemple l'identifiant du joueur). Les deux autres attributs de `WaitingList` sont le nombre de joueurs en attente et le nombre de joueurs présents dans la liste pouvant être mis en relation (leur ratio victoire / défaite est similaire).

Le constructeur de `WaitingList` initialise la file d'attente avec un nombre de noeud défini. Ceux-ci ne contiendront donc pas encore d'information.

Lorsqu'un joueur veut lancer un duel et qu'il doit donc être mis dans la file d'attente, la méthode `putInQueue` est appelée. Deux cas sont alors possible. Soit la file est vide et le noeud associé au joueur est alors pointé par l'attribut `firstNode` et par l'attribut `lastInQueue`. Soit la file n'est pas vide (`firstNode` ne pointe alors pas vers `nullptr`), auquel cas il suffit de faire pointer l'attribut `NextNode` du noeud `lastInQueue` vers le noeud représentant le joueur. Ainsi, avoir un pointeur vers le noeud représentant donc le dernier joueur arrivé dans la file nous permet de rajouté un joueur dans la dite file sans pour autant la parcourir de bout en bout.

Coté serveur du jeu, un thread tourne en boucle pour vérifier si un joueur n'a pas fait une requête de matchmaking. Cette requête est écrite sur un pipe père / fils. Chaque processus fils du serveur s'occupe d'un joueur et en gère ses demandes. Du coté du processus père, avec lequel le thread décrit précédemment tourne en parallèle, un tableau de pipe (pour communiquer avec ses nombreux fils) devra être créé. Le thread effectue en boucle un read non bloquant sur chaque pipe afin de voir si un des fils n'a pas demander le lancement d'un duel. L'indice du pipe sur lequel un requête a été émise indique alors le socket sur lequel le client en question communique avec le serveur. Les numéros de sockets commençant à 1, le pipe d'indice 4 correspond au client communiquant sur le socket numéro 5. Le thread se charge alors de placer le joueur dans la file en appelant la méthode `putInQueue` décrite plus haut.

En parallèle au thread ajoutant des joueurs dans la file, un autre thread s'occupe quant à lui de sélectionner deux joueurs à mettre en relation afin qu'ils s'affrontent. Cette paire de joueur est choisie en fonction de leur ratio victoire / défaite. Pour ce faire, la méthode `browseAndTestRatio` de la classe `WaitingList`, appelée après l'ajout d'un joueur à la file, s'occupe de comparer deux noeuds (le dernier arrivé dans la liste avec tous les autres), plus précisément de comparer le ratio de ces deux joueurs représentés par ces noeud. Elle divise le ratio le moins élevé des deux joueurs par le plus élevé et si le quotient est égal ou supérieur à 0.75, c'est deux joueurs sont marqués comme pouvant être mis en liaison afin de disputer un duel. La méthode `deQueue` s'occupe en même temps de retirer deux joueurs marqués et de renvoyer un vecteur contenant les dits joueurs.

Finalement, deux noeuds ayant été retirés, la liste se doit d'être réordonnée afin de ne pas avoir de trous et donc que certains noeuds ne soient plus liés aux autres. C'est à cela que sert la méthode `replaceDeqNode`.

Dans le cas où un joueur devrait attendre trop longtemps notamment dû au fait qu'aucun joueur de son niveau ne soit disponible, il sera retiré de la liste après deux minutes de recherche d'adversaire. Puisque la progression d'un joueur dépend entièrement de son niveau, il serait injuste de faire s'affronter deux joueurs totalement déséquilibrés. Le joueur le plus faible n'aura pas l'occasion de gagner de meilleur carte et donc de gagner plus de combat, et le joueur plus fort aurait une très grande facilité à progresser bien que ce ne soit mérité. C'est pour cette raison qu'un joueur ayant un très faible ratio ne tombera jamais avec un joueur ayant un gros ratio.

Grâce à la méthode `getAvailability` de la classe `WaitingList`, il est possible de savoir si oui ou non il existe dans la file d'attente deux joueurs pouvant être mis en relation. Cela permet de ne pas inutilement appelé la méthode `deQueue` dans le cas où il n'y aurait pas de joueur à mettre en commun.

11 Module Affichage console

11.1 Partitionnement de l'affichage

La gestion de l'affichage a été découpée de manière à représenter de la façon la plus fidèle le "chemin" emprunté par l'utilisateur lorsqu'il utilise le programme. On peut donc scinder de l'utilisation en 3 parties qui sont :

- Inscription/Connexion
- Choix du service souhaité (jeu,classement,amis,...)
- Affichage du jeu

11.2 Inscription / Connexion

La première partie concerne donc l'inscription ou la connexion d'un utilisateur à la plateforme. Pour ce faire l'utilisateur doit passer par une première interface lui laissant le choix entre une inscription au service et une connexion via un compte existant. Nous avons décidé de placer cette première partie du service dans la classe Menu qui représente la page "publique" du programme. Voici une capture d'écran du terminal :

11.3 Choix du service

La seconde partie concerne le choix du service souhaité par l'utilisateur. Il y a donc 7 options mises à disposition du joueur :

- voir le profil du joueur ou de l'un de ses amis,
- voir les cartes que le joueur possède,
- créer un deck ou modifier un de ses decks,
- obtenir le classement de tous les joueurs,
- lancer un duel,
- se déconnecter,
- accéder à la page d'aide.

Voici une nouvelle capture d'écran du terminal du terminal :

L'ensemble de ces méthodes sont disponibles dans la classe nommée Hub qui permet de faire le lien entre la page d'accueil et le jeu en lui-même. Les quatre premières options sont également implémentées dans la classe Hub. Il ne nous reste plus qu'à détailler la dernière partie, à savoir l'affichage du jeu.

11.4 Affichage du duel

L'ensemble de l'affichage du jeu est effectué dans la classe Game. Cette classe permet d'imprimer au joueur les informations dont il a besoin afin de pouvoir jouer à savoir :

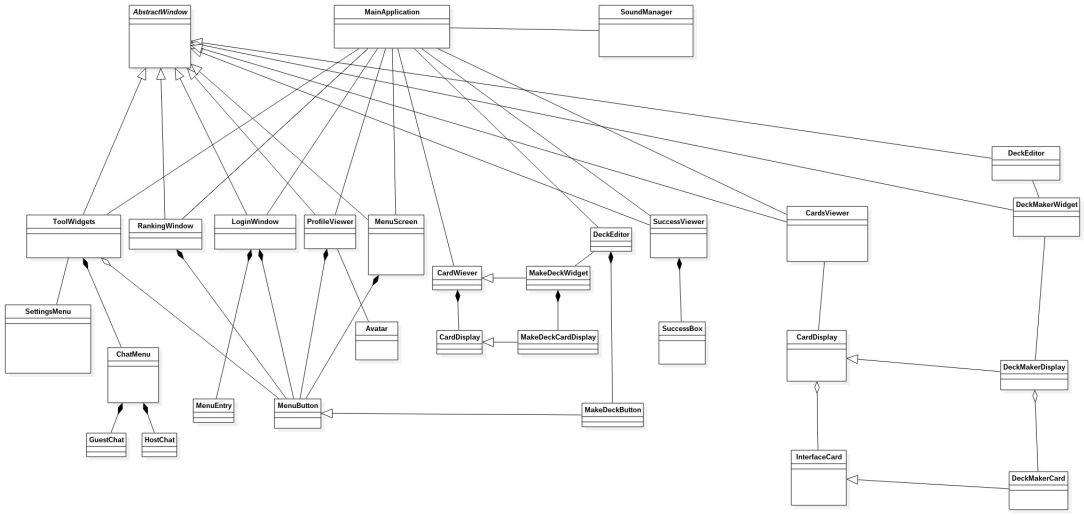
- Le nombre de cartes contenues dans le deck du joueur adverse
- Le nombre de cartes contenues dans son deck ainsi que les cartes contenues dans sa main
- Les créatures déployées sur le terrain
- Les informations concernant le joueur adverse (pseudo,vie,...)
- Les informations le concernant (pseudo,vie,...)

12 Module Interface graphique

La partie interface graphique du jeu a été réalisée en dernier lieu, venant se greffer sur le jeu déjà préconçu. Elle a été réalisée à l'aide de la bibliothèque "qt".
Beaucoup de moyens ont été mis en oeuvre pour que l'esthétique colle le mieux possible au thème du jeu, la saga "Star Wars". Ainsi, les images, les musiques, les avatars, les logos et même les sons sont en adéquation avec ce thème.
Les options disponibles sont les mêmes que dans l'affichage console, bien qu'elles soient évidemment plus visuelles et plus faciles d'utilisation.
Le profil, notamment, a été largement remanié de manière à ce que l'utilisateur puisse choisir un avatar, consulter son avancée dans les succès ou encore son expérience avec chaque faction.
A tout moment sur les menus, le joueur peut choisir de faire basculer l'interface graphique entre 2 thèmes : "Jedi" ou "Sith". Les images, les couleurs et l'avatar de son profil basculent alors instantanément entre ces 2 thèmes.

Pour implémenter cette interface graphique personnalisée, nous avons dérivé des classes existantes dans la bibliothèques "qt" pour y ajouter quelques options visuelles et fonctionnelles. Cette héritage permet de garder une bonne structure orienté-objet et de ne définir qu'une seule fois les modifications apportées (il suffit ensuite d'instancier l'objet de la classe réalisée là où c'est nécessaire).

Toutes les fenêtres sont contenues dans une application et, à chaque changement de fenêtre, l'ancienne est détruite et la nouvelle est instanciée.
Un diagramme de classe reprenant la structure de notre implémentation est fourni ci-après.



Voire fichier joint dans la racine du dossier git pour une image de bonne taille. (nom : guiDia.jpg)

13 Le matchmaking 2.0

Le matchmaking décrit précédemment dans ce rapport a subit quelques modifications. En effet, le fait qu'un joueur ne pouvait jouer que contre un adversaire de son niveau, c'est-à-dire ayant un ratio similaire, provoquait souvent une attente longue d'un joueur à affronter. Dans le but d'apporter une expérience de jeu plus compétitive mais également dans le soucis de faire attendre le moins possible les joueurs, le matchmaking est dès lors basé sous le système "elo". Cela signifie que n'importe quels joueurs peuvent s'affronter, indépendamment du ratio de ceux-ci. Néanmoins, si un joueur bat un joueur moins expérimenté que lui, il ne gagnera que peu d'expérience. A l'inverse, si le joueur bas un adversaire plus fort que lui, il gagnera beaucoup de points d'expériences. Par convention, un nouveau joueur commencera avec 1200 points elo.

14 Le chat graphique

Avec l'apport de la version graphique du jeu, il fallait fournir une version plus agréable du chat.

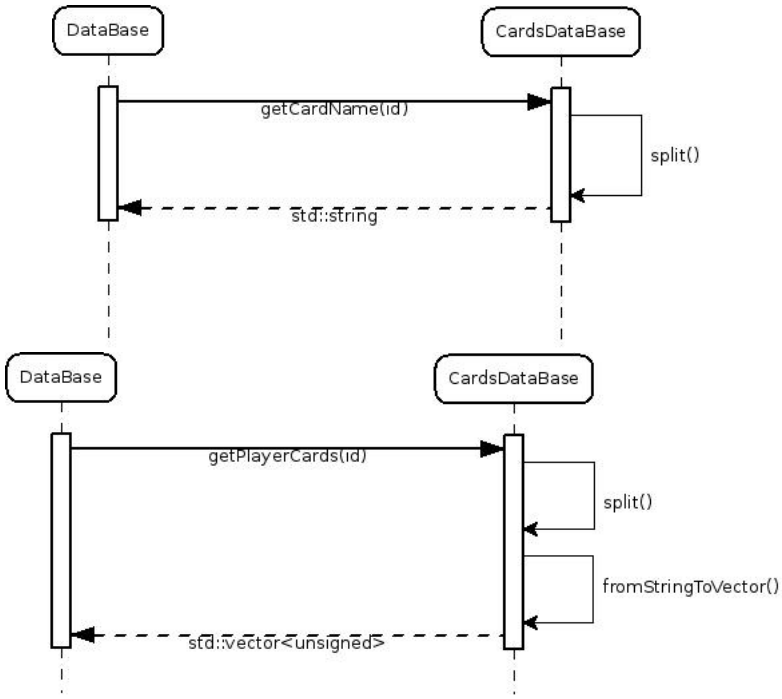
L'interface graphique possède donc trois boutons dans le coin supérieur gauche. Le premier permet de régler certaines options du jeu, notamment les sons des bruitages, le volume de la musique ainsi que le choix du thème (jedi ou sith) ayant un impact visuel sur le jeu. Le deuxième bouton quant à lui permet de gérer le chat avec ses amis.

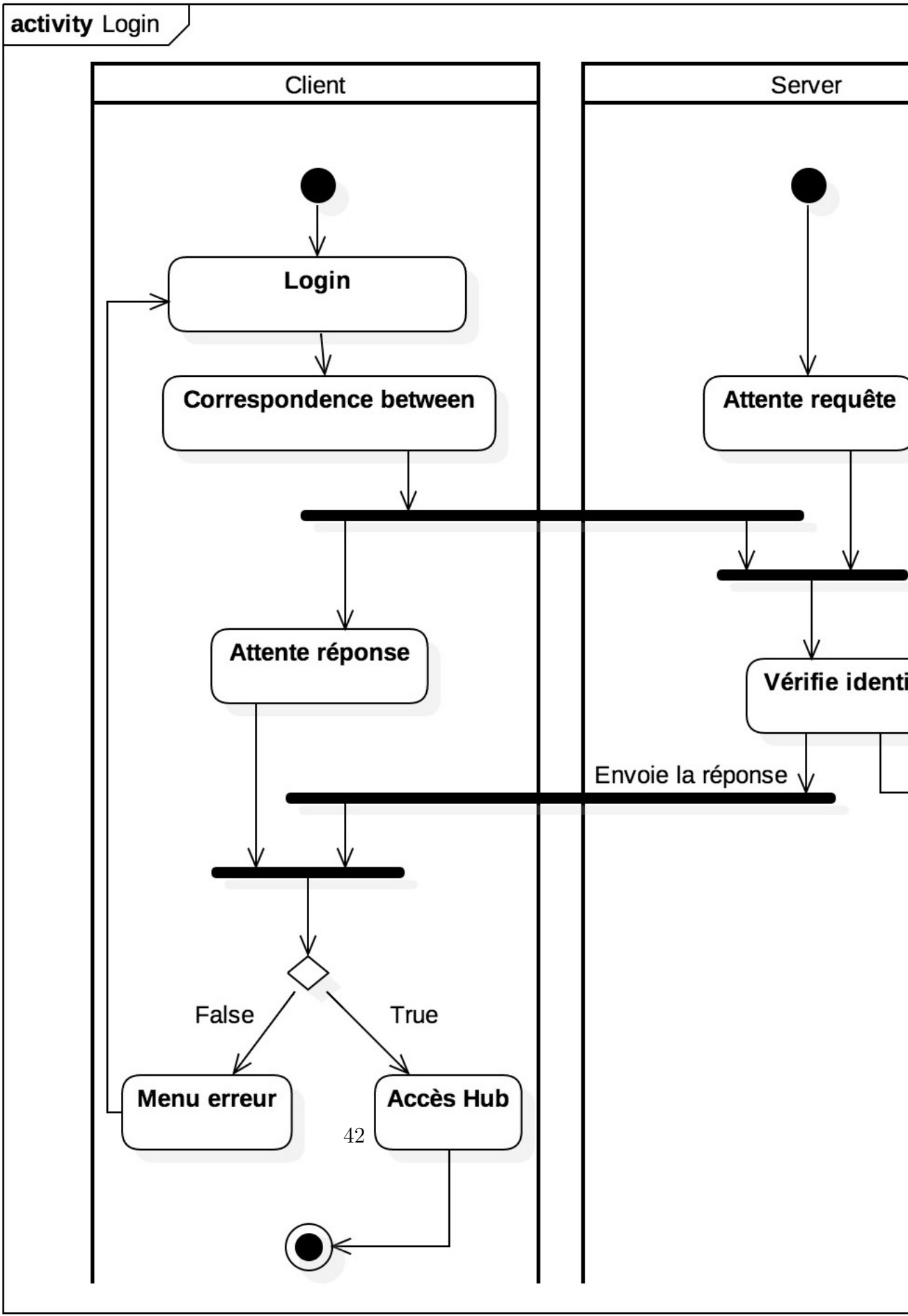
Lorsque le joueur clique sur le dit boutons, une petite fenêtre apparaît. Elle est composée d'une zone de texte, où les messages envoyés et reçus seront écrits, une zone d'entrée, où le joueur entre le message qu'il veut envoyer, une liste des amis dans laquelle le joueur peut sélectionner la personne avec qui discuter ; et enfin le boutons "send" qui permet d'envoyer le message entré dans la zone d'entrée. La liste d'ami est établie en faisant des requête en base de donnée lorsque le joueur s'est connecté.

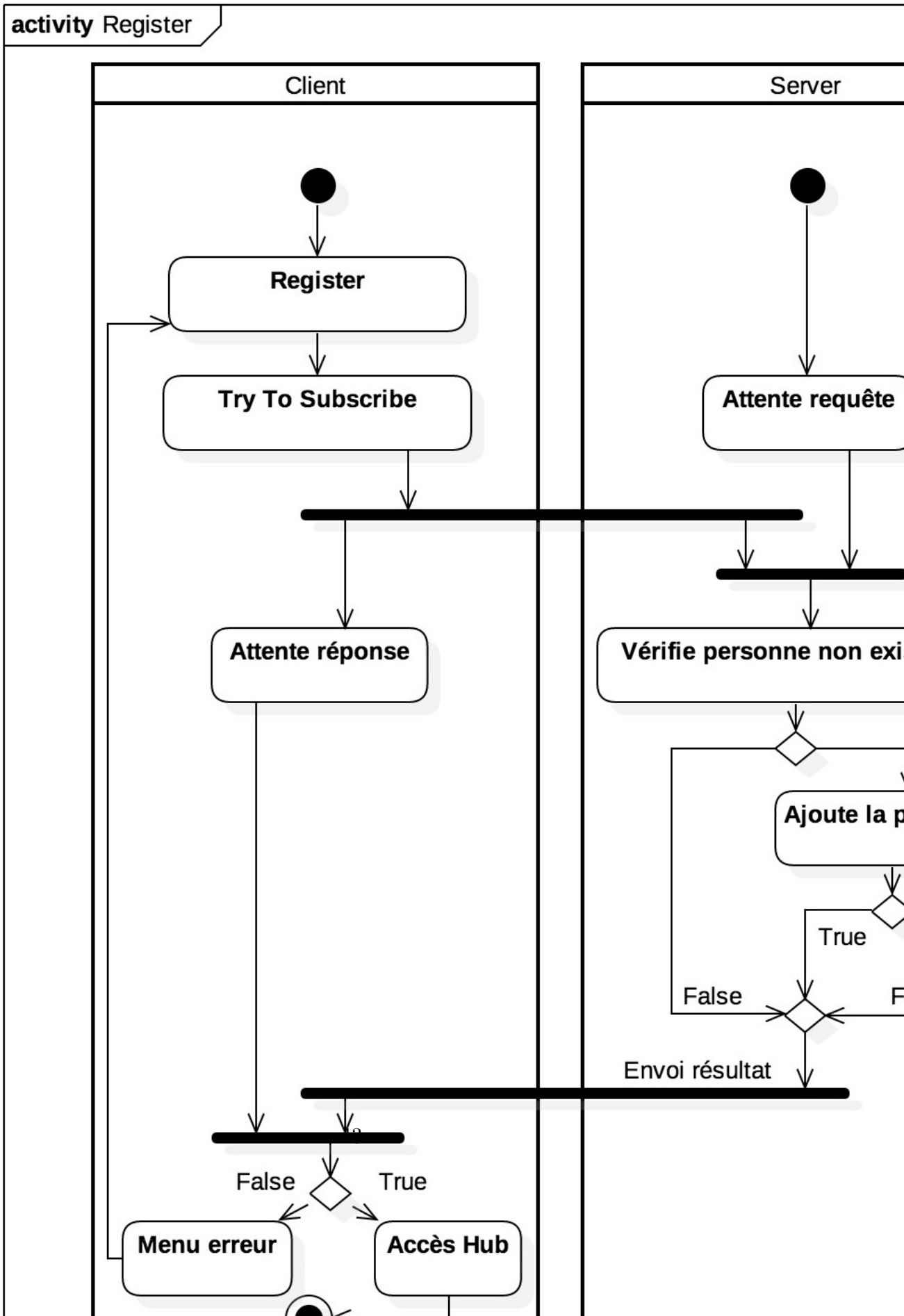
Afin de garder la compatibilité avec le chat en version terminal, la solution du peer-to-peer a été gardée. Deux cas sont donc possibles : soit le joueur est hôte de la conversation, soit il est l'invité. Le joueur sélectionnait son ami en premier dans la liste d'amis est désigné hôte. Lorsqu'il choisit son ami, il initialise un serveur dans la classe `HostChat` qui attendra que son ami s'y connecte. Pour cela, il envoie au serveur du jeu une requête pour signaler à son amis qu'il désire discuter. Un thread coté client réceptionnera cette requête transférée par le serveur du jeu et instanciera la classe `GuestChat` qui se connectera à son ami toujours en attente de connection. Cette ami recevra également une notification, son boutons de chat sera alors indiqué en rouge, et une icône de message sera affichée à coté du nom de l'ami (l'hôte) dans la liste des amis. Il peut dès lors discuter avec son ami.

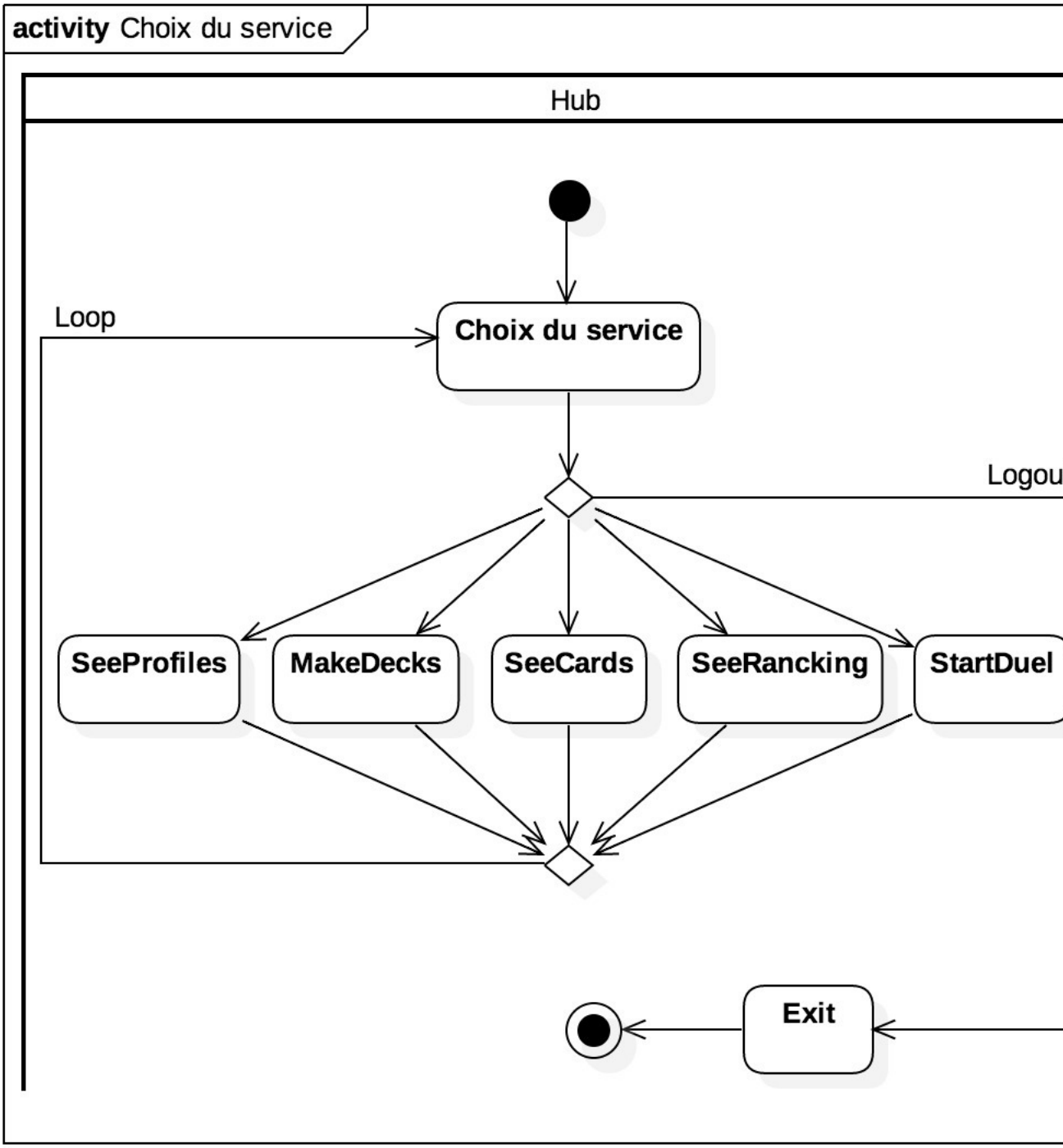
En cas de déconnexion d'un d'eux (volontaire ou non), le thread gérant ce chat sera annulé et un message à l'ami encore connecté sera envoyé pour signaler que son correspondant a quitté la conversation.

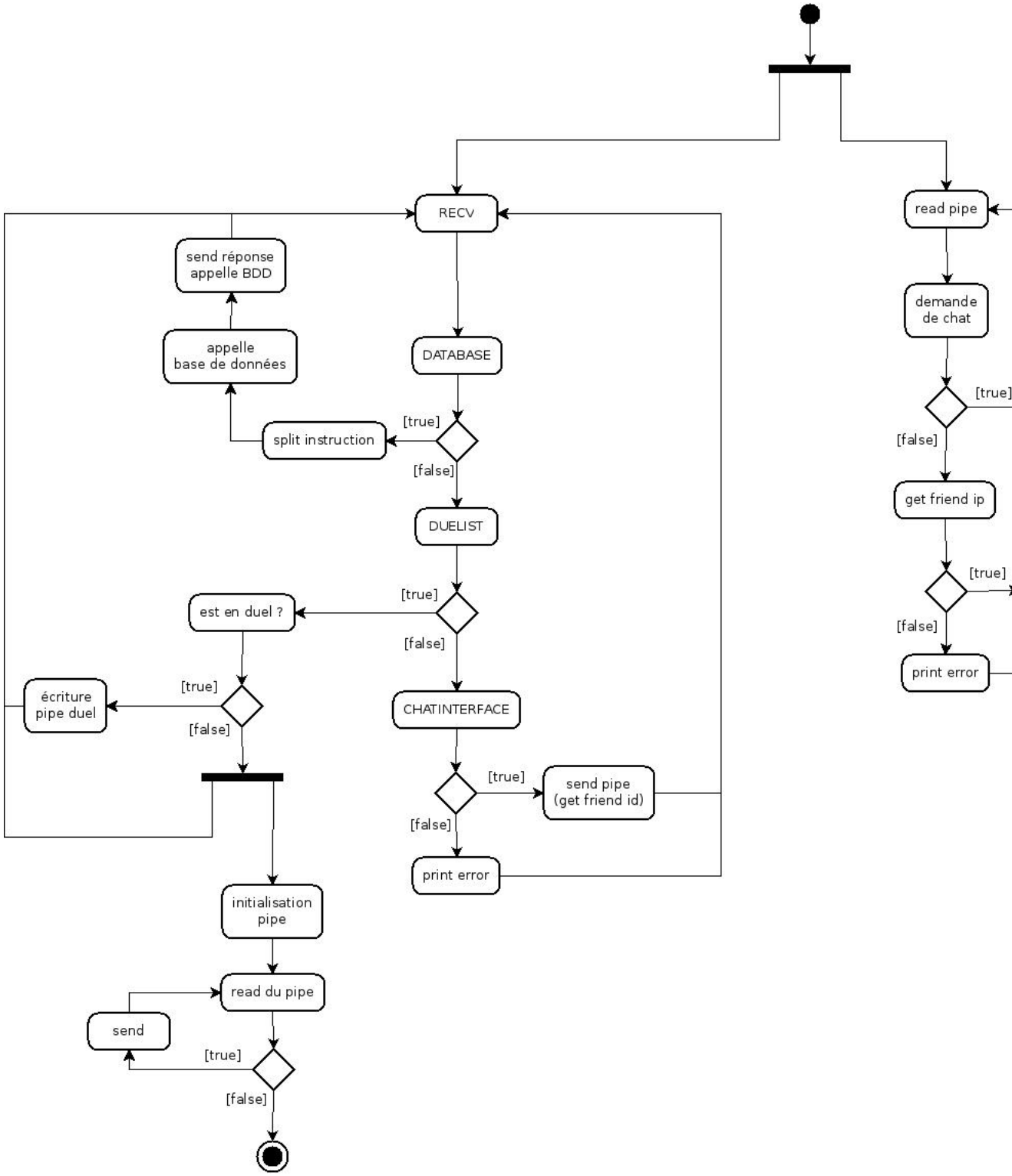
15 Annexe

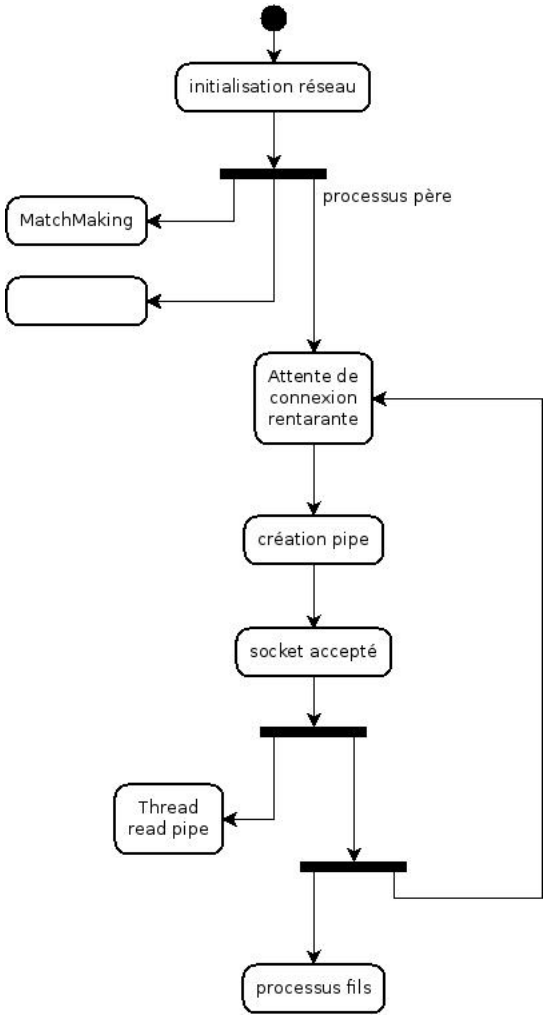

DataBase
CardsDataBase
getCardName(id)
split()
std::string
DataBase
CardsDataBase
getPlayerCards(id)
split()
fromStringToVector()
std::vector<unsigned>

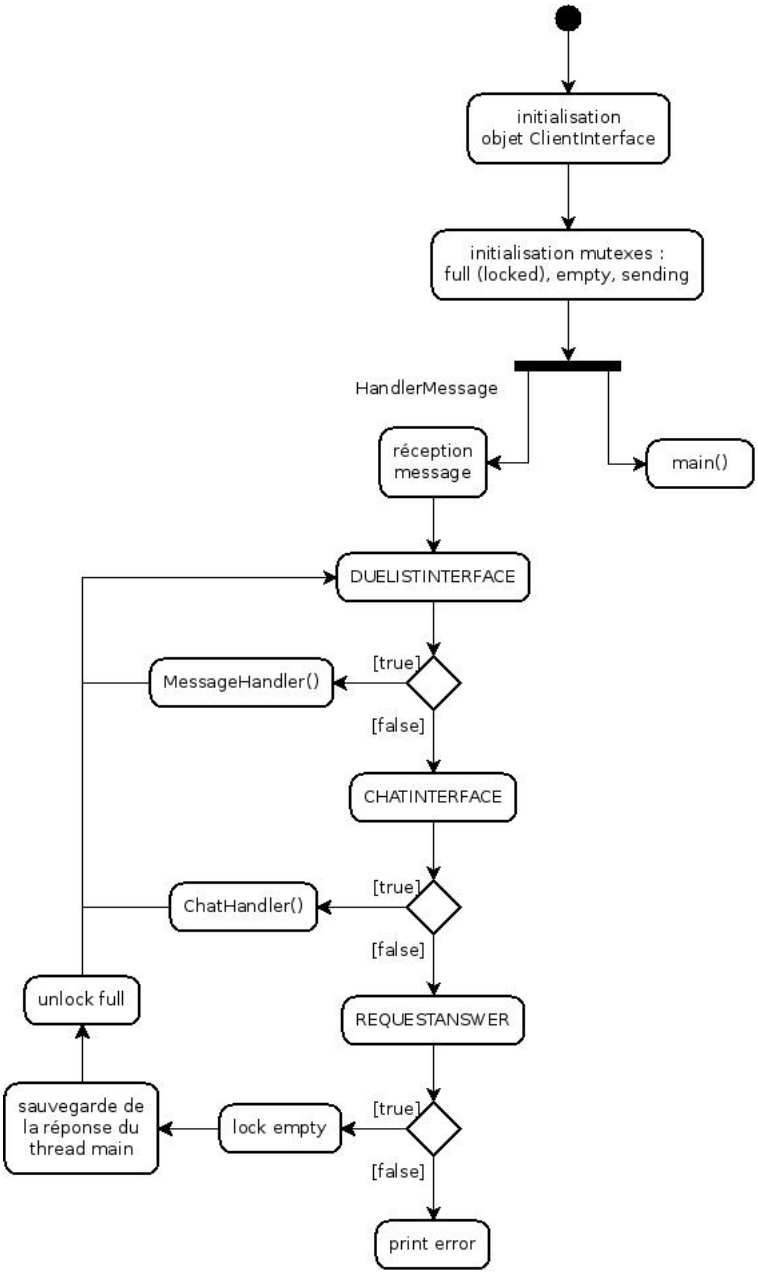

activity Login
Client
Server
Login
Correspondence between
Attente requête
Attente réponse
Vérifie identi
Envoie la réponse
False
True
Menu erreur
Accès Hub

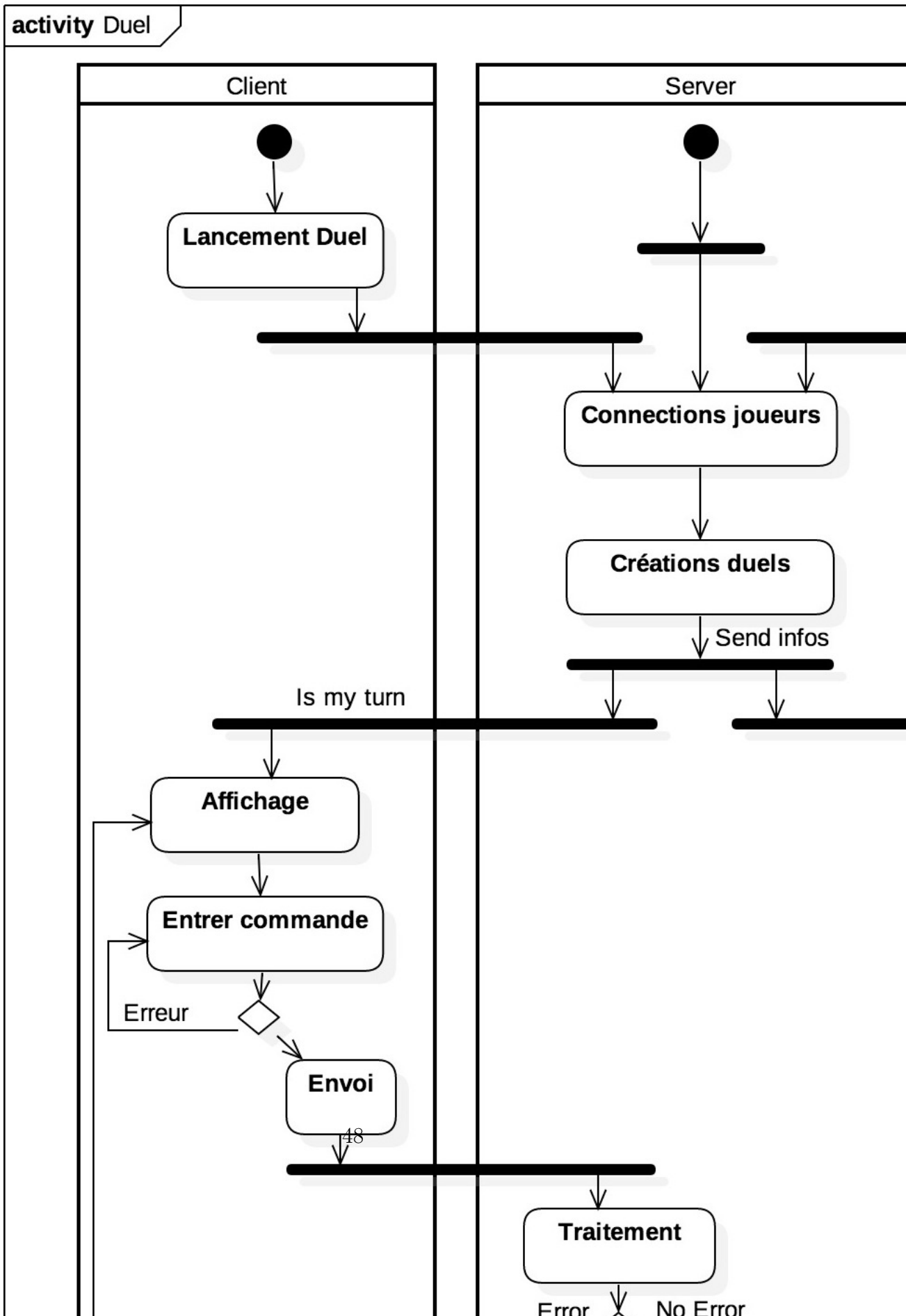

activity Register
Client
Server
Register
Try To Subscribe
Attente requête
Attente réponse
Vérifie personne non exi
Ajoute la p
True
False
Envoi résultat
False
True
Menu erreur
Accès Hub

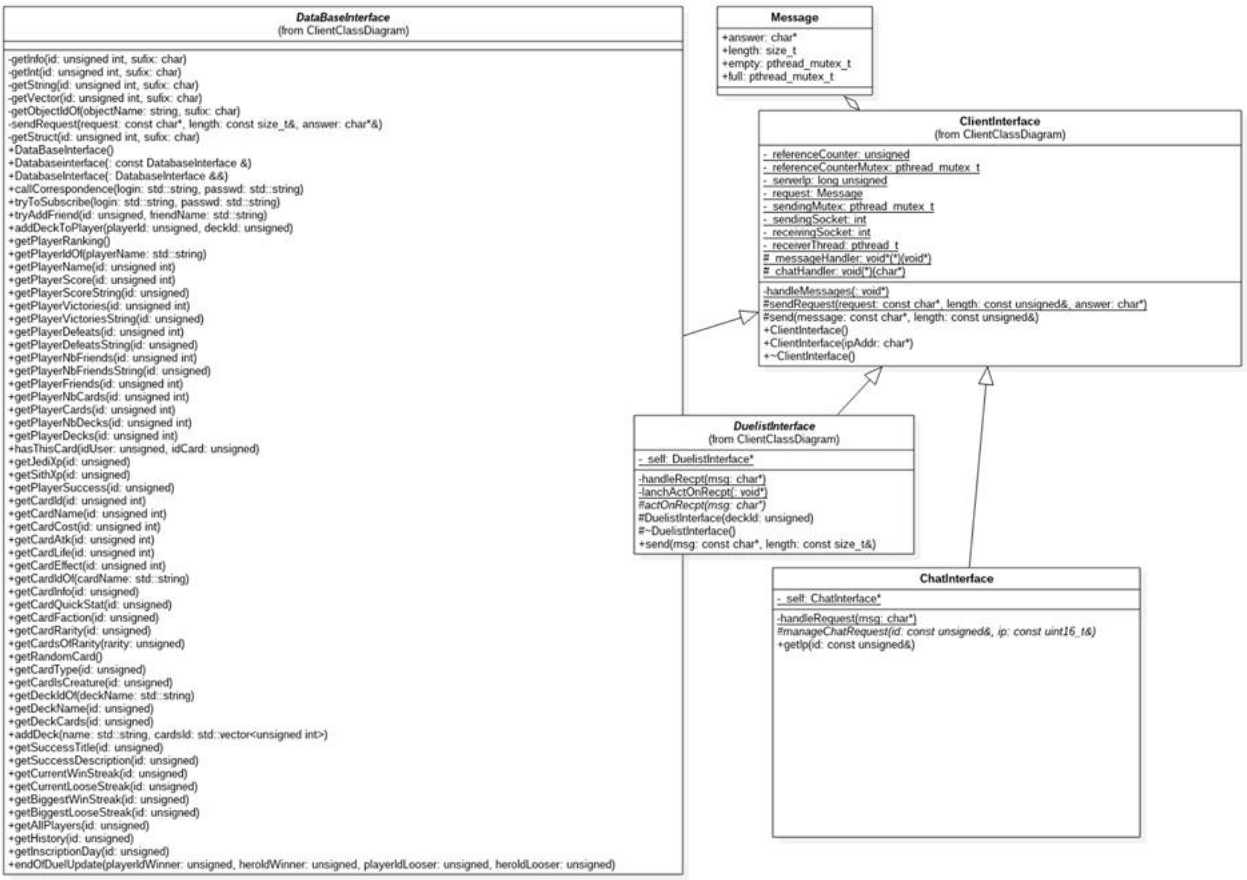

activity Choix du service
Hub
Loop
Choix du service
Logou
SeeProfiles
MakeDecks
SeeCards
SeeRancking
StartDuel
Exit

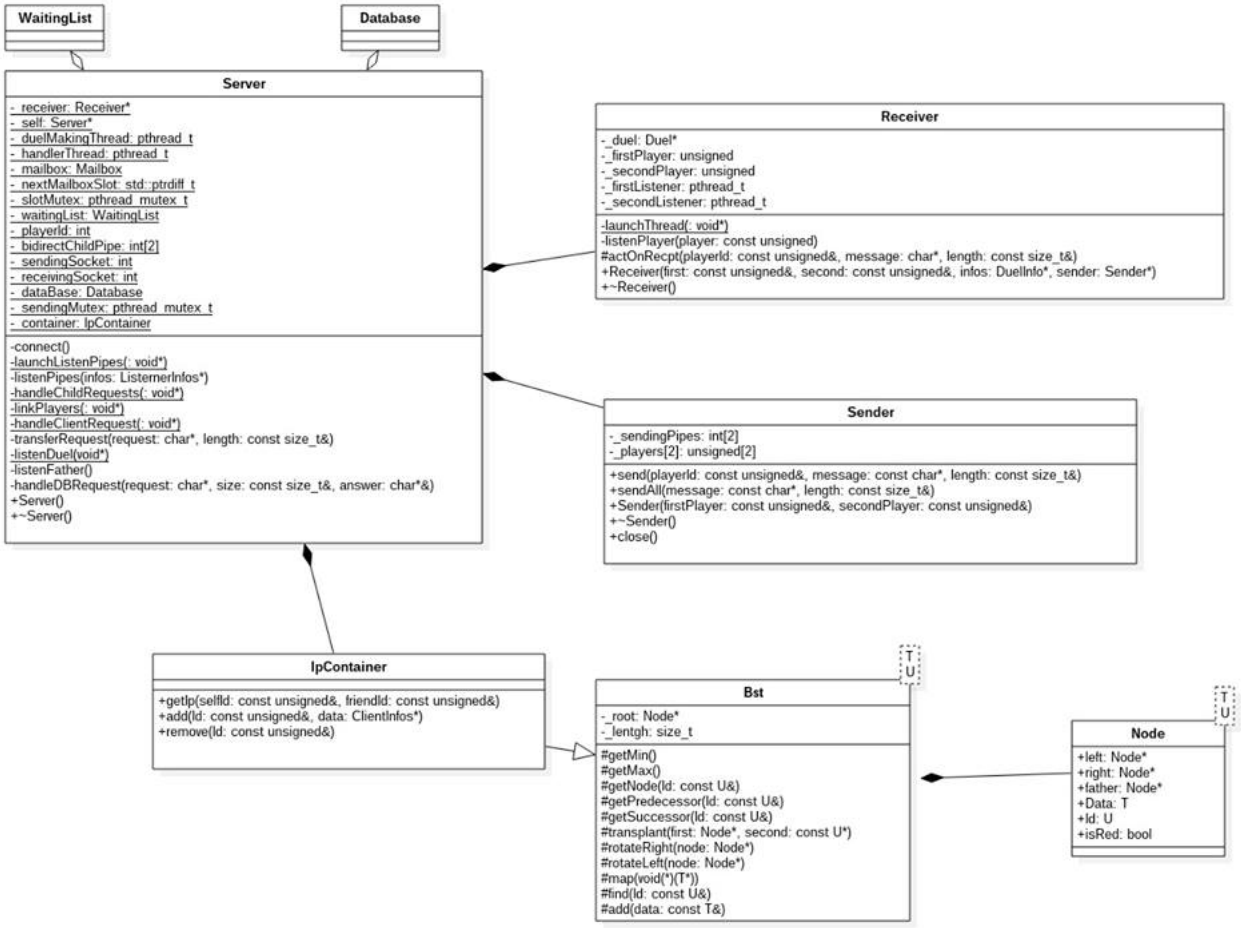

RECV
read pipe
send réponse
appelle BDD
demande
de chat
appelle
base de données
DATABASE
[true]
split instruction
[false]
[true]
[false]
DUELIST
get friend ip
[true]
est en duel ?
[true]
[false]
[false]
écriture
pipe duel
[true]
[false]
CHATINTERFACE
print error
[true]
send pipe
(get friend id)
[false]
print error
initialisation
pipe
read du pipe
send
[true]
[false]

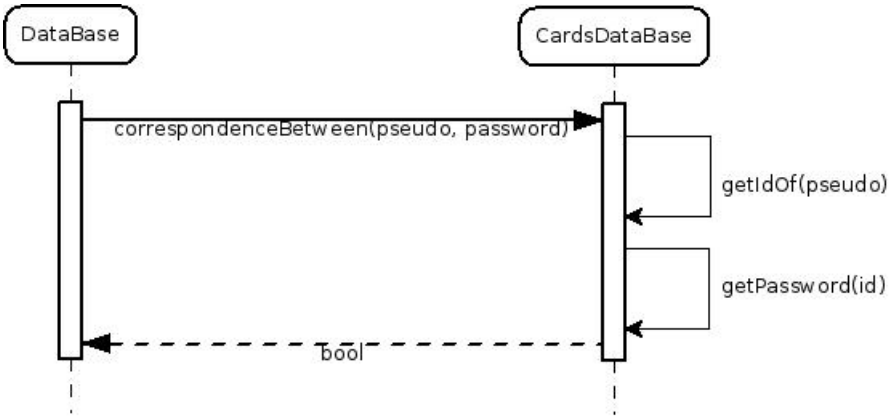

initialisation réseau
processus père
MatchMaking
Attente de connexion rentarante
création pipe
socket accepté
Thread read pipe
processus fils

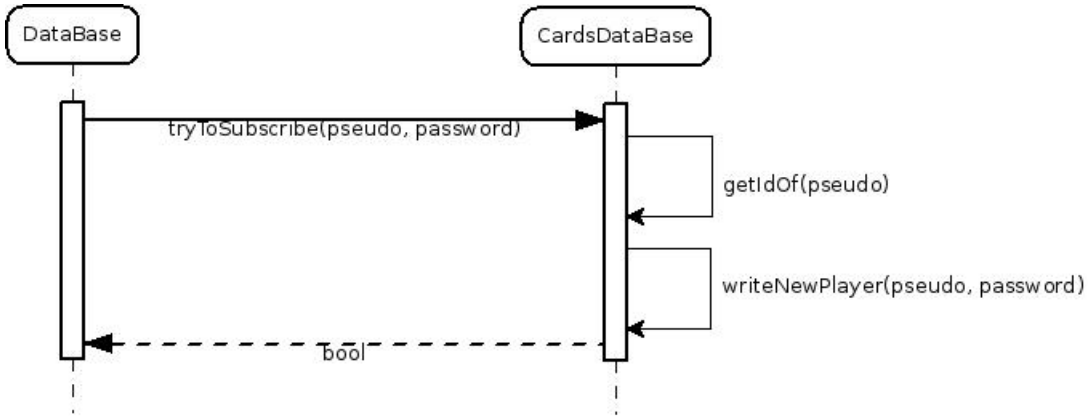

initialisation
objet ClientInterface
initialisation mutexes :
full (locked), empty, sending
HandlerMessage
réception
message
main()
DUELISTINTERFACE
[true]
MessageHandler()
[false]
CHATINTERFACE
[true]
ChatHandler()
[false]
unlock full
REQUESTANSWER
sauvegarde de
la réponse du
thread main
lock empty
[true]
[false]
print error


activity Duel
Client
Server
Lancement Duel
Connections joueurs
Créations duels
Send infos
Is my turn
Affichage
Entrer commande
Erreur
Envoi
Traitement
Error
No Error


DataBaseInterface
(from ClientClassDiagram)
-getInfo(id: unsigned int, sufix: char)
-getInt(id: unsigned int, sufix: char)
-getString(id: unsigned int, sufix: char)
-getVector(id: unsigned int, sufix: char)
-getObjectIdOf(objectName: string, sufix: char)
-sendRequest(request: const char*, length: const size_t&, answer: char*&)
-getStruct(id: unsigned int, sufix: char)
+DataBaseInterface()
+DatabaseInterface(: const DatabaseInterface &)
+DatabaseInterface(: DatabaseInterface &&)
+callCorrespondence(login: std::string, passwd: std::string)
+tryToSubscribe(login: std::string, passwd: std::string)
+tryAddFriend(id: unsigned, friendName: std::string)
+addDeckToPlayer(playerId: unsigned, deckId: unsigned)
+getPlayerRanking()
+getPlayerIdOf(playerName: std::string)
+getPlayerName(id: unsigned int)
+getPlayerScore(id: unsigned int)
+getPlayerScoreString(id: unsigned)
+getPlayerVictories(id: unsigned int)
+getPlayerVictoriesString(id: unsigned)
+getPlayerDefeats(id: unsigned int)
+getPlayerDefeatsString(id: unsigned)
+getPlayerNbFriends(id: unsigned int)
+getPlayerNbFriendsString(id: unsigned)
+getPlayerFriends(id: unsigned int)
+getPlayerNbCards(id: unsigned int)
+getPlayerCards(id: unsigned int)
+getPlayerNbDecks(id: unsigned int)
+getPlayerDecks(id: unsigned int)
+hasThisCard(idUser: unsigned, idCard: unsigned)
+getJediXp(id: unsigned)
+getSithXp(id: unsigned)
+getPlayerSuccess(id: unsigned)
+getCardId(id: unsigned int)
+getCardName(id: unsigned int)
+getCardCost(id: unsigned int)
+getCardAtk(id: unsigned int)
+getCardLife(id: unsigned int)
+getCardEffect(id: unsigned int)
+getCardIdOf(cardName: std::string)
+getCardInfo(id: unsigned)
+getCardQuickStat(id: unsigned)
+getCardFaction(id: unsigned)
+getCardRarity(id: unsigned)
+getCardsOfRarity(rarity: unsigned)
+getRandomCard()
+getCardType(id: unsigned)
+getCardIsCreature(id: unsigned)
+getDeckIdOf(deckName: std::string)
+getDeckName(id: unsigned)
+getDeckCards(id: unsigned)
+addDeck(name: std::string, cardsId: std::vector<unsigned int>)
+getSuccessTitle(id: unsigned)
+getSuccessDescription(id: unsigned)
+getCurrentWinStreak(id: unsigned)
+getCurrentLooseStreak(id: unsigned)
+getBiggestWinStreak(id: unsigned)
+getBiggestLooseStreak(id: unsigned)
+getAllPlayers(id: unsigned)
+getHistory(id: unsigned)
+getInscriptionDay(id: unsigned)
+endOfDuelUpdate(playerIdWinner: unsigned, heroIdWinner: unsigned, playerIdLooser: unsigned, heroIdLooser: unsigned)
Message
+answer: char*
+length: size_t
+empty: pthread_mutex_t
+full: pthread_mutex_t
ClientInterface
(from ClientClassDiagram)
- referenceCounter: unsigned
- referenceCounterMutex: pthread_mutex_t
- serverIp: long unsigned
- request: Message
- sendingMutex: pthread_mutex_t
- sendingSocket: int
- receivingSocket: int
- receiverThread: pthread_t
messageHandler: void*(*)(void*)
chatHandler: void(*)(char*)
-handleMessages(: void*)
#sendRequest(request: const char*, length: const unsigned&, answer: char*)
#send(message: const char*, length: const unsigned&)
+ClientInterface()
+ClientInterface(ipAddr: char*)
+~ClientInterface()
DuelistInterface
(from ClientClassDiagram)
- self: DuelistInterface*
-handleRecpt(msg: char*)
-lanchActOnRecpt(: void*)
#actOnRecpt(msg: char*)
#DuelistInterface(deckId: unsigned)
#~DuelistInterface()
+send(msg: const char*, length: const size_t&)
ChatInterface
- self: ChatInterface*
-handleRequest(msg: char*)
#manageChatRequest(id: const unsigned&, ip: const uint16_t&)
+getIp(id: const unsigned&)


WaitingList
Database
Server
- receiver: Receiver*
- self: Server*
- duelMakingThread: pthread_t
- handlerThread: pthread_t
- mailbox: Mailbox
- nextMailboxSlot: std::ptrdiff_t
- slotMutex: pthread_mutex_t
- waitingList: WaitingList
- playerId: int
- bidirectChildPipe: int[2]
- sendingSocket: int
- receivingSocket: int
- dataBase: Database
- sendingMutex: pthread_mutex_t
- container: IpContainer
-connect()
-launchListenPipes(: void*)
-listenPipes(infos: ListernerInfos*)
-handleChildRequests(: void*)
-linkPlayers(: void*)
-handleClientRequest(: void*)
-transferRequest(request: char*, length: const size_t&)
-listenDuel(void*)
-listenFather()
-handleDBRequest(request: char*, size: const size_t&, answer: char*&)
+Server()
+~Server()
Receiver
-_duel: Duel*
-_firstPlayer: unsigned
-_secondPlayer: unsigned
-_firstListener: pthread_t
-_secondListener: pthread_t
-launchThread(: void*)
-listenPlayer(player: const unsigned)
#actOnRecpt(playerId: const unsigned&, message: char*, length: const size_t&)
+Receiver(first: const unsigned&, second: const unsigned&, infos: DuelInfo*, sender: Sender*)
+~Receiver()
Sender
-_sendingPipes: int[2]
-_players[2]: unsigned[2]
+send(playerId: const unsigned&, message: const char*, length: const size_t&)
+sendAll(message: const char*, length: const size_t&)
+Sender(firstPlayer: const unsigned&, secondPlayer: const unsigned&)
+~Sender()
+close()
IpContainer
+getIp(selfId: const unsigned&, friendId: const unsigned&)
+add(Id: const unsigned&, data: ClientInfos*)
+remove(Id: const unsigned&)
T
U
Bst
-_root: Node*
-_lentgh: size_t
#getMin()
#getMax()
#getNode(Id: const U&)
#getPredecessor(Id: const U&)
#getSuccessor(Id: const U&)
#transplant(first: Node*, second: const U*)
#rotateRight(node: Node*)
#rotateLeft(node: Node*)
#map(void(*)(T*))
#find(Id: const U&)
#add(data: const T&)
T
U
Node
+left: Node*
+right: Node*
+father: Node*
+Data: T
+Id: U
+isRed: bool


DataBase
CardsDataBase
correspondenceBetween(pseudo, password)
getIdOf(pseudo)
getPassword(id)
bool


DataBase
CardsDataBase
tryToSubscribe(pseudo, password)
getIdOf(pseudo)
writeNewPlayer(pseudo, password)
bool

Index des termes utilisés

A Description des diagrammes use case

Use Case	Pré-condition	Post-condition	Cas général	Cas exceptionnel
Créer compte	Avoir un accès réseau	- L'utilisateur a un compte - L'utilisateur reçoit les cartes de base	L'utilisateur entre une adresse mail, un mot de passe et un pseudonyme	- E-mail non valide - Pseudonyme déjà pris
Se connecter	Avoir un compte	L'utilisateur est connecté	L'utilisateur entre son adresse mail et son mot de passe	- Compte inexistant ou erroné
Se déconnecter	Etre connecté	L'utilisateur est déconnecté	L'utilisateur lance la déconnexion	- Arrêt brutal
Consulter classement	/	Le classement s'affiche	L'utilisateur clique sur le bouton et consulte classement	/
Consulter liste amis	Avoir des amis	La liste d'amis s'affiche	L'utilisateur clique sur le bouton et voit ses amis s'afficher	/
Chat	Avoir des amis	La fenêtre de dialogue s'affiche	L'utilisateur affiche sa liste d'amis et choisit un ami avec lequel discuter	L'ami choisi n'est pas connecté
Voir profil	/	Le profil s'affiche	L'utilisateur clique sur son profil ou choisit le profil d'un ami dans sa liste d'amis	/
Consulter cartes	/	La liste de toutes les cartes du jeu s'affiche	L'utilisateur clique sur le bouton « consulter carte »	/
Création deck	/	L'interface de création de deck s'affiche: - L'utilisateur voit ses cartes et ses docks déjà créés	L'utilisateur clique sur « créer deck » et crée son deck	- Deck non valide
Jouer	/	- L'utilisateur est dans le lobby	L'utilisateur clique sur jouer	/

FIGURE 11 – Description du diagramme use case hors duel

Use case	Pré-condition	Post-condition	Cas général	Cas exceptionnel
Lancer duel	/	Le duel est lancé	- Choisir mode de jeu - Choisir deck	En attente d'un adversaire
Choisir deck	Avoir un deck valide	Le deck est sélectionné	L'utilisateur choisit son deck	/
Choisir mode de jeu	/	Mode de jeu sélectionné	L'utilisateur choisit le mode de jeu	/

FIGURE 12 – Description du diagramme use case inter duel

Use case	Pré-condition	Post-condition	Cas général	Cas exceptionnel
Afficher carte en main	Avoir des cartes en main	La carte s'affiche en grand	L'utilisateur survole une carte de sa main avec la souris	/
Fin de tour	Etre le joueur actif	Le tour de l'adversaire commence	L'utilisateur clique sur « fin de tour »	/
Jouer carte créature	- Avoir une carte créature en main et avoir énergie requise pour la carte - Les conditions de la carte sont respectées	- La créature est déployée - Retrait d'énergie - La carte est retirée de la main	L'utilisateur clique sur la carte puis sur l'emplacement du terrain où la déployer	- Pas de cible valide - Terrain plein
Jouer carte sort	- Avoir une carte sort en main et avoir l'énergie requise pour la carte - Les conditions de la carte sont respectées	- Le sort est activé puis défaussé - Retrait d'énergie	L'utilisateur clique sur la carte sort puis sur la ou les cibles	Pas de cible valide
Attaque créature	Avoir un personnage capable d'attaquer	Application des dégâts	L'utilisateur clique sur l'attaquant puis sur l'attaqué	Cible non valide

FIGURE 13 – Description du diagramme use case en duel