EPITECH INNOVATIVE PROJECT

AA1 – MAGIC TACTIL

EIP - 2014

[Farsi_m – Jouhri_o – Periph_a – Pucheu_m – Jaber_a – Ortis_l – Labori_b – Barouk_r]

7/25/2012

Ce bilan a pour but de vérifier la bonne conception de l'architecture du projet. Ce document est voué à évoluer itérativement dans les premières phases d'implémentation.

SOMMAIRE

Contents

1. INTRODUCTION

1.1 Rappel de l'EIP

L'Epitech Innovative Project (EIP) est l'étape principale du cursus master de l'EPITECH. Il a pour but d'accompagner l'étudiant vers la professionnalisation. Il se déroule sur 18 mois et est à effectuer par groupe de 6 personnes minimum. C'est donc le moment du cursus ou les étudiants peuvent montrer « au monde extérieur » de quoi ils sont capables dans un projet plus ou moins ambitieux. Le projet dont ce document traite est Magic Tactil.

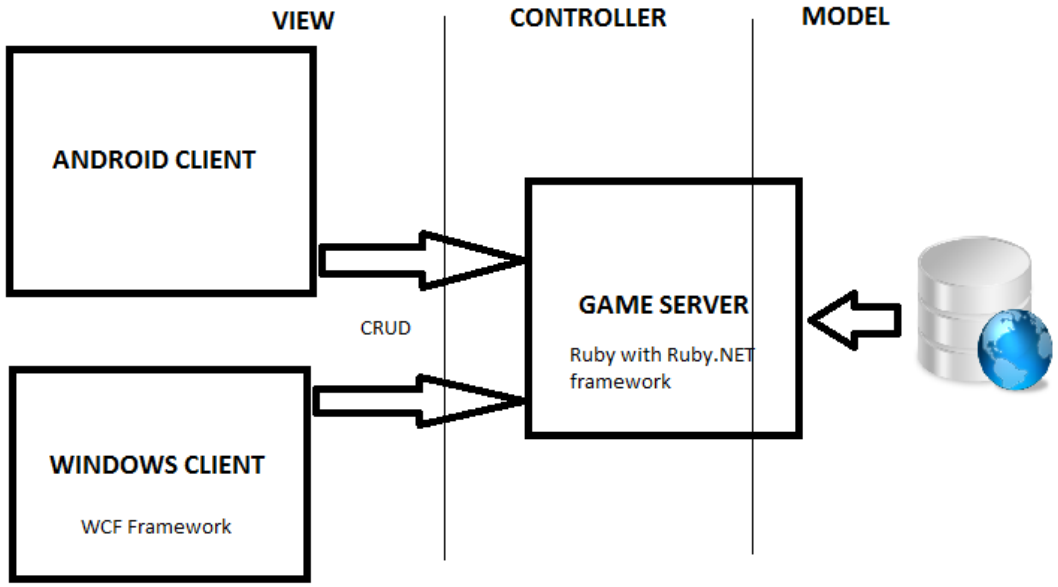
1.2 Contexte et périmètre du projet

Le projet consiste en la réalisation d'une plate-forme basé sur le jeu de carte Magic : the gathering. Cette dernière permettrait aux utilisateurs d'avoir les mêmes interactions que le vrai jeu de carte. L'application sera disponible sur Surface, Surface 2, Windows 8 ainsi que sur Android. Nous avons donc décidé de cibler une grande majorité des utilisateurs. Notre objectif principale est donc de relancer le marché du jeu de carte Magic en ligne. Notre application permet bien sûr aux utilisateurs de s'affronter, mais nous voulons également développer une plateforme « sociale » afin de créer une communauté. Enfin notre Bussiness Plan est de permettre à l'utilisateur d'acheter de nouvelles fonctionnalités (dans notre cas des cartes et des boosters) et de permettre à un utilisateur de revendre ces cartes tout en récupérant une commission sur cette vente. Le coût principal engendré par le projet serait la location d'un serveur.

2. REPRESENTATION DE L'ARCHITECTURE GLOBALE

Ce projet est basé sur une architecture MVC classique : Le serveur aura pour rôle de proposer un certains nombre de couche d'accès aux données (ORM) qui seront, chacun, gérés par leurs contrôleurs associés. Le client aura simplement à utiliser le modèle CRUD proposé par le serveur.

Voici un schéma illustrant l'architecture globale du projet :



3. CHOIX TECHNOLOGIQUE

Pour ce projet nous avons fait le choix d'utiliser les technologies suivantes :

- C# SDK Surface 2 pour le client Windows
- Ruby. NET pour le serveur
- Java SDK Android pour le client Android

Client sous Windows

Nous avons fait le choix Surface 2 pour les raisons suivantes :

- La montée en flèche des produits tactiles.
- La sortie de Windows 8, un OS tactile. Une compatibilité avec la table Surface 2, les Pcs munis d'un Windows et les tablettes munies d'un Windows.
- Une compatibilité avec les anciennes versions de Windows.
- C'est une technologie nouvelle.

Ainsi nous misons sur le fait que notre client sera utilisable par le plus grand nombre, les utilisateurs de Windows étant les plus nombreux. Nous pensons que l'avenir tend vers le tout tactile, nous souhaitions donc travailler dans cette ergonomie, Magic étant un jeu de cartes, le cadre est idéal. Cette conviction que l'avenir de l'informatique tend vers le tout tactile est amplifié par l'annonce de Windows 8 et des tablettes Surface (ne pas confondre avec la table) annoncé très récemment.
Cela nous laisse entendre que d'ici peu, les PC avec écran tactile seront un standard.

Le client sous Android

Dans le but de ne pas supporter uniquement les utilisateurs Microsoft, nous avons décidé aussi d'être présent sur les tablettes Android. À ce niveau-là le choix technologique est assez trivial.

Le serveur

Nous avons choisi Ruby.NET pour les raisons suivantes :

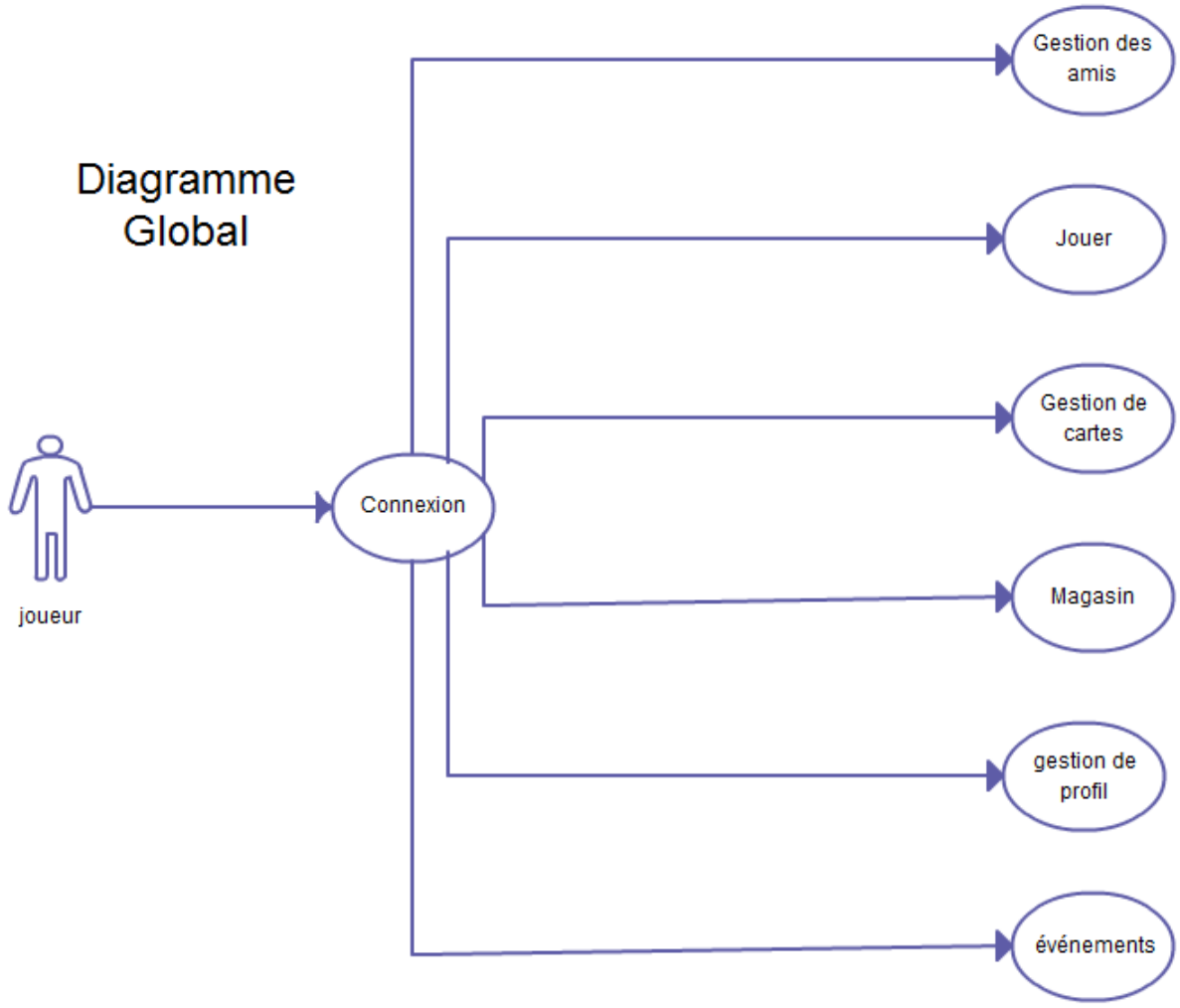
- implementation de WCF
- compatibilité facile de la WCF vers les autres plateformes (Android, IOS...)
- La puissance de l'outil JSON

Avec ce choix, notre serveur pourra communiquer de manière identique avec nos deux clients, et si par la suite nous créons un client IOS, nous n'aurons pas à le modifier. L'implémentation de la WCF nous évitera la mise en place d'un protocole.

4. VUE GLOBALE DU PROJET

Voici, dans un premier temps, une définition des différents choix possibles de navigation qui s'offre à l'utilisateur par rapport aux fonctionnalités défini dans le cahier des charges.

Diagramme de cas d'utilisation global :



5. VUE PROCESSUS

Le diagramme ci-dessus a permis de présenter le projet d'un point de vue « usage ». Ce chapitre va lui, présenter ces interactions cette fois-ci d'un point de vue beaucoup plus technique.

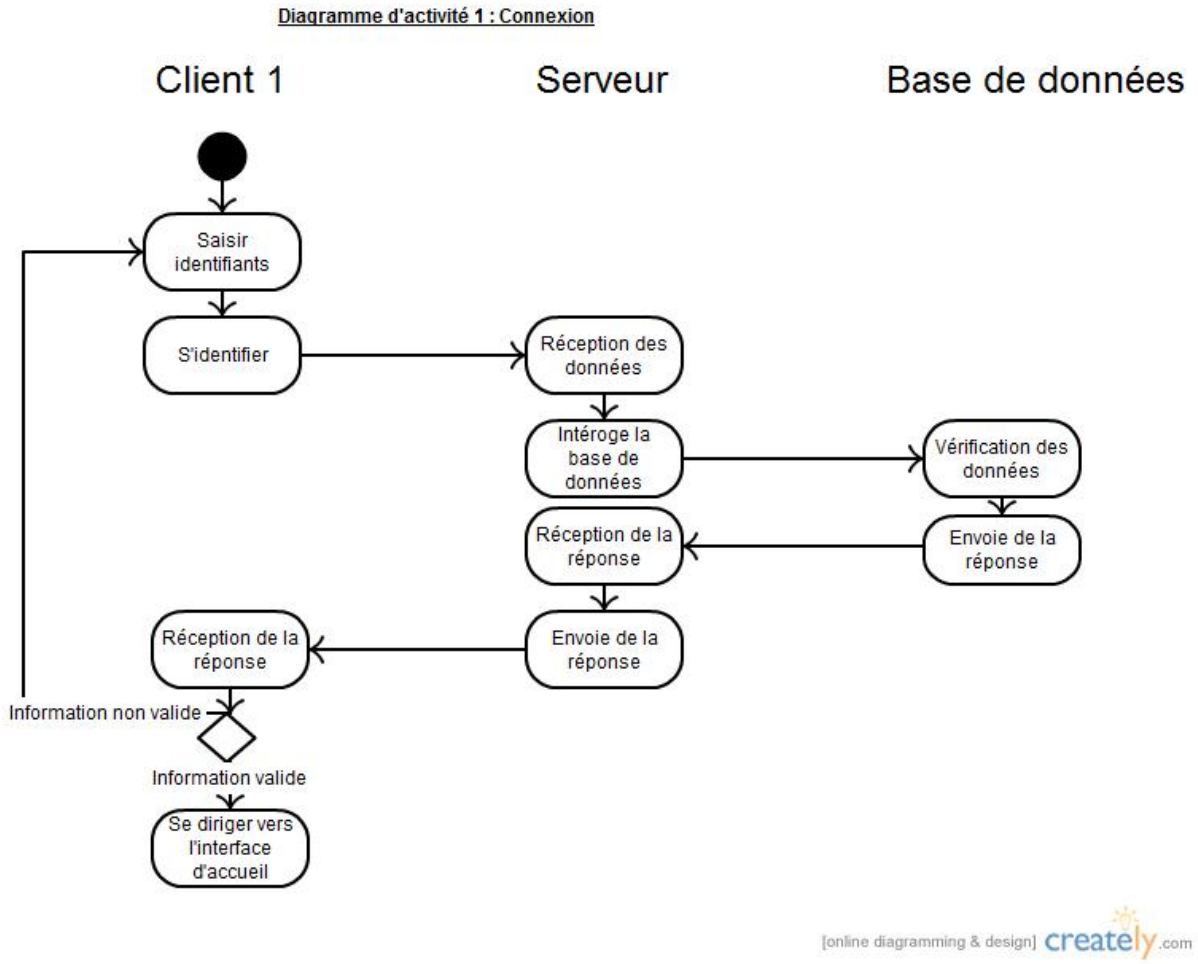


Diagramme d'activité 2 : Accueil

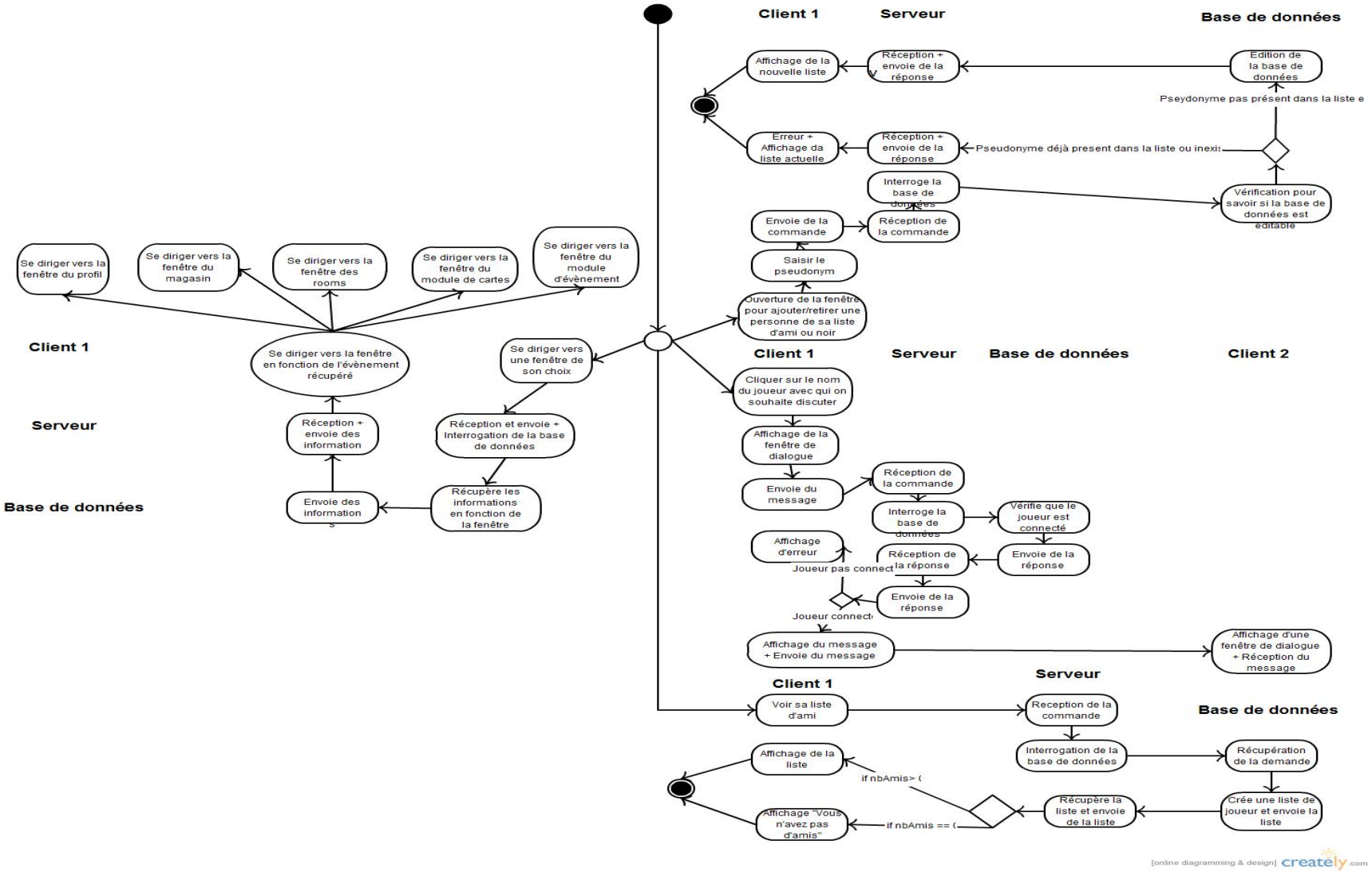


Diagramme d'activité 3 : Le profil

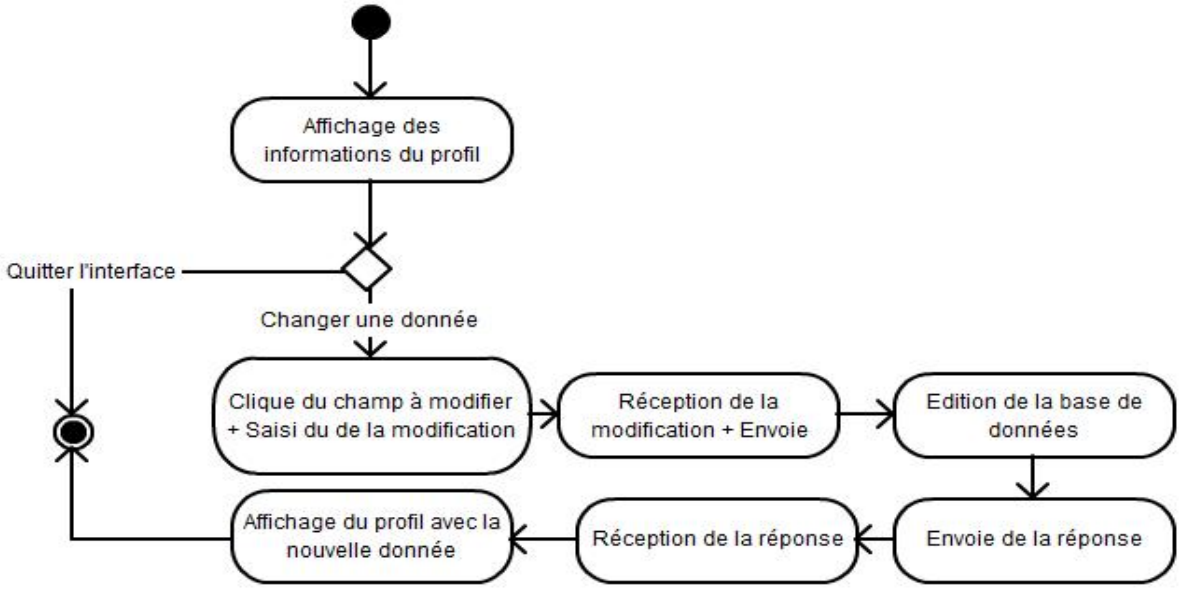


Diagramme d'activité 4 : Magasin

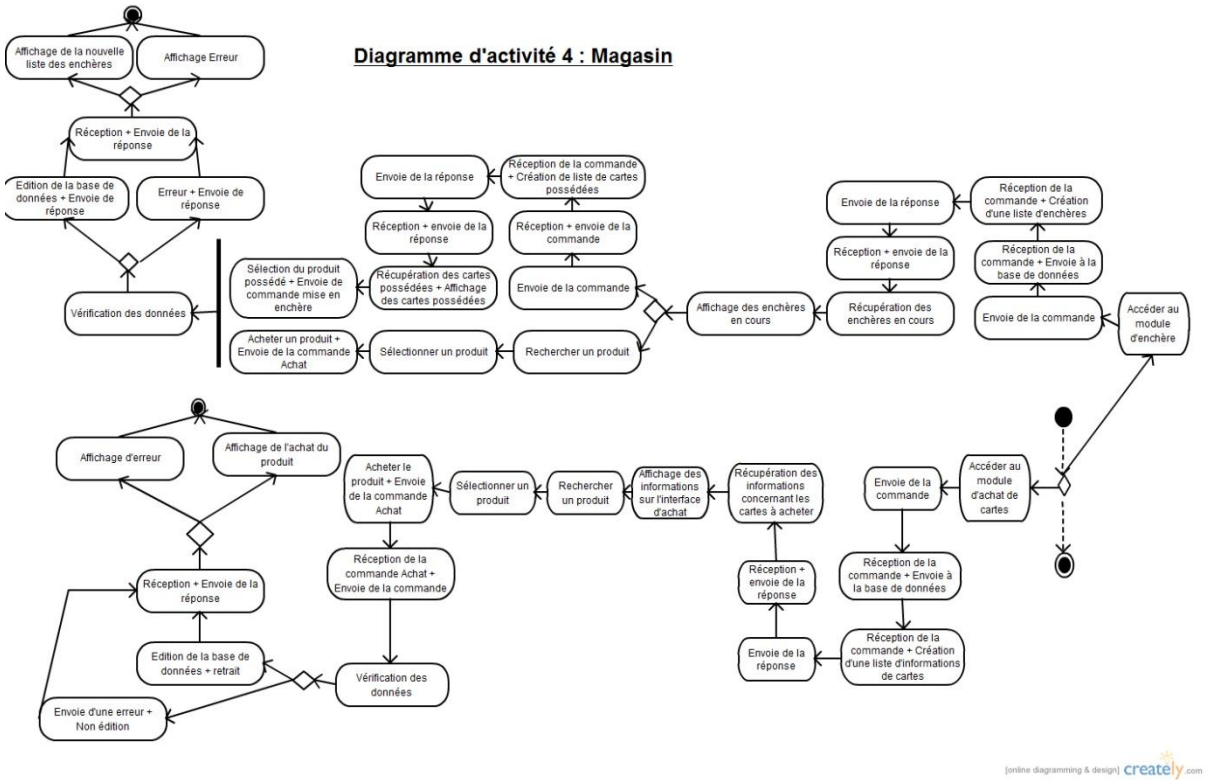


Diagramme d'activité 5 : Module de cartes

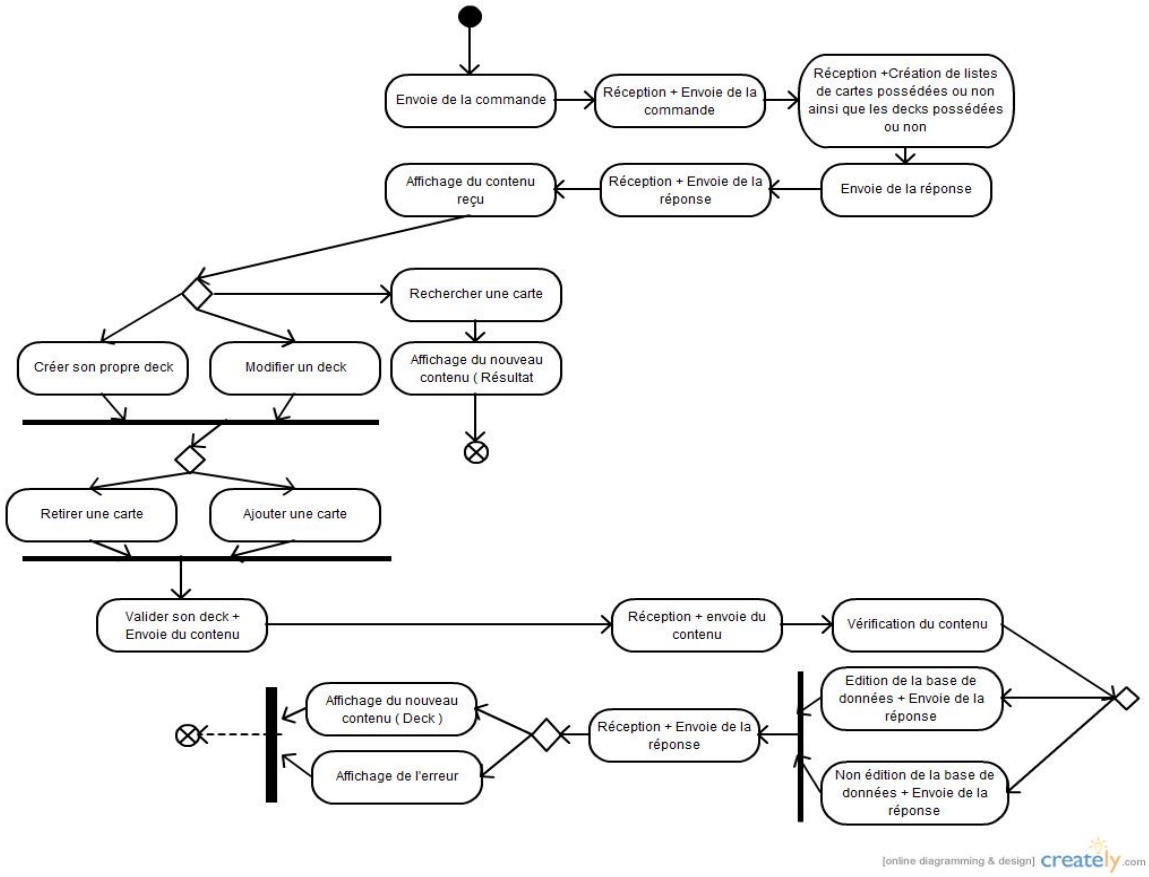


Diagramme d'activité 6 : Module de rooms

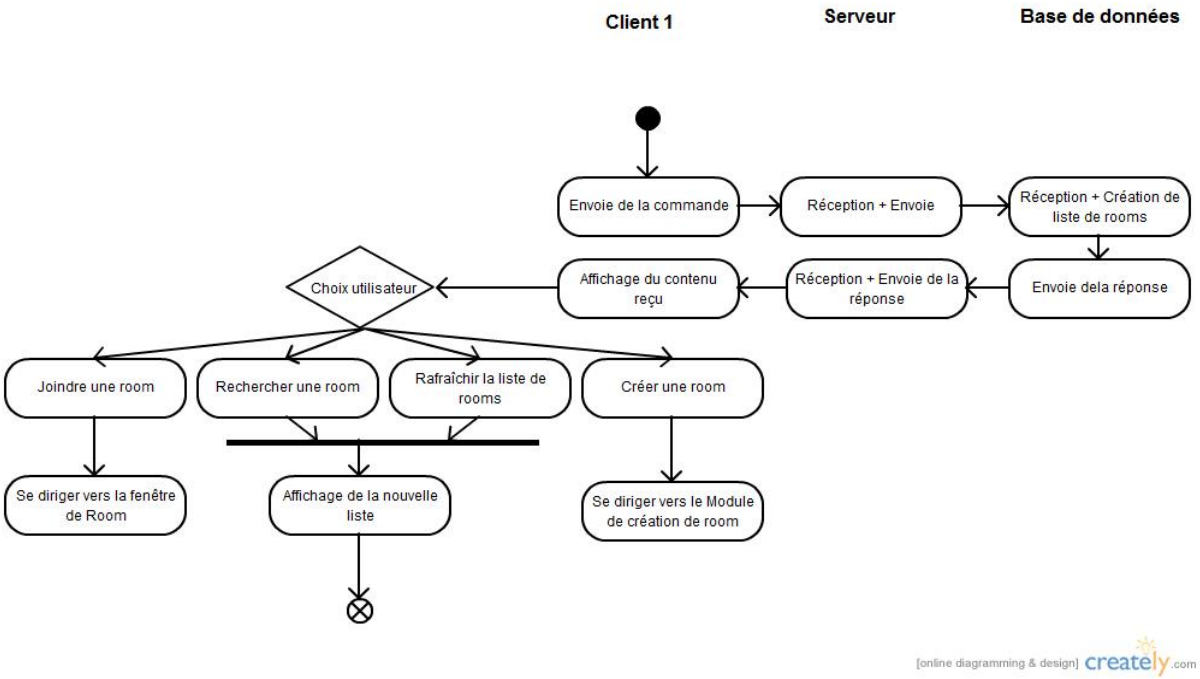


Diagramme d'activité 7 : Module de création de rooms

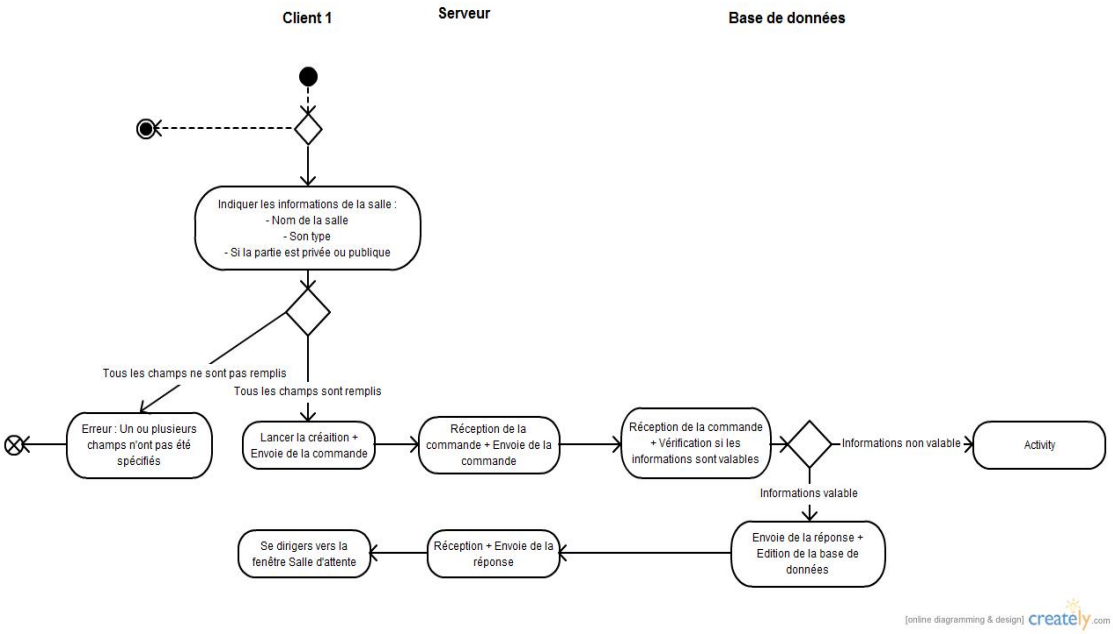


Diagramme d'activité 8 : Salle d'attente

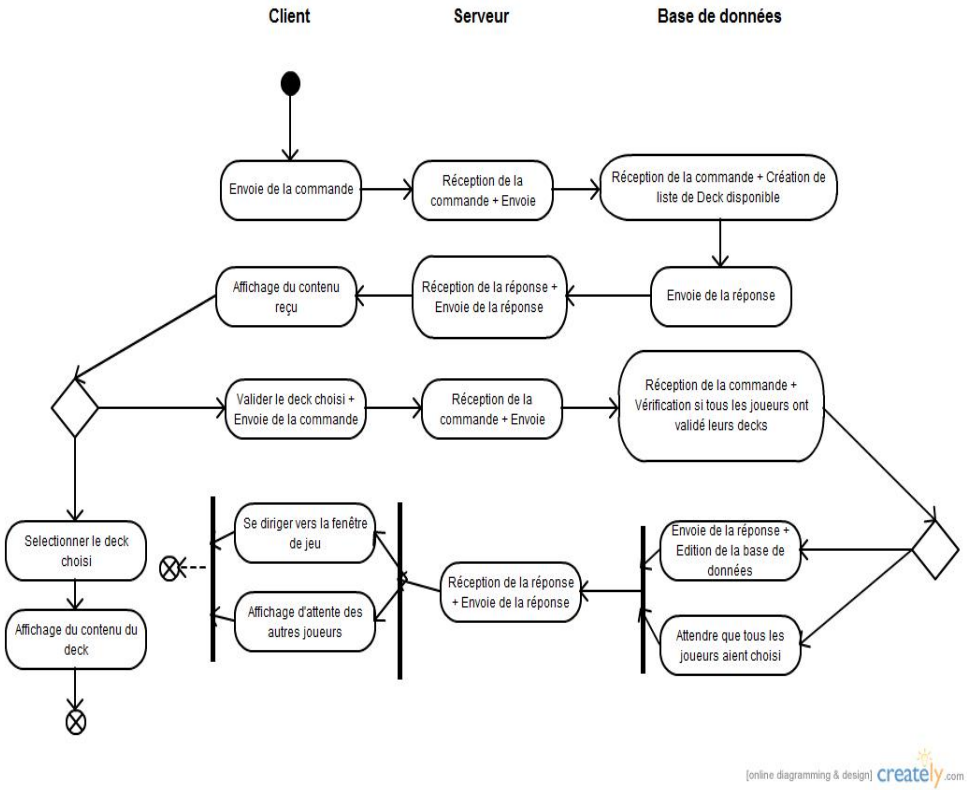


Diagramme d'activité 9 : Module d'évènement

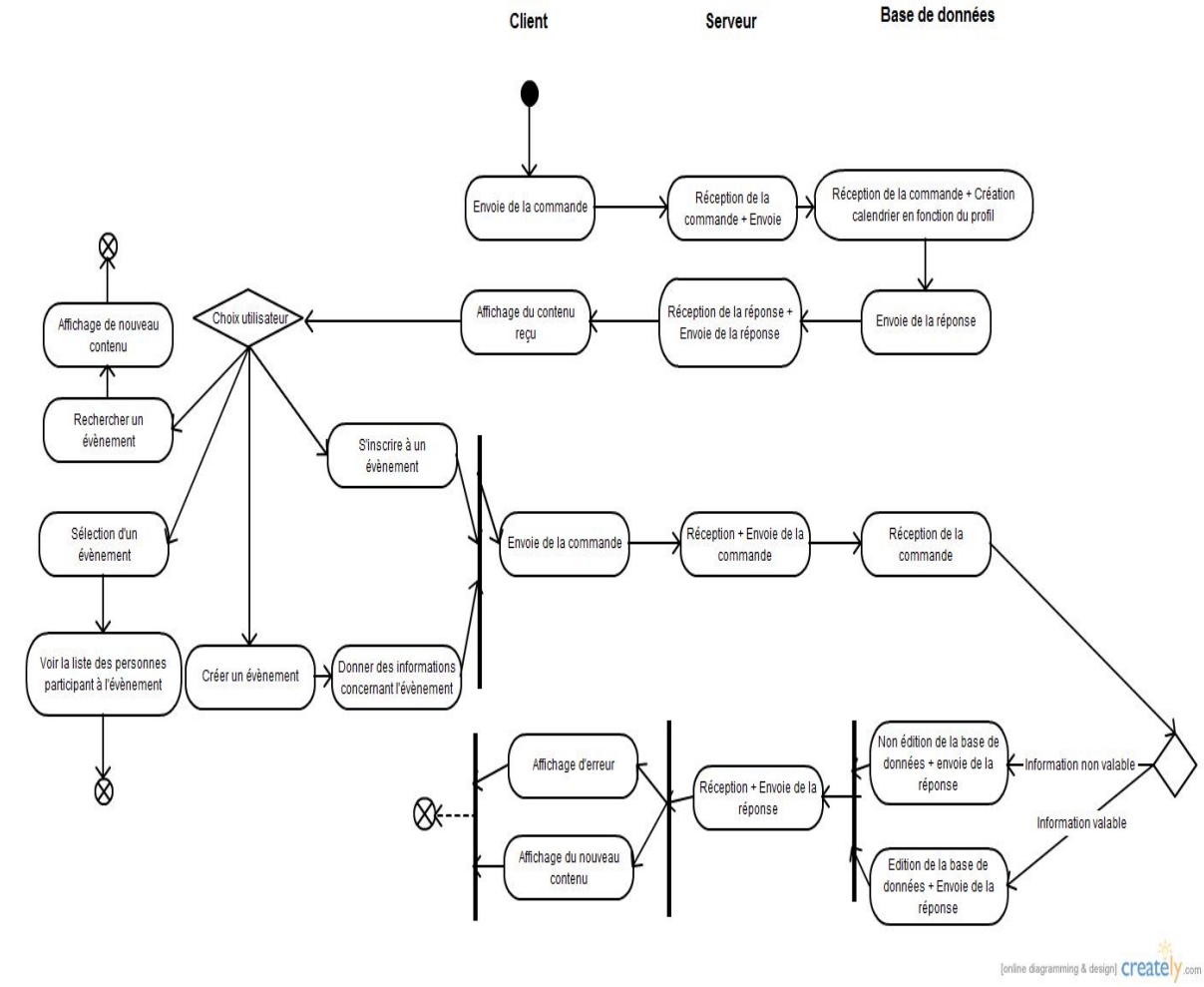
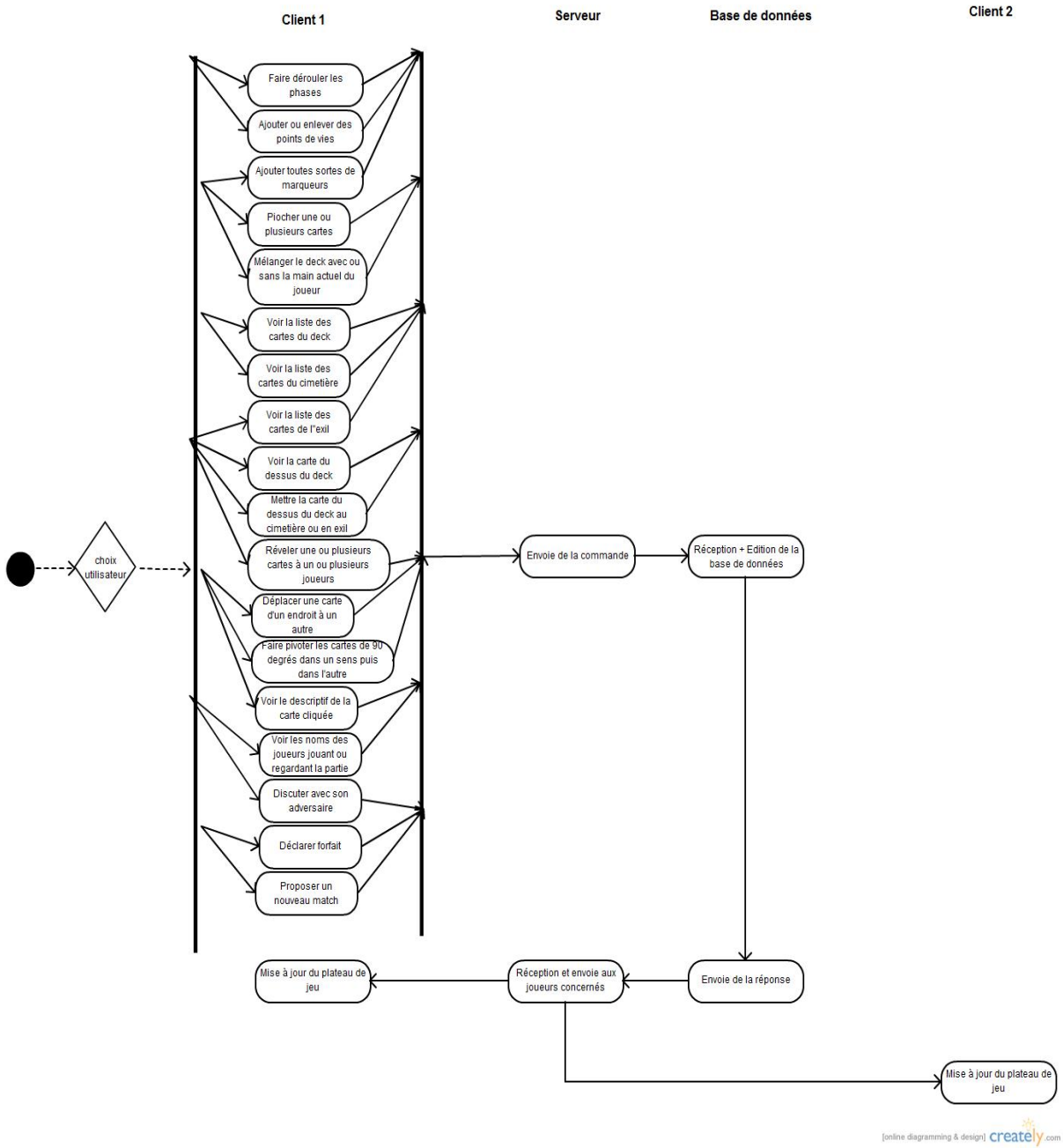


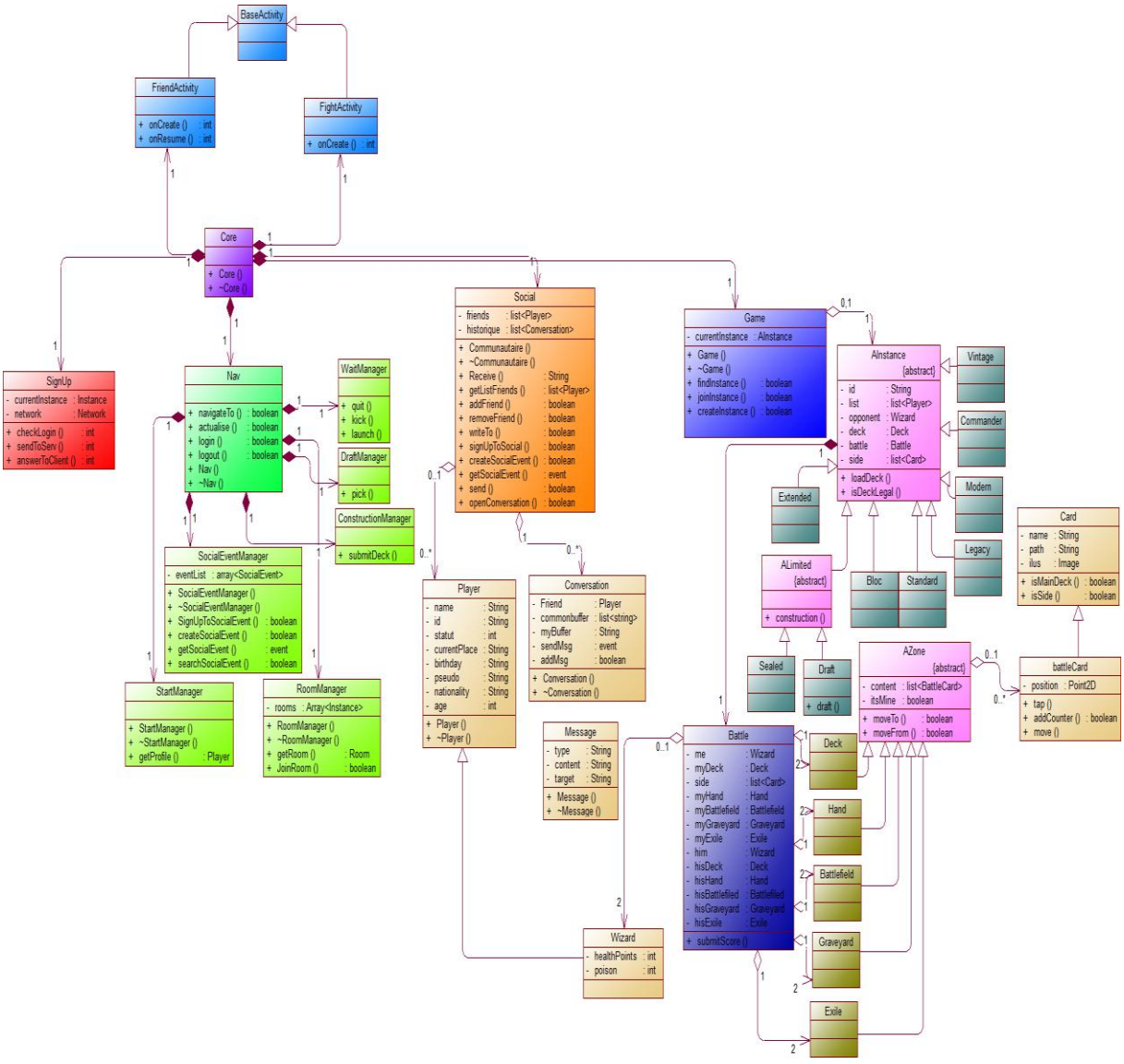
Diagramme d'activité 10 : Fenêtre de jeu



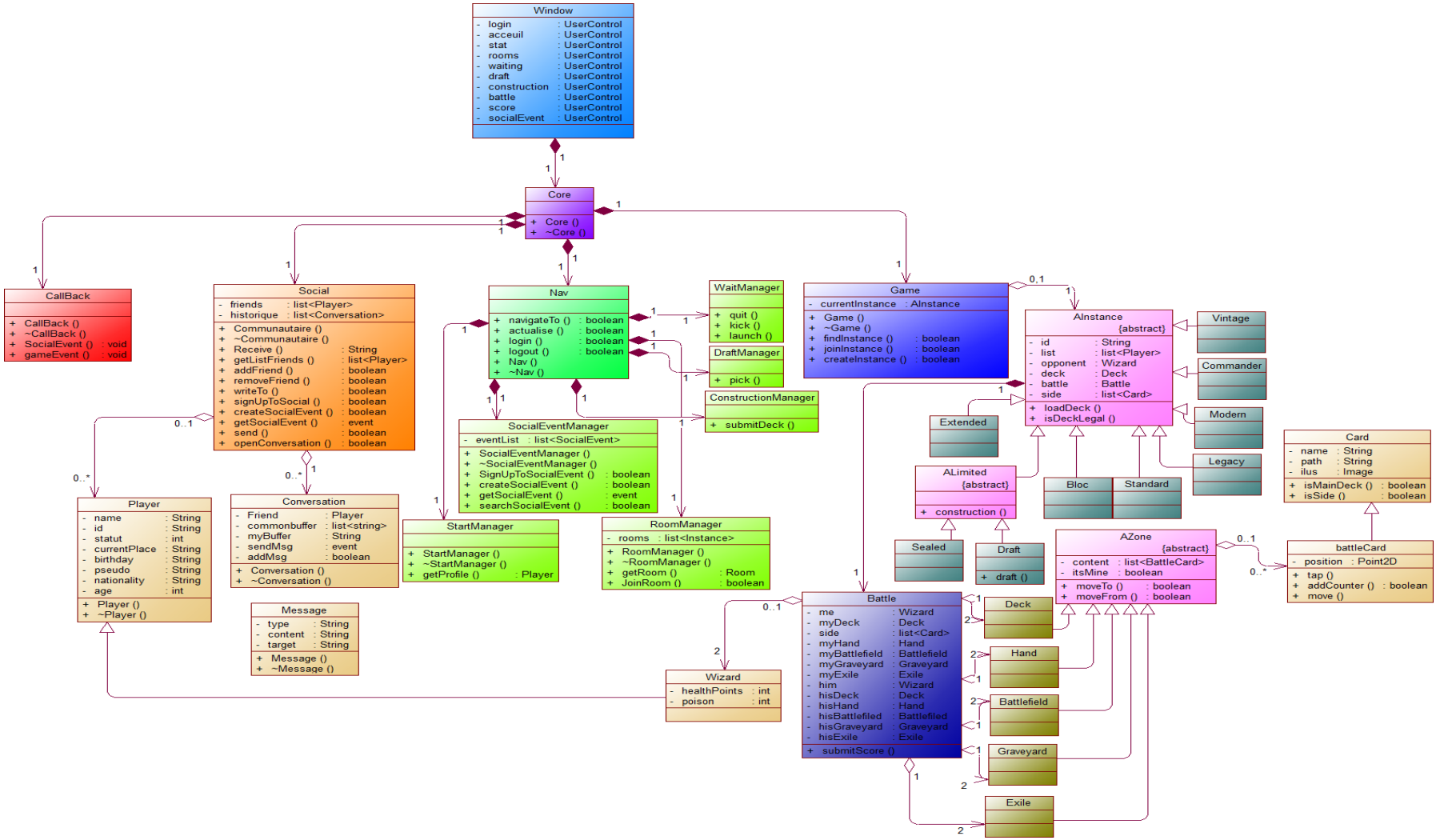
6. COUCHES APPLICATIVES

Les diagrammes suivants proposent une vue plus détaillée des clients Android et Windows ainsi que du serveur.

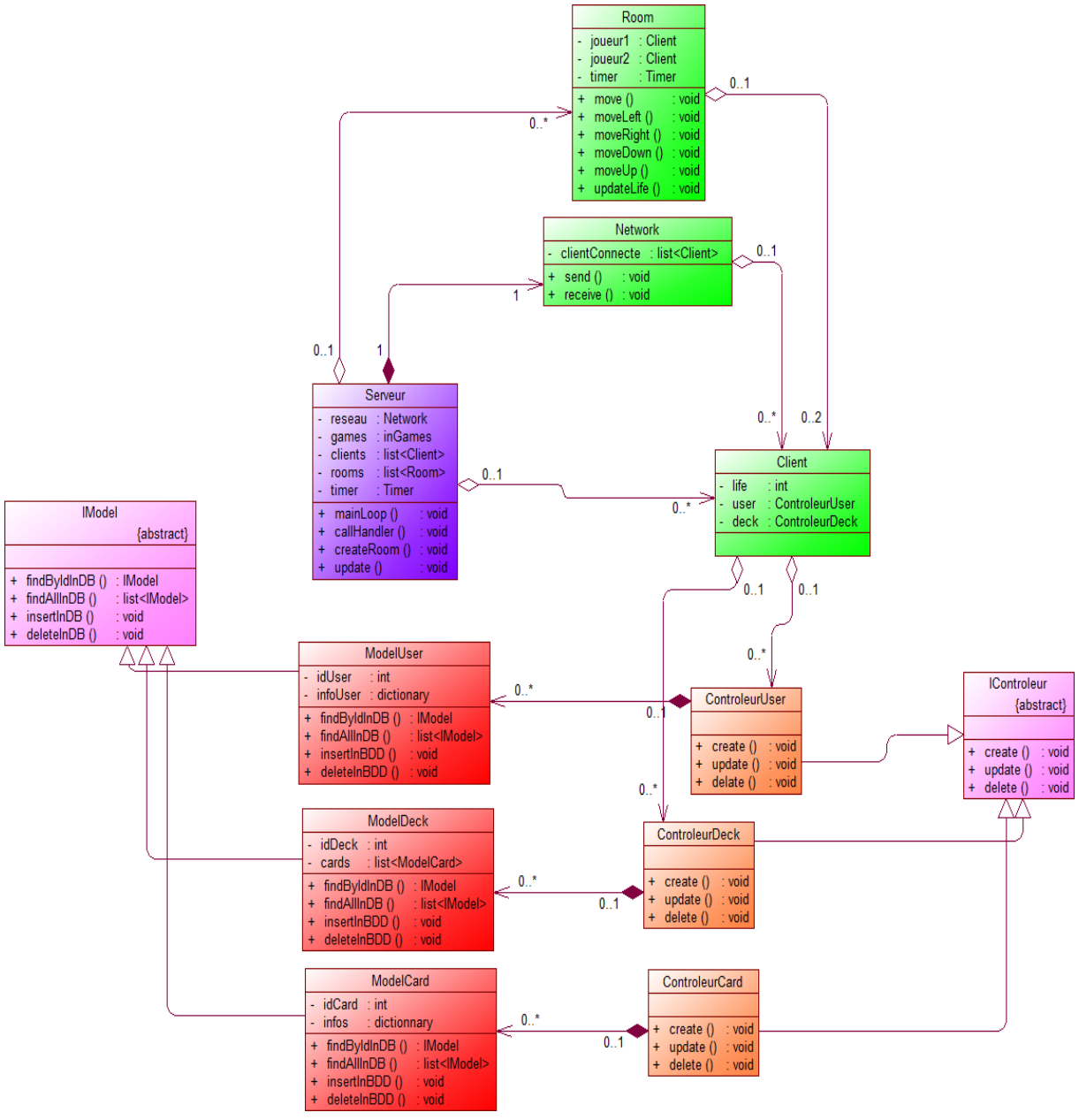
Client Android



Client Windows

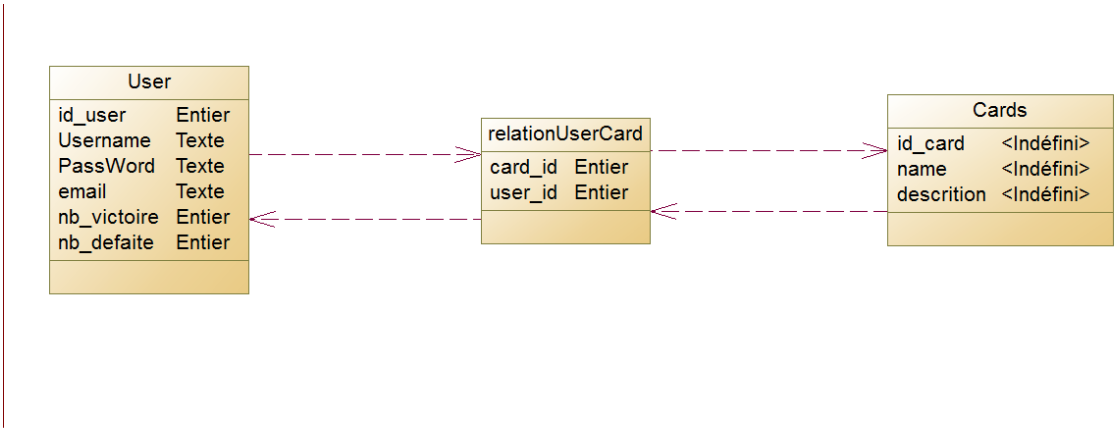


Serveur


Room
- joueur1 : Client
- joueur2 : Client
- timer : Timer
+ move () : void
+ moveLeft () : void
+ moveRight () : void
+ moveDown () : void
+ moveUp () : void
+ updateLife () : void
Network
- clientConnecte : list<Client>
+ send () : void
+ receive () : void
Serveur
- reseau : Network
- games : inGames
- clients : list<Client>
- rooms : list<Room>
- timer : Timer
+ mainLoop () : void
+ callHandler () : void
+ createRoom () : void
+ update () : void
Client
- life : int
- user : ControleurUser
- deck : ControleurDeck
IModel
{abstract}
+ findByIdInDB () : IModel
+ findAllInDB () : list<IModel>
+ insertInDB () : void
+ deleteInDB () : void
ModelUser
- idUser : int
- infoUser : dictionary
+ findByIdInDB () : IModel
+ findAllInDB () : list<IModel>
+ insertInBDD () : void
+ deleteInBDD () : void
ControleurUser
+ create () : void
+ update () : void
+ delate () : void
IControleur
{abstract}
+ create () : void
+ update () : void
+ delete () : void
ModelDeck
- idDeck : int
- cards : list<ModelCard>
+ findByIdInDB () : IModel
+ findAllInDB () : list<IModel>
+ insertInBDD () : void
+ deleteInBDD () : void
ControleurDeck
+ create () : void
+ update () : void
+ delete () : void
ModelCard
- idCard : int
- infos : dictionnary
+ findByIdInDB () : IModel
+ findAllInDB () : list<IModel>
+ insertInBDD () : void
+ deleteInBDD () : void
ControleurCard
+ create () : void
+ update () : void
+ delete () : void
0..1
0..*
1
0..2

7. VUE DONNEES

La base de données est composée d'une table **User** ainsi que d'une table **Card**. En admettant que plusieurs **User** peuvent posséder la même **Card** dans leur decks respectifs, il serait peut inspiré de créer autant de fois une **Card** qu'il y a d'utilisateur qui la possède. Afin de ne pas être « gourmand en mémoire » et d'optimiser les requêtes il a été décidé de mettre en place une **table de jointure** nommée **RelationUserCard**.



8. QUALITE

Architecture

Ce projet touchant une forte communauté de joueur, ce dernier va être amené à évoluer de façon itérative par rapport au feedback des utilisateurs. Il serait donc intéressant de ne pas se retrouver avec une architecture « rigide » qui ne permettrait pas ce genre d'évolution sans sacrifier des heures à bricoler les différents exécutables. Dans l'optique de permettre une certaine « souplesse », l'architecture MVC permet de séparer naturellement les taches entre le client, qui a simplement accès a des données qu'il manipule via le modèle CRUD et le serveur qui, lui, abstrait le client des mécanismes interne de traitement de données.