Perdaens Martin
Michotte Martin
Valentin Morgan

17/12/2019

RAPPORT DE PROJET : BATTLEGROUND

Table of Contents

Cahier des charges

Contexte du projet et objectifs :

Dans le cadre du cours de Développement informatique avancé orienté applications (Java), il nous est demandé de réaliser une application utilitaire ou bien un jeu qui se présente à la fois sous forme d'interface graphique et en ligne de commande. Ceux-ci doivent comporter une communication réseau ou une interaction avec une base de données ou éventuellement un service web.

Nous avons opté pour la réalisation d'un jeu vidéo de type 1 contre 1 en réseau se basant sur le jeu populaire « bataille navale ».

Dans notre version du jeu, le terrain de jeu n'est plus l'océan mais la terre ferme, les bateaux sont remplacés par des bâtiments et des véhicules.

Certains bâtiments et véhicules offrent des compétences supplémentaires au joueur tel qu'un raid aérien depuis un aéroport ou un tir de rockets depuis un véhicule lance-roquettes. Ces compétences spéciales ne sont évidemment pas accessibles à tout moment :

- Une fois utilisée, une compétence doit être « rechargée » (elle redeviendra disponible après un certain temps).
- Si le bâtiment/véhicule est détruit, la compétence liée à celui-ci n'est plus disponible.

Contraintes techniques :

⇨ Le jeu doit pouvoir être joué de 2 manières :
- Par le biais d'une interface en ligne de commande
- Par le biais d'une interface graphique

Attention, les deux joueurs ne doivent pas spécialement être sur la même interface.

⇨ Le jeu doit pouvoir être joué sur deux machines distinctes dans un réseau local.

⇨ Les bonus sont utilisables uniquement si l'unité associée est toujours « vivante ».

Déroulement d'une partie :

1. Les deux joueurs se connectent.
2. Les deux joueurs placent leurs unités sur leur terrain de jeu.
3. Une fois les deux joueurs prêts, le joueur A débute son tour.
4. Une fois que le joueur A a fini son attaque, le tour est donné au joueur B.
5. Le point 4 est répété jusqu'à ce qu'un des deux joueurs :
    a. n'ait plus d'unités.
    b. quitte la partie.

Description des besoins fonctionnels :

Nous décrivons ci-dessous les besoins fonctionnels du programme pour un joueur :

Spécificité générale :

Interface de jeu :

- 2 grilles de 13x13 cases
- Une grille sur laquelle le joueur place ses unités et voit les attaques infligées par son adversaire.
- Une grille sur laquelle le joueur peut placer ses propres attaques et en voir le résultat.

Unités :

Le joueur dispose de 6 unités (3 bâtiments et 3 véhicules) :

Unité	Taille	Bonus
Airport	2x4	Air-strike
Radar Tower	2x3	Radar-discovery
Headquarter	2x2	/
Railway Gun	1x6	Big-shot
MMRL[1]	2x2	Rocket-strike
Tank	1x2	/

Bonus :

Air-strike : Un avion largue des bombes sur 7 cases en ligne.

Radar-discovery : 4 cases adjacentes une à une peuvent être découverte.

Big-shot : Un obus de 800mm explose sur une case et détruit toutes les cases adjacentes.

Rocket-strike : 4 missiles tombent dans une zone de 7x7 de manière aléatoire.

[1] Mobile Multiple Rocket Launcher

Use Case[2] :

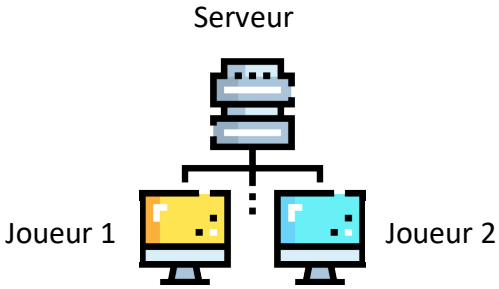

Figure 1: système de communication

Le serveur doit pouvoir :

1. Établir une connexion entre deux joueurs sur un même réseau.
2. Donner le tour à chacun des joueurs.
3. Recevoir les informations[3] du joueur qui a le tour.
4. Traiter les informations reçues.
5. Renvoyer des informations aux deux joueurs. (La case où le tir a été effectué, l'état d'une unité, …)

Le joueur doit pouvoir :

1. Se connecter au serveur.
2. Avant le commencement de la partie, placer ses unités sur la grille de jeu.
3. Visualiser les deux grilles de jeu :
    a. Sa grille contenant ses unités et leur état. (non touchée, touchée, détruite)
    b. La grille de l'adversaire avec la position de ses propres tires et l'état de ceux-ci. (touché, non touché) (cette grille est vierge au début de la partie)
4. Lorsque c'est son tour, tirer sur une case de la grille de son adversaire ou utiliser une attaque spéciale.
5. Lorsque ce n'est pas son tour, le joueur ne peut rien faire hormis observe l'attaque de son adversaire.
6. Quitter le jeu sans avoir terminé la partie. (L'adversaire aura alors gagné par abandon)

[2] Les numéros de puces indiquent l'ordre de priorité de la fonctionnalité
[3] Par informations on entend : la position des unités, la position des tirs, …

Diagramme UML du modèle

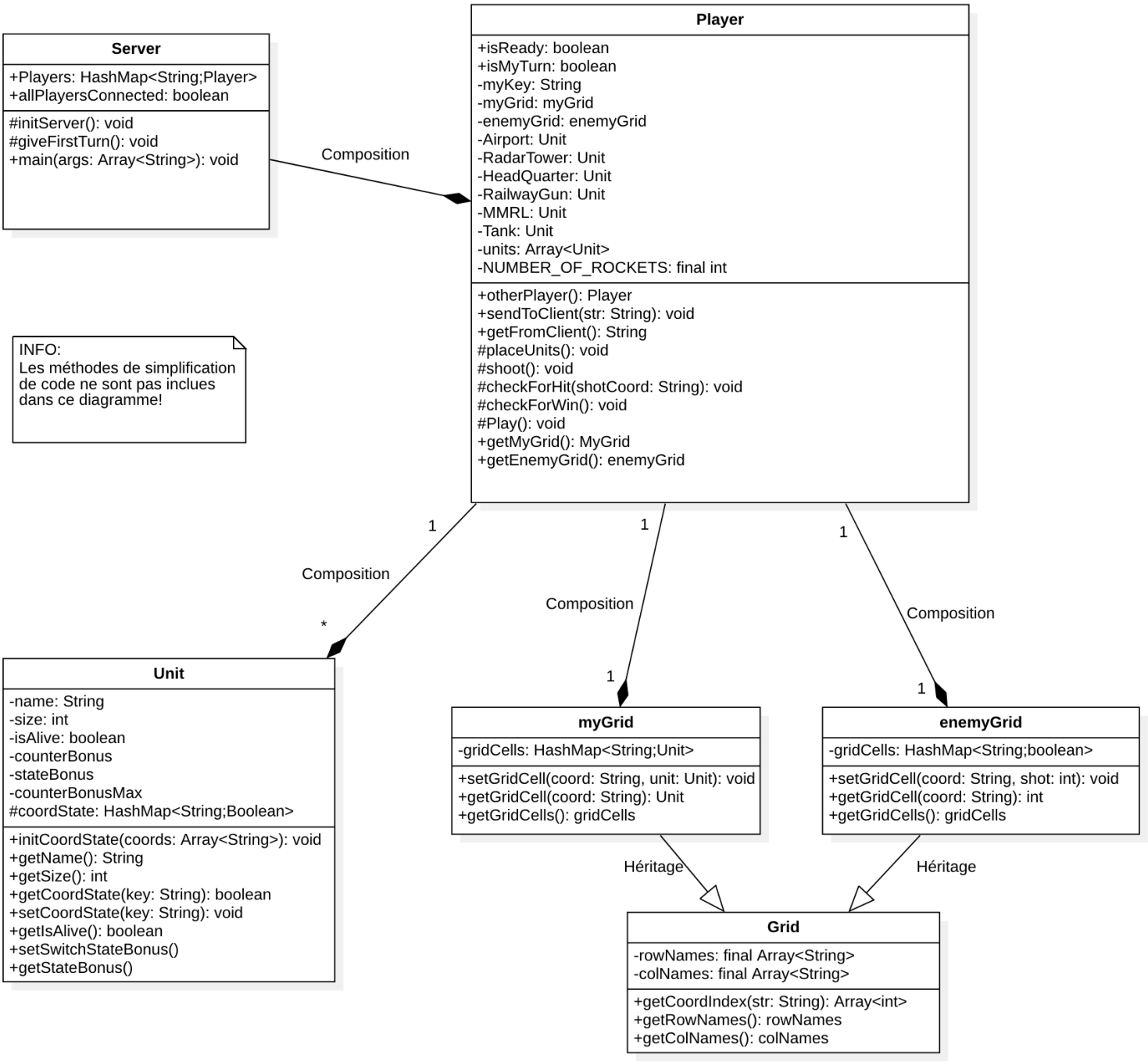

Figure 2 : Diagramme UML

Choix d'implémentation

Il nous est demandé d'implémenter l'une des trois possibilités suivantes à notre application :

- ✓ **Une communication réseau**,
- ❖ Une interaction avec une base de données,
- ❖ Ou un service web.

Nous avons opté pour une **communication réseau**, car notre objectif est de réaliser un jeu à 2 joueurs (du type 1 contre 1), se jouant sur 2 ordinateurs différents.

Nous nous sommes basés sur le modèle **serveur-client** pour comprendre comment tout fonctionne :

- ⇨ Le serveur attend les requêtes du client et lorsque celui-ci reçoit une requête, il répond.

Et puis nous avons fait en quelques sortes l'inverse :

- ⇨ C'est les clients qui attendront que le serveur envoie une information disant qui peut « parler ».
  - Dans notre jeu, les clients attendent que le serveur envoie un message « *c'est à toi de jouer* » ou « *ce n'est pas à toi jouer* ».
  - Et en fonction du message reçu, le client pourra envoyer des informations (choix du tir, envoie des coordonnées où l'on souhaite tirer, …)

Du coup, nos clients n'ont pas besoin d'avoir tous les fichiers du jeu, vu que tout le traitement se fait au niveau du serveur et pas au niveau du client.

- Le client envoie des informations au serveur (si le serveur lui a donnée son tour).
- Le serveur traite ces informations et les envoie au 2 clients.
  - Par exemple, lorsque joueur A tir sur une case de la grille de joueur B, l'information est envoyée au serveur.
  - Celui-ci envoie au joueur A s'il a touché ou non une case de la grille de joueur B (sous forme d'actualisation de la grille de jeu)
  - Le serveur envoie également au 2ème joueur (joueur B) des informations à propos du tir du joueur A (sous forme d'actualisation de la grille de jeu).

Game-Flow général

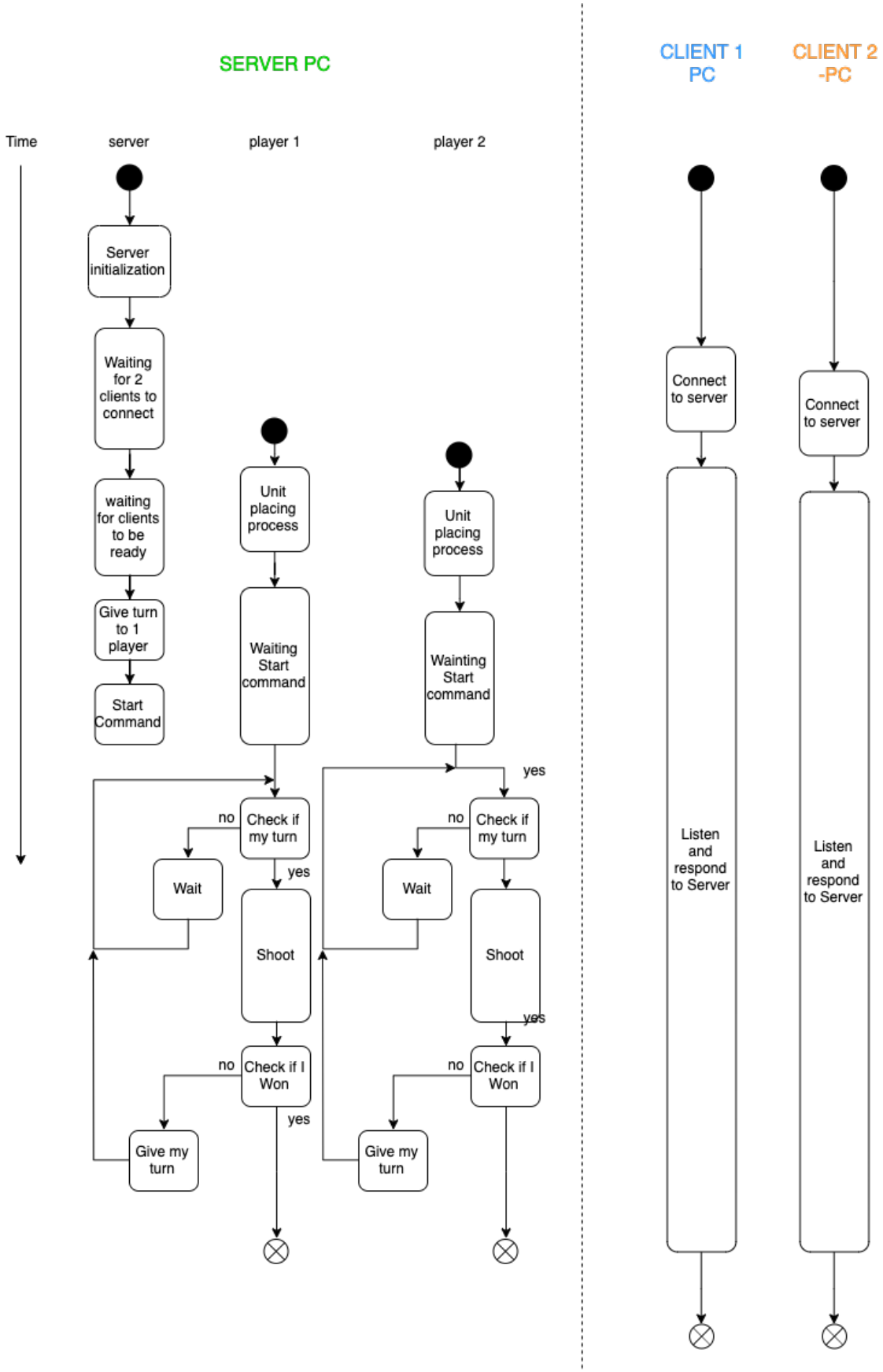

Figure 3: Game-flow

Difficultés rencontrées

1. Maitrise de l'outil « Git » :

Ce n'était pas une difficulté à proprement parlé, néanmoins nous avons dû y consacrer beaucoup de temps. Un cours ou des explications plus précises (sur le fonctionnement général de Git ainsi que la création et la fusion (merge) de branches) de la part d'un professeur aurait permis à tout le monde d'avoir une bonne base avant de se lancer simultanément dans un projet Java et dans l'utilisation de Git.

2. Implémentation de la communication Client-Serveur avec le modèle :

Une fois le diagramme UML fini, nous nous sommes penchés sur la problématique de la communication entre 2 PC. En faisant des recherches sur la communication client-serveur en Java, nous nous sommes aperçus que dans la vaste majorité des cas le serveur est dit « passif » tandis que le client est dit « actif ». En d'autres termes, le serveur est en attente d'une requête d'un client quelconque et y répond une fois la requête reçue et validée, tandis que le client initie une requête vers un serveur quand il en a besoin.

Contrairement à ce système de communication, notre jeu a besoin d'un serveur « actif » et de deux clients « passifs ». Dès lors c'est non pas le client qui initie une requête mais le serveur vers un ou deux clients.

Notre serveur a donc en permanence deux voies de communications ouvertes et par conséquent un protocole de communication complexe a dû être mis place.

3. Implémentation de l'architecture MVC :

Étant donné qu'aucun de nous avait déjà codé de manière sérieuse en Java, que notre projet était peut-être un peu trop ambitieux par rapport à nos connaissances et au temps disponible pour ce projet, nous avons essayé de nous avancer un maximum dès le début de celui-ci.

Nos plus grandes craintes étaient la réalisation du jeu en ligne de commande ainsi que l'implémentation de la communication client-serveur. Nous avons dès lors passé énormément de temps durant les premières semaines à essayer d'implémenter ces fonctionnalités.

Nous avons volontairement mis de côté l'implémentation de l'architecture MVC car celle-ci devait encore être vue au cours et nous pensions pouvoir facilement l'implémenter par-dessus notre code existant. Ce fut une fameuse erreur !

En effet, nous avons eu beaucoup de mal à implémenter l'architecture MVC par-dessus notre code. Nous avons dû complètement changer la façon dont le serveur interagissait avec le client pour au final avoir une structure MVC partielle.

Nous considérons notre structure MVC comme étant partielle car nous n'avons pas réussi à obtenir la synchronisation des vues GUI et cmd-line chez le client.[4]

Pourquoi ?

Une structure MVC repose sur la figure suivante :

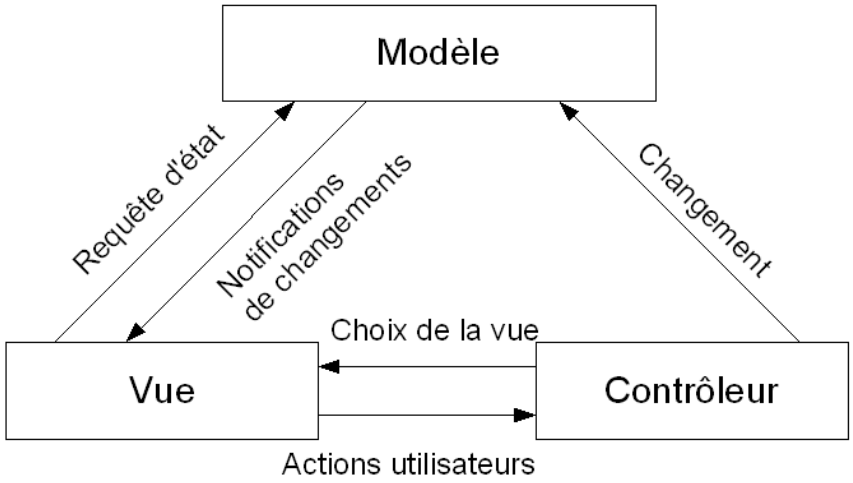

Figure 4 MVC structure

https://www.supinfo.com/articles/single/8729-architecture-mvc-qu-est-ce-que-c-est

Cela fonctionne très bien dans la plupart des cas car le modèle, les vues et le contrôleur se trouvent sur un même PC. Dans notre cas, les vues se trouvent sur le PC des clients et n'ont donc aucun lien direct avec le modèle ou le contrôleur ! La seule manière d'avoir des informations depuis et vers les vues est d'utiliser le protocole de communication client-serveur mis en place et cela ne permet pas une conception standard d'une structure MVC.

Nous avons donc mis en place un structure MVC du côté du serveur avec :

- Le modèle : toute la partie logique du jeu, envoie des données au clients.
- Le contrôleur : réceptionne les données du client, les vérifie et les transmet au modèle si elles sont valides, si non, renvoi la requête au client.
- La vue : n'est pas une vue au sens propre du terme mais plutôt une passerelle vers les vues du coté client. Cette vue est appelée lorsqu'un changement du modèle a lieu. Elle regarde ce qui a été modifié dans le modèle et envoie des instructions aux clients.

[4] Bien que pas explicitement demandé dans les consignes du projet, nous aurions aimé que les vues soient synchronisées.

Solution possible :

Nous avons réfléchi à une solution possible afin de synchroniser plusieurs vues du coté client, néanmoins par manque de temps nous ne l'avons pas implémentée.

Solution en 3 étapes dans le code coté client :

1. Dissocier la connexion au serveur d'une vue :

   Actuellement lorsqu'une vue est lancée, celle-ci demande à l'utilisateur de se connecter au serveur. Dans le processus de connexion le serveur va associer un client/utilisateur à un joueur[5]. Si un même utilisateur lance plusieurs vues, elles seront toutes perçues comme étant des nouveaux clients et donc joueurs par le serveur.

   Il faudrait donc commencer par demander à l'utilisateur de se connecter (par le biais d'une interface dédiée en GUI ou ligne de Cmd) et ensuite lui proposer de lancer une ou plusieurs vues.

2. Créer un modèle fictif qui pourra être observé par de multiples vues :

   Le client n'ayant toujours pas accès au modèle en tant que tel, il faudrait créer un pseudo modèle qui ne reprendrait que les informations critiques à l'affichage des données. Celui-ci serait le seul à communiquer avec le serveur par le biais d'un nouveau protocole de communication plus complet.

3. Convertir les vues en observateurs et ajouter un contrôleur :

   Cette étape-ci revient à concevoir une structure MVC classique où les vues observent le modèle et se mettent à jour en fonction des changements de celui-ci.

   Lorsque l'utilisateur doit interagir, les données sont envoyées directement au contrôleur du serveur tout comme nous le faisons actuellement.

[5] Le serveur va créer une instance de la classe "Player" et l'associer à un socket.

Pistes d'amélioration

- Revoir complètement la structure de notre architecture MVC

(explication voir point précédent)

- Ajouter le bonus « radar-discovery »

Une des armes/bonus que nous n'avons pas implémenté par manque de temps est le bonus « radar-discovery » qui permettrait à un joueur de connaitre l'emplacement des unités ennemies dans une zone délimitée.

Nous en avons néanmoins tenu compte lors de la réalisation du code :

- ⇨ Dans le **GUI** elle est grisée => bouton désactivé mais présent.
- ⇨ Dans la **ligne de commande** => commentée mais présent.

- Amélioration du système de positionnement des unités

Il n'est actuellement pas des plus évidents de placer les unités dans la grille de jeu. Bien qu'avec le temps l'utilisateur commence à s'y habituer, une amélioration conséquente est indispensable ! Nous avions pensé aux solutions suivantes :

- ⇨ En GUI : système de drag&drop + rotate.
- ⇨ En Cmd-line : une coordonnée à entrer suivie d'une direction.

- Amélioration de la gestion des tirs

Afin de simplifier le code et de nous concentrer sur le projet au complet, nous avons volontairement omis deux parties de l'implémentations des tirs :

1. Tir partiellement en dehors de la grille :
   Un tir ne peut jamais être introduit à l'extérieur de la grille, néanmoins, lors de l'utilisation de bonus tel que le « Airstrike » ou le « Big-shot », certains des tirs peuvent tomber à l'extérieur de la grille.

2. Tir sur une case qui a déjà été tirée :
   Pour le moment, il est tout à fait possible de tirer sur une case ayant déjà été tirée.

La gestion des tirs devrait donc être revue afin de résoudre ces « problèmes ».

- Gestion d'une fermeture d'un client/serveur en pleine partie

Même si actuellement nous avons essayé de quitter proprement les jeux lorsque le serveur plante ou qu'un des clients ferme sa fenêtre, aucune information n'est donnée au client. Il serait donc utile de prévoir un protocole de fermeture en cas d'erreur critique.

Conclusion individuelle

Martin Perdaens

En ayant jamais programmé en Java, j'avais des craintes sur la manière de coder le projet. Au début je ne voyais pas comment faire mais par la suite avec les cours et le fait de chercher sur internet cela m'a permis de mieux comprendre.

L'idée de faire un projet en groupe était chouette pour approfondir notre compréhension du langage mais j'aurais aimé avoir vu toute la matière avant de commencer le projet.

Morgan Valentin

Étant débutant en Java, j'appréhendais un peu le langage. Cependant, ce projet m'a vraiment poussé à rentrer dans le langage et chaque difficulté que j'ai rencontrée m'a permis d'aller plus loin dans la matière.

Ce projet était une très bonne idée malgré que j'aurais aimé avoir aucun cours pratique et juste le projet **ou bien** un quadrimestre où l'on apprend le langage (comme pour le cours de programmation de 1ère année) et le 2ème quadrimestre où l'on a un projet.

Martin Michotte

Étant en « année passerelle » il n'a pas toujours été facile de combiner ce projet avec les autres cours de 1ere et de 2ème année, néanmoins, après plus d'un mois et demi de travail je pense que nous sommes arrivés à ± 95% de ce que nous nous étions fixés dans le cahier de charges. Les 5 derniers pourcents auraient très certainement pu être réalisés si nous avions encore quelques semaines devant nous. Le jeu est cependant loin d'être parfait et une grande quantité de choses peuvent encore être améliorées et/ou re-factorisées afin de rendre le jeu plus fiable et attrayant.

Je pense par ailleurs, tout comme mes collègues, qu'il aurait été préférable de d'abord voir toute la matière du cours avant de commencer ce projet. Nous avons « malheureusement » voulu prendre de l'avance et nous l'avons payé cher en devant repenser complètement notre implémentation du jeu à plusieurs reprises. Par contre, si nous n'avions pas pris de l'avance, je ne pense pas que nous aurions réussi à terminer le jeu à temps vu sa complexité.

Je me pose dès lors les questions suivantes :

- Notre projet de jeu était-il trop ambitieux ?
- Avons-nous suffisamment réfléchi à la conception de la logique du jeu avant de coder ?
- Avons-nous correctement réparti les tâches ?