ENSemenC

Justifications techniques

LEBRETON Gwenaëlle, GRONDIN Abigaëlle, CHAUVEL Eloïse

1A ENSC, Groupe 3

Promotion 2027

SOMMAIRE

I- Introduction

Ce document de justification technique a pour objectif de clarifier le code C# proposé pour créer le jeu de simulation de potager ENSemenC. Nous avons développé le jeu sur Visual Studio et nous avons utilisé Github afin de pouvoir collaborer entre nous et de partager nos avancées de programmation du jeu.
Dans ce rapport nous aborderons tout ce que nous avons utilisé afin de pouvoir faire marcher notre jeu de simulation de potager (classes abstraites ou non, classes dérivées, constructeurs, méthodes, etc.), ainsi que les méthodes d'organisation que nous avons mises en place au sein de notre équipe.

II- L'univers du jeu et ses spécificités

Le simulateur de potager ENSemenC que nous avons développé plonge le joueur dans une expérience immersive de jardinage virtuel. L'univers du jeu permet de cultiver une grande variété de plantes (fruits, fleurs, légumes, herbes) et de croiser des créatures imaginaires telles qu'un géant, un dragon ou encore des bonnes fées.

Les caractéristiques principales des terrains :

- **Environnement dynamique** : Un système météorologique qui permet au joueur d'en savoir un peu plus sur le monde dans lequel il va faire évoluer ses plantes.
- **Système de surveillance par webcam** : Détection des obstacles permettant une réaction rapide du joueur, notamment lorsque le mode urgence est déclenché.
- **Double temporalité** : Alternance entre le mode classique et le mode urgence.
- **Diversité végétale** : Large choix de plantes avec des caractéristiques propres (légume, herbe, fleur et fruit) qui évoluent au fil du jeu.
- **Éléments imaginaires** : Inclusion d'individus imaginaires (pas de géant, dragons, bonnes fées) pensés pour animer le jeu et donner au joueur des clés pour agir sur son terrain.

Notre jeu possède une originalité particulière grâce à la modélisation détaillée des plantes et de leurs interactions avec l'environnement. Chaque végétal possède des coordonnées précises (PositionX, PositionY) et des attributs spécifiques comme les besoins en eau, en lumière, en type de sol, ainsi que des vulnérabilités aux maladies et parasites. Cette complexité permet une simulation réaliste qui encourage le joueur à développer une véritable stratégie de jardinage.

III- Structure du jeu : déroulement d'une partie et possibilités pour les joueurs

A- Classe Simulation

```
public class Simulation
{
    1 reference
    public void ChoisirTerrain(Texte t)
    {
        Console.WriteLine("Hello, il est maintenant temps de choisir à quoi va ressembler ton terrain.");
        t.DefinirTerrain();

        ConsoleKey clic = Console.ReadKey().Key;

        Terrain terrainChoisi = null;


        switch (clic)
        {
            case ConsoleKey.D1:
                terrainChoisi = new TerrainArgileux();
                break;
            case ConsoleKey.D2:
                terrainChoisi = new TerrainLimoneux();
                break;
            case ConsoleKey.D3:
                terrainChoisi = new TerrainForestier();
                break;
            case ConsoleKey.D4:
                terrainChoisi = new TerrainSableux();
                break;
            default:
```

Extrait du code de la classe Simulation

La classe Simulation sert de point d'entrée principal pour initier une partie dans notre jeu de gestion de potager. Sa structure repose sur une méthode centrale : ChoisirTerrain(), qui permet au joueur dès le début d'une partie de choisir le terrain de sémence de son choix entre quatre terrains différents : TerrainSableux 🟨, TerrainLimoneux 🟫, TerrainForestier 🟩et TerrainArgileux 🟧. Ce choix est réalisé via un système de lecture de touches clavier (ConsoleKey), ce qui rend l'expérience interactive.

Une fois le terrain sélectionné, il est initialisé et affiché, ce qui permet de visualiser la grille du potager avant de commencer. Ensuite, le mode de jeu ModeClassique est lancé, ce qui enchaîne directement sur le cœur de la simulation (choix des plantes, actions, etc.).

En résumé, Simulation agit comme le gestionnaire de lancement du jeu : il combine interface utilisateur simple, instanciation dynamique de terrain, et enchaînement logique vers le gameplay.

B- Mode classique

```
public class ModeClassique
{
    0 references
    private Random ramdom = new Random();

    1 reference
    public void LancerPartie(Terrain terrain, Texte t)
    {
        t.PresenterFleurs();
        ConsoleKey clic = Console.ReadKey().Key;
        Console.WriteLine();

        Plante? fleurChoisie = null;

        switch (clic)
        {
            case ConsoleKey.D1:
                fleurChoisie = new Tulipe();
                break;
            case ConsoleKey.D2:
                fleurChoisie = new Rose();
                break;
            case ConsoleKey.D3:
                fleurChoisie = new Tournesol();
                break;
            default:
                Console.WriteLine("Action invalide");
                return;
        }
```

Extrait du code de la classe ModeClassique

La classe ModeClassique incarne le mode de jeu principal basé sur une progression par tours dans notre simulateur de potager. Sa structure repose sur une méthode clé LancerPartie() qui guide le joueur pas à pas à travers les différentes étapes de plantation sur le potager. Cette classe s'appuie majoritairement sur les méthodes codées dans les classes Terrain et Plante.

Le mode classique offre une expérience de jardinage structurée et progressive :

Plantation :

- Sélection séquentielle de plantes par catégories : Fleurs (Tulipe, Rose, Tournesol), Fruits (Fraise, Pastèque, Pomme), Légumes (Aubergine, Carotte, Piment, Salade, Tomate)
- Placement automatique des plantes sur des positions aléatoires libres du terrain
- Plante semée récemment prend l'apparence d'un semi : 🌱

Entretien :

- 3 actions possibles par tour : Arroser, Soigner, Exposer au soleil (choix doné 3 fois d'affilée)
- Chaque action influence directement la santé et la croissance des plantes. But = atteindre l'état final qui fait apparaître la plante mature sur le terrain (ex : 🍎)

Toutefois, un dysfonctionnement apparaît lorsqu'une plante autre qu'une fleur est choisie. En effet, bien que l'arrosage soit censé augmenter la croissance de 1,5, ce qui permet normalement à la plante de se développer jusqu'à son état final , cette progression fonctionne correctement pour une fleur, mais pas pour un fruit ou un légume.

Par ailleurs, la classe gère également l'affichage du terrain et des plantes à chaque étape, rendant l'expérience immersive. Ce mode permet donc d'observer les effets directs de chaque décision du joueur.

Action des obstacles et des bonnes fées :

Après que le joueur ait fini de faire ses actions choisies, un obstacle aléatoirement déterminé parmi les 4 rentre en action. Si l'obstacle déterminé n'est pas un dragon, il fait alors son action et le joueur est informé des dégâts infligés sur son terrain. C'est alors ensuite aux bonnes fées d'agir et d'aider le joueur, informé cette fois-ci des bienfaits venant d'être produits (partie développée dans Simulation.cs). Enfin, le tour (une semaine) se termine et le joueur est à nouveau amené à faire des choix.

Cependant, si au moment des obstacles, le hasard désigne le dragon, le mode Urgence est alors activé car l'ensemble des plantes du potager se retrouvent brûlées. Après avoir choisi une action parmi 2, le tour du joueur se termine sans passer cette fois-ci par l'action des bonnes fées.

C- Mode urgence

```
public class ModeUrgence
{
    20 references
    private Terrain terrain;
    2 references
    private Dragon dragon;
    2 references
    private Random random;

    0 references
    public ModeUrgence(Terrain terrain)
    {
        this.terrain = terrain;
        this.dragon = new Dragon { Univers = terrain }; // on lie le dragon au terrain
        this.random = new Random();
    }

    0 references
    public void DeclencherAttaqueDragon()
    {
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("🚨 ALERTE URGENCE : DRAGON EN APPROCHE !");
        Console.ResetColor();

        Console.WriteLine($"🐉 Un dragon attaque le terrain {terrain.Nom} ({terrain.TypeSol}) !");
        Console.WriteLine($"🔥 Température : {terrain.Temperature}°C | Météo : {terrain.Meteo}");
        Thread.Sleep(1000);
```

Extrait du code de la classe ModeUrgence

La classe ModeUrgence représente un mode de jeu alternatif et critique où un événement destructeur, ici une attaque de dragon, vient perturber la gestion normale du potager. Elle se compose comme ModeClassique d'un terrain sur lequel les plantes sont cultivées, d'un dragon lié à ce terrain, qui peut déclencher une attaque et enfin d'un générateur aléatoire, utilisé pour les choix de plantes sacrifiées.

Déclenchement automatique :

- Alerte visuelle dramatique avec codes couleur (rouge pour danger)

- Présentation immédiate du terrain en situation de crise. Le terrain prend feu car le dragon l'attaque, nous voyons la propagation de l'incendie en temps réel 🔥

Gestion de l'attaque du dragon :

- Choix stratégique à faire entre deux approches de sauvetage
- Conséquences durables et irréversibles sur l'écosystème du jardin

Options de réponse :

1. Protection totale : Toutes les plantes survivent mais deviennent malades (contamination par "fumée du dragon")
2. Évacuation partielle : 50% des plantes sont sauvées en parfaite santé, les autres sont définitivement perdues

La méthode DeclencherAttaqueDragon() simule une attaque immédiate, avec un affichage visuel. Le dragon agit sur le terrain en déclenchant un feu impactant la globalité du terrain (dragon.Agir()). Après l'attaque du dragon, la méthode ProposerChoixUrgence() invite le joueur à réagir via deux options. La première correspond à la protection totale, toutes les plantes sont conservées mais deviennent malades (ExecuterProtectionTotale()). La deuxième permet le sauvetage partiel de quelques plantes présentes sur le terrain, ainsi une moitié aléatoire des plantes est sauvée, l'autre est détruite (ExecuterEvacuationPartielle()).
Enfin la méthode NettoyerTerrainEtPlacerPlantes() est appelée pour retirer les traces de feu et repositionner les plantes survivantes sur la grille.

IV- Modélisation objet

Notre jeu s'appuie sur une architecture orientée objet plutôt rigoureuse. Nous avons exploité les mécanismes d'héritage, de polymorphisme et d'encapsulation pour modéliser efficacement notre simulateur de potager.

A- Diagramme de classes UML simplifié

Nous avons réalisé un diagramme UML qui répertorie les différentes classes mères qui sont au nombre de 7 (en vert plus foncé) et les classes dérivées qui sont au nombre de 28 (vert plus clair). Nous avons modélisé les relations entre elles et détaillé les attributs et les méthodes utilisés dans chacune des classes. Cela nous permet d'avoir une architecture claire et complète de notre jeu de simulation de potager.

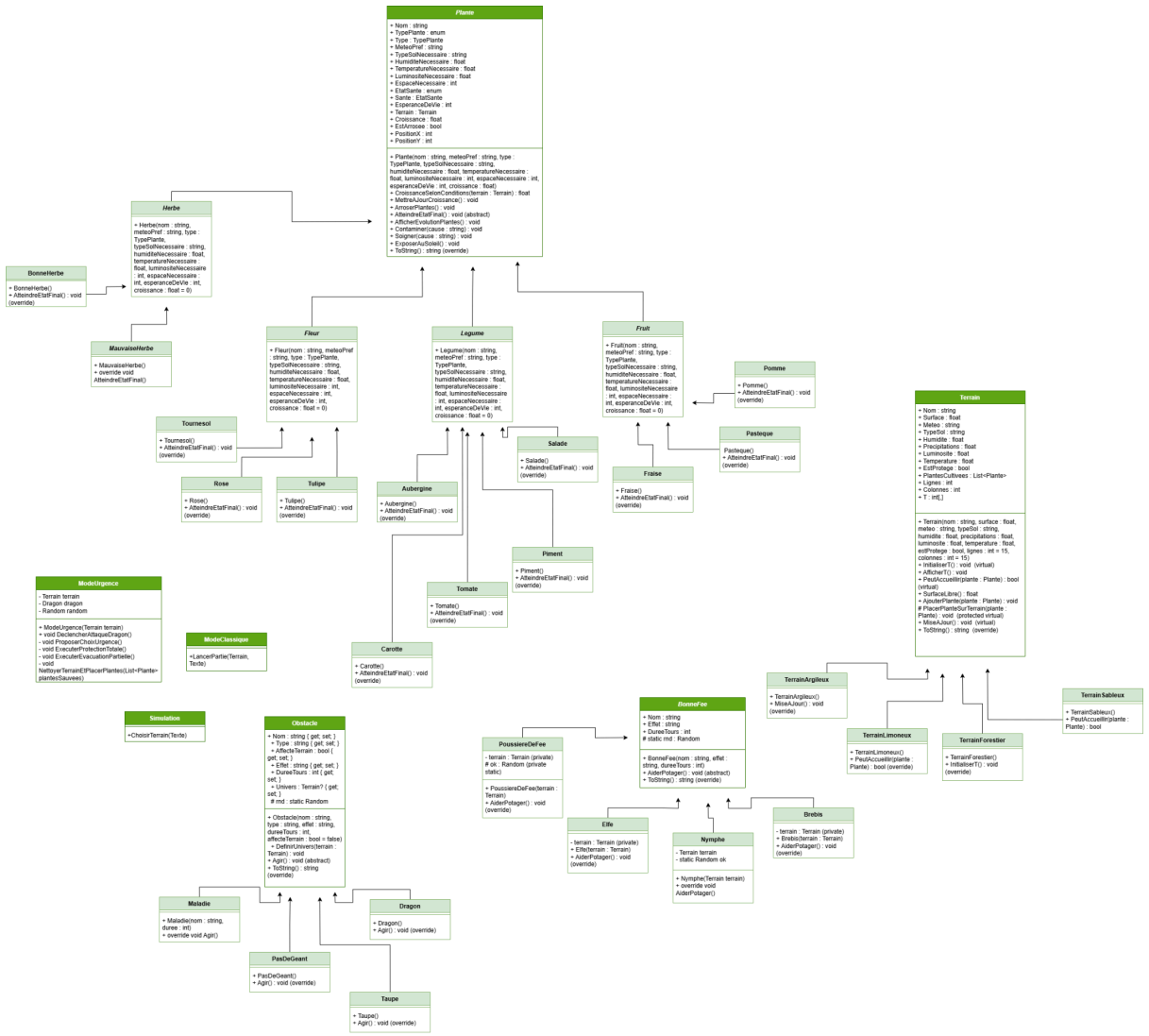

Diagramme UML de notre simulateur de potager effectué sur draw.io

B- Classes du jeu

Le simulateur de potager est organisé autour de 7 classes principales (voir diagramme ci-dessus) réparties en plusieurs catégories fonctionnelles.

1. Architecture des terrains

La classe abstraite Terrain représente l'environnement où les plantes sont cultivées. Elle intègre plusieurs paramètres essentiels à la simulation de notre jeu. Elle est fondamentale pour le bon déroulement de notre jeu.

Cette classe repose sur un système de matrice visuelle : une grille de 15x15 cases modélisée par un tableau T[,], où chaque cellule est codée numériquement pour représenter l'état du terrain (0 = vide, 1 = semis, 21 = feu, etc.). Pour visualiser le terrain on utilise la méthode AfficherT(), qui permet de visualiser les différents émojis que nous avons choisis pour rendre plus agréable la visualisation de notre potager virtuel.

Ensuite, la classe intègre une gestion dynamique de l'espace, calculant automatiquement la surface libre SurfaceLibre() en fonction des plantes déjà cultivées, et évaluant la possibilité d'en ajouter de nouvelles grâce à la méthode PeutAccueillir(). Enfin, le positionnement intelligent des plantes est assuré par la méthode PlacerPlanteSurTerrain(), qui sélectionne aléatoirement une case vide pour chaque nouvelle plante, tout en enregistrant ses coordonnées exactes (x, y) pour un suivi précis de son emplacement.

En complément, le terrain prend également en compte des paramètres environnementaux (météo, humidité, luminosité, température) mis à jour dynamiquement avec MiseAJour(), dans le but de donner plus de réalisme à notre jeu.

```
public abstract class Terrain
    public int Lignes { get; set; }
    15 references
    public int Colonnes { get; set; }

    // Matrice représentant l'état du terrain (0=vide, 1=semis, ...)
    40 references
    public int[,] T { get; set; }

    4 references
    public Terrain(string nom, float surface,  string meteo, string typeSol, float humidite, float precipitations, fl
    {
        Nom = nom;
        Surface = surface;
        Meteo = meteo;
        TypeSol = typeSol;
        Humidite = humidite;
        Precipitations = precipitations;
        Luminosite = luminosite;
        Temperature = temperature;
        EstProtege = estProtege;
        PlantesCultivees = new List<Plante>();
        Lignes = lignes;
        Colonnes = colonnes;
        T = new int[Lignes, Colonnes]; // initialise matrice vide
    }

    // Initialiser la matrice T à vide (0)
    3 references
    public virtual void InitialiserT()
```

Extrait du code de la classe Terrain

Notre classe Terrain possède 4 classes dérivées dont TerrainArgileux, TerrainForestier, TerrainLimoneux et TerrainSableux :

La classe TerrainArgileux hérite de la classe abstraite Terrain. Elle définit dans son constructeur les paramètres spécifiques à un sol argileux (nom, surface, météo, humidité, etc.) et redéfinit la méthode MiseAJour() pour adapter le comportement climatique à ce type de sol. Voici le code de la classe :

```
public class TerrainArgileux : Terrain
{
    1 reference
    public TerrainArgileux() : base("Terrain argileux", 225f, "Printemps", "Argileux", 70f, 3f, 60f, 18f, false, 15,
    {
        TypeSol = "Argileux";
    }

    2 references
    public override void MiseAJour()
    {
        base.MiseAJour();
        if (Humidite < 50)
        {
            Humidite += 5;
        }
    }
}
```

Extrait du code de la classe TerrainArgileux

2. Choix des plantes

La classe abstraite Plante est la classe mère des classes dérivées fruits, légumes, fleurs et herbes, ces classes sont d'ailleurs, elles même des classes mères. Par exemple, la classe fruit a pour classes dérivées : pastèque, fraise et pomme.

```
public Plante(string nom, string meteoPref, TypePlante type, string typeSolNecessaire, float humiditeNecessaire,
{
    Nom = nom;
    MeteoPref = meteoPref;
    Type = type;
    TypeSolNecessaire = typeSolNecessaire;
    HumiditeNecessaire = humiditeNecessaire;
    TemperatureNecessaire = temperatureNecessaire;
    LuminositeNecessaire = luminositeNecessaire;
    EspaceNecessaire = espaceNecessaire;
    Croissance = croissance;
    Sante = EtatSante.EnBonneSante;
    EsperanceDeVie = esperanceDeVie;
}

1 reference
public float CroissanceSelonConditions(Terrain terrain)
{
    int nbConditionsTotal = 6;
    int conditionsOk = 0;
    if (terrain.Meteo == MeteoPref)
    {
        conditionsOk++;
    }
    if (terrain.TypeSol == TypeSolNecessaire)
    {
        conditionsOk++;
```

Extrait du code de la classe Plante

Cette classe constitue la base de toutes les plantes dans notre jeu de potager. Elle regroupe les propriétés essentielles telles que le nom, le type (fruit, légume, fleur, herbe), les conditions idéales de croissance (météo, sol, humidité, température, luminosité, espace), et la santé (en bonne santé, malade, morte). Chaque plante connaît également sa croissance, son espérance de vie, sa position dans le terrain, et peut interagir avec son environnement.

La classe intègre plusieurs méthodes de comportement comme MettreAJourCroissance(), ArroserPlantes(), Contaminer(), Soigner() ou encore ExposerAuSoleil(). Elle est aussi composée d'une méthode abstraite AtteindreEtatFinal() que chaque plante dérivée devra définir pour spécifier l'emoji adapté à la plante lors de sa maturité.

Exemple de plante : Fruit

La classe Fruit est une classe abstraite dérivée de la classe Plante. Elle représente le modèle générique de tous les fruits cultivables dans le jeu. Bien qu'elle ne définisse pas de comportement spécifique supplémentaire, elle hérite de tous les attributs et fonctionnalités de Plante, comme les préférences de météo, de sol, ou encore les conditions de croissance. Elle sert de classe mère pour des fruits concrets comme Fraise, Pomme, ou Pastèque, en centralisant leurs caractéristiques communes.

```
public abstract class Fruit: Plante
{
    3 references
    public Fruit(string nom, string meteoPref, TypePlante type, string typeSolNecessaire,
                 float humiditeNecessaire, float temperatureNecessaire, int luminositeNecessaire,
                 int espaceNecessaire, int esperanceDeVie, float croissance = 0)
        : base(nom, meteoPref, type, typeSolNecessaire, humiditeNecessaire, temperatureNecessaire,
               luminositeNecessaire, espaceNecessaire, esperanceDeVie, croissance) {}
}
```

Extrait du code de la classe Fruit

```
2 references
public class Pomme : Fruit
{
    1 reference
    public Pomme() : base("Pomme", "Tempéré", TypePlante.Fruit, "Limoneux", 35f, 23f, 6, 500, 10) {}

    6 references
    public override void AtteindreEtatFinal()
    {
        Terrain.T[PositionX, PositionY] = 5;
        Console.WriteLine($"{Nom} a atteint son état final !");
    }
}
```

Extrait du code de la classe Pomme

La classe Pomme est une classe dérivée de Fruit, elle-même dérivée de la classe Plante. Elle définit les caractéristiques propres à ce fruit dans le base() à l'aide du constructeur de la classe parente Fruit. De plus, elle redéfinit AtteindreEtatFinal(), héritée de la classe mère, afin de modifier l'emoji de la plante lorsqu'elle apparaît sur le terrain dans son état final.

3. Système d'obstacles

Classe abstraite Obstacle :

```
public abstract class Obstacle

    4 references
    public Obstacle(string nom, string type, string effet, int dureeTours, bool affecteTerrain = false)
    {
        Nom = nom;
        Type = type;
        Effet = effet;
        DureeTours = dureeTours;
        AffecteTerrain = affecteTerrain;
    }

    0 references
    public void DefinirUnivers(Terrain terrain)
    {
        Univers = terrain;
    }

    5 references
    public abstract void Agir();

    0 references
    public override string ToString()
    {
        return $"{Nom} ({Type}) - Effet : {Effet}, Durée : {DureeTours} tour(s)";
    }
}
```

Extrait du code de la classe Obstacle

La classe Obstacle constitue une structure abstraite servant de base à l'introduction d'éléments perturbateurs dans notre jeu, apportant du dynamisme et de l'imprévu au sein du potager, notamment pour notre dragon qui intervient de manière aléatoire ce qui déclenche le mode urgence de notre jeu. De plus, elle définit des propriétés communes à tous les obstacles, tels que leur nom, leur type (animal, maladie, intempérie, etc.), la durée de leur effet, la nature de cet effet, ainsi que leur impact éventuel sur le terrain. La méthode abstraite Agir() oblige chaque classe dérivée à définir son propre comportement, ce qui permet une grande variété d'interactions avec le terrain. Les classes comme Taupe, PasDeGeant, Dragon ou encore les maladies sont des implémentations concrètes d'Obstacle : elles incarnent des événements spécifiques capables de modifier l'état du terrain, de perturber les plantes ou de créer des situations critiques lors d'une partie.

Exemple d'obstacle destructeur : PasDeGeant

```
public class PasDeGeant : Obstacle
{
    0 references
    public PasDeGeant() : base("Pas de géant", "destructeur", "Ecrase toutes les plantations", 1)
    { }

    1 reference
    public override void Agir()
    {

        Random random = new Random();
        int PositionX = random.Next(0, Univers.Lignes - 1);
        int PositionY = random.Next(0, Univers.Colonnes - 1);
        Univers.T[PositionX, PositionY] = 6;
        Univers.T[PositionX + 1, PositionY] = 6;
        Univers.T[PositionX, PositionY + 1] = 6;
        Univers.T[PositionX - 1, PositionY] = 6;
        Univers.T[PositionX, PositionY - 1] = 6;
        Univers.T[PositionX + 1, PositionY + 1] = 6;
        Univers.T[PositionX - 1, PositionY - 1] = 6;
        Univers.T[PositionX + 1, PositionY - 1] = 6;
        Univers.T[PositionX - 1, PositionY + 1] = 6;
    }
}
```

Extrait du code de la classe PasDeGeant

La classe PasDeGeant est un exemple de classe dérivée de notre classe abstraite Obstacle. Elle hérite de toutes les propriétés de base des obstacles (nom, type, effet, durée, etc.), tout en redéfinissant la méthode Agir() pour simuler l'impact d'un géant marchant sur le potager. Lorsqu'il est activé, cet obstacle choisit une position aléatoire sur le terrain, puis écrase les plantations alentour en modifiant plusieurs cases autour de ce point (notées par le code 6 dans la matrice).

4. Système de bonnes fées

```
public abstract class BonneFee
{
    5 references
    public string Nom { get; set; }
    2 references
    public string Effet { get; set; }
    2 references
    public int DureeTours { get; set; }
    1 reference
    protected static Random rnd = new Random();

    4 references
    public BonneFee(string nom, string effet, int dureeTours)
    {
        Nom = nom;
        Effet = effet;
        DureeTours = dureeTours;
    }


    // À définir dans les classes dérivées (ex: Maladie, Intemperie, etc.)
```

Extrait du code de la classe BonneFee

La classe BonneFee entre en action juste après les obstacles afin d'essayer de contrer leurs dégâts sur le potager. Elle définit des propriétés communes à toutes les bonnes fées, tels que leur nom, leur effet et la durée de leur effet (attribut sans réel impact). La méthode abstraite AiderPotager() oblige chaque classe dérivée à définir son propre comportement, ce qui permet une grande variété d'interactions avec le terrain. Comme exemples, l'elfe plante aléatoirement des fleurs directement à

leur état final, la brebis mange 2 mauvaises herbes, la nymphe guérit toutes les plantes sur 5 lignes aléatoires et la poussière de fée fait passer toutes les plantes à leur état final sur 6 lignes aléatoires.

Exemple de Bonne fée : Poussière de fée

```
public class PoussiereDeFee : BonneFee
{
    4 references
    private Terrain terrain;
    1 reference
    private static Random ok = new Random();

    0 references
    public PoussiereDeFee(Terrain terrain)
        : base("Poussière de Fée", "Fait passer toutes les jeunes pousses en état final sur 6 lignes d'un terrain", 1
    {
        this.terrain = terrain;
    }

    1 reference
    public override void AiderPotager()
    {
        int nbLignes = terrain.Lignes;
        int nbColonnes = terrain.Colonnes;

        int lignesAAgir = Math.Min(6, nbLignes);

        // Choisir un indice de départ aléatoire pour 6 lignes consécutives
        int startIndex = ok.Next(0, nbLignes - lignesAAgir + 1);

        // Parcourir toutes les plantes cultivées
        foreach (var plante in terrain.PlantesCultivees)
        {
```

Extrait du code de la classe PoussiereDeFee

V- Tests réalisés

Afin de garantir la fiabilité de notre simulateur de potager, nous avons mis en œuvre différents tests couvrant les différents aspects du simulateur. Ces tests ont été réalisés au fur et à mesure de notre projet dans la partie Program.cs.

- **Tests de sélection de terrain** : Validation du choix utilisateur et initialisation correcte d'un terrain choisi parmi 4 types de terrains
- **Tests de plantation et autres méthodes** : Vérification du placement aléatoire des plantes via l'apparition des emojis associés sur le terrain ainsi que des différents effets des obstacles ou des bonnes fées sur ces plantes
- **Tests du mode urgence** : Simulation d'une attaque de dragon et validation des deux possibilités de sauvetage du terrain par le joueur
- **Synchronisation modes de jeu** : Test du bon fonctionnement du mode classique et son alternance avec le mode urgence

VI- Gestion de projet

A- Organisation de l'équipe

Nous sommes une équipe de trois étudiantes n'ayant pas pratiqué la programmation avant d'entrer à l'ENSC.

- Abigaëlle Grondin: classe préparatoire des INP spécialité biologie
- Eloïse Chauvel: ancienne élève de classe préparatoire B/L
- Gwenaëlle Lebreton: ancienne élève de classe préparatoire B/L

Nous avons appris à programmer dès septembre grâce aux CM et aux TD de programmation de C#.

Nous avons donc décidé de constituer une équipe de trois afin que l'on puisse s'entraider et se compléter en fonction des compétences que nous avons acquises depuis la rentrée. Après avoir réussi à prendre en main Github, nous avons décidé de nous répartir les tâches de programmation: Abigaëlle pour les plantes, Gwenaëlle pour le terrain et les bonnes fées, et enfin Eloïse pour les obstacles, les modes et la simulation. Le but était que chacune puisse réussir à programmer sans dépendre du code des autres, en se décidant de réunir nos parties de code après les avoir terminé chacune sur nos propres machines. Au fur et à mesure de l'avancée, nous avons cependant plus ou moins changé nos rôles. Aujourd'hui, la structure actuelle et fonctionnelle de notre terrain est majoritairement celle d'Eloïse alors qu'Abigaëlle et Gwenaëlle se sont plus penchées sur l'arrangement final des modes et de la simulation.

Nous avons utilisé tous les cours de TD pour avancer et réussir à terminer notre projet. Ces cours nous ont permis de pouvoir programmer à proximité pour pouvoir échanger concernant nos idées, nos attentes, nos parties de code, nos questionnements, etc.

Nous avons commencé à programmer individuellement puisque les parties que nous nous sommes attribuées ne nécessitaient pas de travailler à plusieurs sur une unique partie de code. De plus, la conception de différentes classes a rendu le projet plus facilement collaboratif qu'au premier semestre.

B- Planning de réalisation

Notre développement s'est déroulé sur 6 semaines selon les phases suivantes :

Phases	Objectifs	Livrable
1	Conception globale et architecture Idéation du projet	Diagramme UML, structure du projet
2	Implémentation des classes de base	Classes abstraites et premières classes concrètes
3	Développement du mode classique	Système de tour par tour fonctionnel
4	Implémentation du mode urgence	Mécanique de gestion des événements
5	Intégration des modes et amélioration	Version jouable complète
6	Tests finaux et corrections	Application stable et documentation

VII- Bilan critique

Pour conclure, nous nous sommes suffisamment bien organisées afin de pouvoir avancer la programmation du jeu de simulation lors des TD mais aussi individuellement chez nous. Nous avons réussi à mettre en place les différentes classes principales afin de présenter un simulateur de potager proche de celui demandé.

Cependant, le sujet présentait de nombreuses consignes et propositions qui nous ont un peu éparpillées au moment de commencer le code. Nous avons été très ambitieuses au début en créant beaucoup de classes avec surtout beaucoup d'attributs et de constructeurs, avant de vérifier que tout fonctionne vraiment correctement. Nous avons alors ensuite repris toutes les classes que nous avions faites en les testant au fur et à mesure sur une autre branche. C'est pourquoi notre programme est aujourd'hui allégé, plus simple qu'imaginé au départ mais majoritairement fonctionnel. Avec plus de temps, nous aurions fait en sorte que l'ensemble des fonctions évoquées aient un réel impact sur une partie composé de nombreux tours et nous aurions complexifié le mode de jeu général. Nous aurions alors ajouté plus de méthodes telles que récolter et la mise en place d'un magasin. Par ailleurs, nous regrettons de ne pas avoir commenté le code. Nous aurions certainement dû le faire dès le début, mais nous avons rapidement été rattrapées par le manque de temps.

Ce projet nous a finalement permis de monter en compétences en programmation C#, notamment sur la partie programmation orientée objet, tout en prenant du plaisir par la partie ludique de la création d'un jeu et de son imaginaire.