helb
Ilya Prigogine

HELBArmy

Seconde Session

Nom : Sadiqi
Prénom : Amir
Intitulé du cours : Programmation Java III
Nom de l'enseignant : Mr Riggio Jonathan

Table des matières

Introduction

Dans le cadre du cours de Java III, une simulation appelée HELBArmy a été développée pour modéliser l'affrontement stratégique entre deux armées opposées. Le jeu se déroule sur une carte divisée en cases, où chaque unité et chaque élément occupe une position bien définie. Chaque unité est définie par des propriétés précises (position, type, cible).Les unités sont générées automatiquement par leur ville d'origine, puis se déplacent selon leur rôle pour interagir avec les ressources et les obstacles présents sur le terrain.

L'objectif du projet a été de concevoir une application fonctionnelle en JavaFX en respectant les contraintes techniques imposées par l'énoncé. Le développement a été amorcé à partir de la base SnakeFX, qui a été adaptée pour intégrer les nouvelles mécaniques de jeu. Les règles de gestion des ressources, des déplacements et des interactions entre unités ont été codées conformément aux spécifications. L'application permet ainsi une visualisation en temps réel de la simulation, offrant une expérience interactive et intéressante.

Toutes les fonctionnalités ont été implémentées, à l'exception de la gestion des combats entre plusieurs unités. Cette limitation est due à la complexité de la gestion des collisions entre unités, à la difficulté à gérer les affrontements entre unités placées sur des cases adjacentes, et à un temps de développement limité. Malgré cela, la simulation reste pleinement fonctionnelle et illustre efficacement les principes de la programmation orientée objet et de l'interaction graphique en JavaFX.

Structure du rapport

Ce rapport est structuré de manière à offrir une vue claire et détaillée du projet :

Fonctionnalités de base : Description détaillée des fonctionnalités principales du projet afin de permettre au lecteur de comprendre ce qui a été implémenté et comment elle fonctionne.

Analyse : Présentation de la structure du projet à l'aide de diagrammes pour clarifier la logique et les choix de conception.

Limitations : Analyse des limites de la simulation et réflexion sur les améliorations possibles, en détaillant les contraintes rencontrées et les solutions envisageables avec plus de temps.

Conclusion : Bilan du projet, mettant en avant les réussites et les difficultés rencontrées, accompagné d'une réflexion personnelle sur les apprentissages acquis au cours du développement, illustrée par des exemples concrets.

Fonctionnalités

La Grille de Jeu

La simulation repose sur une grille composée de lignes et de colonnes, représentant l'espace de jeu. Chaque élément, qu'il s'agisse d'une unité, ou d'une ressource, y est positionné de manière unique. Aucun chevauchement n'est autorisé, sauf exception précisée dans les règles. Les bords de la carte sont considérés comme infranchissables, ce qui limite les déplacements des unités.

```
public boolean isCoordinateInBoard(GameElement coord, int maxX, int maxY) {
    return coord.getX() >= INIT_INDEX && coord.getY() >= INIT_INDEX
            && coord.getX() < maxX && coord.getY() < maxY;
}
```

La méthode *isCoordinateInBoard* vérifie si une position est sur le plateau. Elle retourne true si x et y sont ≥ 0 et inférieurs à la largeur et hauteur, assurant que les coordonnées restent dans les limites et respectent les bords infranchissables.

Positionnement des Éléments

Différents types d'éléments, comme les arbres et les villes, ont été placés sur la carte. Les arbres et les rochers sont répartis aléatoirement. Les arbres contiennent initialement 50 unités de bois et peuvent en contenir jusqu'à 100. Les rochers contiennent initialement 100 unités de minerai, peuvent en contenir jusqu'à 200 et occupent une zone de 2x2 cases. Tant que ces éléments sont présents, ils bloquent le passage des unités et disparaissent une fois entièrement récoltés par un collecteur.

Deux villes ont été placées aux extrémités opposées de la carte, chacune chargée de générer les unités de son armée. Ces villes sont positionnées en case (0,0), en haut à gauche et en case (n,m), en bas à adroite , selon les dimensions de la grille. Les unités créées par les villes sont ensuite envoyées sur le terrain pour accomplir leur mission.

```
public void setupCity() {  1 usage   asadiqi
    int lastRow = GRID_ROWS - LAST_INDEX_OFFSET;
    int lastCol = GRID_COLS - LAST_INDEX_OFFSET;

    northCity = new City(new GameElement(INITIAT_INDEX, INITIAT_INDEX),  isNorth: true);
    southCity = new City(new GameElement(lastRow, lastCol),  isNorth: false);

    allElements.add(northCity);
    allElements.add(southCity);

    System.out.println("North city added on cell: "+northCity.getX() + " " + northCity.getY());
    System.out.println("South city added on cell: "+southCity.getX() + " " + southCity.getY());
}
```

La méthode *setupCity* place les villes aux extrémités de la grille, (0,0) pour le Nord et (n,m) pour le Sud. La logique est de garantir que chaque ville commence à une position fixe et opposée pour organiser la génération des unités sur le terrain.

Défis Techniques

Pendant le développement des arbres et des rochers, un défi technique a été rencontré pour placer ces éléments sans chevauchement sur la grille, surtout pour les éléments plus grands comme les rochers 2x2. La solution a été de créer une méthode qui génère aléatoirement des positions et vérifie si elles sont libres avant de placer un élément.

```java
public static GameElement getRandomFreeCell(int gridCols, int gridRows, List<GameElement> allElements) {
    Random rand = new Random();
    int maxAttempts = MAX_RANDOM_ATTEMPTS; // Nombre maximum d'essais pour trouver une case libre
    for (int i = INIT_INDEX; i < maxAttempts; i++) {
        int x = rand.nextInt(gridCols);
        int y = rand.nextInt(gridRows);

        if (!isOccupied(x, y, allElements)) {
            return new GameElement(x, y);
        }
    }
    System.out.println("No free position found after " + maxAttempts + " attempts.");
    return NO_POSITION; // Retourne une position null
}
```

La méthode *getRandomFreeCell* permet de trouver une case libre pour placer un arbre, un rocher ou une ville sans chevauchement. Elle teste plusieurs positions aléatoires et renvoie la première libre. Si aucune n'est disponible, elle renvoie une valeur spéciale (null) pour signaler l'échec, garantissant le respect des règles de la grille.

Les unités

Collecteur

Les collecteurs ont été conçus pour récolter les ressources disponibles sur la carte Après leur création, les collecteurs se dirigent automatiquement vers la ressource la plus proche. La collecte s'effectue depuis une case adjacente, et chaque seconde, l'unité récoltée par les collecteurs est transférée à la ville d'origine.
Les deux types de collecteurs ont été implémentés, les bûcherons et les piocheurs.
Les bûcherons récoltent deux unités de bois par seconde sur un arbre, les piocheurs trois unités de minerai par seconde sur un rocher. Chaque collecteur peut cependant récolter les deux types de ressources selon la case sur laquelle il se trouve.

La génération des collecteurs dépend des ressources présentes : plus il y a d'arbres sur la carte, plus les chances de créer un bûcheron sont élevées. Plus il y a de rochers, plus les chances de créer un piocheur augmentent.

```
public void generateCollectorBasedOnResources(List<Tree> trees, 5 usages  asadiqi *
                                               List<Stone> stones,
                                               List<GameElement> allElements,
                                               int gridCols, int gridRows,
                                               int maxDistance) {
    int totalResources = trees.size() + stones.size();
    double lumberjackProbability = (totalResources == INIT_INDEX) ?
            DEFAULT_LUMBERJACK_PROBABILITY : (double) trees.size() / totalResources;
    GameElement pos = findPlacementForUnit(allElements, gridCols, gridRows, maxDistance);
    addUnitIfPossible(allElements, pos, new Collecter(pos, city: this, isLumberjackCollecter: Math.random() < lumberjackProbability));
}
```

La méthode *generateCollectorBasedOnResources* crée un collecteur selon les ressources présentes. Elle calcule la probabilité qu'il soit bûcheron ou piocheur, cherche une case libre proche de la ville et ajoute le collecteur à la grille. La logique est d'adapter le type de collecteur à l'environnement pour optimiser la récolte.

```
public boolean removeIfDepleted(List<GameElement> allElements) {  2 usages  asadiqi
    if (isDepleted()) {
        allElements.remove( o: this);
        System.out.println(getResourceName() + " removed: " + getX() + "," + getY());
        return true;
    }
    return false;
}
```

La méthode *removeIfDepleted* supprime la ressource de la grille si elle est épuisée. Cela garantit que les arbres ou rochers disparaissent automatiquement lorsque leur quantité atteint zéro, évitant qu'une unité collecte une ressource inexistante.

Les Semeurs

Les semeurs génèrent de nouvelles ressources sur la carte. Lorsqu'un semeur atteint une case adjacente à une zone libre, le semis commence. Deux unités de ressource sont ajoutées chaque seconde jusqu'à atteindre le maximum. Pour un arbre, le semeur choisit une case libre proche d'un arbre existant, ou aléatoirement s'il n'y en a pas. Pour un rocher, la position la plus éloignée des autres rochers est privilégiée.

```
public void chooseTarget(String resourceType,   2 usages   asadiqi *
                         List<? extends Resource> resources,
                         int maxX, int maxY, List<GameElement> occupied) {

    this.targetRessourceType = resourceType;
    if (TREE_TYPE.equals(resourceType)) {
        chooseTreeTarget((List<Tree>) resources, maxX, maxY, occupied);
    } else if (STONE_TYPE.equals(resourceType)) {
        chooseStoneTarget((List<Stone>) resources, maxX, maxY, occupied);
    }
}
```

La méthode *chooseTarget* permet au semeur de choisir sa cible selon le type de ressource. Si le type est arbre, elle appelle *chooseTreeTarget* pour sélectionner et planter un arbre. Si c'est un rocher, elle appelle *chooseStoneTarget* pour sélectionner et planter un rocher. Elle stocke aussi le type de ressource ciblé dans *targetRessourceType*.

Les Assassins

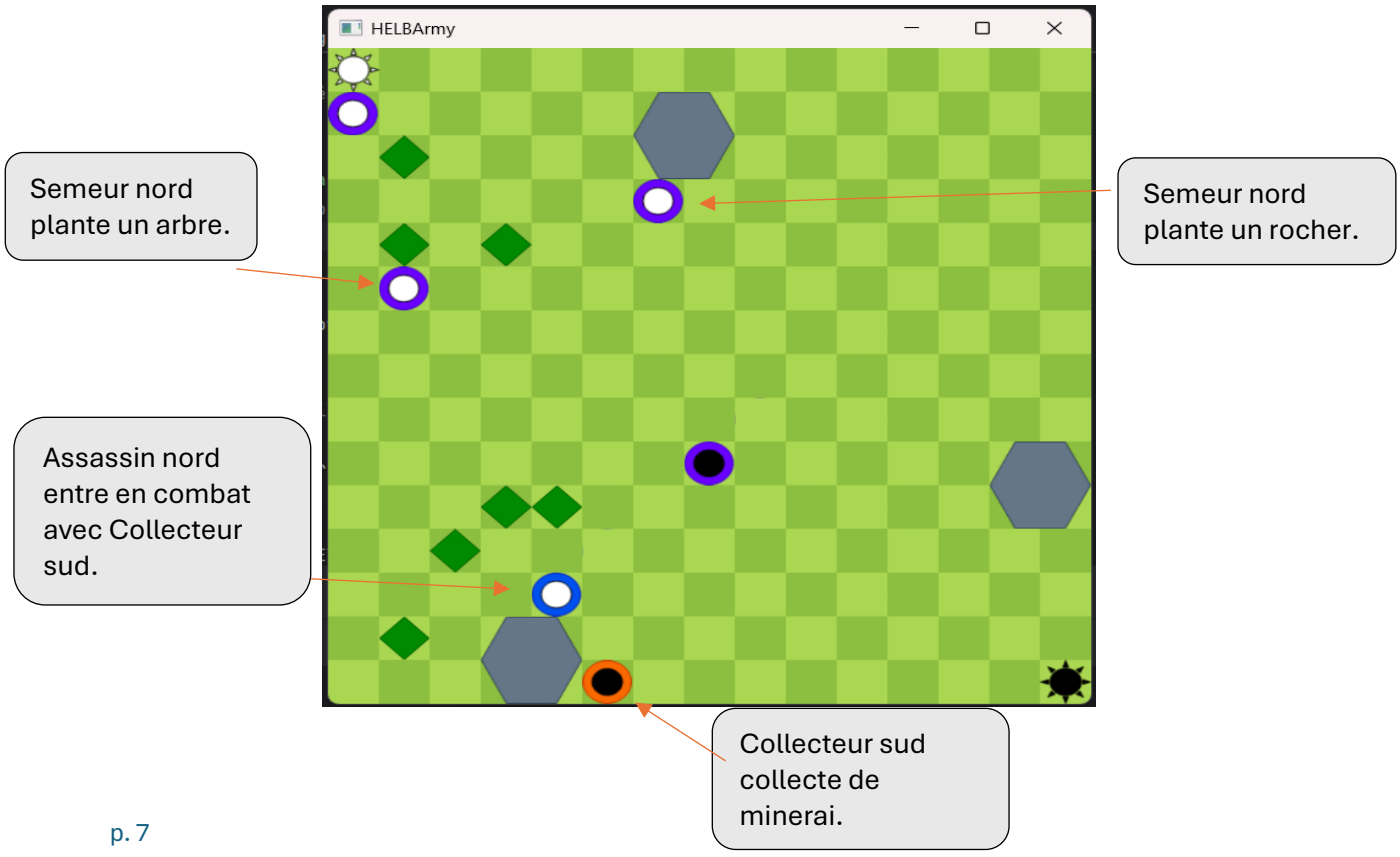
Les assassins traquent les unités ennemies. Leur priorité cible les assassins adverses, puis les collecteurs, et enfin les semeurs. Lorsqu'un assassin est adjacent à une unité ennemie, un combat se déclenche. Un dé à deux faces est lancé, et l'unité correspondant au résultat est éliminée du terrain.

Défis Techniques: À ce stade, un problème a été rencontré et la planification de la fonctionnalité de combat a été un peu difficile. Il a été prévu que l'assassin recherche ses adversaires selon une priorité, d'abord les assassins ennemis, puis les collecteurs, et enfin les semeurs.
La méthode *handleAssassin* a été implémentée pour gérer ce comportement.

```
public void handleAssassin(int gridCols, int gridRows, List<GameElement> allElements) {  1 usage   asadiqi *
    List<GameElement> enemyAssassins = new ArrayList<>();
    List<GameElement> enemyCollectors = new ArrayList<>();
    List<GameElement> enemySeeders = new ArrayList<>();

    for (GameElement element : allElements) {
        if (element instanceof Assassin a && a != this && a.city.isNorth != this.city.isNorth) {
            enemyAssassins.add(a);
        } else if (element instanceof Collecter c && c.city.isNorth != this.city.isNorth) {
            enemyCollectors.add(c);
        } else if (element instanceof Seeder s && s.city.isNorth != this.city.isNorth) {
            enemySeeders.add(s);
        }
    }
```

```java
// 2. Trouver la cible prioritaire (assassin > collecteur > semeur)
GameElement targetEnemy = findClosestInList(enemyAssassins);
if (targetEnemy == GameElement.NO_POSITION) targetEnemy = findClosestInList(enemyCollectors);
if (targetEnemy == GameElement.NO_POSITION) targetEnemy = findClosestInList(enemySeeders);

// 3. Vérifier si l'ennemi est adjacent pour le combat
if (targetEnemy != GameElement.NO_POSITION) {
    int dx = Math.abs(this.x - targetEnemy.getX());
    int dy = Math.abs(this.y - targetEnemy.getY());
    if (dx <= MAX_ADJACENT_DISTANCE  && dy <= MAX_ADJACENT_DISTANCE) {
        // Combat aléatoire
        int winner = (int) (Math.random() * COMBAT_OUTCOMES );
        if (winner == COMBAT_WINNER_ATTACKER) {
            allElements.remove(targetEnemy);
            System.err.println(this + " defeated " + targetEnemy);

            // cette unité continue son déplacement
        } else {
            allElements.remove( o: this);
            System.err.println(this + " was defeated by " + targetEnemy);
            return; // cette unité est supprimée, on arrête
        }
    } else {
        setTarget(targetEnemy);
        moveTowardsTarget(gridCols, gridRows, allElements);
    }
```

La méthode *handleAssassin* gère le comportement d'un assassin. Les ennemis sont d'abord classés par type et par camp. La cible prioritaire est ensuite choisie d'abord les assassins, puis les collecteurs, et enfin les semeurs. Si la cible est adjacente, un combat aléatoire est déclenché et l'unité perdante est supprimée du terrain. Si la cible n'est pas à portée, l'assassin se déplace vers elle. Cette méthode permet de gérer automatiquement la recherche de cible, le combat et le déplacement de l'assassin.



Les Collectables

Le Drapeau

À intervalles réguliers (toutes les deux minutes), un drapeau apparaît sur une case choisie aléatoirement sur la carte. Il reste visible pendant une courte durée (10 secondes), puis disparaît automatiquement s'il n'est pas récupéré. Sa apparition provoque une perturbation temporaire telle que toutes les unités interrompent leurs actions habituelles et adoptent un déplacement aléatoire sur le terrain. Dès que le drapeau disparaît, soit par expiration du temps, soit par collecte les unités reprennent leur comportement normal.

```java
public static Flag createFlagIfNone(List<GameElement> allElements, int gridRows, int gridCols) {
    for (GameElement e : allElements)
        if (e instanceof Flag) return NO_FLAG;

    GameElement freePos = GameElement.getRandomFreeCell(gridCols, gridRows, allElements);
    for (GameElement e : allElements)
        if (e.getX() == freePos.getX() && e.getY() == freePos.getY() && e instanceof MagicStone)
            return NO_FLAG;
    if (!freePos.equals(GameElement.NO_POSITION)) {
        Flag newFlag = new Flag(freePos);
        allElements.add(newFlag);
        return newFlag;
    }
    return NO_FLAG; // pas de place pour le drapeau
}
```

La méthode *createFlagIfNone* vérifie s'il n'existe pas déjà un drapeau. Si ce n'est pas le cas, elle cherche une case libre, y place un nouveau drapeau et le retourne.

```java
private void createFlag() {  1 usage   asadiqi *
    flagTimeline = new Timeline(
            new KeyFrame(Duration.seconds(FLAG_CREATE_DELAY_SEC ), e -> {
                currentFlag = Flag.createFlagIfNone(allElements, GRID_ROWS, GRID_COLS);
                view.drawAllElements();
            }),
            new KeyFrame(Duration.seconds(FLAG_REMOVE_DELAY_SEC ), e -> {
                allElements.remove(currentFlag);
                currentFlag = Flag.NO_FLAG;
                view.drawAllElements(); // Met à jour l'affichage pour retirer le drapeau
            })
    );
    flagTimeline.setCycleCount(Animation.INDEFINITE);     // La timeline se répète indéfiniment
    flagTimeline.play();
}
```

Ensuite la méthode *createFlagIfNone* est utilisée dans le contrôleur dans la méthode *createFlag* pour créer automatiquement un drapeau après 120 seconde, l'afficher, puis le retirer après 10 seconde. Le cycle se répète indéfiniment grâce à une Timeline.

La pierre philosophale

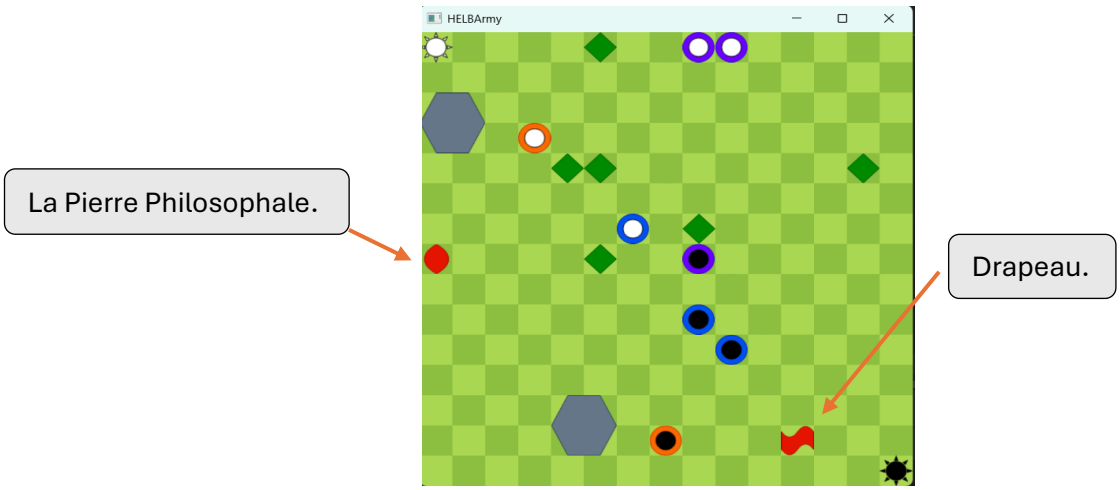
La pierre philosophale agit comme un portail magique. Lorsqu'une unité entre en contact avec elle, elle est immédiatement téléportée vers un autre point aléatoire de la carte. Contrairement au drapeau, la pierre ne disparaît jamais et peut être utilisée plusieurs fois.

```java
public static void createMagicStoneAtRandomPosition(List<GameElement> allElements, int gridRows, int gridCols) {
    GameElement freePos = GameElement.getRandomFreeCell(gridCols, gridRows, allElements);
    for (GameElement e : allElements) {
        if (e.getX() == freePos.getX() && e.getY() == freePos.getY() && e instanceof Flag) {
            System.out.println("MagicStone not created: cell occupied by a Flag.");
            return;
        }
    }
    if (!freePos.equals(GameElement.NO_POSITION)) {
        allElements.add(new MagicStone(freePos));
    } else {
        System.out.println("No free position available to create a MagicStone.");
    }
}
```

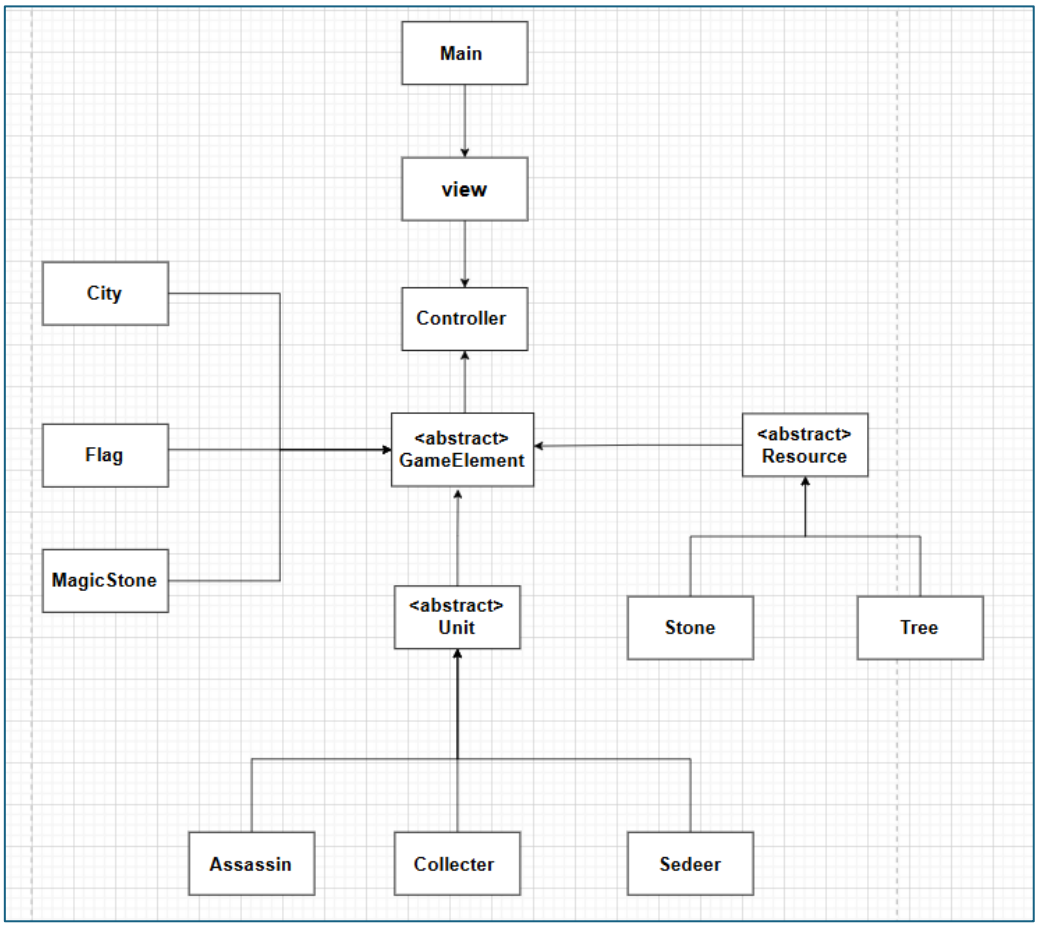
La logique de la méthode *createMagicStoneAtRandomPosition est de* placer une pierre magique sur une cellule libre choisie au hasard, seulement si elle n'est pas occupée par un drapeau. Aucune pierre n'est créée si aucune position libre n'est trouvée.

```java
public void handleKeyPress(KeyCode code) {  1 usage   asadiqi *
    if (isKeyPressBlocked) {
        return;
    }
    switch (code) {
        case P -> MagicStone.createMagicStoneAtRandomPosition(allElements, GRID_ROWS, GRID_COLS);

        default -> {}
    }
    view.drawAllElements();   // Redessine tous les éléments après chaque action

}
```

La méthode *createMagicStoneAtRandomPosition* est appelée dans *handleKeyPress* quand la touche P est pressée. Cela permet de créer une pierre magique sur une position libre, si possible, puis de mettre à jour l'affichage.



Analyse



Dans le projet HELBArmy, plusieurs classes ont été introduites pour structurer le code de manière cohérente et conforme aux principes de la programmation orientée objet (POO). Ces classes ne sont pas de simples choix techniques elles répondent à des besoins concrets de conception.

La classe *GameElement* sert de classe de base pour tous les éléments placés sur la carte. Elle définit des propriétés communes comme la position ainsi que des méthodes pour récupérer ces coordonnées, calculer des distances ou vérifier si une case est libre. Elle fournit aussi des méthodes utilitaires pour placer un élément aléatoirement sur la grille. Grâce à cette abstraction, toutes les entités du jeu (villes, ressources, unités, drapeaux, pierres magiques) peuvent être manipulées de manière uniforme, ce qui simplifie la gestion du plateau et le code des autres classes.

La classe *Unit* définit le comportement commun des unités. Elle gère le déplacement, le suivi d'une cible et l'évitement d'obstacles. Les classes comme *Collector*, *Semeur* et *Assassin* héritent de cette logique pour ne réimplémenter que leur comportement spécifique, ce qui réduit la duplication de code et facilite la maintenance.

La classe *Resource* définit le comportement commun des éléments récoltables. Elle gère la quantité disponible, la croissance et la suppression quand la ressource est épuisée. Les classes *Tree* et *Stone* héritent de cette logique et appliquent leurs propres règles, ce qui centralise la gestion des ressources tout en permettant des comportements spécifiques.

Voici une analyse complète de toutes les classes du projet HELBArmy et de leurs rôles dans l'architecture :

Main

Rôle : Point d'entrée de l'application.

- **Motivation** : Centraliser l'initialisation du système, instanciant la classe View et Controller pour initialiser l'affichage et les interactions.

Controller

Rôle : Coordonne les actions du jeu.

Motivation : Séparer la logique métier de l'affichage. Elle gère les interactions entre les unités, les ressources et les objets spéciaux, permettant une simulation cohérente.

View

Rôle : Gère l'affichage graphique.

Motivation : Offrir une visualisation claire et dynamique de l'état du jeu. Elle permet aux utilisateurs de suivre l'évolution de la simulation en temps réel, tout en respectant le principe de séparation des responsabilités.

GameElement *(Abstraite)*

Rôle : Classe mère de tous les éléments sur la carte.

Motivation : Unifier les comportements communs (position, calculer des distances etc) pour éviter la duplication de code et faciliter l'extension du système.

City

Rôle : Hérite de *GameElement* et représente une ville génératrice d'unités et collectrice de ressources.

Motivation : La ville est un endroit important dans le jeu. Elle sert de point de départ pour les unités et permet de stocker les ressources récoltées.

Resource *(Abstraite)*

Rôle : Hérite de *GameElement* et représente une ressource récoltable.

Motivation : Modéliser les éléments exploitables du terrain. Permet de gérer la logique de collecte et d'épuisement, tout en étant extensible pour différents types de ressources.

Tree

Rôle : Hérite de *Resource* et représente une source de bois exploitable.

Motivation : Modéliser la ressource bois avec ses valeurs par défaut et ses règles spécifiques, comme la suppression des arbres épuisés et leur génération aléatoire.

Stone

Rôle : Hérite de Resource et représente une source de minerai exploitable.

Motivation : Gérer la quantité, la taille et les cellules occupées par le rocher, supprimer les pierres épuisées et générer de nouvelles pierres aléatoirement.

Unit *(Abstraite)*

Rôle : Hérite de *GameElement* et représente une unité mobile.

Motivation : Centraliser les comportements de déplacement, de ciblage et d'interaction. Permet de créer facilement de nouveaux types d'unités avec des rôles spécifiques.

Collector

Rôle : Hérite d'*Unit* et récolte les ressources.

Motivation : Déplacer l'unité vers la ressource la plus proche, appliquer des bonus selon le type de collecteur, réduire la ressource et mettre à jour le stock de la ville.

Semeur

Rôle : Hérite d'*Unit* et plante des arbres et des pierres pour la ville.

Motivation : Identifier une ressource cible, se déplacer vers elle, planter l'arbre ou la pierre si l'emplacement est libre, suivre sa croissance et choisir une nouvelle cible une fois mature.

Assassin

Rôle : Hérite d'*Unit*, se déplace sur le terrain puis attaque les ennemis.

Motivation : Identifier les ennemis les plus proches, priorise les unité selon type, se déplacer vers les ennemis pour les attaquer ou se repositionner si aucun n'est proche.

Flag

Rôle : Objet à capturer sur la grille.

Motivation : Introduire un événement aléatoire qui modifie le comportement des unités.

MagicStone

Rôle : Objet magique placé aléatoirement sur la grille.

Motivation : Offrir une mécanique de surprise et changer le déplacement des unités.

Limitations

Malgré une implémentation complète des fonctionnalités principales, plusieurs limitations ont été identifiées, tant sur le plan technique que logique. Ces limites reflètent les contraintes de temps, la complexité du projet, et les choix de conception réalisés durant le développement.

Gestion incomplète des combats

La fonctionnalité de combat entre plusieurs unités n'a pas été finalisée. Actuellement, les combats ne peuvent avoir lieu qu'entre deux unités adjacentes, avec un dé à deux faces. Cela limite la richesse du jeu, notamment lors de regroupements massifs d'unités. Si j'avais eu plus de temps, j'aurais implémenté une logique de combat en groupe, avec un dé à plusieurs faces et une boucle de résolution jusqu'à ce qu'un seul survivant reste. Cela aurait permis des affrontements plus dynamiques et réalistes.

Placement des ressources aléatoire et statique

Dans le projet actuel, les arbres et rochers sont placés de façon aléatoire au démarrage, et leur apparition ultérieure dépend uniquement des semeurs. Cela crée des zones de la carte peu exploitées et d'autres trop saturées, générant un déséquilibre. Si j'avais plus de temps, je mettrais en place un système de génération équilibrée des ressources, qui tiendrait compte de la position des unités et des villes. Cela garantirait une distribution plus juste des ressources sur la carte.

Événements spéciaux limités

Actuellement, seuls deux objets spéciaux existent , le drapeau et la pierre philosophale, avec des effets simples comme la perturbation aléatoire ou la téléportation. Cette limitation réduit la variété et rend le jeu prévisible, avec peu d'interactions surprenantes. Une amélioration possible serait d'introduire plusieurs types d'objets spéciaux avec des effets plus variés. Par exemple, des boosts temporaires qui augmentent la vitesse ou la force des unités, des ralentissements qui freinent les mouvements des adversaires, des pièges qui infligent des pénalités, des objets déclenchés par des conditions spécifiques comme le passage sur une case ou un certain nombre de tours écoulés. Ces événements rendraient le jeu plus dynamique et intéressant, en ajoutant de la stratégie et de la surprise à chaque partie.

Gestion des exceptions et erreurs

Le projet ne gère pas toutes les erreurs possibles, par exemple lorsqu'il n'y a plus de cases libres pour placer une ressource ou une unité. Dans ces situations, le programme peut se bloquer ou planter, ce qui affecte la stabilité de l'application et perturbe le déroulement du jeu. Si j'avais plus de temps, je mettrais en place une gestion complète des exceptions. Cela inclurait des vérifications systématiques avant chaque placement pour s'assurer qu'il reste des cases disponibles. En cas d'absence de place, le programme pourrait choisir automatiquement une alternative, comme déplacer la ressource vers une zone proche libre, repousser l'action à un tour suivant, ou alerter l'utilisateur avec un message clair. Ces améliorations garantiraient que le jeu continue de fonctionner même dans des scénarios extrêmes, en évitant les plantages et en améliorant l'expérience globale.

Limitation de la génération rapide des unités et solution d'optimisation

Une autre limitation du projet est la génération d'unités toutes les deux secondes. Ce rythme peut rapidement saturer la carte, entraînant des ralentissements, des collisions fréquentes et une expérience de jeu moins fluide. Si j'avais plus de temps, je mettrais en place un système de contrôle du flux des unités. Par exemple, limiter le nombre maximum d'unités simultanées sur la carte, espacer la génération selon la densité actuelle ou introduire des priorités pour créer les unités seulement là où elles sont nécessaires. Cela permettrait de maintenir la fluidité du jeu tout en conservant un défi stratégique.

Conclusion

Le projet HELBArmy a été conçu pour créer une application JavaFX simulant une interaction stratégique entre deux armées. Une grille de jeu a été mise en place, les villes et les ressources ont été positionnées sur la grille, et le comportement des unités a été bien développé. Les collecteurs et semeurs ont été programmés pour interagir avec les ressources, tandis que les assassins ont été configurés pour suivre leur logique de déplacement et de ciblage. Les objets spéciaux comme le drapeau et la pierre philosophale, ont été intégrés afin d'apporter des mécanismes de surprise et de perturbation.

Lors de développent de ce projet des défis techniques ont été rencontrés, notamment le placement des éléments sans chevauchement, la génération aléatoire des unités selon les ressources et la gestion des interactions sur la grille. Ces problèmes ont été résolus par des méthodes de vérification des positions libres, des algorithmes de génération aléatoire contrôlée et des règles claires pour les interactions entre unités.
Les principes de la programmation orientée objet ont été appliqués pour centraliser les comportements communs dans des classes abstraites et spécifiques. L'architecture MVC a été utilisée pour séparer la logique métier de l'interface graphique.

Certaines limitations ont été constatées, telles que la gestion des combats entre plusieurs unités et le placement aléatoire imparfait des ressources. Ces contraintes ont permis de réfléchir à des solutions optimisées et d'anticiper les évolutions possibles du projet.

Dans l'ensemble, le projet HELBArmy m'a permis de renforcer mes compétences en JavaFX, en programmation orientée objet et en gestion de projet. J'ai appris à planifier des fonctionnalités complexes, résoudre des problèmes techniques et structurer un code clair et facile à modifier. La simulation montre l'importance d'organiser correctement les classes et d'adapter mes méthodes aux besoins du projet.