
PI4

Rapport de Projet

ML1B

Sommaire

1 Introduction	4
1.1 Présentation	4
1.1.1 Histoire d'Acquire	4
1.1.2 But du Jeu	4
1.1.3 Matériel de Jeu	4
1.1.4 Déroulement du Jeu	4
1.2 Fin de Jeu	5
1.2.1 Règles Spécifiques	5
1.2.2 Stratégie	5
1.3 État du jeu	5
1.4 Objectifs	6
2 Architecture	7
2.1 Modèle	7
2.1.1 tools	7
2.1.2 processor	7
2.1.3 game	8
2.2 Vue	8
2.2.1 assets	8
2.2.2 Components	8
2.2.3 game	8
2.2.4 login	9
2.2.5 menu	9
2.2.6 window	10
2.3 Contrôleur	10
2.3.1 auth	10
2.3.2 database	10
2.3.3 firebaseinit	10
2.3.4 game	10
2.3.5 menu	10
2.3.6 network	10
3 Logique du jeu	11
3.1 Implémentation des règles du jeu	11
3.2 Version réseau du jeu	11
3.3 Chat en jeu	11
4 Interface graphique	11
5 Réseau	14
5.1 Tentative de création de réseau local	14
5.2 Création du réseau avec Firebase	16
6 Monte-Carlo	18
6.1 Introduction et histoire	18
6.2 Principe de la recherche Monte-Carlo	18
6.3 Implémentation dans le projet	18
6.3.1 Structure de MonteCarloAlgorithm	18
6.3.2 Méthode runMonteCarlo	18
6.3.3 Choix de la meilleure action	19
6.4 Résultats obtenus	19

6.4.1	Validation du Concept	19
6.4.2	Impacts stratégiques	19
6.4.3	Défis rencontrés	19
6.4.4	Perspectives futures	19
6.5	Pistes inexplorées	19
7	Conclusion	20
8	Ouverture	20

1 Introduction

1.1 Présentation

1.1.1 Histoire d'Acquire

Acquire est un jeu de société économique conçu par Sid Sackson, un designer de jeux renommé, et publié pour la première fois en 1962 par la société 3M. Depuis sa création, le jeu a connu plusieurs éditions et a été édité par différentes maisons d'édition, notamment Avalon Hill et Hasbro. Le concept unique et stratégique d'*Acquire* a fait de ce jeu un classique apprécié des amateurs de jeux de stratégie et d'économie à travers le monde. Sa popularité durable témoigne de la profondeur et de la qualité de son gameplay.

1.1.2 But du Jeu

Le but du jeu *Acquire* est de devenir le joueur le plus riche en investissant dans des chaînes hôtelières, en les développant et en fusionnant des chaînes pour maximiser les profits. Le joueur ayant accumulé le plus d'argent à la fin du jeu est déclaré vainqueur.

1.1.3 Matériel de Jeu

Le jeu *Acquire* se compose des éléments suivants :

- **Plateau de jeu** : Une grille de 12x9 cases, numérotées de 1A à 12I.
- **Tuiles** : Représentant des emplacements hôteliers sur le plateau.
- **Cartes d'actions** : Indiquant les coordonnées des tuiles.
- **Actions des chaînes hôtelières** : Cartes représentant des actions dans différentes chaînes hôtelières.
- **Billets** : Représentant l'argent utilisé pour acheter des actions et effectuer des transactions.
- **Marqueurs de chaînes** : Utilisés pour indiquer les hôtels sur le plateau.
- **Tableau de valeurs** : Indiquant la valeur des actions en fonction de la taille des chaînes.

1.1.4 Déroulement du Jeu

Le jeu se déroule en plusieurs tours, chaque joueur effectuant les actions suivantes lors de son tour :

1. **Poser une tuile** : Le joueur place une tuile sur le plateau en fonction des coordonnées indiquées sur sa carte.
2. **Créer une chaîne hôtelière** : Si la tuile est placée à côté d'autres tuiles non connectées, une nouvelle chaîne hôtelière peut être créée.
3. **Fusion de chaînes** : Si la tuile connecte deux chaînes existantes, une fusion se produit. La chaîne la plus grande absorbe la plus petite.
4. **Acheter des actions** : Le joueur peut acheter jusqu'à trois actions de n'importe quelle chaîne hôtelière encore en jeu.
5. **Piocher une tuile** : Le joueur pioche une nouvelle tuile pour remplacer celle qu'il a jouée.

1.2 Fin de Jeu

Le jeu se termine lorsqu'il n'est plus possible de placer de nouvelles tuiles ou lorsqu'une chaîne atteint une taille prédéterminée, souvent 41 tuiles. À la fin du jeu, les actions sont liquidées, et les joueurs reçoivent de l'argent en fonction de la valeur de leurs actions et des bonus pour les deux plus grands actionnaires (majoritaire et minoritaire) dans chaque chaîne.

1.2.1 Règles Spécifiques

- **Création d'une chaîne** : Un joueur crée une chaîne en plaçant une tuile à côté d'une tuile déjà posée sur le plateau. Le joueur choisit une chaîne parmi celles disponibles.
- **Fusions** : Lors d'une fusion, les actionnaires de la chaîne absorbée reçoivent des paiements pour leurs actions, puis peuvent choisir de vendre, de conserver ou d'échanger leurs actions contre des actions de la chaîne dominante.
- **Valeur des Actions** : La valeur des actions varie selon la taille de la chaîne. Plus une chaîne est grande, plus la valeur de ses actions est élevée.
- **Majorité et Minorité** : À la fin du jeu ou lors de fusions, les deux plus grands actionnaires dans chaque chaîne reçoivent des bonus financiers.

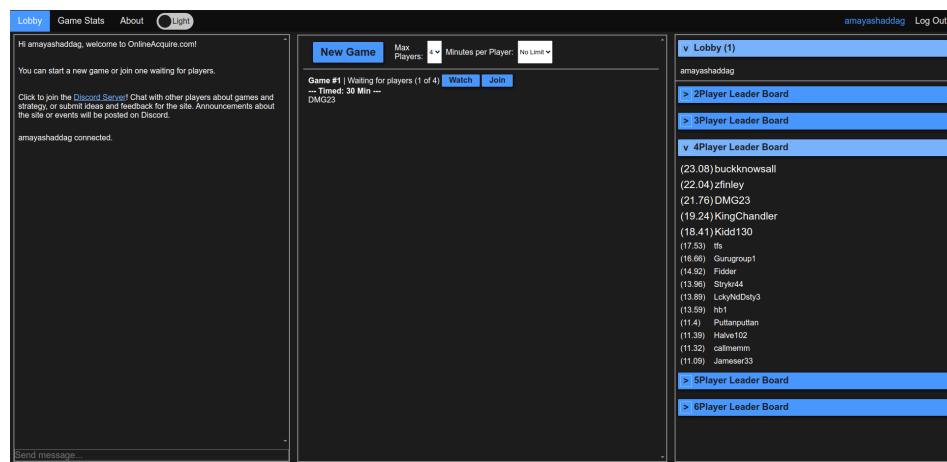
1.2.2 Stratégie

Acquire est un jeu où la stratégie est primordiale. Les joueurs doivent judicieusement investir dans les chaînes hôtelières, anticiper les fusions et gérer leur portefeuille d'actions pour maximiser leurs profits. Une bonne connaissance des règles et une planification attentive sont essentielles pour gagner.

1.3 État du jeu

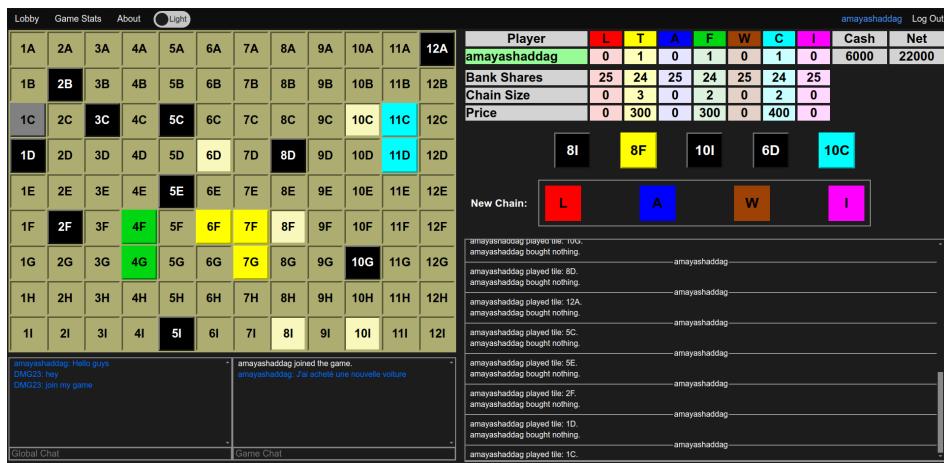
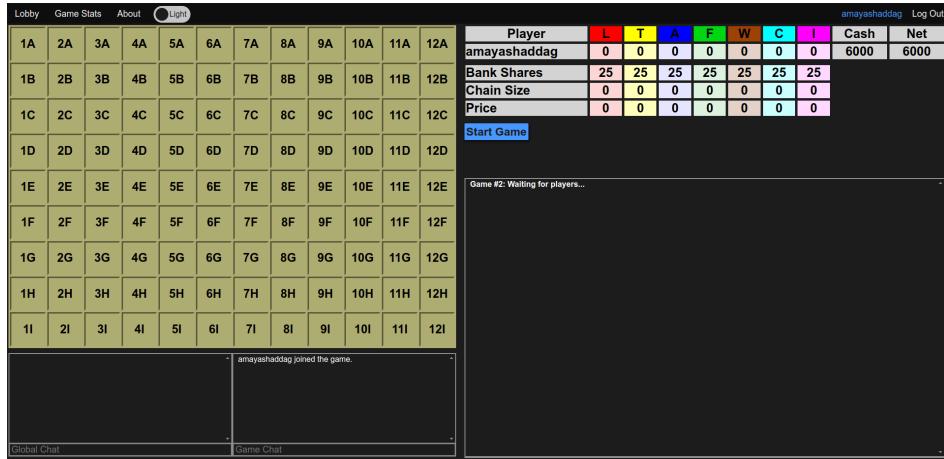
Acquire possède différentes éditions physiques mais nous allons nous intéresser aux éditions virtuelles. Actuellement, il n'existe qu'une seule version jouable sur ordinateur¹.

Cette version implémente toutes les fonctionnalités du jeu et est totalement fonctionnelle. Néanmoins, certains lui approcheront son caractère excessivement minimalisté, notamment ses graphiques.



Ce manque de graphisme nuit à l'attractivité du jeu et à sa dynamique, ce qui le rend difficilement accessible.

¹<https://onlineacquire.com/>



1.4 Objectifs

Au vu des remarques précédentes, nous avons été investi par *Maximilien Lesellier* de redorer l'image du jeu et d'en créer une version accessible et attrayante sur ordinateur.

Quelques contraintes nous ont été imposées :

- Une limite de temps de 4 mois.
- Programmer le jeu en java natif.
- Ne pas utiliser JavaFX.
- Ne pas utiliser de gestionnaire de packages (Maven, Gradle, ...).
- Contraintes d'optimalité.

Une fois le cadre du travail fixé, il nous fallait des objectifs. Nous avons souhaité parfaire et non recréer la version déjà présente en ligne. Nous apprécions la simplicité de cette dernière, mais nous la trouvons excessive. Nos objectifs étaient clairs, rendre le jeu plus beau, plus dynamique, plus attractif. Nous n'avons pas la prétention de réinventer le jeu, nous souhaitons rester sur les acquis et apporter une touche de fraîcheur à *Acquire*. Pour cela, nous avons donc pensé à de nouvelles fonctionnalités que nous aborderons

dans les paragraphes suivants².

Dès le début, notre cahier des charges a été établi :

- **Gameplay** : un style de jeu rapide et interactif, visant à maintenir l'attention du joueur constante tout en permettant des interactions régulières assurant le côté social et stratégique du jeu.
- **Interface** : une interface simple, intuitive et vive. L'interface symbolise le renouveau du jeu et assure son attractivité.
- **Fonctionnalités de base** : toutes les fonctionnalités de la version en ligne devaient être disponibles. Ceci incluait donc le mode réseau (5),
- **Intelligence artificielle** : permettant différents niveaux de difficulté.

2 Architecture

Pour ce projet, nous avons opté pour le design-pattern Model-View-Control en utilisant le paradigme objet. Nous avons donc découpé l'architecture de notre projet en trois parties principales : model, view, et control. Dans les prochaines sous-sections, nous allons donner un aperçu de l'implémentation du pattern Model-View-Control dans notre projet (cette section n'a pas pour but de décrire en détails l'implémentation de notre projet mais seulement la structure et l'organisation des classes).

2.1 Modèle

Nous avons organisé notre dossier de la façon suivante : Un dossier tools ou se trouve des classes intermédiaires que nous avons utilisé pour définir des nos classes principales. Un dossier processor : jsp il sert à quoi. Et enfin un dossier game ou se trouve les classes principales du jeu.

2.1.1 tools

- Action : Contient principalement les setters, getters, printer..etc.
- AutoSetter.
- Box : équivalent à un t* en C.
- MergingChoice : une enum qui contient SELL, TRADE et KEEP.
- PlayerAnalytics : contient les informations relatives aux statistiques des joueurs telles que le nombre de parties gagnées par ce dernier, son meilleur score... ces informations sont sauvegardées grâce à Serializable.
- PlayerCredentials : pour les informations personnelles des joueurs telles que les pseudo, mail et uid. Tout comme pour PlayerAnalytics, ces informations sont sauvegardées en implémentant Serializable.
- Point : utile à chaque fois qu'on manipule des coordonnées (x, y). Contient toutes les fonctions relatives.
- PreGameAnalytics : contient les statistiques et les informations relative à la partie du serveur. On y trouve par exemple le hostName, gameId, nombre maximal et courant de joueurs...

2.1.2 processor

- AutoSetterProcessor.

²Faute de temps, une partie d'entre elles ont dû être abandonnées (8).

2.1.3 game

- Board : Classe principale du model, c'est une grille (matrice) de Cell 9x9 qui regroupe tout les éléments de ce dernier. Chaque action performée par un joueur affecte directement le board qui apporte les modifications nécessaires.
- Cell : représente une case du Board. Peut être dans plusieurs états comme EMPTY, OCCUPIED, OWNED ou DEAD. Elle contiendra également (ou pas selon son état) les corporations.
- Corporation : représente une chaîne d'hôtel qui est de type enum. Ce sont les mêmes chaînes qui sont dans la charte de référence du jeu. A savoir : WORLDWIDE, SACKSON, FESTIVAL, IMPERIAL, CONTINENTAL, TOWER, AMERICAN.
- Player : représente typiquement un objet Player. Ses attributs comme son crédit pendant la game, la liste des stocks qu'il possède, son deck...etc. Les actions qu'il peut performer : dépenser de l'argent, modifier sa liste de stocks..etc.
- ReferenceChart : contient toutes les variables, constantes et les fonctions relatives à la charte de référence, elle inclut donc la logique permettant d'implémentant les règles du jeu. e.g : les prix de chaque corporations, le classement par catégorie des corporations, une fonction qui retourne le prix d'achats... etc.

2.2 Vue

Le dossier view contient les ressources utilisées pour la construction de l'interface utilisateur, facilitant la gestion et l'organisation des éléments visuels.

2.2.1 assets

- Fonts : est conçue pour gérer les polices utilisées dans l'interface du jeu. Elle définit des polices spécifiques pour les titres et les paragraphes, en tentant de charger des polices personnalisées à partir des fichiers.
- GameResources : facilite la gestion et le chargement automatique des images pour l'interface du jeu à partir de fichiers, en suivant une convention de nommage spécifique. Elle contient une classe interne, Assets, qui initialise dynamiquement ces images en tant que champs statiques.
- LoginInterfaceResources : regroupe les constantes textuelles et visuelles pour l'interface de connexion du jeu.
- MenuInterfaceMessages : sert principalement de répertoire pour les messages textuels utilisés dans le menu du jeu.
- MenuResources : gère le chargement et la mise à disposition des ressources visuelles pour le menu du jeu. Elle utilise un processus automatisé pour charger les images à partir d'un chemin de fichier spécifié, en respectant une convention de nommage qui transforme les noms de variables en noms de fichiers.

2.2.2 Components

2.2.3 game

- ChoiceCorpPane : est un composant d'interface utilisateur spécifiquement conçu pour permettre aux joueurs de sélectionner une corporation parmi une liste donnée. Elle étend JComponent et utilise MigLayout pour l'agencement des éléments, offrant un arrangement centré et fluide des options de corporation.

- **EndGame**
- **ColorableArcableFlatBorder** : améliore les capacités de style visuel des composants de l'interface utilisateur en permettant aux développeurs de définir des couleurs spécifiques et un arrondi des coins, tout en restant compatible avec le style de l'interface utilisateur FlatLaf.
- **GameNotifications** : centralise la gestion des messages de notification dans le jeu, incluant les erreurs et les confirmations de succès des actions des joueurs.
- **GameView** : est un composant central de l'interface utilisateur du jeu qui gère l'affichage et l'interaction avec la carte du jeu. Elle hérite de Form et intègre plusieurs composants pour représenter divers aspects du jeu.
- **GlowingItemCorp** : elle est spécialisée pour afficher des éléments graphiques avec un effet de gradient lumineux.
- **GrowingJLabel** : elle étend JLabel, qui utilise un effet d'animation pour agrandir ou réduire sa taille lorsque la souris entre ou sort de sa zone.
- **JetonsPanel** : est un composant personnalisé de type JPanel qui s'intègre dans l'interface du jeu pour gérer l'affichage et l'interaction avec les jetons du joueur.
- **MouseManager** : est un gestionnaire d'événements de souris spécialement conçu pour la GameView.
- **PausePane** : représente un panneau de pause pour le jeu, permettant aux joueurs d'interagir avec le chat, de visualiser les actions des joueurs, et de quitter le jeu.
- **PlayerBoard** : est un panneau personnalisé (JPanel) qui sert à afficher des informations sur les joueurs dans l'interface utilisateur du jeu.

2.2.4 login

- **EmailField**
- **LoginView** : est un panneau (JPanel) conçu pour gérer l'interface utilisateur relative aux actions de connexion et d'inscription des utilisateurs dans le jeu.
- **PasswordField**
- **PseudoField**

2.2.5 menu

- **EventMenu**
- **Menu3D** : hérite de JComponent et propose un système de menu en trois dimensions visuellement dynamique. Cette classe offre une interface personnalisable et interactive pour afficher et gérer des éléments de menu avec un effet 3D.
- **Menu3dItem** : est une composante essentielle du système de menu 3D dans le package view.menu. Elle encapsule les détails et les comportements d'un élément de menu individuel, incluant son rendu en trois dimensions, sa gestion des événements de la souris, et ses animations.
- **MenuAnimator** : est conçue pour gérer les animations des éléments de menu 3D, tels que ceux représentés par des instances de Menu3dItem.
- **MenuView** : agit comme l'interface utilisateur principale du menu, permettant aux joueurs de naviguer entre différentes options telles que commencer une partie, modifier les options, ou quitter le jeu.

- PrettyMenuView
- TableGradientCell

2.2.6 window

- Form
- GameFrame : agit comme la fenêtre principale pour le jeu, offrant une base solide pour l'affichage des divers composants de l'interface utilisateur tels que les vues de jeu, les menus et les écrans d'options.

2.3 Contrôleur

2.3.1 auth

- AuthController : est conçue pour gérer toutes les interactions liées à l'authentification des utilisateurs.

2.3.2 database

- GameDatabaseConnection : agit comme une interface entre les diverses fonctionnalités du jeu et la base de données Firestore. Elle joue un rôle essentiel en permettant de gérer efficacement les interactions relatives aux données du jeu.

2.3.3 firebaseinit

- FirebaseClient : sert à initialiser et à gérer la connexion avec Firebase, une plateforme développée par Google pour créer des applications mobiles et web.

2.3.4 game

- BotController
- GameController : gère les aspects principaux du jeu, tels que le placement de tuiles, la gestion des actions des joueurs, et l'interaction avec la vue et la base de données.
- MonteCarloAlgorithm

2.3.5 menu

- MenuController : gère le menu principal du jeu, y compris les interactions avec les sessions de jeu, les joueurs, la base de données et la vue. Elle est responsable de la gestion des sessions de jeu, du lancement de nouvelles parties (en ligne et hors ligne), de la gestion des joueurs et de l'interface utilisateur associée au menu.

2.3.6 network

- Client : permet à un client de se connecter à un serveur via un socket, d'envoyer des messages et de recevoir des réponses.
- Server : permet de gérer plusieurs clients se connectant à un serveur via des threads. Chaque client est géré par une instance de la classe ServiceThread.

3 Logique du jeu

3.1 Implémentation des règles du jeu

Acquire étant un jeu de plateau avec des actions de joueurs fixes et définies, nous avons voulu élaborer l'implémentation qui allierait au mieux efficacité et optimalité du code. Pour ce faire, nous avons fait de notre mieux pour centraliser toutes les informations nécessaires dans la classe `Board` et éviter les calculs inutiles. Cette classe est ensuite appelée par le contrôleur afin de faire déclencher les actions.

Dans le but de concevoir le modèle le plus efficace possible, nous avons opté pour les méthodes de recherche de graphes de type DFS appliquées directement sur le plateau de jeu afin de calculer les données nécessaires pour le déroulement du jeu.

3.2 Version réseau du jeu

Dans la version réseau du jeu, nous avons tenté une approche atypique et assez différente des approches classiques de la programmation réseau. Les raisons qui nous ont poussé à ceci, se cachent dans les limites imposées par les serveurs réseau en local et leur manque de flexibilité quant à ce qu'il s'agit de la gestion des adresses IP. Aussi, étant donné que le jeu que nous avons programmé est bâti sur des règles assez simples avec des tours définis à l'avance, nous avons tenté d'implémenter une version réseau totalement jouable sur base de données.

L'approche que nous avons élaboré se base sur le principe que chaque joueur simule les calculs nécessaires à son tour dans son propre client puis envoie les résultats sur la base de données. Les autres joueurs lisent les nouvelles données et mettent à jour leurs clients.

La base de données utilisée est hébergée sur le site web de Google Firebase³, ce qui garantit un accès permanent et fluide à la base de données, ainsi pouvoir lancer des parties en multi-joueur à tout moment.

3.3 Chat en jeu

Vu que la version en ligne du jeu a été implémentée en utilisant uniquement une base de données, il nous fallait alors une approche similaire afin de pouvoir intégrer une discussion en direct.

L'approche qui a été choisie est de stocker un à un les messages des joueurs envoyés dans la base de données. Pour définir une relation d'ordre qui trie ces messages en ordre chronologique, nous avons muni chaque message d'une valeur qui représente l'heure exacte en GMT+0 du moment où il a été envoyé.

Ainsi, vu qu'également, on fait le tri entre les messages qu'on envoie et ceux qu'on reçoit, on est arrivé à recréer un chat en direct où on peut envoyer et recevoir des messages au milieu de la partie. Le chat se trouve dans le menu pause du jeu qui est accessible en cliquant sur n'importe quelle touche du clavier. Le chat possède aussi une fonctionnalité d'identification des joueurs par le biais du symbol @ en lui ajoutant le pseudo de la personne, ceci fait office d'envoyer une notification à la personne en question pour l'avertir que le message lui était destiné.

4 Interface graphique

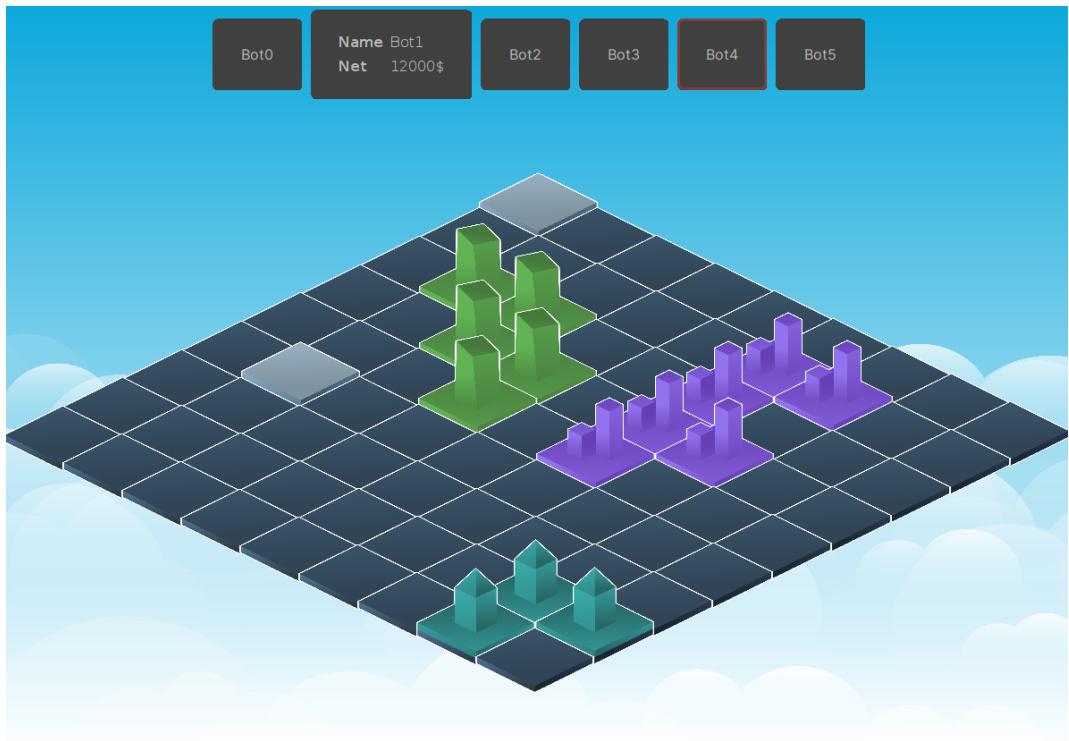
L'amélioration de l'interface graphique par rapport à la version en ligne fut dès le début un des axes majeurs de notre projet. C'est pourquoi nous avons choisi de mettre l'accès sur un interface moderne, épuré et efficace.

³<https://firebase.google.com/>

Dans cette optique, nous opté pour l'utilisation du style *DarkLaf* de la bibliothèque FlatLaf. Nos composants s'inspirent directement des interfaces modernes.

Pour ce faire, nous avons utilisé de nombreuses librairies communautaires et nous nous sommes inspirés de nombreux développeurs professionnels comme par exemple Ra Ven. Notre tactique était simple : plutôt que de tout réinventer comme nous l'avons fait dans nos anciens projets, nous avons choisi de rechercher sur internet des composants graphiques correspondants à nos besoins, puis de les personnaliser, pour qu'ils correspondent parfaitement à nos besoins.

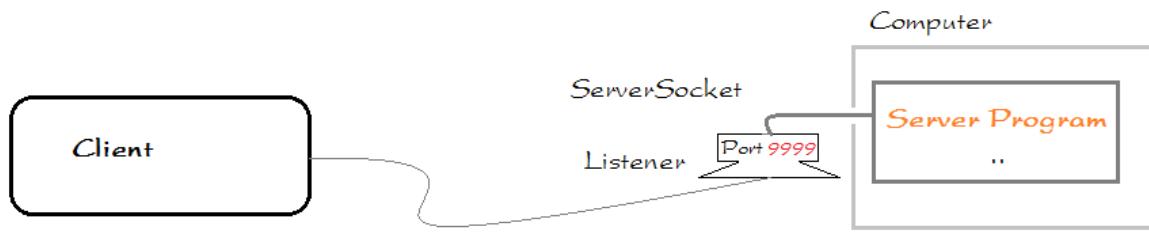




5 Réseau

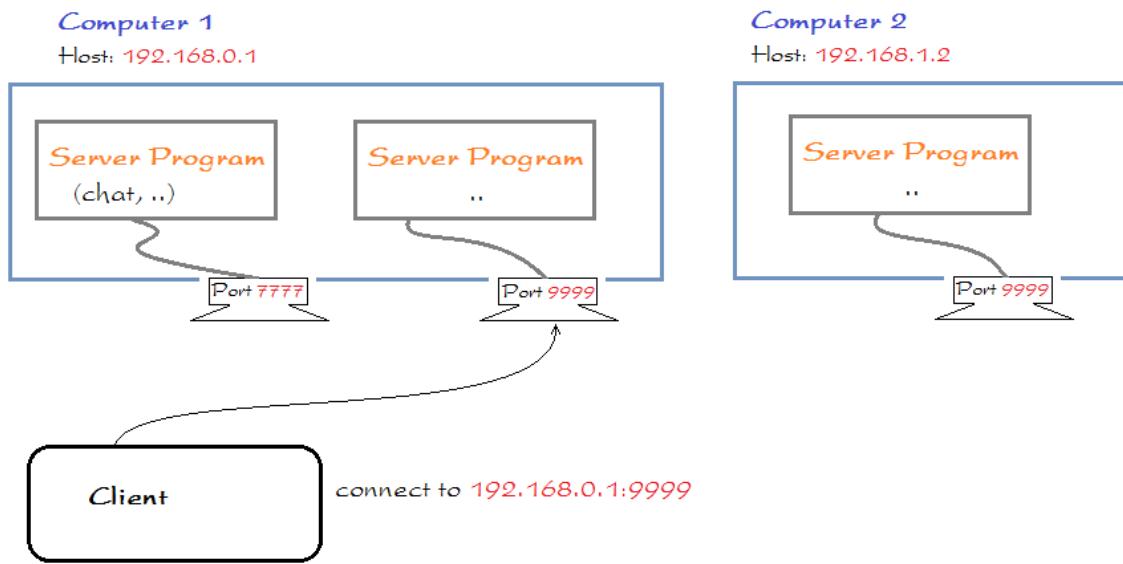
5.1 Tentative de création de réseau local

En ce qui concerne le réseau, il était d'abord prévu que l'on en crée un local. En effet la machine d'un des joueurs servait alors de serveur sur laquelle les autres participants se connectaient. Pour coder cette fonctionnalité nous avons utilisé le modèle Client/Serveur classique. On utilisait pour cela les classes ServeurSocket et Socket de java, dans la classe Server était créer un objet ServerSocket avec un numéro de port, qui gérait les flux de connexions des clients.



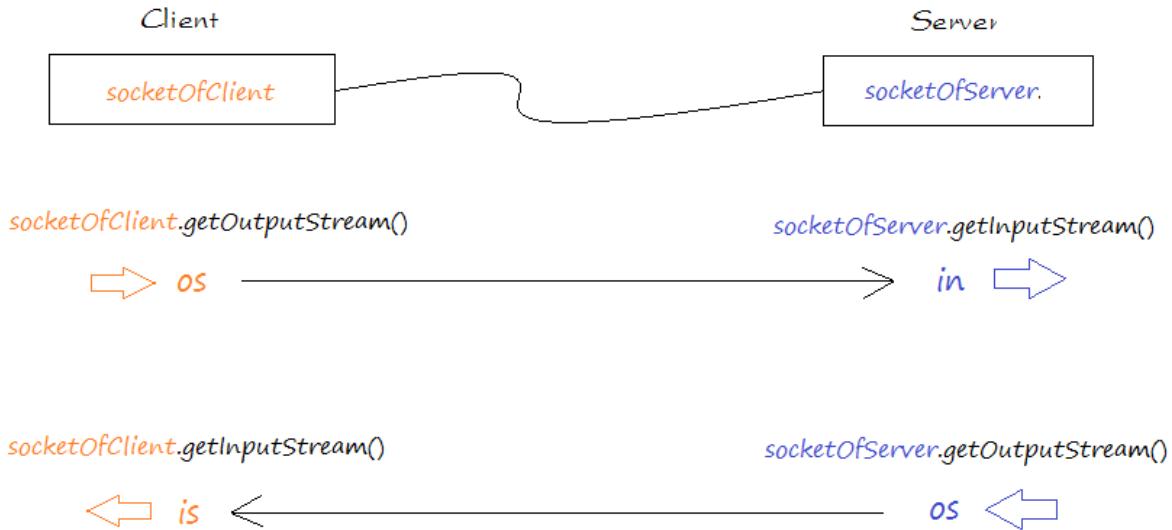
Send a request to connect to the server

Sur cette image on voit bien bien le serveur recevoir une demande de connexion d'un client. Les clients eux se connectaient à l'aide d'un objet Socket en précisant l'adresse IP de la machine qui nous servait de serveur ainsi que le numéro de port. Ici on voit bien l'importance de préciser à la fois le numéro de port et



l'IP de la machine qui sert de serveur.

Une fois connecté le client et le serveur communiquent en écrivant et en lisant sur les flux d'entrée et de sortie grâce aux classes InputStreamReader et OutputStreamWriter et aux méthodes getInputStream, getOutputStream ainsi que flush (pour envoyer sous une forme de paquet les données)



5.2 Création du réseau avec Firebase

Cependant bien qu'on avait bien avancé sur la création de ce réseau local nous avons finalement décidé de nous tourner vers une autre solution, celle d'utiliser Firebase comme base de données. En effet, il y avait plusieurs raisons à cela, la première était qu'avec Firebase nous pouvions créer un réseau global et non plus seulement local ce qui est quand même un gros avantage, la deuxième raison est que nous n'avions pas réellement besoin d'implémenter un serveur à proprement parler, nous avions juste besoin d'une base de données qui centraliseraient les informations de la partie tel que le nombre d'actions ou de cash d'un joueur ou l'état de la map etc... et c'est exactement ce que Firebase nous fournissait. Enfin Firebase nous mâchait un petit peu le travail avec des méthodes pour lire et écrire sur la base de données à notre disposition. Tous ces arguments écrasants nous ont conduits à l'unanimité au sein de notre groupe et avec notre chef de projet de nous tourner vers cette solution.

Nous avons structuré notre base de données avec les 10 tables suivantes:

analytics	chat	games	placed-cells	registered-users
-uid	-uid	-creator	-corporation	-uid
-best-score	-game-id	-game-id	-game-id	-email
-played-games	-message	-max-players	-x-position	-password
-won-games	-time	-state	-y-position	-pseudo

stocks	players	current-player	notifications	major-corporation
-uid	-uid	-uid	-message	-corporation
-game-id	-game-id	-game-id	-game-id	-game-id
-amount	-cash		-time	-time
-corporation	-net			
	-pseudo			

analytics: stock toutes les informations qui concernent le classement universel le pseudo des joueurs qui ont gagné le plus de parties qui ont fait les meilleurs scores etc... .

games: contient toutes les informations relatives à une partie, son état (en cours, terminé, vient d'être créée), son créateur et le nombre maximum de joueurs qu'elle peut accueillir.

placed-cells: garde les informations concernant les cellules placées sur la map d'une certaine partie. Cette table nous sert par exemple à centraliser tous les coups réalisés par les joueurs et permet donc que tout le monde soit à jour au même instant t de la partie.

registered-users: stock les données qui concernent les informations de connexions des utilisateurs, l'email, le mot de passe le pseudo et l'identifiant du joueur.

stocks: contient les informations sur le nombre d'actions achetés dans l'entreprise x dans la partie avec l'identifiant stocké dans game-id.

players: garde toutes les informations relatives aux joueurs dans une partie comme leur nombre d'actions d'argent etc... .

current-player: stock l'identifiant du joueur auquel c'est le tour de jouer dans une partie.

major-corporation: garde l'information de l'entreprise dominante lors d'une fusion.

chat: nous permet de stocker les messages des joueurs au cours d'une partie.

notifications: contient les messages et les moments auxquelles des notifications ont été lancés au cours d'une partie.

Nos méthodes qui communiquent avec cette base de données se décomposent en 2 catégories, celles qui lisent la base de données afin de mettre à jour les informations du joueur, et celles qui écrivent sur celle-ci pour centraliser les données ainsi les autres joueurs pourront lire sur la BDD et mettre à jour leur partie.

6 Monte-Carlo

6.1 Introduction et histoire

La méthode de Monte Carlo, nommée d'après la célèbre ville monégasque connue pour son casino, est une approche statistique qui utilise des simulations aléatoires pour résoudre des problèmes numériques complexes. Développée initialement durant le projet Manhattan dans les années 1940 par des scientifiques tels que Stanislaw Ulam et John Von Neumann, cette méthode a depuis trouvé des applications dans divers domaines allant de la finance à l'ingénierie et la recherche scientifique.

Dans le contexte de notre jeu Acquire, un jeu de stratégie économique complexe, l'adoption de cette méthode s'est avérée naturelle. Acquire, qui implique des prises de décision sous incertitude et une compétition entre plusieurs joueurs, requiert une approche qui peut modéliser et anticiper les conséquences des actions multiples et diversifiées des joueurs. La méthode de Monte Carlo, avec sa capacité à simuler des milliers de scénarios possibles et à en extraire des statistiques significatives, offre un outil puissant pour optimiser les stratégies du bot, améliorant ainsi la compétitivité et l'interaction dans le jeu.

6.2 Principe de la recherche Monte-Carlo

La méthode de Monte Carlo est essentiellement basée sur l'échantillonnage aléatoire pour estimer des solutions probabilistes à des problèmes qui seraient autrement trop complexes pour des approches déterministes. En générant un grand nombre de scénarios aléatoires et en observant la proportion des scénarios qui répondent à certains critères, on peut approximer des solutions à des problèmes de physique, d'optimisation, de finance, et plus récemment, de stratégie dans les jeux informatiques. Cette méthode est particulièrement utile dans les contextes où les dynamiques du système sont trop complexes pour être modélisées par des équations simples.

6.3 Implémentation dans le projet

6.3.1 Structure de MonteCarloAlgorithm

La classe MonteCarloAlgorithm dans notre projet utilise la méthode de Monte Carlo pour évaluer les meilleures actions possibles pour un bot jouant au jeu Acquire. Elle contient les éléments suivants:

- BotController botController: Un contrôleur qui gère l'état de la simulation.
- int numSimulations: Le nombre de simulations pour estimer l'efficacité des actions.

6.3.2 Méthode runMonteCarlo

La méthode runMonteCarlo simule chaque action possible et évalue son efficacité selon les étapes suivantes:

1. Initialisation d'une hashmap pour garder le score de chaque action.
2. Pour chaque action, répéter un nombre de fois défini par numSimulations:

- (a) Cloner le contrôleur du jeu pour ne pas affecter l'état réel.
 - (b) Exécuter l'action et simuler jusqu'à la fin du jeu.
 - (c) Calculer et accumuler le gain net du joueur.
3. Calculer le score moyen pour chaque action et choisir l'action avec le meilleur score.

6.3.3 Choix de la meilleure action

La méthode `chooseBestAction` examine les scores de toutes les actions testées et sélectionne celle qui maximise les retours.

6.4 Résultats obtenus

6.4.1 Validation du Concept

La mise en œuvre de la méthode de Monte Carlo dans le développement de l'IA pour le jeu Acquire a validé le concept de simulation aléatoire pour la prise de décision dans un environnement de jeu complexe. Cette approche a permis à l'IA de gérer efficacement de multiples scénarios et de réagir de manière adaptative aux stratégies des adversaires, démontrant la faisabilité et l'efficacité de l'intégration de techniques avancées de simulation dans les jeux de stratégie. De plus, la flexibilité de la méthode Monte Carlo permet d'ajuster le nombre de simulations selon les besoins spécifiques du moment, offrant ainsi la possibilité d'augmenter ou de diminuer l'intensité des calculs en fonction de la complexité de la situation ou des ressources disponibles, ce qui optimise les performances de l'IA tout en contrôlant les coûts computationnels.

6.4.2 Impacts stratégiques

L'utilisation de simulations de Monte Carlo a permis une meilleure anticipation des mouvements des adversaires et une planification stratégique plus approfondie. Cela a été particulièrement utile dans des situations de jeu complexes, où plusieurs joueurs interagissent et où les conséquences des actions peuvent être très variables.

6.4.3 Défis rencontrés

Bien que les résultats soient globalement positifs, plusieurs défis ont été rencontrés lors de l'implémentation. Parmi ceux-ci, la gestion de la complexité computationnelle et la nécessité d'optimiser le temps de calcul pour permettre une simulation en temps réel sans délai perceptible par les utilisateurs.

6.4.4 Perspectives futures

Les résultats obtenus ouvrent plusieurs perspectives pour l'avenir, notamment l'intégration de techniques avancées de réduction de variance et l'exploration de l'application de l'apprentissage automatique pour affiner davantage les décisions stratégiques du bot.

6.5 Pistes inexplorées

Bien que les résultats actuels soient prometteurs, il existe plusieurs axes d'amélioration:

- **Adaptation dynamique du nombre de simulations :** L'efficacité de la méthode de Monte Carlo peut être améliorée en adaptant dynamiquement le nombre de simulations en fonction de la complexité de la décision à prendre. Des décisions plus critiques pourraient bénéficier d'un nombre plus élevé de simulations pour assurer une analyse plus robuste, tandis que pour des décisions moins impactantes, le nombre pourrait être réduit pour économiser des ressources.

- **Utilisation de techniques de réduction de variance** : Des techniques telles que l'antithétique variates, le control variates, ou le stratified sampling pourraient être utilisées pour réduire la variance des estimations, ce qui augmenterait la précision des résultats avec le même nombre de simulations, ou atteindrait une précision donnée avec moins de simulations.
- **Exploitation du Monte Carlo Tree Search (MCTS)** : Le Monte Carlo Tree Search est une variante avancée qui construit un arbre de décision au fur et à mesure que les simulations progressent. Cette méthode est particulièrement adaptée aux jeux de stratégie et pourrait offrir une meilleure prise de décision en explorant plus systématiquement les conséquences futures des actions actuelles.

7 Conclusion

Ce fût notre troisième projet après celui de POO, néanmoins, il nous rendût notre humilité. En effet, nous avions oublié à quel point s'organiser était dûr. Sur ce projet, nous avions décidé de faire un effort d'organisation en suivant la méthodologie Scrum. Nous avons mis en place, en plus des réunions hebdomadaire, d'autres réunions régulières car nous avions souhaité miser sur la collaboration et la communication.

Ce projet nous a également introduit à de nouveaux objets et concepts, parfois intrinsèques à Java, mais pourtant inconnus. Contrairement aux projets du premier semestre, nous avons choisi d'aborder celui-ci en tant que développeur et non en tant qu'étudiant. Ceci impliquait d'utiliser au maximum des outils pré-existants, quitte à parfois devoir les améliorer.

En somme, nous considérons ce projet comme notre réel premier projet : nous étions libres de nos choix et de notre travail, nous avons tous tenté de donner le meilleur de nous-même et nous sommes fiers de vous le présenter.

8 Ouverture

Ce projet nous a effectivement ramené à l'humilité, nous avons à certains moment eu l'ambition d'implémenter certaines fonctionnalités mais par manque de temps et d'organisation nous avons été ramené à la réalité. C'est le cas par exemple du marché noir que nous voulions mettre en place. Il s'agissait d'intégrer un moyen alternatif d'obtenir des actions d'entreprises à un prix moins chère en exploitant le fait que les joueurs pourraient s'en échanger et en vendre entre eux. Cela aurait apporté plus de diversité dans les stratégies mises en place par les joueurs et ajouté du piment aux parties.

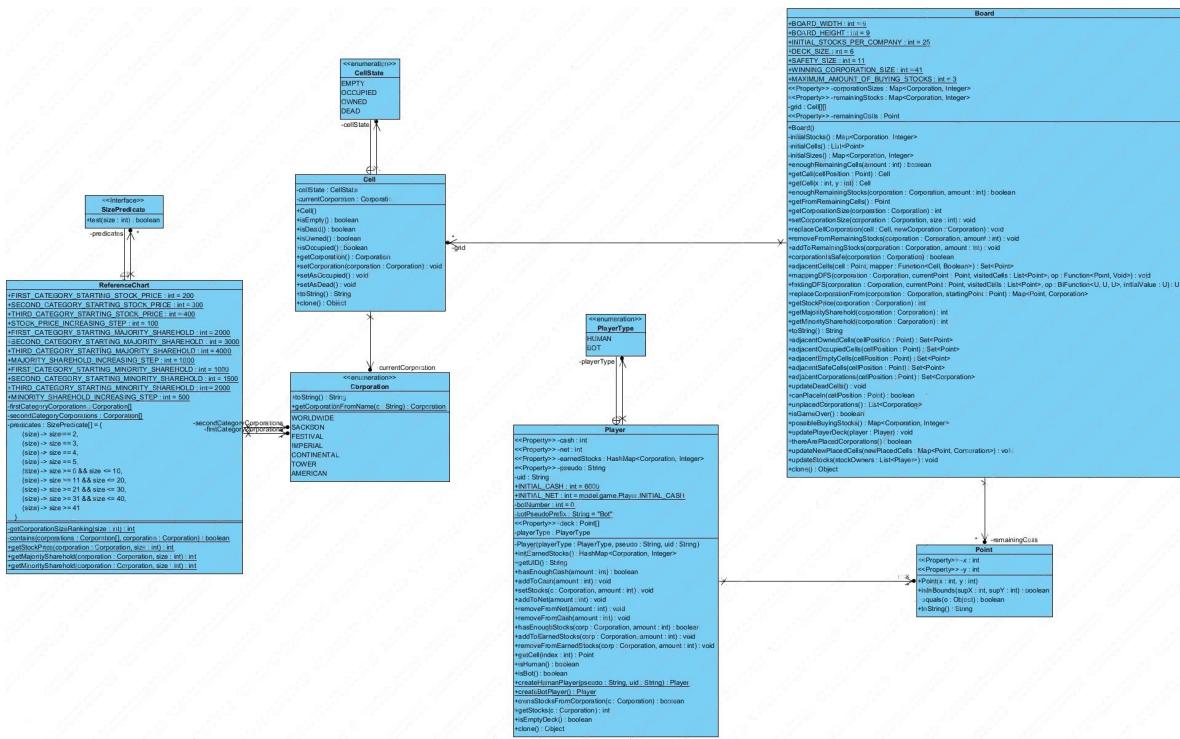


Diagramme UML de model.game

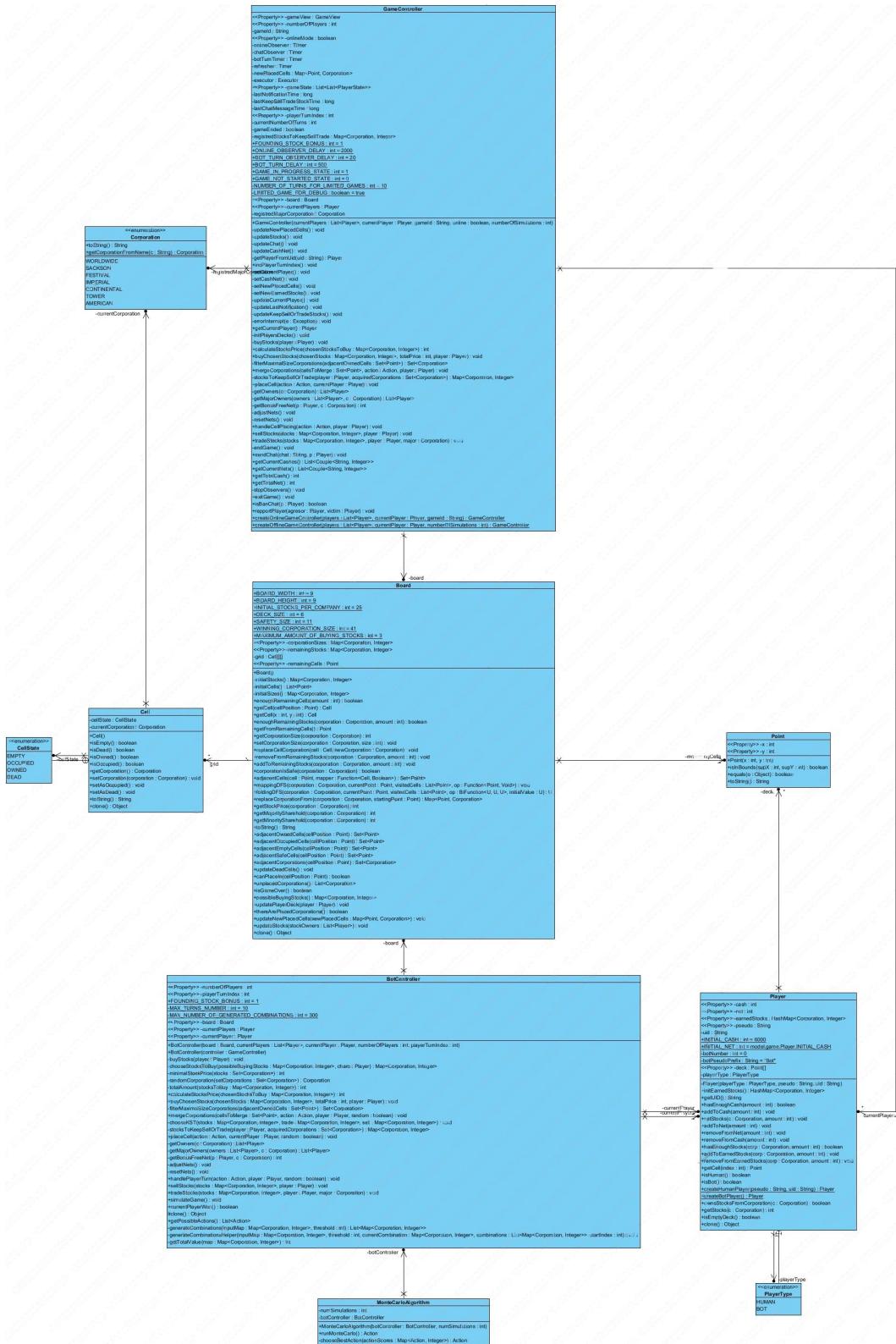


Diagramme UML de `control.game`

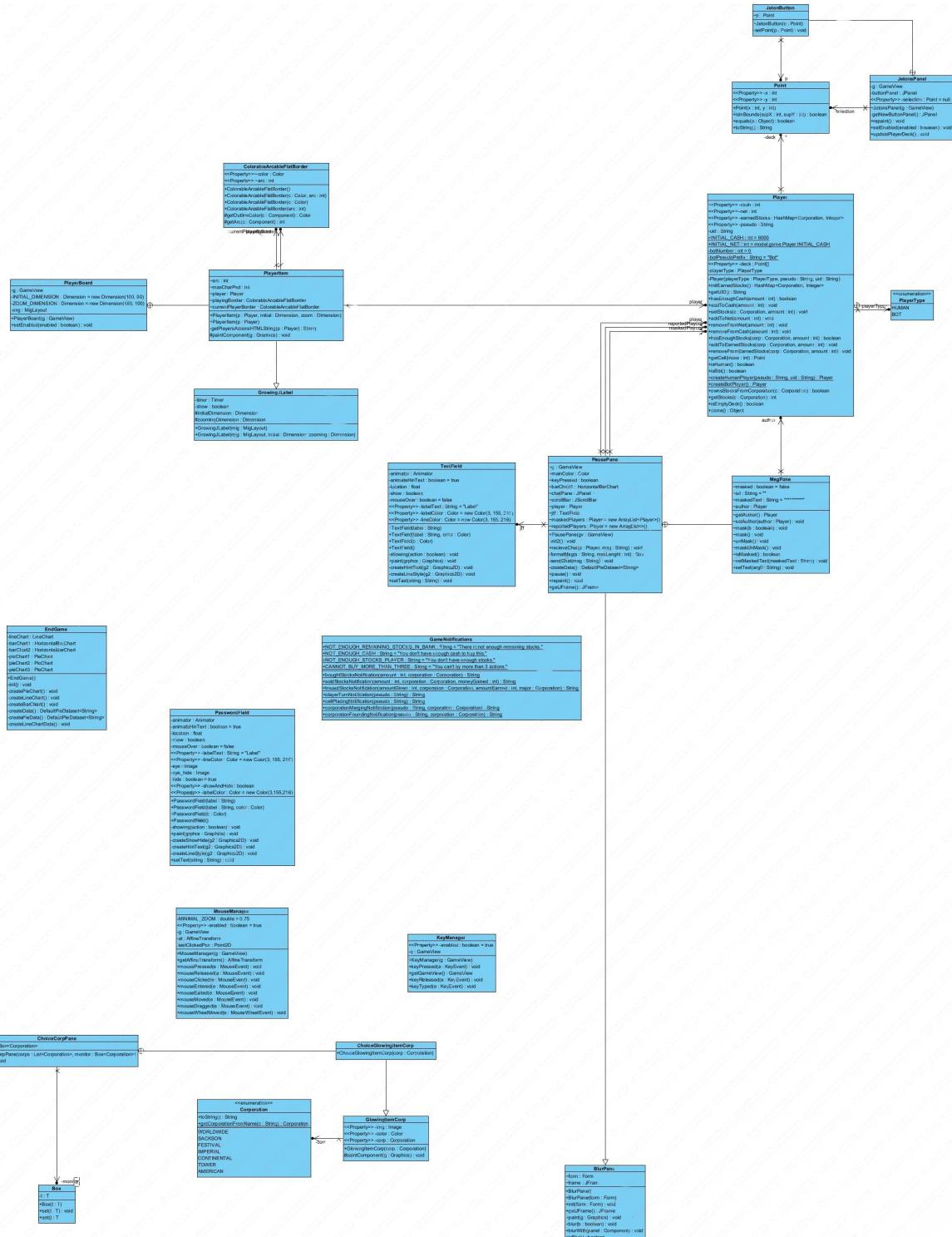


Diagramme UML de `view.game`