# Conception d'Applications Interactives

## développement d'IHM en TkInter

*Alexis NEDELEC*

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



# Objectifs du cours

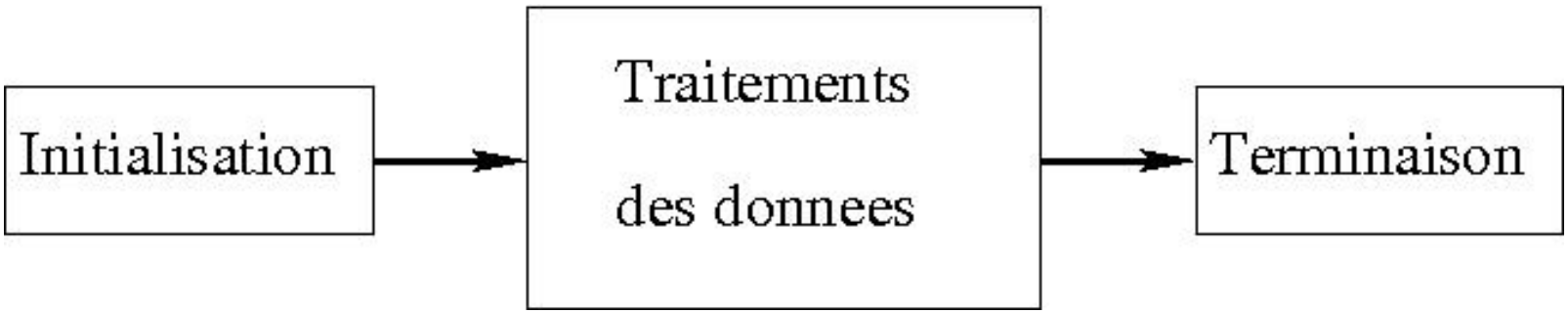
## Programmation d'Interfaces Homme-Machine

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 classique

## Les trois étapes séquentielles

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



# Programmation événementielle

## 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
   - éxécution des traitements liés à l'action de l'utilisateur
3. terminaison
   - sortir "proprement" de l'application

# Programmation événementielle

## Commande classique :interaction faible

peu d'interaction (textuelle) avec l'utilisateur

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

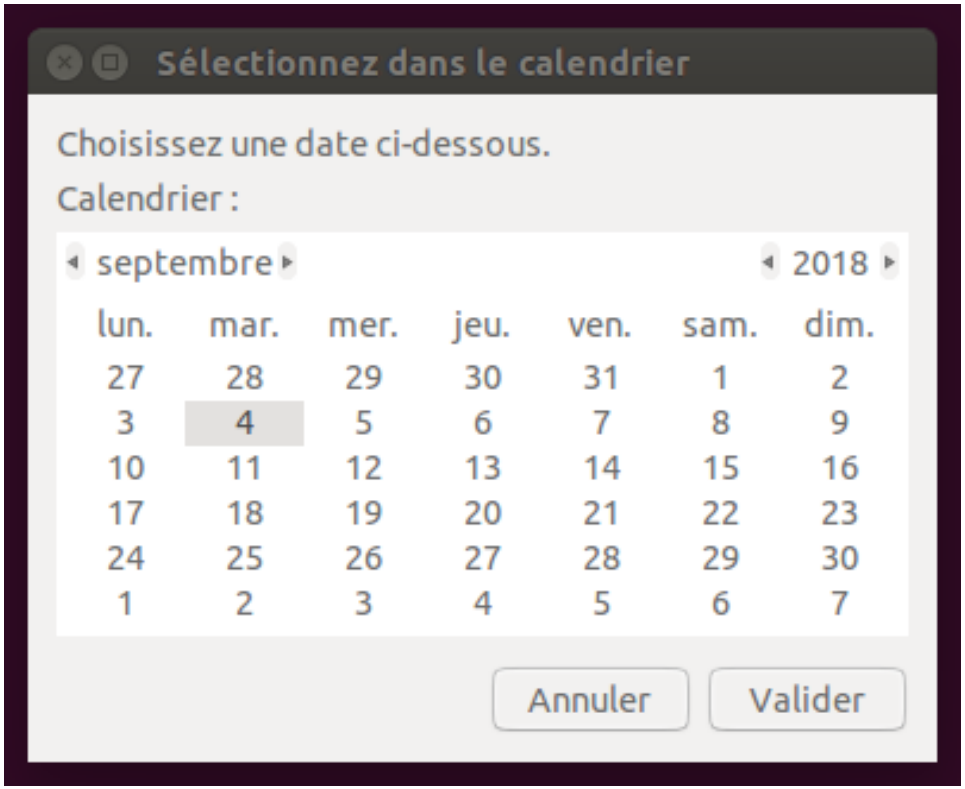
```
   Septembre 2018
di lu ma me je ve sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

# Programmation événementielle

## IHM : forte interaction forte

L'application est à l'écoute de l'utilisateur

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



# Programmation événementielle

## A l'écoute de l'utilisateur

- à l'écoute des périphériques (clavier, souris ...)
- réaction suivant l'arrivée d'un événement
- événement détecté suivant l'action d'un utilisateur
- envoi d'un message au programme
- éxécution d'un bloc de code (fonction) spécifique

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

# Python/TkInter : Hello World

## hello.py

```
from Tkinter import Tk,Label,Button
mw=Tk()
labelHello=Label(mw, text="Hello World !",fg="blue")
labelHello.pack()
buttonQuit=Button(mw, text="Goodbye World", fg="red",\
                        command=mw.destroy)
buttonQuit.pack()
mw.mainloop()
exit(0)
```

# Python/TkInter : Hello World

## Création de la fenêtre principale et de composants

- `mw=Tk()`
- `labelHello=Label(mw, ...)`
- `buttonQuit=Button(mw, ...)`

## Interaction sur le composant

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

## Affichage : positionnement des composants

- `labelHello.pack(), buttonQuit.pack()`

## Boucle d'événements : en fin de programme

- `mw.mainloop()`

# Python/TkInter : Personnalisation d'IHM

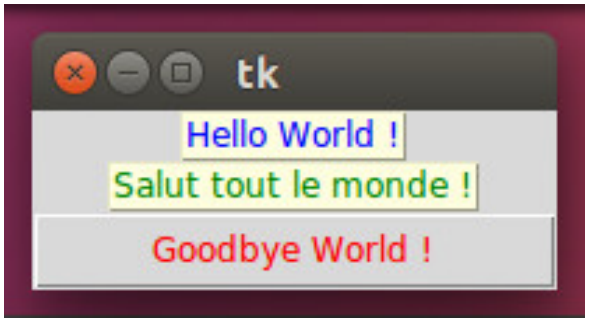
## Fichier d'option : chargement dans une application

```
from Tkinter import Tk,Label,Button
mw=Tk()
mw.option_readfile('hello.txt')
labelHello=Label(root,text="Hello World !")
labelBonjour=Label(root,name="labelBonjour")
buttonQuit=Button(root,text="Goodbye World !")
...
```

# Python/TkInter : Personnalisation d'IHM

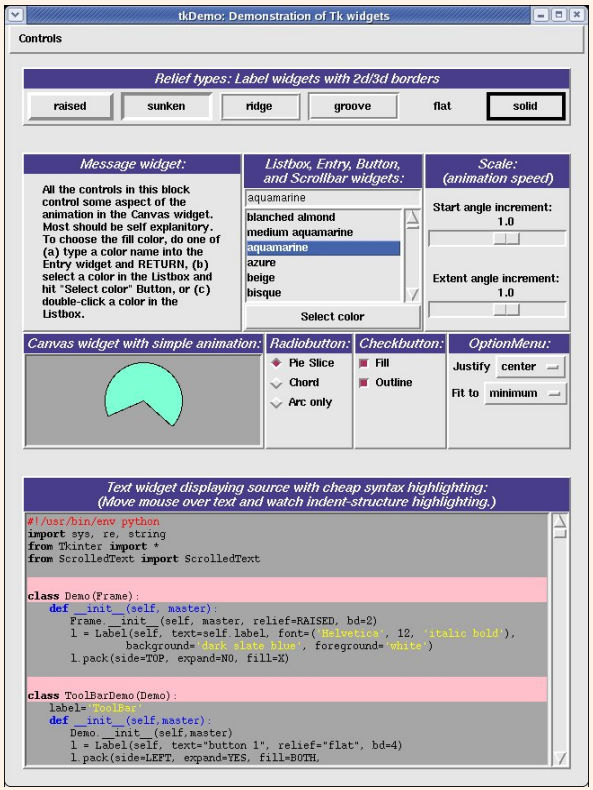
## Fichier d'option : contenu (`hello.txt`)

```
*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

## TkInter : Tk (de Tcl/Tk) pour python



# Composants graphiques

## Widgets : **W**indow ga**dgets**

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

# Etapes de programmation

## Structuration d'un programme

```
# ------- Initialisation ------------------------
from Tkinter import Tk,Label,Button
# ------- Composants graphiques -------------------
mw=Tk()
labelHello=Label(mw,text="Hello World !",fg="blue")
buttonQuit=Button(mw,text="Goodbye World",fg="red",
                  command=mw.destroy)
# ------- Positionnement des composants -----------
labelHello.pack()
buttonQuit.pack()
# ------- Definition des actions      -------------
# ------- Liaison composant-événement-action  ------
mw.mainloop()
exit(0)
```

# Gestion d'événements : callbacks

## Interaction par défaut : option `command`

- en cas de "click gauche" exécuter la fonction associée

## Paramétrer l'interaction utilisateur : méthode `bind()`

- lier (bind) l'événement au comportement d'un composant

## gestion des interactions

```
# ------- Definition des actions      --------------
def callback(event) :
    mw.destroy()
# ------- Liaison composant-événement-action  ------
buttonQuit.bind("<Button-1>",callback)
mw.mainloop()
exit(0)
```

# 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

## Récupération d'informations

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

# Gestion d'événements

## Callback : implémentation de l'action

```
# ------- Definition des actions      --------------
def callback_info(event) :
    print(dir(event))
    print(event.widget.cget("text"))
    event.widget.configure(text="x="+str(event.x)\
                                +"y="+str(event.y))
    print("position of the mouse on the screen",
          event.x_root,event.y_root)
# ------- Liaison composant-événement-action  ------
buttonQuit.bind("<Button-1>",callback_info)
```

# Gestion d'événements

## Callback : passage d'arguments

```
# ------- Definition des actions      --------------
def callback_data(event,argument) :
    print(argument)
# ------- Liaison composant-événement-action  ------
data="hello"
buttonQuit.bind( "<Button-1>",
                 lambda event,argument=data : \
                 callback_data(event,argument) )
```

# Gestion d'événements


## Callback : passage d'arguments

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

# Gestion d'événements

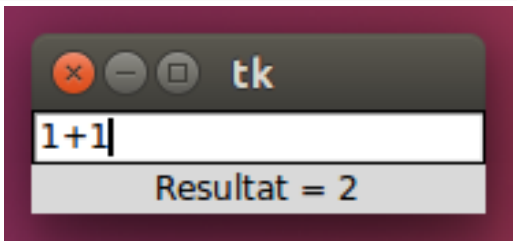
## Implémentation de l'action

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


# Gestion d'événements

## Traitement des données Utilisateur

```
# ------- Initialisation ----------------------------
from Tkinter import Tk,Entry,Label
from math import *
# ------- Composants graphiques ---------------------
mw = Tk()
entry = Entry(mw)
label = Label(mw)
# ------- Positionnement des composants ------------
entry.pack()
label.pack()
```



# Gestion d'événements

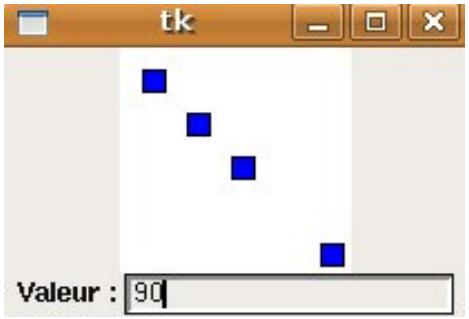
## Traitement des données Utilisateur

```
# ------- Definition des actions      --------------
def evaluer(event):
    label.configure(text = "Resultat = "
                           + str(eval(entry.get()))
                    )
# ------- Liaison composant-événement-action  ------
entry.bind("<Return>", evaluer)
mw.mainloop()
```

# Gestion d'événements

## Émission d'événements :event_generate()

```
from Tkinter import Tk,Canvas,Label,Entry
mw = Tk()
canvas=Canvas(mw,width=100,height=200,bg="white",bd=1)
label= Label(mw, text = "Valeur :")
entry = Entry(mw)
canvas.pack()
label.pack(side="left")
entry.pack(side="left")
```



# Gestion d'événements

## Émission d'événements :event_generate()

```
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>')

mw.bind("<Control-Z>", display)
entry.bind("<Return>", set_value)
mw.mainloop()
```

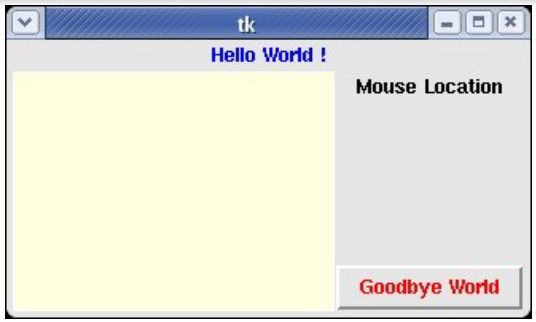
# Positionnement de composants

## TkInter : Layout manager

- `pack()` : "coller" les widgets par leur côté
- `grid()` : agencer en ligne/colonne
- `place()` : positionner géométriquement

## `pack()` : exemple de positionnement

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



# Positionnement de composants

## Frame : agencement de composants graphiques

```
frameCanvas = Frame(mw,bg="yellow")
canvas = Canvas(frameCanvas, width=200, height=150,\
                bg="light yellow")
labelPosition = Label(frameCanvas,text="Mouse Location")
labelHello.pack()
frameCanvas.pack(fill="both",expand=1)
buttonQuit.pack()
canvas.pack(fill="both",expand=1)
labelPosition.pack()
```

# Positionnement de composants

## grid() : exemple de positionnement

```
labelNom = Label(mw, text = "Nom :")
labelPrenom = Label(mw, text = "Prenom :")
entryNom = Entry(mw)
entryPrenom = Entry(mw)
labelNom.grid(row=0)
labelPrenom.grid(row=1)
entryNom.grid(row=0,column=1)
entryPrenom.grid(row=1,column=1)
```

# Positionnement de composants

## place() : exemple de positionnement

```
mw.title("Layout Manager : Place")
msg = Message(mw, text="Place : \n
                  options de positionnement de widgets",
                  justify="center",
                  bg="yellow", relief="ridge")
okButton=Button(mw,text="OK")

msg.place(relx=0.5,rely=0.5,
          relwidth=0.75,relheight=0.50,
          anchor="center")
okButton.place(relx=0.5,rely=1.05,
               in_=msg,
               anchor="n")
```

# Positionnement de composants

## place() : exemple de positionnement

```
def move(event):
    print "Deplacement sur la fenetre X =" \
          + str(event.x_root) \
          + ", Y =" + str(event.y_root)
def position(event):
    print "Position sur le composant X =" \
          + str(event.x) + ", Y =" + str(event.y)
msg.bind("<Motion>", move)
msg.bind("<ButtonPress>", position)
```

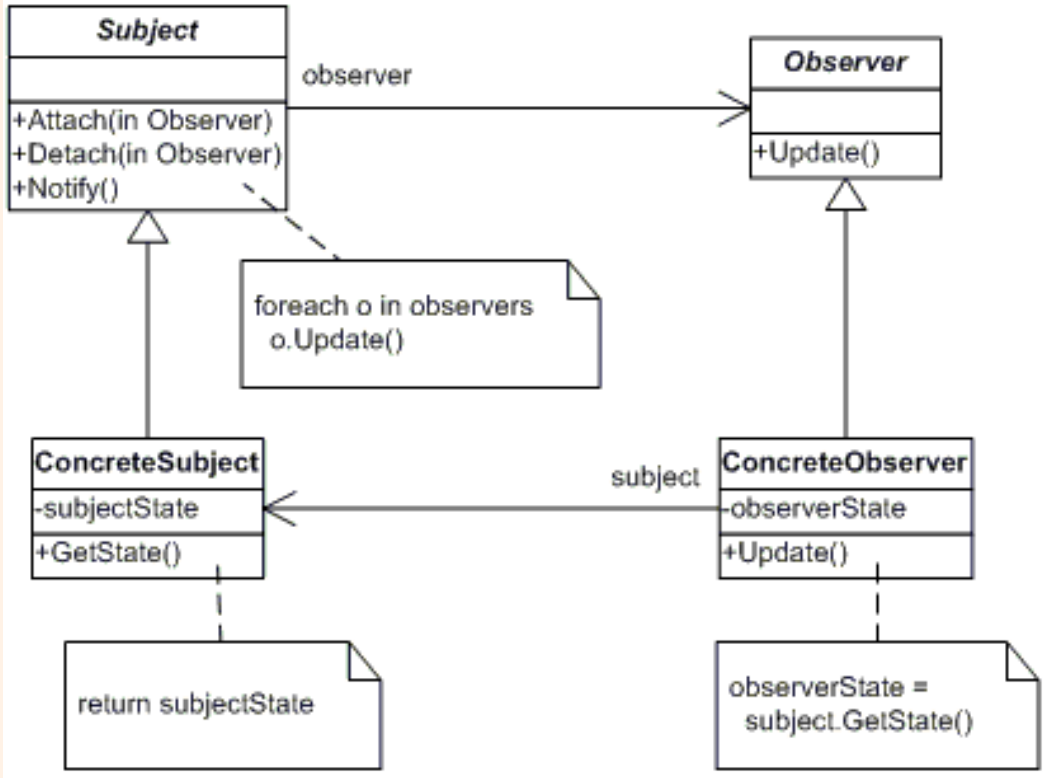
# Patrons de conception

## 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()
    def distribute(self,amount):
        self.amount=self.amount-amount
        self.notify()
```

# 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

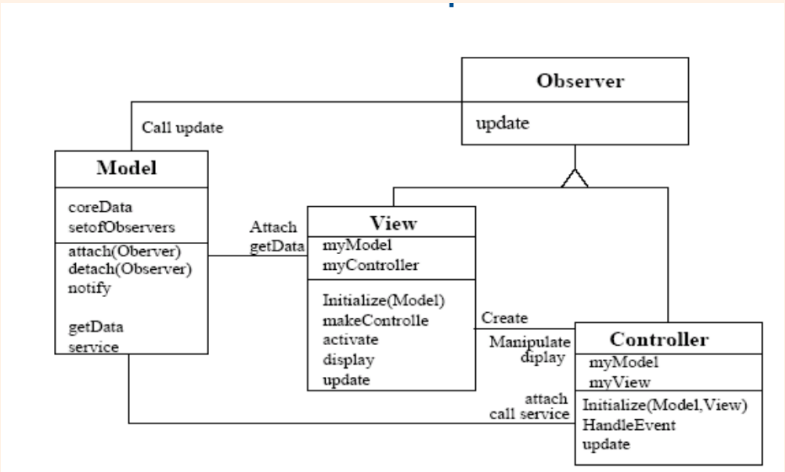
"MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility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

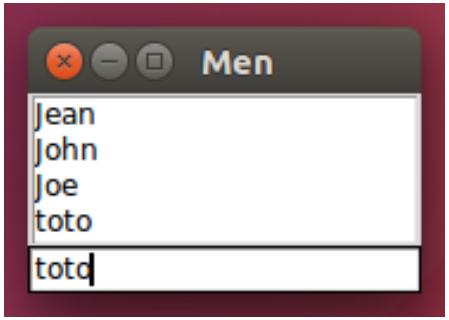



O. Boissier, G. Picard (SMA/G2I/ENS Mines Saint-Etienne)

# Exemple d'application

## gestion d'une liste de noms

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



# Modèle : le Subject observé

## Modification du modèle : Subject.notify(self)

```
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()
    def delete(self, index):
        del self._data[index]
        self.notify()
```

# 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

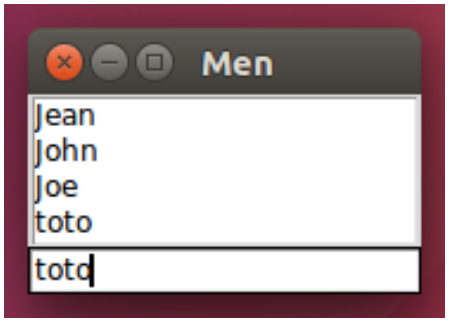
## Contrôle du modèle : action utilisateur

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

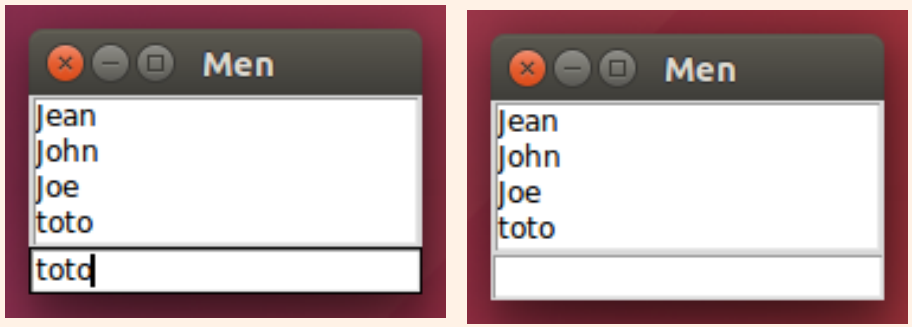


# Test IHM

## 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 : un modèle, des vues, des contrôleurs

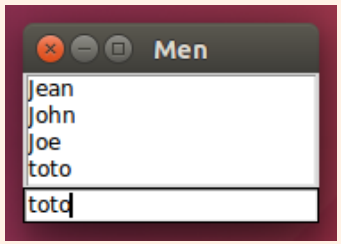


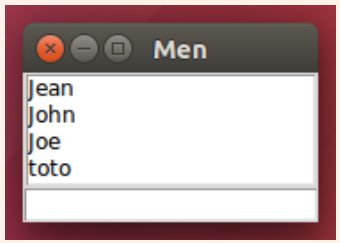
# Test IHM

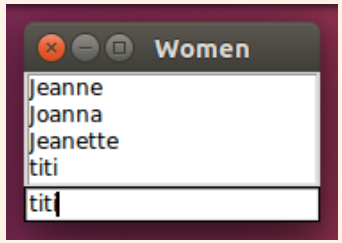
## Des modèles, des vues, des contrôleurs

```
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)
```

## Test : des modèles, des vues, des contrôleurs



# Courbes de Bézier (cf : annexe)

## Programme d'application

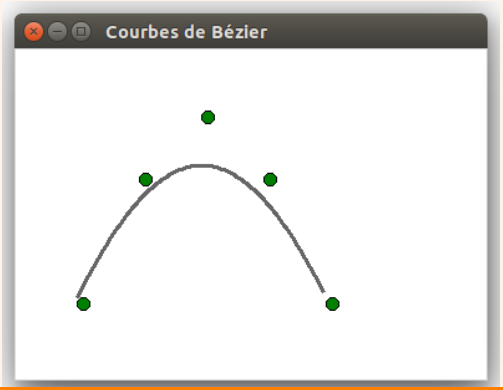
```
if __name__ == "__main__" :
    root=Tk()
    control_points=[(50,200),(100,100),
                    (150,50),(200,100),(250,200)]
    bezier=Bezier(control_points)
    bezier.compute_curve()
    screen=Screen(root,bezier)
    bezier.attach(screen)
    bezier.notify()
    screen.update_control_points(bezier)
    screen.packing()
    canvas=screen.get_canvas()
```

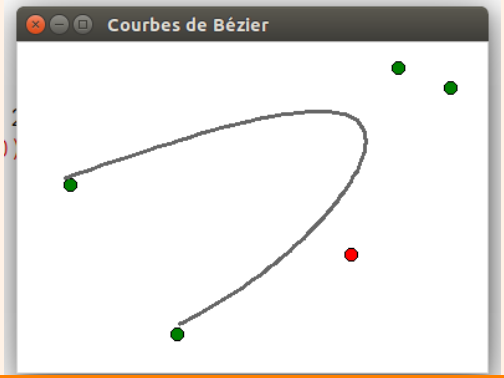
# Courbe de Bézier (cf : annexe)

## Programme d'application

```
canvas.bind("<Button-1>",
            lambda event,model=bezier :
               screen.select_point(event,model))
canvas.bind("<Motion>",
            lambda event,model=bezier :
               screen.move_point(event,model))
canvas.bind("<ButtonRelease>",screen.release_point)
```

## Visualisation



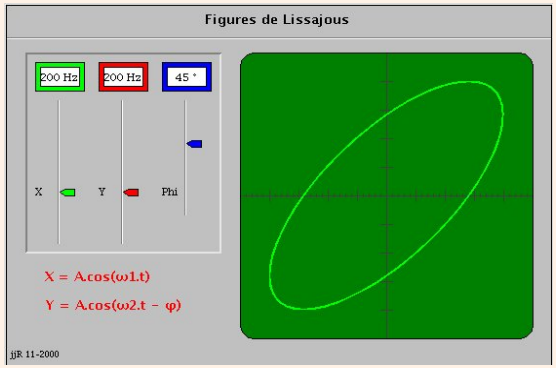



# Oscilloscope

## Principe

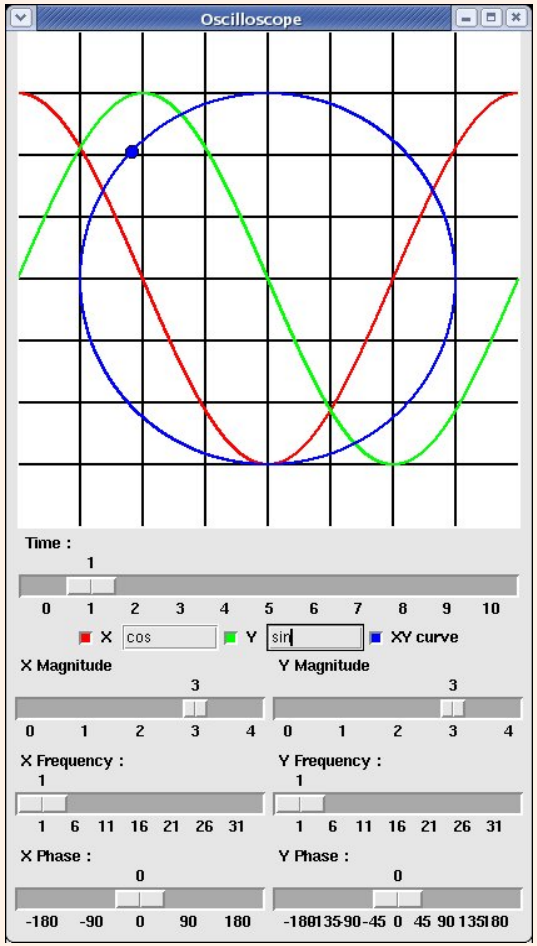
- visualisation de mouvement vibratoire harmonique
- contrôle en Amplitude, Fréquence et Phase
- gestion de la base de temps
- oscilloscope en mode XY

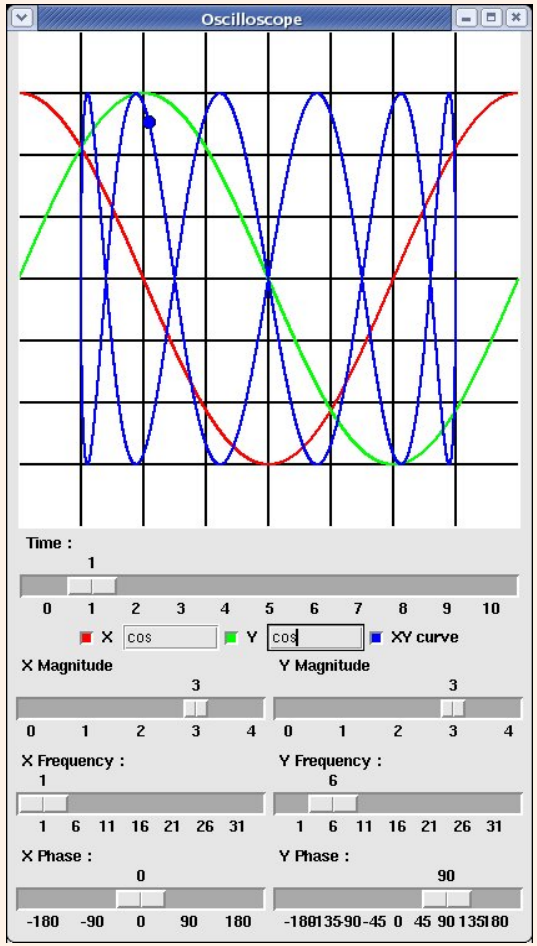
## Exemple



# Oscilloscope

## Autre exemple



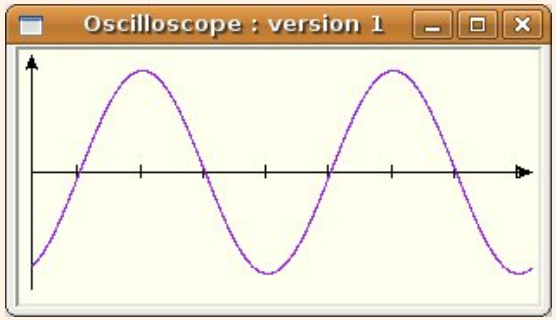
# Oscilloscope : Modèle

## Mouvement vibratoire harmonique

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

- $e$ : élongation
- $A$ : amplitude
- $f$ : fréquence
- $t$ : temps
- $\phi$ : phase

## Représentation



# Oscilloscope : Modèle

## Générateur de signal

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

# Oscilloscope : Modèle

## Générateur de signal

```
    def set_magnitude(self,a):
        self.a=a
        self.generate_signal()
    def generate_signal(self):
        del self.signal[0:]
        samples=1000
        for t in range(0, samples,5):
            samples=float(samples)
            e=self.a*sin((2*pi*self.f*(t*1.0/samples))
                         -self.p)
            self.signal.append((t*1.0/samples,e))
        self.notify()
        return self.signal
```

# Oscilloscope : Contrôleur

## Contrôleur de signal

```
class Controller :
    def __init__(self,model,view):
        self.model=model
        self.view=view
        self.view.magnitude.bind("<B2-Motion>",
                                 self.update_magnitude)
    def update_magnitude(self,event):
        x=int(event.widget.get())
        self.model.set_magnitude(x)
        self.model.generate_signal()
    ...
```

# Oscilloscope : Vue

## Ecran d'oscilloscope

```
class Screen(Observer):
    def __init__(self,parent,bg="white"):
        self.canvas=Canvas(parent,bg=bg)
        self.signal_id=None
        self.magnitude=Scale(parent,length=250,
                             orient="horizontal",
                             label="Magnitude",
                             sliderlength=20,
                             showvalue=0,from_=0,to=5,
                             tickinterval=25)
        ...
```

# Oscilloscope : Vue

## Mise à jour du modèle

```
def update(self,model):
    signal=model.get_signal()
    self.plot_signal(signal)
def plot_signal(self,signal,color="red"):
    signal_id=None
    w=self.canvas.cget("width")
    h=self.canvas.cget("height")
    width,height=int(w),int(h)
    if self.canvas.find_withtag("signal") :
        self.canvas.delete("signal")
```

# Oscilloscope : Vue

## Affichage du signal

```
        if signal and len(signal) > 1:
            plot=[ (x*width, height/2.0*(y+1))
                    for (x, y) in signal ]
            signal_id=
                self.canvas.create_line(plot,
                                        fill=color,
                                        smooth=1,
                                        width=3,
                                        tags="signal")
        return signal_id
```

# Oscilloscope : Vue

## Grille de visualisation

```
    def grid(self, steps):
        w=self.canvas.cget("width"),
        h=self.canvas.cget("height")
        width,height=int(w),int(h)
        self.canvas.create_line(10,height/2,
                                width,height/2,
                                arrow="last")
        self.canvas.create_line(10,height-5,
                                10,5,arrow="last")
```

# Oscilloscope : Vue

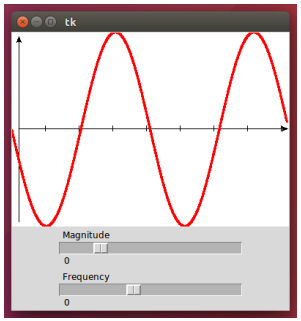
## Affichage du signal

```
step=(width-10)/steps*1.
for t in range(1,steps+2):
    x =t*step
    self.canvas.create_line(x,height/2-4,
                            x,height/2+4)
```

# Oscilloscope : test IHM

## Programme d'application

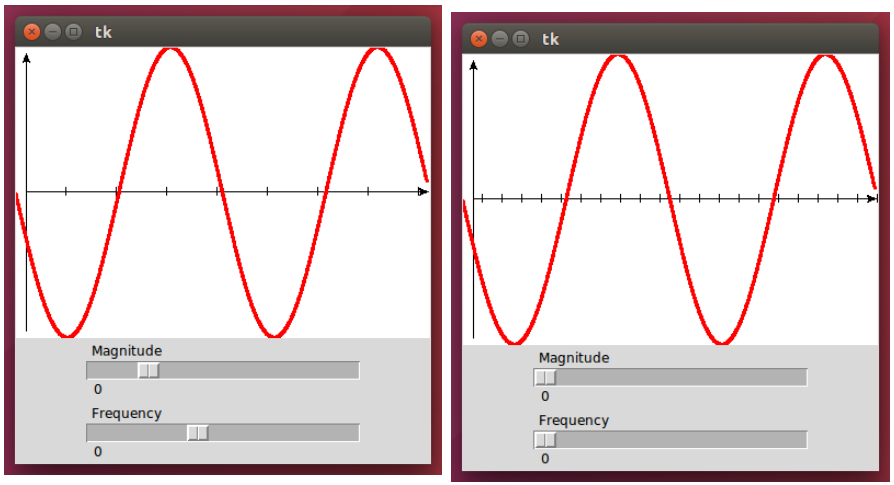
```
if __name__ == "__main__" :
    ...
    model=Generator()
    view=Screen(root)
    view.grid(8)
    model.attach(view)
    view.update(model)
    ctrl=Controller(root,model,view)
    ...
```



# Oscilloscope : test IHM

## Programme d'application

```
if __name__ == "__main__" :
    ...
    top=Toplevel(root)
    view=Screen(top,model)
    view.grid(20)
    view.packing()
    model.attach(view)
```



# Proies-Prédateur : Modèle

## Équations de Lotka-Volterra

$$\frac{dx(t)}{dt} = x(t).(\alpha - \beta.y(t))$$

$$\frac{dy(t)}{dt} = -y(t).(\gamma - \delta.y(t))$$

- $t$ : le temps
- $x(t)$ : effectif des proies en fonction du temps
- $y(t)$ : effectif des prédateurs en fonction du temps
- $\mathrm{d}x(t)/\mathrm{d}t, \mathrm{d}y(t)/\mathrm{d}t$ : variation de population au cours du temps

# Proies-Prédateur : Modèle

Paramètres caractérisant les interactions entre les deux espèces :

- $\alpha$ : taux de reproduction des proies
  - constant, indépendant du nombre de prédateurs
- $\beta$ : taux de mortalité des proies
  - dû aux prédateurs rencontrés
- $\gamma$ : taux de mortalité des prédateurs
  - constant, indépendant du nombre de proies
- $\delta$ : taux de reproduction des prédateurs
  - lié aux proies rencontrées et mangées

# Proies-Prédateur : Implémentation

## Programme d'application

```
# Jose OUIN : algorithmique et calcul numerique,
# ellipses 2013
import matplotlib.pyplot as plt
import numpy as np

def proies(u,v,a,b) :
  return u*(a - b*v)
def predateurs(u,v,c,d) :
  return v*(-c + d*u)

if __name__ == "__main__" :
  a,b=0.8,0.4
  c,d=0.6,0.2
```

# Proies-Prédateur : Implémentation

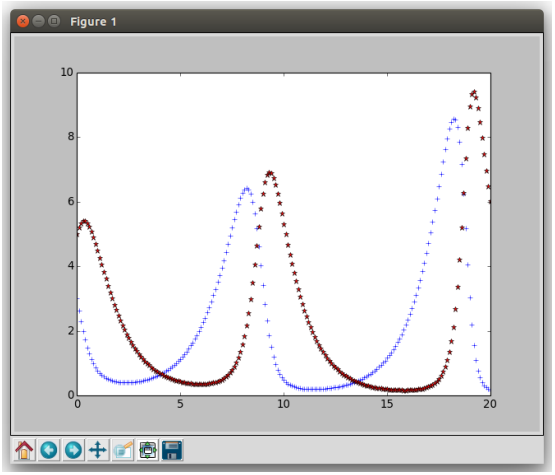
## Programme d'application

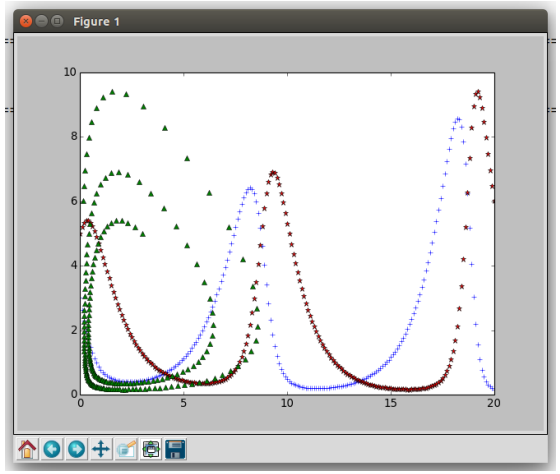
```
u0,v0=3,5
start,stop=0.0,20.0
n=200
h = (stop-start)/n
t = np.zeros(n+1)
u = np.zeros(n+1)
v = np.zeros(n+1)

t[0],u[0],v[0]=start,u0,v0
for i in range(n) :
  t[i+1]=t[0] + h*(i+1)
  u[i+1]=u[i] + h*proies(u[i],v[i],a,b)
  v[i+1]=v[i] + h*predateurs(u[i],v[i],a,b)
```

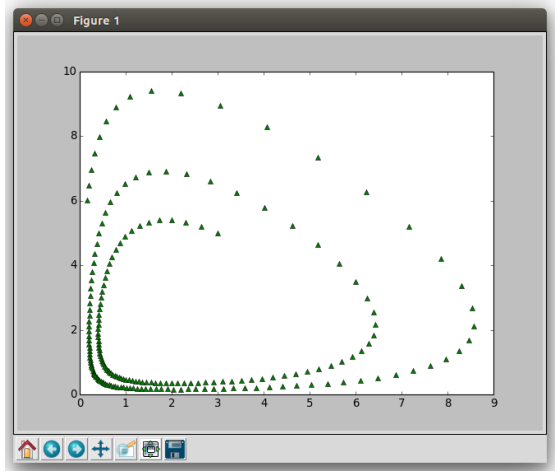
# Proies-Prédateur : Implémentation

## Programme d'application

```
plt.plot(t,u,"+b")
plt.plot(t,v,"*r")
plt.plot(u,v,"^g")
plt.show()
```



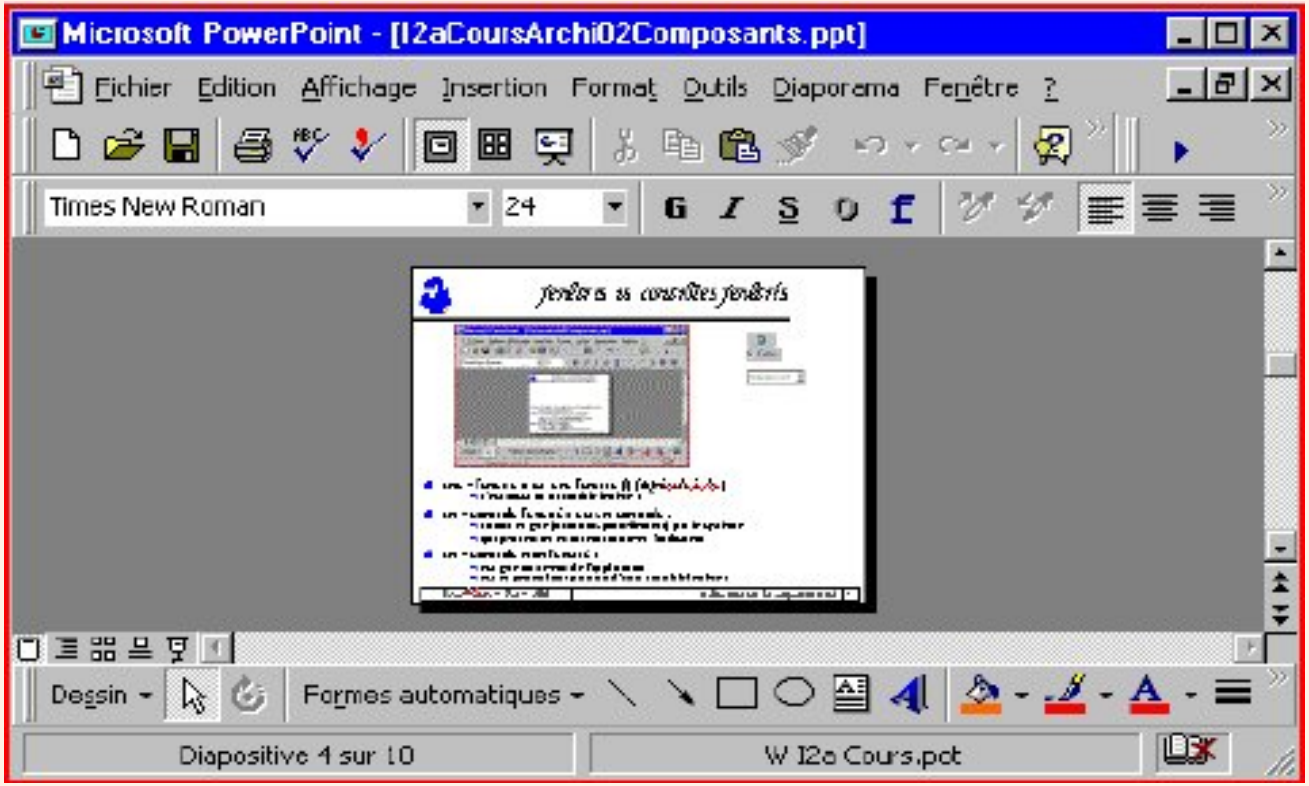




A faire : Lotka-Volterra en MVC (python, TkInter uniquement)

# Composants d'IHM

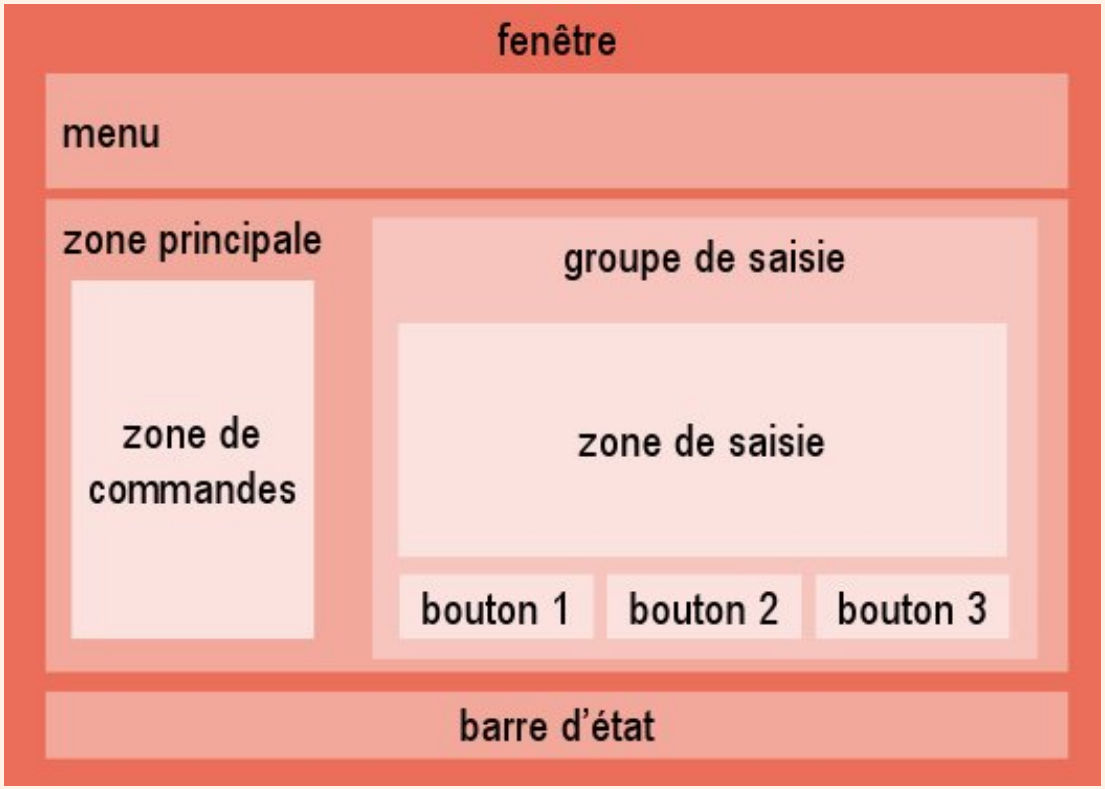
## Exemple "classique"




(cf Didier Vaudène : "un abc des IHM")

# Composants d'IHM

## Organisation hiérarchique "classique"




(cf Stéphanie Jean-Daubias : "Programmation événementielle")

# Composants d'IHM

## Types de fenêtres

- définition des fenêtres de l'application
  - primaire, secondaires
  - boîte de dialogues, de messages
- organisation de leur contenu
- logique d'enchaînement des fenêtres

## Composants de fenêtre

- barre d'actions (menu)
- région client, menus surgissants
- barre d'outils
- barre d'états

# Fenêtre principale

## MainWindow : Objet d'application

```
class MainWindow(Tk):
  def __init__(self, width=100,height=100,bg="red"):
    Tk.__init__(self)
    self.title("Editeur Graphique")
    self.canvas =Canvas(self,width=width-20,
                        height=height-20, bg=bg)
    self.libelle =Label(text ="Serious Game",
                        font="Helvetica 14 bold")
    self.canvas.pack()
    self.libelle.pack()
if __name__ =="__main__":
    MainWindow().mainloop()
```

# Fenêtre principale

## MainWindow : Composant d'application

```
class MainWindow(Frame):
  def __init__(self,parent=None,width=200, ..."):
    Frame.__init__(self)
    self.parent=parent
    ...
   def packing(self) :
    self.canvas.pack()
    self.libelle.pack()
    self.pack()
```

# Fenêtre principale

## Application de test

```
if __name__ =="__main__":
    root = Tk()
    root.title("Editeur Graphique")
    mw = MainWindow(root)
    mw.packing()
    root.mainloop()
```

# Barre de menu

## MenuBar : Menu File

```
class MenuBar(Frame):
  def __init__(self,parent=None):
    Frame.__init__(self, borderwidth=2)
    button_file = Menubutton(self, text="File")
    button_file.pack(side="left")
    menu_file = Menu(button_file)
    menu_file.add_command(label='Save', underline=0,
                          command=parent.save)
    menu_file.add_command(label='Quit', underline=0,
                          command=parent.destroy)
    button_file.configure(menu=menu_file)
```

# Barre de menu

## MenuBar : Menu Edit

```
button_file = Menubutton(self, text="Edit")
button_file.pack(side="left")
menu_file = Menu(button_file)
menu_file.add_command(label='Hide', underline=0,
                      command=parent.delete_circle)
menu_file.add_command(label='Show', underline=0,
                      command=parent.create_circle)
button_file.configure(menu=menu_file)
```

# Barre de menu

## MainWindow : avec MenuBar

```
class MainWindow(Frame):
  def __init__(self,parent=None):
    Frame.__init__(self)
    menubar = MenuBar(self)
    self.canvas = Canvas(self, ...)
    ...
  # actions
  def destroy(self):
    exit()
```

# Barre de menu

## MenuBar : Sauvegarde de fichiers

```
import tkFileDialog
# python3 : from tkinter import filedialog
class MainWindow(Frame):
  ...
  def save(self):
    formats=[('Texte','*.py'),
             ('Portable Network Graphics','*.png')]
    filename=
     tkFileDialog.asksaveasfilename(parent=self.parent,
                                    filetypes=formats,
                                    title="Save...")
    if len(nfilename) > 0:
      print("Sauvegarde en cours dans %s" % filename)
```

# Zone Client

## ScrolledCanvas : zone de travail défilante

```
class ScrolledCanvas(Frame):
  def __init__(self,parent,
               width=100,height=100,bg="white",bd=2,
               scrollregion =(0,0,300,300)):
    Frame.__init__(self,parent)
    self.canvas=Canvas(self,width=width-20,
                       height=height-20,bg=bg,bd=bd,
                       scrollregion=scrollregion)
    self.canvas.grid(row=0,column=0)
```

# Zone Client

## ScrolledCanvas : zone de travail défilante

```
scv=Scrollbar(self,orient="vertical",
              command =self.canvas.yview)
sch=Scrollbar(self,orient="horizontal",
              command=self.canvas.xview)
self.canvas.configure(xscrollcommand=sch.set,
                      yscrollcommand=scv.set)
scv.grid(row=0,column=1,sticky="ns")
sch.grid(row=1,column=0,sticky="ew")
self.bind("<Configure>", self.resize)
self.started =False
```

# Zone Client

## ScrolledCanvas : zone de travail défilante

```
def resize(self,event):
    if self.started:
        w=self.winfo_width()-20,
        h=self.winfo_height()-20
        self.canvas.configure(width=w,height=h)
    else :
        self.started=True
def get_canvas(self) :
    return self.canvas
```

# Zone Client

## MainWindow : avec ScrolledCanvas

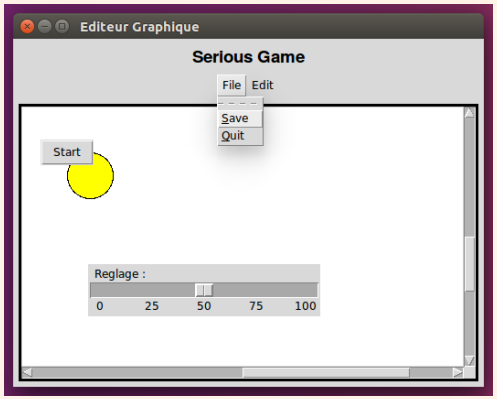
```
class MainWindow(Frame):
  def __init__(self, parent=None):
    menubar = MenuBar(self)
    self.area=ScrolledCanvas(self,
                             width=500,height=300,
                             scrollregion=(-600,-600,600,
    self.libelle =Label(text="Serious Game",
                        font="Helvetica 14 bold")
    ...
```

# Animation de spot (cf : annexe)

## Contrôle et animation et d'un spot

```
if __name__ =="__main__":
    root = Tk()
    root.title("Editeur Graphique")
    mw = MainWindow(root)
    mw.packing()
    root.mainloop()
```

## Visualisation



# 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 necessaires 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

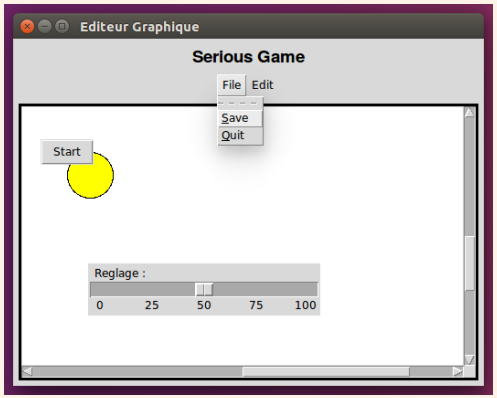
```
{logname@hostname} python classes.py
<Point('10','10')>
<Rectangle('<Point('10','10')>','100','200')>
<Carre('<Point('10','10')>','100')>
```

# Annexes : Animation de spot

## Programme d'application

```
if __name__ =="__main__":
    root = Tk()
    root.title("Editeur Graphique")
    mw = MainWindow(root)
    mw.packing()
    root.mainloop()
```

## Lancement de l'application



# Annexes : Animation de spot

## MainWindow : initialisation

```
class MainWindow(Frame):
  def __init__(self,parent=None):
    Frame.__init__(self,parent)
    self.parent=parent
    menubar = MenuBar(self)
    self.area=self.create_area()
    self.button_start,self.scale_circle=
                                  self.create_controls()
    self.libelle=Label(text="Serious Game",
                       font="Helvetica 14 bold",
                       bg="white",fg="red")
```

# Annexes : Animation de spot

## MainWindow : initialisation

```
    self.x,self.y=100,100
    self.circle_bb=100
    self.circle=self.create_circle()
    self.animation_id=None
```

## MainWindow : zone cliente

```
def create_area(self) :
    canvas=ScrolledCanvas(self,
                          width=500, height=300,
                          scrollregion=(-600,-600,
                                        600,600))
    return canvas
```

# Annexes : Animation de spot

## MainWindow : contrôleurs

```
def create_controls(self) :
    canvas=self.area.get_canvas()
    start = Button(self,
                   text="Start",command=self.start)
    scale = Scale(self,
                  length=250, orient="horizontal",
                  label='Rayon :',
                  troughcolor='dark grey',
                  sliderlength=20,
                  showvalue=0,
                  from_=0, to=100,
                  tickinterval=25,
                  command=self.update_circle)
```

# Annexes : Animation de spot

## MainWindow : contrôleurs

```
    scale.set(50)
    canvas.create_window(50,200,window=start)
    canvas.create_window(250,200,window=scale)
    return start,scale
```

## MainWindow : création du spot

```
def create_circle(self):
    canvas=self.area.get_canvas()
    circle=canvas.create_oval(self.x,self.y,
                              self.x+self.circle_bb,
                              self.y+self.circle_bb,
                              fill='yellow',
                              outline='black')
    return circle
```

# Annexes : Animation de spot

## MainWindow : modification du spot

```
def delete_circle(self):
    canvas=self.area.get_canvas()
    canvas.delete(self.circle)

def update_circle(self, size):
    canvas=self.area.get_canvas()
    canvas.delete(self.circle)
    self.circle_bb=2*int(size)
    self.circle=self.create_circle()
```

# Annexes : Animation de spot

## MainWindow : contrôle de l'animation

```
def stop(self):
    self.after_cancel(self.animation_id)
    self.button_start.configure(text="Start",
                                command=self.start)

def start(self):
    self.button_start.configure(text="Stop",
                                command=self.stop)
    self.animation()
```

# Annexes : Animation de spot

## MainWindow : animation du spot

```
def animation(self):
    self.x += randrange(-60, 61)
    self.y += randrange(-60, 61)
    canvas=self.area.get_canvas()
    canvas.coords(self.circle,
                  self.x,
                  self.y,
                  self.x+self.circle_bb,
                  self.y+self.circle_bb)
    self.libelle.config(text="Cherchez en %s %s" \
                              % (self.x, self.y))
    self.animation_id=self.after(250, self.animation)
```

# Annexes : Observer de boîte de vitesse

## Subject : boîte de vitesse

```
class Car(Subject):
    def __init__(self):
        Subject.__init__(self)
        self.speed=0
        self.rpm=1000
        self.gearbox_speed=0
        self.gearbox=[0,0.006,0.009,0.012,0.014]
```

# Annexes : Observer de boîte de vitesse

## Subject : accélérer

```
    def accelerate(self,value):
        self.rpm=1000*value
        self.speed=
            self.rpm*self.gearbox[self.gearbox_speed]
        self.notify()
```

# Annexes : Observer de boîte de vitesse

## Subject : rétrograder

```
def downshift(self,value):
    self.gearbox_speed=value
    self.speed=
        self.rpm*self.gearbox[self.gearbox_speed]
    self.rpm=+1000
    self.notify()
```

# Annexes : Observer de boîte de vitesse

## Subject : changer de vitesse

```
    def upshift(self):
        self.gearbox_speed=value
        self.speed=
            self.rpm*self.gearbox[self.gearbox_speed]
        if self.rpm > 1000 :
            self.rpm-=1000
        self.notify()
```

# Annexes : Observer de boîte de vitesse

## Observer : vitesse et compte-tours

```
class Observer:
    def update(self, subject):
        raise NotImplementedError
class Speed(Observer):
    def update(self, subject):
        print("speed", subject.speed)
class Rpm(Observer):
    def update(self, subject):
        print("rpm", subject.rpm)
```

# Annexes : Observer de boîte de vitesse

## Application de test

```
if __name__ == "__main__" :
    car=Car()
    car.attach(Speed())
    car.attach(Rpm())
    car.accelerate(1)
    for i in range(1,5) :
        car.upshift(i)
    car.downshift(i-1)
```

# Annexes : MVC, boîte de vitesse

## MVC

- Modèle : la classe `Car`
- Vue : affichage des valeurs (`speed,rpm`) de la voiture (`Car`)
- Contrôleur : levier de vitesse et pédale d'accélération (`Gearbox`) de la voiture

# Annexes : MVC, boîte de vitesse

## Vue : afficher l'état du modèle

```
class Dashboard(Observer) :
    def __init__(self,parent):
        self.parent=parent
        self.speed=Label(parent)
        self.rpm=Label(parent)
        self.speed.pack()
        self.rpm.pack()
    def update(self, subject):
        self.speed.configure(text=str(subject.speed))
        self.rpm.configure(text=str(subject.rpm))
```

# Annexes : MVC, boîte de vitesse

## Contrôleur : modifier l'état du modèle

```
class Gearbox :
    def __init__(self,parent,model):
        self.model=model
        self.speed_data=IntVar()
        self.accelerator=
              Scale(parent,
                    variable=self.speed_data,
                    label="Acceleration",
                    orient="horizontal",
                    length=200, from_=0,to=10,
                    showvalue=0,tickinterval=1,
                    command=self.update_speed)
```

# Annexes : MVC, boîte de vitesse

## Contrôleur : modifier l'état du modèle

```
        self.gear_data=IntVar()
        self.gear=Scale(parent,
                        variable=self.gear_data,
                        label="Gear Speed",
                        orient="horizontal",
                        length=200,from_=0,to=4,
                        showvalue=0,tickinterval=1,
                        command=self.update_gear)
        self.accelerator.pack()
        self.gear.pack()
```

# Annexes : MVC, boîte de vitesse

## Contrôleur : modifier l'état du modèle

```
def update_speed(self,event):
    self.model.accelerate(self.speed_data.get())
def update_gear(self,event):
    if self.model.get_gearbox_speed() \
                < self.gear_data.get() :
        self.model.upshift(self.gear_data.get())
    elif self.model.get_gearbox_speed() \
                > self.gear_data.get() :
        self.model.downshift(self.gear_data.get())
```

# Annexes : MVC, boîte de vitesse

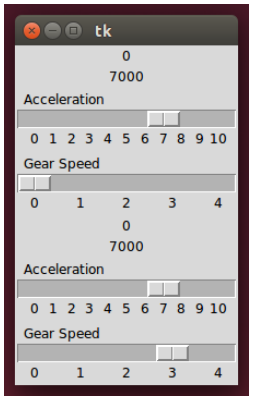
## Application de test

```
if __name__ == "__main__" :
    root=Tk()
    model=Car()
    view=Dashboard(root)
    model.attach(view)
    control=Gearbox(root,model)
    root.mainloop()
```

# Annexes : MVC, boîte de vitesse

## Application de test

```
if __name__ == "__main__" :
    root=Tk()
    ...
    view=Dashboard(root)
    model.attach(view)
    control=Gearbox(root,model)
    root.mainloop()
```



# Annexes : MVC, boîte de vitesse

## Application de test

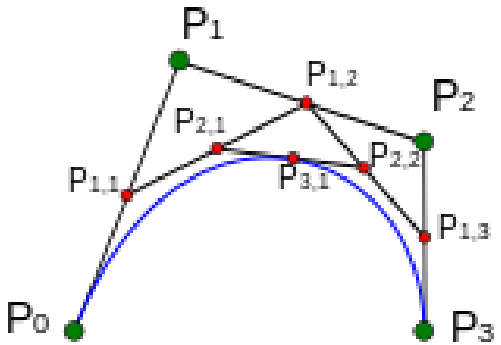
```
...
top=Toplevel()
view=Dashboard(top)
model=Car()
model.attach(view)
control=Gearbox(top,model)
root.mainloop()
```


# Annexes : MVC, courbes de Bézier

## MVC

- Modèle : points de contrôle, calcul de courbe
- Vue : visualiser points de contrôle et courbe
- Contrôleur : déplacer les points de contrôle

from : `http://www.f-legrand.fr/scidoc/docmml`




$$P(t) = P_{3,1}(t) = (1-t) * P_{2,1} + t * P_{2,2}$$

# Annexes : MVC, courbes de Bézier

## Courbe de Bézier d'ordre 3

$$P_{1,i}(t) = (1-t) * P_i + t * P_{i+1}$$
$$P_{2,i}(t) = (1-t) * P_{1,i} + t * P_{2,i+1}$$
$$P_{3,1}(t) = (1-t) * P_{2,1} + t * P_{2,2}$$

## Courbe de Bézier d'ordre $n$

$n+1$ points de contrôles : $\{P_0, ..., P_i, ..., P_n\}$
Relation de récurrence pour le calcul des points de la courbe :

$$P_{j,i}(t) = (1-t) * P_{j-1,i} + t * P_{j-1,i+1}$$

# Annexes : MVC, courbes de Bézier

## Le modèle : Bezier

```
class Bezier(Subject):
    def __init__(self,points=[]):
        Subject.__init__(self)
        self.ctrl_pts=copy(points)
        print(self.ctrl_pts)
        self.curve=[]
```

# Annexes : MVC, courbes de Bézier

## Le modèle : notifications

```
    def set_ctrl_pts(self,points) :
        self.ctrl_pts=copy(points)
        self.notify()
    def get_ctrl_pts(self) :
        return self.ctrl_pts
  def set_curve(self,curve) :
        self.curve=copy(curve)
        self.notify()
    def get_curve(self) :
        return self.curve
```

# Annexes : MVC, courbes de Bézier

ref : incolumitas.com/2013/10/06/plotting-bezier-curves

## Le modèle : calcul de la courbe

```
    def compute_point(self,points,t):
        if len(points)==1:
            return points[0]
        else:
            casteljau_points=[]
            for i in range(0,len(points)-1):
                x=(1-t)*points[i][0]+t*points[i+1][0]
                y=(1-t)*points[i][1]+t*points[i+1][1]
                casteljau_points.append((x, y))
            return self.compute_point(casteljau_points,t)
```

Relation de récurrence : $P_{j,i}(t) = (1-t) * P_{j-1,i} + t * P_{j-1,i+1}$

# Annexes : MVC, courbes de Bézier

## Le modèle : calcul de la courbe

```
def compute_curve(self,step=0.01) :
    t=0
    del self.curve[:]
    while (t<=1):
        self.curve.append(
          self.compute_point(self.ctrl_pts,t)
                           )
        t+=step
    self.notify()
    return self.curve
```

# Annexes : MVC, courbes de Bézier

## La vue : Screen

```
class Screen(Observer) :
    def __init__(self,parent,model):
        self.parent=parent
        self.model=model
        self.canvas=Canvas(parent)
        self.curve_id=-1
        self.ctrl_pts_id=[]
        self.ctrl_pt_index=-1
    def set_canvas(self,canvas) :
        self.canvas=canvas
    def get_canvas(self) :
        return self.canvas
```

# Annexes : MVC, courbes de Bézier

## La vue : mise à jour

```
def update(self, subject):
    curve=subject.get_curve()
    self.canvas.delete(self.curve_id)
    self.curve_id=
          self.canvas.create_line(curve,
                                  width=3,
                                  fill='gray40')
```

# Annexes : MVC, courbes de Bézier

## La vue : mise à jour

```
    def update_ctrl_pts(self, model):
        ctrl_pts=model.get_ctrl_pts()
        del self.ctrl_pts_id[:]
        i=0
        while i < len(ctrl_pts):
            x,y=ctrl_pts[i]
            self.ctrl_pts_id.append(
              self.canvas.create_oval(x,y,x+10,y+10,
                                      outline='black',
                                      fill='green'
                                    )
                                  )
            i=i+1
```

# Annexes : MVC, courbes de Bézier

## Vue/Contrôleur : Sélection de point de contrôle

```
    def select_point(self,event,model) :
        ctrl_pts=model.get_ctrl_pts()
        i=0
        while i<len(ctrl_pts) :
            x,y=ctrl_pts[i]
            if x-10<event.x<x+10 and y-10<event.y<y+10 :
                self.ctrl_pt_index=i
                self.canvas.itemconfigure(
                      self.ctrl_pts_id[\
                            self.ctrl_pt_index],
                      fill='red'
                                        )
                    break
                i=i+1
```

# Annexes : MVC, courbes de Bézier

## Vue/Contrôleur : Création de point de contrôle

```
        if self.ctrl_pt_index==-1 :
            i=0
            while i < len(ctrl_pts)-1 :
                x1,y1=ctrl_pts[i]
                x2,y2=ctrl_pts[i+1]
                if (x1<event.x<x2 or x2<event.x<x1) and \
                   (y1<event.y<y2 or y2<event.y<y1) :
                    ctrl_pts.insert(i,(event.x,event.y))
                    model.set_ctrl_pts(ctrl_pts)
                    self.update_ctrl_pts(model)
                    break
                i=i+1
```

# Annexes : MVC, courbes de Bézier

## Vue/Contrôleur : déplacement de point de contrôle

```
    def move_point(self,event,model) :
        if 0<=self.ctrl_pt_index<len(self.ctrl_pts_id):
            coords=self.canvas.coords(
                    self.ctrl_pts_id[self.ctrl_pt_index]
                                      )
            x1,y1=coords[0],coords[1]
            x1,y1=event.x-x1,event.y-y1
            ctrl_pts=model.get_ctrl_pts()
            ctrl_pts[self.ctrl_pt_index]=event.x,event.y
            model.set_ctrl_pts(ctrl_pts)
            self.canvas.move(
              self.ctrl_pts_id[self.ctrl_pt_index],x1,y1
                             )
            model.compute_curve()
```

# Annexes : MVC, courbes de Bézier

## Vue/Contrôleur : déselection de point de contrôle

```
def release_point(self,event) :
        self.canvas.itemconfigure(
            self.ctrl_pts_id[self.ctrl_pt_index],
            fill='green'
                                  )
        self.ctrl_pt_index=-1

def packing(self) :
    self.canvas.pack(fill='both', expand=True)
```

# Annexes : MVC, courbes de Bézier

## Programme de test

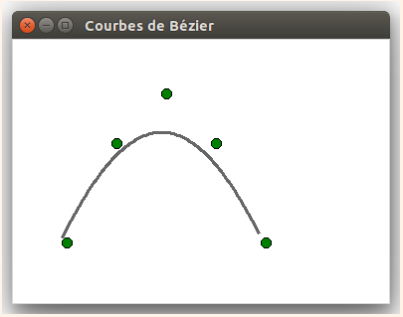
```
if __name__ == "__main__" :
    root=Tk()
    control_points=[(50,200),(100,100),
                    (150,50),(200,100),(250,200)]
    bezier=Bezier(control_points)
    bezier.compute_curve()
    screen=Screen(root,bezier)
    bezier.attach(screen)
    bezier.notify()
    screen.update_control_points(bezier)
    screen.packing()
    canvas=screen.get_canvas()
```

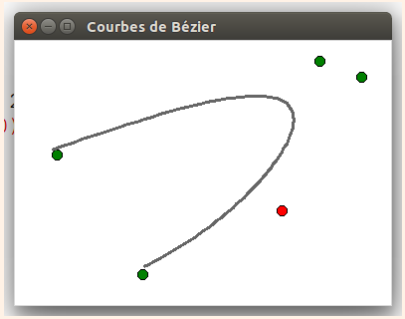
# Courbes de Bézier

## Programme d'application

```
canvas.bind("<Button-1>",
            lambda event,model=bezier :
               screen.select_point(event,model))
canvas.bind("<Motion>",
            lambda event,model=bezier :
               screen.move_point(event,model))
canvas.bind("<ButtonRelease>",screen.release_point)
```

## Visualisation



# 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 Develoment Blueprints” (2015)

# Bibliographie

## Adresses “au Net”

- inforef.be/swi/python.htm
- python.developpez.com
- wiki.python.org/moin/TkInter
- www.jchr.be/python/tkinter.htm
- effbot.org/tkinterbook
- www.pythonware.com/library/tkinter/introduction