BOOGLE ZOA

Plus qu'un projet

SOMMAIRE

Introduction

1. Contexte du projet

Dans le cadre du module « Algorithmique et Programmation Orientée Objet », nous avons réalisé un jeu en C# sous forme d'application console. Ce projet, à effectuer en binôme, nous a permis d'apprendre à collaborer efficacement en utilisant des outils comme **GitHub**. Il constituait une opportunité de mettre en pratique toutes les notions de **C#** acquises depuis l'année dernière. Enfin, nous avons tenté d'approcher une démarche professionnelle en intégrant des **commentaires XML**, des tests unitaires et en respectant diverses conventions, que nous détaillerons par la suite.

PS : Les études exigées par le module sont annotées dans le rapport par un ***** *(voir le sommaire).*

2. Description générale du jeu

Le jeu se déroule sur un plateau carré composé de dés (par défaut 4x4). Chaque dé possède six faces, avec une lettre différente sur chacune d'elles. Au début de la partie, les dés sont lancés, et seule la lettre de leur face supérieure est visible. Ce lancement est simulé par un tirage aléatoire pour chaque dé.

Une fois le plateau généré, un compte à rebours est déclenché pour fixer la durée totale de la partie. Les joueurs jouent à tour de rôle, disposant d'un temps limité pour trouver des mots sur le plateau.

Lors de chaque tour, un nouveau plateau est généré. Les joueurs doivent identifier des mots formés par des lettres adjacentes sur le plateau, où l'adjacence inclut les cases horizontales, verticales et diagonales. Les mots doivent comporter au moins deux lettres et respecter certaines règles : ils peuvent être au singulier, au pluriel ou conjugués, et les mots trouvés dans un même tour doivent être uniques.

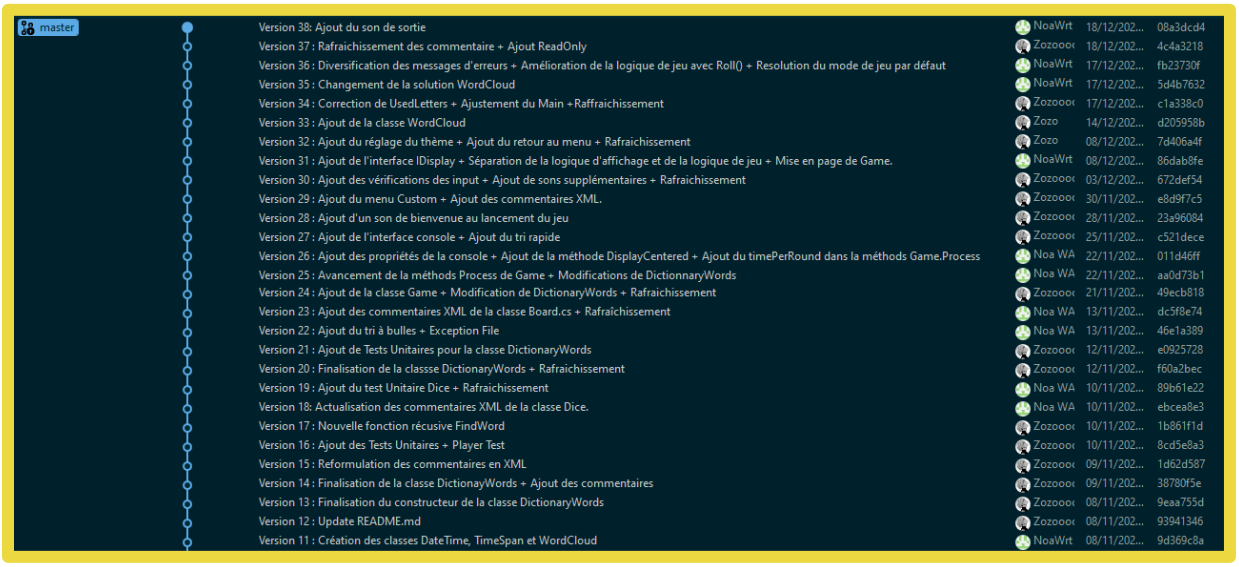
Les joueurs saisissent leurs mots au clavier. À la fin du compte à rebours général, la partie se termine, et les résultats des joueurs peuvent être comparés.

3. Organisation de l'équipe

Notre binôme, composé de Noa WAROT et Enzo BOTTURA, a fait preuve d'une grande efficacité tout au long de la réalisation de ce projet. Dès le départ, nous avons mis en place un document de suivi (**Google Docs**) dans lequel nous avons consigné tous les problèmes rencontrés, les solutions apportées, l'utilisation de **ChatGPT**, ce que cet outil nous a permis d'apprendre, ainsi que les optimisations réalisées. Ce document contenait également des notes relatives au développement de certaines fonctionnalités spécifiques, telles que la recherche de mots dans un dictionnaire ou la génération d'un nuage de mots.

L’utilisation de **GitHub** a été un atout majeur pour assurer une collaboration fluide et efficace. Nous avons réalisé un total de 39 commits, répartis entre 16 commits pour Noa et 22 commits pour Enzo.

Liste des derniers commits du projet Boogle Zoa (branche : master)



Analyse initiale et Conception

1. Etude du problème

Le projet consistait à concevoir un jeu console en C# simulant une version améliorée du célèbre jeu "**Boggle**". Le jeu devait répondre à plusieurs exigences principales, et sa structure devait s'appuyer sur une architecture orientée objet comprenant les classes suivantes :

- Classe **Player** (Joueur)
  - Représente un joueur du jeu.
  - Attributs principaux : nom du joueur, score, et liste des mots trouvés.
  - Méthodes clés :
    - Ajout d'un mot à la liste des mots trouvés.
    - Vérification de l'unicité d'un mot dans la liste.

- Classe **Dice** (Dé)
  - Modélise un dé avec six faces portant des lettres.
  - Attributs principaux : liste des lettres et la lettre visible (face supérieure).
  - Méthodes clés :
    - Génération aléatoire de la lettre visible.

- Classe **DictionaryWords** (Dictionnaire)
  - Gère les mots valides selon la langue sélectionnée.
  - Attributs principaux : langue et structure optimisée pour la recherche de mots.
  - Méthodes clés :
    - Vérification qu'un mot appartient au dictionnaire.
    - Méthode de tri optimisé.

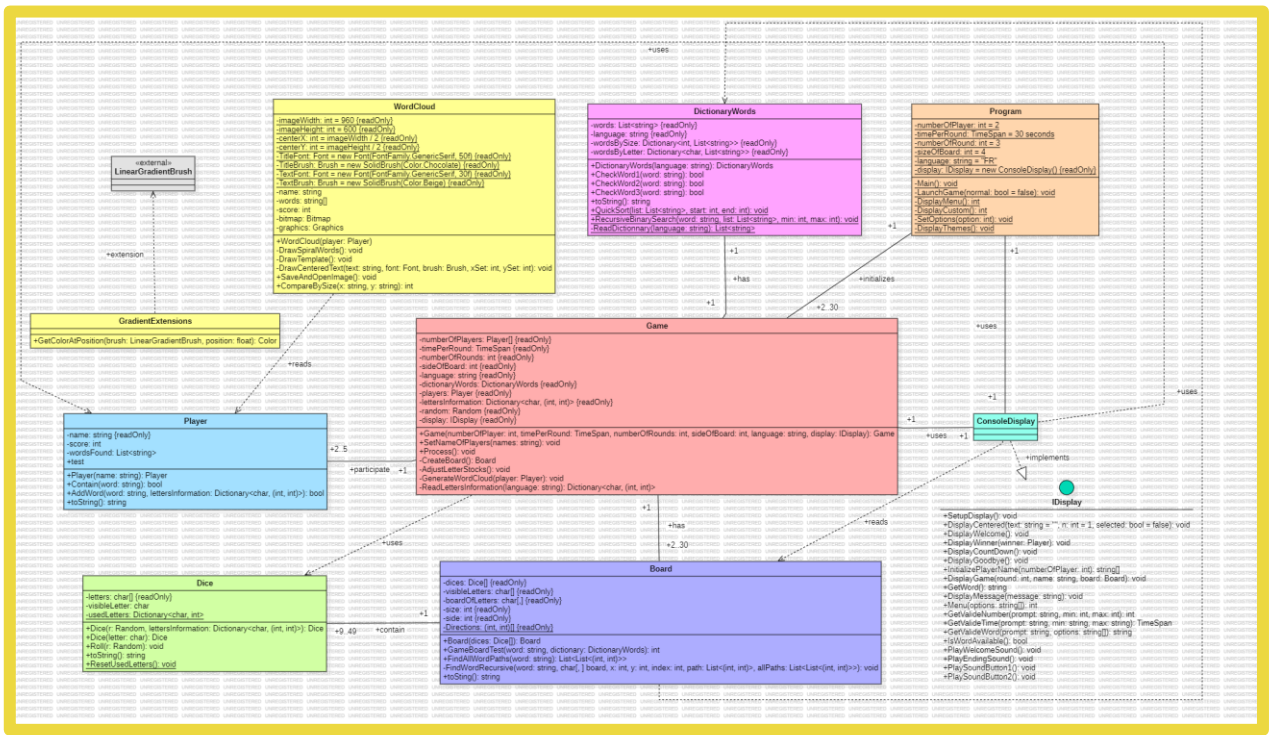
- Classe **Board** (Plateau)
  - Représente le plateau de jeu composé de dés.
  - Attributs principaux : tableau de dés et représentation sous forme de matrice des faces visibles.
  - Méthodes clés :
    - ❖ Génération d'un nouveau plateau.
    - ❖ Vérification qu'un mot est formable sur le plateau (respect des contraintes d'adjacence).

- Classe **Game** (Jeu)
  - Coordonne le déroulement de la partie.
  - Attributs principaux : liste des joueurs, paramètres de la partie.
  - Méthodes clés :
    - ❖ Gestion des tours de jeu.
    - ❖ Détermination du gagnant.
    - ❖ Lecture des informations concernant les lettres (poids, quantités)

- Classe **WordCloud** (Nuage de Mots)
  - Génère un nuage de mots pour représenter graphiquement les mots trouvés des joueurs.
  - Attributs principaux : liste des mots trouvés par un joueur et leurs fréquences.
  - Méthodes clés :
    - ❖ Création d'un visuel graphique pour le nuage de mots.
    - ❖ Gestion de la taille et de l'apparence des mots en fonction de leur fréquence.

2. Modélisation UML

Liste des derniers commits du projet Boogle Zoa (branche : master)



Développement du jeu

1. Architecture du programme

Pour l'architecture de notre programme, nous avons fait le choix d'écrire le **code en anglais** afin de nous habituer aux environnements de développement couramment utilisés et parce que cela rend le programme plus agréable à lire et à partager. Les noms des variables, des fonctions et des classes sont soigneusement choisis en anglais pour qu'ils soient clairs, concis et reflètent précisément leur rôle dans le programme. Les **commentaires**, en revanche, sont rédigés en français pour faciliter la compréhension des intentions et des choix techniques derrière chaque partie du code.

A. Classe Player

La classe Player représente un joueur qui dispose d'un **nom** ainsi que d'un **score**, et de la **liste des mots** qui l'a trouvé, respectivement initialisé à 0 et par une liste vide.

Méthode AddWord()

La méthode AddWord a pour rôle d'ajouter un mot à la liste des mots trouvés par un joueur et de mettre à jour son score en fonction des lettres qui composent ce mot. Avant tout, elle vérifie que le mot n'est pas déjà présent dans la liste afin d'éviter les doublons. Si le mot est nouveau, il est ajouté à la liste et le score du joueur est calculé en additionnant les poids des lettres du mot, selon les valeurs définies dans le dictionnaire ***lettersInformation*** (nous reviendrons plus tard sur ce dictionnaire). La méthode retourne un **booléen** indiquant si l'ajout a réussi (*true*) ou si le mot était déjà présent (*false*).

Elle repose également sur la supposition implicite que le mot passé en paramètre existe dans le dictionnaire de la langue concernée, ce qui réduit son champ d'action à la gestion des mots trouvés et au calcul du score.

```
1. public bool AddWord(string word, Dictionary<char, (int, int)> lettersInformation)
 2. {
 3.     bool succeeded = false;
 4.
 5.     word = word.ToUpper();
 6.
 7.     if (!Contain(word))
 8.     {
 9.         succeeded = true;
10.
11.         wordsFound.Add(word);
12.
13.         foreach(char c in word)
14.         {
15.             score += lettersInformation[c].Item1;
16.         }
17.     }
18.     return succeeded;
19. }
```

B. Classe Dice

La classe Dice représente un dé à six faces portant des **lettres**, conçu pour modéliser le mécanisme d'apparition **aléatoire** des lettres dans le jeu Boogle.

Pertinence du dictionnaire usedLetters

Le dictionnaire statique usedLetters est une variable essentielle pour garantir l'équité dans la répartition des lettres tout au long de la partie. En enregistrant le nombre d'apparitions de chaque lettre visible, il nous permet de respecter les contraintes définies par le jeu comme la **limite d'apparition** des lettres ou leur **disponibilité**. Cette approche assure une gestion des statistiques permettant de contrôler suffisamment la génération des dés aux besoins du plateau de jeu. Ainsi, en centralisant ces données au sein d'une structure unique, la classe évite les redondances et facilite le suivi des lettres utilisées, renforçant ainsi la **cohérence** et la **jouabilité**.

```
1. private static Dictionary<char, int> usedLetters = new Dictionary<char, int>
2. {
3.     { 'A', 0 }, { 'B', 0 }, { 'C', 0 }, { 'D', 0 }, { 'E', 0 },
4.     { 'F', 0 }, { 'G', 0 }, { 'H', 0 }, { 'I', 0 }, { 'J', 0 },
5.     { 'K', 0 }, { 'L', 0 }, { 'M', 0 }, { 'N', 0 }, { 'O', 0 },
6.     { 'P', 0 }, { 'Q', 0 }, { 'R', 0 }, { 'S', 0 }, { 'T', 0 },
7.     { 'U', 0 }, { 'V', 0 }, { 'W', 0 }, { 'X', 0 }, { 'Y', 0 },
8.     { 'Z', 0 }
9. };
```

Méthode ResetUsedLetters

Pour gérer efficacement les parties successives, nous avons implémenté la méthode statique ResetUsedLetters, qui réinitialise les statistiques d'utilisation des lettres, autrement dit, elle remet à zéro **usedLetters**. Cette fonctionnalité nous permet de démarrer une nouvelle partie. Ainsi, elle garantit un état initial cohérent pour chaque nouvelle partie, éliminant ainsi tout problèmes provenant des parties précédentes. En offrant un moyen accessible de restaurer les données du jeu, cette méthode est pratique pour d'éventuel tests ou débogage en assurant un contrôle total sur l'état des lettres utilisées.

```
1. public static void ResetUsedLetters()
2. {
3.     usedLetters = new Dictionary<char, int>
4.     {
5.         { 'A', 0 }, { 'B', 0 }, { 'C', 0 }, { 'D', 0 }, { 'E', 0 },
6.         { 'F', 0 }, { 'G', 0 }, { 'H', 0 }, { 'I', 0 }, { 'J', 0 },
7.         { 'K', 0 }, { 'L', 0 }, { 'M', 0 }, { 'N', 0 }, { 'O', 0 },
8.         { 'P', 0 }, { 'Q', 0 }, { 'R', 0 }, { 'S', 0 }, { 'T', 0 },
9.         { 'U', 0 }, { 'V', 0 }, { 'W', 0 }, { 'X', 0 }, { 'Y', 0 },
10.        { 'Z', 0 }
11.    };
12. }
```

Constructeur Dice

Le constructeur principal de la classe Dice a été conçu pour générer des dés en tenant compte des contraintes spécifiques du jeu Boogle. Grâce à une boucle de validation intégrée, il s'assure que chaque face respecte les limites imposées sur la limite d'apparition des lettres. Tant que la lettre choisie aléatoirement ne peut pas être choisi, nous choisissons une nouvelle lettre aléatoirement.

```
1. public Dice(Random r, Dictionary<char, (int,int)> lettersInformation)
2. {
3.     letters = new char[6];
4.
5.     for (int i = 0; i < 6; i++)
6.     {
7.         int n;
8.         do
9.         {
10.            n = r.Next(lettersInformation.Count);
11.            letters[i] = lettersInformation.ElementAt(n).Key;
12.        }
13.        while (usedLetters[letters[i]] >= lettersInformation[letters[i]].Item2);
14.    }
15. }
```

C. Classe DictionaryWords

Elle représente un dictionnaire structuré contenant les mots d'une langue donnée, organisés de manière à permettre des recherches et des manipulations optimisées.

Pertinence des structures wordsBySize et wordsByLetter

Nous avons intégré deux structures, wordsBySize et wordsByLetter, pour optimiser l'organisation et la recherche des mots. La première associe chaque taille de mot à une liste de mots correspondante, ce qui facilite la recherche par **longueur**. La seconde associe chaque lettre à une liste de mots commençant par cette lettre, permettant une recherche rapide par **préfixe**. Ces structures combinées réduisent significativement le nombre de mots à parcourir lors des **recherches**, tout en minimisant l'utilisation de la **mémoire**, car seules les tailles et lettres réellement présentes dans les données sont stockées.

```
1. private Dictionary<int, List<string>> wordsBySize = new Dictionary<int, List<string>> { };
2. private Dictionary<char, List<string>> wordsByLetter = new Dictionary<char, List<string>> { };
```

Tri rapide avec QuickSort

Nous avons choisi de développer la méthode QuickSort pour trier nos mots, car elle offre une grande efficacité en termes de **complexité temporelle**, avec un temps d'exécution moyen de **O(n log n)**. Le tri fonctionne en choisissant un **pivot**, en réorganisant les éléments de manière que ceux inférieurs au pivot soient à gauche et ceux supérieurs à droite, puis en appliquant récursivement cette logique à chaque sous-partition. Initialement, nous avions conçu une méthode basée sur le **tri à bulles**, mais celle-ci s'est révélée trop lente pour des listes volumineuses. C'est pourquoi nous avons exploré des alternatives plus performantes, dont le tri rapide, qui a considérablement amélioré le **temps de compilation**.

```
 1. public static void QuickSort(List<string> list, int start, int end)
 2. {
 3.     if (end - start < 1) return;
 5.     string pivot = list[end];
 6.     int wall = start;
 7.     int current = start;
 8.     string temp;
 9.
10.     while (current < end)
11.     {
12.         if (string.Compare(list[current], pivot) < 0)
13.         {
14.             if (wall != current)
15.             {
16.                 temp = list[current];
17.                 list[current] = list[wall];
18.                 list[wall] = temp;
19.             }
20.             wall++;
21.         }
22.         current++;
23.     }
25.     string tmpPivot = list[wall];
26.     list[wall] = list[end];
27.     list[end] = tmpPivot;
29.     QuickSort(list, start, wall - 1);
30.     QuickSort(list, wall + 1, end);
31. }
```

Recherche dichotomique avec RecursiveBinarySearch

La méthode RecursiveBinarySearch repose sur un algorithme de recherche **dichotomique**, qui est performant. Elle divise la liste triée en deux à chaque étape, comparant l'élément cherché au milieu de la liste, pour ne garder que la moitié pertinente. En ne nécessitant qu'un temps de recherche logarithmique, **O(logn)**, cette méthode est idéale pour des ensembles de données volumineux. De plus, en la rendant récursive et générique, nous avons pu l'intégrer facilement à nos différents cas de recherche, tout en conservant une logique claire et compacte.

```
1. public static bool RecursiveBinarySearch(string word, List<string> list, int min, int max)
2. {
3.     if (min > max || list == null || list.Count == 0)
4.     {
5.         return false;
6.     }
7.
8.     int mid = (min + max) / 2;
9.
10.    int comparaison = string.Compare(word, list[mid]);
11.
12.    if (comparaison == 0)
13.    {
14.        return true;
15.    }
16.    else if (comparaison < 0)
17.    {
18.        return RecursiveBinarySearch(word, list, min, mid - 1);
19.    }
20.    else
21.    {
22.        return RecursiveBinarySearch(word, list, mid + 1, max);
23.    }
24. }
```

D. Classe DictionaryWords

La classe Board modélise un plateau de jeu carré (ex. 4x4), généré à partir de dés, où chaque case contient une lettre visible permettant aux joueurs de rechercher des mots.

Fonctionnement de FindWordRecursive

La méthode FindWordRecursive effectue une recherche récursive sur le plateau pour trouver tous les chemins possibles formant un mot donné à partir d'une case de départ. Elle explore chaque lettre en vérifiant les cases adjacentes (en suivant les 8 directions possibles définis dans **directions**) et s'assure que les conditions suivantes sont respectées : les coordonnées sont valides ; la lettre correspond à celle attendue dans le mot ; une case ne peut pas être utilisée plus d'une fois dans le même chemin. Lorsqu'un chemin valide pour le mot entier est trouvé, il est ajouté à la collection de chemins.

```
1. private void FindWordRecursive(string word,char[,] board,int x,int y,int index,List<(int, int)> path, List<List<(int, int)>> allPaths)
 2. {
 3.     if (x < 0 || x >= board.GetLength(0) || y < 0 || y >= board.GetLength(1) || board[x, y] != word[index] || path.Contains((x, y)))
 4.     {
 5.         return;
 6.     }
 7.
 8.     path.Add((x, y));
 9.
10.     if (index == word.Length - 1)
11.     {
12.         allPaths.Add(new List<(int, int)>(path));
13.     }
14.     else
15.     {
16.         foreach (var (dx, dy) in Directions)
17.         {
18.             int newX = x + dx;
19.             int newY = y + dy;
20.             FindWordRecursive(word, board, newX, newY, index + 1, path, allPaths);
21.         }
22.     }
23.     path.RemoveAt(path.Count - 1);
24. }
```

Fonctionnement de FindAllWordPaths

Cette méthode parcourt chaque case du plateau et utilise **FindWordRecursive** pour lancer la recherche de chemins possibles à partir de chaque position. Elle compile ainsi tous les chemins valides qui forment le mot recherché.

De plus, les positions des lettres utilisées dans les chemins sont stockées, ce qui ouvre des possibilités supplémentaires, comme l'ajout d'une animation mettant en surbrillance les mots trouvés par le joueur, rendant le jeu plus interactif et visuel.

```
 1. public List<List<(int, int)>> FindAllWordPaths(string word)
 2. {
 3.     List<List<(int, int)>> allPaths = new List<List<(int, int)>> { };
 4.     word = word.ToUpper();
 5.
 6.     for (int x = 0; x < boardOfLetters.GetLength(0); x++)
 7.     {
 8.         for (int y = 0; y < boardOfLetters.GetLength(1); y++)
 9.         {
10.             FindWordRecursive(word, boardOfLetters, x, y, 0, new List<(int, int)>(), allPaths);
11.         }
12.     }
13.
14.     return allPaths;
15. }
```

Fonctionnement de GameBoardTest

Cette méthode vérifie si un mot donné est à la fois sur le plateau et présent dans le dictionnaire. Elle utilise le dictionnaire pour une première vérification, puis cherche les chemins possibles pour le mot sur le plateau en appelant FindAllWordPaths. Elle retourne :

- 0 si le mot est présent sur le plateau et dans le dictionnaire.
- 1 si le mot est uniquement dans le dictionnaire mais pas sur le plateau.
- 2 si le mot est absent des deux.

Une idée d'amélioration pour cette méthode pourrait inclure le développement des règles de jeu en multipliant le score d'un mot trouvé par le nombre de fois qu'il est présent sur le plateau. Cela offrirait une dimension stratégique et enrichirait l'expérience du joueur.

```
1. public int GameBoardTest(string word, DictionaryWords dictionary)
2. {
3.     int wordFound = 2;
4.
5.     if (dictionary.CheckWord3(word))
6.     {
7.         List<List<(int, int)>> allPaths = new List<List<(int, int)>>();
8.         allPaths = FindAllWordPaths(word);
9.
10.        if (allPaths.Count > 0)
11.        {
12.            wordFound = 0;
13.        }
14.        else
15.        {
16.            wordFound = 1;
17.        }
18.    }
19.    return wordFound;
20. }
```

E. Classe Game

La classe Game gère le déroulement complet d'une partie de jeu, en gérant les joueurs, le plateau, les règles, les interactions et les données nécessaires.

Rôle de IDisplay et ConsoleDisplay :

Le champ IDisplay display permet de séparer la **logique de jeu**, qui est centralisée dans la classe Game, de la **logique d'affichage**. En utilisant l'interface IDisplay, le jeu peut déléguer toutes les tâches d'affichage à une implémentation spécifique, telle que ConsoleDisplay, offrant une modularité et une flexibilité accrues pour le développement de l'affichage.

```
1. private readonly IDisplay display;
```

Rôle d'AdjustLetterStocks :

La méthode AdjustLetterStocks garantit que le stock de lettres disponibles dans le jeu est ajusté en fonction des **paramètres** de la partie (nombre de joueurs, de manches et taille du plateau). Cela assure une répartition **équilibrée** des lettres pour le bon fonctionnement du jeu, en évitant les **pénuries** ou **excédents** qui pourraient déséquilibrer la partie.

```
 1. private void AdjustLetterStocks()
 2. {
 4.     int totalStock = sideOfBoard * sideOfBoard * numberOfPlayers * numberOfRounds;
 5.
 7.     int currentStock = 0;
 8.     foreach ((int, int) info in lettersInformation.Values)
 9.     {
10.         currentStock += info.Item2;
11.     }
12.
14.     double adjustmentFactor = (double)totalStock / currentStock;
15.
17.     foreach (char letter in lettersInformation.Keys)
18.     {
19.         (int, int) info = (lettersInformation[letter].Item1,
(int)Math.Round(lettersInformation[letter].Item2 * adjustmentFactor) + 1);
20.         lettersInformation[letter] = info;
21.     }
22. }
```

Rôle de Process :

La méthode Process est le cœur du jeu, gérant chaque manche et chaque tour de chaque joueur. Elle orchestre la génération des plateaux, l'interaction des joueurs (validation des mots et calcul des scores), et la détermination du gagnant. Elle utilise également des outils comme **IDisplay** pour afficher des messages et fournit des fonctionnalités comme la génération de nuages de mots.

```
 1. public void Process()
 2. {
 3.     TimeSpan duration;
 4.     DateTime startTime;
 5.     DateTime endTime;
 6.
 7.     for (int r = 0; r < numberOfRounds; r++)
 8.     {
10.         for (int p = 0; p < numberOfPlayers; p++)
11.         {
12.             duration = timePerRound;
13.             startTime = DateTime.Now;
14.             endTime = startTime + duration;
15.
16.             Board board = CreateBoard();
17.
18.             display.DisplayGame(r, players[p].Name, board);
19.
20.             while (DateTime.Now < endTime)
21.             {
22.                 (…)
23.             }
24.         }
25.     }
27.     (…)
29.     display.DisplayWinner(winner);
30. }
```

2. Problèmes rencontrés et solution apportées

Tout au long de notre programme, nous avons naturellement rencontré divers problèmes, que nous avons parfois résolus grâce à des ressources telles que des documentations, des sites question-réponse comme "Stack Overflow" ou l'intelligence artificielle. Nous avons jugé essentiel de documenter les difficultés rencontrées, afin d'éviter de répéter les mêmes erreurs à l'avenir, que ce soit dans le cadre du projet Boogle ou de nos futurs travaux. De plus, nous notons aussi les solutions apportées afin de développer nos compétences en c#, en POO et aussi en algorithmique.

05/11/24	**Comment peut-on stocker les mots trouvés par le joueur dans un tableau string[] alors que ce dernier à une taille fixe ?**	*Player.cs* Constructeur
Nous avons remplacé le tableau **string[]** par une liste dynamique **List<int>** qui permet d'ajouter ou supprimer des éléments, dynamiquement. De plus, cette solution permettra d'alléger le code (au lieu de créer un nouveau tableau avec une taille plus grande et copier les éléments de l'ancien tableau vers le nouveau).		
06/11/24	**Comment vérifier qu'une lettre peut être sur une face de dé sachant qu'elle a une limite d'apparition dans la partie ?**	*Dice.cs* Constructeur
Nous avons ajouté à la classe **Dice** un attribut de classe **usedLetters** qui est un dictionnaire qui a pour clé un **char** et pour valeur un **int**. Initialement, cet attribut prends en clé toutes les lettres de l'alphabet et la valeur associé est de 0. Quand le dé est lancé, une lettre est choisie et est donc utilisée. Lors de la création des faces du dé, on vérifie que le nombre d'utilisation de la lettre n'est pas supérieur au quota prédéfini.		
08/11/24	**Comment lire un fichier .txt pour mettre les mots qu'il contient dans une liste ?**	*DictionaryWords.cs* Constructeur
Nous avons utilisé la classe **File** et plus particulièrement sa méthode **ReadAllText()**. Puis nous avons complété la liste **words** en utilisant la méthode **Split()** qui permet de découper un **string** selon un **char**. De plus, **words** est une liste dynamique de type **List<string>** car nous ne connaissons pas sa taille d'avance.		
08/11/24	**Comment vérifier qu'une clé est contenue dans un dictionnaire quand ce dictionnaire est initialement null ?**	*DictionaryWords.cs* Constructeur
Nous avons tout simplement initialisé les deux dictionnaires **wordsBySize** et **wordsByLetter** par un dictionnaire vide afin de ne pas déclencher l'erreur.		
09/11/24	**On vérifie que la clé est présente dans le dictionnaire, pour ne pas provoquer d'erreur (i=0; i=1)**	*DictionaryWords.cs* Constructeur

Nous avons configuré le constructeur pour que lorsqu'il parcourt chaque mot de la liste, il récupère ses clés correspondantes : première lettre et taille du mot. Puis il vérifie si ses clés sont présentes dans les clés des dictionnaires. Si c'est le cas, il ajoute simplement le mot à la liste raccordée à la clé. Si ce n'est pas le cas, il crée une nouvelle paire (clé, liste) à l'aide de la clé du mot et une liste vide dont on a ajouté le mot.		
10/11/24	**Comment créer une méthode récursive qui permet de relever les différents chemins qui forment le mot recherché ?**	*Board.cs* GameBoard_Test
Nous avons pris comme base notre première fonction récursive et nous avons cherché à réaliser une fonction récursive différente. Initialement, notre fonction renvoie 0 si le mot n'a pas été trouvé, et **n**, s'il a été trouvé **n** fois. Désormais, nous voulions une fonction qui renvoie une liste de liste de couple de int, afin qu'il renvoie la liste des chemins possible en les décrivant à l'aide des positions des lettres formant le mot. Nous voulions avoir ce type de retour pour qu'ensuite le joueur ait la possibilité de choisir quel mot il souhaite retirer. Malheureusement, réaliser ce type de retour fut compliqué, nous l'avons donc simplifié : au lieu de réaliser une fonction récursive qui renvoie une **List<List<(int, int)>>,** qui est compliqué car générer le chemin au fur et à mesure puis l'ajouter à la liste globale est complexe, nous avons réalisé une fonction récursive void qui ajoute les chemins qui aboutissent dans une liste externe. Nous avons eu recours à chatGPT pour trouver une optimisation sur la réalisation des chemins adjacents qui réduit beaucoup la complexité de notre fonction.		
12/11/24	**Comment étudier l'égalité entre deux List ?**	*BoogleZoaTest.cs* DictionaryWords
Nous avons rencontré une erreur lors des tests unitaires de la classe **DictionaryWords**. En effet, lorsque nous effectuons **Assert.AreEqual(,)** sur 2 **List<string>**, le message d'erreur suivant était retourné : "**Assert.AreEqual failed**. *Expected:<System.Collections.Generic.List1[System.String]>. Actual:<System.Collections.Generic.List1[System.String]>.*" Le problème était que **Assert.AreEqual** comparait les objets de liste eux-mêmes plutôt que leur contenu; la solution consiste à utiliser **CollectionAssert.AreEqual** qui permet de comparer le contenu des deux listes.		
19/11/24	**Comment effectuer les tests unitaires de la classe Board étant donné que la génération du plateau est aléatoire ?**	*Board.cs* Constructeur
Nous avons créé un second constructeur pour la classe **Board** qui prend comme paramètre en plus une liste de lettres afin de pouvoir générer un plateau en définissant les lettres en avance?		
21/11/24	**Comment initialiser une seule fois lettersInformation (attribut statique) à l'aide d'instructions afin de ne pas effectuer ce programme à chaque instance de Game ?**	*Game.cs* lettersInformation

Nous avons d'abord essayé de mettre les instructions dans le constructeur. Mais le programme nous renvoie à une erreur incompréhensible. Donc nous avons cherché un moyen de le déclarer en dehors du constructeur. En utilisant son constructeur statique, nous pouvons effectuer les instructions une seule fois (optimisation) et affecter une valeur à **letterInformation** lors du lancement du programme.

21/11/24	**Comment améliorer l'algorithme de tri BubbleSort pour améliorer le temps d'exécution ? (>30 min)**	*DictionaryWords.cs* Constructeur

Lors de la construction de **DictionnaryWords**, nous récupérons la liste de mots dans le fichier, nous répartissons ces mots dans les deux structures (**wordsByLetter**, **wordsBySize**). Puis nous parcourons toutes les sous-listes présentes dans ces deux structures, et nous lançons un algorithme de tri (**BubbleSort**) sur chacune de ces sous-listes. Au final, c'est comme si on triait deux fois la liste entière de mots.
La solution que nous apportons c'est de trier dans un premier temps la liste entière de mots. Ainsi lors de la répartition des mots dans les deux structures, les mots seront déjà triés et les sous-listes des deux structures seront naturellement triées. Le temps de calcul est réduit au minimum par deux ! **(>20 min)** La solution n'est pas efficace.

22/11/24	**Comment bloquer le redimensionnement de la Console avec c# ?**	*Program.cs* Main

Pour bloquer le redimensionnement de la console, le code utilise des fonctions natives de **Windows** importées depuis les bibliothèques **user32.dll** et **kernel32.dll**. La fonction **GetConsoleWindow** récupère le **handle** (identifiant unique) de la fenêtre de la console, nécessaire pour interagir avec elle. Ensuite, **GetSystemMenu permet** d'accéder au menu système associé à cette fenêtre. Enfin, la fonction **DeleteMenu** est utilisée pour supprimer une option spécifique de ce menu, ici celle liée au redimensionnement, identifiée par le code **0xF000**.

23/11/24	**Comment bloquer le défilement horizontal de la Console ?**	*Program.cs* SetupDisplay

En plus de fixer la taille de la fenêtre avec **SetWindowSize()**, nous avons également fixé avec la même taille le tampon de sortie à l'aide de **SetBufferSize()**. Un tampon de sortie d'une console en C# est une zone mémoire où sont stockées temporairement les lignes de texte à afficher.
Si le tampon de sortie est plus grand que la fenêtre, le texte qui dépasse les dimensions visibles peut être consulté en défilant avec les barres de défilement horizontales ou verticales. Ainsi, si le tampon de sortie est égal à la taille de la fenêtre, le défilement horizontal est bloqué, car la taille des données visibles correspond exactement à la zone disponible.

08/12/24	**Comment gérer l'affichage du jeu et la logique de jeu de façon séparée ?**	*DictionaryWords.cs* Constructeur

Réaliser l'affichage dans la classe Game était difficile car nous devions utiliser beaucoup de méthodes et de variables provenant du Main (width, display(), bordure). Pour résoudre ce problème, nous avons introduit une interface **IDisplay** pour séparer l'affichage de la logique de jeu. Cette interface définit des méthodes telles que **DisplayWelcome()**, **DisplayCentered()**, et **InitializePlayerName()**. Une classe **ConsoleDisplay** a été créée pour implémenter cette interface, centralisant ainsi la logique d'affichage.
Ainsi, la classe **Game** prend désormais une instance de **IDisplay** (**ConsoleDisplay**) en paramètre, l'utilisant pour toutes les opérations d'affichage. Le Main a été simplifié pour initialiser **ConsoleDisplay** et le passer à **Game**.

3. Optimisation et Performance

Lors du développement de notre application, nous avons identifié plusieurs points d'amélioration pour optimiser les performances et réduire la consommation de mémoire. Ces ajustements permettent de rendre le code plus efficace, tout en conservant sa lisibilité et sa robustesse. Nos efforts se sont concentrés sur la gestion des ressources, la réduction des calculs redondants et l'utilisation de structures de données adaptées. Voici un aperçu des optimisations réalisées et leurs impacts.

- **Réduction des variables redondantes :**

Dans le constructeur de la classe **DictionaryWords** par exemple, une seule variable est utilisée pour les traitements répétés au lieu d'en déclarer une nouvelle à chaque itération, ce qui simplifie le code et réduit la consommation de mémoire.

- **Utilisation d'un couple d'entiers au lieu d'une liste de chaînes :**

Pour représenter les informations des lettres **(poids et nombre disponibles)**, nous utilisons désormais un dictionnaire dont la valeur associée à chaque lettre est un couple d'entiers **(int, int)** plutôt qu'une liste de chaînes ou d'entiers. Cette optimisation réduit l'espace mémoire utilisé et améliore l'accès aux données grâce à une structure plus compacte et spécifique.

Optimisation de la recherche d'un mot

Dans le cadre de notre projet, nous avons exploré différentes approches pour optimiser le temps de recherche d'un mot dans le dictionnaire. Ces optimisations visaient à réduire la complexité des opérations, notamment en limitant le nombre de mots sur lesquels la recherche devait être effectuée, et en tirant parti d'algorithmes efficaces comme la recherche dichotomique.

Voici un résumé des techniques que nous avons testées :

❖ Tentative 1 : Filtrage par longueur de mot

Dans cette première approche, nous avons réduit la liste des mots sur laquelle la recherche était effectuée en sélectionnant uniquement les mots ayant la même longueur que le mot recherché. Cela nous a permis de restreindre considérablement le nombre d'éléments à analyser. Ensuite, nous avons appliqué une **recherche dichotomique** sur cette liste restreinte, qui était triée au préalable.

Avantages :

- Réduction de la complexité grâce à un espace de recherche plus petit.
- Simplicité de mise en œuvre.

Limites :

- Si plusieurs mots partagent la même longueur, la liste peut rester relativement grande, rendant la recherche moins efficace dans certains cas.

```
1. public bool CheckWord1(string word)
2. {
3.     bool exist = false;
4.
5.     word = word.ToUpper();
6.     int n = word.Length;
7.
8.     if (wordsBySize.ContainsKey(n))
9.     {
10.        List<string> sameLengthWords = wordsBySize[n];
11.        exist = RecursiveBinarySearch(word, sameLengthWords, 0, sameLengthWords.Count - 1);
12.    }
13.    return exist;
14. }
```

❖ Tentative 2 : Filtrage par longueur et première lettre

Pour améliorer davantage les performances, nous avons introduit un second critère de filtrage : la première lettre du mot. Nous avons ainsi récupéré deux listes : une contenant les mots de même longueur et une autre contenant les mots commençant par la même lettre. La **recherche dichotomique** était ensuite appliquée sur la plus petite des deux listes, pour minimiser le nombre d'éléments à examiner.

Avantages :

- Réduction significative de l'espace de recherche dans la plupart des cas.
- Optimisation de la mémoire : nous avons évité de recréer des listes intermédiaires inutiles.

Limites :

- Le croisement des deux critères pouvait encore laisser des listes importantes lorsque les mots partageaient des caractéristiques communes.

```
1. public bool CheckWord2(string word)
2. {
3.     bool exist = false;
4.
5.     word = word.ToUpper();
6.     int n = word.Length;
7.     char c = word[0];
8.
9.     if (wordsBySize.ContainsKey(n) && wordsByLetter.ContainsKey(c))
10.    {
11.        List<string> sameLengthWords = wordsBySize[n];
12.        List<string> sameLetterWords = wordsByLetter[c];
13.
14.        if (sameLengthWords.Count < sameLetterWords.Count)
15.        {
16.            exist = RecursiveBinarySearch(word, sameLengthWords, 0, sameLengthWords.Count-1);
17.        }
18.        else
19.        {
20.            exist = RecursiveBinarySearch(word, sameLetterWords, 0, sameLetterWords.Count - 1);
21.        }
22.    }
23.
24.    return exist;
25. }
```

❖ Tentative 3 : Intersection des listes

Dans cette dernière approche, nous avons combiné les deux critères de manière plus rigoureuse en utilisant l'intersection des listes obtenues (mots de même longueur et mots commençant par la même lettre). La liste résultante, plus petite et plus précise, était ensuite soumise à une **recherche dichotomique**.

Avantages :

- Réduction maximale de l'espace de recherche.
- Approche plus ciblée, garantissant une recherche rapide et efficace.

Limites :

- La méthode d'intersection peut introduire une surcharge computationnelle si les listes initiales sont très grandes.
- La conversion de la liste résultante en un type manipulable (par exemple List<string>) peut nécessiter des ressources supplémentaires.

```
1. public bool CheckWord3(string word)
2.  {
3.      bool exist = false;
4.
5.      word = word.ToUpper();
6.      int n = word.Length;
7.      char c = word[0];
8.
9.      if (wordsBySize.ContainsKey(n) && wordsByLetter.ContainsKey(c))
10.     {
11.         List<string> sameLengthWords = wordsBySize[n];
12.         List<string> sameLetterWords = wordsByLetter[c];
13.
14.         List<string> commonWords = sameLengthWords.Intersect(sameLetterWords).ToList();
15.         exist = RecursiveBinarySearch(word, commonWords, 0, commonWords.Count - 1);
16.     }
17.     return exist;
18. }
```

Ces différentes tentatives nous ont permis de progresser vers une solution plus performante en équilibrant la complexité algorithmique et les ressources nécessaires. Les choix effectués ont été guidés par l'objectif de réduire la **complexité temporelle** de la recherche, tout en s'assurant de la faisabilité et de la robustesse des méthodes. Cette réflexion nous servira également pour optimiser d'autres opérations similaires dans nos projets futurs.

Utilisations de ChatGPT

1. Quand avons-nous utilisé ChatGPT ?

Établissement des cas de test pour les tests unitaires :
Nous avons utilisé ChatGPT pour identifier certains cas à tester dans une classe, ce qui nous a permis de créer une couverture de tests unitaires complète. Cela a renforcé la robustesse et la fiabilité de notre code.

Reformulation de phrases pour la documentation :
Afin d'améliorer la clarté et la lisibilité de la documentation associée à notre projet, ChatGPT a reformulé certaines phrases complexes ou ambiguës. Cela a permis une meilleure compréhension.

Recherche d'optimisations :
ChatGPT a proposé diverses optimisations pour améliorer l'efficacité de notre code, comme l'utilisation de StringBuilder pour la concaténation de chaînes dans des boucles, ou la déclaration préalable de variables utilisées dans des itérations.

Complétion des méthodes toString() :
Pour assurer la validité des tests unitaires liés aux méthodes toString(), ChatGPT a assisté dans la génération des chaînes de sortie attendues.

Manipulation d'attributs statiques :
En explorant la gestion des attributs statiques dans une classe, ChatGPT nous a fourni des explications et des exemples concrets pour affecter des valeurs à ces attributs.

2. Qu’est-ce qu’on a appris ? avec ChatGPT ?

Tests unitaires et organisation :

- Utilisation de Assert.AreEqual(valeur_attendue, valeur_obtenue) pour une meilleure lisibilité des messages d’erreur.
- Adoption de la structure AAA (Arrange, Act, Assert) pour rendre lesnoms de tests plus clairs et conformes aux bonnes pratiques.

Concepts en C# :

- Le mot-clé readonly garantit une immutabilité après initialisation.
- Différences entre les blocs statiques et les constructeurs statiques : les blocs s'exécutent avant l'accès aux membres, tandis que les constructeurs s'exécutent au premier accès à la classe.

Manipulation de collections :

- Comparaison de listes avec CollectionAssert.AreEqual.
- Conversion d’une collection en liste avec .ToList().

Utilisation de la console :

- Gestion du curseur avec Console.SetCursorPosition et Console.CursorTop.
- Mesure et gestion des dimensions de la console via Console.BufferWidth.

Optimisation du code :

- Génération d'une matrice de lettres dans le constructeur pour la réutilisation.
- Utilisation d’un tuple (int, int) au lieu de listes pour économiser la mémoire.
- Introduction de StringBuilder pour optimiser la concaténation de chaînes dans des boucles.

Bonnes pratiques de nommage :

- Préférence pour des noms explicites comme actualScore plutôt que result, pour une meilleure lisibilité et professionnalisme.

3. Sources utilisées

- **Documentation officielle de Microsoft** :
  Unit Testing Best Practices – pour approfondir nos connaissances sur les tests unitaires.

- **ChatGPT** :
  Fourniture de suggestions, clarifications et exemples de code adaptés.

- **ASCII Code Reference** :
  The ASCII Code – pour la gestion des caractères dans les sorties console.

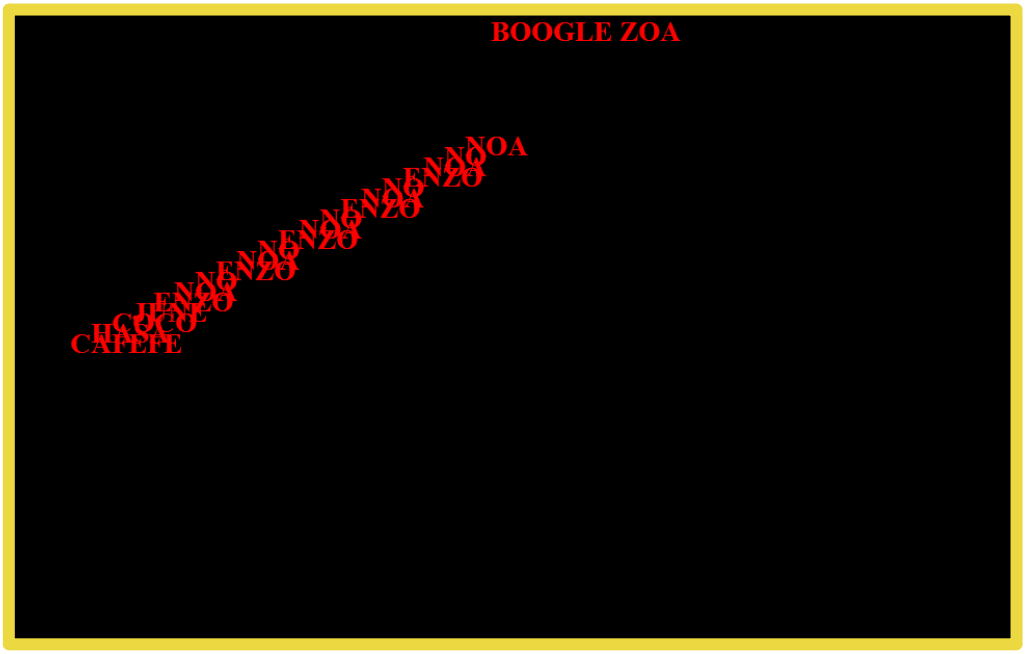
- **Tutoriel vidéo YouTube** :
  Introduction à la programmation C# – pour comprendre certaines fonctionnalités avancées du langage.

Nuage de mots

1. Nos différentes approches

Pour concevoir un nuage de mots optimal, nous avons exploré plusieurs méthodes. Initialement, nous avons envisagé l'utilisation de la représentation graphique de fonctions, où quelques points représenteront les positions où seront affichés les mots, nous offrant la liberté de choisir des formes à notre convenance. Cependant, cette approche a rapidement révélé des difficultés, notamment des superpositions de mots et des espacements inadéquats.

Nuage de mots avec l'approche « Fonction »



Nous avons ensuite considéré la méthode des "briques", qui consistait à considérer chaque brique comme étant un mot. Nous avons développé un programme qui construit un mur en suivant la disposition à demi-briques. Plus il y a de mots, plus le mur est grand. Le résultat de cette solution pouvait poser des problèmes d'affichage lorsque le mot est relativement grand. De plus, le nuage était assez monotone.

Nuage de mots avec l'approche « Brique »

Finalement, nous avons opté pour une disposition en spirale. Cette méthode place les mots le long d'une spirale centrée, en augmentant progressivement l'angle et le rayon pour chaque mot, permettant ainsi une répartition harmonieuse tout en minimisant les chevauchements.

Nuage de mots avec l'approche « Spirale »

2. Classe WordCloud

La classe WordCloud en C# est conçue pour générer des nuages de mots personnalisés à partir des données d'un joueur, notamment son nom, son score et les mots qu'il a trouvés. Elle crée une image où les mots sont disposés selon une spirale centrée, avec des variations de taille et de couleur pour chaque mot.

- **Fonctionnement de la Classe**

Lors de l'instanciation de la classe, les données du joueur sont initialisées, et les mots trouvés sont triés par longueur croissante. Une image bitmap est ensuite créée avec un fond noir, et les méthodes DrawTemplate() et DrawSpiralWords() sont appelées pour dessiner respectivement le modèle de base et les mots en spirale.

```
1. public WordCloud(Player player)
2. {
3.     name = player.Name;
4.     score = player.Score;
5.     words = player.WordsFound.ToArray();
6.
7.     Array.Sort(words, CompareBySize);
8.
9.     bitmap = new Bitmap(imageWidth, imageHeight);
10.    graphics = Graphics.FromImage(bitmap);
11.
12.    graphics.Clear(Color.Black);
13.
14.    DrawTemplate();
15.    DrawSpiralWords();
16. }
```

- **Logique de Dessin de la Spirale**

La méthode DrawSpiralWords() place les mots le long d'une spirale centrée pour éviter les chevauchements. Pour chaque mot, la taille de la police est déterminée en fonction de sa position dans la liste, assurant une variation graduelle entre une taille minimale et maximale. Le mot est ensuite positionné sur la spirale en calculant des coordonnées polaires (angle et rayon) converties en coordonnées cartésiennes. Si le mot chevauche un autre déjà placé, l'angle et le rayon sont ajustés jusqu'à trouver une position adéquate.

- **Définition des Couleurs et Tailles des Mots**

Les tailles des polices varient entre une valeur minimale et maximale, déterminées en fonction de la position du mot dans la liste triée. Les couleurs des mots sont définies à l'aide d'un dégradé linéaire allant du chocolat au beige, appliqué proportionnellement à la position du mot dans la liste, créant ainsi une transition harmonieuse des couleurs à travers le nuage de mots.

```
 1. private void DrawSpiralWords()
 2. {
 3.     int numWords = words.Length;
 4.     double angle = 0;
 5.     double radius = 10;
 6.     Font font;
 7.     Brush colorBrush;
 8.     int maxFontSize = 40;
 9.     int minFontSize = 10;
10.
11.     Color startColor = Color.Chocolate;
12.     Color endColor = Color.Beige;
13.     LinearGradientBrush gradientBrush = new LinearGradientBrush(new Point(0, 0), new
Point(imageWidth, imageHeight), startColor, endColor);
14.
15.     (…)
16. }
```

3. Utilisation de ChatGPT

Dans le cadre du développement de notre classe WordCloud, nous avons sollicité l'assistance de ChatGPT pour optimiser la disposition des mots en spirale. Dans un premier temps, nous lui avons poser un contexte pour le placer dans les meilleures conditions.

```
Contexte :

/* Vous développez un jeu console en C# inspiré du célèbre jeu de lettres "Boogle". L'objectif
est de permettre aux joueurs de trouver un maximum de mots à partir d'une grille de lettres. À la
fin de chaque partie, vous souhaitez générer une visualisation graphique des mots trouvés sous
forme d'un nuage de mots en spirale */
```

ChatGPT nous a suggéré d'utiliser une **représentation polaire**, où chaque point de la spirale est défini par une longueur et un angle, similaire à la représentation graphique d'un nombre complexe.

```
Prompt = "Comment calculer la position optimale pour placer des mots sur une spirale compacte ?"
```

Après avoir étudié cette approche, nous avons demandé à ChatGPT des recommandations sur les **constantes d'angle et de longueur** à utiliser, que nous avons ensuite ajustées selon nos préférences pour obtenir une spirale esthétiquement satisfaisante.

```
Prompt = "Quels sont les meilleurs paramètres de constantes pour ajuster une spirale afin qu'elle
soit compacte, lisible et adaptée à un nuage de mots ?"
```

Bien que la spirale générée initialement fût visuellement attrayante, nous avons rencontré un problème : lorsque deux mots se superposaient, l'un d'eux n'était pas affiché, limitant ainsi la taille de la spirale et empêchant l'affichage complet de tous les mots.

```
Prompt = "Comment gérer les chevauchements dans un nuage de mots ?"
```

Pour remédier à cela, nous avons exploré diverses solutions. En nous inspirant de notre travail précédent sur le nuage de mots utilisant des briques, nous avons entouré chaque mot d'un rectangle invisible représentant son espace occupé.

```
1. int x = (int)(centerX + currentRadius * Math.Cos(currentAngle)) - fontSize / 2;
2. int y = (int)(centerY + currentRadius * Math.Sin(currentAngle)) - fontSize / 2;
3.
4. RectangleF wordRectangle = new RectangleF(x, y, font.Size * words[i].Length, fontSize);
```

Si deux rectangles se chevauchaient, nous ajustions la position du second mot en modifiant son angle et sa distance par rapport à l'origine. ChatGPT nous a de nouveau été utile en nous fournissant des indications sur les incréments d'angle et de distance à appliquer, bien que nous ayons expérimenté différentes valeurs pour nous assurer de l'efficacité de la solution.

```
 1. bool overlaps = false;
 2. foreach (var existingWord in wordPositions)
 3. {
 4.     if (existingWord.IntersectsWith(wordRectangle))
 5.     {
 6.         overlaps = true;
 7.         break;
 8.     }
 9. }
10.
11. if (!overlaps)
12. {
13.     DrawCenteredText(words[i], font, colorBrush, x, y);
14.     wordPositions.Add(wordRectangle);
15.     placed = true;
16. }
17. else
18. {
19.     currentAngle += Math.PI / 36;  // Petite augmentation de l'angle
20.     currentRadius += 2;  // Augmentation du rayon pour éloigner le mot
21. }
```

Pour ajuster la couleur de chacun de nos mots et obtenir un rendu plus esthétique, nous avons fait appel à ChatGPT, qui nous a permis de concevoir une méthode efficace appelée GetColorAtPosition(). Cette méthode, intégrée dans une classe d'extension, génère un dégradé de couleur fluide entre une couleur de départ et une couleur d'arrivée. En prenant en entrée la position relative (entre 0 et 1), elle calcule la couleur correspondante dans le dégradé en déterminant les composantes RGBA des deux couleurs définies dans un LinearGradientBrush. Le résultat est un dégradé parfait, parfaitement adapté à nos besoins.

```
 1. public static class GradientExtensions
 2. {
 3.     public static Color GetColorAtPosition(this LinearGradientBrush brush, float position)
 4.     {
 5.         Color startColor = brush.LinearColors[0];
 6.         Color endColor = brush.LinearColors[1];
 7.
 8.         int r = (int)(startColor.R + (endColor.R - startColor.R) * position);
 9.         int g = (int)(startColor.G + (endColor.G - startColor.G) * position);
10.         int b = (int)(startColor.B + (endColor.B - startColor.B) * position);
11.         int a = (int)(startColor.A + (endColor.A - startColor.A) * position);
12.
13.         return Color.FromArgb(a, r, g, b);
14.     }
15.
```