Walid El Aidouni El Idrissi

Monsieur Riggio

**Une image contenant peinture, cheval, art, plein air

Description générée automatiquement**

Projet Java III - 2024-2025 **HELBArmy**

BAC-2

Table des matières

[**Introduction** 2](#_Toc186384600)

[**Fonctionnalités de base** 3](#_Toc186384601)

[**L’espace de jeu** 3](#_Toc186384602)

[**Les arbres :** 3](#_Toc186384603)

[**Les villes :** 4](#_Toc186384604)

[**Les unités :** 4](#_Toc186384605)

[**Les collecteurs :** 4](#_Toc186384606)

[**Les déserteurs :** 5](#_Toc186384607)

[**Les cavaliers :** 5](#_Toc186384608)

[**Les piquiers :** 5](#_Toc186384609)

[**Système de combat :** 6](#_Toc186384610)

[**Système de déplacement :** 6](#_Toc186384611)

[**Déplacement et gestion des obstacles :** 7](#_Toc186384612)

[**Vitesses et gestion du temps :** 7](#_Toc186384613)

[**Génération des unités :** 7](#_Toc186384614)

[**La Pierre Philosophale :** 7](#_Toc186384615)

[**Le Drapeau :** 8](#_Toc186384616)

[**Système de cheat codes :** 8](#_Toc186384617)

[**Analyse** 9](#_Toc186384618)

[**Diagramme UML - Architecture du projet HELBArmy** 9](#_Toc186384619)

[**Relations entre les classes dans HELBArmy** 10](#_Toc186384620)

[**Limitations** 13](#_Toc186384621)

[**1. Comportement du Cavalier** 13](#_Toc186384622)

[**2. Génération des unités** 13](#_Toc186384623)

[**Conclusion** 14](#_Toc186384624)

# **Introduction**

Dans le cadre du cours de Java III, le projet HELBArmy a été développé pour simuler une bataille militaire stratégique entre deux armées. Cette simulation met en œuvre les concepts avancés de programmation orientée objet et utilise JavaFX pour concevoir une simulation visuelle interactive et dynamique. L’objectif principal était de créer un environnement de simulation immersif, où des unités distinctes interagissent dans un espace de jeu structuré, tout en respectant des règles strictes de déplacement, de gestion des ressources, et de combat.

Le projet respecte l’ensemble des fonctionnalités de base spécifiées dans l’énoncé, incluant la gestion de l’espace de jeu, les interactions complexes entre les unités, les éléments collectables, et les commandes interactives. Toutefois, une discussion détaillée sur les éventuelles limitations et pistes d’amélioration sera présentée dans la section "Limitations".

Le rapport est structuré comme suit :

Fonctionnalités de base : Une présentation des principales fonctionnalités implémentées, décrites sous l’angle esthétique et fonctionnel.

Analyse : Une vue détaillée de l’architecture du code, appuyée par un diagramme pour expliquer les choix de conception.

Limitations : Une réflexion critique sur les limites techniques ou contextuelles du projet, avec des suggestions pour améliorer ou étendre les fonctionnalités.

Conclusion : Une synthèse des objectifs atteints, des apprentissages réalisés, et des défis relevés.

Une image contenant capture d’écran, Jeux, cercle, Jeux et sports d’intérieur

Description générée automatiquement

# **Fonctionnalités de base**

## **L’espace de jeu**

Dans **HELBArmy**, l’espace de jeu est une carte organisée en grille, composée de lignes et de colonnes. Cette structure garantit qu’aucun chevauchement d’éléments n’est possible, assurant ainsi une logique claire. Les bords de la carte, représentés par des murs infranchissables, empêchent les unités de sortir de l’espace défini, contraignant la planification des déplacements.

L’affichage de l’espace de jeu est géré dans la classe View, où la méthode **drawBackground** colore les cases pour représenter visuellement la grille. Pour vérifier le chevauchement d’éléments, la méthode **isOccupied** enregistre les positions exactes des éléments et identifie celles déjà occupées. Les déplacements sont validés grâce à **isMoveValid**, qui combine **isOccupied** et **isPositionValid** pour empêcher les unités de sortir de la carte ou de se superposer.

## **Les arbres :**

Les arbres, répartis aléatoirement sur la carte, possèdent chacun une quantité initiale de bois fixée à 100 unités. Lorsqu’un arbre est entièrement collecté, il disparaît pendant 30 secondes avant de réapparaître à sa position d’origine avec sa quantité de bois restaurée. Pendant sa présence, un arbre constitue un obstacle infranchissable ; toutefois, sa case devient libre durant sa période de disparition.

Les arbres sont implémentés dans la classe **Tree**, avec une variable **woodAmount** initialisée à 100 pour gérer la quantité de bois. Leur positionnement initial est assuré par la méthode **generateInitialTrees**, qui les place aléatoirement sur la carte. La méthode **checkAndRegenerateTrees** gère leur régénération en vérifiant régulièrement les arbres collectés et en les réintroduisant après le délai de 30 secondes.

## **Les villes :**

Les villes sont des structures fixes occupant un espace de 5x5 cases, ce qui en fait les éléments les plus imposants de la simulation. Elles génèrent des unités qui sortent par une porte unique orientée vers le centre de la carte. Elles servent également de point de dépôt pour les collecteurs, qui y rapportent le bois récolté. Deux villes adversaires sont positionnées stratégiquement : l’une au centre du bord supérieur et l’autre au centre du bord inférieur.

Pour représenter la position d’une ville sur la carte, des coordonnées **x** et **y** ont été utilisées. Les points clés de la ville, comme le point de dépôt et le point de génération des unités, sont définis par des variables de type **Point** dans la classe City. Une variable de type **String** identifie le type de ville ("White" ou "Black"), permettant de différencier les équipes et de gérer leurs interactions respectives.

## **Les unités :**

Les unités, générées par les villes, nécessitent une quantité spécifique de bois pour leur production, déterminée par leur type. Avant leur création, une vérification est effectuée pour s’assurer que les ressources requises sont disponibles et que le délai nécessaire est respecté. Cette logique est gérée par la méthode **generateUnits**, qui valide les conditions avant d’appeler **createUnit** pour finaliser la génération de l’unité.

Le jeu propose quatre types d’unités, chacune jouant un rôle stratégique distinct : les collecteurs, qui récoltent des ressources ; les déserteurs, qui traquent les collecteurs ennemis ; les cavaliers, qui protègent les collecteurs ; et les piquiers, qui sécurisent des positions fixes tout en bénéficiant d’avantages tactiques face aux cavaliers.

## 

## **Les collecteurs :**

Les collecteurs sont chargés de récolter le bois des arbres les plus proches en fonction de leur position sur la carte. Lorsqu’ils atteignent leur capacité maximale de 25 unités de bois, ils retournent automatiquement au dépôt de leur ville pour déposer les ressources, avant de repartir chercher de nouveaux arbres. Ce cycle se répète jusqu’à l’épuisement de tous les arbres sur la carte. La collecte est possible uniquement depuis l’une des huit cases adjacentes à un arbre, garantissant une logique cohérente.

Le comportement des collecteurs est implémenté à l’aide des méthodes suivantes :

* **collectAction** : Gère la collecte du bois auprès des arbres.
* **returnToCityAction** : Permet au collecteur de retourner au dépôt une fois son stock plein.
* **depositAction** : Dépose les ressources à la ville et relance **collectAction** pour continuer le cycle.

Chaque collecteur est initialisé avec ses attributs essentiels : quantité de bois par action, capacité maximale de 25 unités, et une ville assignée pour les dépôts.

**Les déserteurs :**Les déserteurs ont pour objectif principal de chasser les collecteurs de l’armée adverse, tout en fuyant les autres unités ennemies. Ils bénéficient d’un bonus en combat lorsqu’ils affrontent des piquiers ou des déserteurs ennemis.

Leur comportement est géré par la méthode **attackModeDeserter**, qui exploite les fonctionnalités de la classe Unit pour identifier si la cible la plus proche est une unité ennemie non collecteur. En fonction de cette analyse, le déserteur attaque ou fuit. La méthode **escapeEnemies** est utilisée pour gérer les déplacements en cas de fuite.

Le bonus de combat est intégré directement dans la méthode **attack**, qui applique les multiplicateurs requis en fonction du type d’adversaire.

## **Les cavaliers :**

Les cavaliers protègent les collecteurs en chassant les déserteurs de l’armée adverse. Ils se déplacent en groupes alliés, tout en respectant une distance de sécurité dynamique, ajustée en fonction des combats réalisés pendant la simulation. De plus, ils bénéficient d’un bonus en combat lorsqu’ils affrontent des déserteurs ennemis.

Ce comportement est implémenté à l’aide des méthodes suivantes :

* **pursueAndAttack** : Identifie et cible un déserteur ennemi, le poursuivant jusqu’à son élimination.
* **reposition** : Vérifie si la distance de sécurité entre cavaliers alliés est respectée.
* **moveAwayFromAlly** : Réajuste la position du cavalier pour maintenir un espacement correct entre alliés.
* **adjustSafetyDistanceAfterTurn** : Ajuste dynamiquement la distance de sécurité, initialement fixée à 1, en l’augmentant de 1 après chaque tour selon les combats effectués.

## 

## **Les piquiers :**

Les piquiers agissent comme des tourelles. Après leur apparition, ils se déplacent vers une position aléatoire non occupée, générée par la méthode **generateRandomPosition**, grâce à **moveToAssignedPosition**. Une fois positionnés, ils attaquent à vue avec un champ de vision initial limité à 1 case adjacente. Ce champ de vision s’ajuste dynamiquement en fonction du nombre de piquiers alliés présents sur la carte, une vision élargie partagée par tous les piquiers d’une même équipe.

Les combats sont gérés par **engageEnemyPikeman**, qui vérifie, via **isWithinVision**, si un ennemi se trouve dans le champ de vision. La méthode **calculateCollectiveVision** ajuste ce champ en comptant les piquiers alliés sur la carte. De plus, les piquiers bénéficient d’un bonus de combat lorsqu’ils affrontent des cavaliers ennemis.

## **Système de combat :**

Dans **HELBArmy**, le combat se déclenche lorsque deux unités sont adjacentes. Une unité attaque un ennemi choisi aléatoirement parmi ceux à sa portée. Le combat continue jusqu’à ce que l’une des deux unités meure, garantissant des interactions stratégiques et dynamiques.

Le système de combat est centralisé dans la classe Unit, où les variables clés **healthPoints** (points de vie) et **attackValue** (valeur d’attaque) sont définies. Une map nommée **bonusMap** gère les multiplicateurs de bonus attribués selon les types d’unités.

Les fonctionnalités sont implémentées via les méthodes suivantes :

* **attack** : Lance une attaque contre une unité ennemie.
* **getBonusMultiplier** : Calcule le multiplicateur de bonus en fonction du type d’adversaire.
* **takeDamage** : Applique les dégâts reçus.
* **die** : Gère la suppression d’une unité lorsque ses points de vie tombent à zéro.

**Système de déplacement :**  
Dans **HELBArmy**, chaque unité peut se déplacer horizontalement, verticalement ou en diagonale, tout en respectant les obstacles tels que les autres unités, les villes ou tout autre élément du jeu. Ces mouvements offrent une grande flexibilité et adaptabilité à la simulation.

Le déplacement fluide est assuré par les méthodes suivantes :

* **calculateDistance** : Calcule la distance entre une unité et sa cible.
* **moveTowards** : Permet à l’unité de se déplacer vers un point fixe ou une position dynamique.
* **moveToPosition** : Met à jour la position de l’unité après avoir calculé le chemin optimal, garantissant un mouvement fluide et coordonné.

## **Déplacement et gestion des obstacles :**

Dans **HELBArmy**, tout mouvement peut être bloqué par un obstacle, tel qu’une autre unité, une ville, ou tout élément du jeu. Lorsqu’un obstacle est rencontré, l’unité ajuste son trajet pour le contourner sans le traverser.

Ce système est implémenté via la méthode **tryAdjacentPositions**, appelée chaque fois qu’un mouvement dans **moveToPosition** est bloqué. Cette méthode explore toutes les directions adjacentes (horizontale, verticale et diagonale) pour identifier un chemin libre, permettant ainsi à l’unité de contourner l’obstacle et de continuer son déplacement.

## **Vitesses et gestion du temps :**

Dans **HELBArmy**, tous les éléments du jeu effectuent une action par seconde, et le jeu est rafraîchi à la même fréquence pour assurer une synchronisation fluide entre les actions et l’affichage. Cette gestion du temps est implémentée via une **timeline** initialisée dans la classe View. La timeline déclenche la méthode **handleGameTick** du Controller à chaque tick, qui met à jour les éléments de la scène en appelant **view.updateScene**. Ce mécanisme garantit que toutes les actions et les animations sont exécutées de manière coordonnée tout au long de la simulation.

## **Génération des unités :**

Les villes génèrent aléatoirement des unités parmi celles dont les ressources nécessaires sont disponibles. Ce processus est géré par la méthode **generateUnits**, qui vérifie d’abord les conditions de génération (ressources et temps requis). Les unités éligibles sont ajoutées à une liste, **availableUnits**, à partir de laquelle une unité est sélectionnée aléatoirement et générée.

## **La Pierre Philosophale :**

Dès le début du jeu, deux pierres philosophales sont générées à des positions aléatoires sur la carte grâce à la méthode **generatePhilosophicalStone**, située dans le Controller. Lorsqu’une unité entre en contact avec une pierre, la méthode **interactWithUnit**, définie dans la classe dédiée, s’active. Elle gère la disparition de la pierre et appelle **applyRandomEffect**, qui applique un effet aléatoire à l’unité : soit la mort instantanée, soit l’invincibilité avec des points de vie illimités, avec une probabilité de 50 %.

## **Le Drapeau :**

Un drapeau est généré toutes les deux minutes à une position aléatoire sur la carte grâce à la méthode **generateFlag**, gérée par le Controller. Une fois le drapeau apparu, toutes les unités modifient leur comportement pour tenter de le récupérer. Lorsqu’il est collecté, la méthode **collect** de la classe Flag s’active, entraînant la disparition du drapeau et appelant **applyHealthBonus**. Cette dernière attribue un bonus de 50 % de points de vie à l’unité qui l’a récupéré et à toutes les unités alliées déjà présentes sur la carte.

La vérification du délai requis avant chaque apparition du drapeau est assurée par la méthode **checkAndGenerateFlag**, garantissant qu’un seul drapeau soit présent à la fois.

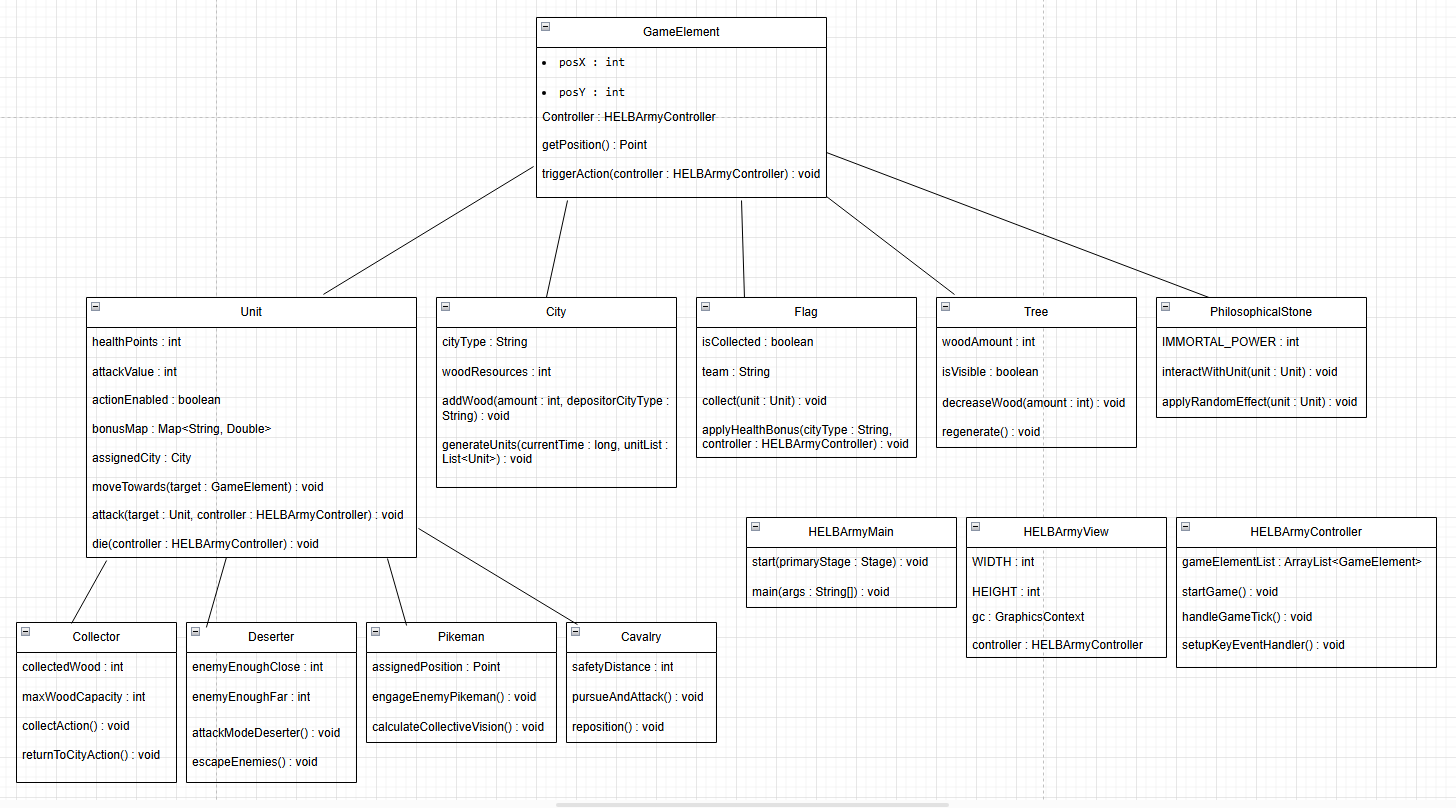
## **Système de cheat codes :**

Dans **HELBArmy**, un système de cheat codes permet de déclencher des événements spécifiques en appuyant sur des touches prédéfinies. Ces commandes influencent directement la simulation.

* **a, z, e, r** : Génèrent respectivement un collecteur, un déserteur, un cavalier ou un piquier dans la ville nord, via la méthode **generateUnitCheatCode**.
* **w, x, c, v** : Génèrent respectivement un collecteur, un déserteur, un cavalier ou un piquier dans la ville sud, également via **generateUnitCheatCode**.
* **j, k, l, m** : Stopper/activer les déplacements des collecteurs, déserteurs, cavaliers ou piquiers avec **toggleUnitActions**.
* **u** : Réduit les points de vie de toutes les unités à 0 avec **killAllUnitsCheatCode**.
* **i** : Fait apparaître un drapeau sur la carte grâce à **generateFlag**.
* **o** : Réinitialise la simulation via **restartGame**.
* **p** : Génère une pierre philosophale à une position aléatoire en appelant **generateRandomPosition** et ensuite créer une seule pierre avec cette position.

# **Analyse**

## **Diagramme UML - Architecture du projet HELBArmy**

****

Le diagramme UML illustre l'architecture du projet **HELBArmy**, mettant en évidence les relations entre les différentes classes et leurs responsabilités respectives. La conception suit une structure modulaire basée sur une hiérarchie claire pour gérer efficacement les éléments de la simulation, les unités, et les interactions dans l’espace de jeu.

La classe **GameElement** sert de base abstraite pour tous les éléments du jeu, comme les unités, les villes, les arbres, le drapeau, et la pierre philosophale. Elle garantit une gestion uniforme des positions et des interactions. Les classes héritant de **GameElement** (telles que **Tree**, **Flag**, et **PhilosophicalStone**) définissent des comportements spécifiques tout en partageant une structure commune.

Les unités (représentées par la classe abstraite Unit) héritent également de **GameElement** et sont spécialisées via des sous-classes (**Collector**, **Deserter**, **Cavalry**, **Pikeman**), chacune jouant un rôle stratégique distinct dans la simulation.

Les classes **HELBArmyController**, **HELBArmyView**, et **HELBArmyMain** structurent la gestion de l’application. Le contrôleur (Controller) coordonne la logique du jeu, la vue (**View**) gère l’affichage graphique, et le point d’entrée (Main) initialise et démarre la simulation.

## **Relations entre les classes dans HELBArmy**

**1. GameElement et ses sous-classes**

* **Raison de la relation :**  
  **GameElement** est la **classe mère** de tous les éléments présents sur la carte, qu’ils soient interactifs (unités, villes, drapeau) ou passifs (arbres, pierres). Cette structure garantit une gestion cohérente des positions et des actions spécifiques.
* **Relations :**
  + **City**, **Tree**, **Flag**, **PhilosophicalStone**, et **Unit** héritent de **GameElement**.
  + Chaque sous-classe ajoute des comportements spécifiques :
    - **City** : Génération d’unités, dépôt de bois.
    - **Tree** : Source de bois pour les collecteurs.
    - **Flag** : Bonus stratégique pour les unités.
    - **PhilosophicalStone** : Effets aléatoires sur les unités.
    - **Unit** : Interaction active avec les autres éléments du jeu.

**2. Unit et ses sous-classes**

* **Raison de la relation :**  
  Les unités partagent des mécanismes communs comme le déplacement, le combat, et l'interaction avec d'autres éléments. Chaque sous-classe (collecteurs, cavaliers, etc.) apporte des fonctionnalités adaptées à son rôle dans la simulation.
* **Relations :**
  + **Avec City :**
    - Les unités sont générées par les villes via **generateUnits().**
    - Les collecteurs déposent le bois dans leur ville assignée.
  + **Avec Tree :**
    - Les collecteurs interagissent avec les arbres pour collecter du bois via **decreaseWood().**
  + **Avec Flag et PhilosophicalStone :**
    - Toutes les unités modifient leur comportement pour capturer le drapeau lorsqu’il apparaît (**collect**()).
    - Les pierres philosophiques appliquent des effets aléatoires sur les unités via **interactWithUnit**().

**3. HELBArmyController avec GameElement et ses sous-classes**

* **Raison de la relation :**  
  Le contrôleur orchestre le fonctionnement du jeu. Il gère la logique des interactions et maintient une vue centralisée de tous les éléments présents sur la carte.
* **Relations :**
  + **Avec toutes les sous-classes de GameElement :**
    - Maintient une liste de tous les éléments (**gameElementList**), séparés par type (unités, arbres, villes, etc.).
    - Gère les interactions :
      * Déplace les unités.
      * Supprime les unités mortes et régénère les arbres collectés.
      * Génère les drapeaux après un certain délai.
  + **Avec HELBArmyView :**
    - Envoie à la vue les données à afficher via **updateScene().**
    - Configure les interactions utilisateur avec **setupKeyEventHandler()**.

**4. HELBArmyView avec HELBArmyController**

* **Raison de la relation :**  
  La vue dépend du contrôleur pour obtenir les données actuelles des éléments du jeu à afficher.
* **Relations :**
  + Le contrôleur fournit une liste des éléments (**gameElementList**) que la vue utilise pour dessiner les éléments graphiquement via **drawGameElements().**

**5. HELBArmyMain avec HELBArmyController et HELBArmyView**

* **Raison de la relation :**  
  HELBArmyMain agit comme le **point d’entrée du programme**, initialisant la vue et le contrôleur et les reliant pour démarrer la simulation.
* **Relations :**
  + Instancie **HELBArmyController** pour gérer la logique du jeu.
  + Instancie **HELBArmyView** pour gérer l’affichage graphique.
  + Relie le contrôleur et la vue pour permettre leur interaction.

**Conclusion**

Ces relations garantissent une **architecture cohérente et modulaire**, où chaque classe est responsable de tâches spécifiques :

* **GameElement** fournit une base commune pour tous les éléments de la carte.
* Les sous-classes spécialisent les comportements en fonction de leur rôle dans la simulation.
* **HELBArmyController** connecte et orchestre toutes les interactions entre les éléments.
* **HELBArmyView** s’occupe de l’affichage des données fournies par le contrôleur.
* **HELBArmyMain** met tout en place et démarre la simulation.

# **Limitations**

Dans la simulation du jeu **HELBArmy**, les limitations identifiées concernent principalement le **comportement du Cavalier** et la **génération des unités**.

## **1. Comportement du Cavalier**

Bien que j’aie compris son rôle stratégique dans le jeu, certaines incohérences sont apparues lors de la simulation. Plus précisément :

* La fonctionnalité de ciblage des ennemis fonctionne correctement.
* Cependant, le mécanisme de gestion de la distance de sécurité entre cavaliers alliés, qui doit augmenter progressivement au fil du jeu, montre des incohérences dans la simulation. Parfois, le cavalier semble surchargé d’informations et cesse de bouger, ou il respecte la distance avec ses alliés, mais celle-ci paraît trop large, rendant son comportement abstrait ou imprévisible.  
  Même si cette fonctionnalité est partiellement opérationnelle, elle pourrait bénéficier d’un ajustement pour améliorer sa fluidité.

## **2. Génération des unités**

La génération des unités fonctionne à **environ 95 %** de ses fonctionnalités attendues :

* La gestion des ressources et le mécanisme aléatoire en fonction des ressources disponibles sont bien implémentés.
* Toutefois, le temps de production des grandes unités, comme les **déserteurs** et les **cavaliers**, semble parfois non respecté. Par exemple, un déserteur peut être généré simultanément avec un piquier, alors qu’il devrait nécessiter un temps de production plus long. En revanche, les délais pour les collecteurs et les piquiers sont correctement gérés.

**Améliorations possibles avec plus de temps**

Si j’avais disposé de davantage de temps pour perfectionner le projet :

* J’aurais peaufiné le **comportement des cavaliers** pour éliminer les incohérences liées à la gestion de la distance de sécurité et offrir une simulation fluide et stratégique.
* J’aurais amélioré la **génération des unités** en intégrant un système de minuterie ou un indicateur visuel informant l’utilisateur qu’une unité est en cours de production, afin de rendre cette fonctionnalité plus claire et immersive.

# **Conclusion**

Le projet **HELBArmy** a été une expérience enrichissante et un défi stimulant. J’ai réussi à implémenter l’ensemble des fonctionnalités principales demandées, ce qui a permis de créer une simulation complète et fonctionnelle. Le système de gestion des unités, des ressources, et des éléments interactifs tels que les arbres, les drapeaux et les pierres philosophiques, a été réalisé avec succès en respectant les principes d’une architecture orientée objet.

Chaque unité a pu être codée avec ses comportements spécifiques, offrant ainsi une diversité stratégique dans la simulation. De même, la gestion des interactions entre les unités, les villes, et les éléments du jeu a été orchestrée avec soin pour garantir une logique cohérente.

Cependant, quelques limitations ont été identifiées, notamment au niveau du comportement des cavaliers et de la gestion des temps de production pour certaines unités, qui pourraient être améliorées. Ces détails n’ont cependant pas compromis la fonctionnalité globale du jeu, qui reste conforme aux attentes du projet.

Ce projet m’a permis de consolider mes compétences en programmation orientée objet, tout en approfondissant des notions essentielles telles que l’organisation du travail, la conception d’un système de classes structuré et la gestion des responsabilités entre celles-ci. J’ai également appris à veiller à la qualité du code, en évitant les redondances, et à développer une vision globale pour construire une architecture claire et maintenable.

Malgré les petites incohérences évoquées dans la simulation, je suis satisfait du résultat final. Ce travail représente un pas important dans mon parcours d’apprentissage, et malgré les défis rencontrés, je suis reconnaissant de cette opportunité. Je tiens également à remercier Monsieur Riggio pour son encadrement. Bien que ce projet ait été exigeant, je suis conscient que cette expérience contribuera de manière significative à mon avenir en tant que développeur.