

---

# **Chasse au trésor**

## **Semestre 5**

### *Justifications techniques*

ENSC - 1<sup>ère</sup> Année - Groupe 4

**MIELE Andrea**

# SOMMAIRE.....

<b>Prélude</b>	<b>3</b>
Présentation du projet	3
Contraintes techniques	3
Formalisme adopté	3
<b>Description d'une partie</b>	<b>5</b>
<b>Répertoire des fonctions utilisées</b>	<b>9</b>
Main	9
Jeu	9
Gestion des pointeurs	10
Choix de la save	10
Choix Recommencer ou Charger	10
Choix du motif	10
Choix du mode ADMIN	10
Gestion de l'accueil	11
Refaire une partie	11
Menu Statistiques	11
Fonctions d'affichage	12
Affichage Règle	12
Centrer Texte	12
Afficher Introduction	12
ASCII ART	13
Affichage des différents types de cases	13
Affichage concernant le score	14
Affichage de grilles	14
Affichage perdre / gagner	15
Affichage états des sauvegardes	15
Fonctions de création de motifs	16
Fonctions de positionnement des mines et des trésors	16
Fonctions utiles au Jeu	17
Versioning	19

# Prélude

## Présentation du projet

Dans ce document, vous trouverez la description d'une partie, le répertoire des sous-programmes utilisés ainsi que le répertoire des variables utilisées. L'objectif de ce projet est de programmer en mode console, en C#, un jeu de Chasse au trésor (équivalent au célèbre jeu "Démineur")

## Contraintes techniques

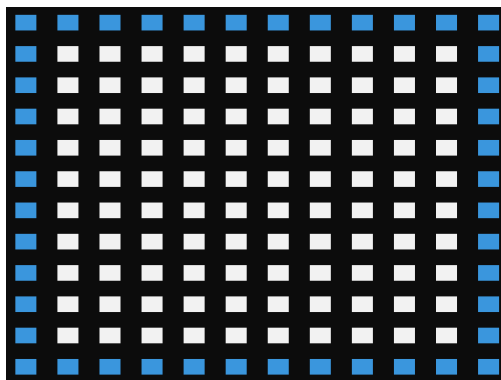
Programmation en mode console, sans utilisation de bibliothèques de fonctions externes. On exclut la programmation objet, et l'utilisation de collection (listes, tables de hachages...)

## Formalisme adopté

- Les 8 voisins de  $(x, y)$  considérés sont les suivants :

$(x-1, y-1)$	$(x-1, y)$	$(x-1, y+1)$
$(x, y-1)$	$(x, y)$	$(x, y+1)$
$(x+1, y-1)$	$(x+1, y)$	$(x+1, y+1)$

- Lorsqu'un voisin n'est pas considéré (ie : on est sur des cases "frontières", on remplace ses coordonnées  $(x, y)$  par  $(-999, -999)$ . Ainsi la liste des voisins renvoyée contient toujours 8 couples.
- Une grille de jeu est formée de cases "utiles" (ie les cases où le joueur peut jouer) et de cases "frontières". Ces dernières permettent de définir les délimitations de la grille de jeu. Elles ne sont pas représentées lors de l'affichage de la grille.



[1] Représentation d'une grille rectangulaire. Les cases "utiles" sont en blanc. Les cases "frontières" sont en cyan.

- Lors de l'enregistrement dans un fichier des informations, on enregistre les informations comme suit :

- Les quatre premières lignes contiennent les informations suivantes (dans l'ordre) : P (*Pleine*) ou V (*Vide*), Hauteur, Largeur, Score

Puis on a un enregistrement de chaque case de la grille par ligne.

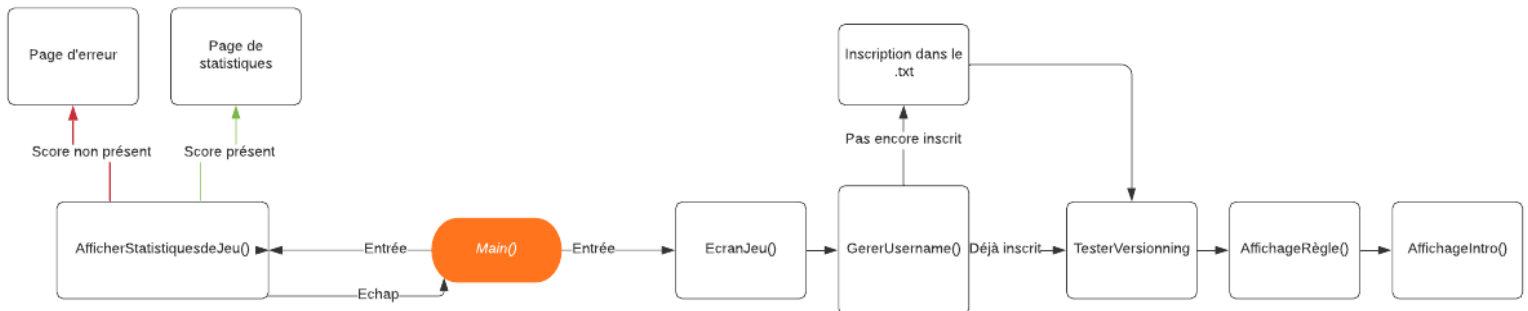
- Lorsqu'une sauvegarde est supprimée et donc considérée comme vide, elle renferme en fait une grille rectangulaire de taille 20 × 20.
  - Concernant les cases d'une grille, on adopte le formalisme suivant :
    - ❖ Case Vide : Vaut " " et est affichée en blanc.
    - ❖ Case Mines : Vaut " M " et est affichée comme ceci M
    - ❖ Case Trésor : Vaut " T " et est affichée comme ceci T
    - ❖ Case Joueur : Vaut " n " et est affichée comme ceci n
    - ❖ Case Visité : Vaut " \_ " et est affichée comme ceci \_
    - ❖ Case Frontière/Non Utile : Vaut "XXX" et n'est pas affichée pour les bordures, ou comme ceci pour le triangle
    - ❖ Case Trésor Visité : Vaut " V " et est affichée comme ceci V
    - ❖ Case Numéro/Décompte : Vaut " n " et est affichée comme ceci 1.
  - Les représentations en ASCII du score sont enregistrées dans ...\\Chasse au trésor\\...\\bin\\Debug sous le format *n.txt* avec *n* le score à afficher.
  - Dans le tableau contenant l'état de chaque sauvegarde, chaque sauvegarde a pour état soit "Vide" soit "Pleine".
  - Concernant les sauvegardes, nous faisons le choix d'en créer 4. Il est possible d'en ajouter d'autres assez simplement.
  - Dans le code, on aborde la notion "GrilleUU". GrilleUU[1:hauteur+1] ∈ M<sub>hauteur,largeur</sub>(N). Dans la case GrilleUU[0], on a un tableau de taille 4, qui contient les 4 premières lignes du .txt.
- Puis chaque ligne de GrilleUU contient une ligne de la grille de jeu.
- Les fichiers permettant le versionning sont les suivants :
    - save1.txt, save2.txt, save3.txt et save4.txt qui contiennent les quatre sauvegardes.
    - OldPosition1.txt, OldPosition2.txt, OldPosition3.txt et OldPosition4.txt qui contiennent les dernières positions jouées des quatre sauvegardes.
    - etatsSaver.txt qui contient l'état de chaque save (Plein ou Vide)
    - motifsSave.txt qui contient le motif sauvegardé dans chaque save (rectangle ou triangle)
    - NomUser.txt qui contient le nom d'utilisateur indiqué lors de la première connexion.
    - ScoreSaver.txt qui contient un enregistrement de tous les scores.

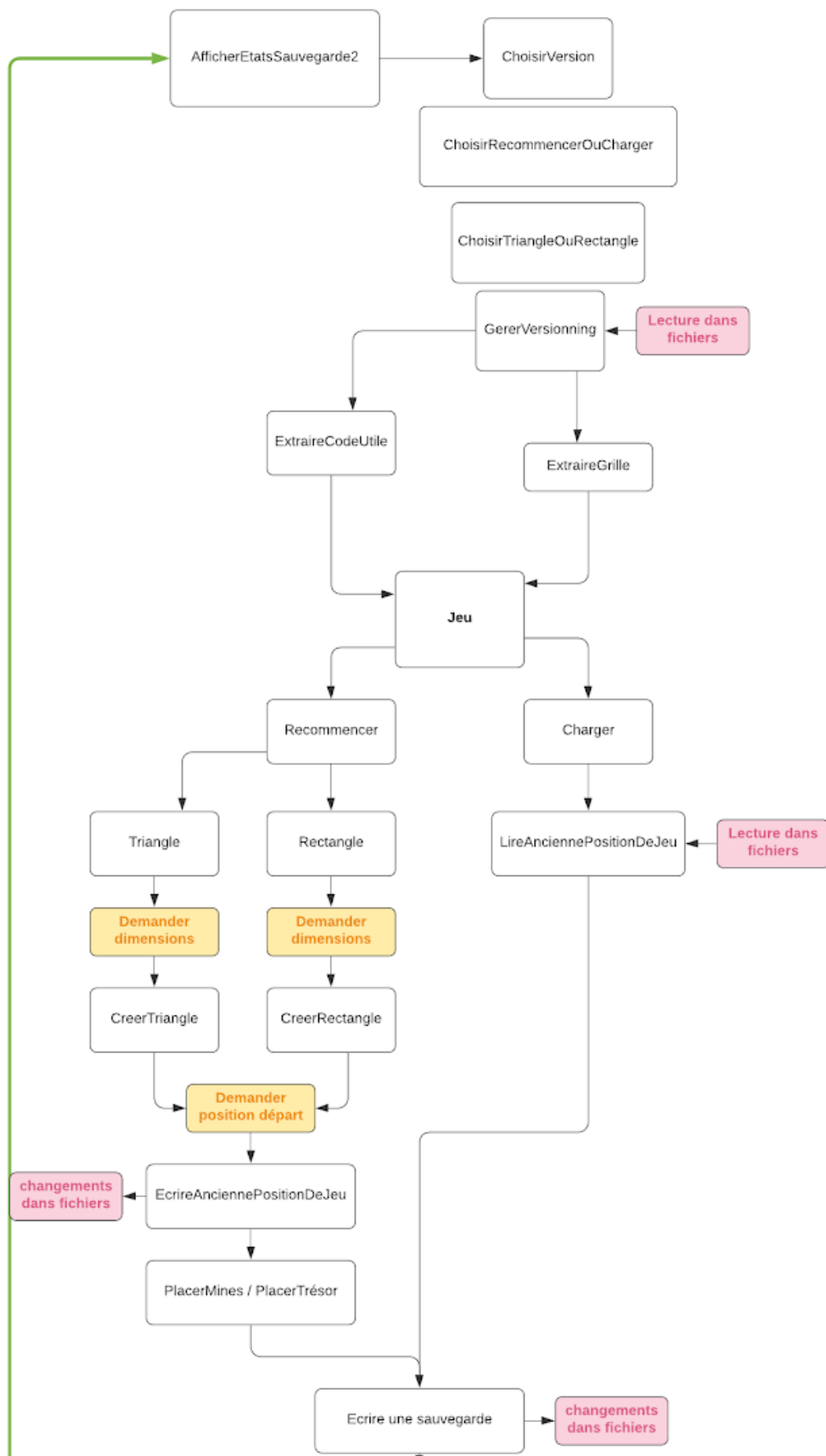
## Description d'une partie

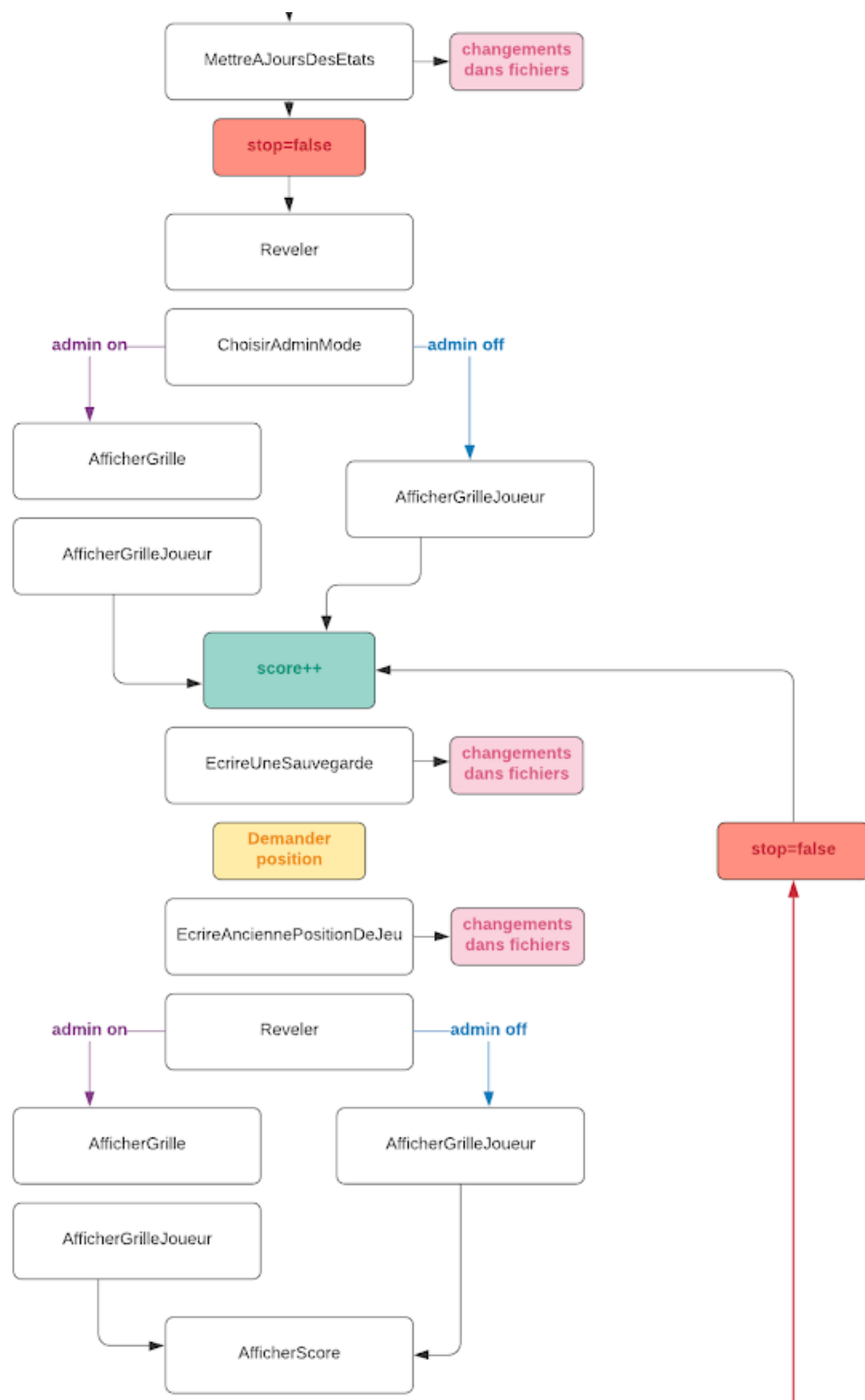
Voici plusieurs diagrammes présentant le déroulement d'une partie.

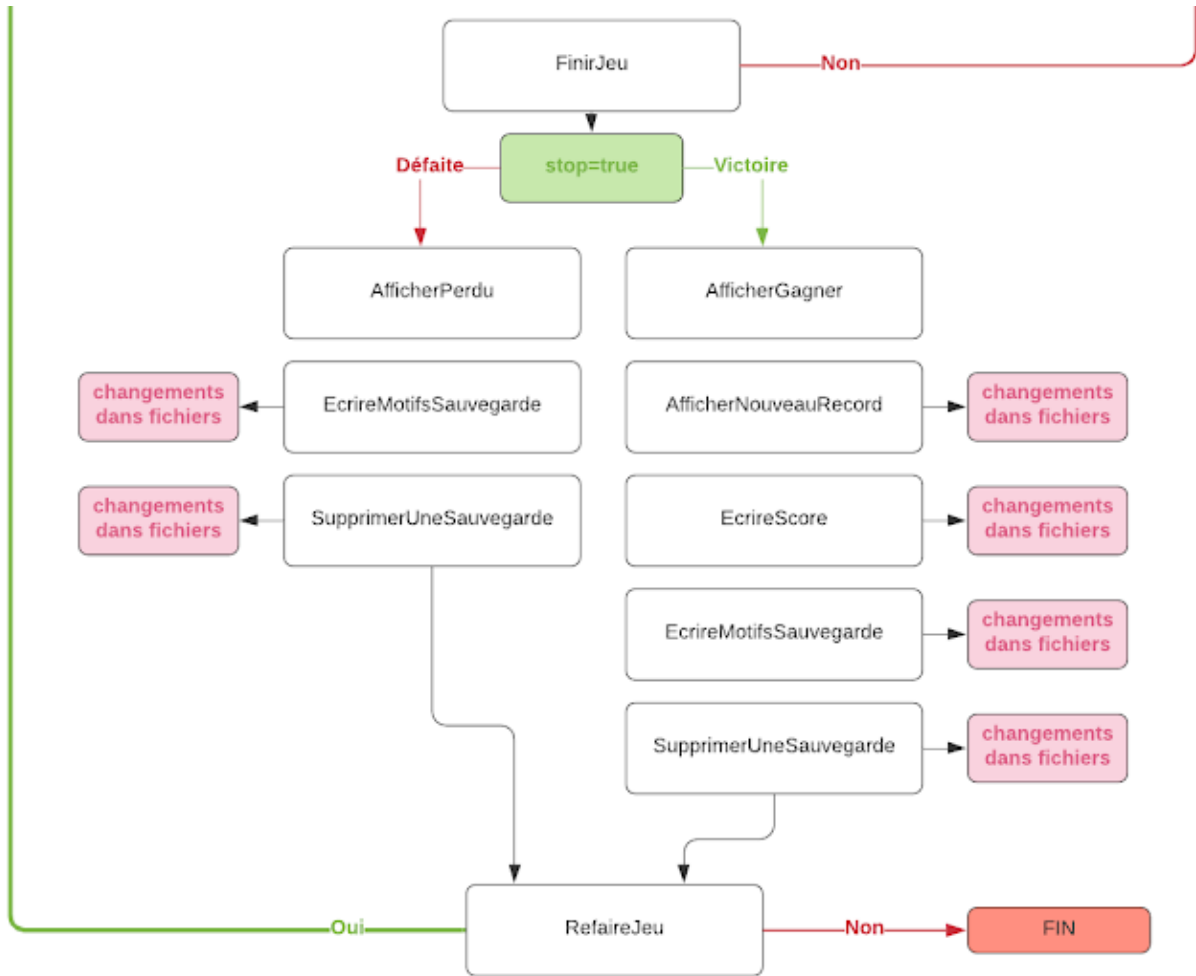
A noter que la fonction `AffichageIntro()` et `AfficherEtatsSauvegarde2()` s'enchaînent.

**Remarque :** Possibilité de voir le deuxième diagramme en une seule fois dans le fichier annexe "Diagramme.pdf"











# Répertoire des fonctions utilisées

**Remarque préalable :** Certains paramètres utilisés dans les fonctions ne sont pas obligatoires à l'exécution des fonctions. Ils pourraient être récupérés d'une autre manière. Cependant, les placer en paramètre permet d'éviter une redondance d'appel à certaines fonctions. Par exemple, les arguments hauteur, largeur ou encore nombre de cases qui permettent d'éviter des appels de `GetLength` ou le recalcule des cases utiles à chaque fois.

## Main

On initialise les variables utilisées. On affiche le choix du menu. Si on choisit, le menu "jeu". On teste tous les outils de versionning tels que la gestion du nom d'utilisateur via `GererUsername()`, et `TesterVersionning()`. De plus, on récupère les états de chaque sauvegarde. Ensuite, on lance le jeu avec le choix des sauvegardes tant que l'utilisateur n'a pas décidé de s'arrêter.

## Jeu

```
Jouer(ref string etat, string saveTxt, string[][] grille, string motifName, string[] etatsTab, string choix, int nb)
```

**Entrée :**

- ref string etat (état de la sauvegarde utilisée)
- string saveTxt (nom de la save, ex : save1.txt)
- string[][] grille (grille de jeu)
- string motifName (nom du motif choisi)
- string[] etatsTab (tableau des états)
- string choix (Recommencer ou Charger)
- int nb (Numéro de la sauvegarde)

**Sortie :** void

Si on a choisi `"Recommencer"`, on détecte le choix du motif et on construit la grille en conséquence. On demande aussi la première position, on la sauvegarde, et on initialise les bombes ainsi que les trésors.

Si on a choisi `"Charger"`, on calcule le nombre de mines.

Puis, on compte le nombre de cases utiles en fonction du motif. On écrit une sauvegarde. On révèle les cases, on demande le choix du mode admin ou non. Puis, on a une boucle while qui s'arrête lorsque :

- On a perdu
- On a gagné

Lors que chaque itération de la boucle while, on demande la position du prochain coup au joueur, on sauvegarde cette position, on révèle les cases et on affiche la grille en fonction de choix concernant l'admin Mode.

Lors de l'arrêt de la boucle while, selon si on a gagné ou perdu, l'affichage diffère. Si on gagne, on enregistre le score. Dans les deux cas, on supprime la sauvegarde.

## Gestion des pointeurs

### Choix de la save

`ChoisirNumeroSave()`

**Entrée :** Aucune  
**Sortie :** void

Fonction de choix de la save utilisée lors de la partie. Permet de choisir entre les 4 avec un curseur. On change d'état à la lecture d'une touche (droite ou gauche) et on affiche le numéro correspondant en surbrillance avec la fonction associée au numéro.

La boucle while s'arrête lors d'un appui sur entrée.

Renvoie un string :

- "1" si on appuie sur "Enter" lorsque le curseur est sur le 1.
- "2" si on appuie sur "Enter" lorsque le curseur est sur le 2.
- etc ...

`AfficherNumeroSave1/2/3/4()`

**Entrée :** Aucune  
**Sortie :** void

Affiche l'écran de choix de la save avec le numéro dans le nom en surbrillance.

### Choix Recommencer ou Charger

Ici on fait exactement la même chose que la fonction précédente mais pour choisir si on veut charger une sauvegarde ou l'écraser et recommencer. Les fonctions utilisées sont `ChoisirRecommencerOuCharger()`, `AfficherRecommencer()` et `AfficherCharger()`.

### Choix du motif

Ici on fait exactement la même chose que la fonction précédente mais pour choisir le motif entre rectangle et triangle. Les fonctions utilisées sont `ChoisirTriangleOuRectangle()`, `AfficherRectangle()` et `AfficherTriangle()`.

### Choix du mode ADMIN

Ici on fait exactement la même chose que les fonctions précédentes mais pour choisir le mode admin ou non. Les fonctions utilisées sont `ChoisirAdminMode()`, `AfficherAdminOui()` et `AfficherAdminNon()`.

### Gestion de l'accueil

`EcranAccueil()`

**Entrée :** Aucune  
**Sortie :** string

Fonction d'accueil lors du lancement du jeu qui permet de choisir entre un curseur sur le menu statistique et le curseur sur le menu de jeu.

La boucle while s'arrête lors d'un appui sur entrée.

Renvoie un string :

- `"stat"` si on appuie sur "Enter" lorsque le curseur est sur le menu statistique.
- `"jeu"` si on appuie sur "Enter" lorsque le curseur est sur le menu jeu.

```
AfficherMenuJouer()
```

**Entrée :** Aucune  
**Sortie :** void

Affiche l'écran d'accueil lors du lancement du jeu. Lors du lancement de cette fonction, "Jouer" est en surbrillance, tandis que "Statistiques" non.

```
AfficherMenuStatistique()
```

**Entrée :** Aucune  
**Sortie :** void

Affiche l'écran d'accueil lors du lancement du jeu. Lors du lancement de cette fonction, "Statistiques" est en surbrillance, tandis que "Jeu" non.

### Refaire une partie

Ici on fait exactement la même chose que les fonctions précédentes mais pour choisir de refaire une partie ou non. Les fonctions utilisées sont `ChoixRefairePartie()`, `AfficherOui()` et `AfficherNon()`.

### Menu Statistiques

```
AfficherStatistiquesDeJeu()
```

**Entrée :** Aucune  
**Sortie :** void

Possède deux affichages différents selon si l'utilisateur a déjà gagné une partie ou non. Pour cela, on vérifie l'existence du fichier `ScoreSaver.txt` qui contient les sauvegardes des différents scores gagnés lors de précédentes parties.

**1er cas :** Si on n'a encore gagné aucune partie.

On affiche juste son nom d'utilisateur, ainsi que le fait qu'aucune partie n'a été gagnée.

**2ème cas :** Au moins une partie a déjà été gagnée.

On récupère les scores obtenues lors des victoires dans le fichier ScoreSaver.txt via

```
string[] scores Tableau = File.ReadAllLines("ScoreSaver.txt");
```

On calcule la moyenne du tableau, ainsi que le score minimum.

On effectue ensuite un histogramme des différentes notes, entre 0 et 100, avec une séparation des notes tous les 10 ainsi qu'une légende. Cf Sujet TP Informatique

Noté, exercice 3.

## Fonctions d'affichage

### Affichage Règle

```
AffichageRegle()
```

**Entrée :** Aucune

**Sortie :** void

Affiche dans la console le tutoriel avec l'explication du jeu, la signification des différentes cases et couleurs.

Utilise la fonction `ConsoleDraw()`

### Centrer Texte

```
CentrerTxt(string mot, int ligne)
```

**Entrée :** string mot (Mot à centrer)

int ligne (Ligne dans la console où centrer le mot)

**Sortie :** void

Prends un mot et le centre dans la console sur la ligne choisie.

### Afficher Introduction

```
AfficherIntro()
```

**Entrée :** Aucune

**Sortie :** void

Affiche l'animation qui défile avec marqué "Bonjour et bienvenue"

### ASCII ART

```
DessinerConsole(IEnumerable<string> lines, int x, int y)
```

**Entrée :** IEnumerable<string> lines (Le message à afficher)

Int x (La position en abscisse)

Int y (La position en ordonnée = la ligne choisie)

Chasse au trésor - Justifications techniques

**Sortie :** void

Permet d'afficher un message de plusieurs lignes dans la console à une certaine position x et une certaine position y.

```
Exploser()
```

**Entrée :** Aucune

**Sortie :** void

Créer une animation d'explosion en utilisant `ConsoleDraw` et `System.Threading.Thread.Sleep(10)`.

### Affichage des différents types de cases

```
AfficherVide()
```

**Entrée :** Aucune

**Sortie :** void

Affiche une case vide dans la grille.

```
AfficherMines()
```

**Entrée :** Aucune

**Sortie :** void

Affiche une case Mine dans la grille .

```
AfficherTrésor()
```

**Entrée :** Aucune


**Sortie :** void

Affiche une case Trésor dans la grille .

```
AfficherJoueur(string x)
```

**Entrée :** string x (décompte)

**Sortie :** void

Affiche une case Joueur dans la grille .

```
AfficherVisité()
```

**Entrée :** Aucune


**Sortie :** void

Affiche une case Visitée dans la grille .

```
AfficherNonUtile()
```

**Entrée :** Aucune


**Sortie :** void

Affiche une case non utile dans la grille .

```
AfficherVisitéTresor()
```

**Entrée :** Aucune


**Sortie :** void

Affiche une case Trésor visitée dans la grille .

```
AfficherNumero(string x)
```

**Entrée :** string x = décompte

**Sortie :** void

Affiche une case Décompte dans la grille .

### Affichage concernant le score

```
AfficherScore(int score)
```

**Entrée :** int score (Score à afficher)

**Sortie :** void

Utilise la fonction `ConsoleDraw()` et va chercher parmi un répertoire de 55 fichiers un score dessiné en ASCII ART.

**Remarque :** On ne peut pas afficher un score supérieur à 55 avec cette méthode. Cependant, on atteint rarement ce type de score lors d'une partie de démineur.

```
AfficherNouveauRecord(int score)
```

**Entrée :** int score (Score à afficher)

**Sortie :** void

Deux cas sont possibles :

- Si aucun score n'est renseigné, ie le fichier `ScoreSaver.txt` n'existe pas, alors on a automatiquement un nouveau record, car c'est le premier record inscrit. On affiche alors dans la console que c'est un nouveau record.
- Sinon on regarde si il est inférieur aux scores déjà inscrits. Si c'est le cas, on affiche dans la console que c'est un nouveau record.

### Affichage de grilles

```
AfficherGrille(string[][] tab, string motifName, int hauteur, int largeur)
```

**Entrée :** string[][] tab (Grille à afficher)  
string motifName (Nom du motif)  
int hauteur (Hauteur de la grille)  
int largeur (Largeur de la grille)

**Sortie :** void

On parcourt la grille et on regarde la valeur de chacune des cases. En fonction de la valeur, utilise une fonction d'affichage de cases pertinentes vu précédemment. Lors de l'appel de

cette fonction, on a une vue admin de la grille, c'est -à -dire qu'on voit les bombes, ainsi que les trésors.

```
AfficherGrilleJoueur(string[][] tab, string motifName, int hauteur, int largeur)
```

**Entrée :** string[][] tab (Grille à afficher)  
string motifName (Nom du motif)  
int hauteur (Hauteur de la grille)  
int largeur (Largeur de la grille)

**Sortie :** void

On parcourt la grille et on regarde la valeur de chacune des cases. En fonction de la valeur, utilise une fonction d'affichage de cases pertinentes vu précédemment. Lors de l'appel de cette fonction, on a une vue joueur de la grille, c'est -à -dire qu'on ne voit pas les bombes, ni les trésors.

### Affichage perdre / gagner

```
AfficherPerdu()
```

**Entrée :** Aucune

**Sortie :** void

Afficher une animation d'explosion à l'aide de Explosion()  
Puis afficher le fait que le joueur a perdu.

```
AfficherGagner(int score)
```

**Entrée :** int score (Score du joueur lors de la victoire)

**Sortie :** void

Afficher une animation d'explosion à l'aide de Explosion()  
Puis afficher le fait que le joueur a gagné, ainsi que son score via la fonction AfficherScore().



### Affichage états des sauvegardes

```
AfficherEtatSauvegarde(string etat, string numero)
```

**Entrée :** string etat  
string numero

**Sortie:** void

Affiche l'état d'une sauvegarde en fonction de son numéro et si `etat=="Vide"` ou non.  
Deux cas :

- La sauvegarde est vide : On affiche un rectangle vert  ainsi que le fait qu'elle soit vide
- La sauvegarde contient des valeurs : On affiche un rectangle rouge  ainsi que le fait qu'elle contienne effectivement des valeurs.

```
AfficherEtatSauvegarde2(string etat1, string etat2, string etat3, string etat4)
```

**Entrée :**        string etat1  
                 string etat2  
                 string etat3  
                 string etat4

**Sortie:**        void

Appelle AfficherEtatSauvegarde pour toutes les sauvegardes.

## Fonctions de création de motifs

```
CreerRectangle(int hauteur, int largeur)
```

**Entrée :**        int hauteur (La hauteur du rectangle)  
                 int largeur (La largeur du rectangle)

**Sortie :**        string[][] (La grille de jeu)

Créer une matrice de type `string[][]` correspondant à la matrice de jeu. Elle est de taille  $hauteur \times largeur$ . De plus, elle est remplie de cases correspondant aux cases non visitées "███".

```
CreerTriangle(int hauteur)
```

**Entrée :**        int hauteur (La hauteur du rectangle)

**Sortie :**        string[][] (La grille de jeu)

Créer une matrice de type `string[][]` correspondant à la matrice de jeu. Elle est de taille  $hauteur \times (2hauteur + 1)$ . Le motif triangle est centré dans la matrice et est rempli de case non visité "███". Le reste de la matrice est rempli de case non utiles au jeu en lui-même, mais utiles au bon fonctionnement du jeu (voisinage etc...).

On remplit le triangle en le remplissant par sa colonne centrale (chaque ligne est constituée d'un nombre impaire d'éléments) d'indice  $j = hauteur$ .

On remplit d'abord toutes les cases d'indice  $(i, hauteur)$ ,  $\forall i \in [1, hauteur - 1]$  puis on remplit, de façon symétrique par rapport à l'axe  $y = hauteur$  les cases.

On les remplit en remarquant que, pour la ligne  $i$ , on doit remplir  $i - 1$  cases sur la droite de l'axe de symétrie et  $i - 1$  sur la gauche.

## Fonctions de positionnement des mines et des trésors

```
PositionnerMines(string[][] motif, int hauteur, int largeur, int nbCases, Random rng, int ligneJ, int colonneJ)
```

**Entrée :**        string[][] motif (Carte de jeu)  
                 int hauteur (Hauteur du rectangle correspondant à la matrice de jeu)  
                 int largeur (Largeur du rectangle correspondant à la matrice de jeu)  
                 int nbCases (Nombre de cases de la matrice de jeu)  
                 Random rng (Permet de générer un nombre aléatoire de mines)  
                 int ligneJ (ligne dans laquelle le joueur s'est positionné)  
                 int colonneJ (colonne dans laquelle le joueur s'est positionné)

**Sortie :**        int (Renvoie le nombre de mines placées)



Choisi un nombre aléatoire de mines (entre nombre de ligne/2 et nombre de cases /2). Puis place chaque mine dans la grille en respectant les conditions suivantes (boucle while avec `bool test` qui passe en `True` lorsque les conditions sont vérifiées):

- On est sur une case vide.
- On est pas sur la case choisie par le joueur au départ `(ligne!=ligneJ-1) && (colonne!=colonneJ-1)`
- On est pas déjà sur une bombe.

Si les conditions sont vérifiées, `test` devient `True` et on place la mine dans la grille. Sinon, on reprends au hasard une ligne et une colonne et on reteste.

**Remarque :** On n'a pas besoin de tester le fait qu'on soit sur un trésor ou non. En effet, l'appel de `PositionnerMines` est fait avant celui de `PositionnerTrésor`. Ainsi, les trésors ne sont pas encore présents dans la grille, et on ne peut donc pas se trouver sur un trésor lors du placement d'une mine.

```
PositionnerTrésor(string[][] motif, int hauteur, int largeur, int nbCases, Random rng, int ligneJ, int colonneJ)
```

**Entrée :** string[][] motif (Carte de jeu)  
int hauteur (Hauteur du rectangle correspondant à la matrice de jeu)  
int largeur (Largeur du rectangle correspondant à la matrice de jeu)  
int nbCases (Nombre de cases de la matrice de jeu)  
Random rng (Permet de générer un nombre aléatoire de mines)  
int ligneJ (ligne dans laquelle le joueur s'est positionné)  
int colonneJ (colonne dans laquelle le joueur s'est positionné)

**Sortie :** int (Renvoie le nombre de mines placées)

Choisi un nombre aléatoire de trésors (entre 1 et 3). Puis place chaque trésor dans la grille en respectant les conditions suivantes (boucle while avec `bool test` qui passe en `True` lorsque les conditions sont vérifiées):

- On est sur une case vide.
- On est pas sur la case choisie par le joueur au départ `(ligne!=ligneJ-1) && (colonne!=colonneJ-1)`
- On est pas déjà sur une bombe.
- On est pas déjà sur un trésor.

Si les conditions sont vérifiées, `test` devient `True` et on place le trésor dans la grille. Sinon, on reprends au hasard une ligne et une colonne et on reteste

### Fonctions utiles au Jeu

```
FinirJeu(string[][] motif, bool perdu, int nbCases, int hauteur, int largeur, int nbMines)
```

**Entrée :** string[][] motif (Grille de jeu)  
bool perdu (True si on a joué sur une bombe, False sinon)  
int nbCases (Nombre de cases de la grille)  
int largeur (largeur de la grille)  
int hauteur (hauteur de la grille)

Int nbMines (Nombres de mines de la grille)

**Sortie :** string ("Perdu" / "Gagné" / "Pas Fini")

Si on a perdu (`perdu==true`) alors on renvoie "Perdu".

Sinon :

- On initialise un compteur à 0.
- On parcourt la grille et on incrémente le compteur à chaque fois qu'on croise une case qui n'est pas une mine, ni non visitée, ni un trésor.
- On vérifie ensuite si le nombre de cases comptées correspond bien au nombre totale de cases à visiter ( $= nbCases - nbMines$ )
  - Si oui, on renvoie "Gagné"
  - Sinon, le jeu n'est pas fini et on renvoie "Pas Fini".

```
Reveler(string[][] motif, int nbCases, int hauteur, int largeur, int  
ligneJu, int colonneJu, ref bool perdu)
```

**Entrée :** string[][] motif (Grille de jeu)  
bool perdu (True si on a joué sur une bombe, False sinon)  
int nbCases (Nombre de cases de la grille)  
int largeur (largeur de la grille)  
int hauteur (hauteur de la grille)  
int ligneJu (ligne de la case choisie par le joueur)  
int colonneJu (colonne de la case choisie par le joueur)

**Sortie :** void

On regarde sur quelle case de la grille le joueur a joué.

Si c'est une mine, on a perdu (`perdu=true`).

Si c'est un trésor, la case devient un trésor visité (" V ")

Sinon, on regarde chez les voisins de la case, et on calcule le décompte selon les règles choisies (+1 pour une mine, +2 pour un trésor).

Si le décompte est égale à 0 (ie dans les 8 voisins, nous n'avons ni Mine, ni Trésor) alors dans ce cas on effectue la fonction révélée dans tous les voisins (sauf ceux qui appartiennent à la frontière) : Récursivité, on fait l'appelle sur tous les voisins non frontières.

Sinon, on calcule le décompte et on l'affiche sur la case.

```
ListerVoisins(string[][] motif, int hauteur, int largeur, int ligneJ,  
int colonneJ)
```

**Entrée :** string[][] motif (Grille de jeu)  
int largeur (largeur de la grille)  
int hauteur (hauteur de la grille)  
int ligneJ (ligne de la case choisie par le joueur)  
int colonneJ (colonne de la case choisie par le joueur)

**Sortie :** int[,] voisinsTab (tableau de taille 8x2 comportant les 8 voisins de la case)

On prend les huit voisins de la case sélectionnée par le joueur. Si jamais un des voisins est une case "frontière" (`motif[x][y]=="XXX"`) alors dans ce cas, on change les coordonnées dans voisinsTab par (− 999, − 999) (*cf formalisme*)

```
CalculerDecompte(string[][] motif, int nbCases, int hauteur, int largeur, int ligneJu, int colonneJu, ref bool perdu)
```

**Entrée :** string[][] motif (Grille de jeu)  
bool perdu (True si on a joué sur une bombe, False sinon)  
int nbCases (Nombre de cases de la grille)  
int largeur (largeur de la grille)  
int hauteur (hauteur de la grille)  
int ligneJu (ligne de la case choisie par le joueur)  
int colonneJu (colonne de la case choisie par le joueur)

**Sortie :** int

On calcule le décompte selon les règles choisies (+1 pour une mine, +2 pour un trésor) en regardant les voisins de la case de coordonnées (*ligneJu*, *colonneJu*).

## Versioning

```
LireAnciennePositionDeJeu(int numero)
```

**Entrée :** int numero  
**Sortie :** string[] positions

Renvoie les coordonnées (*x*, *y*) de la dernière position jouée, de la save d'indice "numéro"

```
EcrireAnciennePositionDeJeu(int x, int y, int numero)
```

**Entrée :** int numero (numéro de la save)  
int x (position en x)  
int y (position en y)

**Sortie :** void

Permet de sauvegarder à chaque tour où se trouve le dernier coup joué par le joueur. Sauvegarde dans le fichier OldPositionnumero.txt

```
GererVersionning(string choix, string saveTxt, ref string etat, string[] etatsTab)
```

**Entrée :** string choix (Charger une save ou la remplacer)  
string saveTxt (nom de la save, exemple "save1.txt")  
string etat (etat de la save)  
string[] etatsTab (tableau des états des saves)

**Sortie :** string[][] grilleUU

Renvoie la grilleUU. Lis juste si on choisit de charger et écrase les données de l'ancienne save avec SupprimerUneSauvegarde(...) si on souhaite recommencer.

```
GererUsername()
```

**Entrée :** Aucune  
**Sortie :** void

Gère la notion Username, et créer un fichier `NomUser.txt` si on n'a pas encore renseigné dans nom d'utilisateur, avec à l'intérieur une ligne contenant le nom d'utilisateur renseigné.

```
TesterVersionning()
```

**Entrée :** Aucune

**Sortie :** void

Test si tous les fichiers de versionning sont présents. Dans le cas inverse, les génère avec des paramètres considérés par défauts, explicités dans le formalisme.

```
SupprimerUneSauvegarde(string saveTxt, string[] etatsTab)
```

**Entrée :** string saveTxt (nom du fichier de sauvegarde à supprimer  
exemple : save1.txt)

string[] etatsTab (tableau des états des saves)

**Sortie :** string

Renvoie `"Vide"` et supprime la sauvegarde choisie. De plus, affiche `"Sauvegarde supprimée !"` dans la console. Et on recrée une sauvegarde vide, et on met à jour les états.

```
EcrireUneSauvegarde(string[][] grille, int score, string etat, int  
hauteur, int largeur, string saveTxt)
```

**Entrée :** string[][] grille (grille de jeu)  
int score (Score actuel)  
string etat (état de la sauvegarde)  
int hauteur (hauteur de la grille)  
int largeur (largeur de la grille)  
string saveTxt (nom du fichier de sauvegarde à supprimer  
exemple : save1.txt)

**Sortie :** void

Écris une sauvegarde de la partie dans le fichier sélectionné.

```
EcrireScore(int score)
```

**Entrée :** int score (Score actuel)

**Sortie :** void

Rajoute un score dans le document `scoreSAVER.txt`

```
EcrireMotifsSauvegarde(int numeroSauvegarde, string nomMotif)
```

**Entrée :** int numeroSauvegarde (numero de la sauvegarde)  
string nomMotif (nom du motif à enregistrer)

**Sortie :** void

Sauvegarde dans `motifsSave.txt` les motifs de chaque save. Par défaut : `"rectangle"`

```
LireSauvegarde(string saveTxt)
```

**Entrée :** string saveTxt (nom du fichier de sauvegarde à supprimer)

**Sortie :**           exemple : save1.txt)  
                  string[][]

Lis le fichier txt saveTxt contenant la sauvegarde sélectionnée et renvoie la grilleUU avec toutes les informations.

```
ExtraireGrille(string[][] grilleUU)
```

**Entrée :**           string[][] grilleUU (cf formalisme)  
**Sortie :**           string[][] grille

Extrait la grille de jeu de GrilleUU.

```
ExtraireCodeUtile(string[][] grilleUU)
```

**Entrée :**           string[][] grilleUU (cf formalisme)  
**Sortie :**           string[] codeU

Extrait le code information de la grilleUU (4 première lignes du .txt) et le renvoie sous forme de tableau de dimension 4.

```
MettreAJourDesEtats(string[] etatsTab)
```

**Entrée :**           string[] etatsTab (tableau des états des saves)  
**Sortie :**           void

Met à jour les états suivant un tableau, dans le fichier **etatsSaver.txt**

```
EcrireUneLigne(string valeur, string saveTxt)
```

**Entrée :**           string valeur (texte à inscrire dans le .txt)  
                  string saveTxt (nom du fichier de sauvegarde à supprimer  
                  exemple : save1.txt)

**Sortie :**           void

Écrire une ligne dans un fichier texte saveTxt, avec la valeur **"valeur"**