PROJET TUTEURÉ

Rapport final

Décembre 2020

Ecrit par: Saad El Din AHMED, Yessine BEN EL BEY, Salim DIB, Youssef HACHIMI, Hugo NORTIER

Tuteurs projet : Audrey OCCELLO, Philippe RENEVIER GONIN

Sommaire

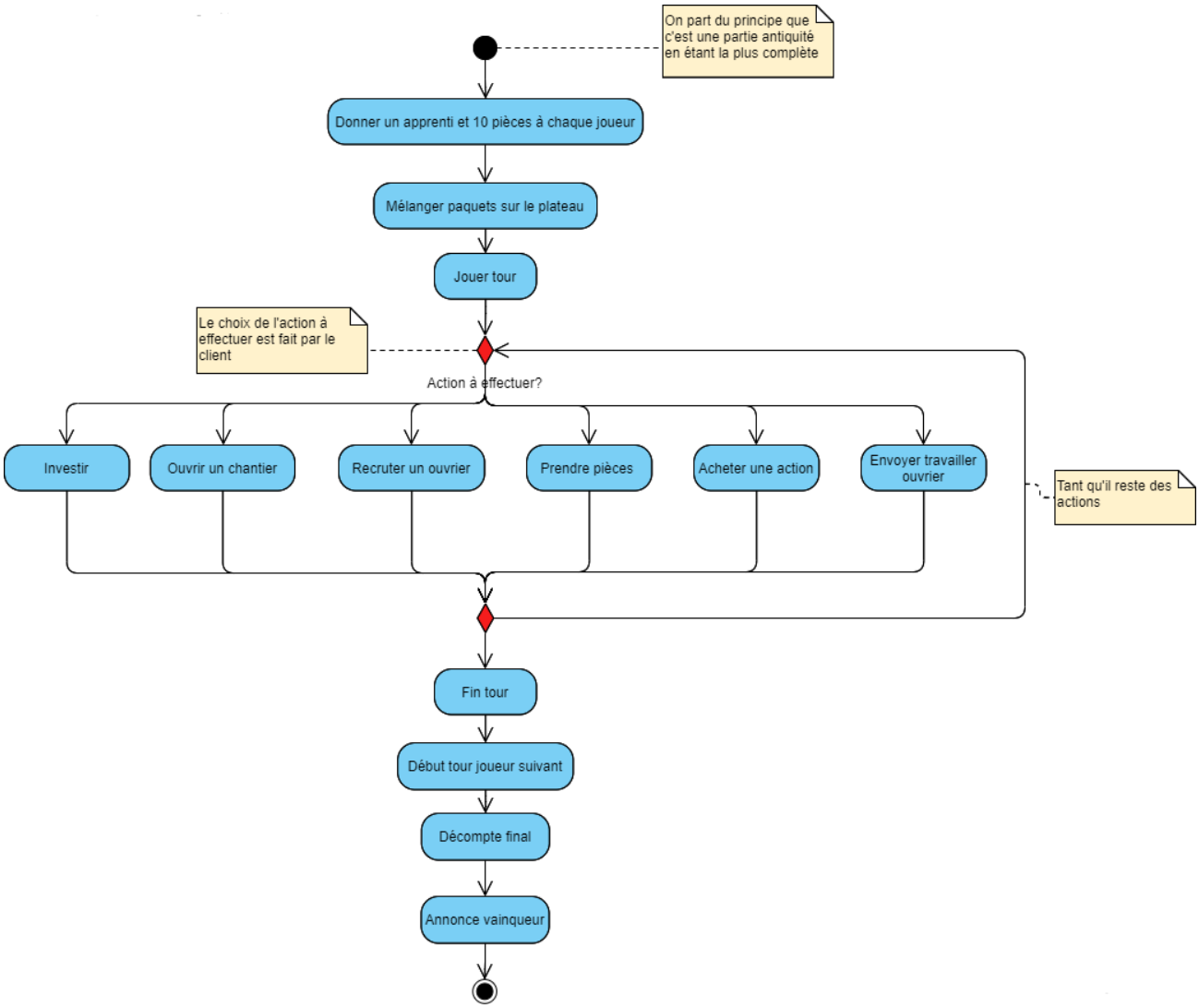
1) Point de vue général de l'architecture et des fonctionnalités

a. Glossaire

Ecus/Sesterces : monnaie de jeu

Points de victoire : points obtenus après avoir fini un chantier et issue du décompte final déterminant le gagnant

b. Représentation générale



c. Fonctionnalités traitées

A l'état final de la solution, il est possible de lancer une ou plusieurs parties entre minimum 2 joueurs. La partie se déroule dans un serveur après que le nombre de clients correspondant au nombre de joueurs attendus se sont connectés. Les jeux disponibles sont le jeu du moyen-âge et le jeu de l'antiquité. Si on a plus d'une partie alors on aura uniquement un affichage des statistiques finales après la fin de chaque partie.

Les fonctionnalités prises en compte (en gras celles qui n'étaient pas prises en compte dans la version précédente) dans le jeu sont :

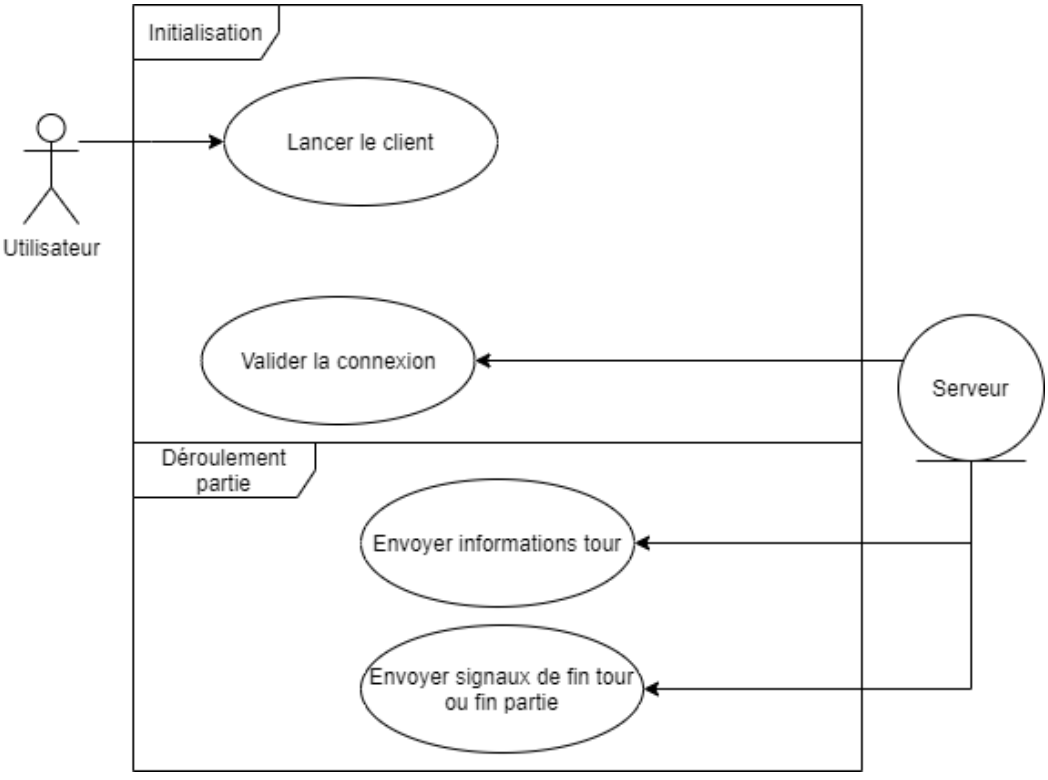
- Démarrage avec la création d'une pioche de cartes ouvriers et bâtiment avec des ressources aléatoires (pas celles des vraies cartes de jeu).
- Une fois les pioches constituées, chaque joueur recevra aléatoirement une carte apprenti.
- Possibilité pour chaque joueur d'effectuer une action issue de l'IA du client **(2 niveaux d'IA possibles) après un échange client-serveur** par rapport à l'état actuel de la partie.
- Toutes les actions du jeu sont disponibles : ouvrir un chantier, recruter un ouvrier, envoyer un ouvrier travailler, prendre des écus, acheter une action. **Aussi toutes les actions d'investissement si la partie est un jeu de l'antiquité.**
- Les cartes machines sont prises en compte, si on termine une machine on recevra l'ouvrier correspondant.
- A la fin de chaque chantier, chaque joueur reçoit les points victoire et écus associés.
- **Quand un joueur atteint 17 points victoire, un décompte final sera exécuté et les statistiques de fin de partie affichées.**
- Affichage textuel **en couleur** de toutes les fonctionnalités citées.

2) Modélisation du client

a. Analyse des besoins (exigences) : Cas d'utilisation

Diagrammes de Cas d'utilisation

Nous avons plusieurs acteurs à l'état final : Le commanditaire qui lancera les clients et le serveur, mais aussi les clients et serveurs eux-mêmes qui seront acteurs dans l'application de l'autre.



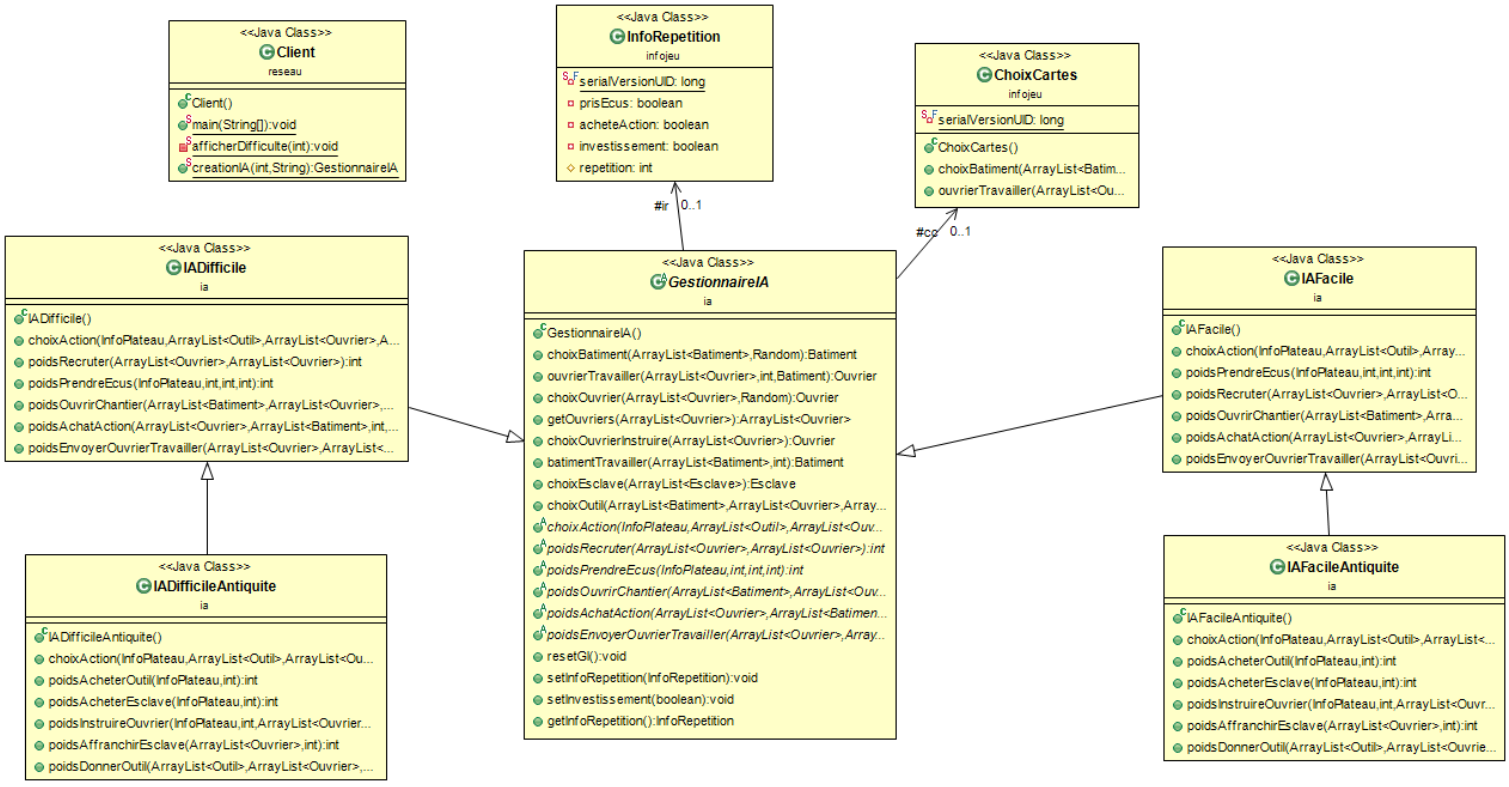
Scénarios

En ce qui concerne le client, on a qu'un seul scénario possible et aucun argument :

Scenario client : lancement du client

1. L'utilisateur lance le logiciel.
2. Le système affiche sa difficulté choisie aléatoirement.
3. Le système essaye de se connecter à un serveur.
    a. Cas échec : serveur occupé ou pas allumé, affichage textuel de message d'erreur et fin d'exécution.
4. Le système affiche textuellement la connexion et attend la fin du jeu.
5. Le système se déconnecte à la suite de la fin de jeu.

b. Conception logicielle



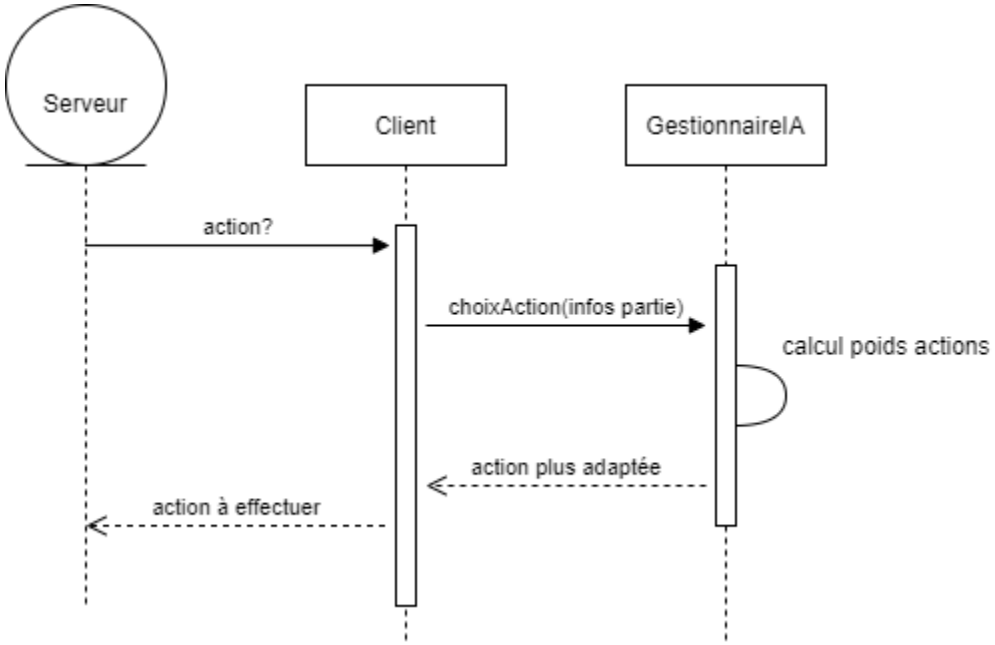
Niveau conception, nous avons la classe Client contenant la main nécessaire au lancement du programme (effectuant toutes les échanges réseau aussi) et tout ce qui est IA.

Pour l'implémentation de l'IA (qui se trouve à 100% dans le client) nous avons créé une classe abstraite gestionnaire IA, et 2 niveaux, une IA facile et une IA difficile.

En ce qui concerne le choix des IA, nous avons mis en place un système de poids pour nous permettre d'identifier le choix le plus adapté. Ce système de poids consiste à mettre en place des extrêmes pour chaque action :

- Les extrêmes négatifs seront les cas où cette action sera impossible à exécuter et 0 si très peu rentable (selon nous les développeurs).
- Les extrêmes positifs seront les cas où cette action sera très rentable à exécuter (toujours selon les développeurs)
- On a des degrés de rentabilité, 2 qui est rentabilité absolue (priorité de l'action) et 1 qui est rentable mais moins prioritaire des actions à 2
- Une fois retrouvé tous ces poids, dans choix action il ne reste plus qu'à d'abord exécuter une action à 2 et s'il n'y en a pas, alors exécuter une action à 1.
- S'il n'y a pas d'action à 2 alors enlever des choix possibles les actions à -1 et à 0. On rentre donc dans une zone grise où on détermine l'action par un random.

Pour avoir une IA moins intelligente, les choses ne se passent pas de la même façon, les poids sont toujours là mais dans le choix d'action on ne réalise pas les actions à 2 (donc très rentables). De plus, les actions qui devraient être dans la zone grise ont 70% de chance de ne pas y être ce qui donne encore moins de choix à cette IA.
Grâce à ça, on a des parties où l'IA avec l'implémentation complète des poids gagne très souvent contre celles avec l'implémentation plus aléatoire mais aussi nous n'avons pas de blocage si on souhaitait joueur entre des IA faciles uniquement.



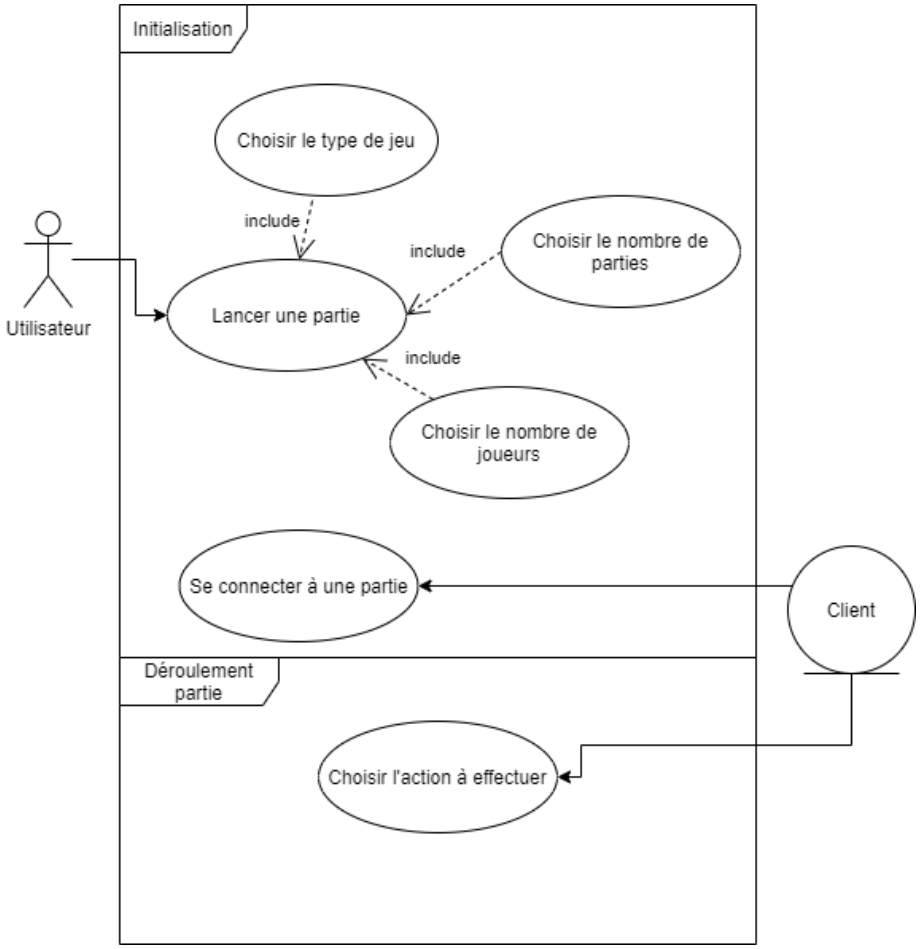
Ceci est un diagramme de séquence recapitulant ce qui a été expliqué, à noter que le gestionnaire d'IA pourra être soit une IA facile soit une IA difficile, et l'action choisie ne sera pas utilisée par le client mais par le serveur qui l'a demandé en premier lieu (cf partie 4).

3) Modélisation du serveur

a. Analyse des besoins (exigences) : Cas d'utilisation

Diagrammes de Cas d'utilisation

Nous avons plusieurs acteurs à l'état final : Le commanditaire qui lancera le serveur, mais aussi les clients qui interagissent avec l'application en tant qu'acteurs.



Scénarios

L'utilisateur peut lancer 2 types de scenarios pour le serveur : lancer une partie unique ou lancer plus qu'une partie à la fois. Il est possible de mettre en argument le type de partie ou le nombre de joueurs s'il ne veut pas qu'ils soient 4.

Scenario serveur : lancement de partie unique

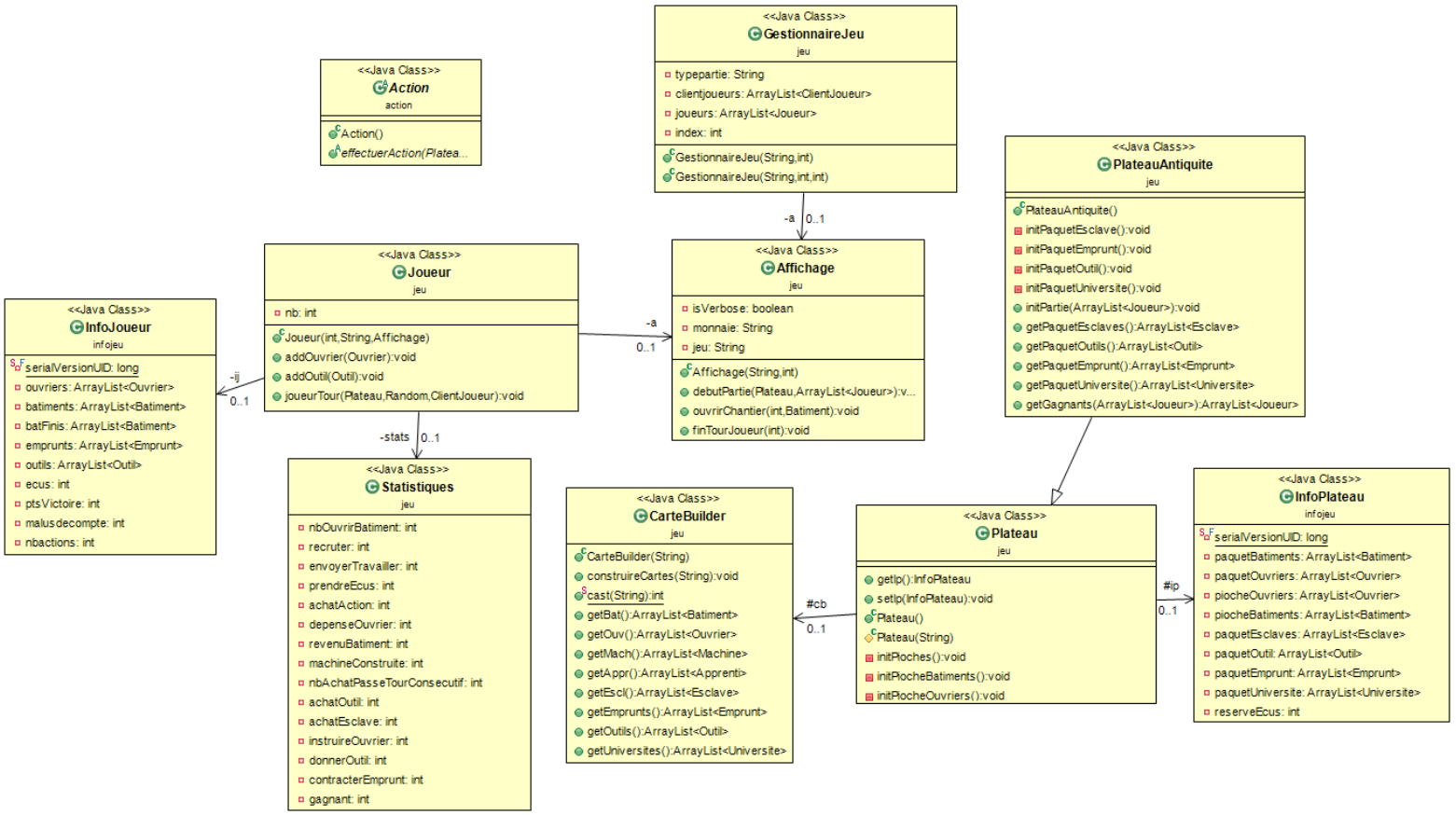
1. L'utilisateur lance le logiciel sans spécifier le nombre de joueurs
2. Le système se met en attente de 4 clients.
3. Le système valide les 4 clients en tant que joueurs et démarre la partie.
4. Le système affiche textuellement le déroulement du jeu à l'utilisateur
5. Le système affiche les statistiques du gagnant à la fin du jeu.
6. Le système déconnecte les clients.

Scenario serveur : lancement de plusieurs parties

1. L'utilisateur lance le logiciel sans spécifier le nombre de joueurs mais en spécifiant 500 parties
2. Le système se met en attente de 4 clients.
3. Le système démarre 500 jeux avec 4 joueurs.
4. Le système affiche les statistiques et pourcentages de victoire pour chaque joueur à la fin des parties.
5. Le système déconnecte les clients.

Dans les 2 scénarios, il y a la possibilité pour l'utilisateur de spécifier le nombre de joueurs souhaités, si rien n'est saisi, le système fera des jeux à 4 joueurs par défaut, on a omis le type de jeu du scénario car ça n'affecte pas considérablement le scénario mais il est important de noter que sans argument de type de jeu alors ça sera une ou plusieurs parties moyen-âge qui sera ou seront exécutés par défaut.

b. Conception logicielle



Pour chaque jeu on initialise un gestionnaire de jeu qui possède un affichage et une liste de joueurs, il crée un plateau par rapport au type de jeu.

Le plateau contient une classe commune d'info avec les pioches et paquets correspondants à chaque type de carte de jeu, les écus ou sesterces et les méthodes pour les manipuler, ces listes sont initialisées

avec une classe chargée de lire les fichiers csv avec les infos sur toutes les cartes dans le projet appelée CarteBuilder.

Le plateau antiquité est initialisé au lieu du plateau si c'est une partie antiquité, il manipule les pioches supplémentaires du jeu en question et hérite toutes celles du plateau classique.

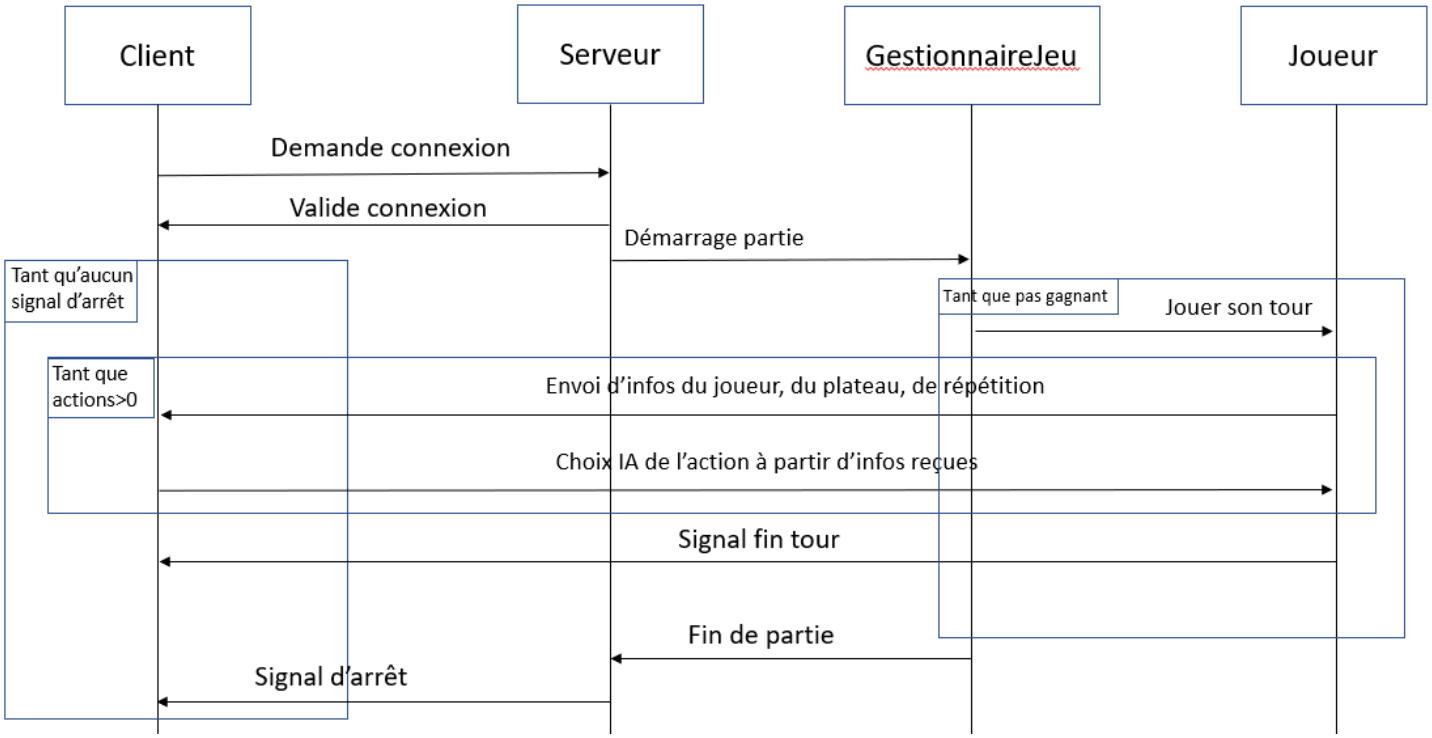
Le joueur possède une classe commune d'info contenant sa liste de bâtiments, d'ouvriers, d'outils, écus, points victoires, les méthodes pour les manipuler, un objet statistique qui est un log de toutes ses actions pour l'affichage final et aussi la méthode jouerTour où la majorité des échanges client-serveur vont s'effectuer à l'issue duquel il effectuera une action.

Action c'est une classe abstraite, chaque type d'action différente hérite d'Action et implémente effectuerAction (pas affichés sur le diagramme de classe pour éviter de prendre trop de place).

L'affichage est l'objet responsable d'afficher toutes les sorties du jeu, il a notamment un isVerbose qui indique si ses méthodes sont muettes ou pas, cela pour ne pas afficher dans les tests les affichages si on le souhaite mais aussi pour quand on lance multiples parties.

Le déroulement d'une partie dans le serveur sera expliqué dans le détail avec le diagramme de séquence dans la partie suivante.

4) Interactions entre le client et le serveur



Le serveur et client sont 2 mains différents : le serveur démarre et se met en attente des clients, le client cherche un serveur à qui se connecter.

Quand un client reçoit la validation de sa connexion au serveur, il initialise son niveau de IA et il se met en attente d'un signal tant qu'il ne reçoit pas de signal d'arrêt.

Le serveur de son côté initialise autant de GestionnaireJeu que de nombre de jeux, s'il y a un seul jeu alors on aura un affichage de la partie complète, s'il y a plusieurs parties alors chaque jeu sera un thread pour augmenter la vitesse d'exécution de chaque jeu individuel.

Dans chaque jeu, chaque joueur qui connaitra préalablement son client spécifique grâce à l'objet ClientJoueur effectuera des échanges avec son client.

Cette échangée est synchronisé pour éviter des chevauchements de signaux sur un même client provenant de threads différents.

L'échange consiste à, tant que le joueur a d'actions, envoyer une liste contenant les infos du plateau, infos du joueur et infos de répétition (infos répétition telles que si un investissement a déjà été fait). Ces informations sont des objets simples dans un module commun que le client et le serveur se départagent.

Le client reçoit ces informations et il s'active, son IA choisira à partir d'elles l'action adéquate et renvoie un String avec nom de l'action à effectuer au serveur.

Quand le serveur reçoit l'action, le joueur l'effectue et met à jour son nombre d'actions.

Quand le tour du joueur est fini, on envoie au client un signal de fin de tour pour qu'il réinitialise les infos spécifiques au tour en cours de son IA.

Tout ceci au long du jeu et quand la partie ou multi partie est finie, le serveur envoie un signal d'arrêt à chaque client pour les déconnecter.

5) Conclusion

a. Points forts et points faibles

- Points forts
    - ✓ Toutes les fonctionnalités demandées ont été implémentées.
    - ✓ Logique de jeu qui marche correctement.
    - ✓ IA ne cause pas de blocage, on ne peut pas tomber sur une partie qui tourne à l'infinie sans gagnant.
    - ✓ Initialisation insensible à la casse, on peut saisir des majuscules ou caractères invalides sans affecter le programme.
    - ✓ Possibilité de modifier les cartes avec le fichier en lecture.
- Points faibles
    - X Le serveur et les clients ne sont pas testés, cela s'explique par le fait que c'est des main principalement.
    - X Echange réseau non testé, il y a donc un risque d'instabilité.

- X Difficulté choisie aléatoirement dans client au lieu d'avoir un argument pour la choisir.
- X Echange réseau lent à cause de l'utilisation de Sockets de java, il faudrait optimiser.

b. Amélioration possibles

La première amélioration qu'on voit c'est de changer l'échange d'objets par une échange de JSON (avec donc refactorisation pour utiliser socketsIO), cela nous permettra d'améliorer la vitesse d'exécution des échanges réseau. Enfin, il ne restera plus qu'améliorer la couverture des tests et la javadoc.