Université libre de Bruxelles

INFO-F-209

Projet d'année 2

Software Requirement Document : Wizard Poker Groupe 5

Auteurs :
Baudoux Nicolas, Berrewaerts Jonathan, Gueniffrey Stanislas, Muranovic Allan, Petit Robin, Reynouard Alexis et Verhelst Théo

26 février 2016

Table des matières

1 Introduction **1**
1.1 But du projet . . . 1
1.1.1 Description du Wizard Poker . . . 1
1.2 Glossaire . . . 2
1.3 Historique . . . 2

2 Besoins de l'utilisateur **3**
2.1 Exigences fonctionnelles . . . 3
2.1.1 Possibilités d'actions de l'utilisateur . . . 3
2.1.2 Création d'un compte . . . 4
2.1.3 Gestion des . . . 5
2.1.4 Début de partie . . . 6
2.1.5 Détail d'un tour . . . 7
2.2 Exigences non-fonctionnelles . . . 7
2.3 Exigences de domaine . . . 7

3 Besoins du système **8**
3.1 Exigences fonctionnelles . . . 8
3.2 Exigences non-fonctionnelles . . . 8
3.3 Design et fonctionnement du système . . . 8
3.4 Diagramme de classe du et des cartes . . . 9
3.5 Diagramme de classe du serveur et de sa base de données . . . 10
3.6 Diagramme des classes intervenant dans une partie (simplifié) . . . 11

4 Index **12**

1 Introduction

1.1 But du projet

L'objectif visé par ce projet d'année de BA2 en sciences informatiques est la réalisation d'une application client-serveur en C/C++. L'application visée est un jeu de carte (appelé *Wizard Poker*) au tour par tour et multijoueurs en réseau. Elle est destinée à tous types de public. Elle est également *open source*, libre de droit et à but non commercial : c'est un projet académique.

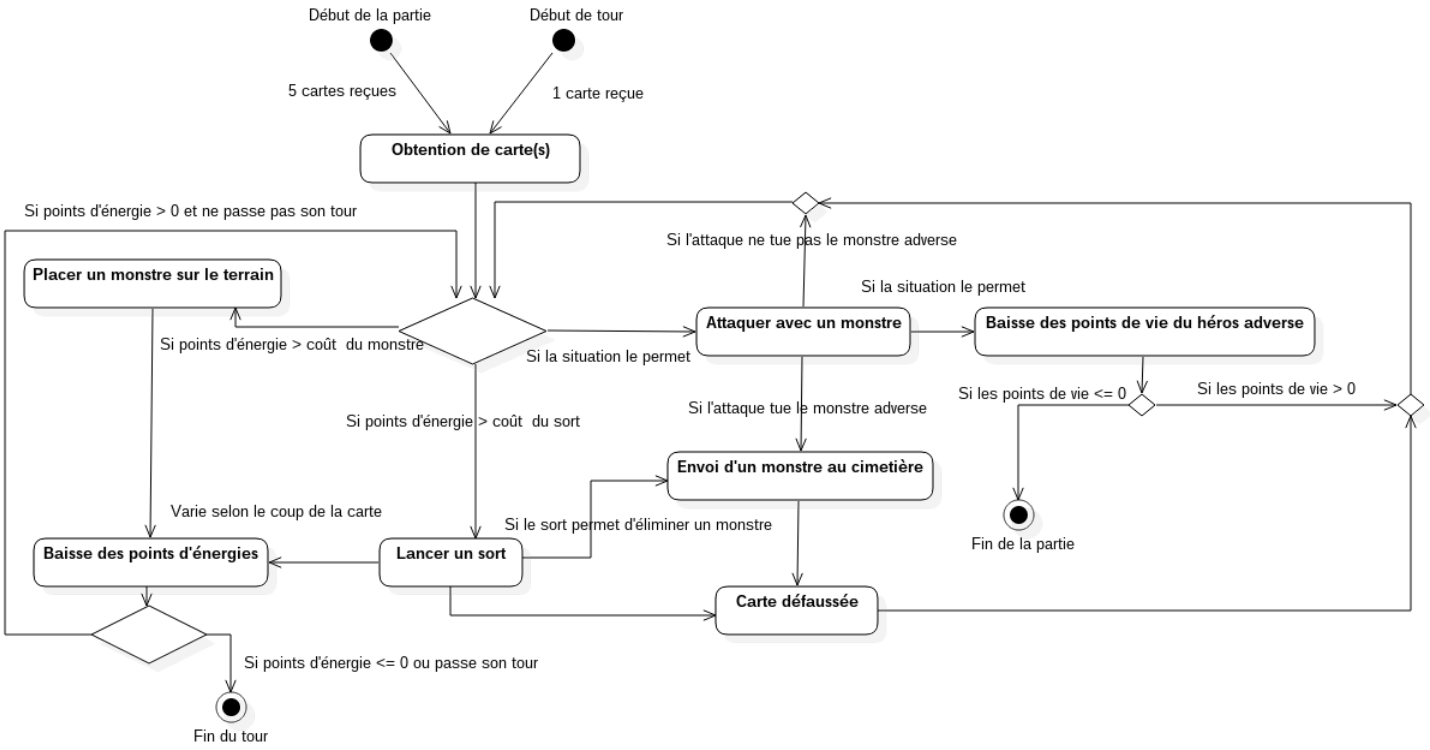
Pour ce faire, notre équipe, composée de sept personnes, dispose de trois phases de développement qui dureront approximativement 4 semaines chacune :

- la première portera uniquement sur la création du squelette respectant toutes les demandes effectuées par les client ;
- la deuxième phase concernera l'implémentation en console uniquement ;
- et enfin la troisième ajoutera l'interface graphique.

1.1.1 Description du Wizard Poker

Le Wizard Poker est un jeu de cartes dans lequel deux joueurs s'affrontent en au tour par tour. Chaque joueur a son propre choisi parmi sa collection de cartes. Au début de chaque duel, les deux joueurs ont le même nombre de points de vie (à savoir 20) et piochent 5 cartes de leur .Le premier joueur sera choisi aléatoirement par le serveur. Ensuite, tout à tour, les joueurs pourront utiliser leurs cartes, qu'elles soient un Sort ou une Creature tant que leur coût d'utilisation/d'invocation/, ne dépasse pas l'énergie disponible du joueur. (Le joueur gagne un point d'énergie supplémentaire par tour jusqu'à atteindre son maximum de 10.) Les cartes peuvent attaquer soit l'adversaire directmeent, soit une , soit une Creature adverse. Ou encore avoir un ou plusieurs effets *spéciaux*[1]. Lorsqu'une carte arrive à la fin de sa période d'existence (points de vie de la carte arrivés à 0, sort détruisant ladite carte, effet terminé, etc.), elle est défaussée, c'est-à-dire qu'elle est envoyée dans la . Lorsqu'une carte est dans la (également appelé cimetière), elle peut éventuellement être réutilisée à l'aide de sorts particuliers par exemple. Une carte défaussée n'est pas perdue, elle reste propriété du joueur : à la fin de la partie, les joueurs ont les mêmes cartes qu'au début de la partie si ce n'est que le gagnant a reçu une carte supplémentaire suite à sa victoire. La partie s'arrête lorsqu'un des deux joueurs a un nombre inférieur ou égal à 0 points de vie ou a passé 10 tours sans carte en main.

1. Par exemple : une carte au hasard, redonner de la vie, augmenter l'attaque, etc.



1.2 Glossaire

1.3 Historique

11/12/2015 Réunion (équipe complète) ;
11/12/2015 Squelette du SRD (équipe SRD) ;
15/12/2015 Création des diagrammes UML (équipe UML) ;
15/12/2015 Rédaction de la version propre du SRD (Robin).
31/01/2016 -> 26/02/2016 Développement du projet (en console uniquement).
25/02/2016 Mise à jour du SRD (Allan).

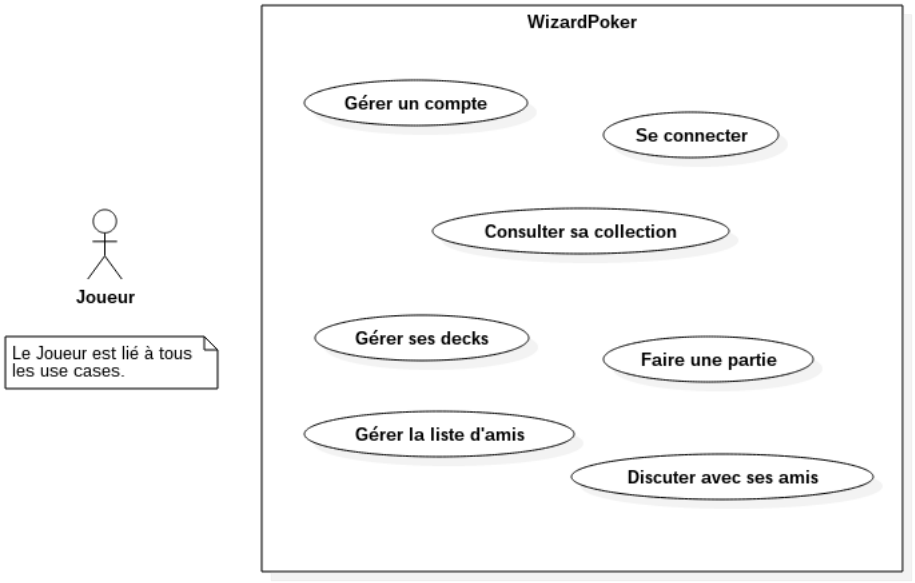
2 Besoins de l'utilisateur

L'utilisateur de l'application, à savoir le joueur, a un certain nombre de besoins. Ces derniers doivent être satisfaits afin de garantir un confort d'utilisation maximal. En utilisant l'application, le joueur **doit** avoir la possibilité de :

— créer un compte utilisateur ;
— consulter, modifier ou supprimer ses informations personnelles ;
— créer un ou en modifier un pré-existant ;
— consulter les cartes dont il dispose ;
— consulter les dont il dispose ;
— ajouter un joueur existant en ami ;
— discuter avec un ami ;
— **:** discuter avec plusieurs amis simultanément ;
— affronter un adversaire aléatoire ;
— défier un joueur de sa liste d'amis ;
— consulter le classement des joueurs.

2.1 Exigences fonctionnelles

2.1.1 Possibilités d'actions de l'utilisateur



— **Créer un compte**
Description Un utilisateur doit créer un compte pour pouvoir accéder à l'application.
Pré-condition Aucune.
Post-condition Un compte est crée, le nom d'utilisateur correspondant est unique.

— **Se connecter**
Description Un utilisateur doit se connecter avec un compte pour pouvoir acceder à l'application
Pré-condition L'utilisateur n'est pas déjà connecté, le compte utilisé est existant est n'est pas déjà connecté.
Post-condition L'utilisateur est connecté.

— **Consulter sa collection**
Description Un utilisateur peut voir la collection de cartes qu'il a acquises.

Pré-condition L'utilisateur doit être connecté au serveur.

Post-condition L'utilisateur a pris connaissance sa collection de carte.

— **Faire une partie**

Description Un utilisateur peut jouer une partie.

Pré-condition L'utilisateur a un deck actif.

Post-condition Si l'utilisateur a gagné la partie, une carte est ajoutée à sa collection. Ses statistiques (parties gagnées / parties perdues) sont mises à jour.

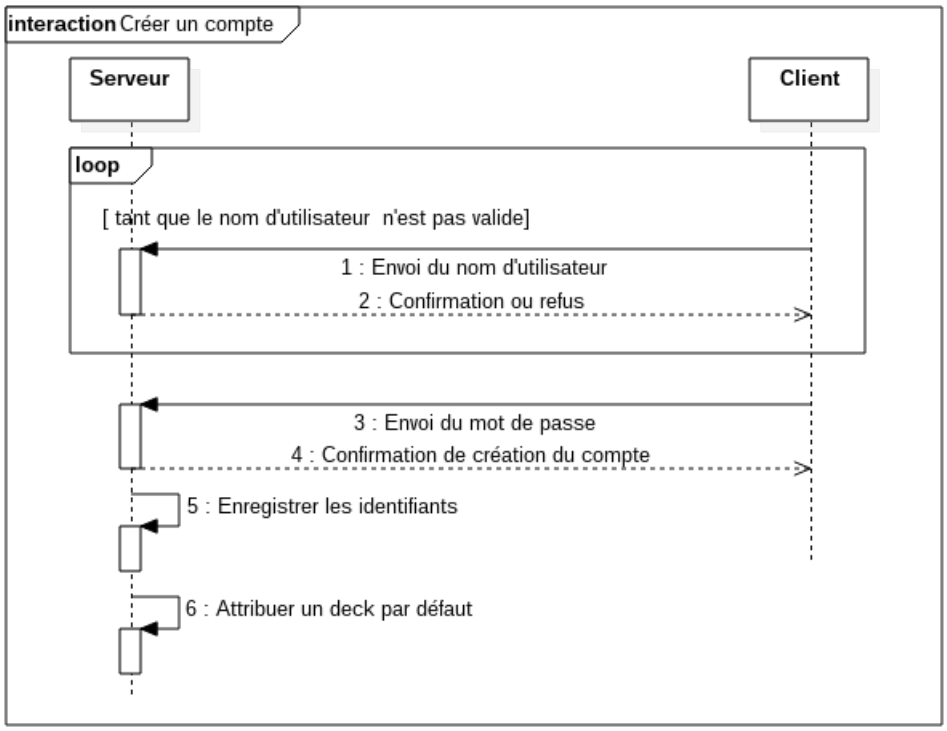
— **Discuter avec ses amis**

Description Un utilisateur peut entamer une conversation avec d'autres joueurs, s'ils sont dans sa liste d'amis.

Pré-condition L'utilisateur a au moins un ami, et l'ami

Post-condition L'utilisateur est en conversation avec son ami.

2.1.2 Création d'un compte

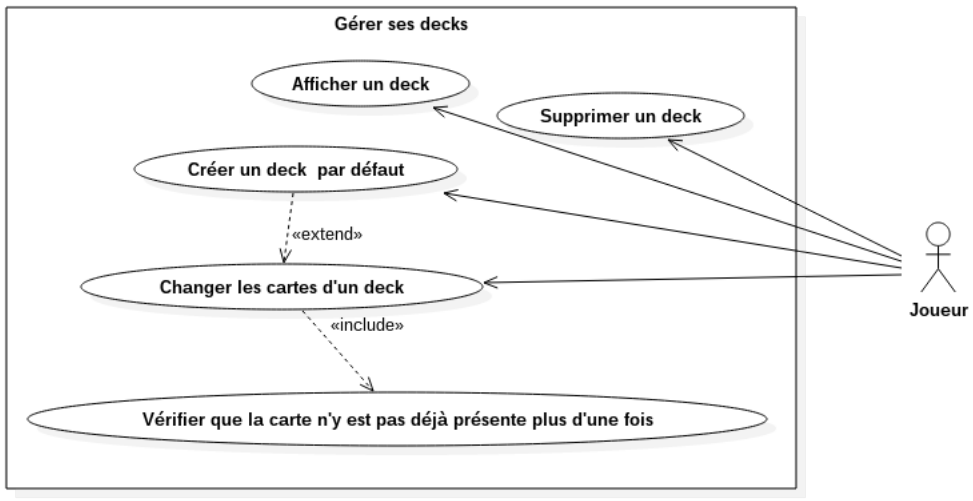


Description L'utilisateur peut créer un compte à l'aide d'un identifiant unique (son pseudo) et d'un mot de passe associé.

Pré-condition L'utilisateur doit être connecté au serveur.

Post-condition Soit la demande est acceptée (identifiant admissible), soit la demande est rejetée (identifiant non-admissible ou déjà utilisé).

2.1.3 Gestion des



— **Supprimer un deck**
Description Suppression d'un deck.
Pré-condition La liste des decks n'est pas vide.
Post-condition Il y a un deck en moins dans la liste des decks.

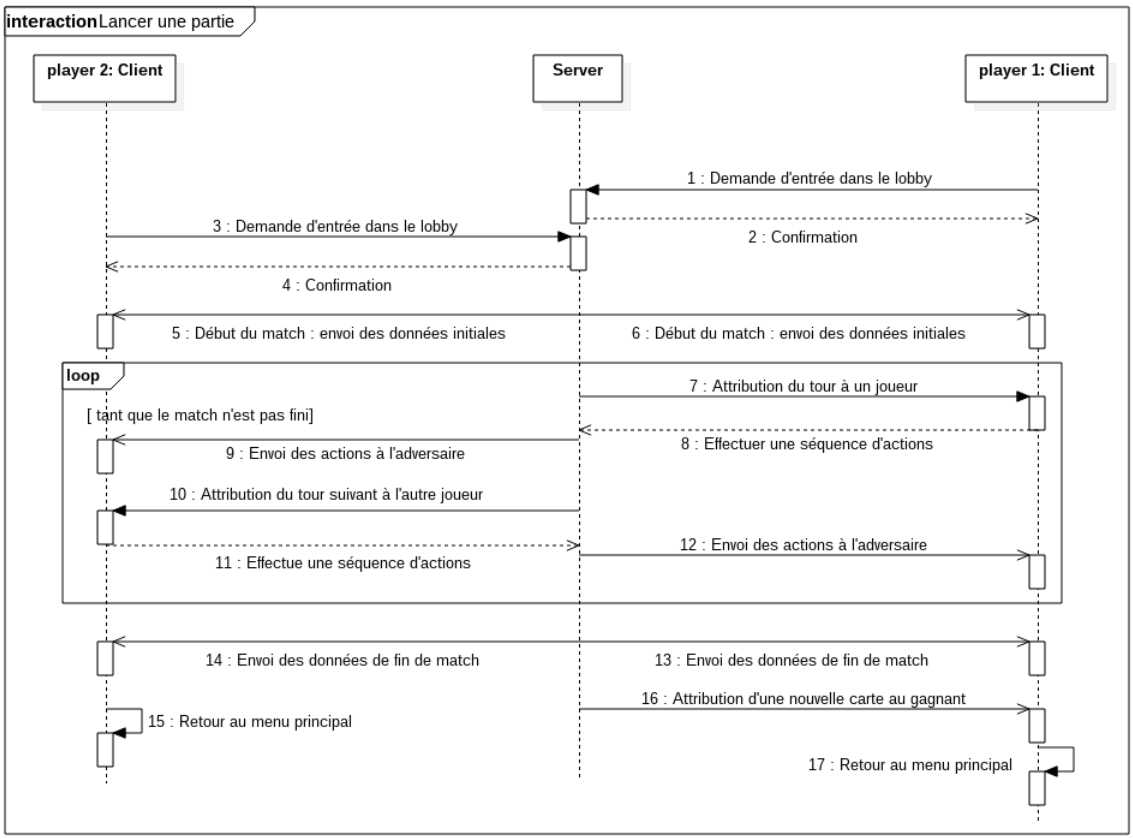
— **Afficher un deck**
Description Affichage d'un deck.
Pré-condition La liste des decks n'est pas vide.
Post-condition L'utilisateur a pris connaissance du contenu d'un deck.

— **Créer un deck par défaut**
Description Pour créer un nouveau deck, le deck par défaut est ajouté.
Pré-condition Aucune.
Post-condition Il y a un deck en plus dans la liste des decks.

— **Changer les cartes d'un deck**
Description Pour modifier un deck, il faut changer une par une les cartes qui le composent.
Pré-condition La liste des decks n'est pas vide.
Post-condition Un des decks la liste des decks a été modifié.

Le fonctionnement de la création d'un nouveau deck nécessite peut-être quelques explications supplémentaire. Le use case "Créer un deck" commence par ajouter dans la liste des decks une copie du deck par défaut (celui donné à la création du compte). Ensuite l'utilisateur est redirigé vers le use case "Changer les cartes d'un deck", qui permet de remplacer une par une les cartes d'un deck. Ainsi, avec une grande modularité et peu de code, il est donné à l'utilisateur la possibilité de créer un tout nouveau deck sans devoir recoder une partie de l'interface qui modifie un deck.

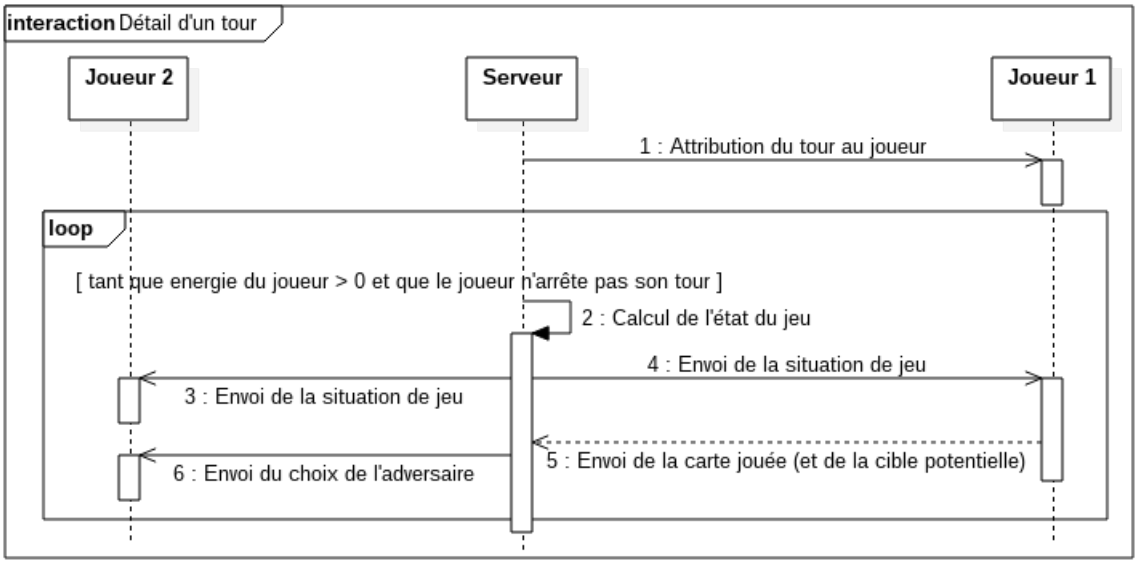
2.1.4 Début de partie



Description Un joueur connecté peut défier un ami ou demander un duel contre un adversaire aléatoire.
Pré-condition Chaque utilisateur doit être identifié et connecté à son compte.
Post-condition Un joueur aura perdu, l'autre aura gagné [2].

2. Le gagnant reçoit une carte aléatoire à ajouter à son .

2.1.5 Détail d'un tour



2.2 Exigences non-fonctionnelles

L'application **doit** tourner sur Linux Debian (salles 008 et 009 du NO.4) et fonctionner en réseau (*a priori* avec clients et serveurs sur des machines séparées).

De plus, l'utilisateur **doit** pouvoir discuter par messages à n'importe quel moment de l'exécution de l'application : tant pendant qu'il gère ses amis, ses cartes ou pendant qu'il est en . Ces discutions ne peuvent se faire avec un joueur n'étant pas ami de l'utilisateur.

2.3 Exigences de domaine

Les exigences implicites au domaine du jeu sont les suivante et sont **facultatives** :

— l'application doit être accessible à tout utilisateur potentiel ;
— l'application doit être *amusante* donc équilibrée ;
— empêcher la triche dans la mesure du possible.

3 Besoins du système

3.1 Exigences fonctionnelles

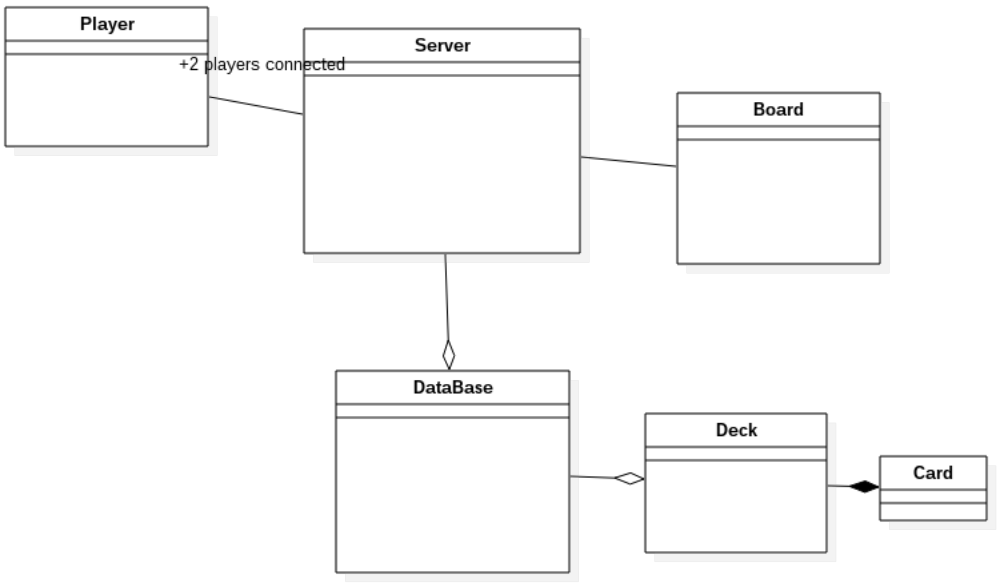
Dans un premier temps, l'application **doit** tourner en ligne de commande et **doit**, dans un second temps adopter une interface graphique. De plus, l'application **doit** être développée en C++.

3.2 Exigences non-fonctionnelles

Le système **doit** être maintenable et pensé dans l'optique d'une future adaptation avec interface graphique.

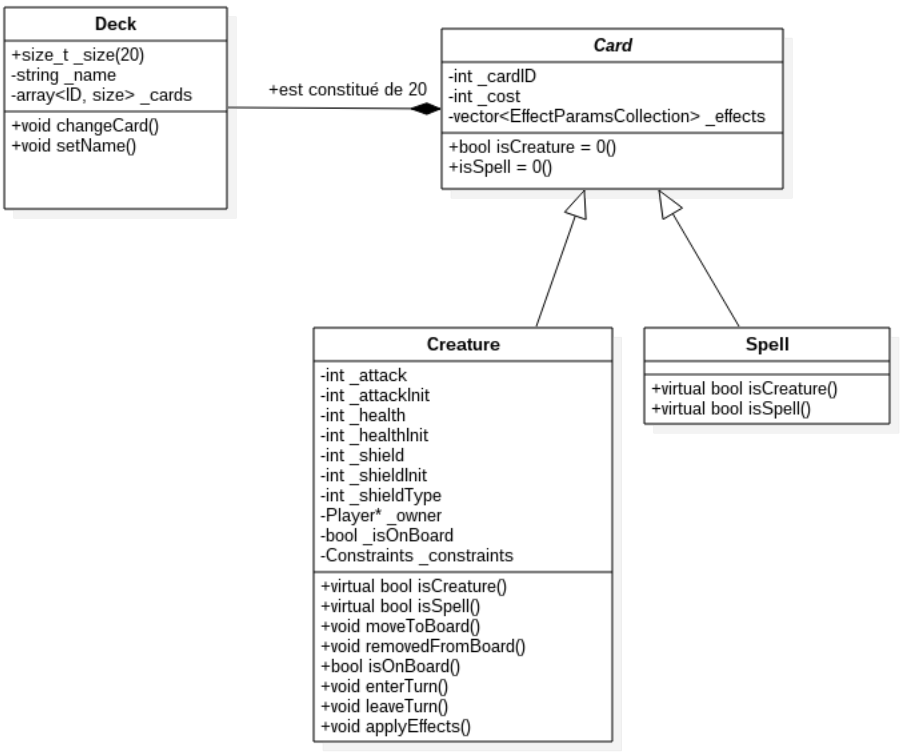
3.3 Design et fonctionnement du système

La structure du programme est résumée dans le diagramme de classe ci-dessous. Chaque élément sera développé par après.



Nous pouvons constater que le client (Player) se connecte au serveur qui, lui, gère les parties (le tableau de jeu) ainsi que la base de donnée et tout ce dont cette dernière comprend.

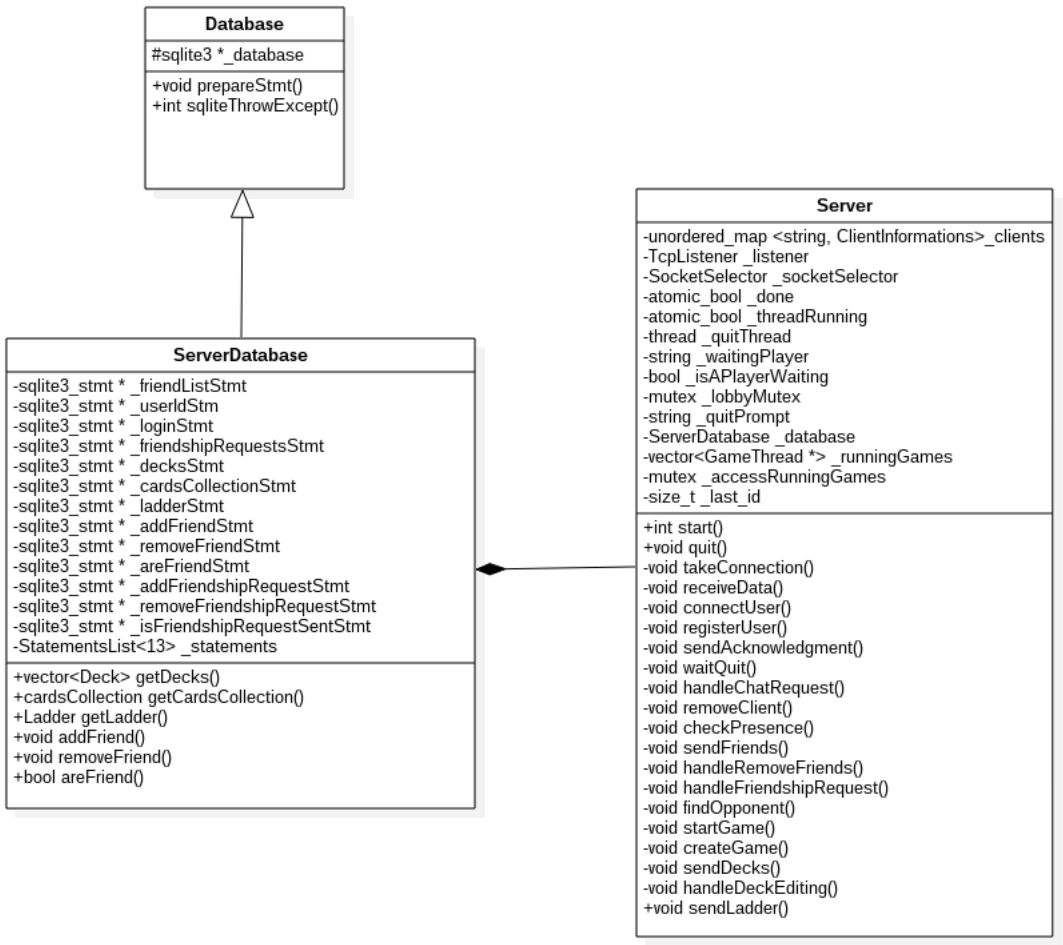
3.4 Diagramme de classe du et des cartes



Comme indiqué sur ce diagramme, un est composé de cartes, 20 pour être précis. Les cartes sont soit des Creature soit des Sort et on ne peut posséder que maximum 2 exemplaire d'une même carte au sein d'un même . A savoir également que les sont indépendants, il est donc possible de posséder uniquement 2x la même carte, mais avoir plusieurs qui possèdent 1 ou 2x cette carte.

L'application des effets est simple : chaque effet est un effet direct, que ce soit infliger X points de dégats, comme soigner un allié ou encore interdire le joueur adverse de jouer pendant Y tours. Un effet plus complexe est également un effet direct, seulement, il va modifier la liste de contraintes de la partie et c'est cette liste qui va permettre d'appliquer des effets sur la durée, ou en respectant bien certaines conditions (applique tel effet tant que tel créature est en vie par exemple).

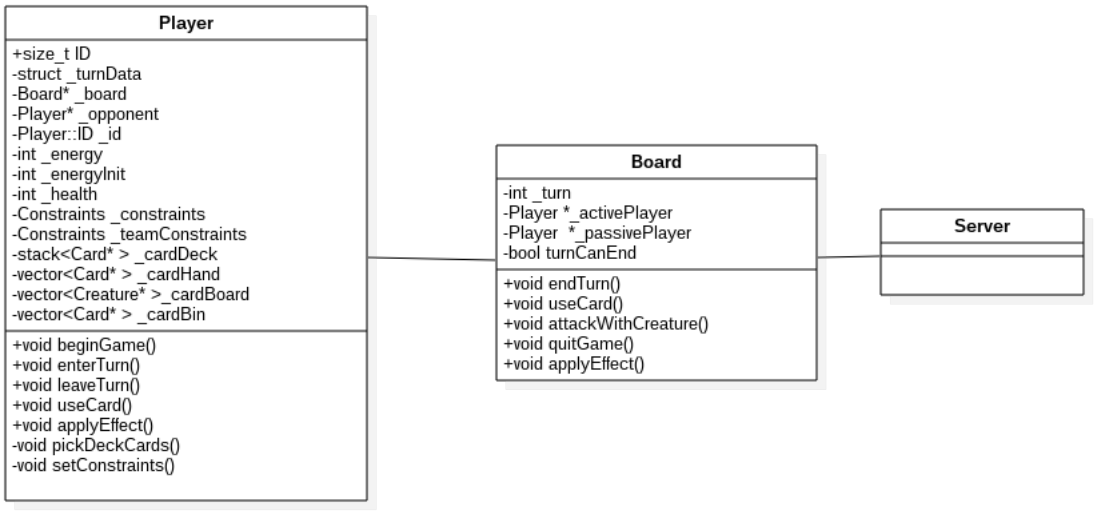
3.5 Diagramme de classe du serveur et de sa base de données



Ici, la structure est un peu plus complexe : Le serveur est composé d'une base de donnée en SQL qui est héritée par une base de donnée utilisable par l'application serveur (en c++ cette fois).

C'est ce serveur qui va naturellement tout gérer, des parties, jusqu'aux requêtes d'amis en passant par la gestion/modification des /de compte. Tout sera stocké de son côté dans la base de données. Le client ne possedera aucune données, il est obligé de passer par les serveur quel que soit son intention.

3.6 Diagramme des classes intervenant dans une partie (simplifié)



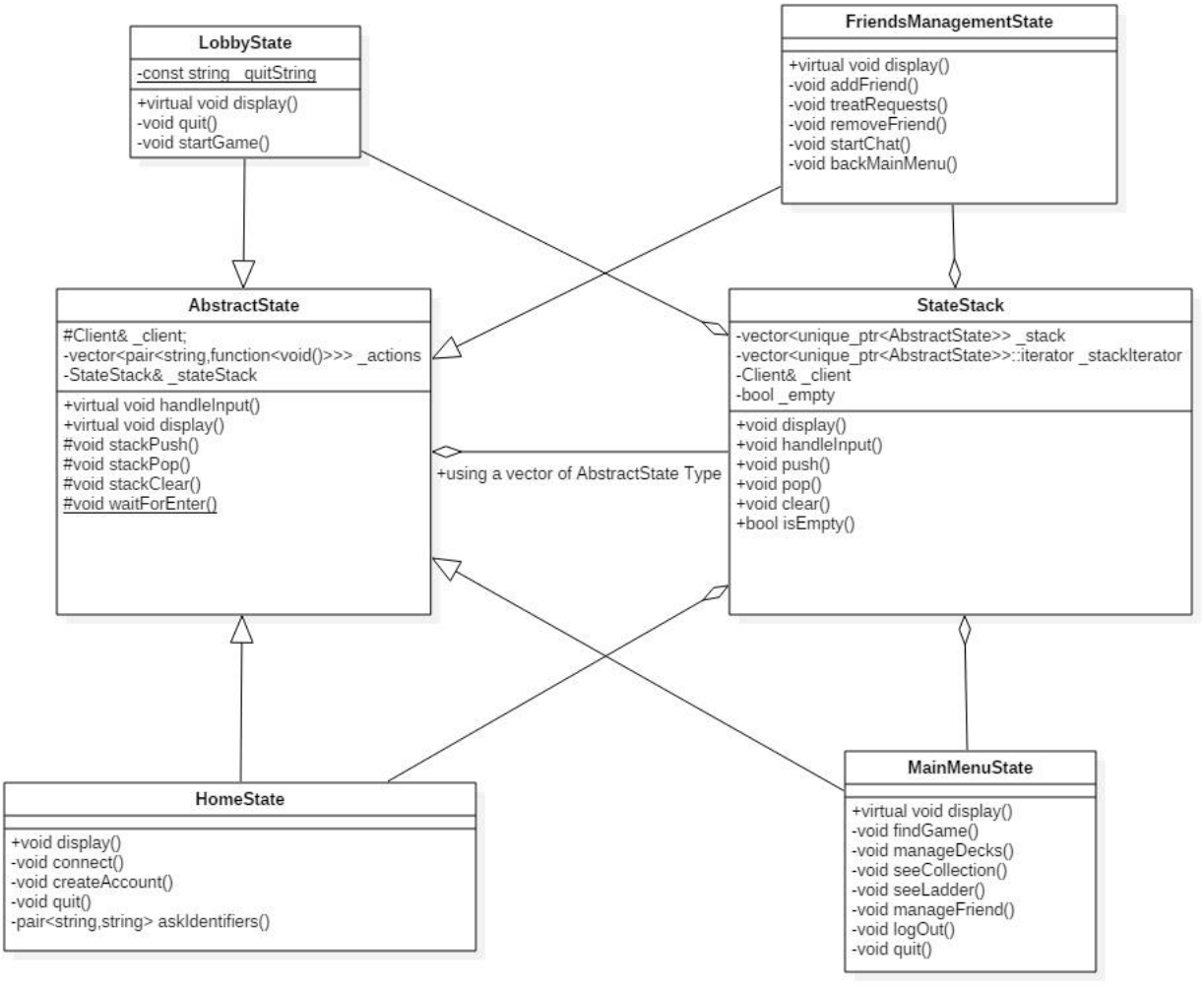
Il est à noter que c'est un diagramme de classe, et non d'activités, il regroupe juste les plus grosses classes intervenant dans la partie, et non pas leur lien/fonctionnement entre elles. Pour qu'une partie puisse avoir lieu, il nous faut 2 joueurs connecté au serveur qui s'occupera lui-même de modifier le plateau de jeu.

3.7 Diagramme des classes nécessaires au menu en console

Toutes les méthodes servant à une expérience de jeu optimal sont implémentées, mais en console. L'interface graphique arrive à la prochaine étape du développement de ce projet, tout ce qui gère le fonctionnement du jeu est néanmoins présent.

Enfin, voici les différentes classes représentant le menu, le tout est divisé en 2 parties pour plus de clarté.

3.8 Première moitié : Menu et fonctionnalités principales

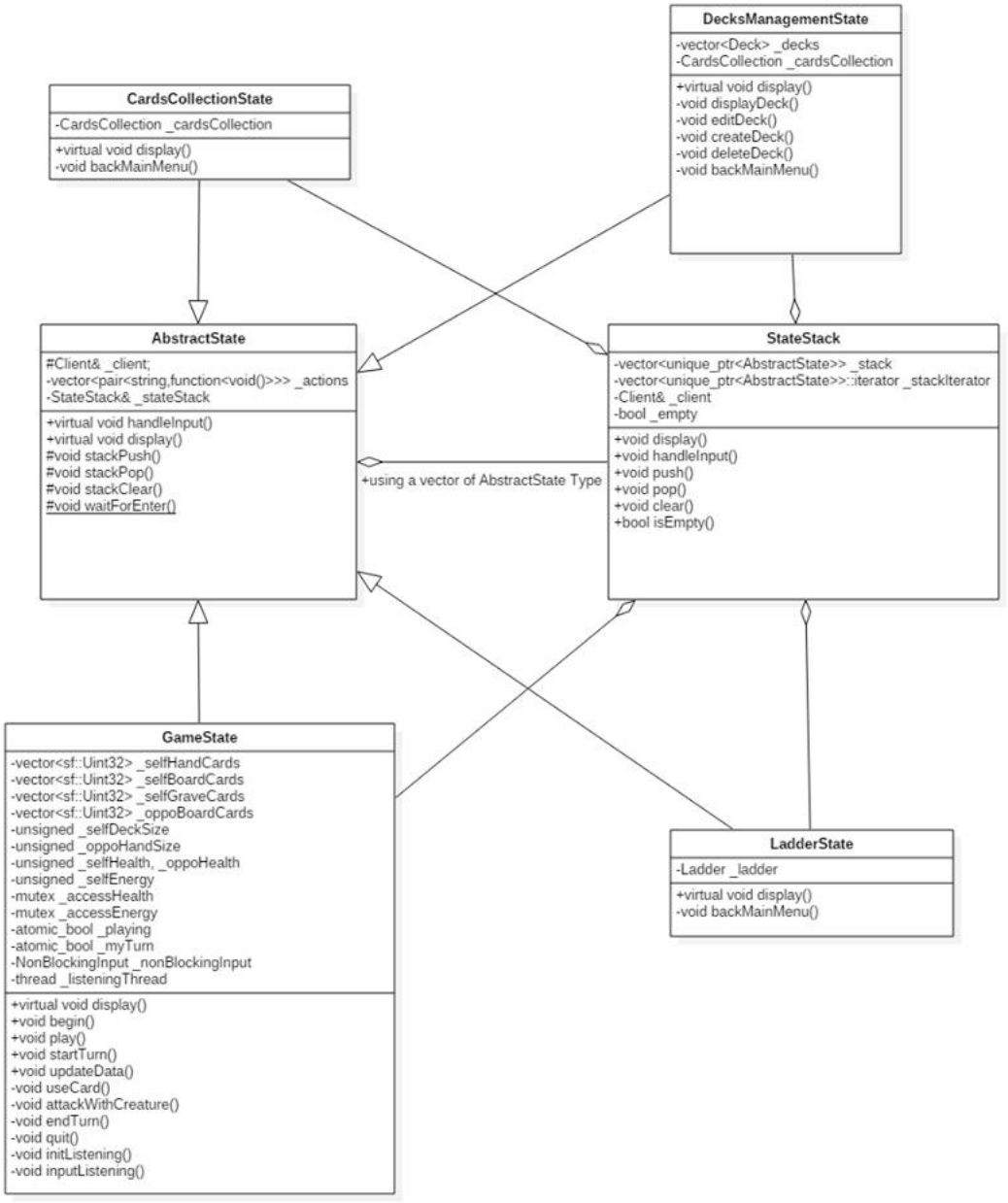


C'est au moyen de stack que nous avons implémenté un menu fluide, efficace et facile d'utilisation.

En effet, pour aller plus loin dans le menu, il suffit de "push" notre choix, et pour un retour en arrière il suffit de faire un "pop", et le tour est joué.

Ainsi, pour le menu et toutes ses fonctionnalités, que ce soit au niveau de la gestion des amis jusqu'au menu principal, nous avons une classe abstraite héritée (**AbstractState**) ainsi qu'une utilisation d'une deuxième classe (**StateStack**). C'est grâce à ces deux dernière que tout implémentation supplémentaire dans le menu ou autre est possible.

3.9 Deuxième moitié : fonctionnalités plus avancées



Ici, nous nous occupons un peu plus de l'expérience de jeu. Nous avons un gestionnaire de cartes, un gestionnaire de deck ainsi qu'un classement des joueurs et tout ce qui est nécessaire au bon fonctionnement d'une partie.

La seule particularité de cette partie réside dans la classe **GameState** qui comporte des mutex, des valeurs atomiques ainsi que des entrées non bloquantes. Etant données qu'il y aura de nombreuses entrées/sorties au niveau de cette classe-là, ces différents éléments permettront de ne pas se marcher sur les pieds lors des accès en mémoires de ses attribus.

4 Index