PPRO0605
2020 / 2021

Boissier Sébastien
Moreaux Léo
Sindic Mathieu
Veron Tristan
Villa Benjamin

IA Magic The Gathering
Projet L3

Delisle Pierre
Rabat Cyril

Université de Reims
Champagne-Ardenne

Table des matières

Architecture de l'application **2**

Carnet de bord **4**
- Organisation 4
- Ébauche du projet 5
- Objectifs et mise en place 6

Conception Logicielle **7**
- Diagramme de classe 8

Modélisation des échanges **14**
- Phase de départ 14
- Phase d'activation de capacités 15
- Phase d'éphémères 15
- Phase des sorts 16
- Phase principale 16
- Phase d'attaque 17
- Phase de blocage 17
- Phase de résolution des dégâts 18
- Phase principale 2 18
- Phase de fin 19

Représentation des données **20**
- Cartes 20
- Effets 24
  - Effets déclenché 24
  - Effets activable 26
  - Effets statique ou Evergreen 27

Implémentation **29**

Mise en service **31**

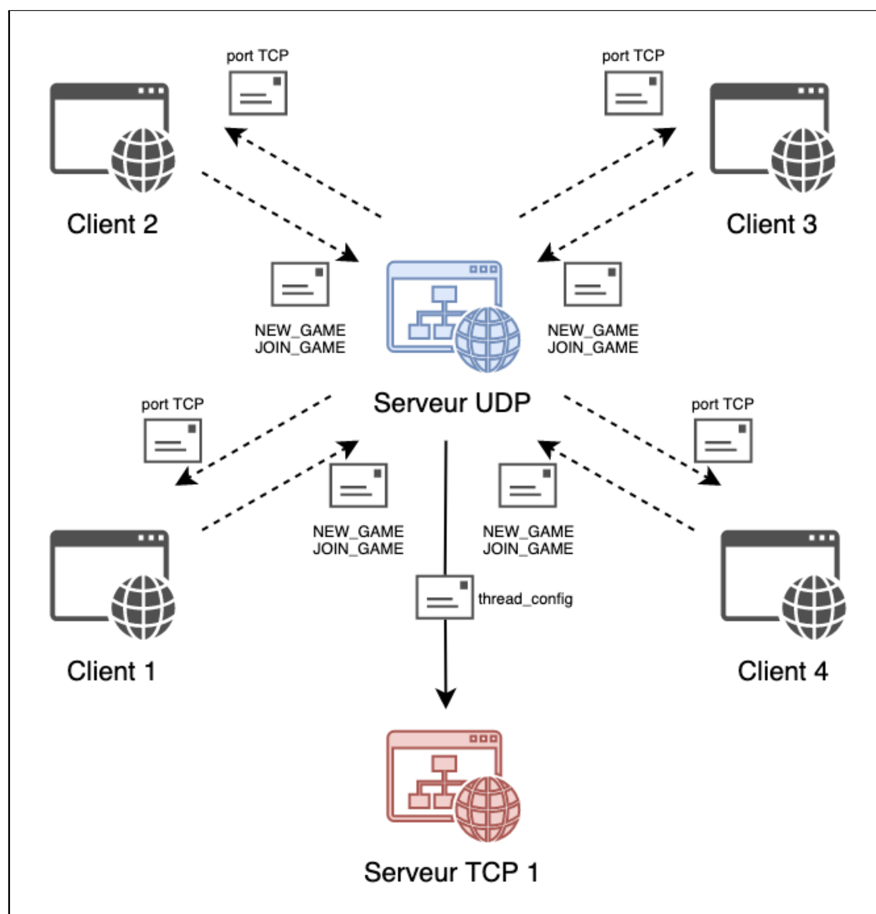
Points à améliorer **33**

Suite du projet **34**

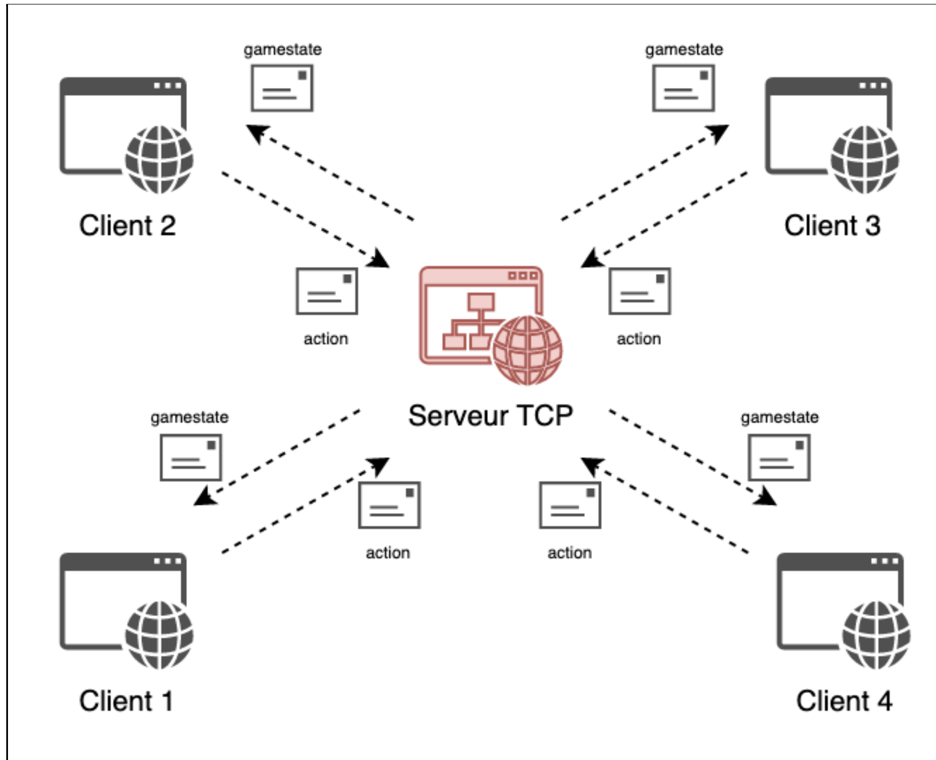
Conclusion **35**

Architecture de l’application

L’application représente l’implémentation du jeu de cartes Magic The Gathering via une interface en lignes de commande et dont la particularité est de permettre l'affrontement en ligne de plusieurs joueurs d’après les règles du Free For All. Chaque joueur peut être contrôlé par un humain ou déléguer son exécution à une IA qui s’occupera de jouer à sa place. Cette application est structurée en 3 types de modules différents : le serveur UDP, le serveur TCP et les clients. Les schémas à suivre ne représentent que l’exécution d’une partie, mais il est tout à fait possible d’exécuter n parties en parallèle car chaque serveur TCP correspond à un thread dissocié.



Initialement, les clients vont se connecter au serveur UDP en indiquant leur volonté de créer ou rejoindre une partie en cours et dans le cas où le serveur accepte la requête il transmettra alors le port du serveur tcp aux concernés.



Et une fois connectés à leur partie (serveur TCP), les clients viendront à transmettre des actions de jeu chacun leur tour en suivant le déroulement strict d'une partie selon les règles officielles.

Les communications entre les différents modules se font via l'envoi de flots binaires représentant les structures concernées à travers des sockets UDP et TCP.

L'implémentation a été faite en Python3 et s'exécute sous un environnement UNIX et Windows.

Carnet de bord

Organisation

Le but de notre projet est de réaliser le déroulement d'une partie du jeu de carte Magic ainsi que de modéliser les éléments nécessaires au bon enchaînement de celle-ci. L'objectif étant de pouvoir jouer à plusieurs joueurs en tant qu'humain, et de créer un bot qui doit jouer en autonomie à partir d'un deck. La finalité de ce projet peut ainsi permettre d'ajouter un aspect d'intelligence artificielle au bot pour que celui-ci apprenne et s'améliore au fil des parties qu'il exécutera.

Pour réaliser ce projet, nous nous sommes réunis en un groupe de cinq étudiants de Licence 3. Notre groupe se compose ainsi de Boissier Sébastien, Moreaux Léo, Sindic Mathieu, Veron Tristan et Villa Benjamin. Pour structurer notre équipe, nous avons désigné Veron Tristan en tant que chef de projet. Le rôle de celui-ci étant de permettre de faire le lien entre le groupe et les responsables des sujets, ainsi que de déléguer les tâches à accomplir.

Nous avions ensuite deux mois en vue de le mener à bien, durant lesquels nous remplissions un journal décrivant les différentes tâches réalisées dans la journée.

Ébauche du projet

Dans l'intention de mener à bien notre projet, nous nous sommes très rapidement réunis par le biais d'un serveur discord afin de pouvoir commencer à discuter et débattre sur la façon dont nous appréhendions et comprenions le sujet que l'on avait choisi. Cela nous permettant ainsi d'exposer les premières idées et solutions auxquelles nous pensions.

De manière à pouvoir avoir tous les mêmes informations ainsi que les mêmes règles du jeu, nous devions dans un premier temps découvrir et jouer à Magic. Bien que nous le connaissions et l'apprécions, presque aucun d'entre nous n'avait fait l'expérience d'y jouer avant le début de ce projet. Celui-ci étant très complet, nous avons eu besoin de quelques temps pour nous y familiariser correctement et pouvoir ensuite nous construire une idée claire de la manière dont nous allions commencer à le modéliser dans le but de le développer.

Une fois qu'une direction dans laquelle nous pourrions avancer s'est dessinée, nous avons décidé de nous organiser en créant des espaces de travail. Notamment un gitlab, nous permettant de mettre en commun l'ensemble des fichiers nécessaire au bon avancement du projet, un trello, nous servant à lister les tâches à accomplir et les personnes chargées de ces tâches, ainsi qu'un tableau blanc, utile à la création de schémas et d'ébauches pratiques dans la mise en place et la compréhension d'idées.

Ensuite, nous devions nous familiariser avec le langage de programmation requis pour la mise en œuvre du projet, à savoir Python. Effectivement, bien que le langage soit assez intuitif et simple à prendre en main grâce à nos connaissances, nous avions besoin de nous acclimater à celui-ci. Notamment pour les parties un peu plus complexes ou différentes des autres langages de programmation.

Suite à ces préparations et à notre mise en place, nous avons décidé de nous réunir tous les jours grâce à discord dans le but d'être régulier dans notre travail.

Objectifs et mise en place

Dans un premier temps, nous avons d'abord réfléchi à la direction dans laquelle nous pourrions nous diriger pour développer l'ensemble du jeu Magic. Après quoi nous nous sommes mis d'accord sur les différents points importants à faire avant même de coder. Nous avions donc besoin d'un diagramme de classe complet afin de savoir les éléments à développer, un chronogramme recensant les différentes étapes du déroulement d'une partie, ainsi qu'une étape de réflexion concernant les échanges entre un client et un serveur en python.

Bien que ces parties aient été avortées par la suite, nous nous sommes aussi penché sur la création d'une base de données et sa connexion à celle-ci en python dans le but d'accéder à un ensemble de cartes ainsi qu'à une interface graphique permettant de rendre visuel une partie de jeu de magic.

Une fois ces premières étapes importantes à la production du projet, d'autres étapes qui nous permettaient de continuer sont apparues. Nous avions en effet maintenant une base solide pour débuter l'implémentation des éléments de codes nécessaires au jeu ainsi qu'une grosse partie nécessitant de nombreuses réflexions en ce qui concerne l'architecture réseau de notre application.

La modélisation des cartes à par la suite été une grand étape puisque nous sommes d'abord parti dans l'idée de créer et d'utiliser une base de données local via python afin d'accéder et d'utiliser les cartes que nous souhaitions. Cette idée à par la suite été laissée de côté pour laisser place à une modélisation de cartes utilisant des fichiers json. Effectivement, cela nous permet d'utiliser une carte déjà implémentée très simplement ainsi que d'en ajouter d'autres en créant de nouveaux fichiers associés.

Ensuite, nous nous sommes dirigés dans les dernières grandes étapes et les plus longues auxquelles nous avions pu penser, notamment le développement du déroulement d'une partie et de ses tours, que ce soit à 1, 2 ou n joueurs, la modélisation,la création et la gestion des effets des cartes de jeu en notre possession, ainsi que la gestion d'un bot possédant plusieurs comportement en fonction de ce que nous souhaitions. Ici le bot a particulièrement des comportements basiques, par exemple un bot permettant de choisir aléatoirement différentes actions ou un bot ne faisant que passer son tour.

Conception Logicielle

Lors de la réalisation de ce projet, nous nous sommes très rapidement rendu compte que la modélisation est une part très importante de celui-ci. Notre premier objectif a donc été de créer une modélisation la plus complète possible et la plus modulaire possible. Effectivement, nous avons estimé qu'une bonne modélisation, à savoir une modélisation très détaillée et permettant un ajout et une modification d'éléments simplement était une idée plus que nécessaire et utile pour notre avancement futur.

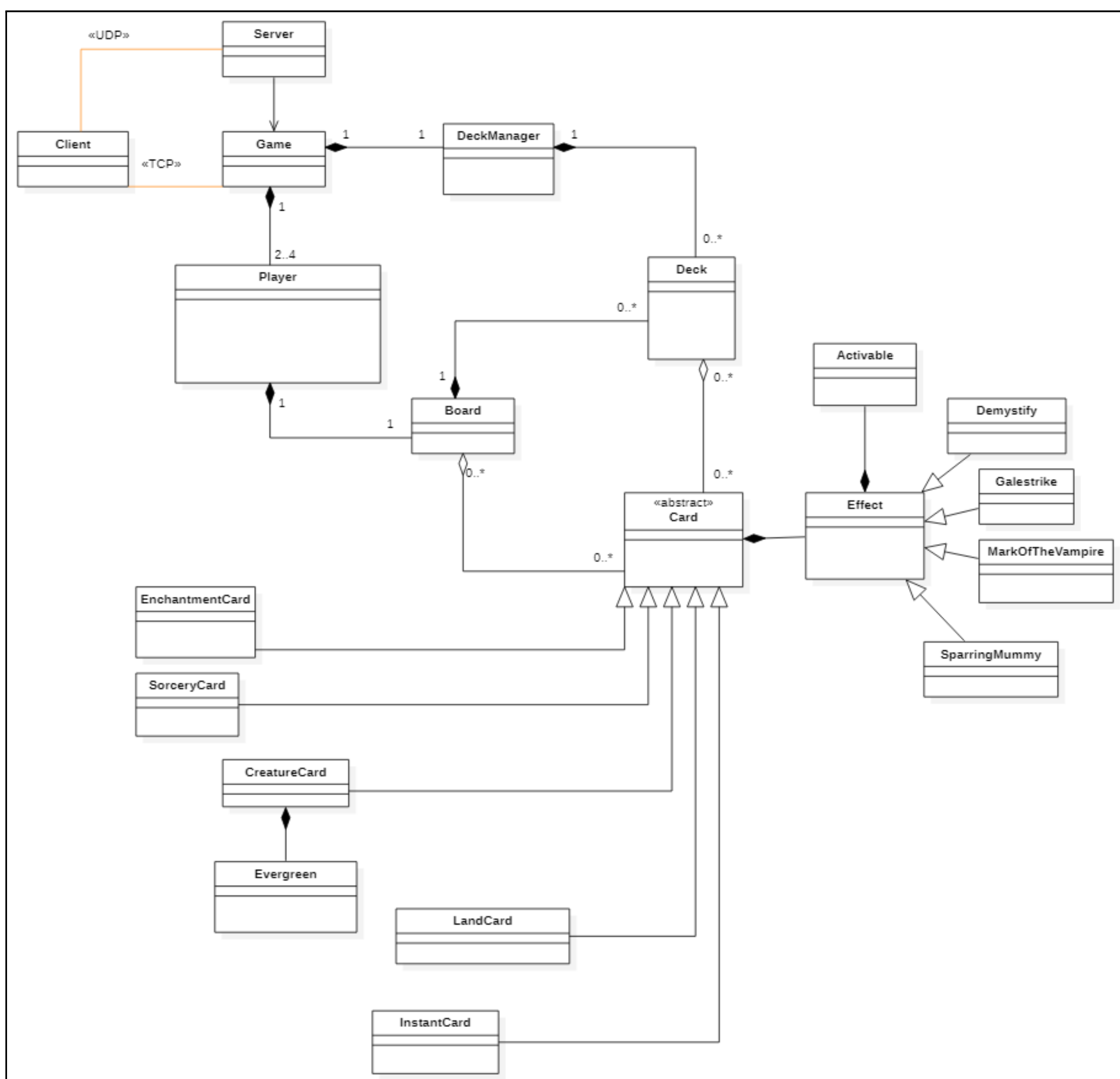
Comme dit précédemment, nous avions donc besoin d'un diagramme de classe, d'un chronogramme ainsi qu'une modélisation concernant les cartes du jeu.

Nous allons donc voir directement ci-après, la partie de la modélisation qui concerne le diagramme de classe.

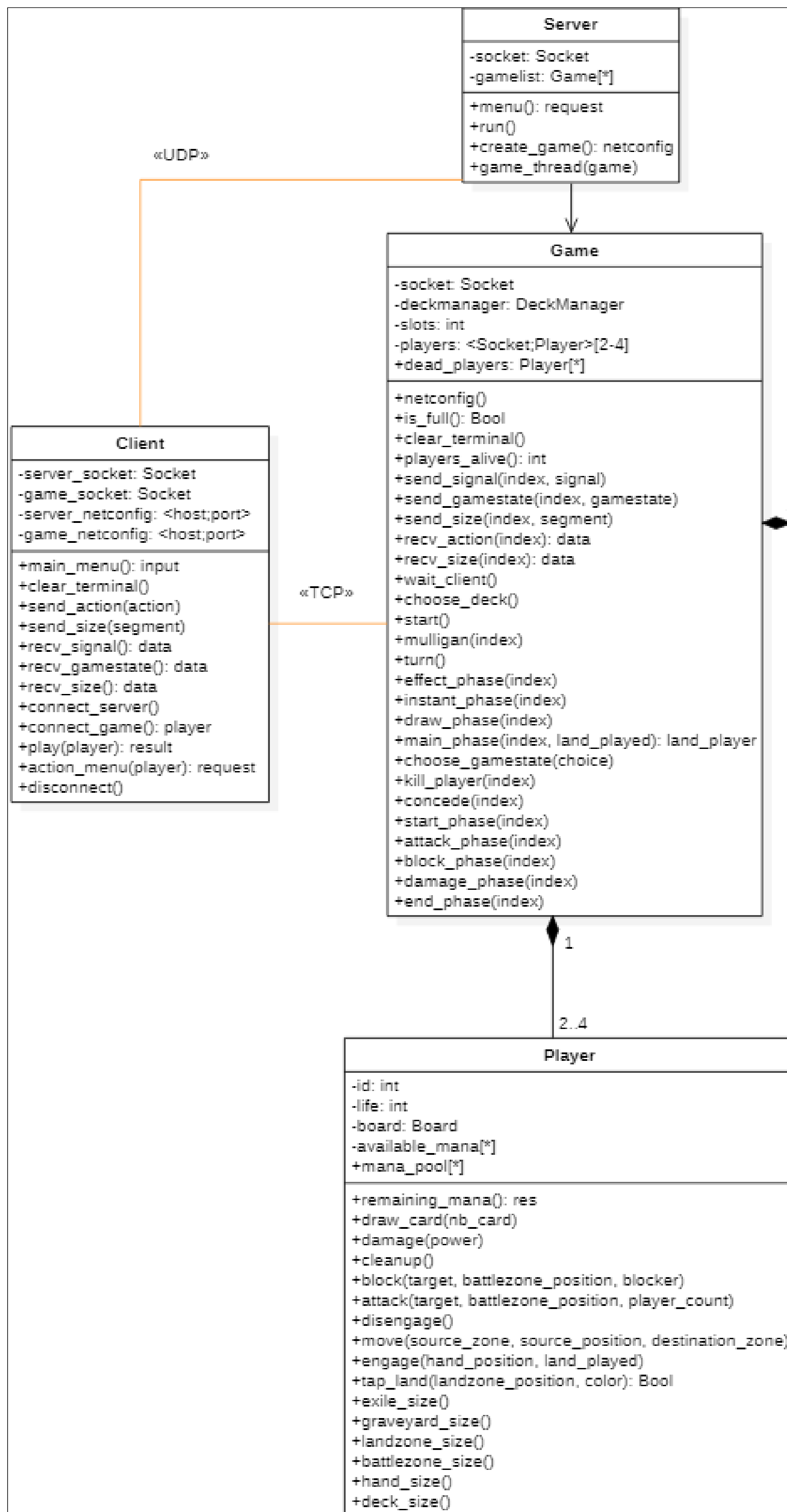
Diagramme de classe

Le diagramme de classe est un diagramme qui nous est très important dans la création du projet. En effet, c'est lui qui a pour rôle de lier l'ensemble de nos classes et méthodes ensembles. C'est grâce à ce diagramme que nous pouvons nous donner une idée globale de comment va être développé par la suite notre projet, ainsi que d'apercevoir les points les plus importants.

Ci-dessous, voici dans un premier temps le diagramme de classe simplifié ne contenant que les classes permettant de l'afficher au complet sur une page.



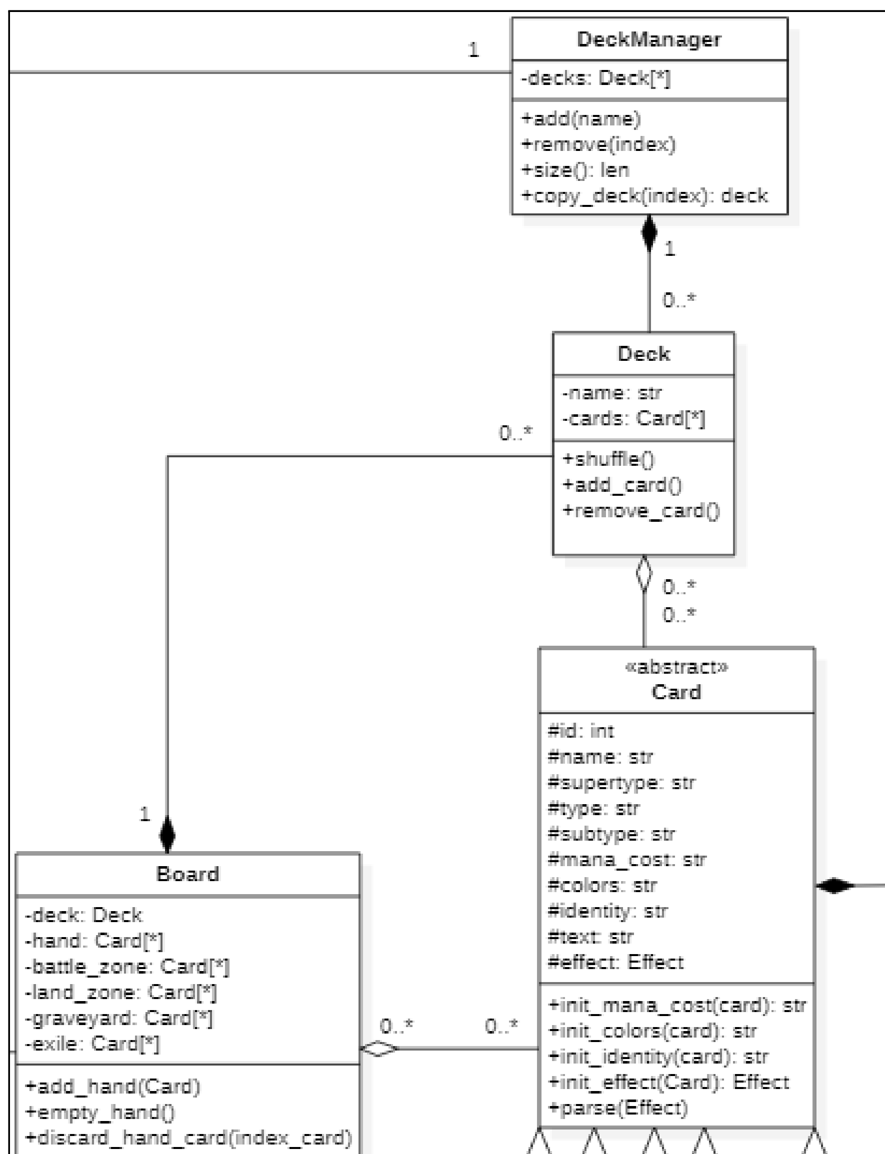
Afin d’expliquer plus en détail de diagramme, nous allons l’expliquer partie par partie à l’aide de notre diagramme de classe complet.



Cette partie est en quelque sorte la partie centrale de notre projet puisque c'est celle qui gère les échanges entre nos clients et le serveur. Quand nous décidons de lancer une partie, nous lançons d'abord un serveur. Son rôle est de recevoir des demandes, sous forme de requêtes, de les interpréter et envoyer une réponse au bon endroit. La partie de notre modélisation qui envoie les requêtes au serveur est le client ainsi que le game. En effet, notre modélisation est faite de sorte à ce que lorsque l'on créer un client connecté au serveur en UDP, le serveur va créer un objet game permettant de lancer une partie et de créer des players associés.

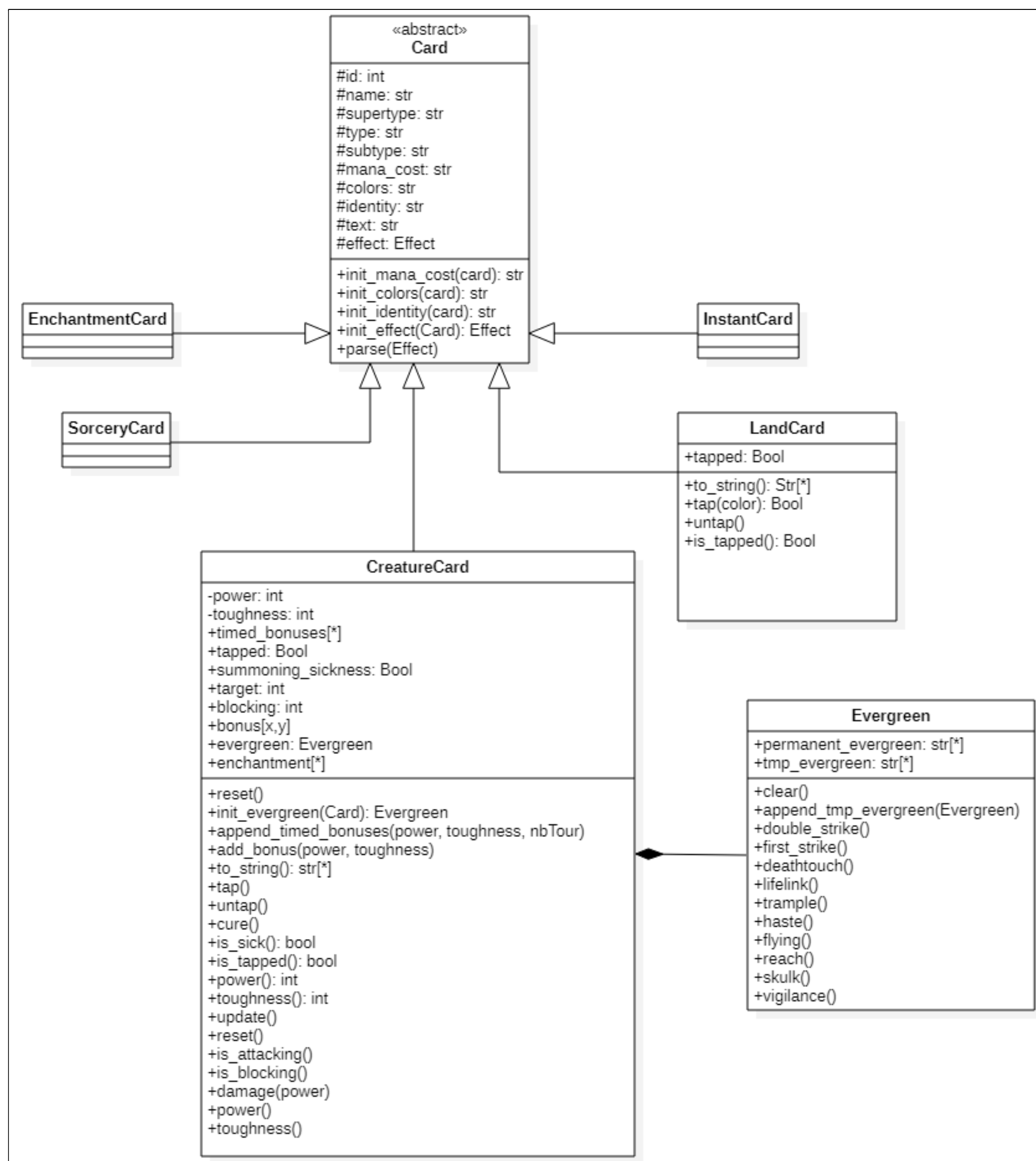
Le client permet ici de faire le lien entre le serveur et le jeu, mais il ne gère pas la partie. Autrement dit, il ne peut faire que des demandes, qui, après traitement par le serveur, peuvent être exécutées ou non.

Passons ensuite à la seconde partie du diagramme. Celle ci-dessous est liée à la première entre la classe `Game` et la classe `DeckManager`, ainsi que la classe `Player` et la classe `Board`.



La classe `Board` est en fait la classe qui permet de donner les éléments nécessaires à un plateau de jeu d'un joueur. Elle est d'ailleurs liée à la classe `Player` par une composition. Ce lien permet ainsi d'expliciter le fait qu'un joueur possède un plateau de jeu, qui possède lui-même d'autres classes mettant en forme les choses dont le jeu a besoin. Il est lié entre autres à la classe `Card`, lui permettant d'obtenir toutes les cartes nécessaires et ainsi pouvoir les utiliser au moment opportun.
Il est en plus de cela lié à la classe `Deck`, tout comme la classe carde l'est aussi. La classe `Deck` est quant à elle liée à une autre classe `DeckManager`. Tout cela nous permet d'avoir une liste de `Deck`, que nous pouvons créer à l'aide de fichiers `JSON`. En effet, si on souhaite créer un nouveau deck avec des cartes choisies afin de pouvoir l'utiliser en partie, cela est possible grâce à cette modélisation. Le `DeckManager` possède donc une liste de `Deck`, qui lui-même possède une liste de `Card`.

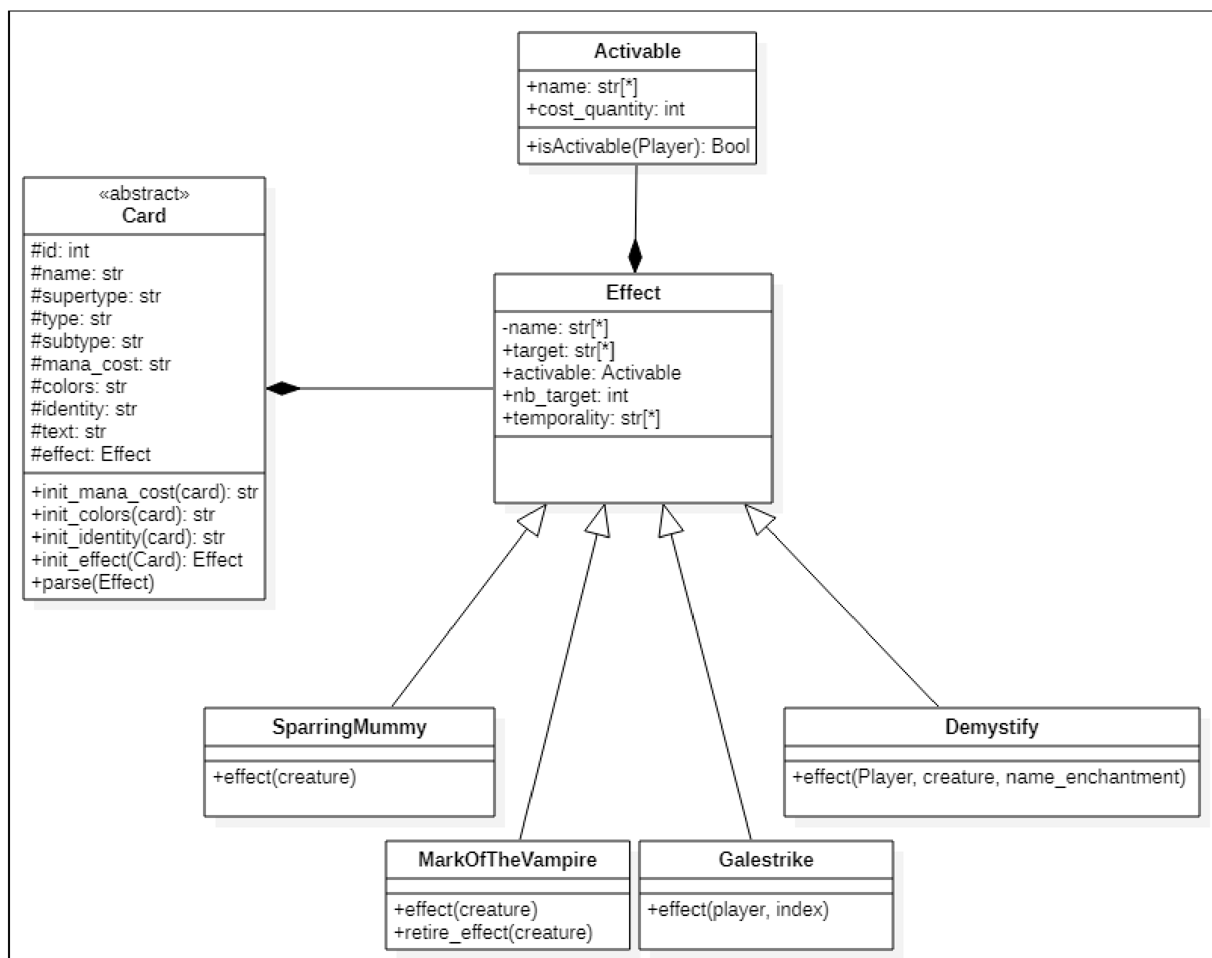
La partie suivante du diagramme correspond plus particulièrement aux différents types de cartes dont nous avons eu besoin tout au long de notre projet.



Cette partie est axée sur les différentes cartes. En effet, cette modélisation nous permet d'avoir une classe `Card` très globale, contenant les attributs utiles à n'importe quelle carte ainsi que des classes filles à la classe `Card`. Ces classes qui héritent de `Card` nous permettent ainsi de spécifier plus en détail d'autres types de cartes. Dans notre modélisation par exemple, la classe `CreatureCard` nous permet de définir des attributs et méthodes supplémentaires par rapport à la classe mère. Cela nous est donc très utile dans l'utilisation de nos cartes lors d'une partie.

La classe `Evergreen` est quant à elle une classe qui compose `CreatureCard`. Cette modélisation est nécessaire dans l'avancement de notre projet puisque c'est ce qui va nous permettre de définir un effet statique (*cf Effets statique*) à nos cartes créatures. Que ce soit un effet vol ou deathtouch par exemple. Ce sont des effets utiles que sur des cartes créatures car ce sont des effets qu'une carte possède la plupart du temps dès le départ.

Pour finir, la dernière partie de l'uml est la partie concernant les effets.



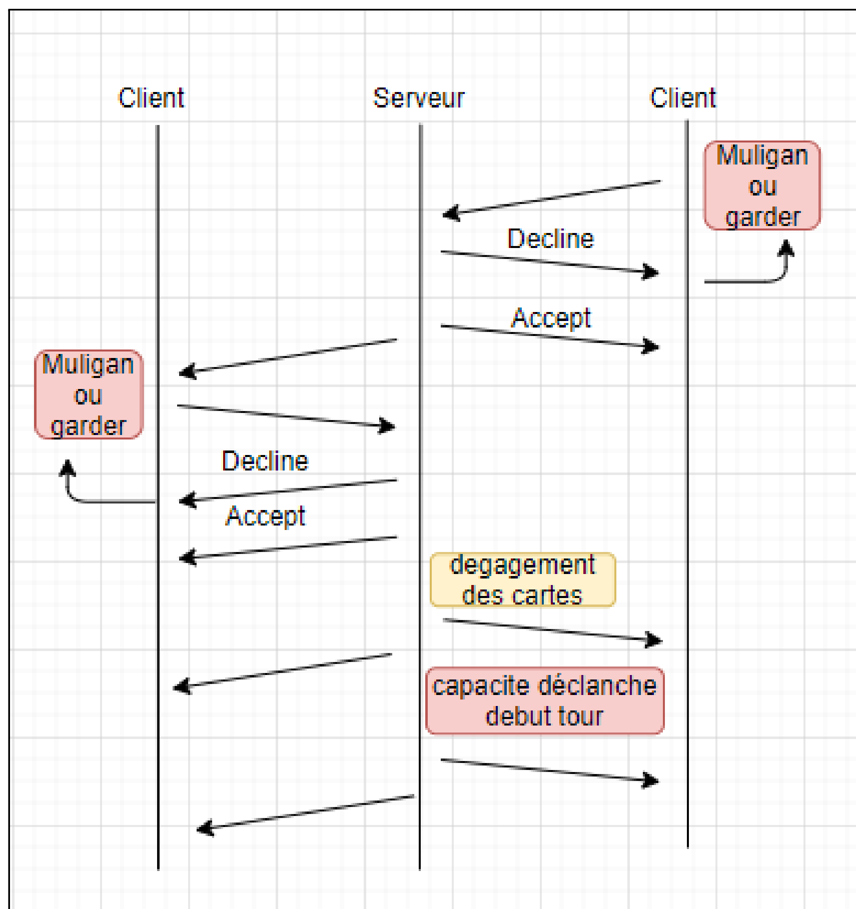
Cette dernière partie modélise les effets des cartes. Nous avons effectivement une classe effet liée à la classe `Card` par une composition et qui a pour rôle de définir ce qu'est un effet. A savoir quels sont ses attributs. Cela nous permet ensuite de créer une classe par effet. En effet, nous étions dans un premier temps partis dans l'idée de faire une classe effet comprenant une à plusieurs méthodes par effets. Cependant lors d'un de nos rendez-vous hebdomadaires, nous avons discuté de cette partie de la modélisation et nous nous sommes rendu compte dans un second temps que réaliser une classe par effet serait une bien meilleure solution. Elle est en effet bien plus modulaire, et plus simple à mettre en œuvre.

Une dernière classe nous est aussi très utile dans la réalisation de nos effets. C'est la classe `Activable`. Celle-ci permet la gestion des effets activables, c'est-à-dire, les effets qui peuvent être utilisés à certains moments de la partie en fonction d'un certain coût notamment un coût en mana par exemple.

Modélisation des échanges

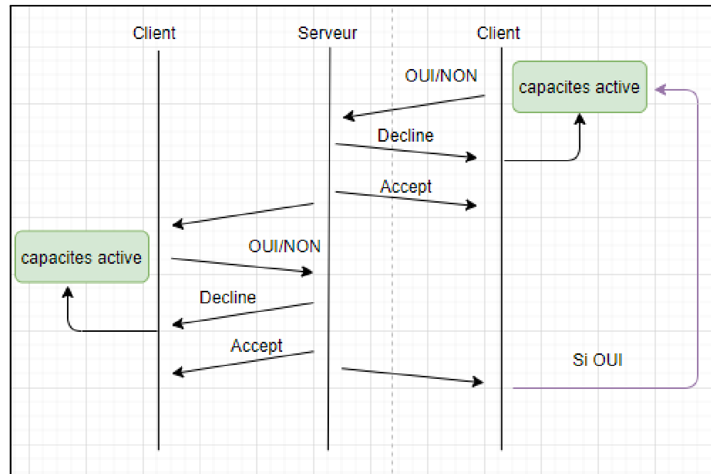
Seuls les échanges liés à un tour de jeu sont représentés ci-dessous, ceux liés à la connexion et déconnexion au serveur sont triviaux.

Phase de départ



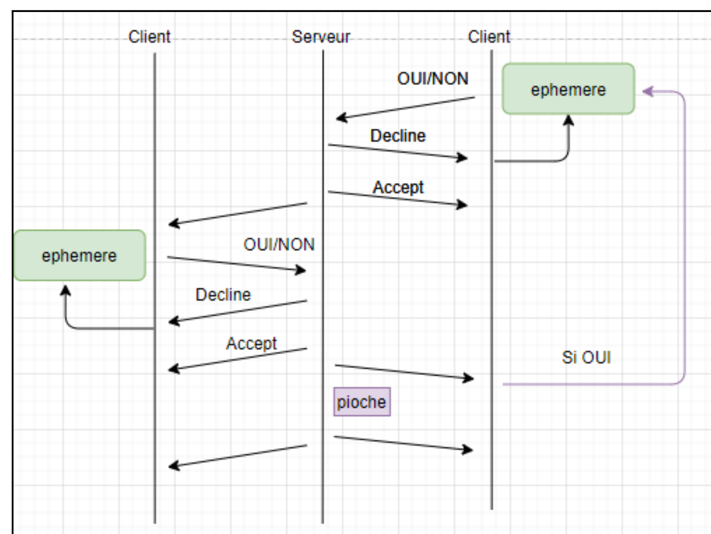
Au début, le client envoie au serveur s'il souhaite mulligan ou s' il souhaite garder son jeu. Le serveur lui répond alors par un `DECLINE` ou un `ACCEPT`. Si celui-ci répond en renvoyant une requête `DECLINE`, le client va devoir renvoyer l'action. Si, à contrario, le serveur renvoie une requête positive au client, alors il envoie au client suivant une requête `PLAY` définissant le fait qu'il peut à son tour mulligan ou garder sa main. Une fois que les deux joueurs ont fait leurs actions, le serveur dégage les cartes du joueur suivant puis envoie l'état au client. Enfin, il déclenche les capacités du joueur suivant au début de son tour.

Phase d’activation de capacités



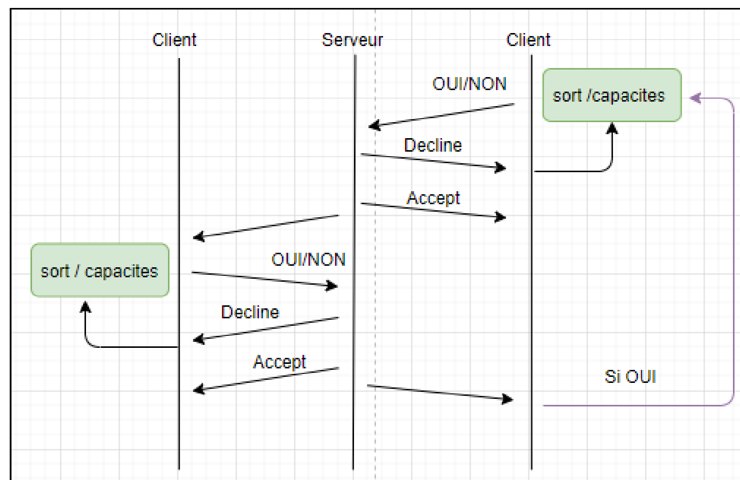
Le client a ici la possibilité d’utiliser ou non les capacités actives. Il envoie au serveur ce qu'il fait et le serveur lui envoie ensuite une requête `DECLINE` ou `ACCEPT` selon la validité de la requête de demande.

Phase d’éphémères



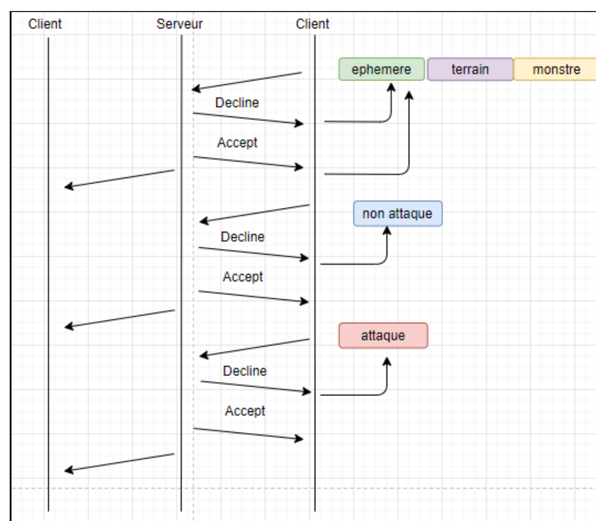
Le deuxième client envoie s'il veut jouer un éphémère ou non. Le serveur lui répond selon si la requête est bonne ou mauvaise. Ensuite, c'est au tour du joueur 1 d’envoyer un éphémère puis ça revient au joueur 2 jusqu'à temps que les 2 joueur ne veulent plus ou ne peuvent plus jouer d'éphémère. Enfin, c’est au tour du second joueur de piocher.

Phase des sorts



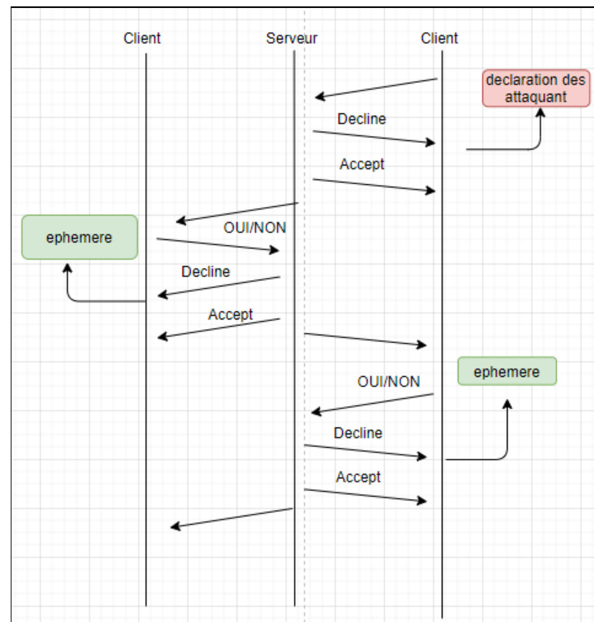
Le deuxième client envoie s'il souhaite jouer ou non un sort ou une capacité. Le serveur lui répond de la même façon que précédemment selon si la requête est bonne ou mauvaise. Ensuite c'est au tour du joueur 1 d'envoyer un éphémère avant de revenir au joueur 2 jusqu'à temps que les deux joueurs ne veulent plus ou ne peuvent plus envoyer de sort ou de capacités.

Phase principale



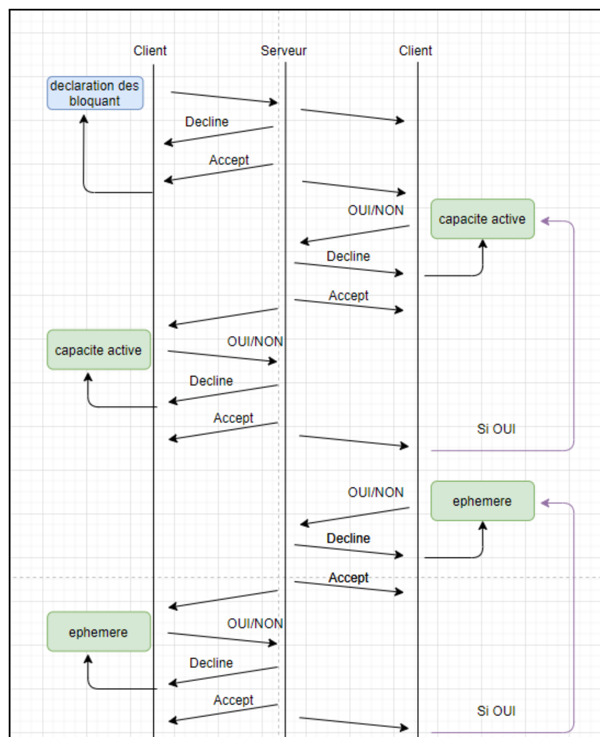
Le client 2 peut jouer des éphémères, terrains et monstres. Le serveur envoie toujours une réponse selon la requête. Si c'est un `DECLINE`, le client peut refaire l'action et si la requête est acceptée le joueur peut jouer une autre ephemere, créature. Ensuite il choisit s'il attaque ou non. Sinon, la phase d'attaque est passée.

Phase d’attaque



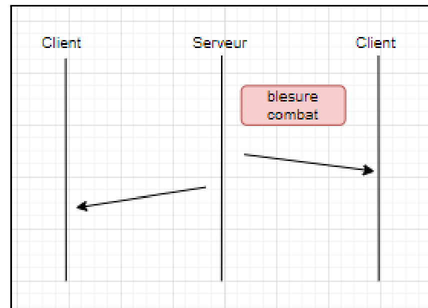
Si une attaque a lieu, le joueur 2 déclare les attaquants. Une phase d’éphémères est ensuite lancée.

Phase de blocage



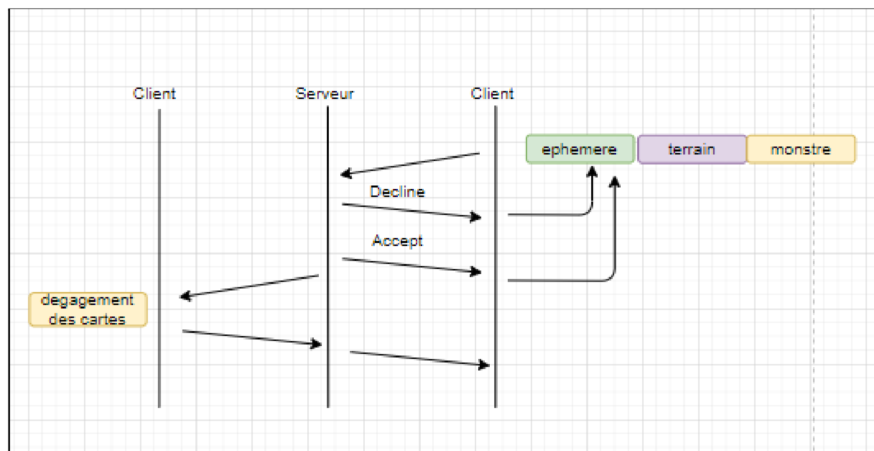
Après la déclaration des attaques, le joueur adverse déclare les créatures bloquantes. Vient ensuite la phase des capacités active ainsi que la phase des éphémères.

Phase de résolution des dégâts



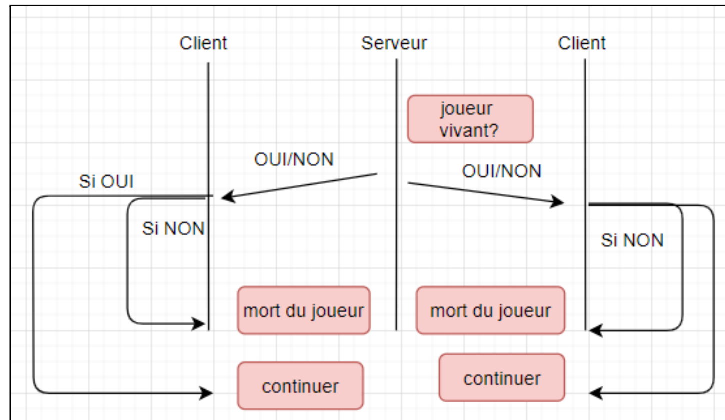
Une fois que la phase de combat est terminée, on attribue les dégâts aux créatures. Si les créatures ont une vie inférieure à 0 alors elle meurt.

Phase principale 2



La phase principale 2 permet de poser des créatures, des éphémères et des terrains. Puis on dégage les cartes du joueur 1.

Phase de fin



Le serveur vérifie si le joueur est vivant. Si le joueur est mort, alors la partie s’arrête. La partie continue tant qu'au moins deux joueurs sont encore en vie.

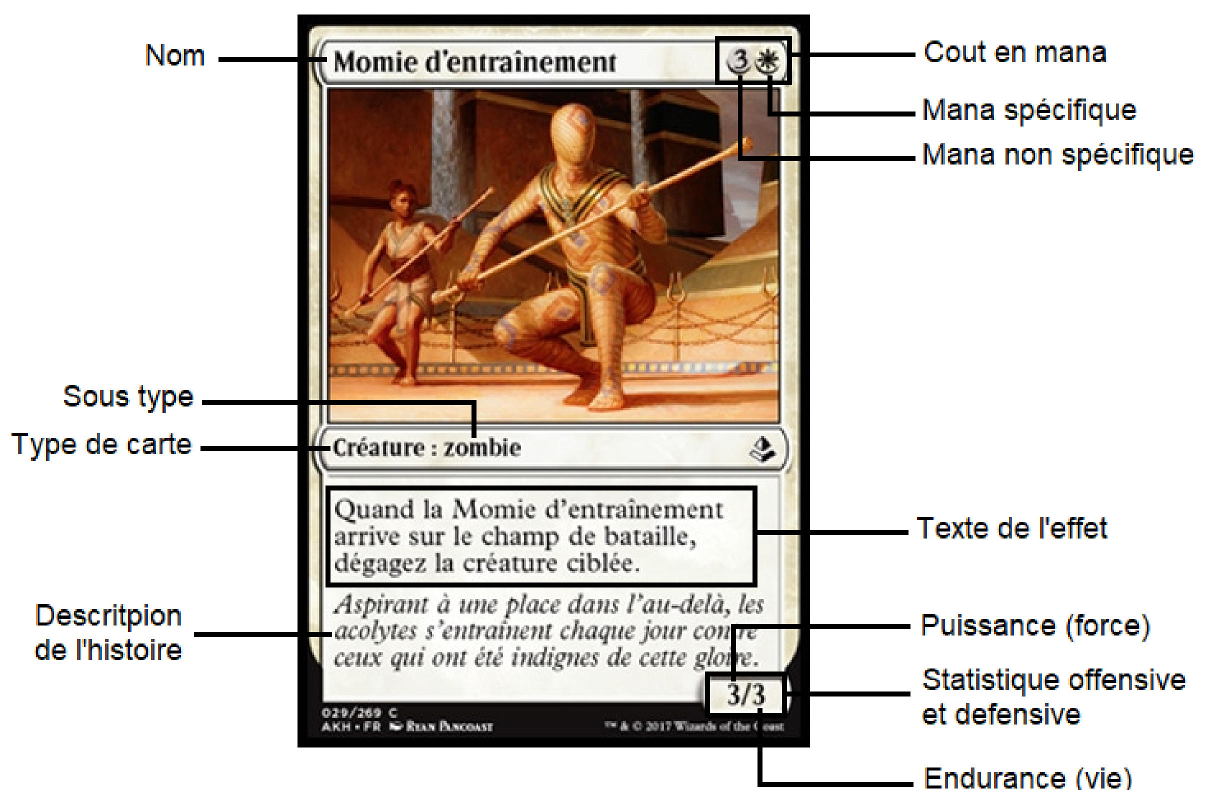
Représentation des données

Cartes

Dans le cadre de la modélisation des cartes, nous avons tout d'abord recensé tous les types de cartes que le jeu magic possède pour en faire une liste exhaustive pour lister les types de cartes utilisables dans le cadre d'une version simple du jeu magic. Dans cette démarche nous avons donc mis de côté les cartes plus compliquées comme les cartes planeswalker ou encore les cartes tribales. Nous avons donc pris le parti de partir simplement sur les cartes de type suivant :

- Terrain
- Créature
- Rituel
- Éphémère
- Enchantement

Ces types ont été choisis arbitrairement, étant les types de cartes les plus classiques qui apparaissent dans chaque partie de magic quelque soit le type de deck choisi en amont par le joueur. Dans un souci de simplicité nous nous sommes donc cantonnés à ces 5 types de cartes la dans l'optique d'avoir une partie simplifiée qui fonctionnera. Il n'est cependant pas exclu de pouvoir ajouter les autres types de cartes à l'avenir dans notre modélisation.



Certains éléments de la carte sont récurrents malgré que les types soient foncièrement différents. Nous pouvons par exemple cité :

- Un nom
- Un super-type
- Un type
- Un sous-type
- Un coût en mana
- Une couleur
- Une description
- Un texte de capacité

Certaines cartes cependant possèdent des champs spécifiques c'est le cas des cartes créature qui en plus d'avoir les éléments précédemment listés, possèdent également une puissance ainsi qu'une endurance. Ces deux attributs entrent en compte lors des phases d'attaque de carte et de défense de carte qui seront développés dans une partie ultérieure.

Dans une optique d'avoir des cartes différentiable nous avons également ajouté un identifiant ayant certaines cartes possédant le même nom de carte mais avec des effets différents et des statistiques différentes. Nous avons pris en temps que identifiant, l'identifiant dans la base de données originellement utilisé pour créer les cartes.

Cette base de données qui était fourni par Mr. Rabat nous a permis de retrouver les valeurs de chaque attribut des cartes assez rapidement. Elle fut cependant remplacée par un système de cartes sous format JSON plus portable et moins contraignant. Cette base de données nous aura néanmoins bien servi à retrouver les champs dans un format utilisable et stockable dans un fichier JSON.

Dans un premier temps nous avons envisagé de stocker toutes les cartes dans un seul fichier JSON pour une accessibilité simplifiée, ayant accès à toutes les cartes à notre disposition un seul appel. Suite à des discussions en Rendez-vous sur cette façon de procéder, nous avons donc entrepris un nouveau moyen de stocker tout cela en utilisant un fichier JSON par cartes pour économiser des ressources. Lors d'un entretien nous avons débattu de l'utilisation d'un tel JSON qui serait sans cesse ouvert dans son intégralité tout cela dans le but de retrouver une carte spécifique. Ouvrir un JSON de quelques lignes semblait donc plus approprié que toujours ouvrir un seul fichier de plusieurs centaines de lignes.

Dans ce fichier JSON est donc recensé toutes les informations liées à la carte notamment les plus simples tel que l'id qui lui sert de nom de fichier, le nom de la carte, le texte brut qui lui est associé ainsi que son type, son super-type et son sous-type. Également deux champs importants par cartes, celui des effets qui peut être sur tout type de carte, et celui des evergreen qui quant à lui, n'est trouvable que dans des cartes de type créature. Ces deux champs seront plus amplement développés ultérieurement dans la partie effet.
Voici deux exemples de format de JSON utilisé :

Format JSON d'une carte de type créature

```
"Id": int,
"Name": String,
"Colors": String,
"Mana Cost": String,
"Identity": String,
"Text": String,
"Evergreen": [String, String],
"Effect":[
        {
            "Name": String,
            "Target": String,
            "Nb_target": int,
            "Temporary": int,
            "Activable":{
                       "Cost_name": [String, String]
                       "Cost_quantity": [String, int]
            },
            "Temporality": String
        }
],
"Power": int,
"Toughness": int,
"Type": String,
"Subtype": String,
"Supertype": String
```

Format JSON d'une carte de type land

```
"Id": int,
"Name": String,
"Colors": String,
"Mana Cost": String,
"Identity": String,
"Text": String,
"Effect":[],
"Type": String,
"Subtype": String,
"Supertype": String
```

De plus chaque JSON comporte trois autres champs ayant en point commun la notion de couleur. Dans l'univers de Magic chaque carte et chaque deck est caractérisé par une couleur, il en existe 6 distinctes :

- W ⇒ White ⇒ Blanc
- B ⇒ Black ⇒ Noir
- R ⇒ Red ⇒ Rouge
- G ⇒ Green ⇒ Vert
- U ⇒ blUe ⇒ Bleu
- C ⇒ Colorless ⇒ Sans couleur

Ces couleurs sont utiles dans le cas de l'identité et de la couleur d'une carte, cela permet de la classer dans des decks spécifiques. Cela rentre notamment en compte lors du calcul du coût en mana de la carte, comme vu précédemment une carte est pourvu d'une quantité de mana représentant le nombre nécessaire de mana d'une couleur pour pouvoir jouer la carte. Cette quantité peut prendre plusieurs formes, sa plus basique étant représentée par les couleurs W, B, R, G, U. Viens ensuite les sans couleurs et les hybrides, les hybrides sont des quantités à choix, par exemple "W/R" qui désigne soit un mana de couleur blanc soit un mana de couleur rouge, ou encore "2/P" qui lui désigne soit deux mana sans couleur, soit deux point de vies. Il peut également arriver que la quantité demandée soit représentée avec un "X", cela veut simplement dire que l'on peut utiliser n cartes de la couleur que l'on veut pour l'invoquer, le plus souvent cela influe sur un potentiel effet qui accompagne la carte.

Effets

Les effets dans Magic sont catégorisés en 3 parties différentes, il y a tout d'abord les capacités déclenchées qui s'activent d'elles même lorsque la carte est jouée ou qu'une condition est rempli pour son activation, dans cette catégorie ce sont les cartes éphémères et rituels qui sont les plus représentées. Les effets activables qui eux ne peuvent être activé que lors d'une phase d'action du joueur, les plus représentés dans cette catégorie sont les cartes de type créature et les cartes de type terrain. La dernière variété d'effet sont les effets statiques, aussi appelé "Evergreen" dans la communauté anglophone étant des mots clé que l'on retrouve à travers les extensions du jeu Magic.

Effets déclenché

Ces effets déclenchées comme expliqué dans l'introduction ci-dessus sont des effets qui s'activent lors de l'invocation de la carte ou lorsqu'une condition est remplie. Dans ce cas la on répertorie dans le JSON d'une carte possédant une telle capacité plusieurs champs pour pouvoir l'activer sans encombre.

JSON d'une carte de type créature possédant une capacité déclenchée

```
...
"Text": String,
"Evergreen": [String, String],
"Effect":[
        {
            "Name": String,
            "Target": String,
            "Nb_target": int,
            "Temporary": int,
            "Temporality": String
        }
],
"Power": int,
"Toughness": int,
...
```

Le nom de l'effet est répertorié dans le champ `Name`, il n'est pas unique, contrairement aux cartes on peut retrouver un effets sur différente cartes en même temps, prenons un exemple simple, le cas d'un effet qui nous fait piocher une carte, celui ci peut se retrouver sur plusieurs cartes en compagnie d'autres effets. On peut notifier que plusieurs effets peuvent être regroupés en surclasse ayant un comportement similaire, on peut par exemple citer les effets de pioche, les effet de création de token, ceux qui modifie les stats offensive et défensive des cartes créature ou qui annule simplement les modification ou l'invocation de carte.

Chaque effet non statique possède une cible, ici stocker dans en valeur de la clé `Target`, les cible sont varié allant d'un joueur à une zone en passant par des cartes de tous types. En voici une liste non exhaustive mais très représentative des différentes cibles possibles avec une classification du dénominateur commun au plus spécifique :

- Créature
    - creature tap (cible une créature engagé)
    - creature attack (cible une créature en phase d'attaque)
    - creature block (cible une créature en phase de blocage)
    - creature self (cible une créature qui appel l'effet sur elle même)
    - creature (cible une créature dans son ensemble)

- Terrain
    - land tap (cible un terrain engagé)
    - land (cible un terrain dans son ensemble)

- Joueur
    - enemy player (cible un joueur ennemi)
    - ally player (cible un joueur allié ou soit meme)
    - player (cible un joueur qu'importe qu'il soit allié ou ennemi)

- Effet
    - effect spell (cible une capacité d'une carte éphémère ou rituel)
    - effect creature (cible une capacité venant d'une carte créature)
    - effect (cible une capacité quel que soit sa provenance)

- Zone
    - area hand (cible la main d'un joueur)
    - area battle zone (cible le champ de bataille d'un joueur)
    - area land zone (cible la zone de terrain d'un joueur)
    - area deck (cible le deck d'un joueur)
    - area graveyard (cible le cimetière d'un joueur)
    - area exile (cible la défausse d'un joueur)

- Enchantement
    - enchant card (cible une carte enchantement lié à une carte créature)
    - enchant player (cible une carte enchantement lié à un joueur)
    - enchant (cible une carte enchantement quel que soit son lien)

- Artéfact
    - artifact (cible un artéfact)

Le champ qui répertorie le nombre de cible touché par l'effet est quant à lui stocker dans `Nb_target`. Celui-ci permet de pouvoir lancer l'effet le nombre de fois stocker dans `Nb_target` en redemandant la cible a chaque activation de l'effet dans le cas ou l'on peut choisir deux cibles différentes entre les deux activations. Cela permet également de gérer les erreurs et les exceptions que l'on pourrait avoir à relancer la même capacité sur une cible ne pouvant plus être atteinte, ou si les cibles potentielles sont détruites entre l'activation d'effet subsidiaire.

La notion de d'effet temporaire se place dans le champ `Temporary` originellement stocké dans un booléen pour gérer les cas ou un effet n'est actif qu'un seul tour. Il est dorénavant stocké dans un entier pour la gestion d'effet à tour multiple, il y est donc entré le nombre de tour d'activation de la capacité.

Le dernière notion abordé dans cette partie est le moment ou l'effet est déclenché appelé `Temporality` il est constitué de chaînes de caractères 5 possibles :

- play (L'effet se déclenche lors de l'invocation de la carte)
- die (L'effet se déclenche lors de la mort de la créature)
- attack (L'effet se déclenche lors de la phase d'attaque)
- block (L'effet se déclenche lors de la phase de blocage)

La chaîne de caractères également trouvable est “activable” qui permet plus rapidement de savoir si une carte de type créature possède un effet activable nous allons décrire cela dans la partie suivante.

Effets activable

Les effets activables sont des capacités qui sont le plus souvent associées à des cartes de type créature mais peuvent tout autant apparaître sur des cartes de type terrain. Il se démarque d'un effet déclenché par son champ Activable

JSON d'une carte de type créature possédant une capacité activable

```
...
"Text":"{2}, {T}: Tap target creature.",
"Effect":[{
            "Name": String,
            "Target": String,
            "Nb_target": int,
            "Temporary": int,
            "Activable":{
                       "Cost_name": [String, String]
                       "Cost_quantity": [String, int]
            },
            "Temporality": String
}],
...
```

L’item `Activable` comporte deux parties, un champ `Cost_name` et un champ `Cost_quantity`, ils ont tous les deux des valeurs stockées dans un dictionnaire python la clé étant `Cost_name` et la valeur associée `Cost_quantity`. Les différents nom que l’on peut retrouver dans les clé du dictionnaire sont :

- mana (l'activation requiert le mana du joueur)
- tap (l'activation requiert l’engagement d’une carte)
- discard (l’activation requiert la défausse d’une carte)
- sacrifice (l’activation requiert le sacrifice d’une carte)
- life (l’activation requiert le sacrifice de point de vie)

Outre le champ mana qui est associé à une valeur en champ de caractère gérer différemment, le reste est associé à une valeur entière représentant l’unité nécessaire pour réaliser l’action.

Effets statique ou Evergreen

Dans notre modélisation de carte les effets statique aussi appelé en anglais *keyword Evergreen* ou plus simplement evergreen sont à part des autres effets déclenchés et activables forcément associé à une carte de type créature. Nous en avons recensé 17 différents dans notre modélisation :

- Deathtouch|Contact Mortel :
  Envoi une Créature au cimetière qu’importe l’endurance de la créature

- Defender|Défenseur :
  Ne peut pas attaquer

- Double strike|Double initiative :
  Attaque deux fois, une fois en même temps que les Créatures First strike et une fois avec toute les autres Créatures

- First strike|Initiative :
  Ajoute une phase d'attaque avant celle des autre autrement dit, attaque avant les autres créature qui n'ont pas first strike

- Flash|Flash :
  Peut être jouée à tout moment où un éphémère pourrait être joué

- Flying|Vol :
  Une créature Flying ne peut être bloquée qu'avec des créatures Flying ou Reach

- Haste|Célérité :
  Ne possède pas de mal d'invocation

- Hexproof|Défense talismanique :
  Ne peut pas être la cible de sort par des adversaires

- Indestructible|Indestructible :
  Ne peut être détruit

- Lifelink|Lien de vie :
  Octroie un montant de point vie égal au dégâts infliger

- Menace|Menace :
  Ne peut être bloquer que par 2 créature ou plus

- Protection|Protection :
  Un permanent avec la protection contre une qualité (le plus souvent une couleur) ne peut pas être ciblé par des sorts ou capacités ayant cette qualité, subir de blessures de sources ayant cette qualité, être enchanté ou équipé par des auras/équipements qui ont cette qualité. Si le permanent est une créature, il ne peut pas être bloqué par des créatures ayant cette qualité.

- Reach|Portée :
  Peut attaquer une créature avec Vol

- Trample|Piétinement :
  Le surplu de dégâts est infligé au joueur directement

- Vigilance|Vigilance :
  Attaquer ne fait pas s'engager une Créature avec Vigilance

- Prowess|Prouesse :
  À chaque fois que le joueur lance un sort, une créature avec Prowess gagne +1/+1 jusqu'à la fin du tour.

- Skulk|Furtivité :
  Une carte avec Skulk ne peut être bloquée que par une carte avec une puissance supérieure à la sien

```
...
"Text": String,
"Evergreen": [String, String],
"Effect":[
...
```

Ces effets statiques sont stockés dans `Evergreen` sous forme de chaînes de caractères, leur spécificité leur fait avoir une classe à part des autres effets. Leur code est principalement écrit mais non implémenté dans les tours de jeu.

Implémentation

L’application a été implémentée en Python3 dans un environnement macOS, linux et Windows afin de permettre une exploitation tout environnement généraliste.

Elle inclut les communications TCP avec le serveur principal et en UDP avec les threads pour chaque serveur de jeu. Lors du déroulement d’une partie, les phases liées aux éphémères sont absentes mais le reste de la structure d’un tour a été implémenté au complet. A l’égard des effets, seul l’engagement de cartes sort ou d’effet à la pose est fonctionnel tandis que l’activation sur le terrain et la pile de sorts ne sont pas gérés.

Concrètement notre application permet de lancer un serveur, puis de recevoir des connexions de différents clients venant soit créer soit rejoindre une partie. Les communications présentent une gestion d’erreur et dès le choix du client opéré, il lui est bien permis et fonctionnel de se connecter au serveur TCP et recevoir ainsi les données binaires permettant de lancer le jeu avec un comportement humain ou IA.

Par rapport à l’utilisation d’IA, nous avons deux comportements disponibles :

- Random : dans le menu d’action du client, les choix sont opérés aléatoirement sans possibilité d’afficher la partie ou d’abandonner.
- Skip : aucune action concrète est envoyée, seulement le passage de phase

La partie s’arrête dès qu’il ne reste plus qu’un seul joueur avec des points de vie.

Chaque étape de la partie est suivie par l’impression d’un log dans le terminal du serveur et dans son fichier spécifique

Les sources à la racine du dossier se découpent en plusieurs fichiers :

```
/* activable.py
/* board.py
/* card.py
/* client.py
/* creaturecard.py
/* deck.py
/* deckmanager.py
/* effect.py
/* evergreen.py
/* game.py
/* instantcard.py
/* landcard.py
/* log.py
/* player.py
/* server.py
/* sorcerycard.py
```

- activable.py : gère l'activation des effets
- board.py : représente toutes les zones de jeu
- card.py : représente une carte
- client.py : gère le module client, il est exécutable
- creaturecard.py, instantcard.py, landcard.py, sorcerycard.py : représente les différences entre les types possibles de carte
- deck.py : représente un deck
- deckmanager.py : gère la création, le stockage et la récupération de decks
- effect.py : gère les effets étendus en lien avec les fichiers dans /effet
- evergreen.py : représente les effets de base de carte
- game.py : gère le module serveur TCP, appelé comme un thread
- log.py : gère la création de logs de jeu
- player.py : gère un joueur
- server.py : gère le module serveur UDP, il est exécutable

L'arborescence des fichiers est représentée telle que :

- effet : contient tous les fichiers de classe liés aux effets pour chaque carte
- JSON : contient les représentation des données
- logs : contient les logs de jeu

Mise en service

Notre application gère le fait que nous pouvons la lancer dans différents environnements, à savoir UNIX, macOS, ou bien Windows.

Afin de lancer l'application, il faut tout d'abord ouvrir 3 terminaux dans le fichier code du projet. Sur le premier, il faut exécuter le fichier serveur.py. Dans les autres, il faut exécuter le fichier client.py. Une fois le fichier exécuté, on peut choisir de créer ou de rejoindre une partie. Une fois cela fait, on devra préciser si l'on souhaite jouer soi-même ou déléguer cela à une IA.

Dans le cadre d'un comportement automatique, 2 comportements sont à ce jour sélectionnables, le premier permettant de jouer de façon aléatoire, tandis que le second consiste uniquement a passer son tour. Si le joueur est défini comme humain, alors tous les inputs doivent être saisis dans le terminal à l'aide de menus.
Il est possible de lancer en parallèle de nombreuses parties, et les différentes actions effectuées sont stockées dans un fichier de log qui permet de revoir le déroulement de la partie.

En tant qu'humain, c'est l'entrée correspondante à l'action voulue (précisé entre parenthèse) qui devra être tapée suivi de la touche ENTER :

```
Joueur 0 (10)
[  SHOW_GAME  (S)  ]
[  MULLIGAN   (0)  ]
[  DRAW_CARD  (1)  ]
[  TAP_LAND   (2)  ]
[  PLAY_CARD  (3)  ]
[  USE_EFFECT (4)  ]
[  SELECT     (5)  ]
[  ATTACK     (6)  ]
[  BLOCK      (7)  ]
[  SKIP_PHASE (8)  ]
[  CONCEDE    (9)  ]
>
```

Sachant que l'action S (SHOW_GAME) permet d'afficher le terrain de jeu (avant la défausse) :

```
Player 0 - 7.0 HP - Deck : 39
{'X': 0, 'C': 0, 'W': 0, 'R': 0, 'G': 0, 'U': 0, 'B': 0}
[Plains] [Mighty Leap] [Ritual of Rejuvenation] [Plains] [Raptor Companion] [Plains] [Sparring Mummy]
 [Ritual of Rejuvenation] [Skyblade of the Legion] [Plains] [Plains] [Plains] [Demystify] [Plains] [P
lains] [Plains] [Plains] [Plains] [Plains] [Plains] [Shining Aerosaur]


Player 1 - 10 HP - Deck : 60
{'X': 0, 'C': 0, 'W': 0, 'R': 0, 'G': 0, 'U': 0, 'B': 0}


[(T)Raptor Companion] [Skyblade of the Legion]
```

Un fichier de log est créé à chaque lancement de partie.

Notre application permet aussi de jouer avec le nombre de joueurs que nous voulons. En effet, l'ensemble de notre modélisation est faite de sorte à ce que nous puissions lancer de 0 à n joueurs. Pour modifier cela il suffit de modifier la variable du nombre de joueurs dans le constructeur de la classe `Game`.

Points à améliorer

Dans l'ensemble, nous avons réalisé la plupart des éléments que nous souhaitions implémenter. Cependant il reste des points à améliorer, ou des points que nous aurions voulu faire si nous avions eu davantage de temps.

La plus grande partie que nous pouvons perfectionner est celle qui concerne les effets. En effet, cette partie a été une des parties les plus difficiles puisqu'il existe un nombre conséquent d'effets différents, et même les plus simples d'entre eux ont une modélisation et une implémentation très différentes.

En ce qui concerne la partie du bot, nous aurions pu créer davantage de comportements différents ainsi que le séparer dans un fichier différent afin que d'autres bots puissent être développés plus facilement et sans modifier le code source.

De plus, bien que nous ayons réussi à être assidu et à nous rejoindre tous les jours par le biais de discord, il aurait néanmoins été plus simple et très intéressant de travailler en présentiel.

Suite du projet

Il sera aussi possible d'étendre les comportements d'automatisation en permettant d'importer des fichiers en JSON définissant des règles spécifiques à suivre lors de l'exécution, actuellement inscrit en dur dans la fonction input() de game.py. L'utilisation de vraie IA pourra être ajoutée via l'intégration d'une API permettant à des applications externes de se connecter au jeu.

Nous avons au début du projet pensé à créer une interface graphique, notamment en pygame. Nous pouvons donc imaginer qu'une suite de ce projet pourrait donc être de créer et d'utiliser une interface graphique.

Pour finir, on peut imaginer réaliser notre projet avec une modélisation permettant un mode multijoueur en ligne. En effet, notre projet actuel fonctionne sur la même machine.

Conclusion

Ce projet nous a permis d’approfondir davantage nos connaissances sur le langage de programmation python ainsi que sur les perspectives qu’il offre. Cela nous a aussi permis de créer et implémenter une modélisation très complète et orientée objet. Nous avons ainsi dû gérer une grosse partie de connexions entre plusieurs clients et serveurs.

Ce projet a également été une très bonne expérience nous rapprochant au plus possible d’une expérience professionnelle, avec un cahier des charges concret. De plus, ce fut une très bonne expérience de travail en groupe, ce qui nous a permis de gérer et nous intéresser à la façon dont nous allions répartir les tâches, notamment en tenant compte des points forts et faiblesses de chacun d’entre nous.