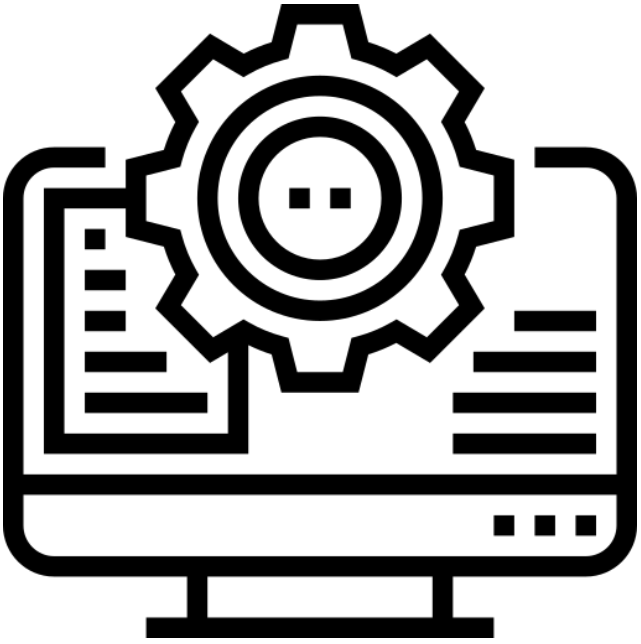
20 Janvier 2020



DOCUMENT DE JUSTIFICATIONS TECHNIQUES

INTRODUCTION À LA PROGRAMMATION

BINET COLINE - PERRIER ALBAN

BORDEAUX INP

ENSC 1A – Groupe 1

SOMMAIRE

Légende du document :

En **magenta** : les fonctions et procédures

En **bleu** : les variables

Présentation

Le programme comporte 28 fonctions et procédures et 9 variables globales.
Nous avons beaucoup de fonctions du fait de notre volonté d'avoir un programme facilement maintenable et réutilisable.
Les variables globales sont des variables utilisées dans l'ensemble du programme ou qu'il était difficile de donner en paramètre à une fonction.
Pour faciliter les diverses vérifications en jeu, chaque entrée clavier de même que l'ensemble des mots des dictionnaires sont mis en majuscule et sans accents.

Pour la gestion de projet, nous avons utilisé git pour pouvoir travailler individuellement sur le projet et, au besoin, restaurer d'anciennes versions du code.
Projet consultable à cette adresse : https://github.com/aperrier004/IPROG_Motus.git
Les dictionnaires " Animaux ", " Noël ", "Couleurs" et "Métiers" sont des dictionnaires composés uniquement de mot reliés à leur thème.
Les musiques viennent du jeu télévisé Motus. Certaines ont été modifiées par le logiciel Audacity.

Les fichiers clés

Le programme utilise différents fichiers. Il y a un dossier de dictionnaires(“Dicos”) et un autre de sons(“Music”).

Le premier dossier contient 5 dictionnaires différents au format txt, ainsi que le dictionnaire réduit que l’on va utiliser pour sélectionner un mot pour le joueur.
Ce dictionnaire réduit se crée grâce à la fonction **reduireFicher**(int **tailleMot**) (détaillé plus tard dans le document). Il contient tous les mots en majuscules qui correspondent au nombre de lettres que le programme lui donne.

Un autre fichier qui se crée lors de l'exécution du programme est le fichier texte “sauvegarde”, qui se place dans /bin/Debug par défaut. Il contient toutes les données des parties enregistrées. Il peut être effacé (réinitialiser des sauvegardes).

L’autre dossier est celui des sons qui sont joués au cours de l'exécution du programme.

Ces fichiers ont été ajoutés en guise d’amélioration du projet à réaliser.

Les variables

Le programme utilise des variables globales d'une part pour maintenir le code à long terme, pour par exemple modifier rapidement le nom des fichiers utilisés ou créés. Et d'autre part pour simplifier le code, en déclarant une seule fois un objet Random, afin de pouvoir l'utiliser plusieurs fois dans des fonctions ensuite. On les utilise aussi pour simplifier notre mode Défi, afin de pouvoir toujours utiliser les même fonctions de jeu, tout en sauvegardant les données que l'on souhaite avoir au cours du mode.

Le format des données choisi dans le programme est ce qui, a notre échelle, semble le plus optimisé et adapté à l'utilisation (par exemple utiliser un int plutôt qu'un double ou un string pour récupérer le numéro de l'option saisie par l'utilisateur).
De manière similaire, nous avons choisi d'utiliser un tableau à 2 dimension de char (plutôt que de string) pour la grille qui contient les propositions du joueur. Ainsi, chaque lettre correspond à une case du tableau char[,] **grille**.
Nous avons créé un tableau d'entiers à deux dimensions de la même taille qui correspond à la couleur que doit avoir ces lettres contenues dans le tableau **grille**. En effet, selon le code couleur que l'on utilise, on va pouvoir remplir cette **grilleCouleur** avec un numéro qui correspond à une lettre dans **grille**. Ainsi, pour l'affichage de la grille, en plus de parcourir les différentes lettres, on va parcourir la **grilleCouleur** et changer la couleur de fond d'affichage selon le numéro correspondant à la lettre. (voir la fonction **AfficherGrille(..)**).

Dans le programme, nous utilisons souvent des variables bool afin de contrôler un état, que ce soit pour la saisie de l'utilisateur et vérifier qu'elle soit bonne, ou bien pour indiquer si la partie est terminée ou si le jeu est en mode défi ou non.

Le programme

Dans cette partie sera développé le fonctionnement général ou détaillé du code source du projet.

Éléments récurrents du programme

Dans le code source, nous reprenons plusieurs fois le même système de boucles. En effet, pour vérifier la saisie attendue d'un utilisateur, nous utilisons une variable bool **saisieIsValid** qui est créée à **false**, et changé à **true** seulement lorsque l'utilisateur entre en saisie ce que l'on attend (par exemple : on commence par vérifier si le type est bien un int d'abord, et ensuite on regarde s'il est compris dans l'intervalle recherché).

```
int caseSwitch=0;
while (caseSwitch !=4)
{
    bool saisieIsValid = false;
    while (!saisieIsValid)
    {
        Console.WriteLine("\nSaisir le numero de l'option que vous souhaitez");
        string saisie = Console.ReadLine();
        if (int.TryParse(saisie, out caseSwitch))
        {
            caseSwitch = int.Parse(saisie);
            if (caseSwitch > 0 && caseSwitch < 5)
            {
                saisieIsValid = true;
            }
        }
    }
```

Figure 1 - code permettant de vérifier une saisie de l'utilisateur

Cela nous permet, en plus de contrôler la saisie, de s'assurer que l'on obtienne un cas géré par notre programme.

Un autre élément que l'on utilise souvent dans le code source est la boucle *switch* qui nous permet, dans une majorité des cas, d'appeler une fonction selon le choix de l'utilisateur.C'est ainsi que fonctionne notre système de menu et de propositions d'actions pour l'utilisateur.

```
// Permet d'appeler la fonction permettant de realiser l'action que l'utilisateur souhaite
switch (caseSwitch)
{
    case 1: // Reset sauvegarde
        ResetFichier(fichierSauvegarde);
        break;
    case 2: // modifier indice
        ModifierIndice();
        break;
    case 3: // Change de dico
        ModifierDico();
        break;
    case 4: // Retour au menu
        AfficherMenu();
        break;
}
```

Figure 2 - exemple de switch

Le dernier élément qui revient souvent au cours du programme est la fonction **toUpper()**, que l'on utilise afin de normaliser et standardiser la chaîne sur laquelle la fonction agit. Ainsi, toutes nos vérifications se font sur des majuscules, quelque soit ce qu'entre l'utilisateur, ou le format des fichiers dictionnaires.

Fonctions de début du programme

*void **Main**(string[] args)*

Le main permet d'afficher les crédits du jeu ainsi que de lancer le générique de celui-ci par le biais de la fonction **JouerUnSonEnBoucle**(..)
Le joueur doit taper sur n'importe quelle touche pour que le main affiche le menu (procédure **AfficherMenu**()).

*void **AfficherMenu**()*

La procédure **AfficherMenu**() permet de faire le lien entre les différentes fonctionnalités du jeu :

- Lancer une nouvelle partie
- Accéder aux statistiques
- Accéder aux options
- Quitter le jeu

Pour sélectionner la fonctionnalité voulue , le joueur doit entrer, à l'aide du clavier, la valeur associée (1 pour "Nouvelle Partie", 2 pour "Statistiques", 3 pour "Options", 4 pour "Quitter"). La lecture d'entrée clavier est effectuée avec le raisonnement récurrent expliqué plus tôt. Tant que le joueur n'entre pas l'un des quatre chiffre attendu, il lui est demandé de recommencer.

Une fois une valeur correcte entrée, un *switch* permet d'appeler la fonction associée.
Dans le cas d'une Nouvelle Partie, la fonction appelée est **ParametrerPartie**().
Dans le cas des Statistiques, la fonction appelée est **AfficherStats**().
Dans le cas des Options, la fonction appelée est **AfficherOptions**().
Pour quitter le jeu, la fonction appelée est **QuitterJeu**().

Fonctions pour jouer une partie

Initialisation de la partie

void ***ParametrerPartie()***

La procédure **ParametrerPartie()** commence tout d'abord par afficher les règles du jeu.

Elle propose ensuite de sélectionner une difficulté.

5 difficultés sont prédéfinies. Le détail de chacune d'entre elles est expliqué.

Difficulté	**Nombre de lettres**	**Nombre maximum d'essais = X**	**Timer**	**Objectif**
Facile	6 ou 7	Autant que le nombre de lettres	Partie chronométrée	Trouver le mot en moins de X essais
Moyen	7, 8 ou 9	Autant que le nombre de lettres -2	Partie chronométrée	Trouver le mot en moins de X essais
Difficile	8, 9 ou 10	Autant que le nombre de lettres -3	Partie chronométrée	Trouver le mot en moins de X essais
Personnalisé	Au choix du joueur	Au choix du joueur	Chronométrée, et minuté au choix du joueur	Trouver le mot en moins de X essais (avec un temps minuté pour chaque proposition de mot)
Défi	Aléatoire	Aléatoire	Minuté	Trouver le plus de mots possible en un temps donné (5 minutes)

Le joueur peut choisir la difficulté qu'il souhaite en effectuant l'entrée clavier demandée.
Pour plus de sécurité, celle-ci s'effectue dans un *while.*
Une fois la difficulté choisie, le programme se charge de déterminer la valeur des variables paramétrant une partie (sauf pour la difficulté personnalisée):
tailleMot, **nbTentatives**, **dureeTour**.
Par défaut, **dureeTour** est à 1 jour : une façon de feindre un temps illimité pour chaque tour.

Dans le cas d'une partie personnalisée, ces variables sont définies dans la fonction **PersonnaliserDifficulte**(ref tailleMot, ref nbTentatives, ref dureeTour)

En fonction de ces paramètres (déterminés par le programme ou par le joueur), deux tableaux de dimension 2 sont créés :
grille : un tableau de **caractères** qui contiendra les propositions de mots du joueur
grilleCouleur : un tableau **d'entiers** qui déterminera les couleurs de chacune des lettres proposées par le joueur.
Ces deux tableaux sont de taille **tailleMot** x **nbTentatives**.

Une fois ces variables déterminées, le programme appelle la procédure **InitialiserPartie**(int tailleMot, int nbTentatives, out char[,] grille, out int[,] grilleCouleur)

Si la difficulté sélectionné correspond au mode défi, on va appeler la procédure **ModeDef**(char[,] grille, TimeSpan dureeMode, int[,] grilleCouleur, int difficulte) que l'on détaillera plus tard.
Sinon, on appelle la procédure **JouerPartie**(grille, dureeTour, grilleCouleur, difficulte).

*void **InitialiserPartie**(int tailleMot, int nbTentatives, out char[,] grille, out int[,] grilleCouleur)*

La procédure **InitialiserPartie**(..) permet d'effectuer le travail nécessaire à la bonne exécution d'une partie.

Il se charge d'appeler la procédure **réduireFichier**(**tailleMot**) qui créera le fichier listant tous les mots de taille **tailleMot** pour faciliter, par la suite, les différentes vérifications nécessaires au déroulement d'un tour de jeu.

Cette procédure se charge aussi de remplir la grille de caractères **grille** de '**.**' : chaque point représente une case vide du tableau. Par la suite, de vérifier si une cellule du tableau contient un point ou un autre caractère permettra de déterminer le nombre de tentatives restantes.

C'est également dans cette procédure qu'est instancié **grilleCouleur**.

*void **PersonnaliserDifficulte**(ref int tailleMot, ref int nbTentatives, ref TimeSpan tempsTour)*

La procédure **PersonnaliserDifficulte**(..) permet au joueur de renseigner lui-même les valeurs des paramètres d'une partie : la taille du mot, le nombre d'essais possibles ainsi que de préciser un éventuel minuteur pour chacun des tours.

Comme précédemment, les entrées clavier gérées éviter les erreurs de saisie.

Mode défi

*void **ModeDefi**(char[,] grille, TimeSpan dureeMode, int[,] grilleCouleur, int difficulte)*

La procédure **ModeDefi**(..) va permettre de changer la variable globale **modeDefi** à **true**. Elle va aussi lancer le chrono du mode et appeler la procédure **JouerPartie**(..). C'est à partir de la là que le mode défi se met en place.

*void **CreerPartieDefi**()*

Cette procédure est appelé lorsqu'une partie du mode défi s'est terminée (par un échec ou un succès de l'utilisateur). Elle permet de mettre en pause le jeu le temps de réaliser les affichages nécessaires et reprend lorsque l'utilisateur le souhaite pour enchaîner sur un autre mot à trouver (une autre "partie").
Elle va donc dans un premier temps remettre à zéro les variables **grille** et **grilleCouleur**, ainsi que générer de nouveaux paramètres de partie (**nbTentatives**, **tailleMot**) et appeller **InitialiserPartie**(..).
Cette fonction se termine en appelant **JouerPartie**(..), ce qui permet de faire l'enchaînement de mots à trouver pour le mode défi.

Déroulement de la partie

*void **JouerPartie**(char[,] grille, TimeSpan dureeTour, int[,] grilleCouleur, int difficulte)*

La procédure **JouerPartie**(..) permet de débuter une partie et de vérifier si une partie est finie, gagnée ou non et de réagir en conséquences.

Tout d'abord, la procédure appelle la fonction **GenererMotAleatoire**() pour déterminer le mot que le joueur devra trouver durant cette partie de Motus.
Ensuite, on vérifie que le joueur n'a pas paramétré, dans les options, que la lettre dévoilée soit à une position aléatoire du mot.

Si c'est le cas, on détermine la position de l'indice aléatoirement puis on le précise dans les deux grilles :

- on inscrit dans la première ligne de la **grille** de caractère la lettre dévoilée à la position générée aléatoirement
- on inscrit, à la même position, dans **grilleCouleur** la valeur **1** (1 étant pour une lettre bien placée).

Dans le cas d'un indice non aléatoire, le procédé est le même, mais la lettre dévoilée ainsi que le **1** seront inscrits dans la première cellule de chaque grille.

On affiche ensuite la grille de jeu via la procédure **AfficherGrille**(grille, grilleCouleur).
On lance en parallèle le chronomètre pour déterminer la durée de la partie.
Ce chronomètre tourne tant que le booléen **finPartie** est à **false** :

Dans une boucle *while* on appelle la fonction **JouerTour**(..) qui permet au joueur de jouer un tour et qui renvoie un booléen selon que le tour permette de finir ou non la partie. Si cette fonction renvoie **true** alors la partie est terminée.
Si la fonction renvoie **false** alors le joueur n'a pas trouvé le mot pendant le dit tour. On recommence.

Dans le cas d'une partie terminée, le programme vérifie si la partie n'était pas en mode défi, il arrête alors le chronomètre. Sinon, on a besoin que le chronomètre total continue afin de pouvoir vérifier depuis combien de temps l'utilisateur joue.
Puis on appelle la procédure **AfficherFinDePartie**().

*bool **JouerTour**(char[,] grille, TimeSpan dureeTour, int[,] grilleCouleur, string motInitial)*

La fonction **JouerTour**(..) commence tout d'abord par lancer un éventuel chronomètre (cas où le joueur a choisi la difficulté " Personnalisée " et qu'il a déterminé un minuteur pour chaque tour).
Si la partie est en mode défi, ce chronomètre est égal à la durée du temps du mode.

La grille ayant été affichée dans la procédure **JouerPartie**(..), le programme demande directement au joueur de proposer un mot.

L'entrée clavier est dans un *while* de sorte à pouvoir vérifier que le mot est de la bonne longueur, et si oui, que le mot existe (par le biais de la fonction VerifierMotDansDico(motPropose)).
Tant que le mot proposé par le joueur ne respecte pas ces conditions, il doit en inscrire un nouveau.

Une fois le mot valide rentré, l'éventuel chronomètre de tour se stoppe.
Si jamais le temps de chronomètre dépasse celui paramétré, le mot proposé est invalidé. Les caractères sont remplacés par des "**"**.

Le programme ajoute le mot dans la grille (à la dernière ligne remplie de "**.**") puis on appelle la fonction VerifierMot(..). Cette fonction retourne un booléen.
Si le booléen est à **false** : le mot proposé est injuste, la partie n'est pas terminée.
Si le booléen est à **true :** le mot proposé est juste, la partie est gagnée.

*void **AfficherGrille**(char[,] grille, int[,] grilleCouleur)*

La procédure AfficherGrille(..) permet d'afficher sur la console un tableau de deux dimensions dans lequel sont inscrites les différentes propositions de mots entrés par le joueur au cours de la partie. Chacune des lettres est affichée sur un fond de couleur :

- **Rouge** pour une lettre bien placée
- **Jaune** pour une lettre mal placée
- **Bleu** pour une lettre absente.

Pour procéder à cet affichage, la procédure reçoit en paramètre **grille** et **grilleCouleur** : deux tableaux associés : **grilleCouleur** est rempli de chiffres qui correspondent à la validité d'une lettre de **grille** à la même position.

On parcourt en parallèle ces deux tableaux pour afficher sur la console le tableau **grille** avec une couleur en fond correspondant au numéro de **grilleCouleur** (1 : rouge , -1 : bleu, 2 : jaune).

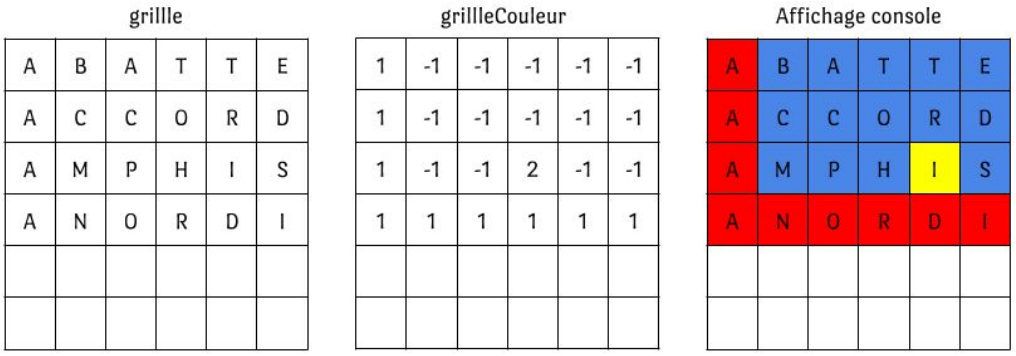


Figure 3 - exemple de combinaison de grille et grilleCouleur pour afficher le résultat au joueur

Gestion du mot

*bool **VerifierMot**(string motInitial, string motPropose, int tourActuel, int [,] grilleCouleur)*

La fonction **VerifierMot(..)** permet de vérifier si le dernier mot proposé par le joueur est juste ou non, mais également de déterminer la validité de chacune des lettres qui le compose. C'est cette fonction qui met à jour **grilleCouleur** pour informer le joueur de la validité des lettres.

La fonction commence par vérifier si le mot est exactement celui que le joueur doit trouver. Si oui, un son est émis pour le notifier, on utilise **Thread.Sleep(..)** afin de laisser le temps pour le son de se jouer correctement.
La ligne de **grilleCouleur** est mise à **1** pour montrer que toutes les lettres sont bien placées.
Le booléen **gagne** est mis à **true.**

Dans le cas contraire, le programme vérifie lettre par lettre si celle-ci existe, et si oui, si elle est bien placée. Un son est émis en fonction du résultat.

- Si la lettre existe et est bien placée, la cellule associée dans le tableau **grilleCouleur** prend la valeur **1.**
- Si la lettre existe et est mal placée, la cellule associée dans le tableau **grilleCouleur** prend la valeur **2.**
- Si la lettre n'existe pas, la cellule associée dans le tableau **grilleCouleur** prend la valeur **-1.**

La fonction retourne le booléen **gagne**.

*bool **VerifierMotDansDico**(string motPropose)*

La fonction **VerifierMotDansDico**(..) est employée pour vérifier que le mot entré par le joueur existe.

Par défaut, le booléen **motExistant** est à **false.**

Dans un *try .. catch* pour lever le risque d'une exception, le programme ouvre un StreamReader qui va parcourir le dictionnaire réduit (contenant uniquement les mots de taille X).

Chacun des mots est comparé au mot proposé par le joueur. Si les deux mots sont égaux, la fonction retourne **true**.

Si tout le fichier est parcouru sans qu'aucun mot ne soit identique, la fonction retourne **motExistant** (toujours à **false**).

*string **GenererMotAleatoire**(string fichierCible)*

Cette fonction a pour objectif de retourner un mot aléatoire du dictionnaire réduit.

Pour ce faire, le programme ouvre tout d'abord un StreamReader qui parcourt le fichier ligne par ligne pour compter le nombre total de mots.

Un chiffre aléatoire est déterminé (chiffre entre 0 et le nombre de mots).

Le StreamReader parcourt ensuite une seconde fois le fichier pour retourner le mot situé à la position déterminée aléatoirement.

Fin de la partie

*void **AfficherFinDePartie**(bool partieGagne, string motInitial, TimeSpan ts, int difficulte)*

Cette procédure est appelée par **JouerPartie**(..) lorsqu'une partie est terminée. Elle reçoit, notamment, entre paramètre le booléen **partieGagne**.
Selon que **partieGagne** vaille **true** ou **false**, la procédure afficher un texte différent et joue un son différent (par le biais de la fonction **JouerUnSon**(..).
Elle affiche également le temps écoulé le temps de la partie.
Cette fonction appelle la procédure **Sauvegarde**(..) pour enregistrer les statistiques de la partie.

Si l'on est en mode défi, on vérifie la partie a été gagnée, si oui alors on incrémente la variable globale **nbPartiesGagneesDefi**, et on incrémente la variable globale **nbMotsDefi** afin de pouvoir les afficher lorsqu'on le souhaite.
On vérifie aussi où en est le chronomètre, et s'il dépasse le temps du mode défi, dans ce cas on appelle la procédure **Rejouer**(), sinon on appelle la procédure **CreerPartieDefi**() qui va permettre de relancer un mot à trouver et de faire continuer le mode.

Sinon, on appelle la procédure **Rejouer**()

*void **Rejouer**()*

Cette procédure permet de demander à l'utilisateur s'il souhaite rejouer ou non.
Si oui, on appelle **ParametrerPartie**() qui va permettre de sélectionner une difficulté et de recommencer une partie rapidement.
Sinon, on appelle **AfficherMenu**().

Fonctions pour les statistiques

*void **Sauvegarder**(string motIni, TimeSpan tempsPartie, bool partieGagne, int difficulte)*

La procédure **Sauvegarder**(..) permet de créer (si besoin) sinon de mettre à jouer un fichier texte de sauvegardes avec les dernières statistiques d'une partie.

La procédure commence par demander le nom du joueur. Dans un *while* il récupère son nom (un string entre 2 et 12 caractères).

Si le fichier n'existe pas, il le crée en déterminant les différents champs pour la sauvegarde (nom du joueur, si la partie est gagnée, le mot à trouver, le temps et la difficulté)

Pour avoir un affichage agréable à lire (chaque paramètre aligné), on précise le nombre de caractère attendus (ici, 20). S'il y en a moins, des blancs remplissent l'espace.

```
// Creation d'une première ligne pour l'intitule des donnees
monStreamWriter1.WriteLine("{0,-20}{1,-20}{2,-20}{3,-20}{4,-20} \n", "Nom Joueur",  "Partie Gagne",  "Mot Initial",  "Temps",  "Difficulte");
```

Figure 4 - exemple de ce que l'on écrit sur la première ligne du fichier sauvegarde.txt

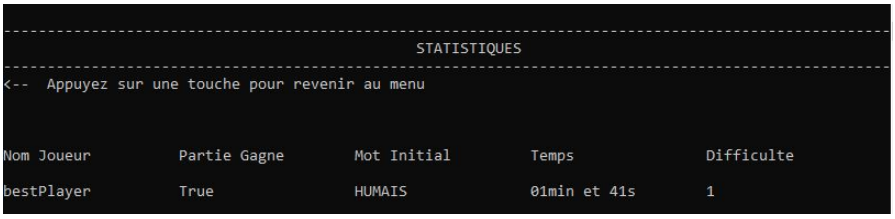


Figure 5 - exemple de rendu de l'écran des statistiques

Les statistiques de la partie sont ensuite ajoutées par le biais d'un StreamWriter.

Pour la durée de la partie, on formate le texte avant de l'inscrire par le biais de la fonction **FormaterTemps**(..).

*void **AfficherStats**()*

Cette procédure, appelée depuis le menu, permet d'afficher les statistiques enregistrées dans un fichier texte. Tout d'abord, le programme vérifie que ce fichier existe. Si ce n'est pas le cas, il affiche sur la console qu'aucune partie n'a été jouée pour le moment.

Dans le cas inverse, un StreamReader parcourt le fichier et affiche ligne par ligne son contenu dans la console.

Appuyer sur une touche permet de revenir au menu via la fonction **AfficherMenu**().

Fonctions pour les options

*void **AfficherOptions**()*

Cette procédure permet d'afficher les différentes options du jeu. Elle est appelée depuis le menu du jeu.

Elle commence par afficher les différentes options possibles :

- Réinitialiser les statistiques
- Modifier la position de l'indice
- Changer de dictionnaire
- Retourner au menu

Comme plus tôt, pour sélectionner la fonctionnalité voulue, le joueur doit entrer la valeur correspondante. Tant qu'il ne saisit pas un caractère valide, il lui est demandé de recommencer (boucle *while*).

Un Switch permet d'appeler la procédure associée à la fonctionnalité : **ResetFichier**(..), **ModifierIncide**(), **ModifierDico**() et **AfficherMenu**().

*void **ResetFichier**(string fichier)*

Cette procédure est utilisée dans le but de réinitialiser les statistiques.

Dans les faits, elle supprime le fichier de sauvegarde si celui-ci existe, sinon elle affiche sur la console que rien n'est à supprimer.

```
File.Delete(fichier); // On supprime le fichier
```

Figure 6 - fonction pour supprimer un fichier

void ***ModifierIndice()***

Cette procédure permet de rendre la position de l'indice aléatoire (durant une partie, une lettre au hasard est dévoilée), ou de la fixer (dans ce cas, seule la première lettre du mot est dévoilée).
Tout d'abord, cette procédure affiche l'état actuel de ce paramètre (aléatoire ou non). Il demande ensuite au joueur s'il souhaite le changer. Si le joueur accepte, on inverse le paramètre.
Un message de confirmation s'inscrit sur la console si jamais le joueur accepte.

void ***ModifierDico()***

Cette procédure permet de modifier le dictionnaire utilisé durant le jeu.
Tout d'abord, elle affiche les différents dictionnaires possibles (Animaux, Couleurs, Métiers, Noel, Normal).
Dans un *while* le joueur sélectionne le dictionnaire qu'il souhaite. Un *switch* permet ensuite d'assigner à la variable globale **fichierDico** le chemin d'accès pour le dictionnaire souhaité.

Fonctions mineures

Quitter

*void **QuitterJeu**()*

Cette procédure permet de quitter le jeu. Elle commence par demander confirmation au joueur. Si le joueur confirme, on ferme le jeu. Sinon, le programme appelle la procédure **AfficherMenu**().

```
Console.WriteLine("\nVoulez-vous vraiment quitter le jeu? O/N");
ConsoleKeyInfo saisie = Console.ReadKey(true);
if (saisie.Key == ConsoleKey.O)
{
    Environment.Exit(0); // ferme la console
}
else
{
    AfficherMenu(); // retourne au menu
}
```

Figure 7 - exemple de code pour fermer la console avec vérification de la touche saisie par l'utilisateur

Manipulations du dictionnaire

*string **SupprimerAccents**(string text)*

Cette fonction permet de supprimer les accents contenus dans une chaîne de caractères.
On standardise la chaîne que l'on nous donne en entrée, et à l'aide d'une boucle *foreach* on regarde chaque élément de cette chaîne et si son caractères Unicode appartient à la catégorie des accents, alors on la remplace par le même caractère mais sans accent.

*void **reduireFichier**(int tailleMot)*

Cette procédure permet de générer un nouveau fichier de mots depuis le fichier dictionnaire en sélectionnant tous les mots d'une taille donnée.
Pour ce faire, le programme emploie conjointement un StreamReader qui va lire chaque ligne du dictionnaire et un StreamWriter qui va écrire dans le dictionnaire réduit les mots de la taille souhaitée. Avant d'écrire les mots, il convient de préciser que ceux-ci passent par la fonction **SupprimerAccents**(mot) qui permet de les normaliser.

Normalisation

string ***FormaterTemps****(TimeSpan ts)*

Cette fonction permet de créer un string d'un TimeSpan passé en paramètre (valeur mesurée par le chronomètre). Ce string est formaté de sorte à afficher uniquement les minutes et les secondes mesurées.

Gestion du son

void ***JouerUnSon****(string nomSon)*

La procédure **JouerUnSon**(..) permet de jouer un son par le biais d'un objet SoundPlayer. Cet objet prend en paramètre un string, qui correspond au nom du fichier sonore.
La méthode **Play()** de cet objet permet de lancer la musique (effectué dans le cas où le fichier existe bel et bien).
Cette procédure est employée lors d'une défaite, d'une victoire, ou lors de la vérification des lettres.

void ***JouerUnSonEnBoucle****(string nomSon)*

Cette méthode fonctionne de manière similaire à **JouerUnSon**(..), à la différence près qu'elle permet de jouer ce son en boucle (via la méthode **PlayLooping()**).

void ***ArreterUnSon****()*

Cette méthode permet d'arrêter un son en cours .