Programmation avancée

ENSemenC

Projet PROGAV 2025

Nickerson Charlotte

Pawelczyk Baptiste

Table des matières

1. Introduction

Ce document constitue le rapport technique du projet de développement d'un simulateur de potager, réalisé dans le cadre du module de Programmation Avancée. L'objectif principal était de concevoir et d'implémenter une application console en C# simulant la gestion d'un jardin potager, en mettant l'accent sur une modélisation objet robuste et l'application des bonnes pratiques de développement.

Le jeu plonge le joueur dans la gestion quotidienne d'un potager. Avant de commencer, le joueur choisit un pays (réel ou imaginaire), ce qui influence directement les types de semis disponibles et les conditions météorologiques rencontrées. Initialement, le joueur dispose d'un ensemble de semis et de parcelles de terrain aux caractéristiques variées. Une particularité du jeu est la présence d'une “webcam” virtuelle qui surveille le jardin en permanence, alertant le joueur en cas d'événements critiques tels que des intempéries sévères ou l'intrusion d'animaux.

Ce rapport détaillera l'univers du jeu, les mécaniques de gameplay, la conception objet sous-jacente, les fonctionnalités implémentées (minimales et bonus), ainsi qu'une réflexion sur la gestion du projet et les aspects techniques.

2. L'univers du jeu et ses spécificités

Le cœur du jeu réside dans la simulation d'un écosystème de potager. Le choix initial du pays est une mécanique centrale, car il détermine le catalogue de plantes cultivables et le profil climatique général (géré par la classe Pays). Par exemple, un pays nordique n'offrira pas les mêmes possibilités qu'un pays tropical.

```
// Extrait de Pays.cs (Illustratif)
```

```
public class Pays
{
    public string Nom { get; private set; }
    public List<string> SemisDisponibles { get; private set; } // Noms des plantes
    public TypeClimat Climat { get; private set; } // Enum: Tropical, Tempere, Aride, etc.
    public Pays(string nom, List<string> semis, TypeClimat climat)
    {
        Nom = nom;
        SemisDisponibles = semis;
        Climat = climat;
    }
}
```

Le joueur commence avec un inventaire de base de semis et quelques parcelles de terrain. Chaque terrain possède des qualités intrinsèques (type de sol, fertilité initiale) qui affectent la croissance des plantes.

La surveillance par webcam (simulée par la classe Webcam) ajoute un élément dynamique, pouvant déclencher des alertes et potentiellement un passage en “Mode Urgence”.

```
// Extrait de Webcam.cs (Illustratif)
public class Webcam
{
    public event EventHandler<AlerteEventArgs> AlerteDetectee;
    public void VerifierJardin(ConditionsEnvironnementales conditions, List<Parcelle> parcelles)
```

```
    {
        // Logique pour détecter une forte tempête
        if (conditions.EstTempeteViolente())
        {
            OnAlerteDetectee(new AlerteEventArgs("Forte tempête détectée !"));
        }
        // Logique pour détecter un intrus
        // ...
    }
    protected virtual void OnAlerteDetectee(AlerteEventArgs e)
    {
        AlerteDetectee?.Invoke(this, e);
    }
}
public class AlerteEventArgs : EventArgs
{
    public string Message { get; }
    public AlerteEventArgs(string message) { Message = message; }
}
```

3. Déroulement d'une partie et possibilités des joueurs

Le jeu se distingue par une double temporalité, offrant deux modes de jeu principaux qui s'alternent : le "Mode Classique" pour la gestion à long terme et le "Mode Urgence" pour les situations critiques.

Mode Classique

C'est le mode principal du jeu, où le temps s'écoule par plusieurs tours représentant des semaines ou des mois. La classe Temps est centrale à cette mécanique.

```
// Extrait de Temps.cs
public class Temps
{
    private int semaineActuelle;
    private Saison saison;
    // ... autres dépendances (Temperature, Precipitations, Affichage)
    public Temps(Affichage affichage, Saison saison, Temperature temperature, Precipitations precipitations)
    {
        this.semaineActuelle = 1;
        this.affichage = affichage;
        this.saison = saison;
        this.temperature = temperature;
        this.precipitations = precipitations;
    }
    public void PasserSemaine()
    {
        semaineActuelle++;
        saison.AvancerSaison(semaineActuelle);
        temperature.MettreAJourTemperature(saison.GetSaisonActuelle());
        precipitations.MettreAJourPrecipitations(saison.GetSaisonActuelle());
        // Autres logiques de mise à jour hebdomadaire
```

```
        Console.WriteLine($"Semaine {semaineActuelle} - Saison : {saison.GetNomSaison()}");
    }
    // ...
}
```

À chaque tour, le joueur peut réaliser diverses actions :

- **gestion des cultures** : semer, arroser, désherber, traiter contre les maladies, récolter ;
- **amélioration du terrain** : pailler, fertiliser ;
- **installation d'équipements** : serres, barrières, pare-soleil.

La simulation météorologique, gérée par les classes Temperature et Precipitations (et consolidée dans ConditionsEnvironnementales), joue un rôle crucial. Elle inclut les taux de précipitations (risque de sécheresse ou d'inondation) et les variations de température (risque de gel ou de canicule).

```
// Extrait de ConditionsEnvironnementales.cs (Illustratif)
public class ConditionsEnvironnementales
{
    public Temperature TemperatureActuelle { get; private set; }
    public Precipitations NiveauPrecipitation { get; private set; }
    public int Luminosite { get; private set; } // Pourcentage
    public ConditionsEnvironnementales(Temperature temp, Precipitations precip, int luminosite)
    {
        TemperatureActuelle = temp;
        NiveauPrecipitation = precip;
        Luminosite = luminosite;
```

```
    }
    public bool ConditionsOptimalesPourPlante(Plante plante)
    {
        // Logique complexe comparant les besoins de la plante aux conditions actuelles
        // Retourne true si au moins 50% des conditions sont respectées
        int conditionsRespectees = 0;
        int conditionsTotal = 4; // Exemple: temp, eau, lumière, type de sol (via parcelle)
        if (plante.EstTemperatureIdeale(TemperatureActuelle.Valeur)) conditionsRespectees++;
        // ... autres vérifications ...
        return (double)conditionsRespectees / conditionsTotal >= 0.5;
    }
}
```

Des “obstacles” (maladies, nuisibles comme les pucerons, rongeurs) et des “bonnes fées” (coccinelles qui mangent les pucerons) sont introduits aléatoirement, impactant la santé des plantes. La classe Obstacle peut modéliser ces éléments.

```
// Extrait de Obstacle.cs (Conceptuel)
public abstract class Obstacle
{
    public string Nom { get; protected set; }
    public abstract void AppliquerEffet(Parcelle parcelle);
}
public class Puceron : Obstacle
{
```

```
    public Puceron() { Nom = "Attaque de pucerons"; }
    public override void AppliquerEffet(Parcelle parcelle)
    {
        if (parcelle.EstPlantee)
        {
            // Logique pour réduire la santé de la plante
            // parcelle.PlanteEnCours.ReduireSante(10);
            Console.WriteLine($"{Nom} sur la parcelle ({parcelle.X},{parcelle.Y}) !");
        }
    }
}
```

L'affichage (géré par Affichage.cs) résume l'état du potager : plantes en place, santé, hauteur (avec des représentations graphiques en console), et propose un menu d'actions. Le tour se termine lorsque le joueur le décide ou après un nombre limité d'actions.

Mode Urgence

Ce mode s'active aléatoirement (via la Webcam ou des événements météo extrêmes) et passe le jeu en temps réel. Il est géré par la classe ModeUrgence.

```
// Extrait de ModeUrgence.cs (Illustratif)
public class ModeUrgence
{
    public bool EstActif { get; private set; }
    private string typeUrgence; // "Intemperie", "Intrus"
    public void DeclencherUrgence(string type, string message)
```

```
    {
        EstActif = true;
        typeUrgence = type;
        Console.WriteLine($"\n--- MODE URGENCE ACTIF : {message} ---");
        // Logique spécifique au mode urgence (boucle de jeu temps réel simplifiée)
    }
    public void ResoudreUrgence()
    {
        EstActif = false;
        Console.WriteLine("--- Mode Urgence terminé ---");
    }
    public void AfficherActionsUrgence()
    {
        Console.WriteLine("Actions d'urgence disponibles :");
        Console.WriteLine("1. Déployer une bâche");
        Console.WriteLine("2. Faire du bruit (contre intrus)");
        // ...
    }
}
```

Les tours simulent des actions rapides pour contrer une menace immédiate (exemple : un rongeur mangeant les récoltes, une averse de grêle). Le joueur choisit parmi des actions d'urgence spécifiques : faire du bruit, déployer une bâche, fermer une serre, etc. Il est important de noter l'interdiction de tuer les animaux.

4. Modélisation objet réalisée

Le projet est développé en C# avec le framework .NET 8, en respectant les principes de la Programmation Orientée Objet (POO).

Diagramme UML

Un diagramme UML complet a été réalisé pour notre projet afin de visualiser les relations entre les différentes entités du jeu. Ce diagramme a été généré à l'aide de PlantUML.

Classes abstraites et héritage

Deux hiérarchies de classes principales structurent le modèle : les plantes et les terrains.

Classe Plante

La classe Plante est une classe abstraite qui définit les caractéristiques et comportements communs à toutes les plantes du jeu.

```
// Extrait de Plantes/Plante.cs
public abstract class Plante
{
    public string Nom { get; protected set; }
    public bool EstVivace { get; protected set; } // true si vivace, false si annuelle
    public bool EstComestible { get; protected set; }
    public List<Saison.TypeSaison> SaisonsSemis { get; protected set; }
    public TypeTerrain TerrainPrefere { get; protected set; } // Enum: Sable, Terre, Argile
    public int EspacementRequis { get; protected set; } // en cm
    public int EspaceVital { get; protected set; } // surface en cm²
```

```
public double VitesseCroissance { get; protected set; } // unité par semaine
public int BesoinEau { get; protected set; } // mm par semaine
public int BesoinLuminosite { get; protected set; } // %
public Tuple<int, int> TemperatureIdeale { get; protected set; } // Min, Max en °C
public List<string> MaladiesPossibles { get; protected set; } // Noms des maladies
public double ProbaMaladie { get; protected set; } // 0.0 - 1.0
public int EsperanceDeVie { get; protected set; } // en semaines
public int RendementMax { get; protected set; } // nombre de fruits/légumes
public int Sante { get; set; } = 100;
public double Hauteur { get; set; } = 0;
public int Age { get; set; } = 0; // en semaines
public Plante(string nom)
{
    this.Nom = nom;
    this.SaisonsSemis = new List<Saison.TypeSaison>();
    this.MaladiesPossibles = new List<string>();
}
public abstract void Croitre(ConditionsEnvironnementales conditions);
public abstract string GetRepresentationConsole(); // Pour l'affichage
public bool DoitMourir(ConditionsEnvironnementales conditions)
{
    // Exemple simplifié: si moins de 50% des besoins sont satisfaits
    // La logique réelle serait dans ConditionsEnvironnementales.ConditionsOptimalesPourPlante
    // ou une méthode dédiée ici.
```

```
        int conditionsSatisfaites = 0;

        if (this.TemperatureIdeale.Item1 <= conditions.TemperatureActuelle.Valeur &&
this.TemperatureIdeale.Item2 >= conditions.TemperatureActuelle.Valeur)

            conditionsSatisfaites++;

        // ... autres conditions (eau, lumière)

        return conditionsSatisfaites < 2; // Supposons 3 conditions majeures

    }

}
```

Les plantes sont caractérisées par :

- **leur nature** : annuelle ou vivace, comestible ou non, mauvaise herbe, etc ;
- **leurs conditions de culture** : saison(s) de semis, terrain préféré, espacement, besoins en eau et luminosité, températures préférées ;
- **leur croissance et leur santé** : vitesse de croissance, espace vital (attention aux espèces envahissantes), maladies potentielles (avec probabilité aléatoire), espérance de vie ;
- **leur rendement** : nombre de pousses/fruits/légumes.

Une plante meurt si ses conditions préférées ne sont pas respectées à au moins 50%. Plus les conditions sont optimales, plus la croissance est rapide et le rendement élevé. Le jeu permet de cultiver des plantes comestibles, commerciales (coton, bambou) ou ornementales.

Exemple de plante dérivée (Tomate.cs) :

```
// Extrait de Plantes/PlantesComestibles/Tomate.cs

public class Tomate : PlanteComestible // Supposant une classe intermédiaire PlanteComestible

{
```

```
public Tomate() : base("Tomate")
{
    EstVivace = false;
    SaisonsSemis.Add(Saison.TypeSaison.Printemps);
    SaisonsSemis.Add(Saison.TypeSaison.Ete);
    TerrainPrefere = TypeTerrain.TerreLimoneuse; // Supposons un enum TypeTerrain
    EspacementRequis = 50;
    EspaceVital = 2500;
    VitesseCroissance = 2.5; // cm par semaine dans de bonnes conditions
    BesoinEau = 30; // mm
    BesoinLuminosite = 80; // %
    TemperatureIdeale = new Tuple<int, int>(18, 28); // °C
    MaladiesPossibles.Add("Mildiou");
    ProbaMaladie = 0.15;
    EsperanceDeVie = 20; // semaines
    RendementMax = 15; // tomates par plant
    // Initialisation spécifique à PlanteComestible
    ValeurNutritionnelle = 7; // Sur 10
    TempsDeMaturation = 8; // Semaines
}
public override void Croitre(ConditionsEnvironnementales conditions)
{
    // Logique de croissance spécifique à la tomate
    // Prend en compte les conditions pour ajuster la croissance réelle
```

```
        if (!DoitMourir(conditions))
        {
            double facteurCroissance = CalculerFacteurCroissance(conditions);
            this.Hauteur += this.VitesseCroissance * facteurCroissance;
            this.Age++;
            // ... autre logique (fructification, etc.)
        }
        else
        {
            this.Sante = 0;
            Console.WriteLine($"La {Nom} est morte faute de conditions adéquates.");
        }
    }
    private double CalculerFacteurCroissance(ConditionsEnvironnementales conditions)
    {
        // Simule comment les conditions affectent la croissance.
        // Retourne un multiplicateur (ex: 1.0 pour optimal, <1.0 si moins bon)
        double facteur = 1.0;
        if (conditions.TemperatureActuelle.Valeur < TemperatureIdeale.Item1 ||
conditions.TemperatureActuelle.Valeur > TemperatureIdeale.Item2)
            facteur *= 0.7;
        // ... ajustements pour eau, lumière ...
        return facteur;
    }
    public override string GetRepresentationConsole()
```

```
    {
        if (Sante <= 0) return "[X]"; // Morte
        if (Hauteur < 10) return "[t]"; // Jeune pousse
        if (Hauteur < 30) return "[T]"; // En croissance
        return "[🍅]"; // Mature avec fruits
    }
}
```

D'autres catégories de plantes comme PlanteAromatique, PlanteCommerciale, PlanteOrnementale suivent une structure similaire, héritant de Plante et spécialisant leurs caractéristiques.

Classe Terrain

La classe Terrain est également abstraite et définit les propriétés communes à tous les types de terrains.

```
// Extrait de Terrains/Terrain.cs
public abstract class Terrain
{
    public string Type { get; protected set; }
    public TypeTerrain TypeDeSol { get; protected set; } // Sable, Argile, Limon, etc.
    public double Fertilite { get; set; } // 0.0 - 1.0
    public double HumiditeMax { get; protected set; } // Capacité de rétention d'eau
    public double Drainage { get; protected set; } // Efficacité du drainage
    public Terrain(string type, TypeTerrain typeDeSol, double fertiliteInitiale, double humiditeMax,
double drainage)
    {
```

```
        Type = type;
        TypeDeSol = typeDeSol;
        Fertilite = fertiliteInitiale;
        HumiditeMax = humiditeMax;
        Drainage = drainage;
    }
    public abstract string GetDescription();
    // Potentiellement des méthodes pour interagir avec les conditions météo
    public virtual void AppliquerPrecipitations(double quantitePluie)
    {
        // Logique de base pour l'absorption d'eau
        // Sera surchargée par les types de terrains spécifiques
    }
}
```

Les classes dérivées (Plaine, Montagne, Desert, etc.) spécifient ces propriétés.

Exemple de terrain dérivé (Plaine.cs) :

```
// Extrait de Terrains/Types/Plaine.cs
public class Plaine : Terrain
{
    public Plaine() : base("Plaine", TypeTerrain.TerreLimoneuse, 0.7, 0.6, 0.5)
    {
        // Valeurs spécifiques pour une plaine
```

```
    }

    public override string GetDescription()

    {

        return $"Une plaine fertile avec un sol limoneux. Bonne rétention d'eau et drainage modéré.
Fertilité: {Fertilite:P0}.";

    }

    public override void AppliquerPrecipitations(double quantitePluie)

    {

        // Logique spécifique à la plaine pour gérer la pluie

        // this.HumiditeActuelle = Math.Min(this.HumiditeMax, this.HumiditeActuelle + quantitePluie
* (1 - Drainage));

        Console.WriteLine($"La plaine absorbe la pluie. Humidité actuelle : {/* ... */}%.");

    }

}
```

Autres classes clés et associations

- **Parcelle.cs** : représente une case du jardin. Elle est associée à un type de Terrain et peut contenir une Plante ;

```
// Extrait de Parcelle.cs

public class Parcelle

{

    public Terrain TypeDeTerrain { get; private set; }

    public Plante? PlanteEnCours { get; private set; }

    public bool EstPlantee => PlanteEnCours != null;

    public int X { get; } // Coordonnée
```

```
public int Y { get; } // Coordonnée
public double Humidite { get; set; } // Niveau actuel
public double Fertilite { get; set; } // Niveau actuel
public Parcelle(int x, int y, Terrain terrain)
{
    X = x;
    Y = y;
    TypeDeTerrain = terrain;
    Humidite = terrain.HumiditeMax * 0.5; // Humidité initiale
    Fertilite = terrain.Fertilite; // Fertilité initiale
}
public bool Planter(Plante nouvellePlante)
{
    if (!EstPlantee)
    {
        PlanteEnCours = nouvellePlante;
        Console.WriteLine($"{nouvellePlante.Nom} planté(e) en ({X},{Y}).");
        return true;
    }
    Console.WriteLine($"La parcelle ({X},{Y}) est déjà occupée.");
    return false;
}
public void Recolter()
{
```

```
    if (EstPlantee && PlanteEnCours.EstComestible /* et mûre */)
    {
        Console.WriteLine($"Récolte de {PlanteEnCours.Nom} en ({X},{Y}).");
        // Logique de rendement, ajout à l'inventaire
        if (!PlanteEnCours.EstVivace)
        {
            PlanteEnCours = null; // La plante disparaît après récolte si annuelle
        }
    }
}
public void FaireCroitre()
{
    if (EstPlantee && PlanteEnCours.Sante > 0)
    {
        // Simuler des conditions environnementales pour cette parcelle
        // Ceci serait idéalement passé en argument ou récupéré d'un gestionnaire global
        var conditionsLocales = new ConditionsEnvironnementales(
            new Temperature(Saison.TypeSaison.Printemps, 20), // Exemple
            new Precipitations(Saison.TypeSaison.Printemps, 15), // Exemple
            80 // Exemple luminosité
        );
        PlanteEnCours.Croitre(conditionsLocales);
    }
}
```

```
}
```

- **Saison.cs** : gère le cycle des saisons et leur impact ;

```
// Extrait de Saison.cs
public class Saison
{
    public enum TypeSaison { Printemps, Ete, Automne, Hiver }
    public TypeSaison SaisonActuelle { get; private set; }
    public Saison()
    {
        SaisonActuelle = TypeSaison.Printemps; // Saison de départ
    }
    public void AvancerSaison(int semaineActuelle)
    {
        // Typiquement 13 semaines par saison
        int saisonIndex = ((semaineActuelle - 1) / 13) % 4;
        SaisonActuelle = (TypeSaison)saisonIndex;
    }
    public string GetNomSaison() => SaisonActuelle.ToString();
}
```

- **Temperature.cs** et **Precipitations.cs** : simulent les aspects météorologiques ;

```
// Extrait de Temperature.cs
public class Temperature
```

```
{
    public int Valeur { get; private set; } // en °C
    private Random random = new Random();
    public Temperature(Saison.TypeSaison saisonInitiale, int valeurInitiale = 15)
    {
        Valeur = valeurInitiale;
        MettreAJourTemperature(saisonInitiale);
    }
    public void MettreAJourTemperature(Saison.TypeSaison saisonActuelle)
    {
        // Logique pour simuler la température en fonction de la saison
        switch (saisonActuelle)
        {
            case Saison.TypeSaison.Printemps: Valeur = random.Next(10, 22); break;
            case Saison.TypeSaison.Ete: Valeur = random.Next(20, 35); break;
            // ... autres saisons
            default: Valeur = random.Next(5, 15); break;
        }
    }
    public bool EstEnGel() => Valeur <= 0;
    public bool EstEnCanicule() => Valeur >= 30; // Seuil simplifié
    public string GetTemperatureString() => $"{Valeur}°C";
}
// Extrait de Precipitations.cs (similaire pour la logique)
```

```
public class Precipitations
{
    public int Niveau { get; private set; } // en mm
    private Random random = new Random();
    public Precipitations(Saison.TypeSaison saisonInitiale, int niveauInitial = 10) { /* ... */ }
    public void MettreAJourPrecipitations(Saison.TypeSaison saisonActuelle) { /* ... */ }
    public bool EstEnSecheresse(Saison.TypeSaison saison) { /* ... */ return Niveau < 5; } // Simplifié
    public bool EstEnInondation() { /* ... */ return Niveau > 50; } // Simplifié
    public string GetPrecipitationsString(Saison.TypeSaison saison, Temperature temp) =>
$"{Niveau}mm";
}
```

- **Affichage.cs** : gère toute la sortie console, y compris le dessin du jardin ;

```
// Extrait de Affichage.cs
public class Affichage
{
    private Parcelle[,] grilleDuJardin; // Référence à la structure de données du jardin
    private int largeurGrille;
    private int hauteurGrille;
    public Affichage(Parcelle[,] grille)
    {
        grilleDuJardin = grille;
        hauteurGrille = grille.GetLength(0);
        largeurGrille = grille.GetLength(1);
```

```
    }
    public void AfficherPlateau()
    {
        Console.Clear();
        for (int i = 0; i < hauteurGrille; i++)
        {
            for (int j = 0; j < largeurGrille; j++)
            {
                Parcelle p = grilleDuJardin[i, j];
                if (p.EstPlantee)
                {
                    Console.Write(p.PlanteEnCours.GetRepresentationConsole() + " ");
                }
                else
                {
                    // Afficher type de terrain (simplifié)
                    Console.Write($"[{p.TypeDeTerrain.TypeDeSol.ToString()[0]}] ");
                }
            }
            Console.WriteLine();
        }
    }
    // ... autres méthodes pour menus, messages, etc.
}
```

- **Program.cs** : Contient le point d'entrée Main() et la boucle de jeu principale.

```
// Extrait de Program.cs
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Bienvenue dans le Simulateur de Potager !");
      // Initialisation (simplifiée)
      // Création des parcelles, de l'affichage, du temps, etc.
      int hauteur = 5, largeur = 10;
      Parcelle[,] monJardin = new Parcelle[hauteur, largeur];
      for(int i=0; i<hauteur; ++i)
         for(int j=0; j<largeur; ++j)
            monJardin[i,j] = new Parcelle(i, j, new Plaine()); // Exemple avec Plaine partout
      Affichage affichage = new Affichage(monJardin);
      Saison saison = new Saison();
      Temperature temperature = new Temperature(saison.GetSaisonActuelle());
      Precipitations precipitations = new Precipitations(saison.GetSaisonActuelle());
      // La classe Temps a été modifiée pour prendre ces dépendances dans son constructeur
      // Temps gestionTemps = new Temps(affichage, saison, temperature, precipitations);
      // gestionTemps.DemarrerJeu(); // Lance la boucle de jeu principale
      // Pour l'exemple, on va juste simuler quelques tours
      for (int tour = 0; tour < 5; tour++)
      {
```

```
        // gestionTemps.PasserSemaine();
        // Mettre à jour l'état des plantes dans chaque parcelle
        foreach (Parcelle p in monJardin)
        {
            if (p.EstPlantee)
            {
                // p.FaireCroitre(); // Appel à la méthode de Parcelle
            }
        }
        affichage.AfficherPlateau();
        // Afficher infos (semaine, saison, météo)
        // Demander action au joueur
        Console.WriteLine("Appuyez sur Entrée pour la semaine suivante...");
        Console.ReadLine();
    }
    Console.WriteLine("Fin de la simulation.");
  }
}
```

Principes de la POO mis en œuvre

- **Encapsulation** : les données des classes sont généralement privées ou protégées, accessibles via des propriétés publiques (getters/setters) ou des méthodes, contrôlant ainsi leur modification. Par exemple, la Sante d'une Plante est modifiée via des méthodes qui peuvent inclure une logique de validation ;

- **Héritage** : utilisé massivement avec les classes Plante et Terrain qui servent de bases à des types plus spécifiques. Cela permet de partager du code commun et de définir des contrats (méthodes abstraites) ;
- **Polymorphisme** : la méthode Croitre() de Plante est abstraite et chaque type de plante l'implémente différemment. Ainsi, une collection de Plante peut être parcourue, et appeler Croitre() sur chaque élément exécutera la version spécifique à cette plante. De même pour GetRepresentationConsole() ;
- **Abstraction** : les classes Plante et Terrain masquent la complexité de leurs sous-classes, offrant une interface simplifiée pour interagir avec elles ;
- **Associations entre classes** :
    - Une Parcelle *a un* Terrain (composition/agrégation) ;
    - Une Parcelle *peut avoir une* Plante (agrégation) ;
    - Le Temps *utilise* Saison, Temperature, Precipitations ;
    - L'Affichage *utilise* les Parcelles pour afficher le jardin.

5. Fonctionnalités minimales attendues

Toutes les fonctionnalités minimales décrites dans le cahier des charges ont été abordées :

- Classes abstraites **Plante et Terrain** : implémentées comme bases pour la modélisation ;
- **Caractéristiques des semis/plants** : détaillées dans la classe Plante et ses dérivées (nature, saison, terrain préféré, espacement, croissance, besoins, maladies, espérance de vie, rendement) ;
- **Condition de survie (50%)** : une plante meurt si moins de 50% de ses conditions préférées sont respectées. La croissance et le rendement sont proportionnels au respect de ces conditions. Ceci est géré via des méthodes comme DoitMourir dans Plante et la logique dans Croitre ;
- **Diversité des plantes** : comestibles, commerciales, ornementales, toutes héritant de Plante ;

- **Double temporalité** :
  - **Mode Classique** : géré par Temps.cs, simule le passage des semaines. Le joueur effectue des actions (semer, arroser, etc.). La météo (Temperature, Precipitations) et les obstacles (Obstacle) sont simulés. L'affichage (Affichage) résume l'état ;
  - **Mode Urgence** : géré par ModeUrgence.cs, se déclenche aléatoirement pour des événements critiques (intempéries, intrus via Webcam). Le joueur a des actions spécifiques.
  - **Affichage en console** : la classe Affichage gère la représentation du jardin, y compris des “dessins” simplifiés des plantes.

6. Fonctionnalités bonus

Plusieurs fonctionnalités bonus ont été conceptualisées et/ou implémentées.

Mode magasin

Géré par ModeMagasin.cs. Permet au joueur de vendre ses récoltes et d'acheter des améliorations.

```
// Extrait de ModeMagasin.cs (Illustratif)
public class ModeMagasin
{
    public double ArgentJoueur { get; private set; } = 100.0; // Argent initial
    public void VendreRecolte(Plante planteRecoltee, int quantite)
    {
        // Logique pour déterminer le prix de vente
        double prixUnitaire = DeterminerPrix(planteRecoltee);
```

```
        ArgentJoueur += prixUnitaire * quantite;

        Console.WriteLine($"{quantite} {planteRecoltee.Nom} vendu(s) pour {prixUnitaire * quantite}
unités monétaires.");

    }

    public bool AcheterSemis(string nomSemis, int quantite)

    {

        // Logique pour vérifier disponibilité et prix, puis débiter ArgentJoueur

        // et ajouter à l'inventaire du joueur.

        return true; // si achat réussi

    }

    public void AfficherBilanComptable()

    {

        Console.WriteLine($"Bilan actuel : {ArgentJoueur} unités monétaires.");


    }

    private double DeterminerPrix(Plante plante) { /* ... */ return 10.0; }

}
```

Mode IA

Géré par ModeIA.cs. Fournit des recommandations au joueur.

```
// Extrait de ModeIA.cs (Illustratif)

public class ModeIA

{

    public string ObtenirRecommandation(Parcelle parcelle, ConditionsEnvironnementales
conditions)

    {
```

```
        if (parcelle.EstPlantee)
        {
            Plante p = parcelle.PlanteEnCours;
            if (p.Sante < 50) return $"Recommandation : La plante {p.Nom} en ({parcelle.X},{parcelle.Y}) est mal en point. Envisagez de la traiter ou d'améliorer ses conditions.";
            // ... autres logiques de recommandation (arrosage, récolte, etc.)
        }
        else
        {
            // Suggérer de planter si la saison est bonne pour certaines plantes de l'inventaire
        }
        return "Aucune recommandation spécifique pour le moment.";
    }
}
```

Mode écologique

Géré par ModeEcologique.cs. Introduit des aspects d'agriculture durable.

```
// Extrait de ModeEcologique.cs (Illustratif)
public class ModeEcologique
{
    public double BilanCarbone { get; private set; } = 0; // Positif = mauvais
    public void ActionEcologique(string typeAction) // ex: "Compostage", "RotationCulture"
    {
        if (typeAction == "Compostage") BilanCarbone -= 5;
```

```
        Console.WriteLine($"Action écologique '{typeAction}' effectuée. Bilan carbone ajusté.");
    }
    public void CalculerImpactEnvironnemental()
    {
        // Logique basée sur les actions du joueur, types de plantes, etc.
        Console.WriteLine($"Impact environnemental actuel (bilan carbone) : {BilanCarbone}");
    }
}
```

Sauvegarde de la partie

Gérée par SaveManager.cs. Permet de sauvegarder et charger l'état du jeu.

```
// Extrait de SaveManager.cs (Conceptuel - la sérialisation peut être complexe)
using System.Text.Json; // Pour la sérialisation JSON
public class SaveManager
{
    private const string SaveFilePath = "savegame.json";
    public void SauvegarderPartie(object etatDuJeu /* ex: une classe 'GameState' */)
    {
        try
        {
            string jsonString = JsonSerializer.Serialize(etatDuJeu);
            File.WriteAllText(SaveFilePath, jsonString);
            Console.WriteLine("Partie sauvegardée !");
        }
```

```
    catch (Exception ex)
    {
        Console.WriteLine($"Erreur lors de la sauvegarde : {ex.Message}");
    }
}
public object ChargerPartie() // Devrait retourner GameState
{
    try
    {
        if (File.Exists(SaveFilePath))
        {
            string jsonString = File.ReadAllText(SaveFilePath);
            // return JsonSerializer.Deserialize<GameState>(jsonString);
            Console.WriteLine("Partie chargée !");
            return new object(); // Placeholder
        }
        Console.WriteLine("Aucune sauvegarde trouvée.");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Erreur lors du chargement : {ex.Message}");
    }
    return null;
}
```

```
}
```

7. Exigences techniques et bonnes pratiques

Notre projet respecte les exigences techniques spécifiées :

- **Application Console C# .NET 8** ;
- **Programmation Orientée Objet** : Encapsulation, héritage, classes abstraites, polymorphisme sont utilisés judicieusement (voir section 4) ;
- **Listes privilégiées aux tableaux** : List<T> est utilisée pour les collections dynamiques (ex: SaisonsSemis dans Plante). Les tableaux sont utilisés pour des structures fixes comme la grille du jardin ;
- **Limitation de la duplication de code** : L'héritage et la création de méthodes utilitaires contribuent à réduire la redondance ;
- **Code indenté et commenté** : Le code source s'efforce de respecter ces pratiques pour une meilleure lisibilité ;
- **Conventions de codage C#** : Les conventions de nommage (PascalCase pour classes et méthodes, camelCase pour variables locales) et autres standards C# sont suivis.

8. Tests réalisés

Bien que la mise en place d'un framework de tests unitaires formel n'ait pas été l'objectif principal de cette phase pour tous les modules, la vérification du bon fonctionnement a été effectuée par :

- **des tests manuels** : exécution répétée du jeu dans divers scénarios pour identifier les bugs et valider les mécaniques ;

- **du débogage et Console.WriteLine** : utilisation intensive des outils de débogage de Visual Studio et des affichages console pour tracer l'état des variables et le flux d'exécution.

Idéalement, des tests unitaires automatisés seraient développés pour chaque classe afin de garantir la non-régression et la fiabilité des composants individuels. Des tests d'intégration seraient également pertinents pour valider les interactions entre les modules majeurs (exemples : Temps, Plante, ConditionsEnvironnementales).

9. Gestion de projet

Notre projet a été réalisé en binôme, en utilisant GitHub comme plateforme de partage et de versionnement du code. Le dépôt Git a été créé via GitHub Classroom comme demandé. L'organisation du travail s'est faite par la répartition des tâches de développement des différentes classes et fonctionnalités, suivie de phases d'intégration. La matrice d'implication, située à la racine du dépôt, a été complétée pour refléter la contribution de chaque membre de l'équipe.

10. Bilan critique sur le projet

Ce projet de simulateur de potager a été une expérience d'apprentissage enrichissante, permettant de mettre en pratique les concepts de programmation avancée et de conception objet.

Points Forts :

- **modélisation objet** : la structure basée sur les classes abstraites Plante et Terrain offre une bonne flexibilité et extensibilité ;
- **richesse fonctionnelle** : le jeu intègre de nombreuses mécaniques (météo, saisons, types de plantes, modes de jeu) qui contribuent à une simulation engageante ;

- **respect des contraintes** : les exigences techniques et les fonctionnalités minimales du cahier des charges ont été globalement respectées.

Axes d'amélioration possibles :

- **Interface utilisateur** : une interface graphique (même simple) améliorerait grandement l'expérience utilisateur par rapport à la console pure ;
- **Équilibrage du jeu** : les paramètres de croissance, les probabilités d'événements, et l'économie du jeu nécessitent un affinage plus poussé pour une expérience de jeu optimale ;
- **Persistance des données** : l'implémentation de la sauvegarde (SaveManager) est conceptuelle ; une solution de sérialisation robuste (JSON, XML, binaire) serait nécessaire pour une sauvegarde complète et fiable ;
- **Complexité de la simulation** : certains aspects (exemples : propagation des maladies, interactions entre espèces) pourraient être approfondis pour plus de réalisme.

Difficultés rencontrées :

- la gestion de la complexité croissante du modèle objet et des interactions entre les classes ;
- l'équilibrage des multiples variables de simulation pour obtenir un comportement de jeu cohérent et intéressant ;
- la conception d'un affichage console à la fois informatif et lisible.