

PROGRAMMATION AVANCÉE

ENSemenC :

Rapport de projet

Réalisé par : Léa-Ava MAIRE et Noémie TCHOUA

Promo 2027 - 1A - Groupe 2



mai 2025

Sommaire :

| | |
|--|-----------|
| I. Présentation du jeu..... | 2 |
| 1. Univers du jeu et spécificités..... | 2 |
| 2. Déroulement d'une partie type..... | 2 |
| II. Structure du code et choix de modélisation..... | 3 |
| 1. Diagramme UML..... | 3 |
| 2. Les classes et sous-classes..... | 4 |
| - Plantes..... | 4 |
| - Terrains..... | 5 |
| - Animaux..... | 8 |
| - Maladie..... | 9 |
| - Potager..... | 10 |
| Gestion générale du potager (actions mode normal) :..... | 10 |
| Gestion des obstacles (actions mode urgence) :..... | 11 |
| Animaux..... | 11 |
| Maladies :..... | 12 |
| Mauvaises herbes :..... | 12 |
| Inondations :..... | 12 |
| - Météo..... | 13 |
| - Simulation..... | 14 |
| 3. Tests réalisés..... | 16 |
| II. Organisation de l'équipe..... | 16 |
| 1. Planning de réalisation et répartition des tâches..... | 16 |
| 2. Matrice d'implication..... | 17 |
| III. Bilan..... | 17 |
| 1. Difficultés rencontrées..... | 17 |
| IV. Conclusion..... | 18 |
| Annexes..... | 18 |

I. Présentation du jeu

1. Univers du jeu et spécificités

“ENSEmenC” immerge le joueur dans la gestion d’un vaste potager découpé en parcelles de types Terre, Sable ou Argile, chacune dotée d’un indice d’absorption d’eau propre qui influe sur la quantité d’eau réellement fournie aux plantes.

Les semis disponibles correspondent à différents types de fruits, possédant chacun des particularités en termes de nature (vivace ou annuelle), saisons de semis, terrain préféré, espacement minimal, vitesse de croissance (en unité par tour), besoin en eau et en lumière, températures préférées, espérance de vie, nombre de fruits maximum pouvant être produit et saison d’apparition de ces fruits.

Au fur et à mesure des simulations, des mauvaises herbes peuvent rapidement coloniser les parcelles et étouffer les cultures, une faune variée peut surgir et impacter la santé des semis, et plusieurs pathologies menacent à tout instant la vitalité du potager

Enfin, le jeu alterne entre un mode normal hebdomadaire (où la météo, la croissance et les interactions se déroulent en cycles de sept jours) et un mode urgence déclenché par un événement critique (tempête, intrusion ou épidémie), qui exige une réaction rapide du joueur pour éviter la perte de ses plantations.

2. Déroulement d’une partie type

Le déroulement d’une partie se structure en une boucle hebdomadaire : au début de chaque tour, la date et la météo sont affichées (température, ensoleillement ou précipitations), puis toutes les plantes évoluent : elles vieillissent, absorbent l’eau disponible, poussent selon leurs caractéristiques et peuvent être affectées par les maladies ou par la propagation des mauvaises herbes. Après cette phase de simulation, le menu d’action s’ouvre et le joueur choisit ses opérations. Une fois ses actions épuisées ou son tour validé, la semaine se termine, la date avance de sept jours et un nouveau cycle débute.

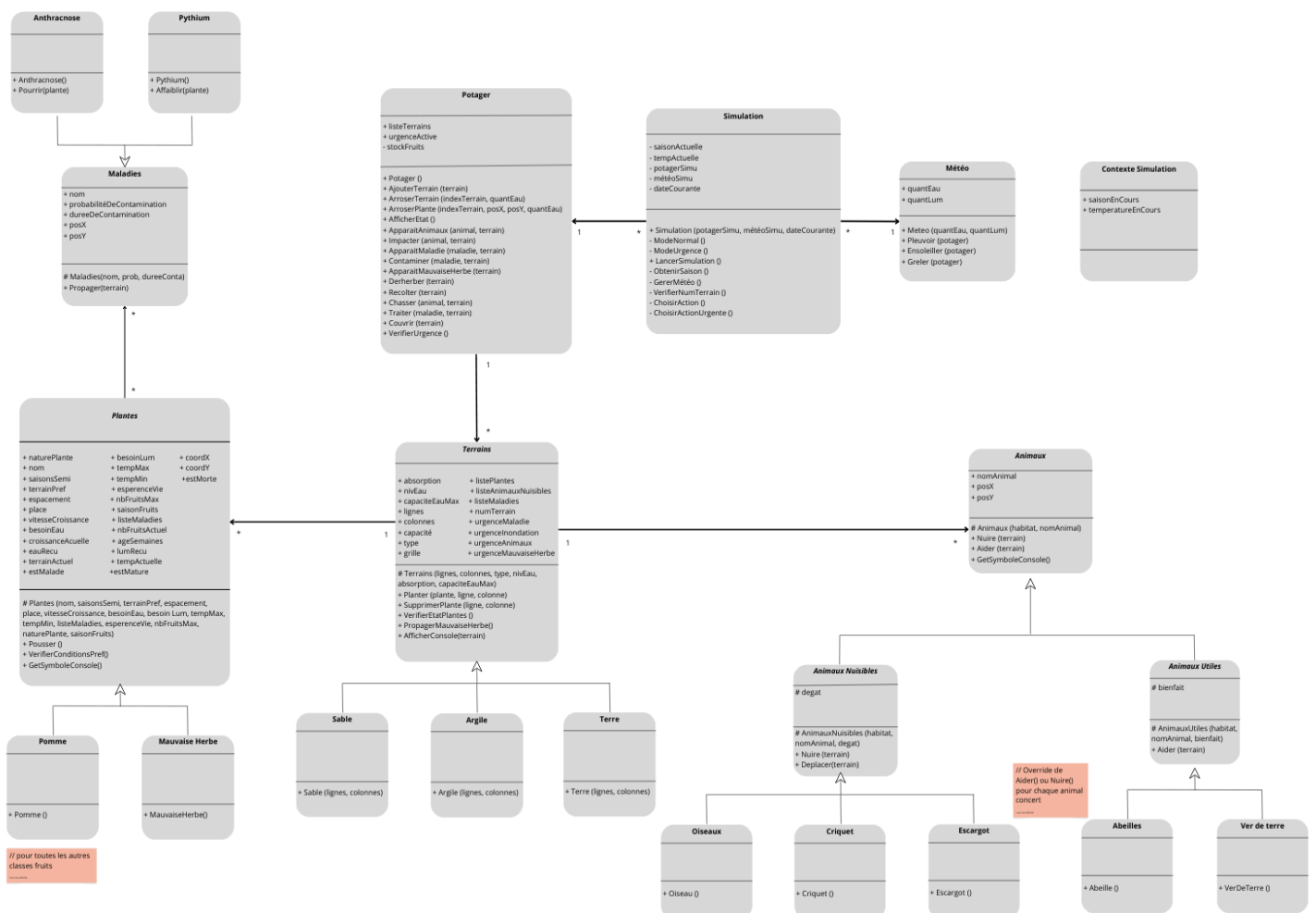
Chaque semaine, le joueur dispose d’un nombre restreint d’actions à répartir entre ses différents terrains. Il peut tout d’abord planter un semis parmi ceux disponibles : chaque espèce impose des contraintes de saison, de type de sol et d’espacement pour assurer un développement optimal. Il peut ensuite arroser l’un de ses terrains en choisissant la quantité d’eau à verser, sachant que seules les parcelles adaptées retiendront suffisamment

l'humidité pour alimenter ses cultures. Lorsque des parasites, des maladies ou des mauvaises herbes menacent la santé de ses plantations, il peut effectuer une action de traitement (désherbage, traitement) pour préserver ses récoltes. Enfin, dès qu'un semi atteint sa maturité et qu'il se trouve dans sa saison de fructification, le joueur peut récolter ses fruits et les stocker pour générer des ressources.

Dans certains cas critiques, la propagation de mauvaises herbes, l'attaque d'un intrus ou l'épidémie d'une maladie sévère, le jeu bascule en mode urgence. La cadence ralentit pour permettre au joueur de visualiser en temps réel la menace et d'intervenir rapidement. Une fois l'urgence maîtrisée, la partie reprend son rythme hebdomadaire normal.

II. Structure du code et choix de modélisation

1. Diagramme UML



2. Les classes et sous-classes

- Plantes

“**Plantes**” est une classe abstraite portant tous les attributs et comportements communs aux plantes. Elle regroupe les informations génériques pour chaque type de plante telles que le nom, la nature (annuelle ou vivace), les saisons de semis et de fructification, les préférences de sol, les besoins en eau et en lumière, les températures préférées, la vitesse de croissance, l’espérance de vie, la liste de maladies. Comme elles sont invariantes et dépendantes du type de plante, ces informations sont remplies directement dans les constructeurs de chaque type de plantes (sous-classes de “**Plantes**”).

La classe “**Plantes**” regroupe aussi les états dynamiques de la plante au cours des simulations, telles que sa croissance actuelle, son âge (en semaines), le nombre de fruits produit, la quantité d’eau et de lumière reçue pendant le tour, la température actuelle, le terrain sur laquelle elle est plantée ainsi que ses coordonnées x et y. De plus, trois booléens indiquent l’état de la plante (si la plante est malade, arrivée à maturité ou morte).

Concernant les méthodes, la classe “**Plantes**” en contient trois : Pousser(), VérifierConditionsPref() et GetSymboleConsole().

La méthode **Pousser()** incarne la logique de croissance : à chaque tour (semaine), la plante pousse jusqu’à atteindre le stade de maturité (**croissanceActuelle** \geq 1) . Mais sa vitesse de croissance va dépendre des conditions dans lesquelles la plante se trouve. On fait alors appelle à la méthode **VérifierConditionsPref()** qui passe en revue les quatres conditions pouvant influencer la croissance de la plante : le terrain sur lequel elle se trouve (si c’est son préféré ou non), la quantité d’eau et de lumière reçu (calcul du ratio par rapport à ses besoins) et la température actuelle (si elle se situe dans l’intervalle de températures préférées de la plante). Ainsi, si au moins deux de ces conditions sont remplies (50%), alors la plante va pousser normalement, sinon, elle est ralentie de moitié. Une fois la plante arrivée à maturité (**estMature** = **true**), la croissance est complète donc on ne passe plus dans cette boucle. En parallèle, la plante vieillit et à chaque tour on ajoute 1 semaine à son âge (**ageSemaines** ++). Si la plante atteint son âge maximal (**ageSemaines** \geq **esperenceVieSemaines**), alors elle meurt si c’est une plante **annuelle**, ou elle reprend sa croissance de zéro si c’est une plante **vivace**. La méthode **DonnerFruits()** permet de faire pousser des fruits sur chaque plant lorsque les conditions sont favorables.

Enfin, la méthode **GetSymboleConsole()** renvoie le symbole de la plante selon son stade de croissance (un point si la plante est juste semée, son initiale si elle a une croissance plus avancée puis un émoji correspondant à son fruit lorsqu’elle atteint la maturité). Ce symbole est récupéré dans la méthode **AfficherConsole()** de la classe “Terrain” permettant d’afficher le terrain et les plantes contenues à l’intérieur.

Fruits (pomme, fraise, poire, mangue, kiwi, pastèque) :

Chaque espèce concrète (comme “**Pomme**” ou “**Fraise**” ...) hérite de cette base en appelant le constructeur de “**Plantes**” et en y mettant ses propres arguments.

Mauvaises Herbes :

La sous-classe “**MauvaiseHerbe**” illustre un cas particulier : elle n’a pas besoin d’espacement avec les autres plantes, elle pousse sur tous types de terrains, et elle est directement mature (`estMature = true` et `croissanceActuelle = 1`). Une fois apparue, elle se propage rapidement sur le terrain (voir méthode `PropagerMauvaiseHerbe()` modélisée dans la classe “Terrains”).

- Terrains

Les terrains, modélisés par la classe abstraite “**Terrains**”, créent un cadre de culture permettant de gérer les plantes et la quantité d’eau présents sur un terrain. Ainsi, chaque terrain est caractérisé par : son numéro, ses dimensions (nombre de lignes et de colonnes), les plantes présentes (stockées dans une matrice et une liste), sa capacité d’eau maximale, son coefficient de rétention (absorption), et son niveau d’eau actuel. Ils possèdent également une variable booléenne pour chaque menace pouvant survenir (animaux nuisibles, maladies, inondations ou mauvaises herbes) ; permettant de savoir si le terrain est en danger ou non. Nous avons choisi d’utiliser une matrice nommée “grille”, de type `Plantes` (ne pouvant contenir que des objets issus de la classe `Plantes`), afin de faciliter la gestion des positions des plantes sur le terrain.

La méthode `Planter(Plantes plante, int i, int j)` permet de planter une “`plante`” sur une case précise (“`i`” : ligne, “`j`” : colonne) du terrain avec lequel on fait appel à la méthode. Elle est de type booléenne et renvoi `true` si la plante a pu être correctement plantée ou `false` si ce n’est pas le cas. Pour être plantée, il faudra que les coordonnées de la plante vérifient trois conditions : être dans les bornes du terrain, être sur une case vide (vérifier qu’il n’y a pas déjà une plante dessus) et enfin respecter l’espacement minimal requis par la plante pour pousser correctement. Afin de vérifier cette dernière condition, différentes variables locales sont utilisées : “`d`” correspondant à la distance minimale (en nombre de cases) à respecter avec les autres plantes (stockée dans `plante.Espacement`, dépend du type de plante) ; et les coordonnées maximales (`iMax`, `jMax`) et minimales (`iMin`, `jMin`) calculées en ajoutant ou en soustrayant “`d`” aux coordonnées “`i`” et “`j`”. Les fonctions `Math.Max()` et `Math.Min()` sont utilisées pour assurer le respect des bornes du terrain. Ces nouvelles coordonnées permettent de délimiter la zone d’espacement requise par la plante autour de la case cible (case sur laquelle on souhaite planter). Deux boucles “`for`” permettent de parcourir lignes par lignes, toute cette zone, et de vérifier que toutes les cases soient bien libres. Une fois toutes les conditions vérifiées, la plante peut être plantée sur le terrain : elle

est ajoutée dans la grille du terrain aux coordonnées i, j (`grille[i, j] = plante`) et dans la liste des plantes du terrain (`ListePlantes.Add(plante)`). Les coordonnées i et j deviennent les coordonnées de la plante, que les attributs “coordX” et “coordY” de la classe “**Plantes**” récupèrent (`plante.coordX = i` et `plante.coordY = j`). Ceci permet un accès aux coordonnées de la plante directement depuis la classe **Plantes**, sans passer par la classe **Terrains**.

Une autre méthode `SupprimerPlante(int i, int j)` permet de retirer la plante située aux coordonnées “ i ”, “ j ” entrées en arguments par l'utilisateur. Cette méthode est utilisée lorsqu'une plante meurt, naturellement ou suite à un incident. Elle renvoie *true* si la suppression est réussie, ou *false* sinon. On vérifie d'abord que la case cible ne soit pas vide (donc pas “null”) et qu'elle contienne bien une plante. Ensuite on passe l'attribut booléen “estMorte” de la classe **Plantes** à *true*, puis on retire la plante de la grille et de la liste du terrain (`grille[i, j] = null` et `ListePlantes.Remove(plante)`).

La méthode `VerifierEtatPlantes()` ne prend rien en paramètre et ne renvoie également rien. Elle est appelée dans la classe “**Simulation**” à chaque tour de jeu, après la méthode `plante.Pousser()` pour vérifier que la plante ne soit pas morte, notamment après avoir dépassé son espérance de vie. On parcourt donc toute la liste des plantes du terrain grâce à un “foreach” et on regarde si l'attribut “estMorte” de la plante est à *true*, si oui, alors on la retire du terrain en appelant la méthode `SupprimerPlante()` décrite juste avant.

La méthode `PropagerMauvaiseHerbe()` simule la manière dont les mauvaises herbes se propagent sur un terrain. En effet, une fois semées, elles se multiplient rapidement aux tours suivant en colonisant les cases adjacentes, tuant les plantes sur lesquelles elles tombent. Pour implémenter ceci, des variables locales ont été utilisées : “`aPropager`” qui contient les coordonnées de toutes les mauvaises herbes situées sur ce terrain et “`propagationReussie`” qui passe à *true* une fois la propagation terminée. La première boucle *for* collecte les positions des mauvaises herbes pour les stocker dans la liste “`aPropager`”. La seconde boucle “foreach” parcourt les coordonnées de chaque mauvaise herbe de la liste “`aPropager`” (une coordonnées (i, j) = une mauvaise herbe) pour essayer d'envahir une des huit cases adjacentes. Tant que la propagation n'est pas réussie (`propagationReussie == false`), on tente d'aller sur une case adjacente tirée aléatoirement (`newI = i + rnd.Next(-1,2)` et `newJ = j + rnd.Next(-1,2)`), tout en évitant tous dépassement des bornes du terrain (`newI > 0 && newI <= lignes && newJ > 0 && newJ <= colonnes`). Si cela est vérifié, alors on regarde si la nouvelle case est occupée ou non. Si elle est **vide** (`terrain.grille[newI, newJ] == null`) alors on y sème une nouvelle mauvaise herbe (`terrain.Planter(new MauvaiseHerbe(), newI, newJ)`). Si elle est **occupée**, alors c'est soit par une autre mauvaise herbe (dans ce cas on ne fait rien), soit par une autre plante, et dans ce cas, on active le mode d'urgence (`urgenceMauvaiseHerbe = true`).

Enfin, la méthode `AfficherConsole()` permet, comme son nom l'indique, d'afficher la matrice représentant le terrain, et les plantes présentent dessus. Chaque type de terrain à une couleur prédéfinie qui colore la grille avec `Console.BackgroundColor`. On commence par afficher les indices de colonnes du terrain. Pour chaque ligne, on écrit l'indice de la ligne, et le contenu de chaque case en affichant un espacement si la case est vide (`grille[i, j] == null`) ou le symbole de la plante sinon (en faisant appel à la méthode `GetSymboleConsole()` de la classe `Plantes`). Les colonnes sont délimitées par des traits verticaux `"|"` et les lignes sont délimitées par des traits horizontaux `"+---+"`.

Les sous-classes `"Terre"`, `"Sable"` et `"Argile"` fixent leurs coefficients hydriques (respectivement 0.6, 0.3 et 0.8) et leur capacité pour éviter les `"if(type==...)"` répétitifs et assurer un code ouvert à l'extension.

Les trois types de sols se distinguent avant tout par leur capacité à retenir l'eau, ce qui influe directement sur la quantité d'humidité réellement mise à disposition des plantes :

Terre :

C'est un sol équilibré, assez meuble pour absorber les pluies sans engorgement trop rapide. Nous lui avons attribué une capacité d'eau maximale de 100 unités et un coefficient de rétention de 0.6 (soit 60 % de l'eau versée est stockée et distribuée aux plantes, le reste s'écoule).

Sable :

C'est un sol très drainant, qui laisse rapidement percoler l'eau vers les couches inférieures. Sa capacité est plus faible (80 unités) et son coefficient de rétention ne dépasse pas 0.3 (30 %). Concrètement, ce sol demande des arrosages plus fréquents pour éviter la sécheresse.

Argile :

C'est un sol lourd et compact, excellent pour conserver l'humidité mais susceptible de se gorgier et de provoquer l'asphyxie racinaire. Il bénéficie d'une capacité maximale de 120 unités et d'un coefficient de rétention élevé de 0.8 (80 %). Ainsi, il faut se méfier de l'excès d'eau qui favorise les maladies.

Lorsque le terrain reçoit de l'eau, par la pluie (méthode `Pleuvoir()` de la classe `Météo`) ou par arrosage (méthode `Arroser()` de la classe `"Potager"`), le taux d'absorption noté `Absorption` (coefficient entre 0 et 1) multiplie directement la quantité d'eau apportée pour calculer la réserve disponible (`NivEau`). C'est cette réserve qui est ensuite distribuée aux plantes selon leurs besoins hydriques. Plus un terrain retient d'eau, plus il peut subvenir aux consommations de ses semis sans intervention immédiate du joueur, tandis qu'un sol à faible rétention exigera un suivi plus régulier pour maintenir une humidité suffisante.

- Animaux

Une **faune variée** est introduite via la hiérarchie : “**Animaux**” (classe abstraite) → “**AnimauxNuisibles**” et “**AnimauxBenefiques**” (classes abstraites) → espèces spécifiques (“**Oiseau**”, “**Criquet**”, “**Abeille**”, ...). Chaque animal possède un nom, des coordonnées (initialisées une fois apparu sur le terrain) et implémente une méthode qui inflige des dégâts ou offre des bonus de croissance à la plante sur laquelle il est positionné.

Animaux Nuisibles :

La sous-classe abstraite “**AnimauxNuisibles**” contient un attribut “**Degat**” qui détermine l’ampleur des dégâts causés par chaque animal et une méthode abstraite **Nuire(Terrains terrain)** qui va infliger ces dégâts aux plantes situées sur un terrain. Cette méthode sera réécrite (“override”) dans chaque classe concrètes issues d’**AnimauxNuisibles**, afin d’illustrer la manière particulière selon laquelle chaque animal nuisible s’attaque aux plantes. Une autre méthode commune à tous les animaux nuisibles est **Déplacer()**, utilisée notamment lorsque le mode urgence est activé, afin de montrer en temps réel le déplacement aléatoire de l’animal. Cette méthode ne sera pas utilisée pour les animaux utiles, dont le déplacement ne nous intéresse pas, c’est pourquoi nous ne l’avons pas créé dans la classe Animaux.

- Criquet :

Les criquets sont des insectes vivant dans un environnement sableux et sec. Ils s’attaquent aux plantes et les affaiblissent grandement. En effet, si la plante n’est pas mature (**if(!p.estMature)**) et qu’elle n’est plus une graine (**p.croissanceActuelle > 0.5**) sa vitesse de croissance est divisée par les dégâts (**p.VitesseCroissance /= Degat**).

- Escargot :

L’escargot est un animal nuisible qui se déplace sur les terrains en terre et grignote les plantes qui sont encore en croissance. On vérifie si une plante n’est pas mature (**if(!p.estMature)**), dans ce cas la taille de la plante est divisée par les dégâts (**p.croissanceActuelle /= Degat**).

- Oiseaux :

Les oiseaux sont des animaux adeptes des fruits des plantes qui poussent sur des terrains argileux, et n’hésitent donc pas à les manger. Si une plante possède assez de fruits (**p.nbFruitsActuel >= Degat**), l’oiseau mange et la plante perd 2 fruits (**p.nbFruitsActuel -= Degat**).

Animaux Utiles :

Ces animaux sont utiles pour le bien être des plantes. La sous-classe abstraite “**AnimauxUtiles**” issue de la classe Animaux contient donc une instance “**Bienfait**” qui correspond aux bienfaits que l’animal va prodiguer à la plante, et une méthode abstraite **Aider(Terrains terrain)** illustrant comment l’animal applique ces bienfaits aux plantes situées sur un terrain entré en paramètre. Cette méthode sera réécrite (“override”) dans chaque classe concrètes issues d’**AnimauxUtiles**, afin d’illustrer comment chaque animal utile aide les plantes.

- Abeille :

L’abeille à un pouvoir pollinisateur qui va permettre aux plantes soit de produire plus de fruit si elle plante est mature (**plantes.nbFruitsActuel += Bienfait**) (sans dépasser le nombre limite de fruits qu’elle peut produire), soit de grandir (**p.croissanceActuelle += Bienfait/10**).

- Verre de Terre :

Le ver de terre est un animal souterrain qui agit comme un engrais en remuant la terre, ce qui va augmenter la vitesse de croissance des plantes (**p.VitesseCroissance += Bienfait/10**) qui ne sont pas encore matures (**!p.estMature**).

- **Maladie**

Des maladies comme l’anthracnose et le pythium sont aussi un danger pour le potagr et peuvent apparaître dans des conditions totalement différentes. Dans la classe “**Maladies**”, chacune d’elles possède un nom (**nom**), des coordonnées (**posX**, **posY**), une probabilité (**prob**) et une durée de contamination (**dureeConta**). Elles peuvent également impacter les autres plantes en se propageant sur le terrain, la méthode **Propager(Terrains terrain)** permet alors de générer de nouvelles coordonnées pour modéliser le mouvement de la maladie.

- L’anthracnose:

Cette maladie attaque les fruits présents sur les plantes et les fait pourrir progressivement, via la méthode **Pourrir(Plantes plante)**. Elle se développe lorsque le temps est très sec et chaud (**if(ContexteSimulation.TempEnCours > 30 && terrain.NivEau < 15)**). Elle a une probabilité de se propager de 30% et perdure pendant une période de 10 jours si elle n’est pas traitée.

- Le pythium:

Cette maladie s’attaque seulement au graine en ralentissant leur croissance avec la méthode **Affaiblir(Plantes plante)**. Elle apparaît lorsqu’il pleut, plus précisément lorsque le terrain est très humide et que la température est très basse (**if(ContexteSimulation.TempEnCours < 5 && terrain.NivEau > 80)**). Il y a 10% de chance qu’elle se propage et continue d’affaiblir la plante pendant 15 jours si elle n’est pas traitée.

- Potager

La classe “**Potager**” joue un rôle clé pour le joueur : elle centralise tous les terrains qu’il possède dans une liste “**ListeTerrains**”, conserve ses réserves de récoltes et expose les actions qu’il peut effectuer sur son jardin à travers des méthodes (appelées lors de la simulation). Un indicateur booléen “**urgenceActive**” passe à true dès qu’un incident (maladie, animaux nuisibles, mauvaises herbes,...) survient quelque part dans le potager, et informe la boucle de simulation qu’il est nécessaire de basculer en mode urgence.

Gestion générale du potager (actions mode normal) :

La méthode **AjouterTerrain(Terrains terrain)** ajoute une nouvelle parcelle (de type Sable, Terre ou Argile) à **ListeTerrains** avec la commande **ListeTerrains.Add(terrain)**. Elle attribue au terrain son **numTerrain** avec un index initialisé à 0 et qui augmente à chaque création d’un nouveau terrain. Les caractéristiques du terrain (type et dimensions) sont demandées à l’utilisateur dans la classe Simulation, juste avant l’appel de cette méthode. Ainsi, elle permet au joueur d’agrandir progressivement son potager.

L’arrosage peut être réalisé de deux manières différentes : l’arrosage global d’une parcelle du potager, ou l’arrosage ciblé d’une seule plante.

La première façon est illustrée par la méthode **ArroserTerrain(int indexTerrain, double quantEau)**. L’objectif est de simuler un arrosage « général » qui remplit la réserve d’eau du sol avant de s’infiltrer dans toutes les racines. On commence par retrouver le terrain ciblé par l’utilisateur grâce à son index dans la liste, et de le stocker dans une variable “terrain” (**Terrains terrain = ListeTerrains[indexTerrain]**). Ensuite, on tire parti du coefficient Absorption du terrain pour calculer combien d’eau il stocke (**terrain.NivEau += quantEau - quantEau * terrain.Absorption**), puis on affecte ce même niveau à chaque plante à travers un “foreach” (**plante.eauRecu = terrain.NivEau**), comme si elles puisaient uniformément dans la même réserve.

La seconde manière est illustrée par la méthode **ArroserPlante(int indexTerrain, int posX, int posY, double quantEau)** permettant d’arroser une plante en particulier (dont les coordonnées sont entrées en paramètres), sur un terrain précis du potager (dont l’index est entré en paramètre) et pour une quantité précise d’eau. L’objectif est de traiter un cas exceptionnel, par exemple une plante desséchée qui a besoin d’un surcroît d’arrosage ciblé, utile pour corriger une carence sans inonder l’ensemble du terrain. On commence par retrouver le terrain indiqué par l’utilisateur grâce à son index dans la liste, et de le stocker dans une variable “terrain” (**Terrains terrain = ListeTerrains[indexTerrain]**). Ensuite, on parcourt la liste des plantes contenue dans ce terrain à l’aide d’un “foreach”. Une fois la plante cible trouvée (**if (plante.coordX == posX && plante.coordY == posY)**), on contourne la réserve du sol et on lui ajoute directement la quantité d’eau choisie par l’utilisateur

(`plante.eauRecu += quantEau`). Si la plante a bien été trouvée et qu'elle a pu être arrosée, la méthode renvoie *true*, sinon elle renvoie *false*. Ceci permet de gérer le cas d'erreur où la plante cible n'existerait pas (mauvaise indication de l'utilisateur).

La méthode `Recolter(Terrains terrain)` permet de ramasser tous les fruits mûrs d'un même type sur une parcelle et de les ajouter à une réserve centralisée (`StockFruits`). Elle demande d'abord à l'utilisateur quel fruit récolter, puis parcourt la liste des plantes du terrain pour cumuler et remettre à zéro le nombre de fruits produits par chacune. Si aucun fruit n'est disponible, elle informe le joueur et quitte. Sinon, elle crée ou met à jour l'entrée correspondante dans le dictionnaire `StockFruits` (clé = nom du fruit, valeur = quantité stockée) avant d'afficher un message de synthèse. Cette approche unifie la gestion de toutes les variétés de fruits sans code spécifique à chaque espèce.

La méthode `AfficherEtat()` affiche, pour chaque terrain : son numéro, son type et son rendu graphique (`terrain.AfficherConsole()`), puis, pour chaque plante qu'il contient, son nom, sa position, son état de santé (maladie ou non), son niveau de croissance, son besoin en eau restant (`plante.BesoinEau - plante.eauRecu`), et le nombre de fruits actuellement produits. Enfin, la méthode récapitule le total des réserves de fruits. L'utilisateur dispose ainsi d'un aperçu complet et détaillé du potager avant de passer son tour.

Gestion des obstacles (actions mode urgence) :

La méthode `VerifierUrgence()` intervient donc pour vérifier qu'il n'y a plus de menace sur aucun des terrains afin de désactiver le mode urgence, ou de l'activer dans le cas contraire. Pour cela, elle vérifie que pour chaque terrain les différents types d'urgence sont négatifs en utilisant une boucle `foreach` qui s'arrête dès qu'un terrain ne remplit pas les conditions (`break`).

Animaux

Plusieurs méthodes permettent la gestion des animaux présents dans le potager. La méthode `ApparaitAnimaux(Animaux animal, Terrains terrain)` place aléatoirement un animal (nuisible ou utile) sur une case libre du terrain, met à jour ses coordonnées et notifie le joueur. Ensuite, la méthode `Impacter(Animaux animal, Terrains terrain)` permet de gérer l'impact des animaux sur le terrain : si l'animal est nuisible et se trouve sur une plante, il appelle `animal.Nuire(terrain)` pour infliger des dégâts, et prévient qu'il y a une urgence animal sur le terrain (`urgenceAnimaux = true`). Si l'animal est bénéfique, il appelle `animal.Aider(terrain)` pour stimuler la croissance.

Enfin, la méthode `Chasser(AnimauxNuisible animal, Terrains terrain)` permet de faire fuir l'animal nuisible qui se trouve actuellement sur le terrain en le supprimant de la liste `terrain.AnimauxNuisibles`. Il n'y a donc plus de menace animale sur le terrain lorsque la liste est vide (`terrain.ListeAnimauxNuisibles.Count == 0`).

Maladies :

De la même manière, les méthodes `ApparaitMaladies(Maladies maladie, Terrains terrain)` & `Contaminer(Maladies maladie, Terrains terrain)` permettent la gestion des maladies dans le potager. Ainsi, une maladie apparaît à un emplacement aléatoire, puis infecte la plante correspondante en position (`posX`, `posY`), modifiant son indicateur `estMalade` et appliquant l'effet spécifique (pourriture, affaiblissement). La maladie active également le mode `urgenceMaladie` du terrain.

La méthode `Traiter(Maladies maladie, Terrains terrain)` permet d'éradiquer progressivement la maladie qui endommage les plantes du terrain et de les soigner. En effet, la plante reprend sa vitesse de croissance ou reproduit ses fruits en fonction de la maladie dont elle est atteinte. De plus, la durée de contamination de la maladie diminue de 5 jours à chaque fois que la plante est traitée. De ce fait, dès qu'elle est égale à 0, la maladie est supprimée de la `ListeMaladie`, la plante n'est plus malade (`p.estMalade = false`) et le mode `urgenceMaladie` est désactivé lorsque la `ListeMaladie` est vide (`terrain.ListeMaladie.Count == 0`).

Mauvaises herbes :

La méthode `ApparaitMauvaiseHerbe(Terrains terrain)` permettant de semer, à une position aléatoire, une mauvaise herbe sur un terrain du potager. Leur propagation est ensuite gérée dans la classe `Terrains` par la méthode `PropagerMauvaiseHerbe()`.

Afin de pallier cette propagation, le joueur peut désherber son terrain via la méthode `Désherber(Terrains terrain)`, permettant de retirer toutes les mauvaises herbes présentes sur le terrain entré en paramètre. Pour cela, nous avons créé une copie "gelée" de la liste, stockée dans une variable locale (`var plantesAVerifier = terrain.ListePlantes.ToList()`). En effet, il est impossible de modifier une liste tout en la parcourant avec un `foreach` par exemple, cela mène à une erreur. C'est pourquoi passer par l'intermédiaire d'une "liste copie" est nécessaire ici. Ainsi, on parcourt la liste des plantes contenues dans le terrain (via la liste copie "plantesAVerifier"), et si on tombe sur une mauvaise herbe, alors on la supprime du terrain (`terrain.SupprimerPlante(p.coordX, p.coordY)`). Une fois la liste parcourue, et le terrain nettoyé, le mode d'urgence peut être désactivé (`terrain.urgenceMauvaiseHerbe = false`).

Inondations :

La méthode `Couvrir(Terrains terrain)` permet de protéger le terrain de la pluie lorsqu'il est inondé, afin de laisser l'eau s'évaporer petit à petit. Le niveau d'eau du terrain diminue progressivement (`terrain.NivEau -= 5`) et dès qu'il repasse sous sa capacité d'eau maximale (`if(terrain.NivEau < terrain.CapaciteEauMax)`), le mode `urgenceInondation` se désactive.

- Météo

La classe “**Meteo**” centralise toute la logique météo et ses impacts sur le potager, en restituant à chaque tour les principaux aléas climatiques (pluie, soleil, grêle) et leurs conséquences sur l’eau des sols, la lumière reçue par les plantes, la température du contexte, ainsi que sur la génération d’événements biologiques (animaux, maladies et mauvaises herbes). Dès son instanciation, Meteo stocke deux valeurs essentielles : **QuantEau** représente la quantité totale d’eau tombée lors d’un épisode de pluie (1 à 100) et **QuantLum** mesure l’ensoleillement (1 à 100). Ces attributs sont mis à jour à chaque appel de **Pleuvor()** ou **Ensoleiller()**, puis redistribués aux terrains et plantes via les méthodes correspondantes.

Lorsque la méthode **Pleuvor(Potager potager)** est appelée, on génère aléatoirement **QuantEau** (entre 50 et 100 unités) et **QuantLum** (0 à 15) pour simuler un épisode plus ou moins intense de pluie et souvent nuageux (très peu de lumière). Pour chaque terrain du potager, on applique d’abord le coefficient d’absorption ($\text{terrain.NivEau} += \text{QuantEau} - \text{QuantEau} * \text{terrain.Absorption}$), puis on met à jour chacun des attributs **eauRecu** et **lumRecu** des plantes pour leur indiquer ce qu’elles peuvent puiser. On réduit ensuite de moitié **terrain.NivEau** pour simuler la consommation initiale et le drainage, et on abaisse la température courante de 3 °C dans “ContexteSimulation”. Enfin, la méthode déclenche automatiquement des événements biologiques en fonction de l’humidité et du type de sol : vers de terre et escargots apparaissent sur sols très humides, maladies comme le Pythium se déclarent si la combinaison froid/humide persiste et mauvaises herbes apparaissent sur des sols terreux ou argileux assez humides. Chaque apparition passe par **ApparaitAnimaux()**, **ApparaitMaladies()** ou **ApparaitMauvaiseHerbe()** du potager, puis est ajoutée aux listes **ListeAnimauxNuisibles** ou **ListeMaladie**, et enfin passe par les méthodes **Impacter()** ou **Contaminer()** et **Planter()**, qui activeront le mode urgence correspondant si nécessaire. Ce mode est aussi déclenché si le terrain est inondé, c’est-à-dire si son niveau d’eau dépasse sa capacité maximale ($\text{urgencelInondation} = \text{true}$).

La méthode **Ensoleiller(Potager potager)** simule un épisode d’ensoleillement en générant une **QuantLum** aléatoire (15 à 75), puis pour chaque terrain fait évaporer l’eau au sol proportionnellement à l’intensité lumineuse ($\text{terrain.NivEau} -= 0.5 * \text{QuantLum}$) et hausse la température de 3 °C dans le contexte global. Les plantes récupèrent la nouvelle réserve d’eau et le flux lumineux via leurs attributs **eauRecu** et **lumRecu**. L’ensoleillement favorise l’arrivée d’auxiliaires (abeilles si $\text{QuantLum} > 30$) et peut provoquer l’apparition de nuisibles lorsqu’un sol sableux devient trop sec (criquets). Au printemps, des oiseaux peuvent également survoler la parcelle, alors qu’une chaleur excessive et un sol desséché déclenchent des maladies comme l’anthracnose. Là encore, chaque apparition est relayée par le potager pour mettre à jour l’état de la simulation.

La grêle est modélisée par la méthode **Greler(Potager potager)** : elle fait baisser la température de 2 °C et supprime jusqu’à deux fruits par plante ($\text{p.nbFruitsActuel} -= 2$),

proportionnellement à la férocité de l'orage. Aucune mise à jour de l'eau du sol n'est réalisée, mais cet épisode peut suffire à déclencher une urgence si le joueur ne réagit pas rapidement pour couvrir ses cultures.

- ContexteSimulation

La classe "**ContexteSimulation**" agit comme un registre global de la temporalité et des conditions climatiques, accessible depuis n'importe quelle partie du code sans avoir à propager ces paramètres en argument de méthode. Elle contient seulement deux propriétés statiques : **SaisonEnCours** (saison actuelle) et **TempEnCours** (température actuelle). Ces propriétés sont mises à jour une fois chaque tour dans les classes Simulation et Météo. Toutes les autres classes peuvent ensuite accéder directement à **ContexteSimulation.SaisonEnCours** et **ContexteSimulation.TempEnCours** pour adapter leur comportement. En résumé, "**ContexteSimulation**" est un point d'accès unique et immuable pour la situation courante de la simulation, garantissant cohérence et lisibilité dans la gestion des effets de saison et de température à travers toutes les composantes du jeu.

- Simulation

Enfin, la classe **Simulation** assure une orchestration temporelle et événementielle différente selon les modes : normal ou urgent. La simulation est lancée grâce à la méthode **LancerSimulation()** dans laquelle la temporalité actuelle et les modes sont gérés avec la méthode **VerifierUrgence()**.

- Mode Normal:

Dans ce mode la classe **Simulation** maintient la date courante, calcule la saison avec **ObtenirSaison()**, génère la météo hebdomadaire (**GererMeteo()**), appelle tour à tour **GererEau()**, **Pousser()**, **PropagerToutes()** et **VerifierEtatPlantes()** sur chaque terrain, avant de proposer au joueur son menu d'actions (**ChoisirAction()**).

La fonction **ObtenirSaison(DateTime d)** permet de changer de saison en fonction du mois actuel : l'hiver tombe en décembre, le printemps arrive en mars, l'été commence en juin et l'automne en septembre.

La méthode **GererMeteo()** établit des intervalles de températures différents en fonction de la saison actuelle, et fait varier aléatoirement le type de météo en faisant appel aux méthodes **Pleuvrir()**, **Ensoleiller()** et **Greler()**. Ces changements sont affichés dans la console afin d'informer le joueur.

La méthode **ChoisirAction()** permet au joueur de gérer son potager en saisissant le numéro des actions qu'il a choisi de faire. Son nombre maximum d'action par tour est limité et

correspond à 3 actions par terrain existant dans le potager (`int nbActionsMax = 3*PotagerSimu.ListeTerrains.Count();`). Pour vérifier qu'il ne dépasse pas ce chiffre, cette méthode utilise une boucle do-while qui s'arrête lorsque le booléen `maxAtteint` devient true. Le joueur a tout de même la possibilité de sortir de la boucle quand il veut en sélectionnant le choix 6 qui fait un `return`. Il peut alors créer des terrains, planter des plantes, les arroser, récolter des fruits ou passer la semaine. Pour chaque action nous utilisons un do-while afin de vérifier si la saisie est mauvaise ou non, et dans ce cas le programme redemande au joueur de saisir sa réponse tant que celle-ci est incorrecte. En effet, cela permet de vérifier le numéro de l'action choisie, la quantité d'eau à verser, le type de plante à semer, les types de terrains et les places disponibles dessus.

La méthode fait également appel à la fonction `VerifierNumTerrain()` qui utilise le même système pour demander au joueur le numéro du terrain sur lequel il veut agir. En effet, le numéro du terrain est récupéré et converti en entier avec `int.TryParse(Console.ReadLine())`, puis il est stocké dans la variable `index`. Ensuite, on vérifie si l'index existe, c'est-à-dire s'il est compris entre 0 et le nombre de terrains total (`PotagerSimu.ListeTerrains.Count`), si c'est le cas le booléen `idxValide` est "true" et la fonction retourne l'index du terrain. Dans le cas contraire, `idxValide` est "false" et la boucle do-while recommence ce processus tant que l'index est invalide. Cette fonction permet d'alléger le code puisqu'elle est utilisée plusieurs fois.

- Mode Urgence:

En cas d'événement critique (grêle, intrusion, maladie grave), la simulation bascule en mode urgence, offre une vision journalière du potager pour le sauver, puis revient au rythme hebdomadaire quand la menace est écartée.

Dans un premier temps, la méthode `ModeUrgence()` va faire évoluer les obstacles sur les différents terrains où ils sont présents. Elle appelle la méthode `Propager()` puis `Contaminer()` en respectant la probabilité de contamination de chaque maladie, puis appelle les méthodes `Deplacer()` et `Impacter()` pour simuler les mouvements des animaux nuisibles contenus dans la `ListeAnimauxNuisibles` de chaque terrain.

Tout comme le mode normal, le mode urgence possède une méthode `ChoisirActionUrgente()` proposant au joueur un choix d'actions pour sauver son potager, comme chasser les animaux, traiter une maladie, couvrir ou désherber un terrain. De la même manière, elle vérifie la saisie du numéro de l'action à effectuer, l'animal à faire fuir, la maladie à traiter avec des boucles do-while et l'index du terrain sélectionné avec la méthode `VerifierNumTerrain()`.

3. Tests réalisés

Pour tester le bon fonctionnement du jeu nous avons réalisé des tests à chaque fois que nous codions une nouvelle méthode. Cela nous permettait de voir les incohérences comme les types de variable (int, bool, double, string) ou de classe (Plantes, Animaux, Terrains, Météo ...), puis de les modifier pour résoudre les problèmes. Ces tests nous permettaient aussi de gérer les sorties de grille, les erreurs de saisie, les boucles infinies et la clarté de l'affichage. Nous avons également effectué des tests lors des récupérations du code sur nos branches respectives et sur le main, afin de régler les conflits et s'assurer que les fonctionnalités du jeu restaient inchangées. Il arrivait, en travaillant chacune de notre côté sur les mêmes classes, que les noms de certaines variables diffèrent ou que de méthodes apparaissent et disparaissent.

Les tests effectués dans le **Program.cs** étaient très simples puisqu'il s'agissait seulement de créer un potager (`new Potager()`), ajouter le premier terrain (`AjouterTerrain()`), puis d'établir une météo (`new Meteo(0,0)`) et enfin créer une simulation (`new Simulation()`) et la lancer (`LancerSimulation()`). Ensuite, il suffisait de jouer avec les paramètres propres à nos méthodes pour vérifier le fonctionnement du jeu dans toutes les situations. Enfin, nous avons fait plusieurs parties en nous mettant à la place d'un joueur pour modifier l'affichage et obtenir un résultat agréable. Des photos illustrant l'affichage console de notre jeu sont disponibles en annexe.

II. Organisation de l'équipe

1. Planning de réalisation et répartition des tâches

Pour mener à bien ce projet nous avons tout d'abord établi un cahier des charges pour définir les fonctionnalités que nous voulions intégrer au jeu et identifier toutes les classes, sous-classes et leurs attributs et méthodes respectifs. Nous avons résumé tout ceci dans un tableau Excel (voir annexe), ainsi que dans le diagramme UML permettant de représenter les liens entre les classes.

Ensuite, nous nous sommes répartis les tâches, chacune d'entre nous travaillait sur des classes précises et en cas de blocage nous mettions en commun pour résoudre le problème. Cette méthode nous a permis d'avancer rapidement et d'éviter de stagner sur une partie du code en particulier.


Une fois les classes principales (Plantes, Terrains, Potager, Météo, ...) quasiment fonctionnelles nous avons pu commencer à coder la classe "Simulation", puis tester les tours de jeu avec les différents modes. Néanmoins, des modifications étaient constamment réalisées afin d'améliorer notre jeu.

La majorité du travail était effectué en dehors des créneaux de TD qui nous servaient principalement à poser des questions au professeur et faire un point sur notre avancée.

Nous codions donc sur notre temps libre, sans suivre de planning prédéfini, mais de manière régulière.

2. Matrice d'implication

Voir le fichier Excel.

 Matrice Implication Potager.xlsx

III. Bilan

1. Difficultés rencontrées

Lors de la réalisation de ce projet nous avons rencontré plusieurs difficultés pour lesquelles nous avons généralement trouvé une solution.

Tout d'abord nous avons remarqué qu'il était impossible d'afficher les maladies ou les animaux dans les terrains qui étaient créés à partir d'une grille de type Plantes. Les animaux et les maladies étant de classes différentes, nous ne pouvions pas les mettre dans la grille. Nous avons donc créé deux listes, ListeAnimauxNuisibles et ListeMaladie afin d'enregistrer leur apparition sur les terrains et nous avons signalé leur présence avec des `Console.WriteLine()`.

Un aspect sur lequel nous avons fait attention est l'appel de certaines méthodes dans d'autres méthodes situées dans des classes différentes. Il fallait bien rappeler l'origine de la méthode et entrer les bons paramètres.

Le mode urgence a posé un léger problème notamment pour gérer son arrêt. Initialement nous utilisions qu'un seul booléen qui devenait true lorsqu'une menace était écartée d'un terrain. De ce fait, lorsque plusieurs menaces apparaissent en même temps, le mode urgence se désactivait bien avant qu'elles ne soient toutes traitées. Nous avons alors créé des booléens pour chaque terrain selon le type d'obstacle, et pour sortir de ce mode il est à présent nécessaire que l'intégralité des booléens renvoie false.

Nous avons également eu des confusions dans la gestion des quantités, plus particulièrement le niveau d'eau des terrains, le besoin en eau des plantes, les inondations, ces paramètres étant tous liés.

Enfin, nous avons dû faire comprendre au code que les saisies des joueurs correspondaient à des éléments de type Animaux, Plantes ou Maladie. Nous ne pouvions pas comparer des string à des variables de type différents donc nous avons décidé d'utiliser les fonctionnalités "switch", "Any" et "First".

IV. Conclusion

Ce projet nous a permis de développer de nombreuses compétences en programmation orientée objet, notamment la création et l'utilisation de listes, la mise en place de classes mère et fille, l'usage des booléens, ainsi que la conception de méthodes et fonctions adaptées à différents cas. Nous avons également appris à mieux gérer les problèmes, notamment en analysant les messages d'erreur.

Tout de même, nous avons identifié des pistes d'amélioration comme la possibilité de revenir en arrière dans les menus des modes normal et urgence, ou encore l'affichage des animaux sur les terrains. Il serait également pertinent d'optimiser au maximum le code pour réduire les répétitions dans le code.

De plus, le travail en groupe a été efficace grâce à une bonne coordination et communication. Nous avons su éviter les problèmes en nous répartissant clairement les tâches, en nous aidant mutuellement face aux problèmes de code, et en partageant régulièrement l'avancement de chacune de nous.

Enfin, l'utilisation de Git a été bien maîtrisée, y compris la gestion des conflits lors des fusions entre les branches, ce qui a été crucial compte tenu du grand nombre de méthodes et de classes impliquées.

Finalement, ce projet nous a permis de progresser autant sur le plan technique que collaboratif, et constitue une base solide pour nos futurs travaux en programmation.

Annexes

```

Semaine du 06 juin 2025 - Saison : Eté

- Météo de la semaine -
Temperature moyenne : 25°C
Temps : pluvieux 🌧️

=== État du potager ===
Terrain numéro 0 de type Terre.
  0  1  2
0  +--+--+
  |  |  |
1  |  |  |
  |  |  |
2  |  |  |
  |  |  |
- Pomme (1,1) :
En bonne santé, croissance terminée (plante mature), besoin en eau : 69,325, nombre de fruits : 2.
Terrain numéro 1 de type Sable.
  0  1  2
0  +--+--+
  |  |  |
1  |  |  |
  |  |  |
2  |  |  |
  |  |  |
- Kiwi (1,2) :
En bonne santé, croissance terminée (plante mature), besoin en eau : -4,900000000000006, nombre de fruits : 2.

```

Fig 1 : Exemple d'affichage console de la simulation d'un potager

```
- Menu d'actions -  
Vous pouvez effectuer 6 actions maximum sur ce tour.  
1) Arroser un terrain, 2) Arroser une plante, 3) Planter une plante, 4) Ajouter un terrain, 5) Récolter un fruit, 6) Désherber, 7) Passer la semaine  
Entrez le numéro de votre choix :
```

Fig 2 : Menu d'actions du mode normal

```
- Stock de fruits -  
POMME : 2  
KIWI : 2  
Il vous reste 4 actions à utiliser.  
Tapez 1, 2, 3, 4, 5 ou 6 pour effectuer une action ; 7 pour continuer la simulation :
```

Fig 3 : Affichage du stock de fruits récoltés sur le potager

```
MODE URGENCE ACTIF !  
Jour 30 mars 2025 - Saison : Printemps  
  
⚠Présence d'un animal nuisible sur le terrain 1.  
Criquet s'est déplacé(e) sur cette position : (1,2)  
Votre plante est en danger !  
  
- Menu d'actions d'urgence -  
1) Chasser les animaux, 2) Traiter une maladie, 3) Désherber, 4) Recouvrir le terrain, 5) Passer un jour  
Entrez le numéro de votre choix :
```

Fig 4 : Exemple d'affichage du mode urgence