ÉCOLE CENTRALE LYON

Rapport de Bureau d'Etude

INF-tc2

BE n°5 : Programmation Orientée Objet : Jeu du Pendu

Groupe Ab2

Marijan SORIC
Rémi LAURENS-BERGE

Encadrant

Stéphane DERRODE

Année 2021 - 2022

avril 1

Table des matières

1 Introduction

Nous nous proposons l'étude de la programmation du "Jeu du Pendu". Possédant les fonctions de base du jeu, nous allons dans ce rapport présenter en détails uniquement les améliorations que nous avons apportés au jeu.

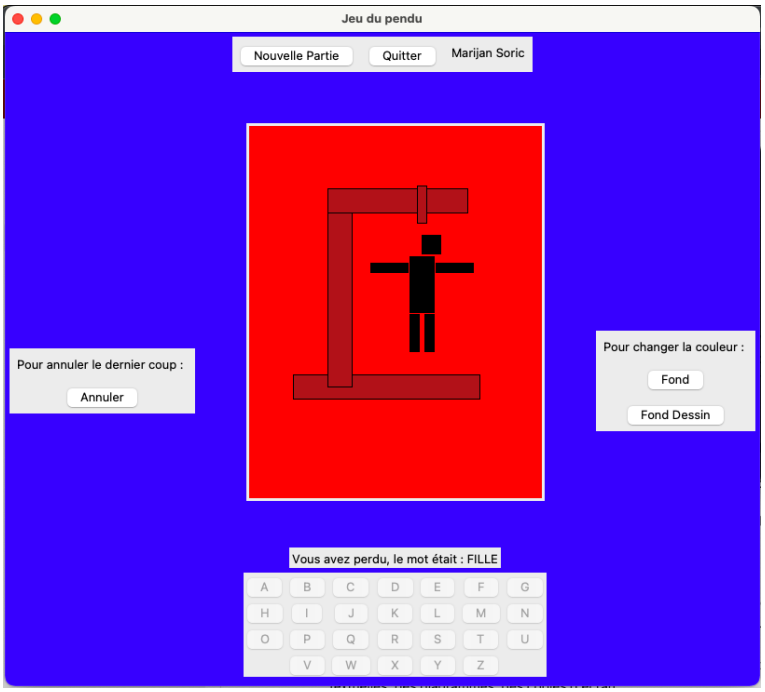

FIGURE 1 – Interface du Jeu du Pendu

L'objectif de ce BE est de travailler sur les notions de classe, d'objets, les relations de composition, d'association, d'aggrégation et d'héritage pour réaliser un projet complet : le jeu.

Le code de cette partie est disponible dans l'archive de ce même document pdf et sera considéré comme acquis dans la suite.

2 Diagramme de classes UML

Voici le diagramme complet de l'application montrant les liens entre les classes, les cardinalités, les attributs et les méthodes.

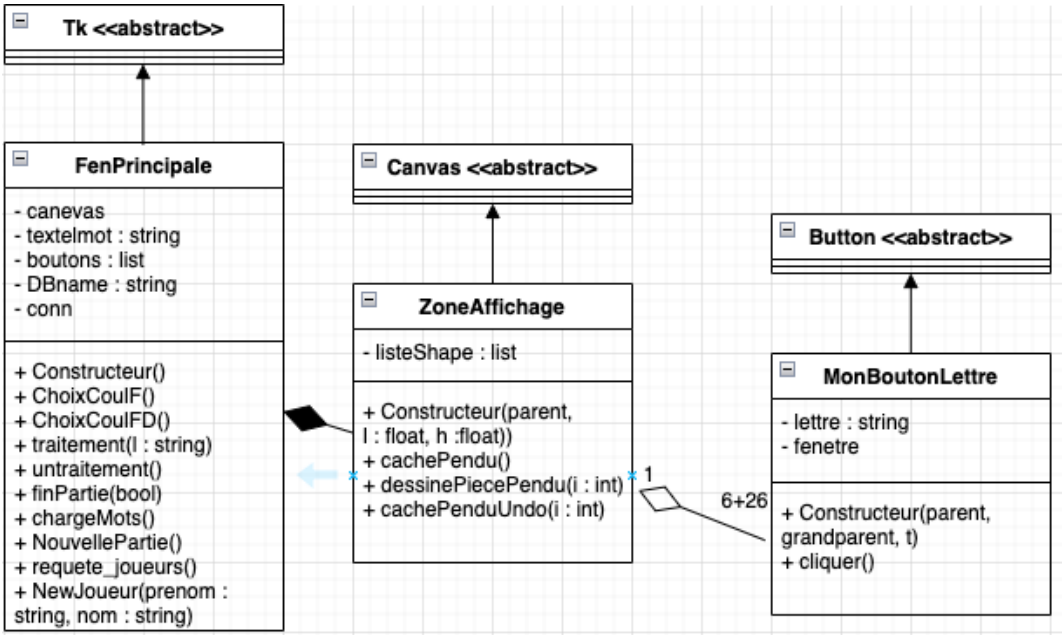

FIGURE 2 – Diagramme de classes UML

La classe $FenPrincipale$ hérite de Tk et gère l'initialisation de l'arbre de scène et des callbacks des widgets. La classe $ZoneAffichage$, quant à elle, hérite de $Canvas$ et gère toutes les opérations de dessin spécifiques à l'application. Enfin, $MonBoutonLettre$ hérite de la classe $Button$. $ZoneAffichage$ est une partie (composition) de $FenPrincipale$ et $MonBoutonLettre$ est un objet de $ZoneAffichage$. On voit dans le diagramme qu'il y a une seule zone d'affichage mais 26 boutons pour chaque lettre et cinq autres boutons dans cette zone (un pour créer une nouvelle partie, un pour quitter l'application, deux pour changer les couleurs du fond et du fond du dessin et un pour revenir un coup en arrière).
Voici le diagramme des classes pour le fichier $formes.py$ permettant de tracer les formes géométriques.

3 Amélioration du jeu

Les modifications sont détaillées en plusieurs sections.

3.1 Apparence

Dans un premier temps, nous souhaitons permettre au joueur de choisir les couleurs principales de l'application. Pour ce faire, nous avons ajouté un menu couleur dans lequel il y a deux boutons pour que l'utilisateur puisse choisir le fond du jeu et le fond du dessin à n'importe quel moment de la partie.

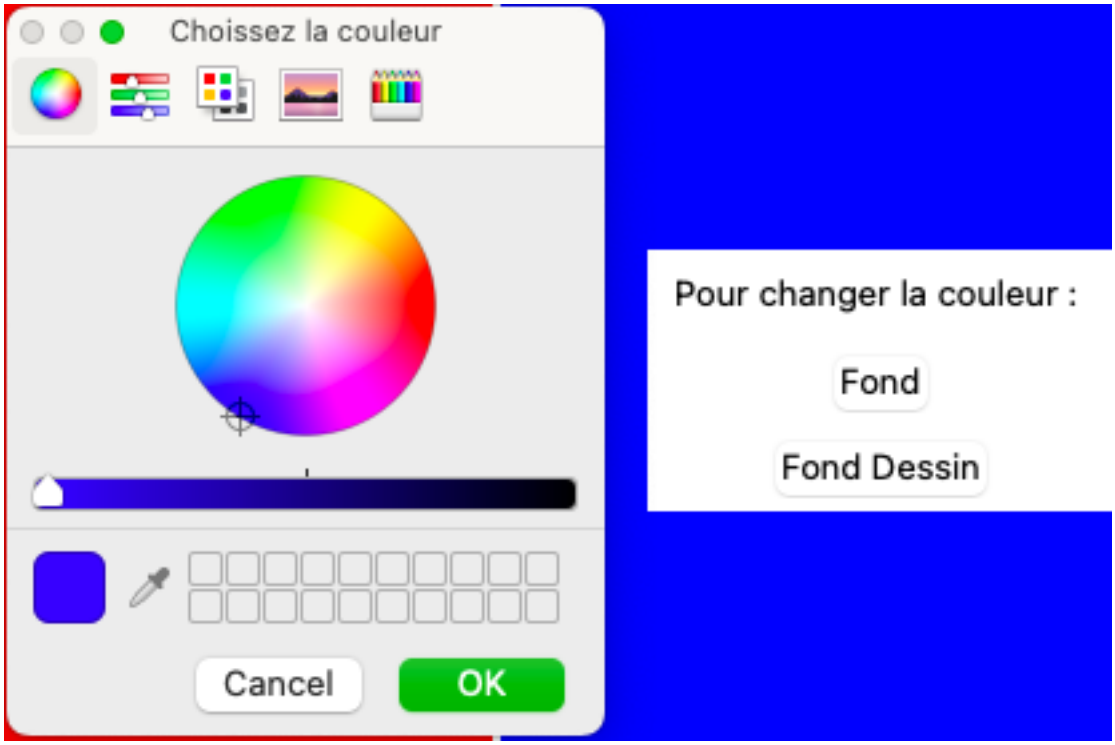

FIGURE 3 – Bouton pour changer les couleurs sur l'appli

Pour cela, nous utilisons le module *colorchooser* de *Tkinter* qui nous permet d'avoir accès à une interface graphique qui facilite de choix de couleur avec une sorte de palette. Voici comment nous avons implémenté cette fonctionnalité (qui correspond uniquement à ce qui a été utile). On créé un *MenuCouleur* qui va contenir nos 2 boutons, dans le constructeur de *FenPrincipale*, c'est ici qu'on construit l'arbre de scène. Sur chacun de ses boutons on applique la méthode *config* qui permet de faire appel à une fonction grâce à *command* afin de faire apparaître un menu pour choisir les couleurs lorsque l'utilisateur clique sur un des 2 boutons.

```
1  class FenPrincipale(Tk):
2      def __init__(self):
3          # C'est un bout de code du constructeur
4          MenuCouleur = Frame(self)
5          ChoixCouleurFond = Button(MenuCouleur, text='Fond')
6          ChoixCouleurFondDessin = Button(MenuCouleur, text='Fond Dessin')
7          chgcouleur = Label(MenuCouleur, text="Pour changer la couleur : ")
8
9          MenuCouleur.pack(side=RIGHT, padx=5, pady=5)
10         chgcouleur.pack(side=TOP, padx=5, pady=5)
11         ChoixCouleurFond.pack(side=TOP, padx=5, pady=5)
12         ChoixCouleurFondDessin.pack(side=TOP, padx=5, pady=5)
13
14
```

```
15          ChoixCouleurFond.config(command=self.ChoixCoulF)
16          ChoixCouleurFondDessin.config(command=self.ChoixCoulFD)
17
18      def ChoixCoulF(self):
19          #couleur du Fond
20          colors = colorchooser.askcolor(color="blue", title="Choissez la couleur")
21          self.configure(bg=colors[1])
22
23      def ChoixCoulFD(self):
24          #couleur du Fond-Dessin
25          colors = colorchooser.askcolor(color="red", title="Choissez la couleur")
26          self.__canevas.configure(bg=colors[1])
```

3.2 Triche !

Nous allons maintenant implémenter une technique 'undo', qui permet de revenir de un ou plusieurs coups en arrière, au cours d'une partie. Pour cela, on va créer un nouvel attribut $self.\_\_nbUndo$ dans la classe $FenPrincipale$ qui va nous permettre de compter le nombre de retour en arrière que le joueur aura fait (initialisé à zéro à chaque nouvelle partie). Pour ce faire, on stock en mémoire, via l'intermédiaire de l'attribut privé $self.\_\_motAffiche$ (une liste), l'ensemble des mots affichés à l'écran depuis le début de la partie.
A chaque utilisation du UndoButton, la fonction *untraiment* sera appelée. Celle-ci affiche d'abord le mot du coup précédent. Ensuite elle vérifie si les 2 derniers mots affichés sont les identiques, auquel cas, aucune nouvelle lettre n'a été découverte, et alors on cache la partie du dessin qui a été affichée au dernier coup. À noter que le bouton 'Annuler' est censé aider le joueur, ainsi, les lettres sur lesquelles on aura appuyer resteront grisées même après avoir cliqué sur ce bouton.
À noter que l'attribut $self._{motAffiche}$ est initialisé comme une liste de 2 éléments identiques (le mot recherché masque par des étoiles) pour ne pas avoir de problème d'indexation. D'autre part, on aurait aussi pu créer une méthode *cliquerUndo* dans la classe $MonBoutonLettre$ pour accéder à la dernière lettre pressée (comme *cliquer* le fait), puis faire appel à *untraitement*... ce n'est pas la méthode que nous avons choisi.

```
1   class FenPrincipale(Tk):
2       def __init__(self):
3           # aperçu du programme
4           MenuUndo = Frame(self)
5           UndoButton = Button(MenuUndo, text='Annuler')
6           UndoButton.pack(side=TOP, padx=5, pady=5)
7           UndoButton.config(command=self.untraitement)
8
9   def untraitement(self):
10          self.__nbUndo += 1
```

```
11          # affichage du mot précédent
12          self.__textelmot.set('Mot : ' + self.__motAffiche[-self.__nbUndo])
13          # vérification quant à l'apparition d'une lettre
14          if self.__motAffiche[-self.__nbUndo] == self.__motAffiche[-self.__nbUndo-1]:
15              self.__canevas.cachePenduUndo(self.__nbManques - self.__nbUndo)
```

3.3 Score joueur

On souhaite désormais implémenter un système de sauvegarde des échecs et des succès pour chaque joueur. On repère chaque joueur par son *nom*, et *prénom*, demandés en début de partie. Voici l'architecture de la base de donnée *pendu.bd* :

Joueur	
idjoueur	integer
nom	string
prenom	string

Partie	
idpartie	integer
mot	string
score	string

Analysons le fonctionnement que nous proposons. Sur l'interface, avant de lancer une nouvelle partie, un menu déroulant affiche l'ensemble des joueurs de la base de donnée *pendu.db*. On valide ensuite son choix. Pour cette première étape, on définit l'attribut privé $self.\_\_list = self.requete\_joueurs()$ où la méthode détaillée ci-dessous.

```
1   class FenPrincipale(Tk):
2       def __init__(self):
3           # un aperçu du constructeur
4           self.__conn = sqlite3.connect('pendu.db') # ficher de la BDD
5           self.__list = self.requete_joueurs()
6
7       def requete_joueurs(self):
8           curseur = self.__conn.cursor()
9           try:
10              S = "SELECT * FROM Joueur"
11              curseur.execute(S)
12          except sqlite3.OperationalError:
13              return None
14          else:
15              return curseur.fetchall()
```

On traite ensuite ensuite la validation du joueur. Voici le détail de l'affichage. L'attribut $self.\_\_nomjoueur$ est une chaîne de caractère variable qui contient le nom et prénom du joueur après validation. En cliquant sur ce-dernier, la méthode $self.get()$ appliqué au

menu déroulant permet d'obtenir le choix fait.

```
class FenPrincipale(Tk):
    def __init__(self):
        Score = Frame(self)

        self.__nomjoueur = StringVar()
        self.__nomjoueur.set("Nom joueur")

        def show(): #une fois le choix 'valider', cette fonction est exécutée
            label.config(text = clicked.get())  # affichage du joueur choisi
            self.__nomjoueur = clicked.get()
            Score.pack_forget() # on cache la zone de sélection

        options = [] # options du menu déroulant
        for joueur in self.__list:
            options.append(joueur[1] + ' ' + joueur[2])
            # remplie via self.__list contenant la liste des joueurs

        clicked = StringVar()
        clicked.set( "Sélectionnez Joueur")

        drop = OptionMenu(Score, clicked , *options)

        ValiderJ = Button(Score, text = "Valider Joueur", command=show)
        label = Label(barreOutils , text = " ")

        drop.pack(side=TOP, padx=5, pady=5)
        label.pack(side=TOP, padx=5, pady=5)
        ValiderJ.pack(side=TOP, padx=5, pady=5)
```

Au moment de la nouvelle partie, on va rajouter une ligne à la table *Partie* via la requête SQL :

```
INSERT INTO Partie(idpartie, idjoueur,mot) VALUES
((SELECT MAX(idpartie) FROM Partie)+1,'{}','{}')
```

Que l'on implémente dans la méthode *NouvellePartie*. On détermine l'*idpartie* en le construisant par rapport à ceux existant pour qu'il soit unique.

Enfin, on passe à la dernière phase : la fin de la partie. On souhaite mettre à jour la base de donnée avec le score obtenu par le joueur. On définit :

$$score = \frac{Card(\text{lettres trouvees})}{len(list(self.\_\_mot))}$$

Ainsi, on modifie la méthode $finPartie$:

```
def finPartie(self, gagne):
        for b in self.__boutons:
            b.config(state=DISABLED)
        if gagne:
            self.__textelmot.set(self.__mot +'- Bravo, vous avez gagné !')
        else:
            self.__textelmot.set('Vous avez perdu, le mot était : '+self.__mot)
        succes=0
        lettres = list(self.__motAffiche[-1])
        for i in range(len(self.__mot)):
            if self.__mot[i] == lettres[i]:
                succes += 1
        succes = succes/len(list(self.__mot))
        curseur = self.__conn.cursor()
        curseur.execute("UPDATE Partie SET succes = '{}' \
                        WHERE idpartie='{}'".format(succes, self.__idpartie))
```

Avec la requête SQL suivant qui met à jour la ligne souhaitée.

`UPDATE Partie SET succes = '{}' WHERE idpartie='{}'`

L'attribut $self.\_\_idpartie$ est obtenue par le constructeur $\_\_init\_\_$ au moment au choix du joueur.

Notons que nous avons aussi la possibilité de rajouter un joueur dans la base de donnée : par l'intermédiaire du widget *Entry* de *Tkinter*, il est possible de récupérer une chaîne de caractère remplie par l'utilisateur. On définit une méthode permettant d'ajouter le joueur s'il n'est pas présent, dans la table *Joueur*.

```
class FenPrincipale(Tk):
    def NewJoueur(self, prenomJ, nomJ):
        curseur = self.__conn.cursor()
        try:
            curseur.execute("SELECT prenom, nom FROM Joueur WHERE prenom ='{}' \
                            AND nom ='{}'".format(prenomJ, nomJ))
        except sqlite3.OperationalError:
            return None
        liste = curseur.fetchall()
        if len(liste) != 0:
            return None
        try:
            curseur.execute("INSERT INTO client(prenom, nom) VALUES  \
                            ('{}','{}')".format(prenomJ, nomJ))
        except sqlite3.OperationalError:
            return None
```