Cours d'algorithmique et programmation 2

Anne Parrain

UFR des Sciences
Licence Sciences et Technologie
mentions Mathématiques et Informatique
Semestre 2

Section 1

Concevoir une application graphique

Concevoir une application graphique

Concevoir une application graphique c'est :

- concevoir l'application, sans se préoccuper de ce qui concerne les interactions avec l'utilisateur (c'est ce qu'on appelera le modèle de l'application)
- concevoir la partie graphique indépendamment de l'application :
  - dessiner la maquette
  - créer les composants et les conteneurs
  - disposer les composants sur les conteneurs

  C'est ce qu'on appelera la partie vue de l'application.
- concevoir les interactions entre le modèle et la vue : c'est la partie contrôle de l'application qui va gérer les événements (les actions de l'utilisateur).

Cette conception séparée de l'interface graphique et du fonctionnement de l'application est le

modèle MVC (Modèle - Vue - Contrôleur)

Section 2

Le fil rouge

Le jeu du nombre mystérieux - Modèle

Principe : le joueur doit trouver le nombre mystérieux compris entre 1 et 100. Le joueur fait des propositions, et à chaque fois on lui indique si elle est trop grande ou trop petite. Le joueur cherche à minimiser son nombre de propositions.

Fonctionnalités du modèle :

- générer un nombre mystérieux
- répondre à une proposition
- gérer un compteur de propositions faites
- donner le nombre de propositions déjà faites

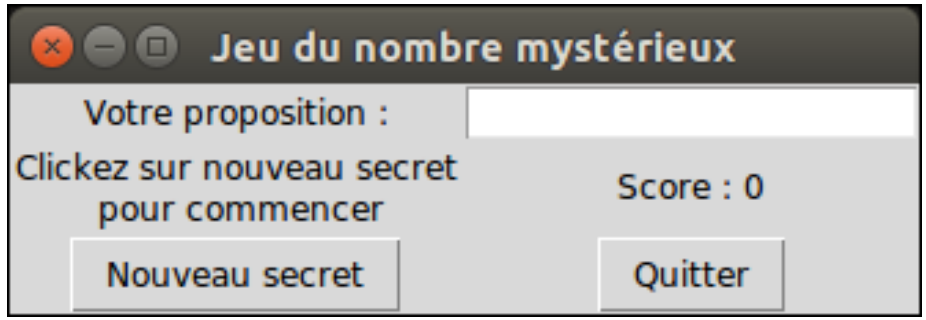
Le jeu du nombre mystérieux - Modèle

```
1  from random import randint
2
3  class Myst :
4    def __init__(self) :
5      '''Myst -> Myst'''
6      self.__score = 0
7      self.__secret = 0
8
9    def generer_secret(self) :
10     '''Myst -> Rien
11     reinitialise le secret aleatoirement
12     dans l'intervalle [1,100] '''
13     self.__secret = randint(1,100)
14     self.__score = 0
```

Le jeu du nombre mystérieux - Modèle

```
1   def comparer(self,prop) :
2     '''Myst,int -> int
3     retourne -1 si la proposition est < au secret,
4     0 si elle est egale, et 1 si elle est plus grande.
5     '''
6     self.__score += 1
7     if self.__secret == prop :
8       return 0
9     elif self.__secret > prop :
10      return -1
11    else :
12      return 1
13
14  def score(self) :
15    '''Myst -> int
16    retourne le score du joueur.
17    '''
18    return self.__score
```

Le jeu du nombre mystérieux (1)




```
1  class VueNbreMyst :
2
3    def __init__(self,mystere) :
4      '''VueNbreMyst, Myst -> VueNbreMyst
5      la vue est toujours initialisee avec le modele
6      auquel elle est liee
7      '''
8      # le modele est memorise dans un attribut
9      self.__myst = mystere
10     fen = Tk()
11     fen.title("Jeu du nombre mysterieux")
12     mess_prop = Label(fen,text="Votre proposition :")
13     mess_prop.grid(row=0,column=0)
```

Le jeu du nombre mystérieux (2)

```
1     self.__zn_prop = Entry(fen)
2     self.__zn_prop.grid(row=0, column=1)
3
4     self.__lbl_reponse =
5        Label(fen, text="Clickez sur nouveau secret\n"+
6                    "pour commencer")
7     self.__lbl_reponse.grid(row=1,column=0)
8
9     self.__lbl_score = Label(fen, text="Score : 0")
10    self.__lbl_score.grid(row=1, column=1)
11
12    btn_nouveau = Button(fen, text="Nouveau secret")
13    btn_nouveau.grid(row=2, column=0)
14
15    btn_quitter = Button(fen, text="Quitter")
16    btn_quitter.grid(row=2, column=1)
17    fen.mainloop()
```

Section 3

Gérer les événements (2)

L'attribut `command` de certains composants (1)

Certains composants ont un événement privilégié.
Par exemple, ces composants attendent un click :

- un bouton (Button)
- un bouton de choix (Radiobutton)
- une case à cocher (Checkbutton)

Ils possèdent un attribut `command` qui attend le nom de la fonction à exécuter. Cette fonction ne doit attendre aucun paramètre (ou être une méthode sans autre paramètre que l'objet sur lequel elle s'applique).

```
1 btn_quitter = Button(fen,text="Bye",
2                      command = fenetre.destroy)
```

Le code en python : `src/exemple1.py`

Les autres composants

Pour la saisie du nom du client dans la zone de saisie, quel est le bon événement ?

- dès que l'utilisateur saisit un caractère ?
- dès que l'utilisateur saisit le caractère `'a'` ?
- dès que l'utilisateur appuie sur la touche *Entrée* pour signifier qu'il a fini sa saisie ?

Cela dépend :

- si on veut juste mémoriser le nom saisi lorsqu'il est complet
- si on veut lui faire une proposition de noms possibles avec une complétion automatique au fur et à mesure de sa saisie.

C'est variable, et il n'y a pas de règle absolue.

Lier un événement sur un composant à une action (1)

D'une manière générale, chaque composant possède la méthode
`def bind(self,sequenceEvt,fonction,add="")`
dont les paramètres sont :

- `self` : le composant lui même
- `sequenceEvt` : l'événement qu'on veut lier à une action. Chaque événement est décrit par une chaîne de caractères spécifique.
- `fonction` : la fonction qui va gérer l'événement. Elle prend un paramètre de type `Event`. On appelle cette fonction un *callback*. Ce sont ces fonctions qui auront le rôle de contrôleur dans notre modèle MVC.
- `add` : par défaut, vaut `""` qui signifie que cette liaison d'une action à un événement annule et remplace toutes les liaisons précédentes sur cet événement et ce composant. Si `add` est positionné à `"+"`, alors la liaison vient s'ajouter aux liaisons déjà existantes.

Lier un événement sur un composant à une action (2)

Exemple Cliquer sur le bouton `Bye` ferme l'application, mais appuyer sur la touche `Entrée` si le bouton `Bye` est actif ferme aussi l'application.

```
1  import tkinter
2  class UneGUI:
3    def __init__(self):
4      self.__fenetre = tkinter.Tk()
5      self.__fenetre.title('Exemple 2')
6      btn_quitter = tkinter.Button(self.__fenetre,
7                                   text="Bye",
8                                   command = self.__fenetre.destroy)
9      btn_quitter.bind("<Return>",self.quitter)
10     btn_quitter.pack()
11     self.__fenetre.mainloop()
12
13   def quitter(self, event) :
14     self.__fenetre.destroy()
```

Les événements

Quelques séquences décrivant les événements :

- `<Key>` : une touche, n'importe laquelle, a été enfoncée
- `a` : la touche `a` a été enfoncée. Pour toutes les touches qui représentent un caractère imprimable, on décrit l'événement de la même façon. Sauf pour les deux cas suivants :
  - `<space>` : la barre d'espace a été enfoncée
  - `<less>` : la touche `<` a été enfoncée
- `<Return>` : la touche `Entrée` a été enfoncée
- `<Button-1>` ou `<1>` : le bouton gauche de la souris a été pressé. Le `2` représente le bouton du milieu et le `3` le bouton droit
- `<Double-Button-1>` : pour un double click. On peut utiliser le préfixe `Triple` également
- ...

Retour au jeu du nombre mystérieux (1)

```
1   # toujours dans le constructeur de la vue,
2   # avant le lancement de fen.mainloop()
3   # apres avoir declare la zone de saisie
4   self.__zn_prop.bind("<Return>",
5                       self.envoyer_proposition)
6
7   # pour declarer le bouton Nouveau secret
8   btn_nouveau = Button(fen, text="Nouveau secret",
9                        command=self.reinit)
10  btn_nouveau.grid(row=2, column=0)
11
12  # pour declarer le bouton Quitter
13  btn_quitter = Button(fen, text="Quitter",
14                       command=fen.destroy)
```

Retour au jeu du nombre mystérieux (2)

On ajoute deux méthodes à la classe `VueNbreMyst`.

Ce sont ces méthodes qui jouent le rôle de contrôleur : elles font le lien entre le *modèle* et la *vue*.

```
1   def reinit(self):
2     '''VueNbreMyst -> rien
3     controleur action reinitialisation
4     reinitialise le jeu'''
5     # prevenir le modele
6     self.__myst.generer_secret()
7     # mettre a jour la vue en fonction du modele
8     self.__lbl_reponse["text"] = "C'est parti!"
9     self.__lbl_score["text"] = "Score : "
10        + str(self.__myst.score())
```

Retour au jeu du nombre mystérieux (3)

```
1   def envoyer_proposition(self,event):
2       '''VueNbreMyst,Event -> rien
3       controleur action proposition du joueur
4       '''
5       # recuperer toutes les informations
6       prop = int(self.__zn_prop.get())
7       # prevenir le modele
8       rep = self.__myst.comparer(prop)
9       # mettre a jour la vue en fonction du modele
10      if (rep == -1):
11          message = "trop petit!"
12      elif (rep == 1) :
13          message = "trop grand!"
14      else :
15          message = "Vous avez trouve en "
16           + str(self.__myst.score()) + " coups!"
17      self.__lbl_reponse["text"] = message
18      self.__lbl_score["text"] = "Score : "
19           + str(self.__myst.score())
```

Section 4

Générer des contrôleurs

Des fonctions qui retournent des fonctions (1)

Soit l'application suivante :

Des fonctions qui retournent des fonctions (2)

```
1   ... # dans le __init__
2   # on memorise les indices des images affichees
3   self.__indices = []
4   for i in range(4):
5     self.__indices.append(
6                   randint(0,len(self.__images)-1))
7
8   # puis on cree chaque composant graphique et
9   # on les pose. On memorise tous les boutons
10  # dans une liste de boutons
11  self.__bouton_images = []
12  for i in range(4) :
13    btn = tkinter.Button(fenetre,
14            image=self.__images[self.__indices[i]])
15    btn["command"] = ???
16    self.__bouton_images.append(btn)
17    self.__bouton_images[i].grid(row=0,column=i)
```

Des fonctions qui retournent des fonctions (3)

Pour chaque bouton `__bouton_images[i]` :

- modifier `__indices`, en incrémentant `__indices[i]`
- changer l'apparence de `__bouton_images[i]` en fonction de la nouvelle valeur de `__images[__indices[i]]`

Problème : on ne peut pas associer à `command` une fonction qui prendrait `i` en paramètre.

Solution : on va construire dynamiquement une fonction spécifique pour chaque bouton

Des fonctions qui retournent des fonctions (4)

une nouvelle méthode dans de la classe VueSphere

```
1 def change_image_bouton(self, i):
2   '''VueSphere, int -> Fct'''
3   def change_image():
4     '''(rien) -> (rien)'''
5     self.__les_indices[i] =
6           (self.__les_indices[i] + 1)%len(self.__images)
7     self.__bouton_images[i]["image"] =
8           self.__images[self.__les_indices[i]]
9   return change_image
```

Des fonctions qui retournent des fonctions (5)

une autre nouvelle méthode dans la classe `VueSphere` :

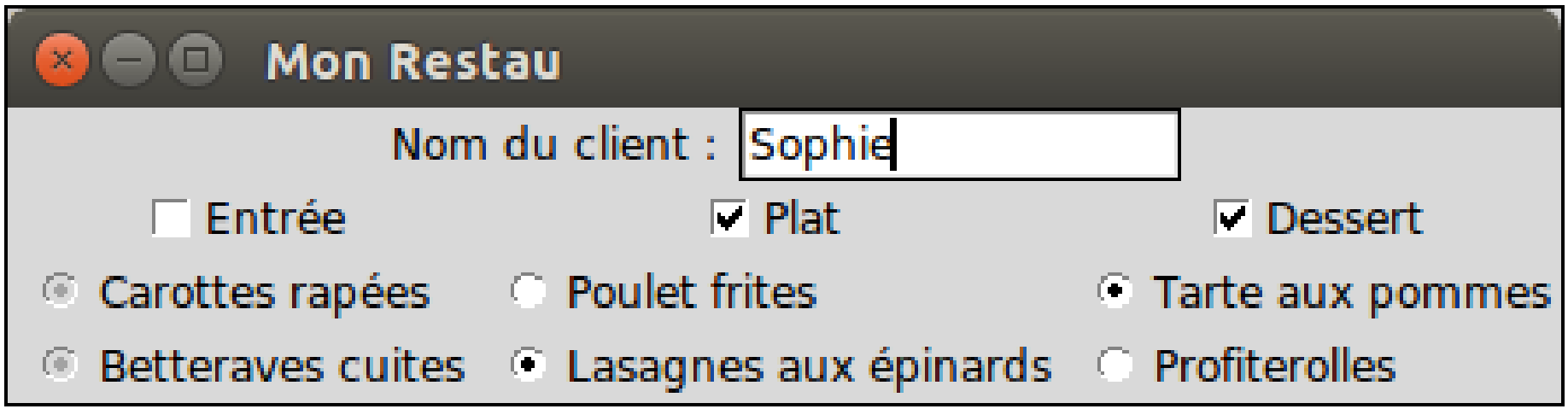
```
def initialise(self):
  for i in range(4) :
    self.__bouton_images[i]["command"] =
        self.change_image_bouton(i)
  # on lance la boucle d'ecoute des evenements
  self.__fenetre.mainloop()
```

Le script principal pour lancer l'application est maintenant

```
### script principal
if __name__ == '__main__' :
    monAppli = VueSphere()
    monAppli.initialise()  ## ou bien self.initialise()
                           ## est appele dans le constru
```

Le code en python : `src/exemple5.py`

Retour au restaurant

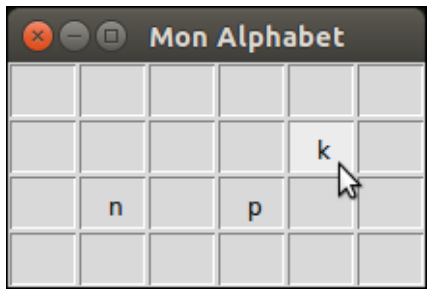


Le code en python initial : `src/restaurantEvt.py`

Le code en python sans répétitions : `src/restaurantEvt2.py`

Exemple MonAlphabet (1)

Supposons un jeu simple : un alphabet est caché, morcelé dans une grille. On peut connaître la valeur de chaque cellule de la grille pour révéler l'alphabet.



Modèle

```
1  import random
2  import string
3
4  class MonAlphabet :
5    def __init__(self,nblig = 4, nbcol = 6) :
6      '''MonAlphabet, int, int -> MonAlphabet
7      uniquement les 24 premieres lettres...
8      '''
9      self.__nblig = nblig
10     self.__nbcol = nbcol
```

Exemple MonAlphabet (2)

Toujours dans la classe MonAlphabet :

```
1   def valeur(self,lig,col) :
2     '''MonAlphabet, int, int -> str'''
3     return string.ascii_lowercase[lig*self.__nbcol + col]
4
5   def get_nb_lignes(self):
6     '''MonAlphabet -> int'''
7     return self.__nblig
8
9   def get_nb_colonnes(self):
10    '''MonAlphabet -> int'''
11    return self.__nbcol
```

Le code en python de MonAlphabet : `src/alphabet.py`

Exemple MonAlphabet (3)

Vue

```
1  import tkinter
2  import alphabet
3
4  class VueMonAlphabet :
5    def __init__(self,mem) :
6      '''VueMonAlphabet,MonAlphabet -> VueMonAlphabet'''
7      self.__mem = mem
8      fen = Tk()
9      fen.title("Mon Alphabet")
10     for i in range(mem.get_nb_lignes()) :
11       for j in range(mem.get_nb_colonnes()) :
12         btn = Button(fen,text="",relief="sunken",width=1)
13         # un clic sur btn doit reveler
14         # self.__mem.valeur(i,j)
15         btn.grid(row=i,column=j)
16     fen.mainloop()
```

Exemple MonAlphabet (4)

Solution dans la **Vue** pour définir un contrôleur adapté à chaque bouton :
On mémorise le boutons dans une liste de liste, et on déporte la création des contrôleurs dans une méthode `valeur()`

```
1  class VueMonAlphabet :
2    def __init__(self,mem) :
3      '''VueMonAlphabet,MonAlphabet -> VueMonAlphabet'''
4      self.__mem = mem
5      self.__fen = Tk()
6      fen.title("Mon Alphabet")
7      self.__les_btns = []
8      for i in range(mem.get_nb_lignes()) :
9        une_ligne = []
10       for j in range(mem.get_nb_colonnes()) :
11         btn = Button(self.__fen,text="",
12                      relief="sunken",width=1)
13         btn.grid(row=i,column=j)
14         une_ligne.append(btn)
15       self.__les_btns.append(une_ligne)
```

Exemple MonAlphabet (5)

Solution dans la **Vue** pour définir un contrôleur adapté à chaque bouton :

```
1  class VueMonAlphabet :
2
3    def valeur(self,i,j) :
4      '''VueMonAlphabet,int,int -> Function
5      controle l'action devoile un bouton
6      '''
7      def la_valeur() :
8        val = self.__mem.valeur(i,j)
9        self.__les_btns[i][j]["text"] = val
10     return la_valeur
```

Exemple MonAlphabet(6)

Une méthode pour démarrer l'application :

```
1     def lance(self):
2     # un clic sur un bouton doit reveler
3     # self.__mem.valeur(i,j)
4     for i in range(mem.get_nb_lignes()) :
5       for j in range(mem.get_nb_colonnes()) :
6         self.__lesBtns[i][j].config(
7                           command=self.valeur(i,j))
8     # lancer la boucle d'ecoute des evenements
9     self.__fen.mainloop()
```

Et le script principal :

```
1 if __name__ == "__main__" :
2     alpha = alphabet.MonAlphabet()
3     mon_appli = VueMonAlphabet(alpha)
4     mon_appli.lance()
```

Le code en python de VueMonAlphabet : `src/alphabet_gui.py`