Programmation Concurrente et Interactive

# Cahier des charges

## Projet “La Jardinerie”

Nguyen Duc Huy - LDD3 Magistrere

## 1. Introduction

- *“La thématique du jeu proposée est celle d’une jardinerie : le joueur contrôle des jardiniers qui plantent et récoltent des fleurs de différents types pour composer des bouquets. Les bouquets rapportent de l’argent qui permettent d’embaucher plus de jardiniers ou d’acheter de nouvelles graines pour faire plus de bouquets. Malheureusement, les vaches du voisinage raffolent des fleurs et vous devrez les empêcher de venir ruiner votre jardinerie.”*
- Le projet utilise le langage Java avec Java Swing comme outil pour développer une interface interactive pour ce programme.

# 2. Analyse globale

- Interface graphique interactive
  - Conception d'une interface utilisateur intuitive et esthétiquement agréable, permettant une interaction fluide avec le jeu. Les joueurs pourront observer en temps réel l'évolution de leur jardinerie, sélectionner et diriger les jardiniers, et accéder à des informations détaillées sur les différents éléments du jeu.
  - Difficulté moyenne, priorité 1

- Gestion de l'état du jeu
  - Implémentation d'un système robuste pour suivre avec précision tous les aspects du jeu, y compris les états dynamiques des unités, les interactions avec l'environnement, et les événements aléatoires affectant la jardinerie.
  - Difficulté moyenne, priorité 1

- Système d'animation utilisant Thread
  - Utilisation efficace des threads pour animer les différents éléments du jeu, assurant une expérience visuelle dynamique et réactive aux actions des joueurs.
  - Difficulté moyenne, priorité 2

- Système sonore (à implémenter) - difficulté moyenne, priorité 3
- Système d'effets spéciaux (à implémenter) - difficulté moyenne, priorité 3

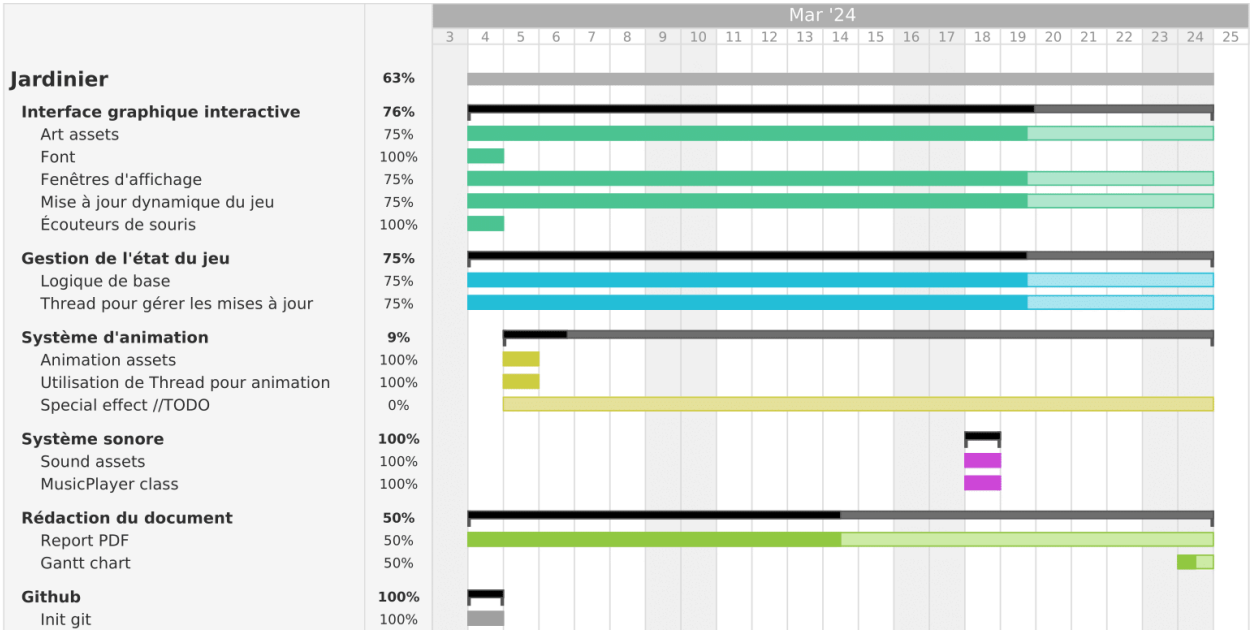
# 3. Plan de développement

- Interface graphique interactive : 4 heures
  - Mise en place des éléments graphiques pour les unités, interface utilisateur : 2 heures
  - Recherche d'une Font adaptée à l'esthétique du design : 15 minutes
  - Utilisation de Java Swing pour configurer les fenêtres d'affichage du jeu : 15 minutes
  - Utilisation de Java Swing pour la mise à jour dynamique du jeu : 1 heure
  - Écouteurs de souris pour enregistrer l'interaction de l'utilisateur : 30 minutes

- Gestion de l'état du jeu : 2.5 heures
  - Mise en place de la logique de base du jeu : 2 heures
  - Utilisation de Thread pour gérer les mises à jour du jeu et les changements : 30 minutes

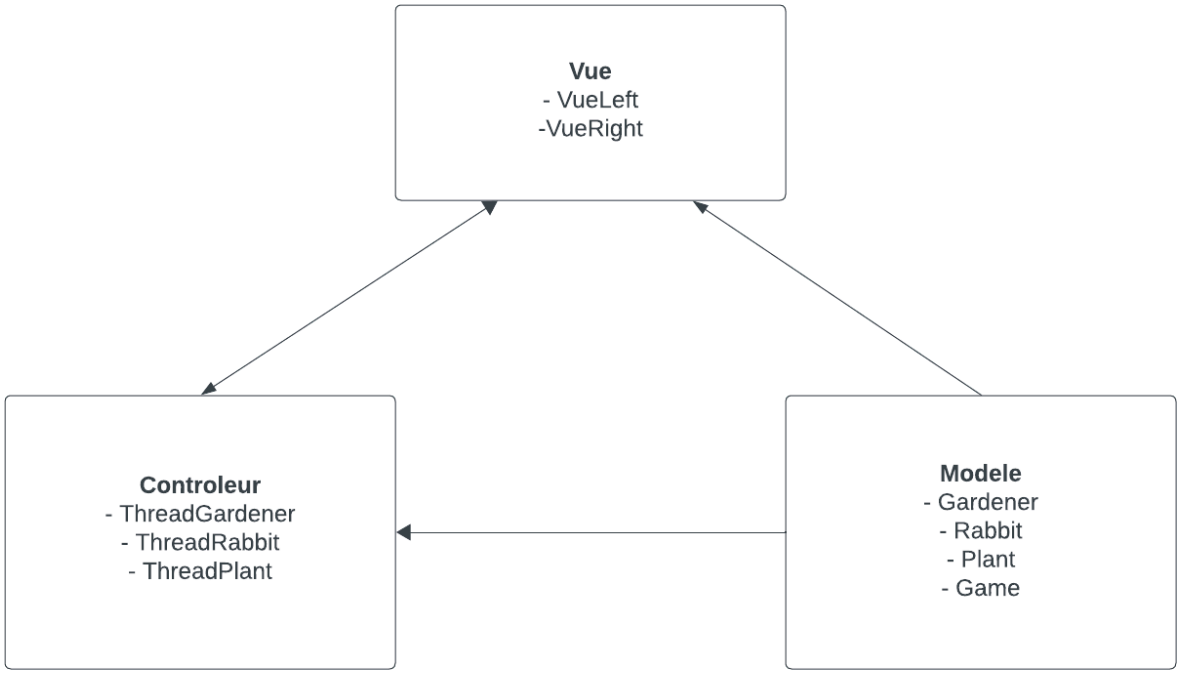
- Système d'animation utilisant Thread : 1 heure
  - Mise en place des éléments graphiques pour l'animation : 1 heure
  - Utilisation de Thread pour ajouter de l'animation au jeu : 10 minutes

- Système sonore : 1 heure
  - Mise en place des éléments : 1 heure
  - Coder la classe pour la Système sonore : 15 minutes
- Système d'effets spéciaux (à implémenter)
- Rédaction du document analyse et conception : 4 heures
- Mise en place une page Github pour l'hébergement de projet : 15 minutes




# 4. Conception generale

Le patron MVC

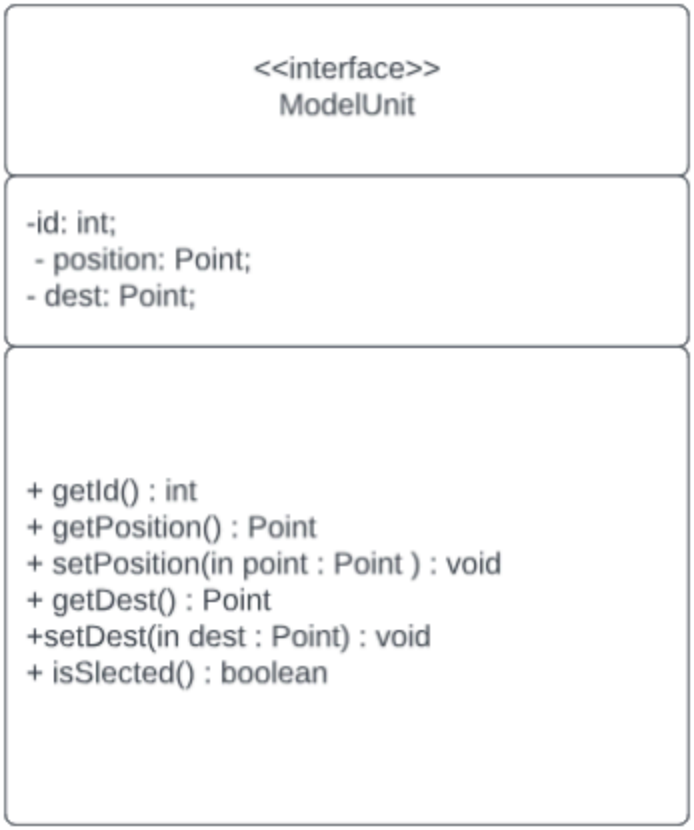


# 5. Conception détaillée

## A) Modèle

### 1. Interface Unit

- utilisée comme base pour d'autres types d'objets Unité
- chaque Unité a un identifiant int, une position Point, une destination Point et un boolean isSelected avec leurs propres accesseurs et mutateurs
- Diagramme:



## 2. Gardener

Entités jouant un rôle central dans le jeu, avec des fonctions spécifiques pour planter, récolter, et défendre les plantes contre les invasions de vaches. Chaque jardinier possède les attributs suivants :

- Identifiant (ID) : Unique pour chaque jardinier, facilitant leur gestion.
- Position : Coordonnées actuelles sur la carte du jeu.
- Destination : Coordonnées de la destination lorsqu'ils se déplacent.
- Vitesse : Détermine la rapidité de déplacement d'un point à un autre.
- État : Inclut différents états comme se déplacer, planter, récolter, ou être en repos.

Les fonctionnalités principales :

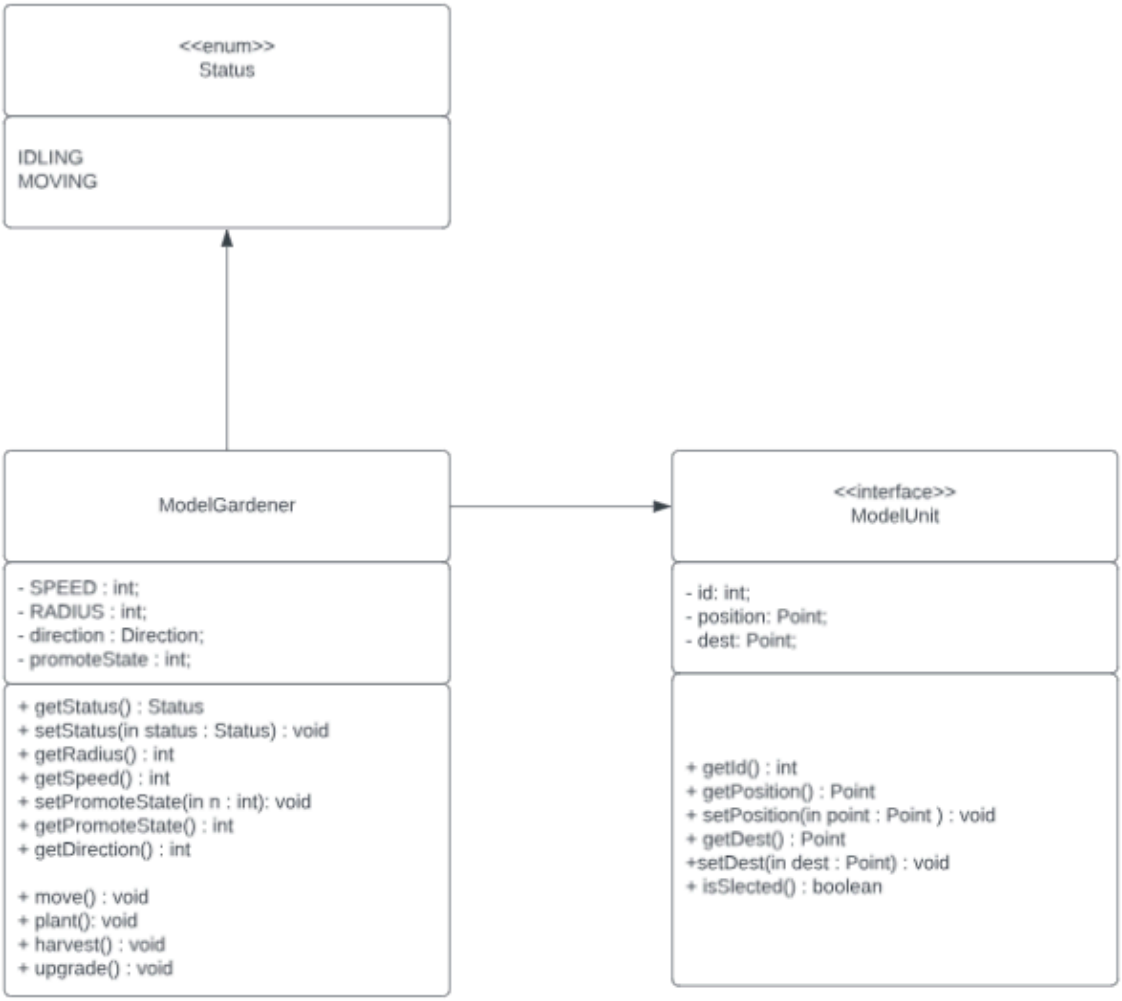
- **move()** - Action de déplacement: Algorithmes pour naviguer vers des destinations spécifiques.
  - Structures de données principales utilisées:
    - ArrayList<Point> *currentPath* contenant le chemin à suivre généré par A* algorithme.
    - Point *position* et Point *dest* déterminant respectivement la position actuelle et la destination. Si *position* === *dest*, le jardinier reste immobile.
    - HashMap<Integer, ModelObstacle> pour stocker les obstacles.
  - Constantes du modèle: *SPEED* pour la vitesse de déplacement du jardinier.
  - **Algorithme**
    - Calculer la distance entre la position actuelle et la destination.
    - Si la distance est inférieure ou égale à SPEED ou si currentPath est vide, déplacer le jardinier directement à la destination et changer son statut à IDLING.
    - Sinon, obtenir le prochain point de currentPath et calculer la distance par rapport à la position actuelle.
    - Si cette distance est inférieure ou égale à SPEED, déplacer le jardinier vers ce point et le retirer de currentPath.
    - Sinon, calculer les pas à faire en x et y en fonction de SPEED, ajuster la position du jardinier, et mettre à jour son statut à MOVING.
  - Conditions limites
    - La destination ne doit pas être trop proche d'un obstacle (vérification dans setDest()).
    - La liste currentPath ne doit pas être vide pour que le déplacement continue.
    - La distance calculée entre la position actuelle et le prochain point ou la destination doit être supérieure à SPEED pour effectuer un déplacement.

- **setDest(Point dest)**
  - Structures de données principales utilisées:
    - ArrayList<Point> *currentPath* contenant le chemin à suivre généré par A* algorithme.
    - Point *position* et Point *dest* déterminant respectivement la position actuelle et la destination. Si *position* === *dest*, le jardinier reste immobile.
    - HashMap<Integer, ModelObstacle> pour stocker les obstacles.
  - Constantes du modèle: *GridSystem.OBSTACLE_SIZE* détermine la taille des obstacles à considérer lors de la vérification de la validité de la nouvelle destination.
  - **Algorithme**
    - Parcourir tous les obstacles (obs) pour vérifier si la destination (dest) est trop proche d'un quelconque obstacle, en utilisant GridSystem.OBSTACLE_SIZE comme critère de proximité. Si oui, sortir de la fonction sans changer de destination.
    - Si la destination est valide (pas trop proche d'un obstacle), mettre à jour dest avec la nouvelle destination.
    - Jouer un son de mouvement pour indiquer le changement de destination.
    - Utiliser l'algorithme de chemin AStarPathfinder pour trouver un chemin asynchrone de la position actuelle à la nouvelle destination, puis mettre à jour currentPath avec le chemin trouvé.
  - Conditions limites
    - La destination ne doit pas être à une distance inférieure ou égale à GridSystem.OBSTACLE_SIZE de n'importe quel obstacle.
    - Le chemin trouvé doit être validé et mis à jour dans currentPath avant que le jardinier puisse se déplacer vers dest.

- **plant()**: Sélection et plantation de graines dans des zones désignées.
  - S'il y a assez d'argent, planter un type de plante aléatoire à sa position actuelle.
  - Sinon, rien est fait

- **harvest()**: Collecter des plantes mûres pour générer des ressources.
  - Utiliser la position actuelle pour vérifier toutes les plantes autour, si la plante est récoltable, retirer cette plante du jeu et ajouter l'argent.
  - Sinon, rien est fait

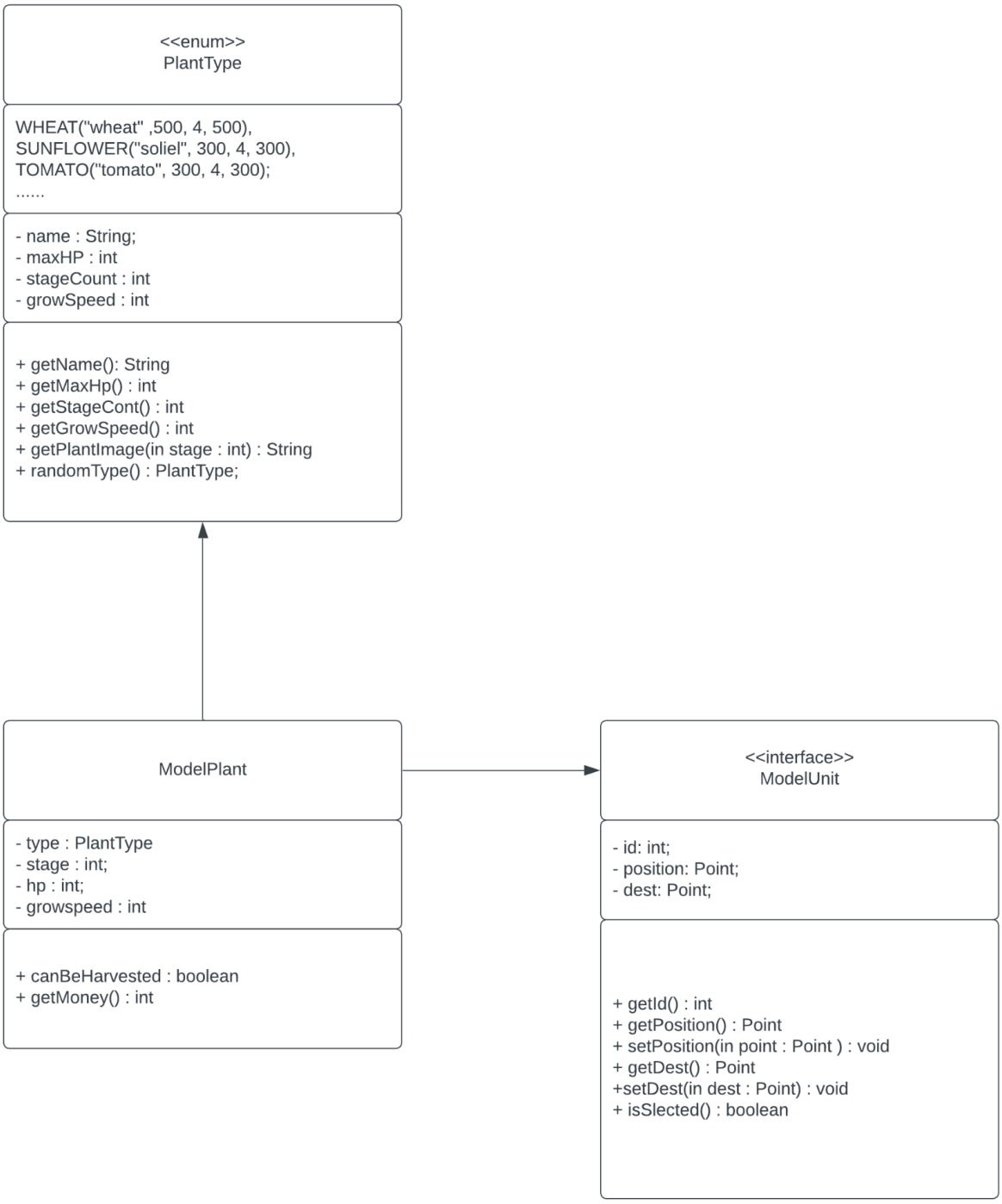
- **promote()**: Amélioration des compétences et des capacités via l'utilisation de ressources.
  - S'il y a assez d'argent, augmenter la vitesse et le rayon de récolte du jardinier.
  - Il existe une limitation de 5 niveaux maximum de promotion pour éviter d'avoir un jardinier qui peut protéger tous les plants en même temps.


<<enum>>
Status
IDLING
MOVING
ModelGardener
- SPEED : int;
- RADIUS : int;
- direction : Direction;
- promoteState : int;
+ getStatus() : Status
+ setStatus(in status : Status) : void
+ getRadius() : int
+ getSpeed() : int
+ setPromoteState(in n : int): void
+ getPromoteState() : int
+ getDirection() : int
+ move() : void
+ plant(): void
+ harvest() : void
+ upgrade() : void
<<interface>>
ModelUnit
- id: int;
- position: Point;
- dest: Point;
+ getId() : int
+ getPosition() : Point
+ setPosition(in point : Point ) : void
+ getDest() : Point
+setDest(in dest : Point) : void
+ isSlected() : boolean

## 3. Plant

Utilisez une énumération PlantType pour représenter les différents types de plantes, chacun avec son propre nom, son art, sa santé maximale (sa résistance face à être mangé par les ennemis), sa vitesse de croissance et le nombre d'étapes qu'elles doivent franchir avant d'être récoltables. Le taux de croissance du plant et le temps nécessaire pour qu'il mûrisse peuvent ajouter un élément de gestion des ressources au jeu.
Les plantes représentent les ressources primaires du jeu, avec différents types offrant divers bénéfices.
Chaque plant possède les attributs suivants :

- Type de Plante (PlantType) : Différencier les plantes par leurs attributs et leur valeur.
- Position : Emplacement dans le jardin.
- Santé : Indicateur de la durabilité de la plante, influencée par les attaques de vaches.
- Croissance : Étapes de développement avant que la plante ne soit prête à être récoltée.


<<enum>>
PlantType
WHEAT("wheat" ,500, 4, 500),
SUNFLOWER("soliel", 300, 4, 300),
TOMATO("tomato", 300, 4, 300);
.....
- name : String;
- maxHP : int
- stageCount : int
- growSpeed : int
+ getName(): String
+ getMaxHp() : int
+ getStageCont() : int
+ getGrowSpeed() : int
+ getPlantImage(in stage : int) : String
+ randomType() : PlantType;
ModelPlant
- type : PlantType
- stage : int;
- hp : int;
- growspeed : int
+ canBeHarvested : boolean
+ getMoney() : int
<<interface>>
ModelUnit
- id: int;
- position: Point;
- dest: Point;
+ getId() : int
+ getPosition() : Point
+ setPosition(in point : Point ) : void
+ getDest() : Point
+setDest(in dest : Point) : void
+ isSlected() : boolean

## 4. Rabbit

Utilise la texture d'une vache pour représenter une vache affamée venant de chez votre voisin pour se régaler de vos plantes.

L'unité vache n'est pas contrôlable par l'utilisateur, elles sont toutes autonomes. Les vaches sont des éléments perturbateurs qui visent à consommer les plantes du joueur.

Chaque vache a 5 principaux états :

- En Attente : la vache reste immobile
- En Mouvement
- En Fuite : lorsqu'une vache se trouve dans le rayon d'un jardinier, elle s'enfuira vers le coin le plus proche du champ
- En Quitte : après un certain temps sans manger, la vache perd toute sa patience et quitte le champ
- En Train de Manger : mange quand elle est près d'une plante

Les fonctionnalités principales :

- **setDest(Point dest)**
  - Structures de données principales utilisées:
    - ArrayList<Point> *currentPath* contenant le chemin à suivre généré par A* algorithme.
    - Point *position* et Point *dest* déterminant respectivement la position actuelle et la destination. Si *position* === *dest*, le jardinier reste immobile.
    - HashMap<Integer, ModelObstacle> pour stocker les obstacles.
  - Constantes du modèle: *GridSystem.OBSTACLE_SIZE* détermine la taille des obstacles à considérer lors de la vérification de la validité de la nouvelle destination.
  - **Algorithme**
    - Parcourir tous les obstacles (obs) pour vérifier si la destination (dest) est trop proche d'un quelconque obstacle, en utilisant GridSystem.OBSTACLE_SIZE comme critère de proximité. Si oui, sortir de la fonction sans changer de destination.
    - Si la destination est valide (pas trop proche d'un obstacle), mettre à jour dest avec la nouvelle destination.
    - Utiliser l'algorithme de chemin AStarPathfinder pour trouver un chemin asynchrone de la position actuelle à la nouvelle destination, puis mettre à jour currentPath avec le chemin trouvé.
  - Conditions limites
    - La destination ne doit pas être à une distance inférieure ou égale à GridSystem.OBSTACLE_SIZE de n'importe quel obstacle.

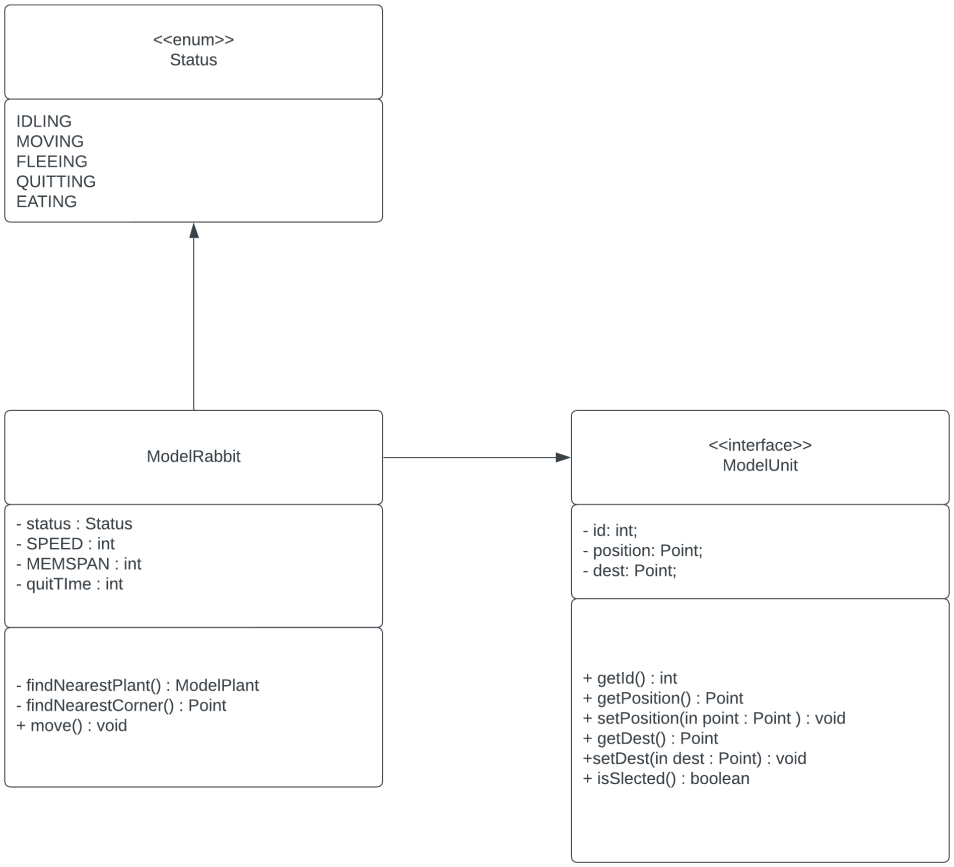
- **findNearestPlant()**
  - Structures de données principales utilisées:
    - HashMap<Integer, ModelPlant> pour toutes les plantes.
  - Constantes du modèle: *GridSystem.OBSTACLE_SIZE* détermine la taille des obstacles à considérer lors de la vérification de la validité de la nouvelle destination.

- **Algorithme**
  - Itère sur toutes les plantes, en ignorant celles qui sont dans le champ de vision d'un jardinier.
  - Retourner la plante la plus proche de la position actuelle du lapin.
- Conditions limites
  - Une plante doit être hors du champ de vision pour être considérée.
- Utilisation
  - Utilisée pour identifier la plante la plus proche à interagir avec la fonction de déplacement

- **findNearestCorner()**
  - Structures de données principales utilisées:
    - Utiliser un tableau Point[] pour les coins.
  - Constantes du modèle: *GridSystem.OBSTACLE_SIZE* détermine la taille des obstacles à considérer lors de la vérification de la validité de la nouvelle destination.
  - **Algorithme**
    - Calcule la distance de la position actuelle du jardinier à chaque coin de la zone de jeu.
    - Sélectionne le coin le plus proche comme destination.
  - Utilisation
    - Utilisée pour identifier à interagir avec la fonction de déplacement quand la vacher veut sortir.

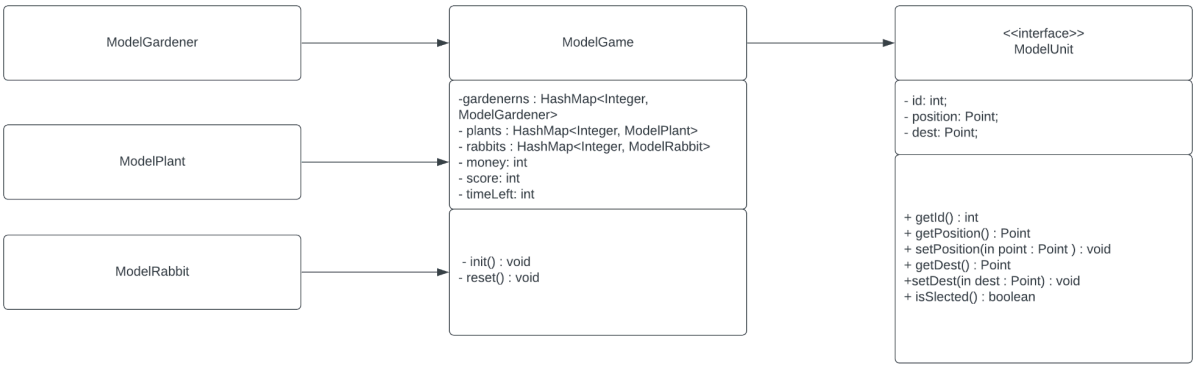
- **move()**
  - Structures de données principales utilisées:
    - ArrayList<Point> *currentPath* contenant le chemin à suivre généré par A* algorithme.
    - Point *position* et Point *dest* déterminant respectivement la position actuelle et la destination.

        - Status, un énumérateur pour status, décrivant l'état actuel de l'entité (par exemple, IDLING, MOVING, FLEEING).
    - Constantes du modèle:
        - SPEED pour la vitesse de déplacement.
        - MEMSPAN pour la durée après laquelle le vache réévalue son état ou son objectif.
        - dieTime pour déterminer si le vache doit cesser d'exister ou changer d'état vers QUITING
    - **Algorithme**
        - Si l'état est ‘Quitting’, déplacer vers la plus proche coin.
        - Vérifier si dieTime est dépassé et que l'état n'est pas QUITING. Si oui, changer l'état selon la visibilité de l'objectif ou le temps écoulé depuis le dernier changement d'état.
        - Si en fuite (FLEEING) et ayant atteint la destination ou après un certain temps (MEMSPAN), changer l'état en IDLING.
        - Si IDLING et après MEMSPAN, chercher la plante la plus proche ou choisir une destination aléatoire, et déplacer vers celle-ci.
        - Si MOVING et la destination est atteinte, changer l'état en EATING.
        - Calculer le déplacement vers la destination ou le prochain point du chemin. Si la destination est atteinte ou qu'il n'y a plus de chemin, mettre à jour la position directement. Sinon, calculer et appliquer les pas de déplacement en x et y.
    - Conditions limites
        - L'entité doit changer d'état de MOVING à EATING si elle est à une distance inférieure ou égale à 25 de sa destination.
        - L'entité ne doit pas essayer de se déplacer si elle est en état IDLING ou EATING.
        - La décision de changer de destination ou d'état dépend de la visibilité de l'objectif, du temps écoulé (MEMSPAN), et si la destination actuelle est atteinte.


<<enum>>
Status
IDLING
MOVING
FLEEING
QUITTING
EATING
ModelRabbit
- status : Status
- SPEED : int
- MEMSPAN : int
- quitTIme : int
- findNearestPlant() : ModelPlant
- findNearestCorner() : Point
+ move() : void
<<interface>>
ModelUnit
- id: int;
- position: Point;
- dest: Point;
+ getId() : int
+ getPosition() : Point
+ setPosition(in point : Point ) : void
+ getDest() : Point
+setDest(in dest : Point) : void
+ isSlected() : boolean

## 5. ModelGame

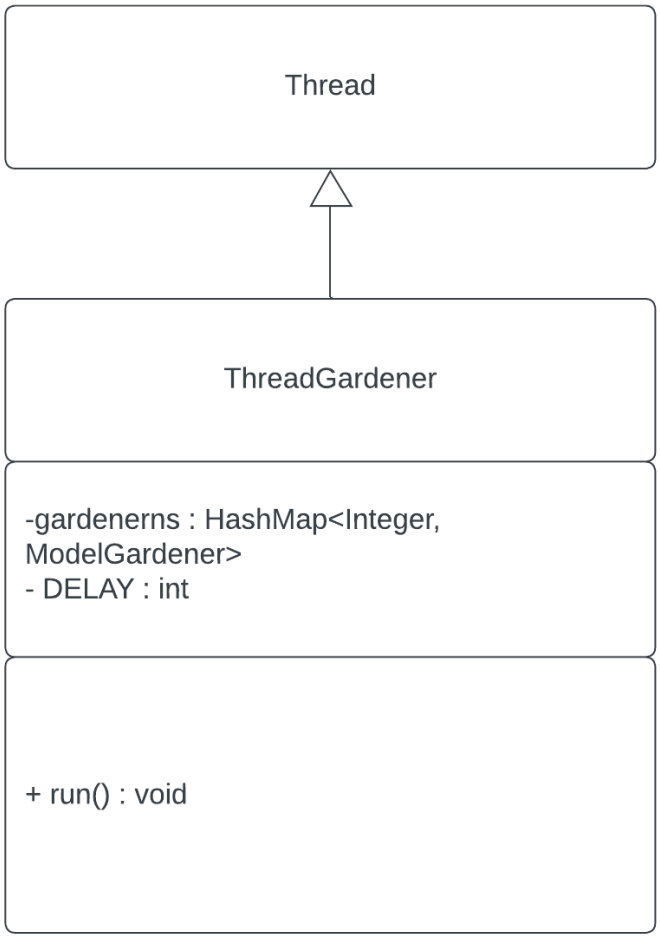
- Utilisez 3 HashMap avec id, unité pour conserver les informations sur les jardiniers, les plantes, les lapins. Contient également des informations sur l'argent, le score et le temps restant dont dispose le joueur.
- Système de score : est mis à jour à chaque fois qu'un jardinier récolte une plante. Le score est calculé en fonction de la vitesse de croissance de cette plante (plus elle met de temps à pousser == plus elle a de chances d'être mangée par une vache, donc donne plus de points).
- Système monétaire : à chaque fois qu'une plante réalise une récolte, ajoutez la même somme d'argent que le score de cette plante. L'argent est utilisé pour acheter des jardiniers, planter de nouvelles graines ou améliorer les jardiniers.
- Fin du jeu : Le jeu se termine automatiquement après une session de 5 minutes ou les joueurs n'ont plus les moyens d'augmenter leur argent (pas assez d'argent pour acheter de nouvelles graines), le seul facteur décisif étant le score obtenu lors de la récolte des plantes.



## B) Contrôleur

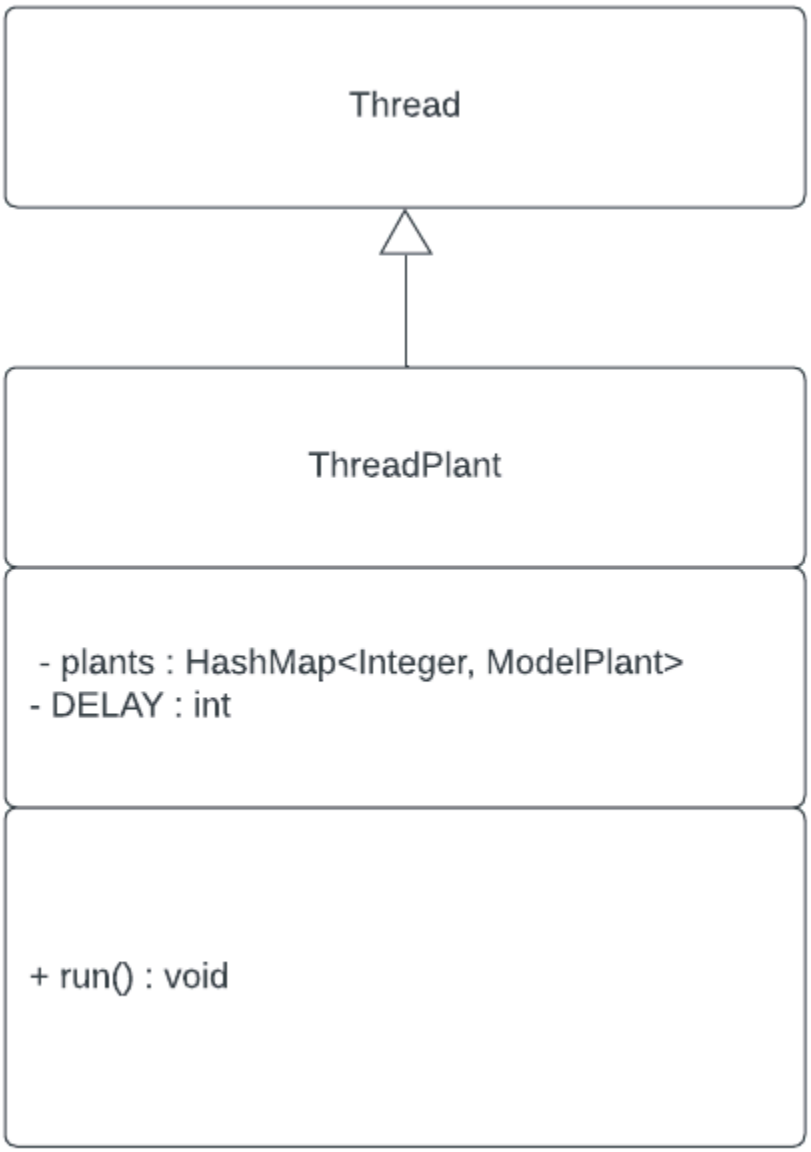
### 1. ThreadGardener

- Extension du Thread
- Après chaque boucle d'un DELAY constant, déplacez tous les gardeners



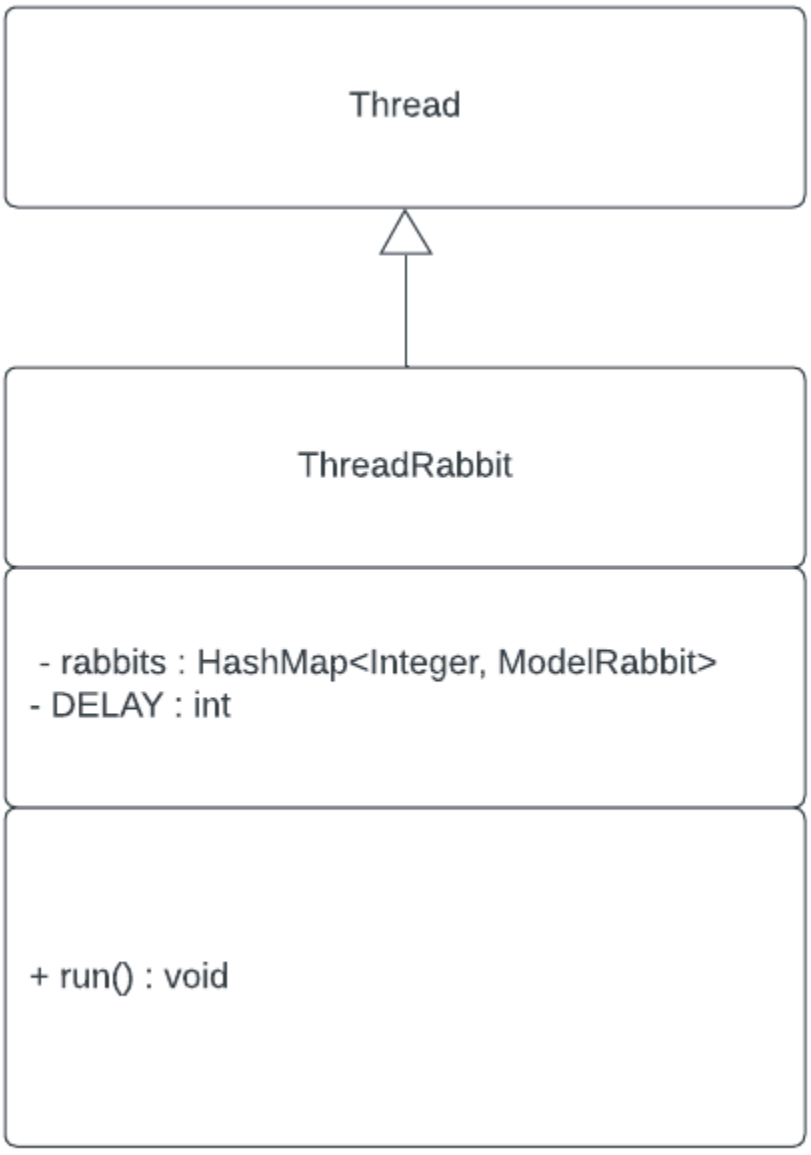
## 2. ThreadPlant

- Extension du Thread
- Après un délai constant, faites pousser toutes les plantes. Si une plante est trop vieille, ou si sa santé est réduite à 0, retirez-la du jeu.



## 3. ThreadRabbit

- Extension du Thread
- Après chaque boucle d'un DELAY constant, déplacez tous les vaches



## C) Vue


Gardener
10
0
293
SHOP
JARDINIER
100


Gardener
10
0
295
JARDINIER
IDLING
PLANTER
10
RECOLTE
UPGRADE
25

La vue principale du jeu se compose de 2 sections principales : la section de gauche et la section de droite.

- La section de gauche (surlignée en bleu clair) montre l'état actuel du jeu, avec les jardiniers, les vaches, les plantes, votre score et votre argent disponible.
- La section de droite (surlignée en vert clair) affiche les informations d'un objet sélectionné (jardinier, vache ou plante.) Si rien n'est sélectionné, elle affiche la boutique où le joueur peut acheter des objets.

Les fonctionnalités principales :

- La section de gauche écoute les entrées de l'utilisateur pour sélectionner et désélectionner des objets. Si rien n'est sélectionné, cette section obtient la position du clic gauche de la souris, si cette position est en corrélation avec la hitbox d'un objet, sélectionnez cet objet. Si quelque chose est déjà sélectionné, un clic gauche le désélectionne si vous cliquez sur un espace vide ou sélectionnez un nouvel objet sur lequel vous cliquez.
- La section de droite utilise l'objet sélectionné dans la section de gauche pour afficher des informations ou afficher la boutique si rien n'est sélectionné.

# 6. Résultat


Gardener
COMMENCE
SORTIE


Gardener
0
0
289
SHOP
JARDINIER
100


# 7. Documentation utilisateur

- Prérequis : Java avec un IDE
- (ou Java tout seul si vous avez fait un export en .jar exécutable)
- Mode d'emploi (cas IDE) : Importez le projet dans votre IDE, sélectionnez la classe Main a la racine du projet puis cliquez sur l'icône ≪ Run ≫.
- Mode d'emploi (cas .jar exécutable) : double-cliquez sur l'icône du fichier .jar
- Cliquez sur la fenêtre pour faire monter l'ovale

## 8. Documentation développeur

- La classes principale : Main.java
- Les packages :
  - assets : les images
  - control : les classes des contrôleur
  - main : contient Main.java
  - model : les classes des modèle
  - view : les classes des vue

## 9. Conclusion et perspectives

- J'ai mis en œuvre avec succès toutes les exigences de base du projet
- Je prévois d'ajouter des effets sonores et des effets spéciaux.
- Je dois également améliorer l'optimisation de mon code pour améliorer la performance.