Documentation du Jeu: Vikings

Un jeu de gestion et de stratégie en temps réel

Auteurs :
Ramy SAIL
Ismael WANE
Mohamed ALMAHDI
Djamel SALAH

2024/2025

1 Cahier des charges

1.1 Introduction

1.1.1 Contexte du projet

Le jeu de stratégie Viking a pour objectif de permettre au joueur d'incarner un chef de village viking. Il devra jongler entre la gestion de la production agricole et la défense contre les attaques d'autres joueurs, qui tenteront de piller son village. Chaque joueur gère son propre village et peut choisir de développer son économie ou de partir en expédition pour attaquer d'autres villages. Ce jeu sera développé en Java en utilisant le framework graphique **Swing** pour l'interface utilisateur. Le projet comprendra des mécaniques de gestion de ressources, de défense, d'attaque et de progression du temps.

1.1.2 Objectif du projet

Le projet consiste à développer un jeu de stratégie et de gestion où le joueur doit gérer la production agricole du village (blé, bétail) tout en repoussant les attaques d'autres joueurs. Le but est de **survivre à l'hiver** en équilibrant les ressources nécessaires à la subsistance du village et les moyens de défense (armée, fortifications). Chaque joueur pourra attaquer d'autres villages pour voler leurs ressources et ainsi accélérer son développement.

1.2 Fonctionnalités du Jeu

1.2.1 Multijoueur

- **Multijoueur en temps réel :** Permet à plusieurs joueurs de se connecter et d'interagir ensemble.
- **Système d'attaque entre joueurs :** Possibilité d'attaquer les villages adverses pour voler des ressources.
- **Synchronisation en temps réel :** Mise à jour de l'état du jeu pour tous les joueurs simultanément.

1.2.2 Interface Graphique

- **Affichage des éléments du jeu :** Icônes représentant les Vikings, leur santé et leurs actions disponibles.
- **Ressources :** Affichage des quantités de blé et de bétail disponibles, ainsi que des barres de progression indiquant leur évolution.
- **Ennemis :** Visualisation des joueurs adverses en approche et de leurs troupes.

- **Zone cliquable :** Le joueur peut cliquer sur des zones spécifiques du terrain (champs, fortifications, etc.) pour assigner des tâches aux Vikings.
- **Interactions avec le jeu :** Zones cliquables pour gérer les récoltes, les déplacements des Vikings et l'organisation des défenses.

1.2.3 Panneau de Contrôle Interactif

- **Objets interactifs :** Chaque objet (Viking, bétail, etc.) dans le jeu est cliquable pour afficher des informations détaillées sur l'état et les actions possibles.
- **Suivi des Vikings :** Affichage des informations relatives à chaque Viking : santé, fatigue, et actions possibles (défendre, planter, nourrir, récolter, se déplacer, attaquer).
- **Ressources :** Barres de progression indiquant les quantités de blé, bétail, etc., disponibles.

1.2.4 Attaques entre joueurs

- **Gestion des attaques :** Chaque joueur peut envoyer ses troupes attaquer d'autres villages pour voler leurs ressources.
- **Défense du village :** Mise en place de stratégies de défense (patrouilles, fortifications, embuscades) pour protéger les ressources.
- **Butin et conséquences :** En cas de victoire, l'attaquant récupère une partie des ressources de la cible. En cas de défaite, ses troupes subissent des pertes.
- **Système de matchmaking :** Les joueurs peuvent attaquer des villages de niveaux similaires pour assurer un équilibre.

1.2.5 Mécaniques de Défense

- **Fabrication d'armes :** Le joueur peut fabriquer des armes (haches, arcs, boucliers) en utilisant les ressources collectées.
- **Fortifications :** Construction de murs et autres infrastructures de défense pour protéger le village.
- **Gestion des Vikings :** Envoi des Vikings aux différents points de défense pour protéger les ressources et les zones stratégiques.

1.2.6 Objectifs et Progression

- **Système de temps :** Une barre affiche le temps restant avant l'hiver. Le joueur doit atteindre certains objectifs avant cette échéance.
- **Système de score :** Le jeu attribue des points en fonction de la gestion des ressources, de l'accomplissement des objectifs et des raids réussis.

1.2.7 Effets Sonores et Musiques

- **Sons d'actions :** Effets sonores distincts pour chaque action (déplacements, combats, récoltes, etc.).
- **Musique de fond :** Une musique dynamique qui varie selon les situations : calme pendant la gestion des ressources et plus intense lors des combats.

1.3 Exigences Techniques

- **Langage de programmation :** Java.
- **Framework graphique :** Java Swing.
- **Gestion des threads :** Mouvements des unités, génération des événements et mises à jour du jeu en temps réel.
- **Architecture multijoueur :** Modèle client-serveur, avec un serveur central pour gérer les connexions et synchroniser les données.

1.4 Critères d'acceptation

- **Interface utilisateur :** Intuitive et ergonomique.
- **Équilibre du gameplay :** Les mécaniques d'attaque et de défense doivent être équilibrées.
- **Multijoueur :** Synchronisation fluide, matchmaking équitable.

2 Analyse

2.1 Analyse Globale

Fonctionnalités :

Le projet *Vikings* est un jeu de stratégie multijoueur en temps réel combinant gestion de ressources, attaques entre joueurs, et survie jusqu'à l'hiver. Chaque fonctionnalité a été évaluée selon sa priorité pour le gameplay et sa complexité technique.

Les fonctionnalités ont été sélectionnées en fonction :

- de leur impact sur l'expérience de jeu (ex : interactions, visibilité, feedback),
- de leur cohérence avec l'univers viking (ressources, raids, survie),
- et de leur faisabilité technique dans un projet en Java avec Swing.

Cependant, une simple priorisation ne suffit pas : chaque fonctionnalité est le fruit de choix de conception spécifiques, que nous détaillons ci-dessous.

Fonctionnalité 1 : Interface graphique du terrain

- **Priorité** : 1 **Difficulté** : Moyenne
- **Choix techniques** : Utilisation de Java Swing pour créer une vue interactive (`ViewPartie.java`). Implémentation d'un système de conversion entre les coordonnées du modèle (`Position.java`) et celles de l'écran.
- **Objectif** : Afficher dynamiquement les unités (Vikings, champs, bétail, pillards) de manière fluide avec zoom et défilement.

Fonctionnalité 2 : Panneau de contrôle interactif

- **Priorité** : 1 **Difficulté** : Moyenne
- **Choix techniques** : Composant `PanneauControle` contenant un `SlidingMenu` interactif avec boutons.
- **Paramètres de jouabilité** : Barres de santé et de fatigue mises à jour en temps réel. Seuils visuels définis : vert (>70%), orange (30–70%), rouge (<30%).
- **Objectif** : Suivre et gérer les unités, assigner les tâches selon leur état et leur position.

Fonctionnalité 3 : Attaques entre joueurs

- **Priorité** : 1 **Difficulté** : Moyenne à Difficile
- **Choix techniques** : Le joueur sélectionne un camp adverse, choisit une ressource cible et envoie un nombre défini de Vikings attaquants. Le serveur gère ensuite les déplacements, les affrontements et le transfert des ressources via des callbacks déclenchés à l'arrivée.
- **Synchronisation réseau** : Chaque action d'attaque est encapsulée dans un paquet (`PacketAttack`) et transmise au serveur, qui actualise l'état global puis notifie les clients.
- **Objectif** : Rendre les interactions entre joueurs dynamiques et stratégiques. Le système d'attaque permet de déséquilibrer un joueur adverse ou d'accélérer son propre développement au risque de perdre des troupes.
- **Jouabilité** : Le joueur doit équilibrer le nombre de défenseurs et d'attaquants, choisir le bon moment pour frapper et évaluer le risque selon la force ennemie.

Fonctionnalité 4 : Défense contre les attaques ennemies

- **Priorité** : 1 **Difficulté** : Moyenne à Difficile
- **Choix techniques** : Le joueur peut sélectionner une ressource de son propre camp (champ, bétail, réserve...) et y assigner un ou plusieurs Vikings pour défendre la zone. En cas d'attaque ennemie, une logique de confrontation est déclenchée si des Vikings adverses sont détectés à proximité.
- **Gestion dynamique des unités** :
  - Le joueur peut **rappeler ses unités offensives** (Vikings envoyés à l'attaque) pour les réaffecter à la défense. Cette action est possible à tout moment via le panneau de contrôle.
  - Lors du rappel, les Vikings retournent automatiquement à leur camp d'origine. Le déplacement est géré par un `ThreadMovement` avec une action spéciale à l'arrivée.
- **Déclenchement du combat** : Lorsqu'un Viking défenseur se trouve à une distance suffisante d'un Viking attaquant, un affrontement est déclenché automatiquement. Chaque unité inflige des dégâts selon sa santé et ses caractéristiques.
- **Objectif** : Introduire une composante stratégique forte : le joueur doit choisir s'il préfère garder ses troupes à l'offensive ou les ramener défendre, selon le déroulement des raids ennemis.

- **Considérations de gameplay** :
  - Une attaque bien préparée peut forcer l'adversaire à désorganiser sa défense.
  - Le rappel d'unité coûte du temps (durée du trajet retour) et doit donc être anticipé.
  - Un Viking défendant une ressource l'empêche d'être pillée tant qu'il est en vie.

Fonctionnalité 5 : Objectifs et progression

- **Priorité** : 1 **Difficulté** : Facile
- **Choix techniques** : Le jeu se déroule sur une durée limitée. Une barre de progression temporelle s'affiche à l'écran et diminue progressivement jusqu'à l'arrivée de l'hiver. Des objectifs sont définis en début de partie (quantité de blé, nombre d'animaux, niveau de défense).
- **Système de score** : À la fin de la partie, un score est calculé en fonction des objectifs atteints, du nombre de combats remportés, et de la santé moyenne des villageois.
- **Objectif** : Donner un cadre au jeu et une pression temporelle. Le joueur ne peut pas adopter une stratégie passive : il doit constamment optimiser ses ressources pour atteindre les seuils requis avant l'hiver.

Fonctionnalité 6 : Effets sonores et musiques

- **Priorité** : 3 **Difficulté** : Facile
- **Choix techniques** : Sons déclenchés lors des actions. Musiques adaptatives selon le contexte (gestion/calme vs. attaque/intense).
- **Objectif** : Améliorer l'immersion et fournir un retour auditif au joueur.

Fonctionnalité 7 : Multijoueur

- **Priorité** : 4 **Difficulté** : Difficile
- **Choix techniques** : Le jeu repose sur une architecture client-serveur. Chaque client est une instance du jeu avec son propre affichage, mais l'état global est maintenu par le serveur, qui synchronise les actions via des sockets TCP.
- **Transmission des données** : Les données échangées (actions, mises à jour de l'état, attaques, défenses, positions...) sont sérialisées avec la bibliothèque `Gson` au format JSON. Cela facilite le traitement et le débogage.

- **Gestion des connexions** :
  - Chaque client est géré par `ThreadCommunicationServer` côté serveur.
  - À la connexion, le client reçoit son `campId` ainsi que l'état initial de la partie.
- **Objectif** : Permettre à plusieurs joueurs d'interagir en temps réel dans une même partie. Cela ouvre la voie à des mécaniques compétitives, comme les attaques et la gestion simultanée de territoires.

Choix techniques transversaux

- **Langage** : Java — choisi pour sa portabilité, son support des threads, et son intégration facile avec Swing.
- **Interface graphique** : Java Swing — simple, intégrée à Java SE, permet un contrôle total sur les éléments graphiques.
- **Architecture** : Patron de conception MVC (Modèle-Vue-Contrôleur) combiné à une architecture client-serveur.
- **Réseau** : Communication via sockets TCP. Données échangées en JSON à l'aide de la bibliothèque `Gson`.
- **Concurrence** : Utilisation de threads pour les tâches parallèles : mouvements des entités, mise à jour de la santé, réception de messages réseau, etc.

2.2 Analyse détaillée du système

Le jeu "Vikings" repose sur une architecture client-serveur permettant à plusieurs joueurs de participer simultanément à une partie multijoueur. Cette section présente en détail les différentes composantes logicielles du système, ainsi que les fonctionnalités réseau assurant la communication entre les clients et le serveur.

2.2.1 Structure logique du jeu

L'architecture logicielle est modulaire et suit les principes de séparation des responsabilités. Elle se décompose en plusieurs couches principales :

- **Interface graphique (UI)** : développée avec Swing, elle permet aux joueurs d'interagir avec le jeu (cartes, entités, boutons d'actions).
- **Moteur de jeu** : contient la logique de gestion des entités, des ressources, des règles du jeu, des combats, etc.

- **Module réseau** : assure les échanges d'informations entre le serveur et les clients.
- **Gestion du temps** : à l'aide de threads pour mettre à jour régulièrement l'état du jeu.

2.2.2 Composants principaux

- **Client** : chaque joueur utilise une instance du client pour se connecter au serveur, envoyer des commandes et recevoir l'état du jeu.
- **Serveur** : centralise les décisions, synchronise l'état du jeu, interprète les actions des joueurs et les répercute à tous les clients.
- **Entités du jeu** : les vikings, les fermiers, les animaux, les champs et les camps sont des entités manipulables par les joueurs.
- **Carte du jeu** : représentée comme une grille sur laquelle les entités se déplacent et interagissent.

2.3 Fonctionnalités

Fonctionnalité : Interface graphique du terrain

Cette fonctionnalité permet d'afficher dynamiquement les éléments du jeu (unités, champs, bétail) sur l'interface Swing, avec prise en charge du zoom et du défilement.

Classes impliquées

- `ViewPartie` : Vue principale où sont dessinées les entités.
- `Position` : Conversion entre coordonnées modèle et vue.
- `Viking`, `Sheap`, `Wheat` : Entités à afficher.

Comportement principal

1. Calcul des coordonnées transformées (modèle → vue).
2. Affichage des entités via Swing avec gestion du zoom et scroll.
3. Redessiner les élements en cas de mouvement ou zoom.

Contraintes et gestion

- Fluidité de l'affichage pour une expérience utilisateur optimale.
- Mise à jour en temps réel de l'affichage en fonction des changements d'état.

Conclusion

Cette fonctionnalité constitue la base visuelle du jeu et assure une représentation cohérente et réactive de l'environnement de jeu.

Fonctionnalité : Panneau de contrôle interactif

Cette fonctionnalité permet au joueur de visualiser les statistiques de ses unités et d'interagir avec elles via une interface dynamique.

Classes impliquées

- `PanneauControle`, `SlidingMenu` : Gérent l'affichage et les interactions.
- `EventBus`, `PlantEvent`, `PlantListener` : Gestion à base d'événements.

Interactions principales

1. Affichage de panneaux d'information lors du clic sur une entité.
2. Attribution de tâches aux Vikings selon leur état.
3. Affichage visuel de la santé à l'aide d'une barre de vie avec codes couleur.

Contraintes et gestion

- Réactivité de l'interface pour des interactions fluides.
- Affichage contextuel selon la position des unités.

Conclusion

Une interface essentielle pour une gestion efficace des ressources et unités, intégrée de manière fluide dans le gameplay.

Fonctionnalité : Attaques entre joueurs

Cette fonctionnalité permet à un joueur d'attaquer un camp adverse pour piller des ressources. Elle implique une coordination entre les classes du modèle, le réseau, et l'interface utilisateur.

Classes impliquées

- `Viking` : entité de base pouvant être envoyée en attaque.
- `Camp` : représente un village, contient les ressources et unités.
- `PacketAttack` : paquet réseau contenant les informations d'attaque.
- `ThreadCommunicationClient / Server` : gèrent la transmission et la réception des actions entre client et serveur.
- `MovementThread` : déclenche le déplacement des unités et appelle des callbacks lors de l'arrivée.

Méthode de lancement d'attaque

1. Le joueur clique sur une ressource adverse dans l'interface (`ViewPartie`).
2. Un menu s'affiche avec un bouton "Attaquer" et un champ pour choisir le nombre de Vikings.
3. Lorsque le joueur valide, un objet `PacketAttack` est généré avec :
    - les coordonnées de la cible,
    - l'identifiant du camp adverse,
    - le nombre de Vikings assignés.
4. Le paquet est envoyé au serveur via `ThreadCommunicationClient`.

Traitement serveur

- Le serveur valide l'action (vérifie que le joueur possède assez de Vikings disponibles).
- Il crée des instances de `MovementThread` pour chaque Viking attaquant.
- Une fois arrivés sur place, les Vikings déclenchent une interaction :
    - si la ressource n'est pas défendue : elle est partiellement volée,
    - si des défenseurs sont présents : un combat est déclenché (cf. section défense).
- Le serveur met à jour l'état global (ressources, santé des Vikings) et renvoie une mise à jour aux clients.

Contraintes et gestion d'erreurs

- Si l'attaquant n'a pas assez de Vikings disponibles, l'action est rejetée.
- Une attaque ne peut pas être lancée si une autre est déjà en cours pour la même cible.
- Un délai peut être imposé entre deux attaques successives pour éviter le spam.

Conclusion

Cette fonctionnalité repose sur une interaction forte entre les couches graphique, réseau et modèle. Elle permet de dynamiser le jeu et d'encourager la prise de risques stratégique.

Fonctionnalité : Défense contre les attaques ennemies

Cette fonctionnalité permet au joueur de protéger ses ressources contre les attaques extérieures en assignant des Vikings à la défense.

Classes impliquées

- `Warrior`, `Camp` : Gestion de la défense et des unités.
- `MovementThread`, `ThreadCommunicationClient/Server` : Logique de déplacement et de confrontation.

Comportement principal

1. Le joueur assigne un Viking à une ressource à défendre.
2. Les attaquants ennemis sont détectés automatiquement à proximité.
3. Un combat est engagé dés que les conditions de confrontation sont remplies.

Contraintes et gestion

- Possibilité de rappeler des Vikings de l'attaque pour renforcer la défense.
- Gestion des dégâts et réaffectation dynamique des unités.

Conclusion

La défense ajoute une dimension tactique importante et implique un arbitrage entre attaque et protection des ressources.

Fonctionnalité : Objectifs et progression

Cette fonctionnalité impose des objectifs à atteindre avant l'arrivée de l'hiver, tout en proposant un système de score.

Classes impliquées

- `Camp`, `ViewPartie` : Affichage de la barre de progression.
- `Entity` : Utilisée pour l'évaluation des unités et ressources.

Comportement principal

1. Affichage d'une barre temporelle avant l'hiver.
2. Objectifs fixés en début de partie (blé, moutons, etc.).
3. Calcul du score final en fonction des résultats.

Contraintes et gestion

- Forcer le joueur à adopter une stratégie proactive.

Conclusion

Un système qui structure la partie et favorise la planification à long terme.

Fonctionnalité : réseau

Ce projet implémente plusieurs fonctionnalités réseau pour permettre la communication entre un serveur et des clients dans un jeu multijoueur. Voici les fonctionnalités réseau en détail :

1. **Connexion des clients au serveur** :
   - Les clients se connectent via une adresse IP et un port.
   - Chaque client est représenté côté serveur par un `ThreadCommunicationServer`.
2. **Gestion des joueurs connectés** :
   - Le serveur maintient une liste des joueurs connectés.
   - Lorsqu'un joueur se connecte, son pseudo et son IP sont diffusés via un paquet `PacketConnectedPlayers`.
3. **Attribution des camps** :

- Chaque client reçoit un ID unique représentant son camp via le paquet `PacketCampIdNbPlaye`
- Le serveur attribue dynamiquement les camps aux nouveaux joueurs.

4. **Synchronisation de l'état du jeu** :
   - Le serveur envoie l'état complet du jeu toutes les 100 ms via un thread `ThreadGameState`.
   - Le paquet utilisé contient une instance de la classe `Partie`.
5. **Envoi de commandes des clients** :
   - Les clients envoient différentes commandes :
     - Déplacement : `PacketMovement`
     - Attaque : `PacketAttack`
     - Plantation : `PaquetPlant`
     - Récolte : `PaquetHarvest`
     - Nourriture : `PaquetEat`
6. **Gestion des actions côté serveur** :
   - Le serveur interprète les commandes reçues et met à jour l'état de la partie.
   - Il traite les déplacements, combats, modifications de champs, alimentation des animaux, etc.
7. **Diffusion des mises à jour** :
   - Toute modification de l'état du jeu est diffusée à tous les clients pour maintenir une vision partagée.
8. **Gestion des paquets réseau** :
   - Les paquets sont sérialisés/désérialisés avec Gson.
   - Chaque message est encapsulé dans un `PacketWrapper` contenant son type et son contenu.
9. **Gestion des erreurs et déconnexions** :
   - Le serveur détecte les déconnexions et met à jour la liste des joueurs.
   - Il peut fermer proprement la connexion en cas d'erreur.
10. **Logs des communications** :
    - Les échanges de paquets sont enregistrés côté serveur.
    - Cela permet un suivi et un débogage des interactions réseau.

Cette architecture réseau garantit une communication efficace, un état du jeu cohérent pour tous les clients et une gestion fluide des interactions multijoueur.

Fonctionnalité : Multijoueur

Le mode multijoueur permet à plusieurs joueurs d'interagir dans une même partie en temps réel.

Classes impliquées

- `ThreadCommunicationClient`, `ThreadCommunicationServer` : Communication réseau.
- `Server`, `Client` : Architecture client-serveur.
- `Packet*` : Représentent les actions à transmettre (attaque, défense, etc.).

Comportement principal

1. Connexion de chaque joueur à un serveur central.
2. Synchronisation des actions via sockets TCP.
3. Mise à jour de l'état global par le serveur, transmis aux clients.

Contraintes et gestion

- Gestion des désynchronisations et des conflits d'actions.
- Performance réseau à surveiller pour un affichage fluide.

Conclusion

Le multijoueur ouvre la voie à des stratégies dynamiques, interactives et compétitives.

Fonctionnalité : Effets sonores et musiques

Cette fonctionnalité améliore l'immersion du joueur en adaptant l'ambiance sonore aux actions en cours.

Classes impliquées

- `SoundManager` (hypothétique), `ViewPartie` : Déclenchement des sons et musiques.

Comportement principal

1. Sons différenciés pour chaque action (clic, attaque, récolte).
2. Musique adaptative selon le contexte du jeu.

Contraintes et gestion

- Nécessité de ne pas surcharger l'interface sonore.
- Prise en charge des transitions douces entre musiques.

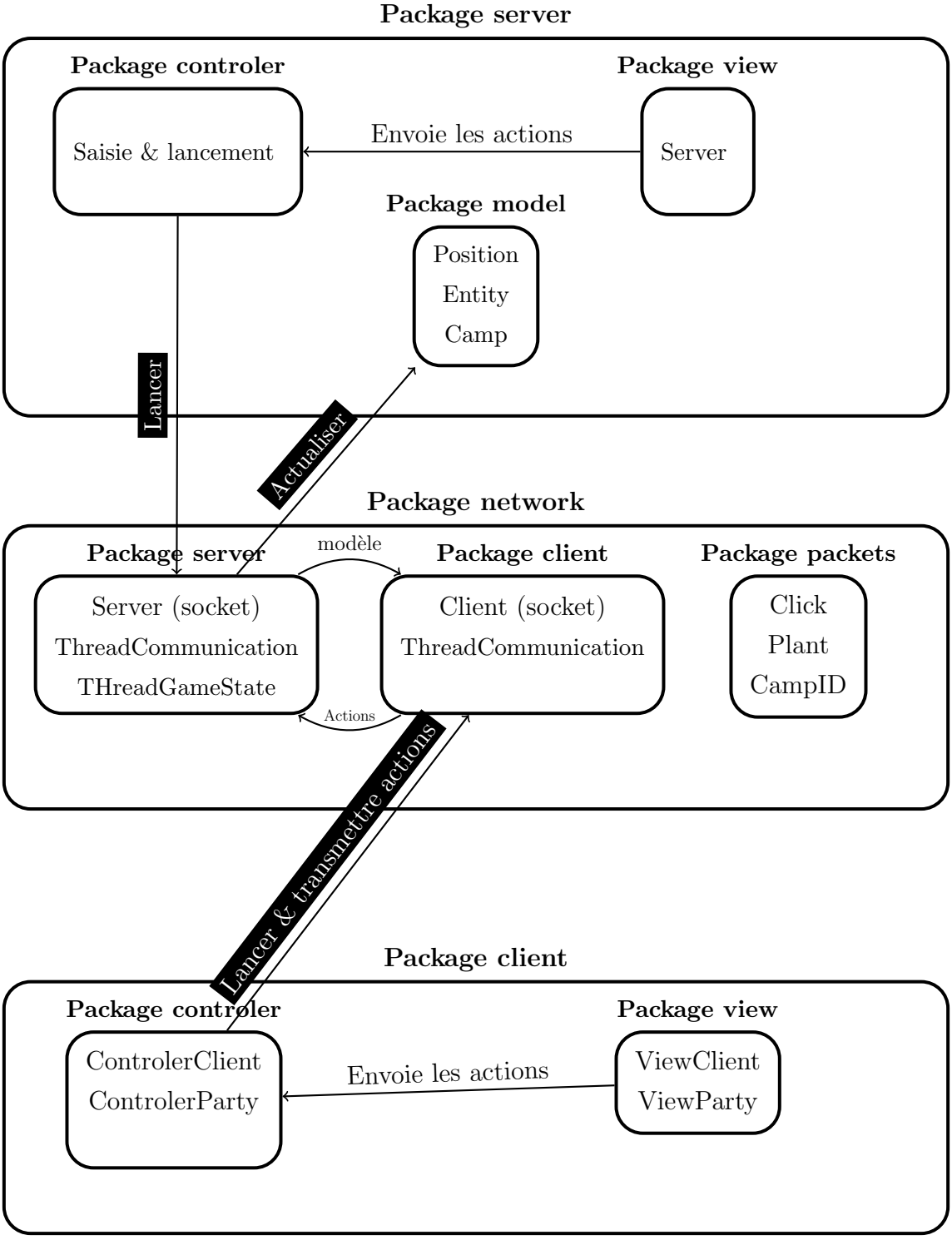
Conclusion

Cette couche sonore apporte du dynamisme et renforce le feedback utilisateur.

3 Conception

3.1 Conception générale

Utilisation du patron MVC au sein d'une architecture Client-Server ce qui donne:



Le client (représenté par le package client) et le serveur donc communique à travers le package network.
Le serveur détient l'état courant de la partie avec le modèle, et l'envoi à chaque fois aux clients pour qu'ils actualisent leurs vus.
Le client de son coté envoi les actions réalisées par le joueur au serveur qui les traitent puis actualise le modèle.
Package packets: regroupe le type de messages que peuvent s'échanger le client et le serveur.

3.1.1 Choix dans l'implémentations

- Le choix a été fait que le serveur soit lancé séparément du client (deux processus différents).
- **Couche transmission:** Initialement des sockets TCP vont être utilisées
- **Echange sur le réseau:** Pour envoyer les objets à travers les flux de sockets on utilise *com.google.gson.Gson* (Github/gson) une bibliothèque réalisée par google qui gère la sérialisation et désérialisation en format json.

3.1.2 Fonctionnement serveur:

Le serveur lancé **(network/server/Server.java)** attends la connexion du nombre précisé de joueurs, pour chaque connexions il crée un thread **(network/server/ThreadCommunicationServer.java)** pour gérer la communication avec ce client.
Quand tout le monde est connecté, le serveur crée l'instance de la partie **(server/model/Partie.java**) et l'envoi à tous les clients pour qu'ils lancent la vue de la partie.

3.2 Le modèle

- **Entités de base** : Vikings (guerriers et fermiers), animaux d'élevage, cultures
- **Environnement** : Champs (`Field`), position des camps (`Position`)
- **Structure centrale** : `Camp`, qui centralise la gestion d'un village viking

La modélisation repose sur l'usage de classes abstraites (`Entity`, `Vegetable`, `Livestock`) et d'interfaces (`Moveable`) assurant la factorisation et l'extensibilité des comportements. Plusieurs threads autonomes permettent de gérer les déplacements (`MovementThread`), la perte de santé (`startHealthDecayThread`), et la croissance biologique.

3.2.1 Fonctionnalités principales du jeu

Le modèle serveur prend en charge les fonctionnalités fondamentales du gameplay viking : gestion de ressources, défense et survie.

1. Ressources agricoles

- `Farmer.plant()`, `Farmer.water()`, `Farmer.harvest()`
- `Wheat.grow()`, `Wheat.isMature()`
- `Camp.addWheat()`, `Camp.removeWheat()`

2. Élevage

- `Farmer.feed()`, `Livestock.move()`, `Camp.moveSheap()`

3. Défense contre les pillards

- `Warrior.attack()`, `Warrior.defend()`, `Warrior.damage()`

4. Santé des unités

- `startHealthDecayThread()`, `Entity.takeDamage()`, `Entity.isAlive()`

5. Déplacements

- `Viking.move()`, `Livestock.move()`

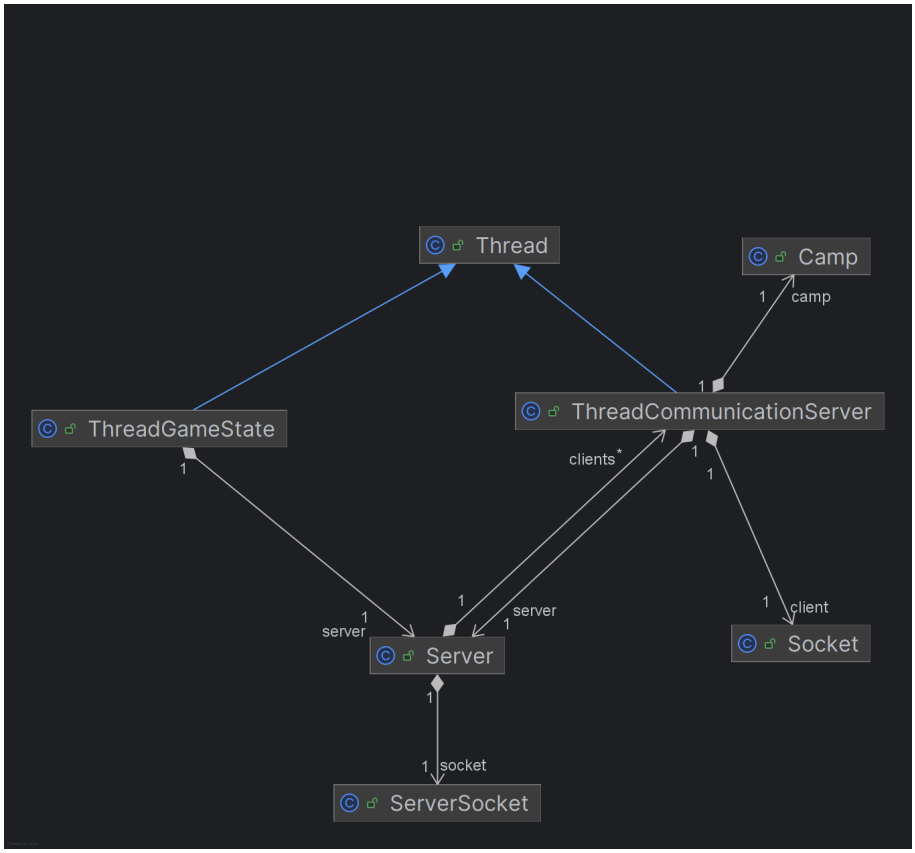
6. Nourriture

- `Viking.eat()`, `Camp.removeSheap()`

3.3 Conception détaillée

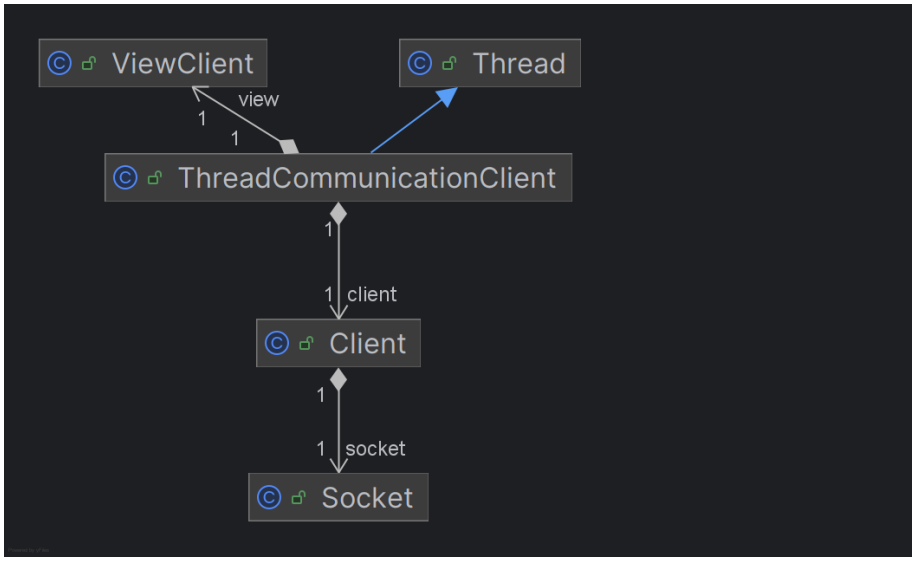
3.3.1 Multijoueur

La communication réseau entre server (qui détient le modèle) et le client (qui détient la vu) est implémenté dans le package network.
- Package: network.server: permet de traiter les rêquetes reçus des clients et actualiser le modèle



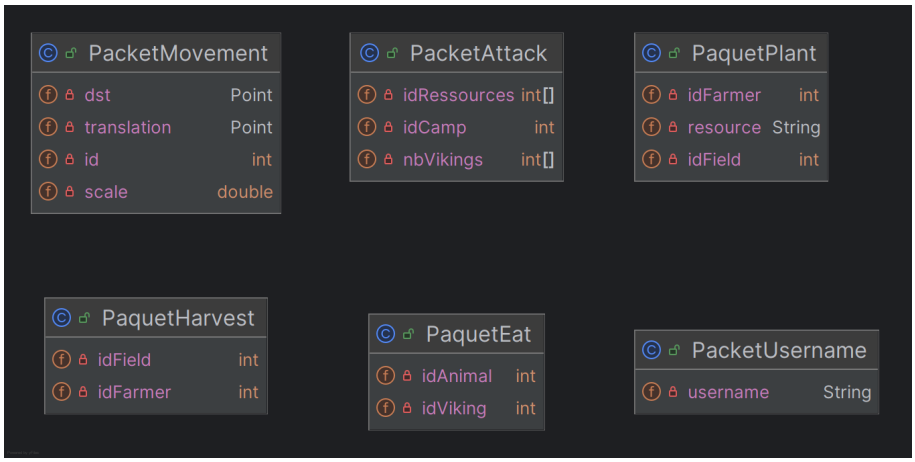
Server: Point de lancement du serveur sur un port déterminé. Pour chaque connexion de client créer une socket pour communiquer avec lui à travers le ThreadCommunication-Server
ThreadGameState: notifie périodiquement tous les clients de l'état courant du modèle.
ThreadCommunicationServer: détient une connexion avec un client pour recevoir ses reqêtes, et lui envoyer l'état courant du modèle.

- **Package: network.client**: l'autre bout de la communication coté client.



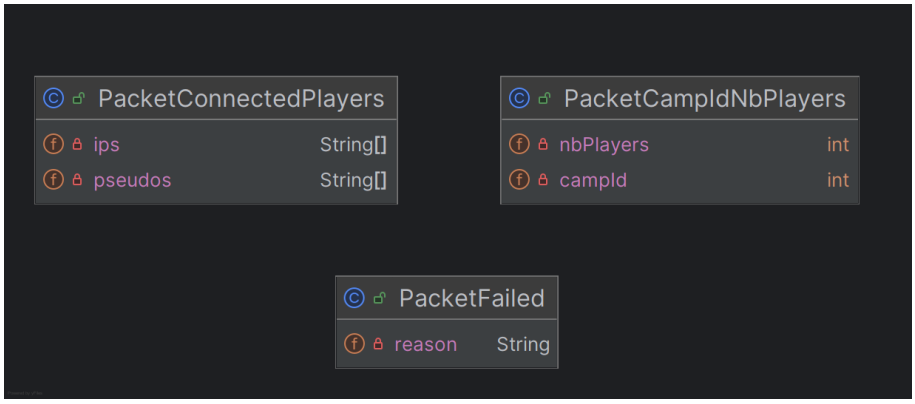
Client: Point de lancement du client, se connecte au serveur sur le port où il est lancé. détient les flux d'entré et de sorti de la socket connectée pour envoyer et recevoir du serveur.
ThreadCommunicationClient: Tourne pour écouter les paquets reçus de la part du serveur et actualiser la vu selon l'état du modèle reçus.

-Package network.packets: Détient ce qu'on appelait dans les sections précédentes des paquets, ce sont des classes à sérialiser et envoyer à traver les sockets entre le client et le serveur.
La structure des classes sont faites pour être la plus légère que possible pour ne pas faire des paquets lours à envoyer sur le réseau.
Voici une liste des paquets envoyés du client au serveur.



- **PacketUsername:** Envoyé le joueur a saisie son pseudo et s'est connecté au serveur.
- **PacketMovement:** Envoyé pour demander au serveur le déplacement de l'entité avec l'identifiant **id** vers le point **dst**. Le point de destination étant dans les coordonnées de la vu on transmet la translation et la mise à l'echelle courante pour que le serveur transforme le point vers les coordonnées du modèle. (plus de détail dans section repère)
- **PacketAttack:** Demande l'envoi d'une attaque sur le camp avec l'identifiant **idCamp**, en précisant la liste des ressources à piler et le nombre de viking à envoyer dessus.
- **PacketPlant:** Demande de planter la ressource sur le champ identifié par **idField**, plantation étant faite par le fermier avec l'identifiant **idFarmer**.
- **PaquetHearvest:** demande de cueillir les plantation du champ identifié par **idField**
- **PaquetEat:** Demande de manger l'animal avec l'identifiant **idAnimal** par le viking identifié par **idViking**

Coté serveur: à la réception des paquets précédents bien sûr le serveur doit vérifier que les actions demandées sont cohérentes avec l'état actuel du modèle avant de les executer et modifier le modèle.
De la même manière voici les paquets envoyés par le serveur aux clients:



- **PacketCampIdNbPlayers:** Pour chaque nouvel joueur connecté le serveur lui attribut son identifiant de camp et le nombre de joueur attendu pour la partie, pour que la vu d'attente de connexion coté client soit initialisée.

- **PacketConnectedPlayers:** Envoyé aux clients à chaque nouvelle connexions pour qu'ils actualisent leur écran d'attente

- **PacketFailed:** Envoyé lorsque une des actions listées pécédemment n'est pas valide.

3.3.2 Détecetion de collision

La détection de collision au sein du camp est faite par le thread **src/server/model/ThreadCollisionC** et voici l'algorithme qu'il utilise pour détecter les collisions.

Algorithm 1: InitialiserEntitésTriées

Input: Liste des entités d'un camp : `entitésCamp`

Output: Structure triée (ABR) des entités selon leur distance à l'origine

1 Créer une structure `ABR` vide avec une clé de tri basée sur la distance à l'origine $(0, 0)$;
2 **foreach** *entité* `e` *dans* `entitésCamp` **do**
3 Calculer la distance $d =$ `distance`$(e.position, (0, 0))$; Insérer `e` dans `ABR` selon d;
4 **return** *ABR*

Algorithm 2: VérifierCollisions

Input: Structure triée (ABR) des entités, seuil de proximité `CLOSEST`

Output: Certaines entités sont arrêtées si trop proches

1 **foreach** *entité* e_1 *dans* *ABR* **do**
2 **foreach** *entité* e_2 *dans* *ABR* **do**
3 **if** $e_1 \neq e_2$ ***et*** `distance`$(e_1.position, e_2.position) <$ *CLOSEST* **then**
4 **if** e_1 *est déplaçable* **then**
5 Arrêter e_1;
6 **else**
7 **if** e_2 *est déplaçable* **then**
8 Arrêter e_2;

3.3.3 Attaque & Défense

3.3.4 Hiérarchie des entités

Classe `Entity`

Classe abstraite commune contenant :

- `float health`, `Point position`, `int campId`
- Méthodes : `getHealth()`, `getPosition()`, `getCampId()`, `isAlive()`

Interface `Moveable`

Méthodes :

- `move(Point destination)`
- `getPosition()`

Classe `Viking`

Hérite de `Entity`, ajoute :

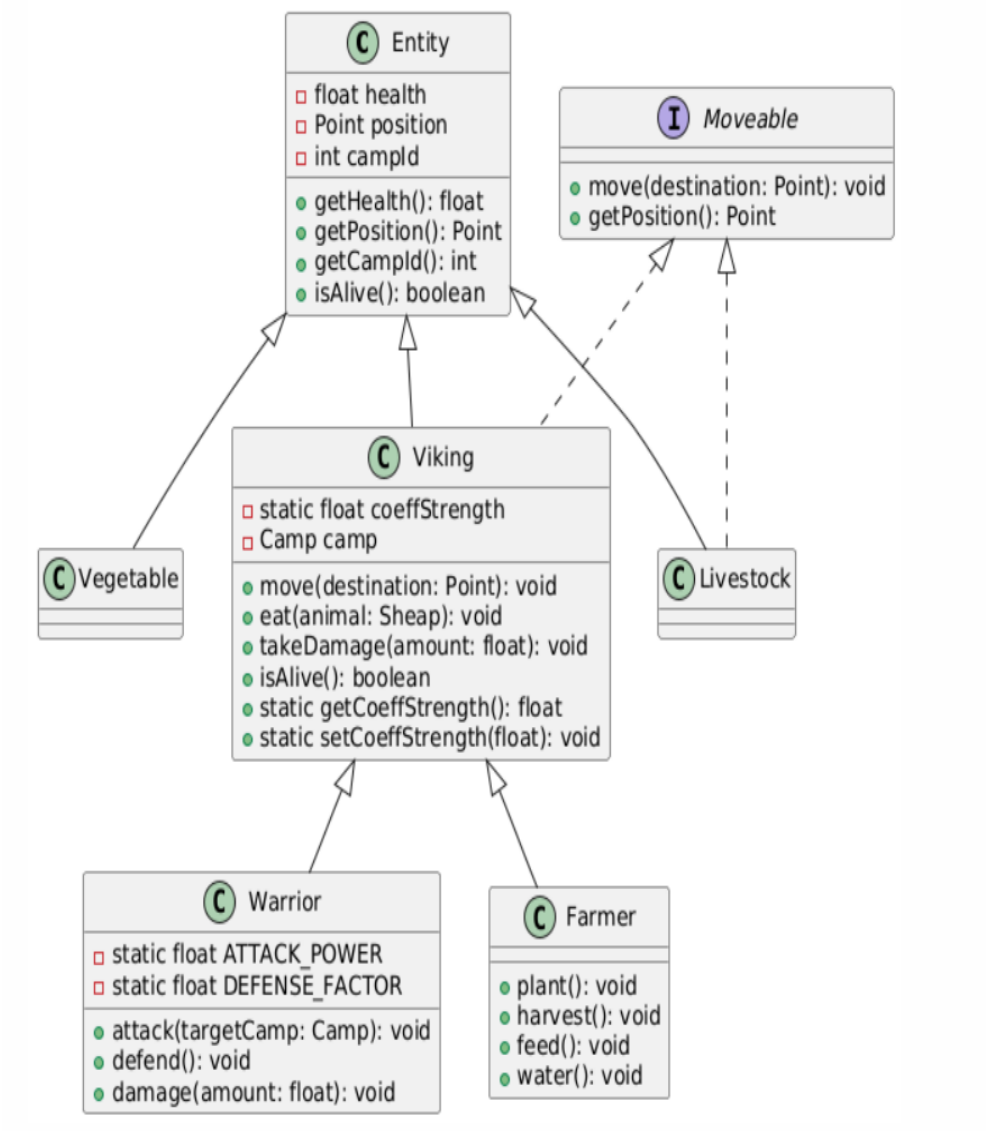
- `move(Point)`, `eat(Sheap)`, `takeDamage(float)`, `isAlive()`

Guerrier (`Warrior`) :

- `attack(Camp)`, `defend()`, `damage(float)`
- `startHealthDecayThread()`

Fermier (`Farmer`) :

- `plant()`, `harvest()`, `feed()`, `water()`



3.3.5 Ressources du camp

Classe abstraite `Vegetable`

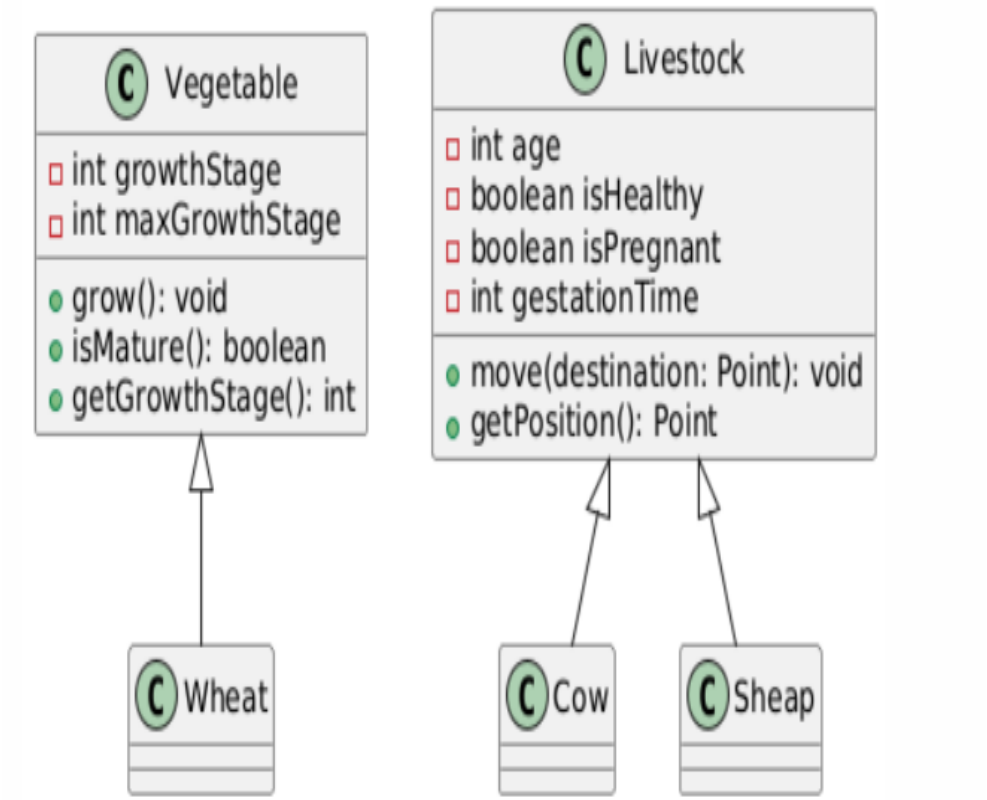
- Attributs : `growthStage`, `maxGrowthStage`
- Méthodes : `grow()`, `isMature()`

Blé (`Wheat`) : Implémentation concrète.

Classe abstraite `Livestock`

- `int age`, `boolean isHealthy`, `boolean isPregnant`
- `gestationTime`, `move(Point)`

Sheap et **Cow** : implémentations concrètes. `Sheap` peut être mangé.

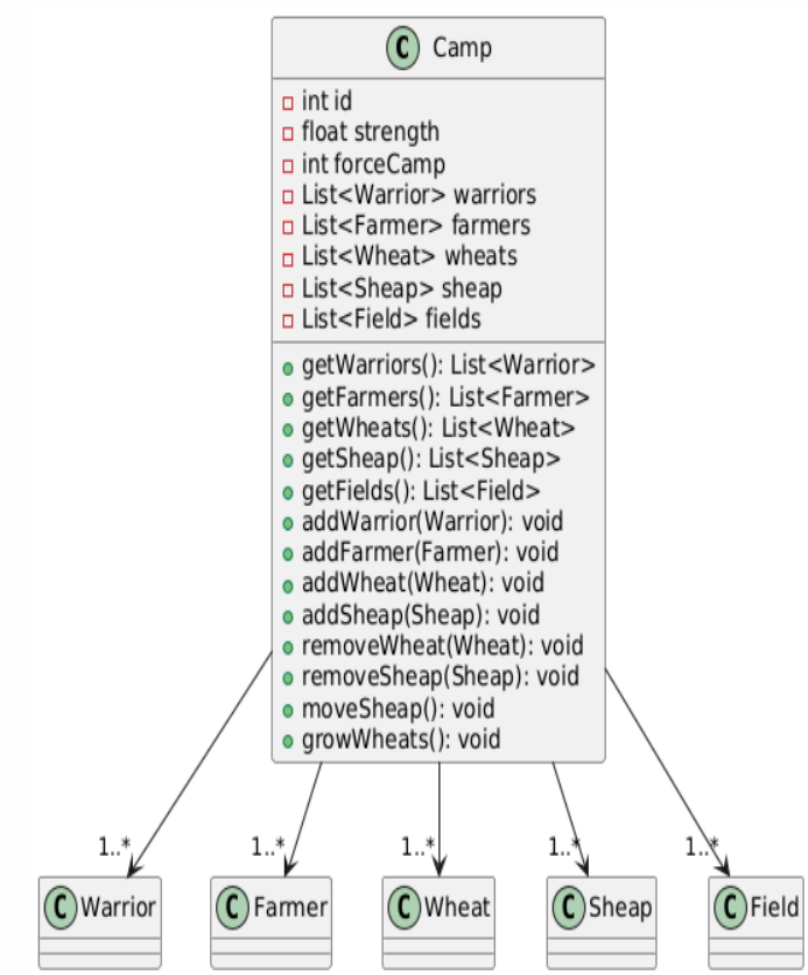

Vegetable
int growthStage
int maxGrowthStage
grow(): void
isMature(): boolean
getGrowthStage(): int
Livestock
int age
boolean isHealthy
boolean isPregnant
int gestationTime
move(destination: Point): void
getPosition(): Point
Wheat
Cow
Sheap

3.3.6

Classe `Camp`

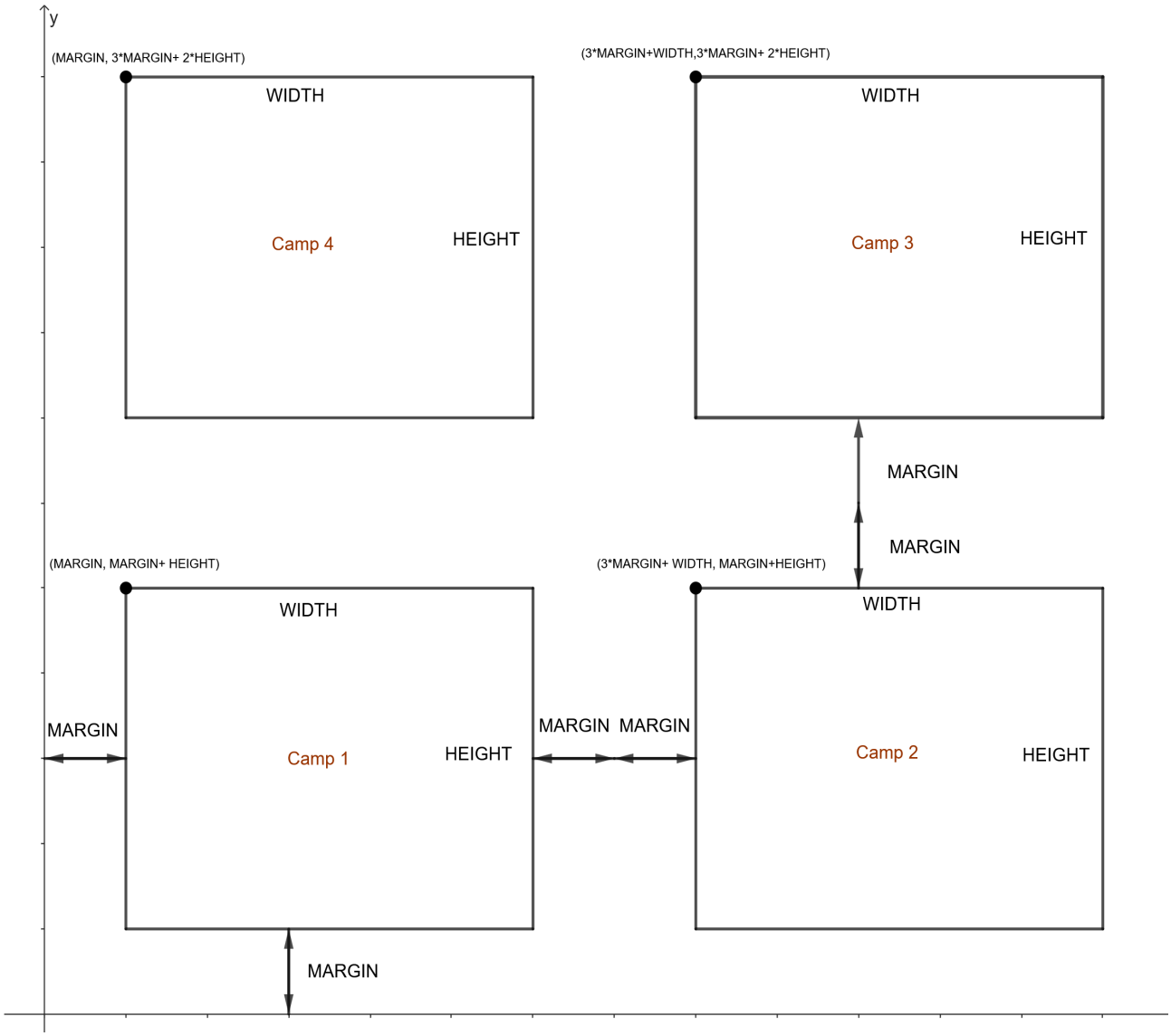
Classe centrale contenant :

- Collections : `ArrayList<Warrior>`, `Farmer`, `Sheap`, `Cow`, `Wheat`, `Field`
- Méthodes : `init()`, `addX()`, `growWheats()`, `moveSheap()`, `getFieldPositions()`, `getForce()`



3.4 Repères de coordonnées

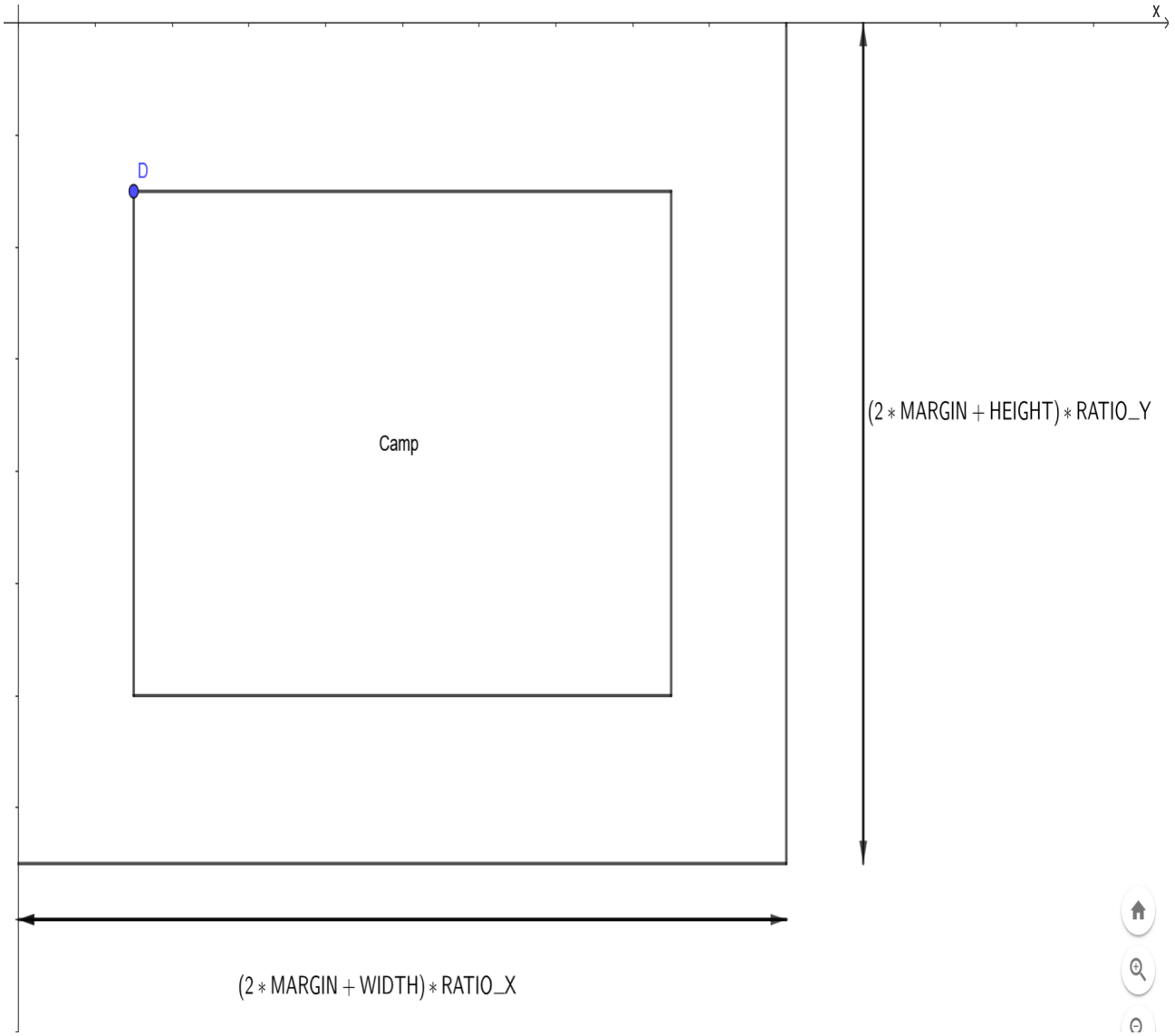
3.4.1 Repère modèle



Constantes modèle: Toutes les constantes qui forme le repère modèle sont dans **(server/model/Position.java)**.
Puis chaque camp est défini par son coin haut gauche

- Camp 1 : (margin, $margin + height$)
- Camp 2 :($3margin + width$, $margin + height$)
- Camp 3 : ($3margin + width$,$3margin + 2height$)
- Camp 4 : (margin, $3margin + 2height$)

3.4.2 Repère vu (Swing)



La fenêtre swing sera de largeur: $2margin + width \times RatioX$ avec RatioX et RatioY des constantes à définir dans la vue (ViewPartie.java), pour la hauteur: $2margin + height \times RatioY$. c'est l'espace qu'il faut pour afficher au client son camp.

Equation transformation: Pour passer un point de modèle (x,y) en un point de la vue swing (x',y'): x'=$x \times RatioX$, y'=$(height + 2 \times margin - y) \times RatioY$

Chaque vue doit connaitre son camp: Pour que chaque vue sachent sur quel camp zoomer il faut que le thread qui communique avec le client lui transmettent son campID, puis quand il boucle pour le dessin il regarde les ids des camp pour trouver le cien.

Mettre son camp sur la vu au lancement: maintenant qu'on sait c'est lequel notre camp, il faut faire les transformation necessaire pour ramener son camp à l'écran.

Pour ça on determine dans la vue le point ancre représenté dans la figure D=(margin', margin') (qui est (MARGIN*RATIOX,MARGIN*RATIOY)).

Il faut translater le point P=(x',y') de son camp à ce point A, pour ça il suffit de calculer le vecteur $\vec{DP} = (margin' - x', margin' - y')$.

Formule passage vu - modèle: Soit t, z, x', y' la translation, zoom, et les coordonées du clic respectivement à un moment données, les coordonnées corespendante (x,y) dans le repère modèle se calcule en appliquant la translation opposé (-t), l'inverse du zoome (1/z) et résoudre pour x et y les équations: x'=$x \times RatioX$, y'=$(height + 2 \times margin - y) \times RatioY$.
On aura : x = x'/RatioX et $y = height + 2 \times margin - (y'/RatioY)$

3.5 Défilement de la vu:

Pour implémenter le fait de pouvoir défiler la vu pour regarder les autres camp on utilise **(client/controler/ControlerParty.java)** qui implémente **MouseMotionListener**.
Pour implémenter le zoom le controleur implémente aussi **MouseWheelListener** pour detecter les évenements de la mollete.
Défilement de la vu: Deux algorithmes simples.

Algorithm 3: EnregistrerClicSouris

Input: Coordonnées du clic (x, y)

Output: Mémorisation de la position du clic précédent

1 `lastMouseClickX` $\leftarrow x$; `lastMouseClickY` $\leftarrow y$;

Algo 3: Est implémenté par le **mousePressed(Event)**

Algorithm 4: DéplacerVueAvecSouris

Input: Nouvelle position de la souris (x, y)

Output: Mise à jour de l'offset de la vue

1 $\Delta x \leftarrow x -$ `lastMouseClickX`; $\Delta y \leftarrow y -$ `lastMouseClickY`;

2 `viewPartie.addToOffset`$(\Delta x, \Delta y)$;

3 `lastMouseClickX` $\leftarrow x$; `lastMouseClickY` $\leftarrow y$;

Algo 4: Est implémenté par le **mouseDragged(Event)** qui détectement le glissement des cliques en Swing.
Pou ce qui est du zoom (**Algo 5**) il est implémenté dans le **mousWheelMoved(Event).**
Remarque:

- **mouseDragged(Event)** est une méthode de l'interface **mouseMotionListener** de la librairie java.awt.event.
- **mouseWheelMoved(Event)** est une méthode de l'interface **mouseWheelListener** de la librairie java.awt.event.

Algorithm 5: ZoomerDézoomerAvecMolette

Input: Position curseur (x, y), rotation molette r

Output: Modification du facteur d'échelle et de l'offset

$\texttt{zoomFactor} \leftarrow 1.1$;

$(\texttt{viewX}, \texttt{viewY}) \leftarrow (x, y)$;

$(\texttt{offsetX}, \texttt{offsetY}) \leftarrow$ offset total de la vue;

$s \leftarrow$ facteur d'échelle actuel de la vue;

$(\texttt{modelX}, \texttt{modelY}) \leftarrow \left(\frac{viewX - offsetX}{s}, \frac{viewY - offsetY}{s} \right)$;

if $(r > 0 \land s > 0.5) \lor (r < 0 \land s < 2.0)$ **then**

  $scaleChange \leftarrow \begin{cases} 1/\texttt{zoomFactor} & \text{si } r > 0 \\ \texttt{zoomFactor} & \text{si } r < 0 \end{cases}$;

  $\texttt{viewPartie.multiplyScale}(scaleChange)$; $s$new $\leftarrow$ nouveau facteur d'échelle;

  $(\texttt{newOffsetX}, \texttt{newOffsetY}) \leftarrow (viewX - modelX \cdot s\text{new}, viewY - modelY \cdot s_{\text{new}})$;

  $(\texttt{offsetCampX}, \texttt{offsetCampY}) \leftarrow$ offset statique du camp;

  $(dx, dy) \leftarrow (newOffsetX - offsetCampX, ; newOffsetY - offsetCampY)$;

  $\texttt{viewPartie.setOffset}(dx, dy)$;

3.6 Intéraction fermier champ de blé

FarmerFieldWrapper

Cette classe sert de conteneur pour les informations relatives à un agriculteur et à un champ. Elle encapsule les coordonnées de l'agriculteur, les coordonnées du champ, l'état de plantation du champ, et la santé de l'agriculteur.

Attributs :

- `farmerX`, `farmerY` : Coordonnées de l'agriculteur.
- `fieldX`, `fieldY` : Coordonnées du champ.
- `isPlanted` : Indique si le champ est planté.
- `farmerHealth` : Santé de l'agriculteur.

Méthodes :

- `Constructeur` : Initialise les attributs avec les valeurs passées en paramètre.
- `Getters` : `getFarmerX()`, `getFarmerY()`, `getFieldX()`, `getFieldY()`, `getIsPlanted()`, `getFarmerHealth()`.

FarmerPositionChecker

Cette classe vérifie périodiquement la position de l'agriculteur par rapport au champ le plus proche et notifie le serveur si l'agriculteur est à proximité d'un champ.

Attributs :

- `CHECK_INTERVAL_MS` : Intervalle de temps entre les vérifications de position.
- `communicationServer` : Serveur de communication pour envoyer des messages.
- `camp` : Camp contenant les champs.
- `farmer` : Agriculteur dont la position est vérifiée.
- `distanceTolerance` : Distance maximale pour considérer que l'agriculteur est près d'un champ.
- `previousNearFieldState` : État précédent de la proximité de l'agriculteur par rapport à un champ.

Méthodes :

- `Constructeur` : Initialise les attributs.

- `run()` : Méthode principale du thread qui vérifie périodiquement la position de l'agriculteur.
- `checkFarmerNearField()` : Vérifie si l'agriculteur est près d'un champ et envoie un message au serveur si l'état change.
- `isNearFieldWithMargin()` : Vérifie si l'agriculteur est à une distance donnée d'un champ.
- `getNearestField()` : Retourne le champ le plus proche de l'agriculteur.

PaquetPlant

Cette classe représente un paquet de données envoyé au serveur pour planter une ressource dans un champ.

Attributs :

- `resource` : Ressource à planter.
- `farmerX`, `farmerY` : Coordonnées de l'agriculteur.
- `fieldX`, `fieldY` : Coordonnées du champ.

Méthodes :

- `Constructeur` : Initialise les attributs.
- `Getters` : `getResource()`, `getFarmerX()`, `getFarmerY()`, `getFieldX()`, `getFieldY()`.
- `sendPlantPacketToServer()` : Envoie le paquet au serveur (méthode non implémentée complètement).

PanneauControle

Cette classe gère l'affichage du panneau de contrôle, y compris le menu coulissant et les boutons de contrôle.

Attributs :

- `slidingMenu` : Menu coulissant.
- `menuWidth`, `menuHeight`, `posMenuX`, `posMenuY` : Dimensions et position du menu.

Méthodes :

- `Constructeur` : Initialise le panneau de contrôle et le menu coulissant.

- `updatePosition()` : Met à jour la position du menu en fonction de la taille de la fenêtre.
- `updateSlidingMenuVisibility()` : Met à jour la visibilité du menu coulissant en fonction de la position de l'agriculteur.
- `setFarmerOnField()` : Met à jour l'état de l'agriculteur sur le champ et la visibilité du bouton de plantation.
- `elseWhereClicked()` : Cache le menu coulissant lorsque l'utilisateur clique ailleurs.

SlidingMenu

Cette classe représente un menu coulissant qui contient des boutons et des composants pour interagir avec le champ.

Attributs :

- `timer` : Timer pour l'animation du menu.
- `targetX` : Position cible du menu.
- `isFarmerOnField`, `isFieldPlanted`, `isVisible` : États du menu et du champ.
- `plantButton`, `plantComboBox`, `healthBar` : Composants du menu.
- `farmerX`, `farmerY`, `fieldX`, `fieldY` : Coordonnées de l'agriculteur et du champ.
- `plantListeners` : Liste des écouteurs d'événements de plantation.

Méthodes :

- `Constructeur` : Initialise le menu et ses composants.
- `updatePosition()` : Met à jour la position du menu.
- `toggle()`, `toggleVisible()`, `toggleHide()` : Gèrent la visibilité du menu.
- `slide()` : Anime le menu pour qu'il glisse vers sa position cible.
- `updatePlantButtonVisibility()` : Met à jour la visibilité du bouton de plantation en fonction de la position de l'agriculteur.
- `elseWhereClicked()` : Cache les composants du menu lorsque l'utilisateur clique ailleurs.
- `addPlantListener()` : Ajoute un écouteur d'événement de plantation.
- `handleComboBoxSelection()` : Gère la sélection d'une ressource dans le combo box et publie un événement de plantation.

EventBus

Cette classe implémente un bus d'événements pour gérer la publication et l'abonnement aux événements.

Attributs :

- `instance` : Instance singleton du bus d'événements.
- `eventListeners` : Map des écouteurs d'événements.

Méthodes :

- `Constructeur privé` : Empêche l'instanciation directe.
- `getInstance()` : Retourne l'instance singleton du bus d'événements.
- `subscribe()` : Ajoute un écouteur pour un type d'événement donné.
- `publish()` : Publie un événement et notifie les écouteurs.

PlantEvent

Cette classe représente un événement de plantation, contenant les informations sur la ressource plantée et les coordonnées de l'agriculteur et du champ.

Attributs :

- `resource` : Ressource plantée.
- `farmerX`, `farmerY`, `fieldX`, `fieldY` : Coordonnées de l'agriculteur et du champ.

Méthodes :

- `Constructeur` : Initialise les attributs.
- `Getters` : `getResource()`, `getFarmerX()`, `getFarmerY()`, `getFieldX()`, `getFieldY()`.

PlantListener

Cette interface définit un écouteur pour les événements de plantation.

Méthodes :

- `onPlant()` : Méthode appelée lorsqu'un événement de plantation est déclenché.

Interactions entre les classes

- `FarmerPositionChecker` vérifie la position de l'agriculteur et envoie des messages au serveur via `ThreadCommunicationServer`.
- `PanneauControle` et `SlidingMenu` gèrent l'interface utilisateur, affichant ou masquant les boutons en fonction de la position de l'agriculteur.
- `SlidingMenu` publie des événements de plantation via `EventBus`, qui sont ensuite traités par les écouteurs (`PlantListener`).
- `PaquetPlant` est utilisé pour encapsuler les données de plantation avant de les envoyer au serveur.

4 Attaque & défense

Attaque: Le client clique sur le camp à attquer, et sur la ressources à attaquer dans ce camp en précisant combien de vikings envoyer sur chaque ressource, après il soummet son attaque.
On envoi alors la requête à la soummission vers le serveur, et les vikings sont envoyés sur ces ressources.
Une fois arriver dessus, ils commencent à piler:

Pour faire ça le thread de déplacement doit prendre une sorte de callback à déclencher pour chaque déplacement d'entitée, dans le cas de l'attaque il déclenchera des méthodes pour réduire les ressources du camp attaqué et les donné à l'attaquant.
Défense: le client clique sur les ressources de son camp choisit l'option défendre, avec un nombre défini de vikings.

Confrontement de viking: à partir d'une certaine distance entre des vikings de camps différent. les attaquant abandonne leur pilage pour aller se confronter (On pourra annuler le confrontement en ordonnant le retour vers le camp.
Pour les dégats causés, une attaque cause des dégats relatifs à la santé du viking