Projet PCII

Membres du groupe pour ce projet:

- COMBEAU Thomas
- LASLUISA NUNEZ Charlies
- SIVACOUMAR Arsène

I- Introduction

Nous avons pour but de réaliser un jeu en temps réel qui se base sur le jeu Fourmizz. L'objectif de ce projet est de réaliser un jeu de la forme 1 vs IA, où chaque joueur possède une liste d'Unités et de Bâtiments. Chaque joueur possède différents types de ressources et le gagnant sera désigné quand l'un des 2 joueurs atteindra une limite de ressource (un système de course à la ressource).

Règles de ce jeu :

- Le joueur commence la partie avec une fourmilière qui lui sert de bâtiment principal (càd que c'est l'endroit où se stockera les ressources récoltées par les fourmis ouvrières.

- Le joueur commence avec un nombre de ressources basique afin que chacun puisse débuter le jeu sans soucis.

- Afin de pouvoir créer des fourmis combattantes, il faudra pour cela implanter un nouveau bâtiment (nommé Caserne) sur la grille de jeu. Mais avant de pouvoir ces futures fourmis, il faudra avant tout avoir les ressources nécessaires.

- Pour la récolte des ressources, le joueur peut cliquer sur une ouvrière et lui ordonner d'aller récolter des ressources sur la case indiquée par le joueur. Lorsque la fourmi commence à récolter les ressources, le compteur de ressources du joueur augmente.

- Au cours d'une expédition pour récolter des ressources, le joueur peut décider d'envoyer, en plus des ouvrières, un certain nombre de fourmis combattantes afin d'assurer une certaine sécurité. Néanmoins, il se peut que lors de cette même expédition, des fourmis combattantes ennemies viennent tout interrompre et fassent échouer cette expédition en repartant avec ces mêmes ressources.

- Lors d'un duel entre fourmis ennemies, en fonction des unités de combats, la fourmi gagnante se fera en fonction de plusieurs paramètres (coup d'attaque et de défense)

Comme annoncé précédemment, pour pouvoir gagner la partie, chaque joueur doit récolter un maximum de ressources. Dans notre jeu l'adversaire correspond à l'environnement qui défend ses ressources en empêchant le joueur de les récupérer, pour cela il génère des unités qui vont attaquer les unités du joueur.

Dans un premier temps, nous avons réalisé un espace de travail respectant le MVC afin de pouvoir programmer sans avoir une quelconque erreur lors de l'affichage ou le changement d'état. De plus, nous y avons inclus des classes abstraites comme Unités qui sera utilisé afin de potentiellement créer différentes unités.

Dans un deuxième temps, nous nous sommes séparés le travail afin de pouvoir exécuter une première version de l'espace de jeu ainsi qu'un prototype d'IA naïve afin d'avoir une idée de comment représenter tous nos éléments un à un et d'ainsi définir de nouvelles classes comme Joueur, Bâtiments…

II- Analyse globale

Dans cette partie, je vais expliquer globalement les différentes fonctionnalités qu'on a implémenté, on y reviendra dans le détail dans une partie qui suit.

La première chose que nous avons implémenté ce fut évidemment le plateau de jeu mais je ne passerai pas beaucoup de temps dessus car ce n'est pas une fonctionnalité importante, il faut juste prendre en compte qu'elle est implémentée de sorte à ce que chaque case est un JPanel, ce qui permet de de changer le contenu de la case assez facilement. Les 9 cases en bas à gauche constituent la base du joueur, elles contiennent les bâtiments et ce sont là où apparaissent les unités du joueur. La case en haut à droite est l'emplacement où sont générées les unités de l'environnement.

L'une des premières fonctionnalités que nous avons implémentées c'est le fait de mettre en place nos unités pour le joueur et l'environnement. Notre jeu contient 3 unités différentes. La première est l'ouvrière, elle permet au joueur de récupérer les ressources placées aléatoirement sur le plateau de jeu. La deuxième est la combattante, elle permet de défendre les ouvrières pour qu'elles puissent continuer de récupérer les ressources. La dernière unité est la combattanteAI, elle a les mêmes caractéristiques que la combattante mais elle attaque les ouvrières et les combattantes pour empêcher le joueur de récupérer les ressources sur le plateau. Chaque unité est implémentée sous forme de Thread qui permet de réguler le déplacement de chaque unité selon un temps donné. L'implémentation des actions des unités est gérée par d'autres Threads. Les déplacements des unités pour le joueur suivent une implémentation sur plusieurs classes : dans un premier temps il faut que le joueur effectue un clic droit sur l'unité qu'il souhaite déplacer puis faire un

clic gauche vers la case vers où il veut se déplacer, on rentrera dans les détails dans une partie plus détaillés.

Ensuite, nous nous sommes intéressés aux ressources. Nous avons implémenté les ressources de façon à ce qu'il y ait un certain nombre de ressources placées aléatoirement dès le lancement du jeu et que le nombre de ressources ne baissent pas en dessous 60 de unités. Les ressources jouent un rôle primordial dans notre jeu, elles permettent au joueur d'établir une stratégie de jeu pour gagner. En effet nous mettons sur le plateau deux types de ressources : de la nourriture (représenté par l'image de miel) qui permet au joueur de générer des ouvrières pour récupérer pour récupérer des ressources plus rapidement, elles permettent aussi de gagner la partie à partir d'un certain nombre de nourritures récupérées, enfin nous avons aussi placés du bois sur le plateau, le bois permet de générer des combattantes pour attaquer les unités de l'environnement qui attaquent les ouvrières, cependant le prix d'une combattante est élevée.

Le joueur doit donc établir une stratégie de jeu pour lui permettre de remporter la partie, la gestion de ses ressources sera l'un des points les plus cruciaux. Les ressources sont gérées grâce à un Thread qui regarde le nombre de ressources sur le terrain pour en rajouter au cas où il serait inférieur à une limite fixée.

Une autre des fonctionnalités importantes ce sont les bâtiments, ils sont liés à la gestion des ressources et sont au nombre de deux : la fourmilière et la caserne. La fourmilière permet de générer des ouvrières contre un prix en nourriture, ils sont placés dans la base du joueur dans les cases en bas à gauche. On peut générer une troupe en cliquant tout simplement sur l'image du bâtiment mais seulement si le joueur a un nombre suffisant de ressources.

Le mécanisme de défense des ressources sur le plateau est ce qui met de la difficulté dans le jeu, c'est donc une fonctionnalité très importante dans notre jeu. Dans notre implémentation, elle prend la forme d'une intelligence artificielle qui possède des unités qui vont attaquer les unités du joueur. La classe de l'intelligence artificielle est simplement un Thread qui donne à chacune de ses unités, une unité du joueur à aller attaquer, il vérifie s'il faut redonner une unité à aller attaquer à une de ses unités au cas où. Chacune de ses unités est aussi un Thread qui gère leurs propres déplacements.

III- Plan de développement

En ce qui concerne le développement de notre jeu nous n'avons pas suivi l'ordre des fonctionnalités décrites précédemment en ordre d'implémentation. Chaque fonctionnalité a eu besoin d'ajout et de finition, de plus chacune des grandes fonctionnalités fut implémentée par quelqu'un du groupe précisément : c'est la façon dont nous nous sommes partagé le travail. Donc chaque fonctionnalité a pris un temps d'implémentation qu'ils nous aient difficile de donner précisément. Cependant décrire l'ordre, faire les sous-fonctionnalités et le temps approximatif avec les recherches effectuées, sont ce que nous allons développer dans cette partie.

Dans un premier temps, la conception du jeu est sûrement ce qui nous a pris le plus de temps. Nous n'avions pas forcément d'idée et avions l'habitude d'avoir un sujet donné. Un membre de notre groupe a alors eu l'idée de nous inspirer du jeu de Fourmizz qui est un jeu en ligne. La conception de notre version de ce jeu est ce qui nous a fait effacer beaucoup de lignes de code et d'idées. Nous en sommes alors à un début de structure avec une grille de cases qui sont chacune des JPanel, pour représenter notre plateau de jeu. Pour en arriver jusqu'ici il s'était déjà écoulé quelques semaines. Mais nous avions une idée plus claire de ce que nous voulions faire et en même temps on avait déjà une bonne structure dans nos classes avec les différents types d'unités, de bâtiments, les ressources, l'environnement, et les classes liées au MVC. En effet, on a essayé de respecter le plus possible le modèle MVC tout au long de notre projet pour que chaque membre de notre groupe se retrouve facilement dans la structure du groupe ou pour toute personne qui regarde notre code. La classe Etat concentre la majorité des changements des états du jeu par exemple, mais on reviendra sur ça dans la conception détaillée de notre projet. Une première version de la structure du workspace a été proposée par Thomas mais une version largement retravaillée et plus propre a été proposée par Charlies.

Pour la suite on s'est donné chacun une des fonctionnalités importantes à implémenter : Arsène s'est occupé des ressources, Thomas des unités et de l'AI (de l'environnement et de ses unités) et Charlies des bâtiments.

En ce qui concerne les ressources, il a fallu trouver un moyen de générer les ressources aléatoirement au lancement du jeu et maintenir un certain nombre de ressources minimale tout le long de la partie. Ce fut le premier Thread de notre jeu que nous avons implémenté. Grâce au tutoriel fait précédemment on n'a pas eu besoin d'effectuer des recherches, nous savions déjà comment utiliser les Thread. Pour le tout début, on génère entre 40 à 60 ressources répartis aléatoirement sur les cases du plateau de jeu sauf sur les cases qui servent de bases pour le joueur et la case en haut à droite qui sert à générer des combattanteAI pour l'environnement.

Les ressources sont définies en deux types comme dit dans la partie précédente : bois et nourriture (avec image de miel). On génère soit l'un soit l'autre. La nourriture permet d'acheter des ouvrières et le bois des combattantes, la nourriture est ce qui permet de gagner à force d'en cumuler. L'ensemble de l'implémentation du Thread dans la classe Etat qui génère des ressources et la classe Ressource ont pris environ deux semaines plusieurs heures (répartis sur plusieurs jours). La conception et la documentation ont pris plus de temps (surtout la conception). Le temps total est environ compris entre 15h et 20h.

Pour les unités, on a rencontré beaucoup de difficultés lors de l'implémentation du déplacement au niveau graphique. Tout d'abord, on a juste créé une classe abstraite unité qui nous permet ensuite de créer les différentes unités possibles, puis l'ouvrier en premier. L'affichage graphique sur une case ne prend que peu de temps à faire (comme pour les ressources). A la suite de cela, on a bloqué assez longtemps sur le déplacement de l'unité car comme les unités sont des images, il faut l'effacer sur les cases qu'il quitte et le refaire apparaître sur la nouvelle, en prenant en compte qu'il doit apparaître sur toutes les cases entre sa case initiale et la case finale (pas de téléportation). Nous avons donc pensé forcément à un Thread, toutes les secondes l'unité va donc se déplacer d'une case jusqu'à la position finale.

Cependant, beaucoup de temps a été perdu pour comprendre comment stocker une unité et ensuite stocker une position finale où il doit aller. Avec de l'aide des différents membres du groupe et beaucoup de réflexion, on a finalement trouver : quand on crée une case de jeu sur le plateau au lancement du jeu, on enregistre pour chaque case sa position géographique, quand on clique droit sur une case pour dire qu'on veut déplacer une unité, le programme enregistre dans une variable dans la classe Etat une position initiale (le programme sait où vous cliquez grâce car chaque case connaît sa position géographique) ; ensuite par un clic gauche, de la même façon le programme enregistre une destination finale dans une variable puis fait appel à une méthode qui donne à l'unité positionné à la position initiale, une position où elle doit aller, le Thread dans chaque unité fait changer la position de l'unité quand elle possède une destination, puis après graphiquement on a juste à afficher dans la case où se trouve l'unité et à effacer où elle était précédemment.

Le déplacement des combattantes de l'environnement est différent, il est automatique. Il utilise la même idée avec la position initiale et la destination finale mais la destination finale est donnée grâce au Thread dans la classe AIPlayer (l'environnement) qui donne à ses unités une cible qui est une unité du joueur à aller attaquer, et en redonne une si une de ses unités à tuer sa cible, la destination finale est alors la position de la cible. Grâce à cela, on a implémenté un déplacement un peu intelligent.

Comme l'ouvrière est la première unité qu'on a créée, on s'est concentré ensuite sur comment ramasser des ressources. Dans la même idée que le Thread Ressource dans la classe Etat, on utilise un Thread pour gérer les actions des unités et aussi l'actualisation de leurs positions. Dans ce Thread on regarde si une ouvrière est sur la même position qu'une ressource et si c'est le cas alors on retire la ressource et dans le nombre de nourriture ou de bois du joueur on augmente de 1.

Quand on a ensuite créé la combattante comme unité puis la combattanteAI, il a fallu implémenter l'attaque. On a alors ajouté une vie à chaque unité et une attaque pour chaque combattant et combattanteAI. Les attaques des combattantes du joueurs vers les combattantes de l'environnement et inversement sont gérés par deux Threads indépendants. Si une unité adverse se trouve sur la même case ou sur une case autour de l'unité alors elle va l'attaquer, quand la vie d'une unité tombe à 0, elle disparaît. Ces Threads ont principalement été fait par Arsène et Thomas a un peu aidé. L'ensemble de la conception, documentation et implémentation a dépassé les 20H sur tout ce qui est lié aux unités, et ce fut la partie la plus difficile à mettre en place avec notamment le plus de bug à régler.

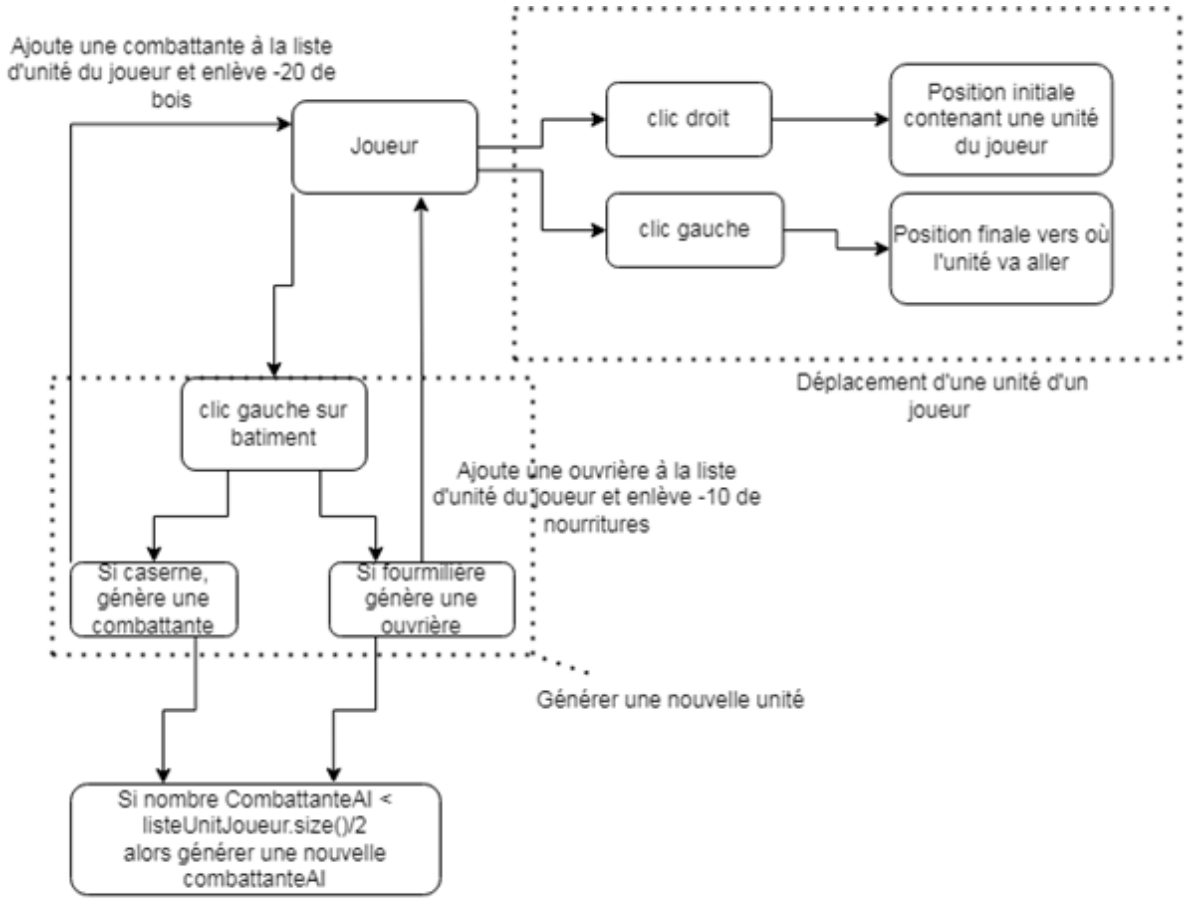
Ensuite par rapport au bâtiment, on a eu deux versions, le premier but était de générer des unités et mettre un Timer qui régulait l'apparition pour pas qu'on puisse générer plusieurs troupes d'affilés par exemple. Malgré la partie interne finie et graphique pratiquement finie, à cause d'un problème de santé, le membre s'occupant de cette partie n'a pas pu continuer et le reste du groupe a repris en main cette partie, cependant vu le temps qu'il nous restait on a grandement simplifié les choses. Les deux idées liées aux bâtiments seront cependant développées dans la partie de la conception détaillée pour laisser une trace de l'ensemble du travail effectué. Dans notre jeu, les bâtiments correspondent aux cases avec les images ressemblant à des fourmilières.

Quand le joueur fait un clic gauche dessus, il génère alors une unité selon le bâtiment sur lequel il a cliqué. La fourmilière génère une ouvrière contre de la nourriture et la caserne génère une combattante contre du bois (il faut évidemment en avoir assez, par exemple la combattante coûte 20 bois et le joueur en démarre avec 100). L'ensemble de la documentation, conception et implémentation pour la première version des bâtiments est d'environ 8h. Et la version faite rapidement par Thomas est d'environ 2h.

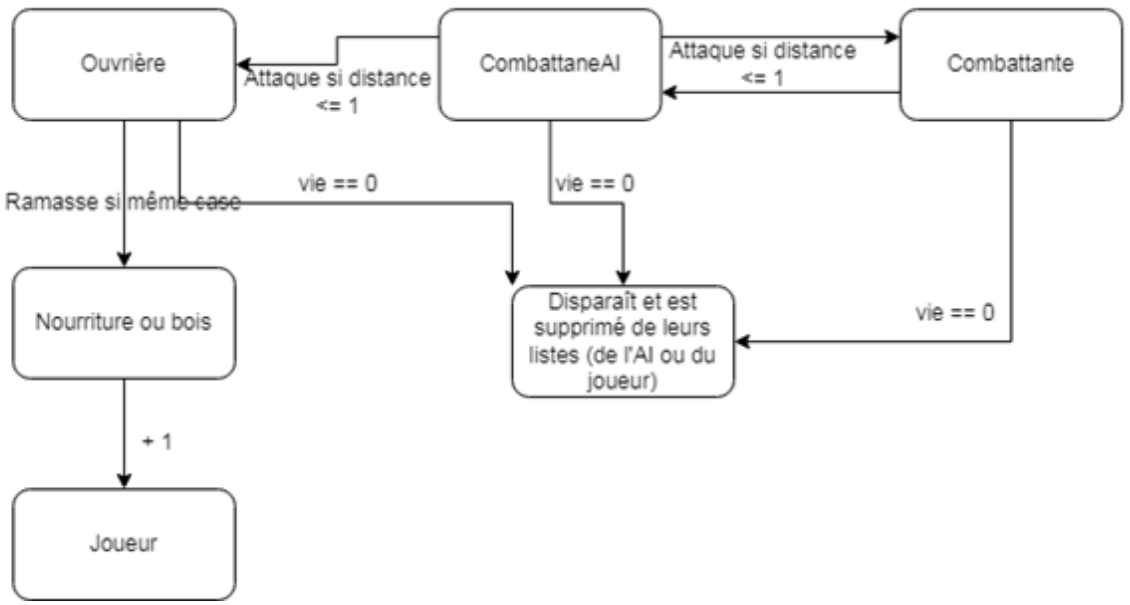
Enfin, dans le Thread des unités dans la classe Etat, comme dit dans la partie précédente, il y a deux conditions qui vérifient si le joueur a gagné ou perdu. Pour gagner il faut un faut que le joueur gagne un certain nombre de nourriture qu'on peut changer selon la difficulté qu'on veut donner au jeu. Pour perdre il faut que le joueur n'ait plus d'unités et qu'il n'ait pas assez pour s'acheter des nouvelles unités avec

les bâtiments. Les deux méthodes win() and lose() génèrent une fenêtre pour annoncer respectivement la victoire et la défaite du joueur.

IV- Conception générale



Fonctionnalités automatiques



V- Conception détaillé

Partie de Combeau Thomas :

Comme pour les autres membres de notre groupe, j'ai surtout eu des problèmes sur la façon de concevoir ce projet dès le début. J'ai au début du projet créer un « workspace » respectant le MVC pour que tout le monde puisse éviter de se perdre. J'ai donc créé l'ensemble des classes de départ que ce soit Etat, Affichage, Contrôle ou encore la grille. La première grille que j'ai proposée était simplement des lignes dessinées donc ils étaient très dur de placer et déplacer des choses à l'intérieur. Un membre du groupe a alors retravaillé le workspace et proposé une façon de faire la grille telle que chaque case était une JPanel et a aussi retravaillé le lien entre certaines classes. J'ai pris beaucoup de temps pour m'habituer à son code alors qu'il était beaucoup plus simple que le mien pour la compréhension. Ce projet m'a permis de travailler sur mes lacunes de travail en groupe et j'ai apprécié pouvoir compter sur le code des membres de mon groupe et aussi sur leurs aides quand je bloquais sur les parties que je devais faire.

Dans ma partie je vais parler précisément du code que j'ai fait qui est principalement sur les unités et l'environnement avec l'AI.

Dans un premier temps, je vais parler de la classe abstraite Unites et des classes qui héritent de cette classe. La classe Unite est donc une classe abstraite qui sert surtout pour la structure de notre projet dans notre workspace, elle permet aussi de mettre un certain nombre de méthodes qui sont communes entre toutes les différentes unités : récupérer la vie de l'unité, récupérer la position de l'unité et mettre une vie à l'unité. Et aussi la méthode run() car Unite hérite de Thread, ce qui permet à ses classes enfants d'être aussi un Thread.

Dans la première classe Ouvrier qu'on a implémenté (qui hérite donc de Unite), on possède les méthodes et fonctions décrites dans la classe précédente. La classe Ouvrier possède en attribut sa position, une position finale qui correspond à la position qu'il doit atteindre et un integer pour sa vie. Ce qui nous intéresse réellement dans cette classe est la méthode run(). Le run de chaque unité possède la même implémentation, elle permet de changer la position de l'unité quand elle possède une position finale. Dedans il y a 8 conditions qui permettent de regarder dans tous les sens pour voir si la position courante de l'unité est où par rapport à la position finale, et toutes les 1,5 secondes elle s'avance d'une case vers la position finale.

L'unité Combattante suit le même cheminement mais possède une attaque, mais l'auteur développe plus cette classe, j'ai cependant implémenté le déplacement dans cette classe aussi comme c'est le même code. De mon côté je vais parler de la dernière unité : la CombattanteAI. C'est une classe qui est plus détaillée que ses sœurs. Déjà on peut remarquer que comme la Combattante elle possède une

attaque, une vie, une position et une position finale à atteindre. Mais elle possède aussi un integer qui correspond à l'ID d'un ennemi et la position de l'ennemi. La notion de l'id je la développerais quand je parlerais de la classe AIPlayer mais pour l'instant il faut juste comprendre que cet ID correspond à une unité du joueur. Aussi une des spécificités de cette unité, c'est que c'est une unité jouée seulement par l'environnement de jeu contre le joueur. La méthode run() dans cette classe est la même que ses sœurs à une condition près. La position finale peut être nulle quand le Thread est lancé comparé à Ouvrier et Combattante. La classe CombattanteAI possède quelques méthodes de plus que Ouvrier : getEnemy pour avoir l'id de l'ennemie que notre unité vise, setEnemy pour donner un ennemi à aller viser à notre unité, c'est aussi ici qu'on lui donne la position finale pour la faire bouger, et setID si besoin de modifier juste l'ID sans modifier une position.

Maintenant je vais parler de la classe AIPlayer qui permet de s'occuper de l'environnement de jeu en gérant les unités combattanteAI sur le plateau de jeu. Cette classe est donc aussi un Thread. Ses attributs sont une liste d'unités de combattanteAI et un Etat, ce qui permet d'effectuer des actions. Dans son constructeur, on établit le lien avec l'Etat qu'on veut.

Cette classe possède une méthode pour récupérer sa liste d'unités. Mais comme pour la plupart des classes avec une Thread dans notre projet, la méthode qui a une utilité c'est run(). Dans le run(), on récupère d'abord la liste d'unités du joueur grâce à l'Etat qu'on a en attribut. Ensuite, pour chaque unité de la liste de AIPlayer on attribue un nombre aléatoire en ID comme ennemie à attaquer. L'ID est donc un nombre aléatoire défini entre 0 et la taille de la liste des unités du joueur. Cela me permet de gérer le changement d'ennemi à aller attaquer : si l'ID est supérieur à la taille de la liste des unités du joueur alors l'unité a éliminé son ennemie et il faut donc en définir un en utilisant la même méthode d'attribution.

Une fois que l'ennemi est attribué il faut pouvoir mettre à jour la position finale des combattanteAI, donc quand une combattanteAI a atteint la position finale alors on va chercher la nouvelle position de l'ennemi à attaquer. Enfin ce run() a une dernière fonctionnalité, elle crée de nouvelle CombattanteAI tel que il y a une CombattanteAI pour deux unités du joueur.

J'ai aussi implémenté la classe Joueur qui n'est pas très longue à décrire. Le joueur possède une liste de bâtiments, une liste d'unités, un nombre de bois et un nombre de nourritures initialisés à 100 dans le constructeur. En méthode on retrouve que des getters et des setters pour les attributs.

Pour rester dans les unités, une partie importante de mon code est le threadUnit dans la classe Etat. C'est une méthode qui permet de gérer les unités en actualisant les positions dans les cases et faire des actions. Dans un premier temps, en

traversant toutes les cases du plateau j'enlève toutes les unités de leurs cases, ce qui permet d'enlever graphiquement les unités des cases d'où elles sont parties. Ensuite, je remets les unités dans les cases selon les positions, grâce à cela on montre graphiquement le déplacement d'une unité.

Dans le cas d'une ouvrière, je regarde si dans sa case il y a aussi une ressource, si c'est le cas alors j'enlève la ressource de la liste des ressources que possède le plateau et je la fais disparaître graphiquement, en même temps j'augmente de 1 le nombre de bois ou nourriture du joueur selon le type de la ressource. Dans cette méthode qui est donc un Thread, je teste aussi les conditions de victoire et de défaite : si le joueur atteint un certain nombre de ressources il gagne, s'il ne peut plus générer d'unités (plus de ressources) et qu'il n'en a plus alors il a perdu. Ce Thread s'effectue toutes les secondes.

La plupart des autres méthodes de la classe sont juste des getters et setters, parmi les setters que j'ai fait, il est à noter que les setFourmilierePlateau() et setCasernePlateau() sont quand même importantes car elles permettent de mettre les bâtiments sur le plateau.

Cependant il y a une méthode qui n'est pas un Thread mais qui est tout autant important. C'est la méthode unitADeplacer(). Le déplacement décrit dans les classes des unités du joueur avec la position finale est possible grâce à cette méthode. Lors d'un clic gauche et droit pour déplacer une unité, les positions initiales sont stockées dans des attributs d'État. Ce qui permet ensuite avec cette méthode d'envoyer à l'unité qui est sur la position initiale, de lui donner une position finale et elle peut ensuite commencer à se déplacer.

Enfin dans la classe Etat, les méthodes win() et lose() sont des méthodes affichant une fenêtre pour dire si le joueur a gagné ou perdu.

Dans la classe Affichage, il y a l'activation de beaucoup de Thread évoqués dans ma partie.

Dans la classe Case on peut voir qu'on définit le placement d'une unité, bâtiment ou ressource grâce aux attributs avec un boolean pour dire si la case possède l'élément et un attribut du type de l'élément. J'ai implémenté le drawUnit, drawCaserne, drawFourmiliere et drawCombattanteAI avec les attributs et getters, setters, correspondant à ces éléments. Les méthodes avec draw cités permettent d'afficher une image tout simplement.

Puis dans la méthode paint(), si la case est occupée par l'élément alors la méthode draw correspondante se lance. J'ai aussi implémenté la méthode importante mouseClicked() qui permet de gérer les clics du joueur. S'il effectue un clic droit alors une position initiale est enregistré dans Etat comme dit plus haut et si un clic

gauche est effectué, on regarde d'abord si la case possède un bâtiment, si c'est le cas alors il génère une unité et fait payer un certain prix au joueur, que ce soit en bois pour générer une Combattante ou en nourriture pour générer un Ouvrier. Si la case ne possède pas de bâtiment alors la méthode clicGauche(e) est appelée et définie une position finale dans Etat et puis lance la méthode unitADeplacer.

Pour finir, il faut aborder l'implémentation des classes de bâtiments, dans ces classes, il n'y a presque rien mis à part une position et un getter. La principale fonction des bâtiments est gérée dans la classe Case. J'ai dû implémenter cette version des bâtiments qui est fonctionnelle car la personne en charge de cette partie a eu un problème de santé et ne pouvant pas finir son code j'ai décidé de refaire ces classes.

J'ai aussi aidé à régler les bugs sur des méthodes comme les Threads liés à l'attaque et j'ai reçu de l'aide des membres de mon groupe pour notamment le déplacement.

Partie de SIVACOUMAR Arsène :

Fonctionnalités implémentées :

-Création et placement des ressources sur la grille de jeu.

-Ajout de ressources durant le déroulement du jeu.

-Création des Combattantes

-Ajout fonctionnalités de combats pour les combattantes du joueur et AI

Liste des class et méthodes implémenter pour les différentes fonctionnalités :

-Toute la class Environnement.Ressource.java

-Toute la class Unites.Combattantes.java

-Dans la class MVC.Etat.java, les méthodes initRessource(), threadRessource() et getListRessource()

-Dans la class MVC.Affichage.java, les méthodes setAllRessource() et refresh()

-Dans la class MVC.Case.java, les méthodes drawRessources(), setRessource(), drawCombattante(, et removeRessource()

-Dans la class MVC.Etat.java, les méthodes threadAttaqueJoueur() et threadAttaqueIA()

La class Ressource.java contient :

-un attribut de type typeRessource qui est un enum avec deux valeurs possibles : bois et nourriture.

-un autre attribut position pour avoir la position.

-2 méthodes : initialiseRessource() qui va initialiser une ressource avec une position aléatoire dans la grille sauf les 9 cases en bas à gauche de la grille et la case (0, 14) car il y aura par la suite une fourmilière, des unités et d'autres bâtiments, et la méthode getPosition() qui retourne la position de la ressource.

La méthode initRessource() va prendre un entier entre 20 et 60 de manière aléatoire , ce qui va déterminer le nombre de ressources à placer dans la grille pour le début du jeu et les ressources crée seront dans l'attribut listRessource(arrayList) de MVC.Etat.java. De plus cette méthode va créer et placer les ressources de manière à ce qu'il y ait seulement 1 ressource par case.

La méthode setAllRessources() va parcourir la grille "plateau"(attribut de la classe Affichage) et placer les ressources dans chaque case en utilisant la méthode setRessource() présent dans la classe Case.

La méthode threadRessource() va lancer un thread qui va ajouter des ressources toutes les 1,75 secondes sur le plateau en prenant en compte la limite de ressources présente dans le jeu qui est fixé à 60 et la règle de 1 ressources par cases.

La méthode refresh() va actualiser l'affichage graphique (repaint) des cases. Pour réaliser cela j'ai effectué un double for pour parcourir le plateau de jeu 2D et pour chaque case faire un repaint si elle est occupée par soit une unité ou une ressource.

La méthode drawRessource() va prendre chaque ressource de listRessources et affiche une image de bois, si la ressource est de type « bois » sinon une image de miel pour le type de ressource « nourriture ».

La méthode setRessources() va prendre une ressource en paramètre et mettre cette ressource dans l'attribut ressource (qui est à null au si aucune ressource n'est présente) et mettre le boolean occupeeRessource a true.

La méthode removeRessource() va mettre l'attribut ressource a null et le boolean occupeeRessource a false. Cette méthode sera utilisée pour supprimer une ressource d'une case, lorsqu'une unité ouvrière va récupérer une ressource.

La méthode threadAttaqueJoueur contient un thread qui permet de gérer l'attaque des troupes du joueur (classe Combattante) envers les unités de l'environnement seulement si les deux unités sont à au plus une case de différence. Pour réaliser cela, j'ai 3 listes, une liste qui contient les unités du joueur, une liste qui contient les

combattantes de l'IA et une liste qui contient les positions des combattantes de l'IA qui ont une vie égale à 0. Par la suite j'ai parcouru avec un premier for la liste des unités du joueur et dans ce for un autre for pour parcourir la liste des combattante de l'IA et vérifier que l'unité uJ du joueur est bien une combattante et récupérer la position de la combattante uJ du joueur et aussi de la combattante uAI pour l'IA, après avoir récupérer les positions p1 et p2 respectivement, je vais voir si la valeur (abs(p1.x – p2.x) <= 1 && abs(p2.x – p2.x)) pour vérifier que les uJ et uAI sont soit sur la même case ou à une distance de 1 case. Si cela est vérifié alors je vais réduire l'attaque de uJ a la vie de uAI et si la vie tombe a 0 alors j'ajoute la position de uAI à la liste qui contient des positions. Pour finir je vais parcourir la liste des positions et supprimer les combattantes de la liste de l'IA et de même de la case où il est positionné.

La méthode threadAttaqueIA contient un thread permettant de gérer l'attaque des unités de l'environnement sur les unités du joueur si les deux unités sont à au plus une case de différence. L'algorithme est quasiment identique à celui du threadAttaqueJoueur, il ne faut pas seulement vérifier que uJ est combattante mais aussi ouvrière car les combattantes de l'IA attaquent toutes les unités du joueur.

Partie de LASLUISA NUNEZ Charlies (1ère version des bâtiments) :

Fonctionnalités implémentées au cours de ce projet pour cette version

- Implémentation des bâtiments du jeu
- Ajout de méthodes permettant de lier les bâtiments au joueur
- Implémentation de la classe Reine qui est l'élément premier de notre jeu

Plan de développement
Temps de :

1.2. Réflexion et documentation, 1h

3. Conception, 3-4h tout en effectuant des tests unitaires pour vérifier que tout est bien implémenté

Conception

Afin de pouvoir créer nos premières unités (qui sont les ouvrières), il nous fallait générer une unité qui est la reine des fourmis. Ainsi, tout au cours de la partie, ce sera cette unité qui se chargera de créer nos ouvrières la classe Reine n'est pas composée de beaucoup d'éléments. En effet, dans notre jeu, la Reine des fourmis se chargera uniquement de générer nos fourmis ouvrières, donc il a été implémenté une méthode createOuvriere() qui se chargera de :

1) créer une ouvrière en un lapse de temps (si le joueur dispose d'assez de ressources pour la générer

2) enlever le coût nécessaire pour créer l'ouvrière des ressources du joueur

3) affecter la fourmi comme étant une unité du joueur qui l'a conçu

Remarque : Afin d'avoir une meilleure lisibilité du code et des classes, j'ai pris l'initiative de réorganiser nos classes et de les insérer dans différents packages en fonction de leur rôle dans notre jeu. Il y aura donc 5 packages :

-Bâtiments (peut être réunis avec le package environnement)
-Environnement
-Joueurs
-MVC
-Unités

Concernant la suite des fonctionnalités implémentés dans le projet, il y a celles qui concernent les bâtiments de notre jeu. En effet, au cours du jeu, nous pouvons implémenter différents types de bâtiments, nous permettant de générer différentes unités (et non plus uniquement des ouvrières). C'est pour cela, que j'ai commencé par implémenter les bâtiments principaux de notre jeu, à savoir :

- La fourmilière (lieu de vie des fourmis et qui servira de base pour chaque joueur)
- La caserne (bâtiment servant à créer différentes troupes de fourmis)

En effet, notre 1ère idée consistait à implémenter des bâtiments tout en y implémentant des algorithmes qui permettait de générer ouvrières à l'aide d'un timer qui, lorsque l'on atteint une certaine valeur (qui est le temps nécessaire pour créer une fourmi) génère une fourmi, ajoute cette fourmi créée à la liste des fourmis du joueur, ajoute la fourmi au bâtiment où elle est créé et termine le timer afin qu'il ne tourne pas en boucle à l'infini et il fallait remettre la constante tempsEcoule (qui est la valeur représentative de création d'une fourmi) à 0 parce que sinon, il y aurait eu un risque que lorsque l'on crée plusieurs fourmis, le timer n'aurait plus jamais atteint cette valeur.

```
public void createTroupe(){
    if(player.getNbNourritures() > 5) { //Si le joueur à suffisamment de ressources, déclenche un chrono pour créer la fourmi
        player.setNbNourritures(-5); //Enlève le nombre de ressources correspondant au prix de la fourmi
        time.schedule(new TimerTask() {
            @Override
            public void run() {
                tempsEcoule++;

            }
        }, tempo,  period: 1000);
    }

    if (tempsEcoule == tempsCreaOuvriere){ //une fois que le chrono atteint le temps de création d'une fourmi
        Unite nUnit = new Ouvrier(); //Nous créons l'unité et nous affectons cette même unité aux unités du joueur, ainsi qu'au bâtiment où elle a été crée
        player.addUnite(nUnit);
        super.addUnite(nUnit);
        time.cancel(); //on arrête le chrono
        tempsEcoule = 0; // on remet le compteur à 0
    }
}
```

La fourmilière

Véritable forteresse pour les fourmis, la fourmilière joue le rôle de base pour nos fourmis. En effet, c'est là où été stockés les ressources récoltées par nos ouvrières, ainsi que la reine (qui servira à créer nos ouvrières) Afin que cette fourmilière ne soit pas implémentée n'importe comment, j'ai décidé de l'associer à 3 éléments :

- La grille de jeu, car elle y sera implémentée graphiquement
- Le joueur, car chaque joueur aura accès à sa propre fourmilière et non pas celle de l'adversaire
- Une Reine, car elle y sera toujours présente (même si sa tâche au final se résume à créer des fourmis)

Dans cette classe, plusieurs méthodes utiles au développement du jeu ont été implémentées :

- addFourmiAll(), qui se chargera d'ajouter la fourmi crée par la reine non seulement au joueur mais aussi à la fourmilière où elle a été créée
- remFourmiBat(), qui lorsqu'une fourmi sera déployé sur la grille de jeu quittera la fourmilière et sera associé à la carte
- createOuvriere(), qui sera associé au clic du joueur lorsqu'il appuiera sur la fourmilière, il pourra décider de quand créer la fourmi
- deploiementUnite(), qui en fonction du joueur, déploiera un certain nombre d'unité sur la grille de jeu

Pour l'instant, il n'y a que les ouvrières de disponible sur notre jeu (07/03), par la suite, il s'agira de créer d'autres types de fourmis dans un bâtiment différent, La Caserne

La Caserne

En ce qui concerne ce nouveau bâtiment, il s'agit de créer autre chose que des fourmis basiques, on à décider de créer des fourmis pouvant être envoyé en même temps que les ouvrières (comme si c'était leurs gardes du corps) afin que ces dernières puissent être protégés lors de leur récolte de ressources

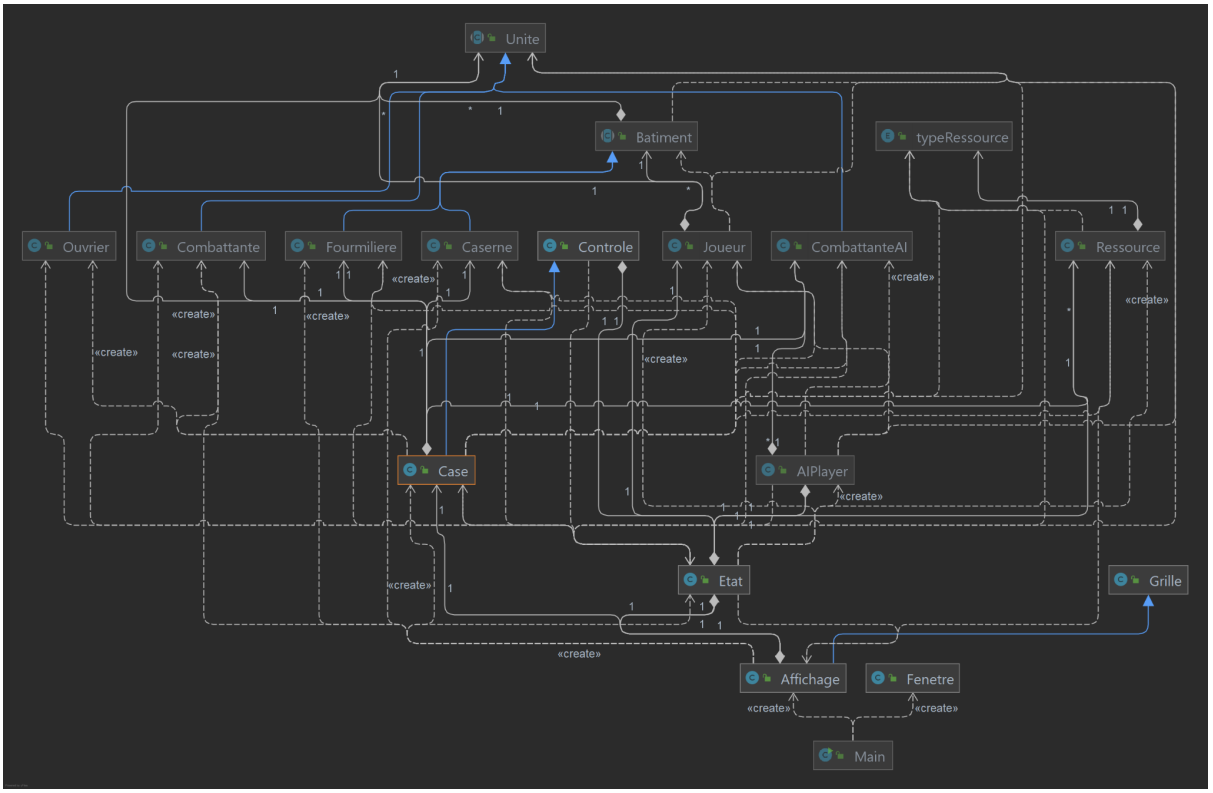
Là aussi, la classe Caserne est créé avec différents attributs :

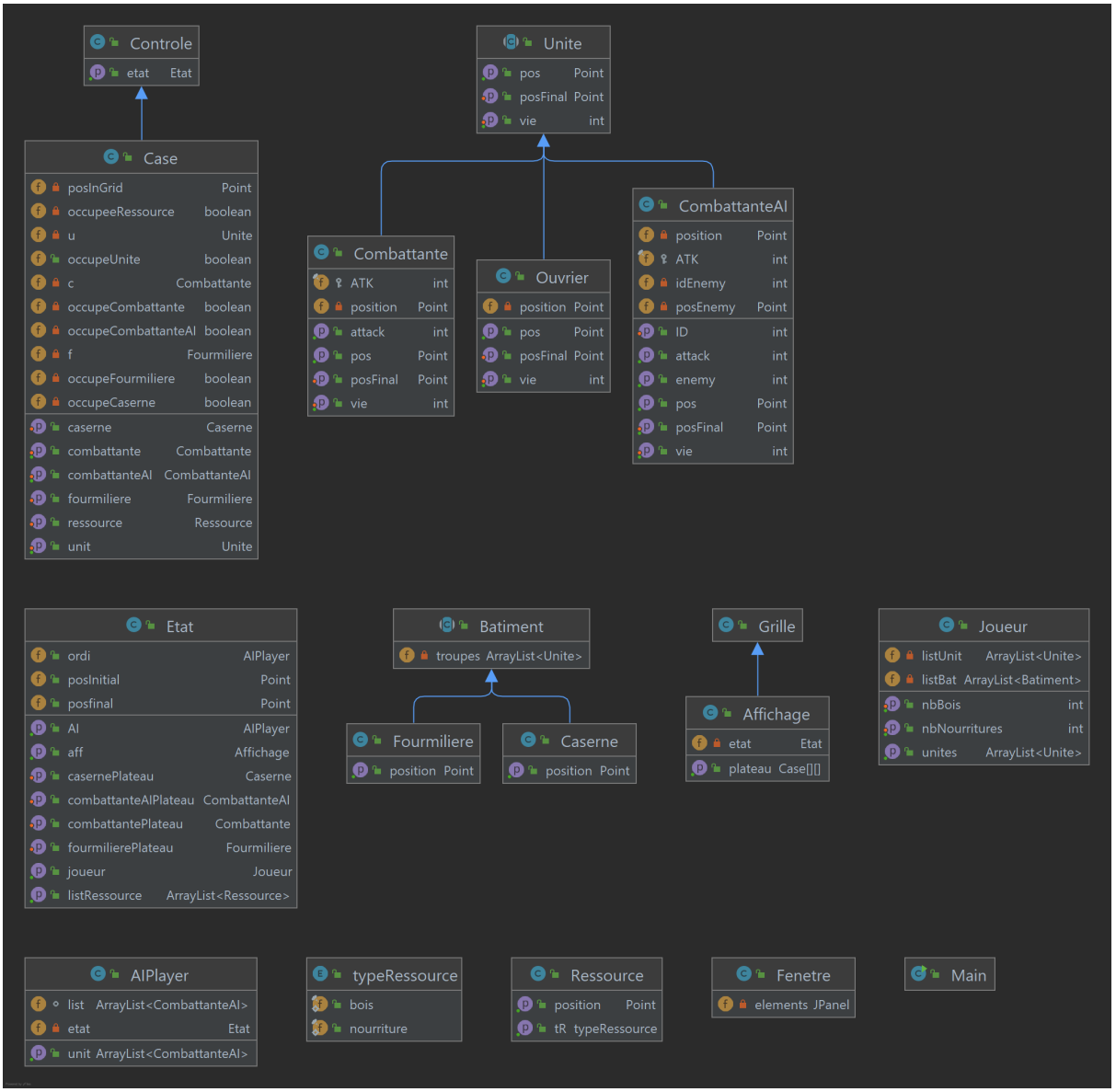
- Joueur, car tout bâtiment construit sur la map, doit être relié à un joueur
- Timer, qui servira de chronomètre afin de créer nos unités en fonction de ce chrono (utilisation de la classe Timer)

Encore une fois, pour le moment, notre caserne ne peut générer que des ouvrières, mais il arrivera très bientôt d'autres unités de combat.

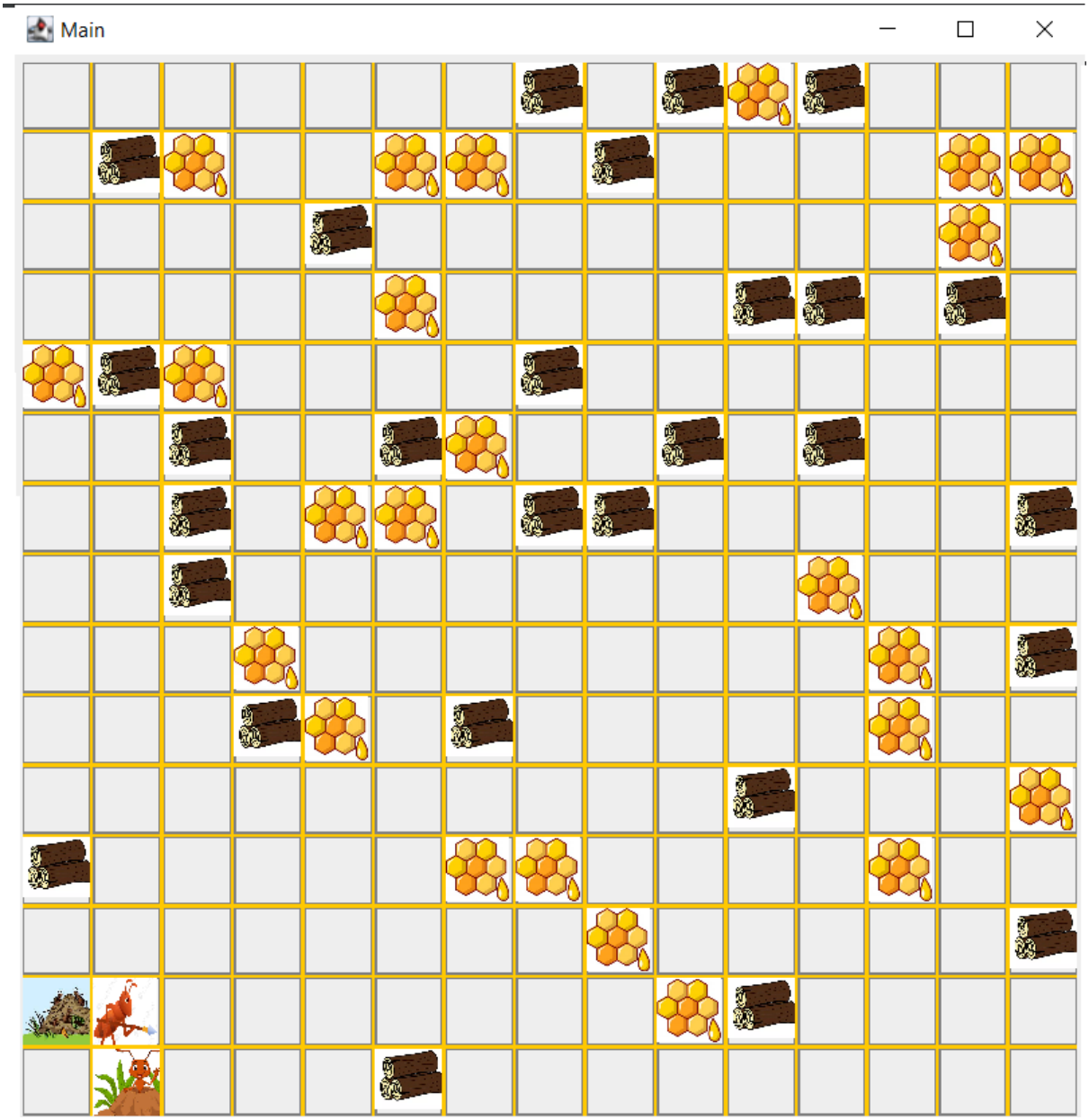
Malgré cette idée, nous avons décidé de changer de bord et de développer une idée plus basique et plus simple afin de pouvoir gérer plus facilement l’utilisation et la création des fourmis.

Diagramme de classes final + liaisons entre les différentes classes




Controle
etat Etat
Unite
pos Point
posFinal Point
vie int
Case
posInGrid Point
occupeeRessource boolean
u Unite
occupeUnite boolean
c Combattante
occupeCombattante boolean
occupeCombattanteAI boolean
f Fourmiliere
occupeFourmiliere boolean
occupeCaserne boolean
caserne Caserne
combattante Combattante
combattanteAI CombattanteAI
fourmiliere Fourmiliere
ressource Ressource
unit Unite
Combattante
ATK int
position Point
attack int
pos Point
posFinal Point
vie int
Ouvrier
position Point
pos Point
posFinal Point
vie int
CombattanteAI
position Point
ATK int
idEnemy int
posEnemy Point
ID int
attack int
enemy int
pos Point
posFinal Point
vie int
Etat
ordi AIPlayer
posInitial Point
posfinal Point
AI AIPlayer
aff Affichage
casernePlateau Caserne
combattanteAIPlateau CombattanteAI
combattantePlateau Combattante
fourmilierePlateau Fourmiliere
joueur Joueur
listRessource ArrayList<Ressource>
Batiment
troupes ArrayList<Unite>
Fourmiliere
position Point
Caserne
position Point
Grille
Affichage
etat Etat
plateau Case[][]
Joueur
listUnit ArrayList<Unite>
listBat ArrayList<Batiment>
nbBois int
nbNourritures int
unites ArrayList<Unite>
AIPlayer
list ArrayList<CombattanteAI>
etat Etat
unit ArrayList<CombattanteAI>
typeRessource
bois
nourriture
Ressource
position Point
tR typeRessource
Fenetre
elements JPanel
Main

VI- Résultat



La capture ci-dessus correspond à l'interface de lancement du jeu. Cette interface contient deux types de ressources, le miel et le bois, aléatoirement sur cette grille. La caserne est bien positionnée sur la case (13:0) et la fourmilière est située sur la case (14:1). La première combattante du joueur est positionnée sur la case (13:1). Au lancement, aucune ouvrière n'est intégrée dans le jeu donc le joueur doit appuyer sur la fourmilière pour initier le premier déploiement d'une ouvrière.

La capture ci-dessus correspond à l'interface de fin du jeu contenant une petite fenêtre qui indique que la partie est terminée et que le joueur a gagné cette partie. Chaque partie est gagnée par le joueur lorsqu'il récupère 110 éléments de nourriture (le miel) sur la carte.

La capture ci-dessus correspond à l'interface de fin du jeu lorsque le joueur perd la partie. La partie est gagnée par l'IA lorsque l'ensemble des combattantes du joueur ont été éliminées par l'IA et que le joueur n'a plus de ressources pour faire apparaître des unités de combat pour défendre. Une petite fenêtre s'ouvre pour indiquer que je joueur a perdu.

VII- Documentation Utilisateur

Étapes à suivre pour lancer notre jeu :

1 : Le jeu est développé en Java donc l'utilisateur doit télécharger un IDE adapté pour le langage de programmation JAVA (IntelliJ ou Eclipse). L'IDE IntelliJ est conseillée car cet outil propose plusieurs versions de JDK téléchargeables directement depuis l'IDE.

2 : Le projet JAVA doit être importé dans l'IDE et lancer en cliquant sur le bouton 'run as Java Application' en se positionnant dans la classe main du projet. Si il y'a une erreur lors de l'exécution pour les images dans la class Case.java, alors il faut mettre le path absolue pour que cela marche.

3 : Lorsque la fenêtre du jeu apparaît, le joueur peut effectuer un clic gauche sur la case (13, 1) pour faire apparaître une combattante et un clic gauche sur la case(14, 2) pour faire apparaître une ouvrière. Pour effectuer les déplacements des unités, le joueur doit effectuer un clic droit sur la case de l'unité combattante (case de départ) puis faire un clic gauche sur la case d'arrivée où le joueur souhaite que l'unité se positionne.

VIII- Documentation Développeur

Dans l'éventualité où une tierce personne décide de reprendre notre projet et de l'améliorer, il devrait se concentrer sur les classes principales qui composent la génération de la carte, c'est-à-dire :

- Etat
- Affichage
- Case
- AIplayer
- Ressources
- Joueur

Dans notre projet, nous avons une classe Main qui contient et qui se chargera de générer le jeu de toute pièce. En ce qui concerne la classe Case, c'est ici où l'on regarde qu'est ce qui est posé sur la case de la map, comme une Caserne, une fourmilière ou encore des ressources ou des fourmis. Ce qui pourrait être amélioré dans notre projet concernant les cases de jeu, serait de modéliser le modèle MVC afin d'avoir une création de cases beaucoup plus simplifiée et qui aura une liaison concernant les unités de jeu ou encore les bâtiments générés sur la carte.

La classe AIPlayer peut aussi être améliorée. En effet, notre IA n'est pas la plus performante, ce qui fait qu'elle est facilement améliorable. Ayant suivi l'UE IA, nous savons qu'il existe des algorithmes permettant de faire jouer notre IA de manière intelligente et plus réfléchi afin qu'il ait plus de chances de gagner la partie plutôt que le joueur en lui-même.

Notre jeu est une bataille entre un joueur et une IA, une des fonctionnalités qu'il aurait été intéressant d'implémenter aurait été de mettre un mode joueur vs joueur, mais également une fonctionnalité permettant de jouer à distance sur 2 ordinateurs différents. Afin d'implémenter nos algorithmes concernant notre IA, il faudrait créer des classes pour chaque algorithme afin de les implémenter sur notre IA actuelle. Concernant le mode joueur vs joueur, il faudrait juste implémenter un autre joueur à notre jeu, mais il faudrait trouver un système afin que les 2 joueurs puissent jouer sur une même machine sans être perturbé par le fait d'utiliser le/la même clavier/souris.

Néanmoins, j'en conçois que l'implémentation de notre carte de jeu n'est pas la plus optimale non plus, donc si quelqu'un reprend notre projet, il pourrait réfléchir à une autre manière de créer la carte de jeu, afin qu'elle soit plus simple et moins complexe à comprendre.

IX- Conclusion et perspective

Pour conclure, ce projet a été très enrichissant pour notre groupe car nous avons acquis beaucoup de compétences en programmation concurrente (multi-threading) et aussi en programmation d'interfaces graphiques. Néanmoins, nous avons rencontré pas mal de difficultés. En effet, tous les membres de notre groupe ne possédait pas le même IDE (IntelliJ et Eclipse ont étés les IDE utilisés), ce qui fait que pour pouvoir coder n'a pas été évident de par le fait d'utiliser des IDE différents mais aussi lors des Updates du projet sur Github qui faisait que nous rencontrions des soucis de synchronisations sur nos IDE, mais comme IntellIJ permet d'importer un projet qui est sous format Eclipse, alors nous avons décidés de créer le projet sur Eclipse et d'importer ce même projet sur IntelliJ afin que chaque membre de notre groupe puissent coder sans soucis.

De plus, certains problèmes techniques et personnelles survenues à l'un d'entre nous a beaucoup retardé l'avancé final de notre projet, ce qui fait que nous n'avons pas eu la possibilité d'implémenter toutes les fonctionnalités prévues au début du projet.

Comme un projet n'est jamais réellement fini, il peut bien sûr être amélioré en ajoutant par exemple un meilleur environnement (plus intelligent), plus d'unités avec plus de fonctionnalités.