

Conception d'Applications Interactives

développement d'IHM en python/TkInter

Alexis NEDELEC

Centre Européen de Réalité Virtuelle
Ecole Nationale d'Ingénieurs de Brest

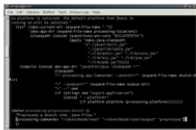
enib ©2018



Interfaces Homme-Machine

Interagir avec un ordinateur

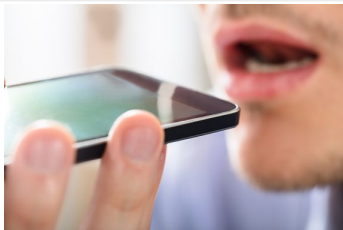
- CLI (Command Line Interface) : interaction clavier
- GUI (Graphical User Interface) : interaction souris-clavier
- NUI (Natural User Interface) : interaction tactile, capteurs



Interfaces Homme-Machine

Interagir avec un ordinateur

- VUI (Voice User Interface) : interaction vocale
- OUI (Organic User Interface) : interaction biométrique
- ...



Interfaces Homme-Machine

Objectifs du cours

Savoir développer des IHM avec une bibliothèque de composants

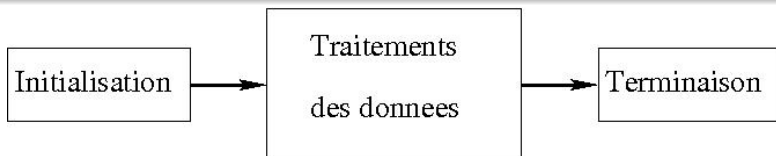
- 1 paradigme de programmation événementielle (Event Driven)
- 2 interaction WIMP (Window Icon Menu Pointer)
- 3 bibliothèque de composants graphiques (Window Gadgets)
- 4 développement d'applications GUI (Graphical User Interface)
- 5 patrons de conception (Observer, MVC)



Programmation événementielle

Programmation classique : trois étapes séquentielles

- ❶ initialisation
 - importer les modules externes
 - ouverture de fichiers
 - connexions serveurs SGBD, Internet ..
- ❷ traitements
 - affichages, calculs, modifications des données
- ❸ terminaison
 - sortir “proprement” de l’application



Programmation événementielle

Programmation d'IHM : l'humain dans la boucle ... d'événements

① initialisation

- modules externes, ouverture fichiers, connexion serveurs ...
- création de **composants graphiques**

② traitements

- implémentation des fonctions correspondant aux **actions**
- liaisons composant-**événement**-action
- attente, dans une **boucle**, d'événement lié à l'interaction utilisateur-composant
- exécution des traitements liés à l'action de l'utilisateur

③ terminaison

- sortir “proprement” de l'application

Programmation événementielle

CLI : Command Line Interface

- interaction faible (textuelle) avec l'utilisateur.
- l'application contrôle le déroulement du programme

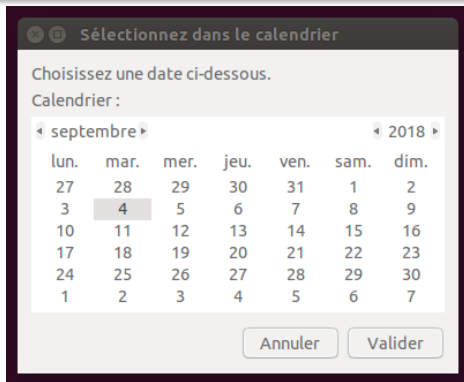
```
{logname@hostname} cal 09 2018
```

```
      Septembre 2018
di lu ma me je ve sa
                        1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
16  17  18  19  20  21  22
23  24  25  26  27  28  29
30
```

Programmation événementielle

GUI : Graphical User Interface

- interaction forte avec l'utilisateur.
- l'application répond aux sollicitations de l'utilisateur



```
{logname@hostname} zenity -calendar
```


Programmation événementielle

A l'écoute de l'utilisateur

- l'utilisateur agit via un périphérique (clavier, souris ...)
- un événement est détecté suivant l'action d'un utilisateur
- l'événement est géré par l'application dans une file d'événements
- pour chaque événement un bloc de code (fonction) est exécuté
- le programme reste à l'écoute des événements (boucle infinie)

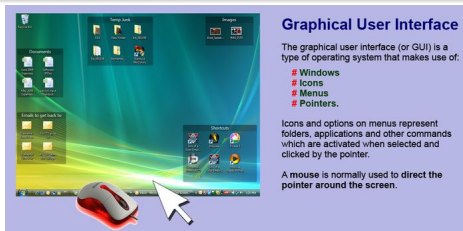
```
// PROGRAMME
Main()
{
    ...
    while(true) // tantque Mamie s'active
    {
        // récupérer son action (faire une maille ...)
        e = getNextEvent();
        // traiter son action (agrandir le tricot ...)
        processEvent();
    }
    ...
}
```



Programmation événementielle

API pour développer des IHM

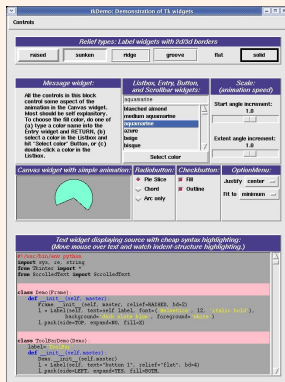
- Java : AWT, SWT, Swing, JavaFX, ..., JGoodies, QtJambi ...
- C, C++ : Xlib, GTK, Qt, MFC, ...
- Python : TkInter, wxWidgets, PyQt, PySide, Kivy, libavg...
- JavaScript : Angular, React, Vue.js, JQWidgets ...
- ...



https://www.ictlounge.com/html/operating_systems.htm

Python/TkInter

TkInter : Tk (de Tcl/Tk) pour python



Documentation python :

<https://docs.python.org/fr/3/library/tk.html>

Hello World

Première IHM (hello.py)

```
import sys
major=sys.version_info.major
minor=sys.version_info.minor
if major==2 and minor==7 :
    import Tkinter as tk
    import tkinterFileDialog as filedialog
elif major==3 and minor==6 :
    import tkinter as tk
    from tkinter import filedialog
else :
    print("Your python version is : ",major,minor)
    print("... I guess it will work !")
    import tkinter as tk
    from tkinter import filedialog
```

Hello World

Première IHM (hello.py)

```
from Tkinter import Tk,Label,Button
mw=Tk()
label_hello=Label(mw,
                   text="Hello World !",fg="blue")
button_quit=Button(mw,
                   text="Goodbye World", fg="red",
                   command=mw.destroy)
label_hello.pack()
button_quit.pack()
mw.mainloop()
```



Hello World

Création de fenêtre principale et de composants

- `mw=Tk()`
- `label_hello=Label(mw, ...)`
- `button_quit=Button(mw, ...)`

Interaction sur un composant

- `button_quit=Button(..., command=mw.destroy)`

Positionnement des composants

- `label_hello.pack(), button_quit.pack()`

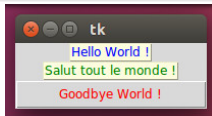
Entrée dans la boucle d'événements

- `mw.mainloop()`

Personnalisation d'IHM

Chargement d'un fichier de configuration d'options

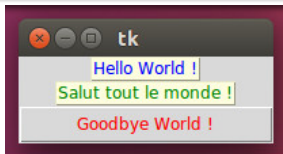
```
from Tkinter import Tk,Label,Button
mw=Tk()
mw.option_readfile("hello.opt")
label_hello=Label(root,text="Hello World !")
label_bonjour=Label(root,name="labelBonjour")
button_quit=Button(root,text="Goodbye World !")
label_hello.pack()
label_bonjour.pack()
button_quit.pack()
mw.mainloop()
```



Personnalisation d'IHM

Contenu d'un fichier d'options (`hello.opt`)

```
*Button.foreground: red
*Button.width:20
*Label.foreground: blue
*labelBonjour.text: Salut tout le monde !
*labelBonjour.foreground: green
*Label.background: light yellow
*Label.relief: raised
```



Composants graphiques

Widgets : **Window gadgets**

Fonctionnalités des widgets, composants d'IHM

- affichage d'informations (label, message...)
- composants d'interaction (button, scale ...)
- zone d'affichage, saisie de dessin, texte (canvas, entry ...)
- conteneur de composants (frame)
- fenêtres secondaires de l'application (toplevel)

Composants graphiques

TkInter : fenêtres, conteneurs

- `Toplevel` : fenêtre secondaire de l'application
- `Canvas` : afficher, placer des “éléments” graphiques
- `Frame` : surface rectangulaire pour contenir des widgets
- `Scrollbar` : barre de défilement à associer à un widget

TkInter : gestion de textes

- `Label` : afficher un texte, une image
- `Message` : variante de label pour des textes plus importants
- `Text` : afficher du texte, des images
- `Entry` : champ de saisie de texte

Composants graphiques

Tkinter : gestion de listes

- Listbox : liste d'items sélectionnables
- Menu : barres de menus, menus déroulants, surgissants

Tkinter : composants d'interactions

- Menubutton : item de sélection d'action dans un menu
- Button : associer une interaction utilisateur
- Checkbutton : visualiser l'état de sélection
- Radiobutton : visualiser une sélection exclusive
- Scale : visualiser les valeurs de variables

Fabrice Sincère, cours sur python, notamment TkInter

Gestion d'événements

Structuration d'un programme

```
# Actions : definition des comportements
if __name__ == "__main__" :
# IHM : creation des composants
    mw=tk.Tk()
    hello=tk.Label(mw,text="Hello World !",fg="blue")
    quit=tk.Button(mw,text="Goodbye World",\
                    fg="red",command=mw.destroy)
# IHM : Gestionnaires de positionnement (layout manager)
    hello.pack()
    quit.pack()
# Interaction : liaison Composant-Evenement-Action
    mw.mainloop()
    exit(0)
```

Gestion d'événements

Interaction par défaut : composant-<Button-1>-action

- option `command` : "click gauche", exécute la fonction associée
`quit=Button(mw,...,command=mw.destroy)`

Paramétrer l'interaction : composant-événement-action

- implémenter une fonction "réflexe" (action)
`def callback(event) :`
`mw.destroy()`
- lier (bind) un événement au composant pour déclencher la fonction réflexe associée
`quit.bind("<Button-1>",callback)`

Gestion d'événements

Types d'événements

représentation générale d'un événement :

- `<Modifier-EventType-ButtonNumberOrKeyName>`

Exemples

- `<Control-KeyPress-A>` (`<Control-Shift-KeyPress-a>`)
- `<KeyPress>`, `<KeyRelease>`
- `<Button-1>`, `<Motion>`, `<ButtonRelease>`

Principaux types

- `Expose` : exposition de fenêtre, composants
- `Enter`, `Leave` : pointeur de souris entre, sort du composant
- `Configure` : l'utilisateur modifie la fenêtre
- ...

Gestion d'événements

Fonctions réflexes

```
def callback(event) :  
    # action to do need information from :  
    # - user: get data from pointer, keyboard...  
    # - widget: get or set widget data  
    # - application: manage data application
```

Récupération d'informations

- liées à l'utilisateur (argument `event`)
- liées au composant graphique :
 - `configure()` : fixer des valeurs aux options de widget
 - `cget()` : récupérer une valeur d'option
- liées à l'application (arguments supplémentaires)

Gestion d'événements

Informations liées à l'utilisateur

```
def callback(event) :  
    print(dir(event))  
    print("pointer coordinates on screen ",  
          event.x_root,event.y_root)
```

Informations liées au composant graphique

```
def callback_info(event) :  
    # display pointer coordinates in the widget  
    event.widget.configure(text="x="+str(event.x)\  
                             +"y="+str(event.y))
```


Gestion d'événements

Informations liées à l'application : passage d'arguments

```
def mouse_location(event,label):  
    label.configure(text= "Position X =" \  
                        + str(event.x) \  
                        + ", Y =" \  
                        + str(event.y))
```

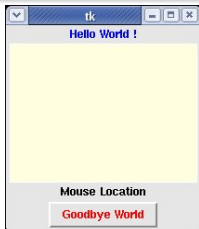
Transmission des données (data) : fonction anonyme (lambda)

```
canvas=tk.Canvas(mw, ... )  
data=tk.Label(mw,text="Mouse Location")  
canvas.bind("<Motion>",  
            lambda event,label=data : \  
                mouse_location(event,label))
```

Gestion d'événements

Transmission des données : fonction anonyme (lambda)

```
canvas=tk.Canvas(mw,\n                  width=200,height=150,bg="light yellow")\n\nlabel=tk.Label(mw,text="Mouse Location")\n\nlabel.bind("<Motion>",\n           lambda event,label=label : \n               mouse_location(event,label))
```



Gestion d'événements

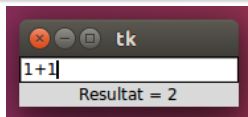
Traitement des données : `eval()`

```
# ----- Initialisation -----  
# importation du module TkInter (version 2 et 3)  
from math import *  
# Actions : definition des comportements  
def evaluer(event):  
    label.configure(text="Resultat = " \  
                    + str(eval(entry.get())))  
if __name__ == "__main__":  
# IHM : creation des composants  
    mw=tk.Tk()  
    entry=tk.Entry(mw)  
    label=tk.Label(mw)
```

Gestion d'événements

Traitement des données : `eval()`

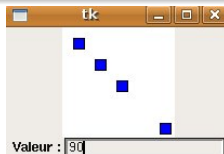
```
# IHM : Gestionnaires de positionnement (layout manager)
    entry.pack()
    label.pack()
# Interaction : liaison Composant-Evenement-Action
    entry.bind("<Return>",evaluer)
    mw.mainloop()
    exit(0)
```



Gestion d'événements

Communication entre composants : `event_generate()`

```
# Actions : definition des comportements
def display(event):
    print("display()")
    x=int(entry.get())
    canvas.create_rectangle(x,x,x+10,x+10,fill="blue")
def set_value(event):
    print("set_value()")
    canvas.event_generate('<Control-Z>')
```



Gestion d'événements

Communication entre composants

```
if __name__ == "__main__" :  
    # IHM : creation des composants  
    mw=tk.Tk()  
    canvas=tk.Canvas(mw,...)  
    label=tk.Label(mw,text="Valeur :")  
    entry=tk.Entry(mw)  
    # IHM : Gestionnaires de positionnement (layout manager)  
    canvas.pack()  
    label.pack(side="left")  
    entry.pack(side="left")  
    # Interaction : liaison Composant-Evenement-Action  
    mw.bind("<Control-Z>",display)  
    entry.bind("<Return>",set_value)  
    mw.mainloop()
```

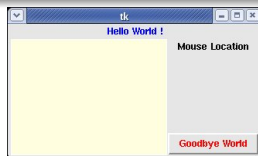
Positionnement de composants

TkInter : Layout manager

- `pack()` : agencer les widgets les uns par rapport aux autres
- `grid()` : agencer sous forme de grille (ligne/colonne)
- `place()` : positionner les composants géométriquement

`pack()` : "coller" les widgets par leur côté

```
labelHello.pack()  
canvas.pack(side="left")  
labelPosition.pack(side="top")  
buttonQuit.pack(side="bottom")
```



Positionnement de composants

Regroupement de composants : Frame

```
if __name__ == "__main__" :  
    # IHM : creation des composants  
    mw=tk.Tk()  
    frame=tk.Frame(mw,bg="yellow")  
    canvas=tk.Canvas(frame,...,bg="light yellow")  
    data=tk.Label(frame,text="Mouse Location")  
    hello=tk.Label(frame,text="Hello World !",fg="blue")  
    quit=tk.Button(frame,text="Goodbye World",fg="red",..
```



Positionnement de composants

Regroupement de composants : Frame

```
# IHM : Gestionnaires de positionnement (layout manager)
frame.pack(fill="both",expand=1)
hello.pack()
canvas.pack(fill="both",expand=1)
data.pack()
quit.pack()
```



Positionnement de composants

grid() : agencement ligne/colonne

```
# IHM : creation des composants
labelNom=tk.Label(mw,text="Nom :")
labelPrenom=tk.Label(mw,text="Prenom :")
entryNom=tk.Entry(mw)
entryPrenom=tk.Entry(mw)
# IHM : Gestionnaires de positionnement (layout manager)
labelNom.grid(row=0)
labelPrenom.grid(row=1)
entryNom.grid(row=0,column=1)
entryPrenom.grid(row=1,column=1)
```



Positionnement de composants

place() : positionnement géométrique

```
# IHM : creation des composants
msg=tk.Message(mw,text="Place : \n \n
                options de positionnement de widgets",
                justify="center",bg="yellow",relief="ridge")
okButton=tk.Button(mw,text="OK")
# IHM : Gestionnaires de positionnement (layout manager)
msg.place(relx=0.5,relx=0.5,
          relwidth=0.75,relheight=0.50,anchor="center")
okButton.place(relx=0.5,relx=1.05,in_=msg,anchor="n")
```



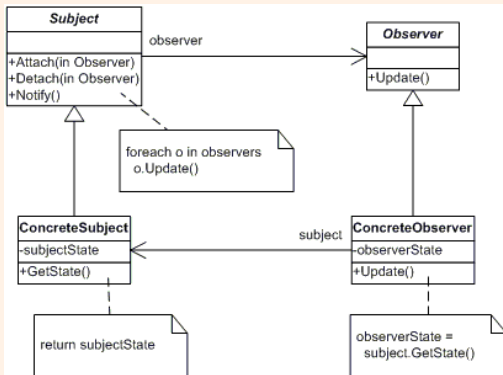
Patrons de conception

Programmer des IHM "proprement"

- Patrons de conception (Design Pattern)
- Modèle **Observer**
 - observateurs (**Observer**)
 - d'observable (**Subject**)
- Modèle **Observer** avec IHM
- Modèle MVC pour IHM
 - M : le modèle (les données)
 - V : l'observation du modèle
 - C : la modification du modèle

Modèle Observer

Observateur-Sujet observé



Modèle Observer

Subject : informer les Observer

```
class Subject(object):  
    def __init__(self):  
        self.observers=[]  
    def notify(self):  
        for obs in self.observers:  
            obs.update(self)
```

En cas de modification des données du modèle :

- `notify()` : demander aux observateurs de se mettre à jour

Modèle Observer

Subject : ajouter/supprimer des Observer

```
def attach(self, obs):  
    if not hasattr(obs, "update"):  
        raise ValueError("Observer must have \  
                           an update() method")  
    self.observers.append(obs)  
def detach(self, obs):  
    if obs in self.observers :  
        self.observers.remove(obs)
```

Modèle Observer

Observer : mise à jour

```
class Observer:  
    def update(self, subject):  
        raise NotImplementedError
```

Lorsque l'observable (Subject) est modifié :

- `update()` : on se met à jour

Modèle Observer

Exemple : Distributeur de billets

```
class ATM(Subject):
    def __init__(self, amount):
        Subject.__init__(self)
        self.amount = amount
    def fill(self, amount):
        self.amount = self.amount + amount
        self.notify()                # obs.update(self)
    def distribute(self, amount):
        self.amount = self.amount - amount
        self.notify()                # obs.update(self)
```

Modèle Observer

Exemple : Distributeur de billets

```
class Amount(Observer):  
    def __init__(self,name):  
        self.name=name  
    def update(self,subject):  
        print(self.name,subject.amount)
```

Modèle Observer

Exemple : Distributeur de billets

```
if __name__ == "__main__" :  
    amount=100  
    dab = ATM(amount)  
    obs=Amount("Observer 1")  
    dab.attach(obs)  
    obs=Amount("Observer 2")  
    dab.attach(obs)  
    for i in range(1,amount/20) :  
        dab.distribute(i*10)  
    dab.detach(obs)  
    dab.fill(amount)
```

MVC

Trygve Reenskaug

"MVC was conceived as a general solution to the problem of users controlling a large and complex data set. The hardest part was to hit upon good names for the different architectural components. Model-View-Editor was the first set. After long discussions, particularly with Adele Goldberg, we ended with the terms Model-View-Controller."

Smalltalk

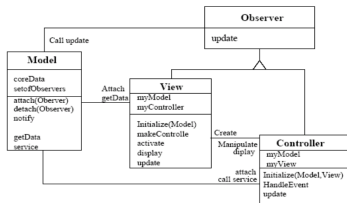
"MVC consists of three kinds of objects. The **Model** is the application object, the **View** is its screen presentation, and the **Controller** defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. **MVC decouples them to increase flexibility and reuse.**"

MVC

Modèle-Vue-Contrôleur

- Modèle : données de l'application (logique métier)
- Vue : présentation des données du modèle
- Contrôleur : modification (actions utilisateur) des données

MVC : diagramme de classes UML

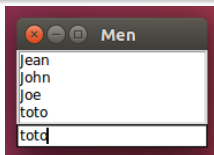


O. Boissier, G. Picard (SMA/G2I/ENS Mines Saint-Etienne)

MVC

Exemple : gestion d'une liste de noms

```
if __name__ == "__main__":  
    mw=tk.Tk()  
    mw.title("Men")  
    names=["Jean", "John", "Joe"]  
    model = Model(names)  
    view = View(mw)  
    view.update(model)  
    model.attach(view)  
    ctrl = Controller(model,view)
```



Modèle

Insertion, suppression de noms

```
class Model(Subject):
    def __init__(self, names=[]):
        Subject.__init__(self)
        self.__data = names
    def get_data(self):
        return self.__data
    def insert(self, name):
        self.__data.append(name)
        self.notify()                # obs.update(self)
    def delete(self, index):
        del self.__data[index]
        self.notify()                # obs.update(self)
```

Vue : l'Observer du modèle

Visualisation du modèle : update()

```
class View(Observer):
    def __init__(self, parent):
        self.parent = parent
        self.list = tk.Listbox(parent)
        self.list.configure(height=4)
        self.list.pack()
        self.entry = tk.Entry(parent)
        self.entry.pack()
    def update(self, model):
        self.list.delete(0, "end")
        for data in model.get_data():
            self.list.insert("end", data)
```


Contrôleur : du Subject à l'Observer

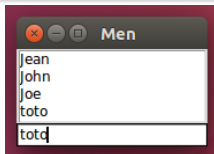
Contrôle du modèle : action utilisateur

```
class Controller(object):
    def __init__(self,model,view):
        self.model,self.view = model,view
        self.view.entry.bind("<Return>",
                               self.enter_action)
        self.view.list.bind("<Delete>",
                              self.delete_action)
    def enter_action(self, event):
        data = self.view.entry.get()
        self.model.insert(data)
    def delete_action(self, event):
        for index in self.view.list.curselection():
            self.model.delete(int(index))
```

Test IHM

Un modèle, une vue, un contrôleur

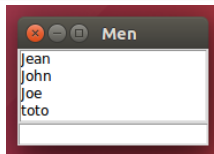
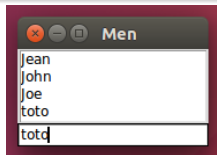
```
if __name__ == "__main__":  
    mw=tk.Tk()  
    mw.title("Men")  
    names=["Jean", "John", "Joe"]  
    model = Model(names)  
    view = View(mw)  
    view.update(model)  
    model.attach(view)  
    ctrl = Controller(model,view)
```



Test IHM

Un modèle, des vues, des contrôleurs

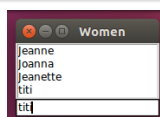
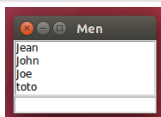
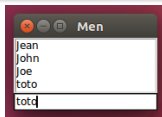
```
...
top = tk.Toplevel()
top.title("Men")
view = View(top)
view.update(model)
model.attach(view)
ctrl = Controller(model,view)
```



Test IHM

Des modèles, des vues, des contrôleurs

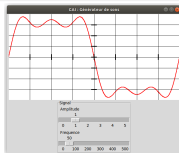
```
top = tk.Toplevel()
top.title("Women")
names=["Jeanne", "Joanna", "Jeanette"]
model = Model(names)
view = View(top)
view.update(model)
model.attach(view)
ctrl = Controller(model,view)
```



Générateur de signal

Objectifs : visualiser, contrôler des signaux

```
model=Generator()  
view=Screen(mw)  
view.grid(8)           # creation de la grille  
model.attach(view)  
model.generate_signal()  
ctrl=Controller(root,model,view)  
view.packing()          # redimensionnement (layout)  
ctrl.packing()          # redimensionnement (layout)
```

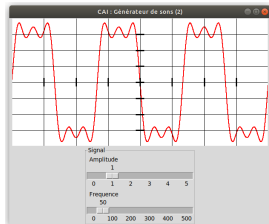
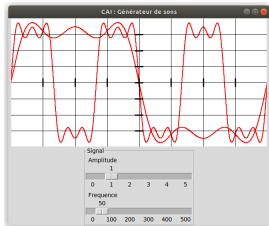


Générateur de signal

Objectifs : visualiser, contrôler un/des signaux

```
top = tk.Toplevel()
model=Generator()
view=Screen(top)
model.attach(view)
model.generate_signal()
ctrl=Controller(top,model,view)
```

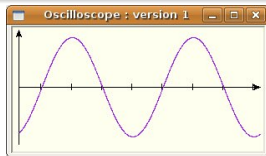
...



Création de signal

Son pur : mouvement vibratoire sinusoïdal

$$e = a \sin(2 \pi f t + \phi)$$



- e, t : élongation , temps
- a, f, ϕ : amplitude, fréquence, phase

Son complexe : mouvement vibratoire avec harmoniques

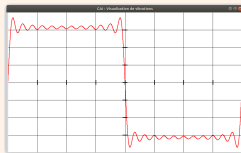
$$e = \sum_{h=1}^n (a/h) \sin(2 \pi (f * h) t + \phi)$$

Visualisation de signal

Application "Tout en Un" (Screen)

```
if __name__ == "__main__" :  
    mw=Tk()  
    view=Screen(mw)  
    view.grid(8)  
    view.packing()  
    view.update()  
    root.mainloop()
```

Visualisation du son avec harmoniques



Le Modèle et la Vue

Modèle (Subject) et Vue (Observer)

- Subject : `Generator.notify()`
- Observer : `Screen.update()`

Subject : le modèle à observer

```
class Generator(Subject) :  
    def __init__(self,name="signal"):  
        Subject.__init__(self)  
        self.name=name  
        self.signal=[]  
        self.a,self.f,self.p=1.0,1.0,0.0
```

Le Modèle et la Vue

Subject : calcul d'élongation

```
def vibration(self,t,harmoniques=1):  
    a,f,p=self.a,self.f,self.p  
    somme=0  
    for h in range(1,harmoniques+1) :  
        somme=somme + (a/h)*sin(2*pi*(f*h)*t-p)  
    return somme
```

Le Modèle et la Vue

Subject : générer le signal

```
def generate_signal(self, period=1, samples=1000):  
    del self.signal[0:]  
    duration=range(samples)  
    Te = period/samples  
    for t in duration :  
        self.signal.append([t*Te, self.vibration(t*Te)])  
    self.notify()  
    return self.signal
```

Le Modèle et la Vue

Observer : observation du modèle

```
class Screen(Observer):  
    def __init__(self, parent, \  
                  bg="white", width=600, height=300):  
        Observer.__init__(self)  
        self.canvas=Canvas(parent, \  
                             bg=bg, width=width, height=height)  
        self.signals={}  
        self.width, self.height=width, height  
        self.canvas.bind("<Configure>", self.resize)
```

Le Modèle et la Vue

Observer : notification du modèle

```
def update(self,subject=None):  
    if subject.get_name() not in self.signals.keys() :  
        self.signals[subject.get_name()]=\  
            subject.get_signal()  
    else :  
        self.canvas.delete(subject.get_name())  
self.plot_signal(subject.get_signal(),\  
                 subject.get_name())
```

Le Modèle et la Vue

Observer : visualisation du modèle

```
def plot_signal(self, signal, name, color="red"):
    w, h = self.width, self.height
    if signal and len(signal) > 1:
        plot = [(x*w, h/2.0*(1-y)) for (x, y) in signal]
        self.signal_id = self.canvas.create_line(plot, \
                                                    fill=color, \
                                                    smooth=1, \
                                                    width=2, \
                                                    tags=name)

    return self.signal_id
```

Le Modèle et la Vue

Observer : grille de visualisation

```
def grid(self,tiles=8):
    tile_x=self.width/tiles
    for t in range(1,tiles+1):
        x =t*tile_x
        self.canvas.create_line(x,0,\
                                x,self.height,\
                                tags="grid")
        self.canvas.create_line(x,self.height/2-10,\
                                x,self.height/2+10,\
                                width=3,\
                                tags="grid")
```

Le Modèle et la Vue

Observer : grille de visualisation

```
tile_y=self.height/tiles
for t in range(1,tiles+1):
    y =t*tile_y
    self.canvas.create_line(0,y,\
                            self.width,y,\
                            tags="grid")
    self.canvas.create_line(self.width/2-10,y,\
                            self.width/2+10,y,\
                            width=3,\
                            tags="grid")
```


Le Modèle et la Vue

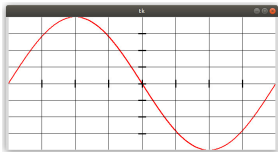
Observer : redimensionnement de la vue

```
def resize(self,event):
    if event:
        self.width,self.height=event.width,event.height
        self.canvas.delete("grid")
        for name in self.signals.keys():
            self.canvas.delete(name)
            self.plot_signal(self.signals[name],name)
        self.grid()
def packing(self) :
    self.canvas.pack(expand=1,fill="both",padx=6)
```

Le Modèle et la Vue

Application : création et visualisation de signal

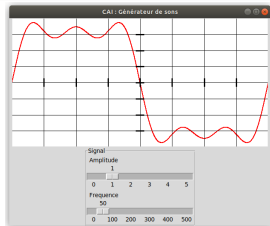
```
if __name__ == "__main__" :  
    mw=tk.Tk()  
    model=Generator()  
    view=Screen(mw)  
    view.grid(8)  
    view.packing()  
    model.attach(view)  
    model.generate_signal()  
    mw.mainloop()
```



Le Modèle, la Vue et le contrôleur

Contrôle et visualisation du Modèle

```
class Controller :  
    def __init__(self, parent, model, view):  
        self.model=model  
        self.view=view  
        self.create_controls(parent)
```



Le Contrôleur a accès au Modèle et à la Vue

Le Modèle, la Vue et le contrôleur

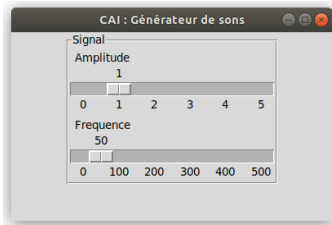
Contrôle et visualisation du Modèle

```
def create_controls(self,parent):  
    self.frame=LabelFrame(parent,text='Signal')  
    self.amp=IntVar()  
    self.amp.set(1)  
    self.scaleA=Scale(self.frame,variable=self.amp,  
                      label="Amplitude",  
                      orient="horizontal",length=250,  
                      from_=0,to=5,tickinterval=1)  
    self.scaleA.bind("<Button-1>",self.update_magnitude)  
    ...
```

Le Modèle, la Vue et le contrôleur

Contrôle et visualisation du Modèle

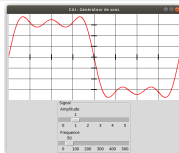
```
def update_magnitude(self,event):  
    self.model.set_magnitude(self.amp.get())  
    self.model.generate_signal()  
    ...  
def packing(self) :  
    self.frame.pack()  
    ...
```



Le Modèle, la Vue et le contrôleur

Un Modèle, une Vue et un Contrôleur

```
model=Generator()  
view=Screen(root)  
view.grid(8)  
model.attach(view)  
model.generate_signal()  
ctrl=Controller(root,model,view)  
view.packing()  
ctrl.packing()
```

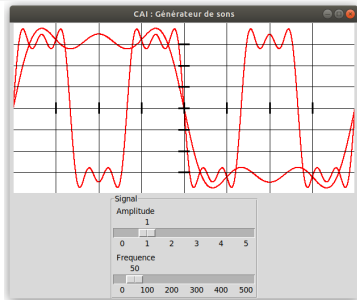


Le Modèle, la Vue et le contrôleur

Des Modèles, Une Vue et un Contrôleur

...

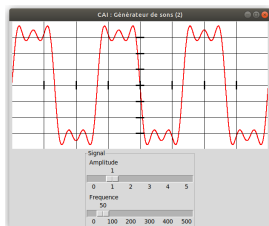
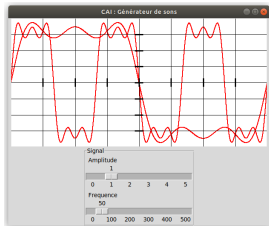
```
model=Generator("another signal")  
model.attach(view)  
model.set_frequency(3)  
ctrl=Controller(root,model,view)  
ctrl.packing()
```



Le Modèle, la Vue et le contrôleur

Des Modèles, des Vues et des Contrôleurs

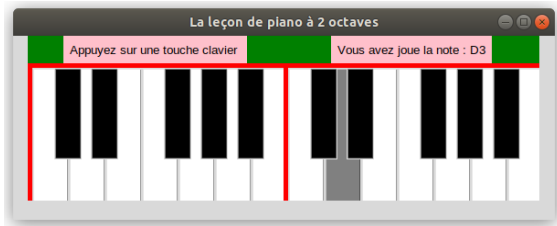
```
top=Toplevel()  
view=Screen(top)  
view.grid(10)  
model.attach(view)  
view.update(model)  
ctrl=Controller(top,model,view)
```



La leçon de Piano

Classes pour un piano MVC

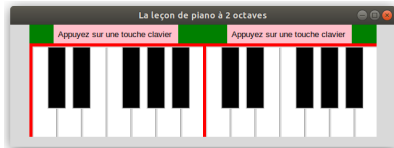
- Piano : nombre d'octaves
- Octave : le modèle à contrôler
- Keyboard : le clavier contrôlant le modèle
- Screen : pour jouer et visualiser la note jouée sur le clavier



La leçon de Piano

L'application

```
if __name__ == "__main__" :  
    mw = tk.Tk()  
    mw.geometry("360x300")  
    mw.title("La leçon de piano")  
    octaves=2  
    mw.title("La leçon de piano à " \  
            + str(octaves) + " octaves")  
    piano=Piano(mw,octaves)  
    mw.mainloop()
```



Classe Piano

Nombre d'octaves

```
class Piano :  
    def __init__(self, parent, octaves) :  
        self.parent=parent  
        self.octaves=[]  
        frame=tk.Frame(self.parent, bg="yellow")  
        for octave in range(octaves) :  
            self.create_octave(self.frame, octave+2)  
        frame.pack(fill="x")
```

Chaque Octave correspondra à un **modèle** auquel sera associé :

- un clavier (Keyboard) pour **contrôler** le modèle
- un écran (Screen) pour **visualiser** et jouer la note contrôlée

Classe Piano

Modèle-Vue-Contrôleur

```
def create_octave(self,parent,degree=3) :  
    model=Octave(degree)  
    control=Keyboard(parent,model)  
    view=Screen(parent)  
    model.attach(view)  
    control.get_keyboard().grid(column=degree,row=0)  
    view.get_screen().grid(column=degree,row=1)  
    self.octaves.append(model)
```

Classe Octave

Le Modèle

```
class Octave(Subject) :  
    def __init__(self,degree=3) :  
        Subject.__init__(self)  
        self.degree=degree  
        self.set_sounds_to_gamme(degree)  
    def get_gamme(self) :  
        return self.gamme  
    def get_degree(self) :  
        return self.degree  
    def notify(self,key) :  
        for obs in self.observers:  
            obs.update(self,key)
```

Classe Octave

Le Modèle

```
def set_sounds_to_gamme(self,degree=3) :  
    self.degree=degree  
    folder="Sounds"  
    notes=["C","D","E","F","G","A","B",\  
           "C#","D#","F#","G#","A#"]  
    self.gamme=collections.OrderedDict()  
    for key in notes :  
        self.gamme[key]="Sounds/"+key+str(degree)+".wav"  
    return self.gamme
```

Vérifier l'existence :

- du répertoire Sounds
- des fichiers au format wav sous ce répertoire

Classe Screen

La Vue

```
class Screen(Observer):
    def __init__(self,parent) :
        self.parent=parent
        self.create_screen()
    def create_screen(self) :
        self.screen=tk.Frame(self.parent,\
                               borderwidth=5,\
                               width=500,height=160,\
                               bg="pink")
        self.info=tk.Label(self.screen,\
                             text="Appuyez sur une touche clavier ",\
                             bg="pink",font=('Arial',10))
        self.info.pack()
```

Classe Screen

La Vue

```
def get_screen(self) :  
    return self.screen  
def update(self,model,key="C") :  
    if __debug__:  
        if key not in model.gamme.keys() :  
            raise AssertionError  
    subprocess.call(["aplay",model.get_gamme()[key]])  
    if self.info :  
        self.info.config(text="Vous avez joue la note: \"\n\n            + key + str(model.get_degree())  
        )
```

aplay : Advanced Linux Sound Architecture

Classe Keyboard

Le Contrôleur

```
class Keyboard :
    def __init__(self,parent,model) :
        self.parent=parent
        self.model=model
        self.create_keyboard()
    def create_keyboard(self) :
        key_w,key_h=40,150
        dx_white,dx_black=0,0
        self.keyboard=tk.Frame(self.parent,\
                                borderwidth=5,\
                                width=7*key_w,\
                                height=key_h,bg="red")
```

Classe Keyboard

Le Contrôleur

```
for key in self.model.gamme.keys() :
    if key.startswith('#',1,len(key)) :
        delta_w,delta_h=3/4.,2/3.
        delta_x=3/5.
        button=tk.Button(self.keyboard,\
                           name=key.lower(),\
                           width=3,height=6,bg="black")
        button.bind("<Button-1>",\
                     lambda event,x=key : self.play_note(x))
        button.place(width=key_w*delta_w,\
                      height=key_h*delta_h,\
                      x=key_w*delta_x+key_w*dx_black,y=0)
```

Classe Keyboard

Le Contrôleur

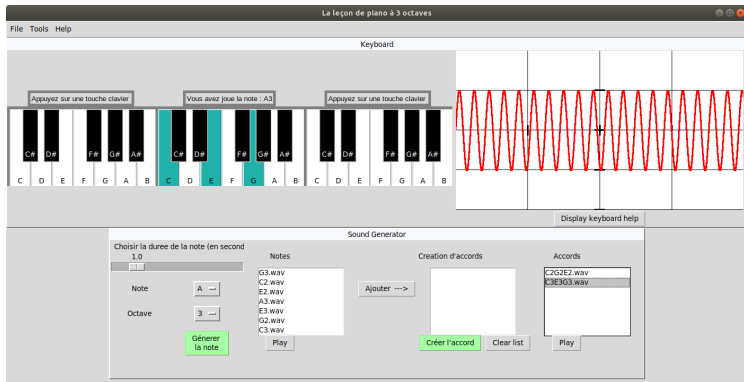
```
    if key.startswith('D#', 0, len(key)) :
        dx_black=dx_black+2
    else :
        dx_black=dx_black+1
else :
    button=tk.Button(self.keyboard,\
                      name=key.lower(),\
                      bg = "white")
    button.bind("<Button-1>",\
               lambda event,x=key : self.play_note(x))
    button.place(width=key_w,height=key_h,\
                 x=key_w*dx_white,y=0)
    dx_white=dx_white+1
```

Classe Keyboard

Le Contrôleur

```
def play_note(self, key) :  
    self.model.notify(key)  
def get_keyboard(self) :  
    return self.keyboard  
def get_degrees(self) :  
    return self.degrees
```

Exemple d'IHM pour piano



Format wav : Fabrice Sincère

Conclusion

Création d'Interfaces Homme-Machine

- un langage de programmation (python)
- une bibliothèque de composants graphiques (TkInter)
- gestion des événements (composant-événement-action)
- programmation des actions (callbacks, fonctions réflexes)
- création de nouveaux composants, d'applications
- mise en œuvre des patrons de conception (Observer, MVC)
- critères ergonomiques des IHM (Norme AFNOR Z67-110)

Annexes : python

Initialisation : variables, fonctions

```
# Importation de variables, fonctions, modules externes
import sys
from math import sqrt, sin, acos
# Variables, fonctions nécessaires au programme
def spherical(x,y,z):
    r, theta, phi = 0.0, 0.0, 0.0
    r = sqrt(x*x + y*y + z*z)
    theta = acos(z/r)
    if theta == 0.0:
        phi = 0.0
    else :
        phi = acos(x/(r*sin(theta)))
    return r, theta, phi
```

Annexes : python

Traitements de données, sortie de programme

```
# Traitements
x = input("Entrez la valeur de x : ")
y = input("Entrez la valeur de y : ")
z = input("Entrez la valeur de z : ")
print "Les coordonnees spheriques du point :", x,y,z
print "sont : ", spherical(x,y,z)
# sortie de programme
sys.exit(0)
```


Annexes : python

Définition d'une classe

```
class Point:
    """point 2D"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return "<Point('%s', '%s')>" \
            % (self.x, self.y)
```

Annexes : python

Association entre classes

```
class Rectangle:
    """Un rectangle A UN coin (Point) superieur gauche"""
    def __init__(self, coin, largeur, hauteur):
        self.coin = coin
        self.largeur = largeur
        self.hauteur = hauteur
    def __repr__(self):
        return "<Rectangle('%s','%s','%s')>" \
            % (self.coin,self.largeur, self.hauteur)
```

Annexes : python

Heritage de classe

```
class Carre(Rectangle):  
    """Un carre EST UN rectangle particulier"""  
    def __init__(self, coin, cote):  
        Rectangle.__init__(self, coin, cote, cote)  
#        self.cote = cote  
    def __repr__(self):  
        return "<Carre('%s', '%s')>" \  
            % (self.coin, self.largeur)
```

Annexes : python

Application de test

```
if __name__ == '__main__':  
    p=Point(10,10)  
    print(p)  
    print(Rectangle(p,100,200))  
    print(Carre(p,100))
```

Lancement de l'application

```
{logname@hostname} python classes.py  
<Point('10','10')>  
<Rectangle('<Point('10','10')>','100','200')>  
<Carre('<Point('10','10')>','100')>
```

Création et visualisation

Tout en un : le Modèle et la Vue

```
from math import sin,pi
## from pylab import linspace,sin
from tkinter import Tk,Canvas
class Screen :
    def __init__(self,parent,\
                  bg="white",width=600,height=300):
        self.canvas=Canvas(parent,\
                             bg=bg,width=width,height=height)
        self.a,self.f,self.p=1.0,2.0,0.0
        self.signal=[]
        self.width,self.height=width,height
        self.units=1
        self.canvas.bind("<Configure>",self.resize)
```

Création et visualisation

Calcul de vibration

```
def vibration(self,t,harmoniques=1):  
    a,f,p=self.a,self.f,self.p  
    somme=0  
    for h in range(1,harmoniques+1) :  
        somme=somme + (a/h)*sin(2*pi*(f*h)*t-p)  
    return somme
```

Son pur de fréquence f avec ses harmoniques $(f * h)$:

$$e = \sum_{h=1}^n (a/h) \sin(2 \pi (f * h) t + \phi)$$

Création et visualisation

Génération de son

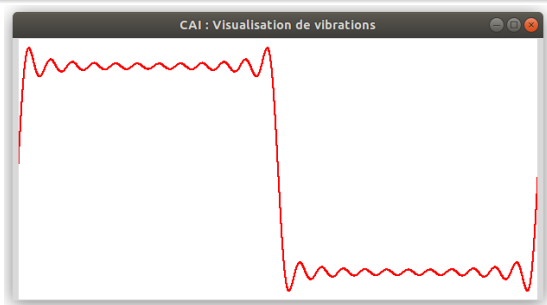
```
def generate_signal(self, period=2, samples=10):  
    del self.signal[0:]  
    echantillons=range(int(samples)+1)  
    Tech = period/samples  
    for t in echantillons :  
        self.signal.append(  
            [t*Tech, self.vibration(t*Tech)]  
        )  
    return self.signal
```

Son échantillonné (`samples`) sur un nombre de périodes (`period`)

Création et visualisation

Génération du signal et affichage

```
def update(self):  
    self.generate_signal()  
    if self.signal :  
        self.plot_signal(self.signal)
```



Création et visualisation

Génération du signal et affichage

```
def plot_signal(self, signal, color="red"):
    w, h = self.width, self.height
    signal_id = None
    if signal and len(signal) > 1:
        plot = [(x*w, h/2.0*(1-y/(self.units/2))) \
                for (x, y) in signal]
        signal_id = self.canvas.create_line(plot, \
                                             fill=color, smooth=1, \
                                             width=3, tags="curve")
    return signal_id
```

Création et visualisation

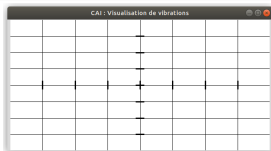
Affichage de la grille

```
def grid(self,tiles=2):
    self.units=tiles
    tile_x=self.width/tiles
    for t in range(1,tiles+1):
        x =t*tile_x
        self.canvas.create_line(x,0,\
                                x,self.height,\
                                tags="grid")
        self.canvas.create_line(x,self.height/2-10,\
                                x,self.height/2+10,\
                                width=3,tags="grid")
```

Création et visualisation

Affichage de la grille

```
tile_y=self.height/tiles
for t in range(1,tiles+1):
    y =t*tile_y
    self.canvas.create_line(0,y,\
                            self.width,y,\
                            tags="grid")
    self.canvas.create_line(self.width/2-10,y,\
                            self.width/2+10,y,\
                            width=3,tags="grid")
```



Création et visualisation

Redimensionnement de la vue

```
def resize(self,event):  
    if event:  
        self.width,self.height=event.width,event.height  
        self.canvas.delete("grid")  
        self.canvas.delete("curve")  
        self.plot_signal(self.signal)  
        self.grid(self.units)  
def packing(self) :  
    self.canvas.pack(expand=1,fill="both",padx=6)
```

Création et visualisation

Le Modèle et la Vue

Séparer le traitement des données de leur visualisation

- **Generator** : les données à observer (**Subject**)
- **Screen** : leur visualisation (**Observer**)

Application : créer et visualiser des sons

```
root=Tk()  
model=Generator()  
model.generate_signal()  
view=Screen(root)  
view.grid(10)  
view.packing()  
model.attach(view)  
root.mainloop()
```

Bibliographie

Documents

- Gérard Swinnen :
“Apprendre à programmer avec Python 3” (2010)
- Guido van Rossum :
“Tutoriel Python Release 2.4.1” (2005)
- Mark Pilgrim :
“An introduction to Tkinter” (1999)
- John W. Shipman :
“Tkinter reference : a GUI for Python” (2006)
- John E. Grayson :
“Python and Tkinter Programming” (2000)
- Bashkar Chaudary :
“Tkinter GUI Application Development Blueprints” (2015)

Bibliographie

Adresses “au Net”

- inforef.be/swi/python.htm
- <https://docs.python.org/fr/3/library/tk.html>
- wiki.python.org/moin/TkInter
- www.jchr.be/python/tkinter.htm
- www.pythonware.com/library/tkinter/introduction
- <https://www.thomaspietrzak.com/teaching/IHM>
- http://fsincere.free.fr/isn/python/cours_python_tkinter.php