BORDEAUX INP Ensc

Projet programmation avancée : ENSemenC

Axelle Julien
Coraline Ohayon
1A - groupe 3 - promo 2027

Table des matières :

1. Introduction

Nous avons choisi de faire notre jardin aux Pays-Bas. Nous pouvons planter des tulipes, des jacinthes et des asters, sur de l'argile, du sable ou de la terre. Chaque fleur a un score de vie, calculé en fonction de différents paramètres.

Nous avons choisi de faire une simulation de la météo assez réaliste, cependant, tous les terrains ne se trouvent pas au même endroit, et les grosses intempéries ne les affectent pas tous.

Le jeu présente 2 modes : le mode classique et le mode d'urgence.

En mode classique, les terrains évoluent chaque mois, et le joueur peut choisir d'effectuer 2 actions au maximum, dont ne rien faire, semer, arroser, ou encore récolter. Des obstacles peuvent intervenir, comme des oiseaux, qui mangent les graines d'un terrain, ou une personne qui vient piétiner un terrain et tue donc des plantes. Il y a aussi des bonnes fées, comme les vers de terre qui fertilisent un terrain, ou des abeilles qui font éclore les fleurs.

Le mode urgence est enclenché à cause des obstacles d'urgence. Il peut s'agir d'une tempête, qui arrache la moitié des fleurs d'un terrain, ou d'une maladie grave qui leur fait perdre le tiers de leur score de vie. Pour pallier ça, le joueur peut reconstruire le terrain en plantant 3 nouvelles fleurs, ou donner une potion magique qui augmente le score des plantes.

2. Choix techniques

a. L'architecture générale (schéma UML)

L'architecture générale comprend 12 classes dont 3 abstraites.

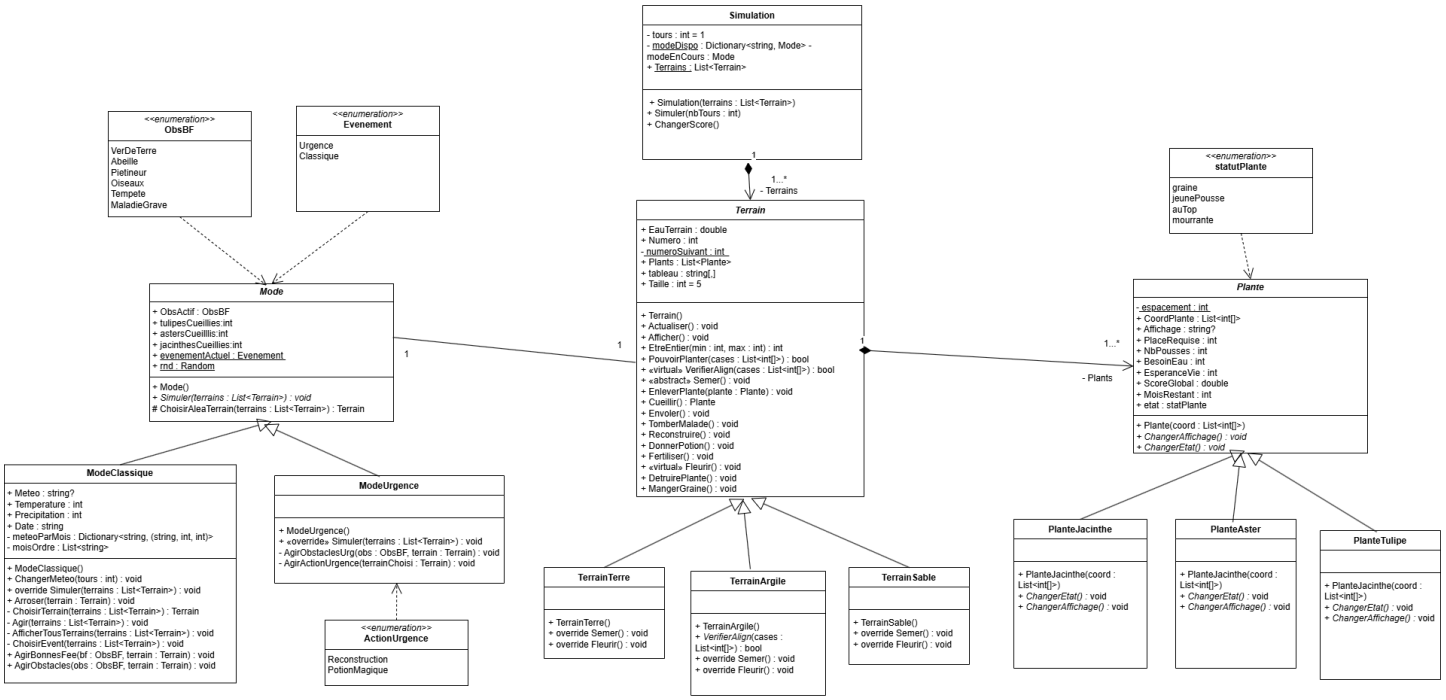

Diagramme UML

b. Les classes et leurs fonctionnement

Classe Plante

Nous avons une première classe abstraite Plante, qui regroupe les caractéristiques et les méthodes communes aux plantes.

Le constructeur permet d'instancier des plantes avec un liste de tableaux à 2 cases contenant le numéro de ligne et de colonne de chaque case occupée par un plant.

Elle présente différents attributs, comme les coordonnées des plantes, leur score de vie, leur état, le temps qu'il leur reste à vivre ou encore la place dont elles ont besoin pour grandir.

Elle contient les méthodes abstraites ChangerAffichage(), ChangerEtat(), qui sont définies dans les classes dérivées de Plante.

Classe PlanteTulipe

Il s'agit d'une classe dérivée de Plante. Elle reprend son constructeur, et définit en plus chacune des caractéristiques des tulipes. Chaque instance occupe une case d'un terrain argileux.

La méthode ChangerEtat() y est définie et change l'état de la tulipe selon son score de vie. L'état est un élément de l'énumération statutPlante définie dans la classe mère est

peut être graine, jeunePousse, auTop, mourrante, ou morte. Il s'agit toujours d'une graine lorsque la plante est instanciée.

La méthode ChangerAffichage() change l'attribut Affichage (chaîne de caractères) de la tulipe selon son état. Elle appelle d'abord ChangerEtat() afin que l'état soit à jour.

Classe PlanteJacinthe

Cette classe, dérivée de Plante, fonctionne exactement comme PlanteTulipe, mais adapte les attributs et les méthodes pour les jacinthes. Chaque instance occupe 2 cases sur un terrain sableux.

Classe PlanteAster

Cette classe, dérivée de Plante, fonctionne également de la même façon que PlanteTulipe, mais adapte les attributs et les méthodes pour les asters. Chaque instance occupe 3 cases sur un terrain terreux.

Classe Terrain

La classe Terrain est abstraite, et regroupe les attributs et les méthodes communes aux différents types de terrains. Nous avons la quantité d'eau du terrain EauTerrain, le numéro du terrain (Numero), qui s'incrémente automatiquement grâce à numeroSuivant, qui est statique. On trouve également une liste des plantes présentes sur le terrain (Plants), et un tableau de chaînes de caractère qui nous permettra d'afficher les terrains et leur contenu au fil du jeu. Il y a également l'entier Taille, d'office égal à 5.

Le constructeur permet de définir Plants comme une nouvelle liste lors de l'instanciation, d'attribuer un tableau de taille Taille=5 ne contenant que des "+", ainsi que de gérer l'auto-incrémentation des numéros de chaque terrain.

Il y a ensuite une méthode Actualiser(), qui, pour chaque plante du terrain (liste Plants), change son Affichage à l'aide de la méthode ChangerAffichage(), puis adapte le contenu de tableau selon son Affichage et les cases qu'elle occupe.

Nous avons ensuite créé une méthode Afficher(), qui appelle la méthode Actualiser() afin de mettre à jour le contenu du tableau du terrain, puis de l'afficher.

La méthode suivante, EtreEntier(int min, int max), permet de vérifier qu'une saisie utilisateur soit bien un entier, et qu'il soit compris entre une valeur minimum et une valeur maximum. Tant que ce n'est pas le cas, la saisie sera redemandée.

Nous avons également une méthode PouvoirPlanter(List<int[ ]> cases), qui retourne un booléen valant *true* si il est effectivement possible de planter sur les cases de la liste passée en paramètre, et *false* sinon. Elle vérifie que les cases soient disponibles, et que les cases autour ne soient pas occupées par une autre plante, afin que la plante semée ait suffisamment d'espace.

La méthode virtuelle VerifierAlign(List<int[ ]> cases) permet de vérifier que les cases de la liste passée en paramètres soient alignées, en vérifiant qu'elles sont sur la même ligne ou la même colonne.

Elle est virtuelle car elle est identique pour les classes dérivées TerrainSable et TerrainTerre, mais est redéfinie pour la classe dérivée TerrainArgile. Sur les terrains de sable et de terre, on offre à l'utilisateur la possibilité de planter uniquement des jacinthes et des asters, qui prennent plusieurs cases, alors que le terrain d'argile permet de planter uniquement des tulipes, qui prennent une seule case. La liste des coordonnées de la plante ne contiendra donc qu'un seul élément, et il sera alors inutile de la parcourir pour vérifier que les éléments sont alignés.

Nous avons ensuite déclaré la méthode abstraite Semer(), qui est redéfinie dans chaque classe dérivée de Plante.

La méthode EnleverPlante(Plante plante) prend une plante en paramètre, remplace les cases qu'elle occupe dans le tableau du terrain par des + et la supprime de la liste.

Nous avons ensuite créé la méthode Cueillir(), qui retourne une plante. Elle demande à l'utilisateur de choisir une seule case du terrain sur laquelle il veut cueillir une plante, puis vérifie que les coordonnées soient bien entières et dans le terrain grâce à la méthode EtreEntier(int min, int max). Ces coordonnées sont ensuite rangées dans un tableau. Pour chaque plante de la liste Plants du terrain, on regarde ensuite si sa liste de coordonnées contient le tableau que l'on vient de créer, et si c'est le cas, on range la plante dans la variable cueillette. On regarde finalement si cette variable est nulle ou pas. Si c'est le cas, on n'a pas trouvé de plante sur la case indiquée et on l'affiche, sinon on appelle EnleverPlante(Plante plante) et on affiche que la plante a été cueillie, puis on la retourne, ce qui nous permettra par la suite de faire le compte des plantes cueillies.

La méthode Envoler() permet d'enlever une plante sur deux du terrain, en se servant d'EnleverPlante(Plante plante).

La méthode TomberMalade() fait perdre un tiers de score de vie à toutes les plantes du terrain.

La méthode Reconstruire() permet de semer 3 plantes, à l'aide de Semer().

La méthode DonnerPotion() permet d'augmenter d'un quart le score de vie de toutes les plantes d'un terrain.

La méthode Fertiliser() permet d'augmenter d'un sixième le score de vie de toutes les plantes d'un terrain. Elle sera appelée lors des interventions des vers de terre, et affiche donc que le terrain va être fertilisé grâce à ces derniers.

La méthode virtuelle Fleurir() est déclarée et sera appelée lors des interventions des abeilles. Elle permet pour l'instant d'indiquer l'action des abeilles, et sera complétée dans les classes dérivées de Terrain. Elle permettra d'adapter le score de chaque plante pour qu'elle soit au meilleur de sa forme, et cette valeur de score variera selon chaque type de plante et donc selon chaque terrain.

La méthode DetruirePlante() sera appelée lorsqu'il y aura des piétineurs sur un terrain, et en supprime un quart des plantes à l'aide d'EnleverPlante(Plante plante).

Finalement, la méthode MangerGraine() enlève toutes les graines d'un terrain (plantes dont l'état vaut "graine") et l'indique au joueur.

Classe TerrainArgile

La classe TerrainArgile caractérise les terrains argileux, sur lesquels le joueur ne pourra planter que des tulipes. Elle est dérivée de Terrain.

Elle reprend le constructeur de la superclasse Terrain en spécifiant que la quantité d'eau EauTerrain est initialement égale à 80.

Elle redéfinit tout d'abord la classe la méthode VerifierAlign(List<int[ ]> cases) afin de retourner true comme expliqué auparavant.

Nous redéfinissons ensuite la méthode abstraite Semer(), qui demande au joueur sur quelle case il souhaite semer, vérifie ses saisies, puis instancie une nouvelle tulipe (PlanteTulipe) après avoir vérifié que c'était possible à l'aide des méthodes VerifierAlign et PouvoirPlanter.

Finalement, nous redéfinissons la méthode Fleurir() comme expliqué précédemment.

Classe TerrainSable

Cette classe fonctionne comme TerrainArgile et est dérivée de Terrain. Elle caractérise les terrains sableux sur lesquels le joueur ne pourra planter que des jacinthes.

Cependant, elle initialise l'eau du terrain à 80 et demande à l'utilisateur de rentrer 2 cases dans la méthode semer, car c'est la place requise par les jacinthes.

Classe TerrainTerre

Cette classe fonctionne également comme TerrainArgile et est dérivée de Terrain. Elle caractérise les terrains terreux sur lesquels le joueur ne pourra planter que des asters.

Cependant, elle initialise l'eau du terrain à 50 et demande à l'utilisateur de rentrer 3 cases dans la méthode semer, car c'est la place requise par les asters.

Classe Mode

La classe mode est une classe abstraite. Elle a pour attribut une énumération qui rassemble les obstacles ou bonnes fées qui apparaissent dans chaque tour de simulation. Les autres attributs sont : l'obstacle en cours ObsBFActif, l'énumération d'événements qui permet d'indiquer quand on passe en mode classique ou en mode urgence, l'événement

actuel evenementActuel qui est statique, un Random rnd et enfin trois entiers tulipesCueillies, jacinthesCueillies et astersCueillies qui permettent de tenir le compte de la cueillette.

Le constructeur Mode() est vide. Il y a deux autres méthodes : Simuler(List<Terrain> terrains) qui prend en paramètre une liste de terrain et qui est abstraite car elle sera redéfinie de deux façons différentes pour les classes dérivées ModeClassique et ModeUrgence et ChoisirAleaTerrain() qui retourne un terrain. Cette méthode est protected car elle va être utilisée uniquement dans les classes dérivées et prend en paramètre une liste de terrain. Après avoir compté le nombre d'éléments dans cette liste, on applique le tirage aléatoire à la variable rnd. Ce numéro aléatoire, choisi entre 1 et le nombre de terrain, permet, en parcourant les terrains de la liste, de trouver un terrain qui a un numéro égal. Ce terrain est renvoyé par la méthode.

Finalement, nous avons redéfini la méthode ToString() de façon à ce qu'elle affiche le compte de chaque plante.

Classe ModeClassique

Le mode classique est une classe dérivée de la classe Mode. Cette classe permet de simuler le mode classique en affichant la météo, permettant au joueur de choisir une action à réaliser et en activant les bonnes fées ou les obstacles.

Elle a pour attributs tout ce qui se rapporte au temps : Meteo, Temperature, Precipitation, ainsi que la Date actuelle qui est initialisée en janvier. Il y a également une liste de mois moisOrdre et un dictionnaire qui a pour clé un string qui correspond à un mois de l'année et a en valeur un ensemble de chaîne de caractères et d'entiers qui se rapportent à la météo, la température et au niveau de précipitation.

Le constructeur ModeClassique() ne prend pas de paramètres, mais utilise la méthode ChangerMeteo(1).

La méthode ChangerMeteo() prend en paramètre le tour actuel. Elle permet de changer la météo en fonction du tour de la simulation dans lequel on se trouve. En effet, dans la méthode, on récupère le tour actuel et change la date actuelle en fonction. On parcourt le dictionnaire meteoParMois et on change le temps actuel.

La méthode Simuler() est une méthode redéfinie qui prend en paramètre une liste de terrains. Elle permet d'afficher la simulation liée au mode classique dans la console avec la météo, l'affichage des terrains, la possibilité de choisir de faire deux actions et le choix aléatoire de l'obstacle ou de la bonne fée. Cette méthode appelle les méthodes suivantes : AfficherTousTerrains(), Agir() et ChoisirEvent().

La méthode Arroser(Terrain terrain) prend un terrain en paramètre et augmente de 10 sa quantité d'eau EauTerrain, et affiche qu'il a été arrosé.

La méthode ChoisirTerrain() est une méthode privée qui prend en paramètre une liste de terrain. Elle demande à l'utilisateur de choisir un numéro de terrain, vérifie la saisie, et retourne le terrain correspondant à ce numéro dans la liste passée en paramètre.

La méthode Agir() est une méthode qui prend également en paramètre une liste de terrains. Elle présente les actions que le joueur peut effectuer avec leur numéro :

- 0 : pas d'action
- 1 : semer
- 2 : arroser
- 3 : récolter.

Elle lui demande ensuite de saisir le numéro de l'action qu'il choisit et vérifie la saisie.

S'il a choisi de ne pas agir, il ne se passe rien.

S'il a choisi de semer, on lui fait choisir sur quel terrain avec ChoisirTerrain() et on appelle Semer() de la classe Terrain sur ce terrain.

S'il a choisi d'arroser, on lui fait choisir sur quel terrain avec ChoisirTerrain() et on appelle Arroser(terrainChoisi) afin d'arroser le terrain en question.

Enfin, s'il a choisi de récolter, on lui fait choisir sur quel terrain avec ChoisirTerrain() , on appelle la fonction Cueillir() de la classe Terrain sur ce terrain. Enfin, si la plante cueillie était au meilleur de sa forme, on ajoute son nombre de pousses à la plante cueillie correspondante. Cela nous permettra ensuite d'afficher le nombre de plantes récoltées en tout à chaque tour.

```
recolte = terrainChoisi.Cueillir();
            if (recolte is PlanteTulipe &&
recolte.etat==Plante.statutPlante.auTop)
            {
                tulipesCueillies += recolte.NbPousses;
            }
            else if (recolte is PlanteJacinthe &&
recolte.etat==Plante.statutPlante.auTop)
            {
                jacinthesCueillies += recolte.NbPousses;
            }
            else if (recolte is PlanteAster &&
recolte.etat==Plante.statutPlante.auTop)
            {
                astersCueillis += recolte.NbPousses;
            }
```

La méthode AfficherTousTerrains() prend en paramètre une liste de terrains, la parcours et affiche les terrains un à un grâce à terrain.Afficher().

La méthode ChoisirEvent() est une méthode private qui prend en paramètre une liste de terrains. Elle a pour but de choisir le type d'événement qui va arriver (obstacle, obstacle

d'urgence ou bonne fée) de façon aléatoire grâce au rnd.Next(). Quand le type d'évènement est choisi, l'évènement en lui-même est aussi choisi de manière aléatoire et se réalise en appelant la fonction d'action adéquate. S'il s'agit d'un obstacle d'urgence, la variable evenementActuel passe de classique à urgence pour permettre le passage en mode urgence dans la classe de simulation.

```
int x = rnd.Next(1, 8);       // choisir si c'est une bonne fée,
obstacles ou obstacles d'urgence
        if (x <= 3)      // 3/8 chances de tomber sur une bonne fée
        {
            int y = rnd.Next(0, 2);      // on choisit aléatoirement
quel évènement va arriver parmi les bonnes fées
            ObsBFActif = (ObsBF)y;
            AgirBonnesFee(ObsBFActif, terrainTrouve);
        }
        else if (x <= 6)     // 3/4 chances de tomber sur une bonne fée
        {
            int y = rnd.Next(2, 4);
            ObsBFActif = (ObsBF)y;
            AgirObstacles(ObsBFActif, terrainTrouve);
        }
        else    // 1/8 chance de tomber sur un évènement qui déclenche
le mode urgence
        {
            int y = rnd.Next(4, 6);
            ObsBFActif = (ObsBF)y;
            evenementActuel = Evenement.Urgence;    //on va passer en
mode urgence
        }
```

La méthode AgirBonnesFee() prend en paramètre une valeur de type ObsBF qui correspond à une bonne fée dans ce cas et un terrain. Cette fonction est appelée dans ChoisirEvent() et permet, en fonction du nombre aléatoire tiré dans cette fonction, d'appeler les méthodes d'action des bonnes fées, c'est-à-dire Fertiliser() ou Fleurir() sur le terrain passé en paramètre.

La fonction AgirObstacles() agit de la même manière, en appelant les méthodes liées au obstacles : DetruirePlante() et MangerGraine() sur le terrain tiré.

ClasseModeUrgence

La classe ModeUrgence est une classe dérivée de la classe Mode. Elle permet de gérer la simulation dans le cas d'urgence : afficher le terrain touché par l'obstacle d'urgence et agir en fonction.

Elle a pour attributs une énumération ActionsUrgence qui rassemble les actions d'urgence possibles (reconstruction et potion magique).

Le constructeur est vide. Il existe 3 autres méthodes. La première est la méthode Simuler(). Elle est redéfinie et prend en paramètre une liste de terrains. Elle affiche la cause de l'urgence, appelle des méthodes pour choisir un terrain, l'afficher, voir les dégâts de l'obstacle et agir en urgence. Enfin la variable evenementActuel prend la valeur "classique" pour permettre de changer de mode dans la classe Simulation.

La fonction AgirObstaclesUrg(), permet d'agir en fonction de l'obstacle d'urgence tiré comme pour les bonnes fées et obstacles classiques et est appelée dans la méthode Simuler().

La méthode AgirActionUrgence() prend en paramètre le terrain choisi et permet à l'utilisateur de choisir l'action qu'il souhaite mettre en place pour contrer la situation d'urgence. Elle permet au joueur de reconstruire ou donner une potion magique en appelant les méthodes correspondantes pour le terrain choisi. Cette méthode a un fonctionnement similaire à celui d'Agir() dans le mode classique.

Classe Simulation

La classe Simulation permet, comme son nom l'indique, de mettre en place la simulation de notre jardin.

Elle a comme attributs le nombre de tours, qui est initialisé à 1, et un dictionnaire qui a pour clé le mot "Classique" ou "Urgence" et qui a pour valeur son mode correspondant. Il y a aussi la variable modeEnCours qui est de type Mode et une liste de terrains statique.

Le constructeur Simulation() prend en paramètre une liste de terrains et initialise le mode en cours au mode classique. La méthode suivante est la méthode Simuler() qui prend en paramètre un nombre de tours. Elle affiche les consignes dans la console, puis appelle une boucle for pour les tours de simulation. Dans cette boucle, quand le mode est classique, la météo est changée selon le mois (à chaque tour, un mois passe) et le pourcentage d'eau du terrain est géré en fonction des précipitations du tour. Ensuite, on appelle la fonction modeEnCours.Simuler() en fonction du mode en cours : c'est la méthode dans le mode classique. Le mode en cours peut changer de valeur dans le if suivant. On entre dans cette boucle if seulement quand Mode.evenementActuel == Mode.Evenement.Urgence, c'est-à-dire, quand dans ModeClassique un obstacle d'urgence est tiré aléatoirement dans la méthode ChoisirEvent(). Dans ce if, on appelle la fonction Simuler() du mode urgence, puis on change la valeur du mode en cours en mode classique avant la fin de la boucle for.

```
      if (Mode.evenementActuel == Mode.Evenement.Urgence)      //si on
passe en mode urgence
                  {
```

```
                    modeEnCours =
modeDispo.GetValueOrDefault("Urgence") ?? new ModeUrgence();
                    modeEnCours.ObsBFActif =
modeDispo["Classique"].ObsBFActif; // transmettre l'urgence en cours
                    modeEnCours.Simuler(Terrains!);
                    modeEnCours =
modeDispo.GetValueOrDefault("Classique") ?? new ModeClassique();
//retour mode classique
                }
```

La dernière méthode est ChangerScore(), qui permet de changer le score global d'une plante. On parcourt chaque terrain, puis chaque plante dans ces terrains et si le nombre de mois de vie restant n'est pas nul, il diminue ainsi que le score global. S'il est nul, le score global le devient aussi et on appelle terrain.Enlever(plante). La méthode ChangerScore() est appelée à la fin de chaque tour dans Simuler(), pour que le score de la plante diminue chaque mois quand elle vieillit.

Programme principal

Afin de tester tout notre jeu, nous avons instancié 2 terrains argileux, 2 terrains sableux, et 2 terrains terreux dans le programme principal. Nous avons ensuite instancié une simulation avec une liste contenant ces terrains.

De plus, nous avons lancé une simulation de 36 mois.

Cependant, afin de tester notre code au fur et à mesure, nous avons été amenées à lancer des simulations bien plus courtes.

3. Organisation de l'équipe

a. Description de l'équipe

L'équipe est composée d'Axelle Julien et de Coraline Ohayon. Nous avons chacune passé autant de temps sur le projet, et la difficulté des tâches à réaliser a également été répartie équitablement.

Nous nous sommes réparties les tâches, puis avons expliqué notre code à l'autre, et l'avons débogué et amélioré ensemble lorsque cela s'y prêtait. De façon générale, nous avons travaillé seules sur nos tâches respectives entre les séances de TD, et nous avons mis en commun pendant ces séances, puis nous avons continué d'avancer chacune de notre côté ou ensemble selon les tâches à réaliser.

b. Matrice d'implication

MATRICE D'IMPLICATION : PROJET PROGAV		
projet : Ensemensc	**version: VF**	**date:**

				23/05/2025
MEMBRES DE L'ÉQUIPE: Coraline OHAYON et Axelle JULIEN				
n°	**Tâche/Fonctionnalité/Fonction/Classe**	**Nom du/des Codeurs**	**Pourcentage de participation si tâche partagée**	**Pourcentage du Projet Global**
1	Mode.cs	Axelle		7
2	ModeClassique.cs	Axelle / Coraline	60 / 40	10
3	ModeUrgence.cs	Axelle		10
4	Simulation.cs	Axelle		8
5	Plantes.cs	Coraline		7
6	PlanteTulipe.cs	Coraline		6
7	PlanteJacinthe.cs	Coraline		6
8	PlanteAster.cs	Coraline		6
9	Terrain.cs	Axelle / Coraline	50 / 50	10
10	TerrainArgile.cs	Axelle / Coraline	40 / 60	6
11	TerrainSable.cs	Axelle / Coraline	40 / 60	6
12	TerrainTerre.cs	Axelle / Coraline	40 / 60	6
13	Programme.cs	Axelle / Coraline	50 / 50	2
14	débogage et relecture du code	Axelle / Coraline	50 / 50	5
15	rédaction du rapport	Axelle / Coraline	50 / 50	5
REMARQUES :				

c. Planning d'exécution

Lors de la première séance de TD, nous avons créé le dépôt git et avons commencé à réfléchir à la structure du projet et à faire un diagramme des classes.

Nous avons ensuite commencé à coder, et avons avancé lors des séances de TD dédiées. Nous avons également avancé en dehors de ces séances, et avons fini par rédiger le rapport et compléter la matrice d'implication.

4. Bilan

Même si nous avons rencontré quelques difficultés, nous avons finalement réussi à répondre à la consigne et à coder toutes les fonctionnalités attendues.

Nous aurions aimé ajouter des fonctionnalités supplémentaires.
Par exemple, la fonction VerifierAlign(), dans la classe Terrain, nous permet de vérifier que toutes les cases en paramètres se trouvent bien sur la même ligne ou sur la même colonne, mais ne nous permet pas de vérifier qu'elles sont juxtaposées.Un joueur peut donc sélectionner des cases éloignées pour un même plant, ce qui est dommage.

De plus, bien que lorsque le joueur choisit de semer, le type de fleur lui est imposé grâce à la méthode Semer() qui est redéfinie dans chaque classe dérivée de Terrain, et donc dans chaque type de terrain, il n'est pas imposé par ailleurs dans la construction des classes.

Nous aurions également pu ajouter d'autres actions ou d'autres types de plantes par exemple.

L'utilisation de GitHub nous a posé quelques problèmes, ce qui a pu nous faire perdre beaucoup de temps. Nous avons malgré tout pu trouver des solutions afin de continuer le projet.

Nous n'avons pas eu de problème d'organisation de l'équipe, nous avons réussi à nous répartir équitablement les tâches qui s'y prêtaient, et à réfléchir ensemble sur les autres tâches. Nous avons passé autant de temps sur le projet l'une que l'autre.