Conception d'Applications Interactives

Développement d'IHM (python/TkInter)
Patrons de Conception (Observer,MVC)

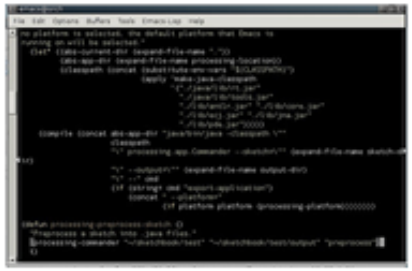
Alexis NEDELEC

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

Interfaces Homme-Machine

Interagir avec un ordinateur

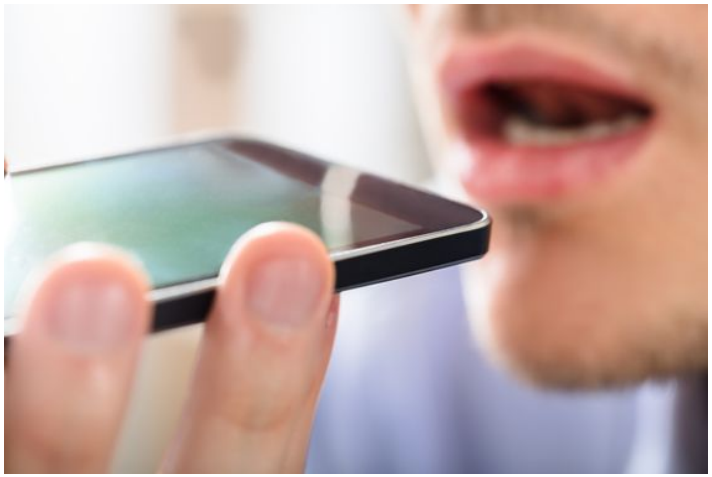
- CLI (Command Line Interface) : interaction clavier
- GUI (Graphical User Interface) : interaction souris-clavier
- NUI (Natural User Interface) : interaction tactile, capteurs

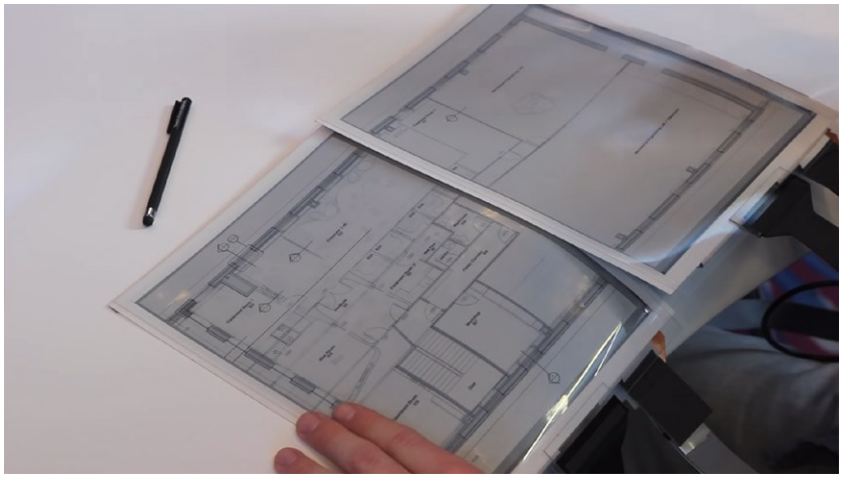


Interfaces Homme-Machine

Interagir avec un ordinateur

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





Interfaces Homme-Machine

Objectifs du cours

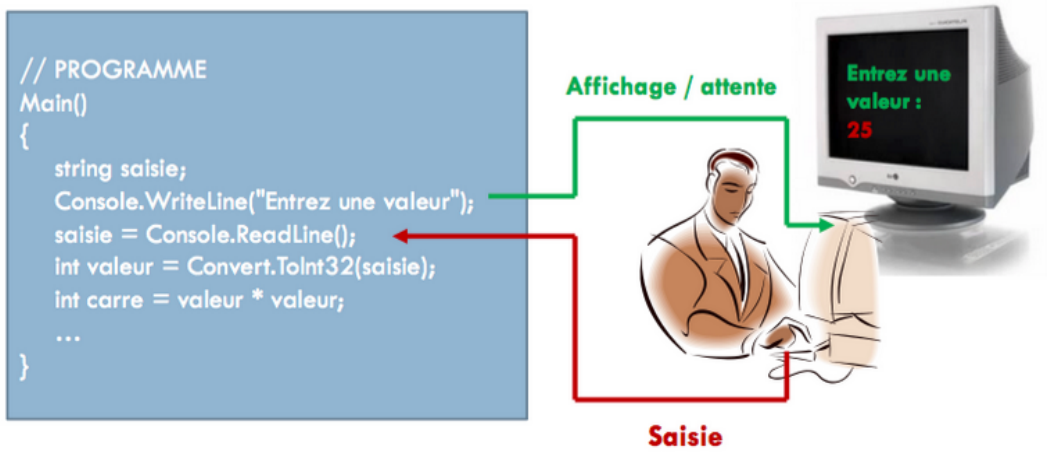
Développement d'IHM basé sur les patrons de conception

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

1. initialisation
   - modules externes, ouverture fichiers, connexion serveurs ...
2. traitements de données
   - affichage, modification, appel de fonctions ...
3. terminaison : sortir “proprement” de l’application



Programmation événementielle

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

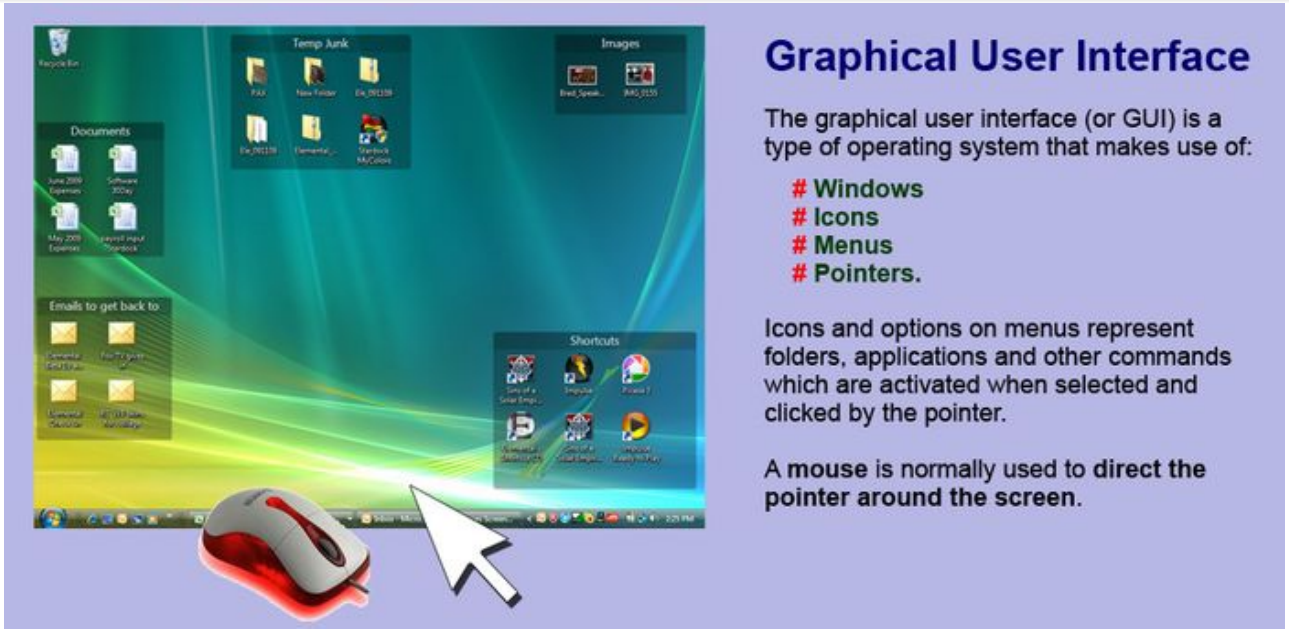
1. initialisation
   - modules externes, ouverture fichiers, connexion serveurs ...
   - création de composants graphiques
2. traitements de données par des fonctions réflexes (actions)
   - affichage de composants graphiques
   - liaison composant-événement-action
   - attente d'action utilisateur, dans une boucle d'événements
3. terminaison : sortir "proprement" de l'application

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

Bibliothèques

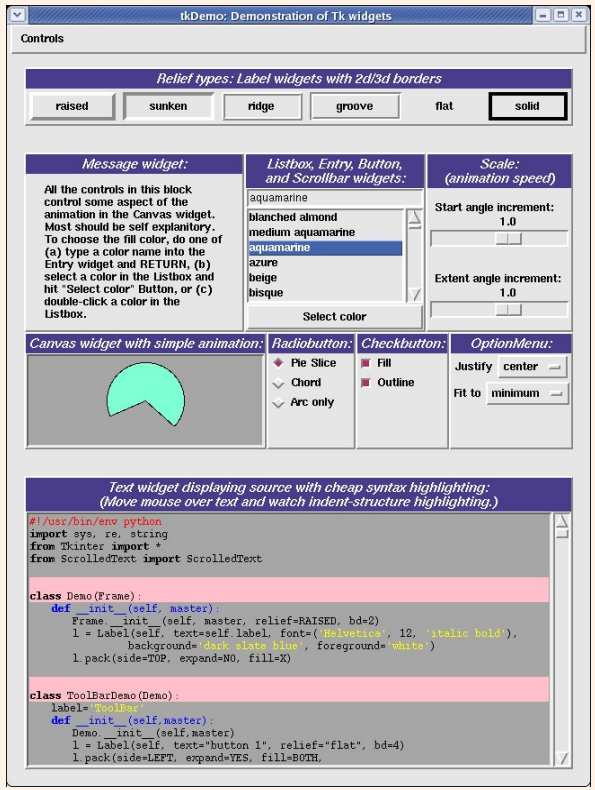
Langages, API, Toolkits pour développer des IHM

- 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 ...
- ...



Python/TkInter

TkInter : Tk (de Tcl/Tk) pour python



Documentation TkInter (sur `docs.python.org`)

Composants graphiques

Widgets : **Wi**ndow 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, entre autres, sur TkInter

Hello World

Création d'IHM

```
import tkinter as tk
root=tk.Tk()
label_hello=tk.Label(root,
                     text="Hello World !",fg="blue")
button_quit=tk.Button(root,
                     text="Goodbye World", fg="red",
                     command=root.destroy)
label_hello.pack()
button_quit.pack()
root.mainloop()
```

Hello World

Création de composants graphiques

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

Interaction sur un composant

- `quit=Button(root,...,command=root.destroy)`

interaction par défaut : "click" bouton gauche ("Button-1")

Positionnement des composants

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

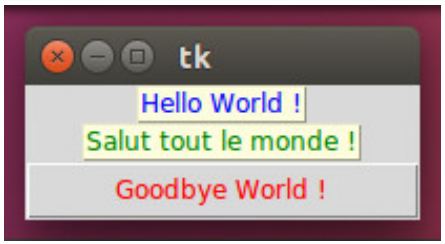
Entrée dans la boucle d'événements

- `root.mainloop()`

Hello World

Fichier de configuration d'options

```
import tkinter as tk
root=tk.Tk()
root.option_readfile("hello.opt")  # widgets options
label_hello=tk.Label(root,text="Hello World !")
label_bonjour=tk.Label(root,name="labelBonjour")
button_quit=tk.Button(root,text="Goodbye World !")
label_hello.pack()
label_bonjour.pack()
button_quit.pack()
root.mainloop()
```



Hello World

Fichier de configuration 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
```

Accès aux widget pour fixer une valeur d'option

- chemin_acces_widget.option : valeur
- nom de classe : toutes les instances auront la valeur d'option
- nom d'instance : seule l'instance aura la valeur d'option

Interaction Utilisateur

Interaction par défaut

- "Click gauche" ("Button-1") pour lancer l'action

```
quit=Button(root,...,command=root.destroy)
```

Paramétrer l'interaction : gestion d'événements

Liaison Composant-Événement-Action :

- créer le composant d'IHM

```
quit=Button(root,...)
```

- implémenter une fonction réflexe (l'action)

```
def callback(event) :
    root.destroy()
```

- lier (bind) l'action au composant via un événement

```
quit.bind("<Button-1>",callback)
```

Gestion d'événements

Accès aux informations sur ce que fait l'utilisateur

- pointeur de souris (accès aux coordonnées)
- claviers (touches alphanumériques)
- informations via d'autres périphériques (kinect ...)

Exemple : affichage des coordonnées souris

```
hello=tk.Label(parent,text="Hello World !")
hello.bind("<Button-1>", on_hello_action)
def on_hello_action(event) :
  print("(x,y) on widget",event.x,event.y)
  print("(x,y) on screen",event.x_root,event.y_root)
```

Gestion d'événements

Accès aux informations sur les propriétés du composant graphique

- accès au widget lié à l'événement (`event.widget`)
- fixer des valeurs d'options aux widget (`configure()`)
- récupérer des valeurs d'options de widget (`cget()`)

Exemple : affichage des coordonnées souris

```
hello=tk.Label(parent,text="Hello World !")
hello.bind("<Button-1>", on_hello_action)
def on_hello_action(event) :
  print("widget text",event.widget.cget("text"))
  event.widget.configure(text=
        "X="+str(event.x)+",Y="+str(event.y))
```

Gestion d'événements

Accès aux informations liées à l'application

- transmettre des arguments aux fonctions réflexes (`callback(event,arg1,arg2,...)`)

Exemple : affichage des coordonnées souris

```
canvas=tk.Canvas(parent,width=400,height=200)
canvas.bind("<Motion>",
            lambda event,data=hello : \
                on_canvas_move_action(event,data))
def on_canvas_move_action(self,event,data):
  self.data.configure(text=
       "x="+str(event.x)+",y="+str(event.y))
```

Appel de fonction réflexes avec arguments : fonctions anonymes

```
widget.bind("<EventName>", lambda arg1,arg2=value,... : callback(arg1,arg2,...))
```

Fonction réflexe

Pour quels événements ?

- `Expose` : exposition de fenêtre, composants
- `Enter, Leave` : passage de souris sur un composant
- `Configure` : modification de taille de fenêtre
- `KeyPress` : utilisation du clavier
- `ButtonRelease` : utilisation de la souris
- ...

Enchaînement d'événements (ex : pour tracer des figures) :
`"<Button-1>", "<Motion>", "<ButtonRelease>"`

Représentation générale d'un événement

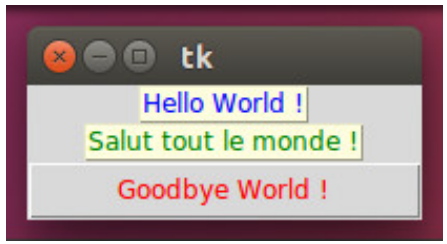
- `"<Modifier-EventType-ButtonNumberOrKeyName>"`

`"<Control-Shift-KeyPress-a>", "<Control-KeyPress-A>"`

Programmation Orientée Objets

classe MainWindow

```
if __name__ =="__main__" :
  root=tk.Tk()
  root.option_readfile("mainwindow.opt")
  mw=MainWindow(root)
  mw.layout()
  root.mainloop()
```



Création de classe

```
class MainWindow :
  def __init__(self,parent) :
    pass
  def create_gui(self) :
    pass
  def actions_binding(self) :
    pass
  def on_<name>_action(self,event) :
    pass
  def layout(self) :
    pass
```

Création de classe

Initialisation

```
def __init__(self,parent) :
  self.parent=parent
  self.create_gui()
  self.actions_binding()
```

Création des composants graphiques

```
def create_gui(self) :
  self.hello=tk.Label(self.parent,
                      text="Hello World !",fg="blue")
  self.bonjour=tk.Label(self.parent,
                        name="labelBonjour")
  self.quit=tk.Button(self.parent,
                      text="Goodbye World",fg="red")
```

Création de classe

Liaison composant-événement-action

```
def actions_binding(self) :
  self.quit.bind("<Button-1>",self.on_quit_action)
def on_quit_action(self,event) :
  self.parent.destroy()
```

Positionnement des composants

```
def layout(self) :
  self.hello.pack()
  self.bonjour.pack()
  self.quit.pack()
```

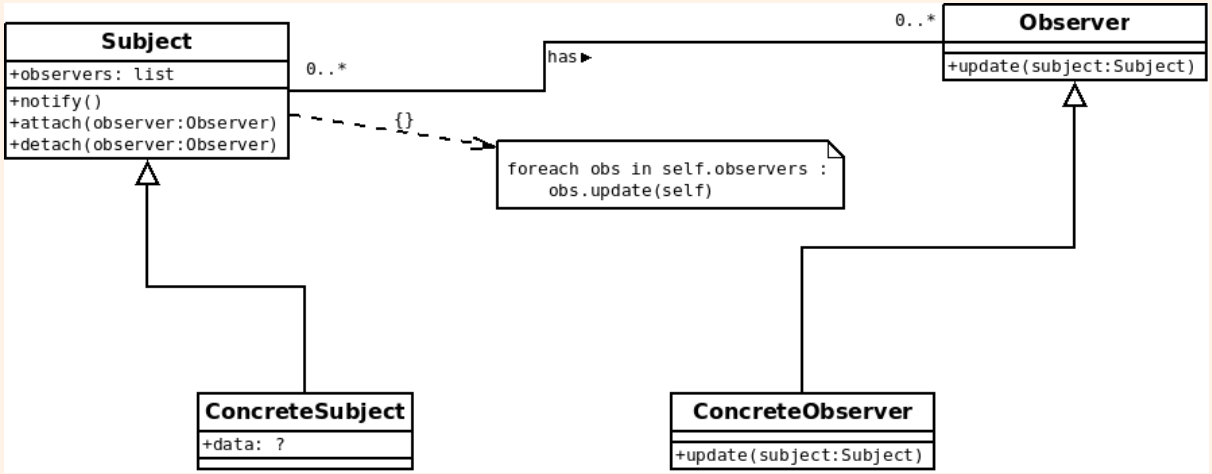
Patrons de conception

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



Rôle d'un `Subject` :

- notifier ses modifications de propriétés (`notify()`)
- aux `observers` qui lui sont associés (`attach()`)

Rôle d'un `Observer` :

- faire des mise à jours (`update(subject)`)
- en cas de notication d'un `Subject` auquel il est attaché

Modèle Observer

Subject : implémentation

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

Modèle Observer

Subject : implémentation

```
    def attach(self,obs):
        if not callable(getattr(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)
```

Un Observer est associé au Subject uniquement s'il implémente une méthode de mise à jour (update())

Modèle Observer

Subject : Héritage

```
class Subject(object):
    def __init__(self):
        self.observers=[]
        ...
class ConcreteSubject(Subject):
    def __init__(self):
      Subject.__init(self)
      self.__data=0
    def set_data(data)
      self.__data=data
      self.notify()
```

Toute méthode de `ConcreteSubject` modifiant ses propriétés doit notifier les modifications à ses `ConcreteObserver`.

Modèle Observer

Observer : mise à jour

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

Exemple

```
class ConcreteObserver(Observer):
    def __init__(self):
        pass
    def update(self,subject):
        print("ConcreteObserver.update() :",
            subject.get_data())
```

Tout `ConcreteObserver` doit avoir une méthode de mise à jour.

Modèle Observer

ConcreteSubject : implémentation

```
class ConcreteSubject(Subject):
  def __init__(self):
    Subject.__init__(self)
    self.__data=0
  def get_data(self):
    return self.__data
  ...
  def increase(self):
    print("ConcreteSubject.increase()")
    self.__data+=1
    self.notify()
  ...
```

Modèle Observer

ConcreteSubject : implémentation

```
class ConcreteObserver(Observer):
  def __init__(self,name):
    self.name=name
  def get_name(self) :
    return self.name
  ...
  def update(self,subject):
    print("Observer :",self.name)
    print("on Subject data :", subject.get_data())
  ...
```

Modèle Observer

Programme de test

```
subject=ConcreteSubject()
name="Observer 1"
obs=ConcreteObserver(name)
subject.attach(obs)
subject.increase()
name="Observer 2"
obs=ConcreteObserver(name)
subject.attach(obs)
subject.increase()
subject.detach(obs)
subject.increase()
```

Quels seront les affichages (`print()`) de ce programme de test ?

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

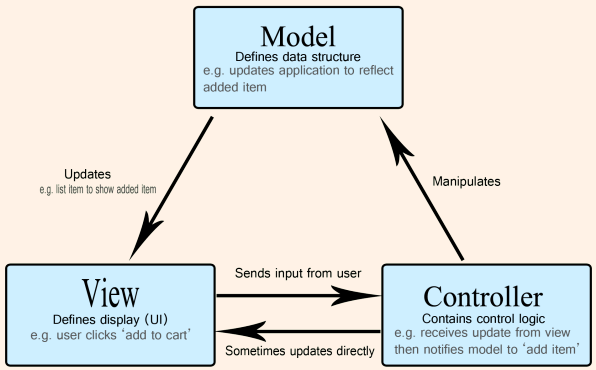
"MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility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

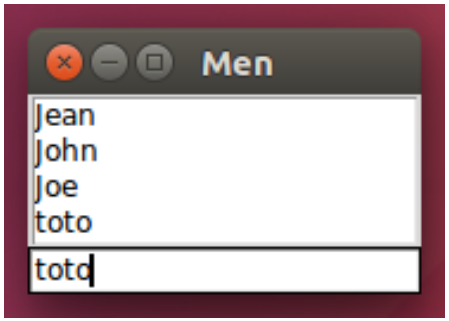


From MDN (Mozilla Developer Network)

MVC

Exemple : 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

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

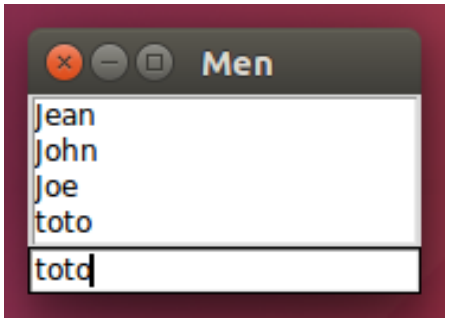
Contrôle du modèle : action utilisateur

```
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

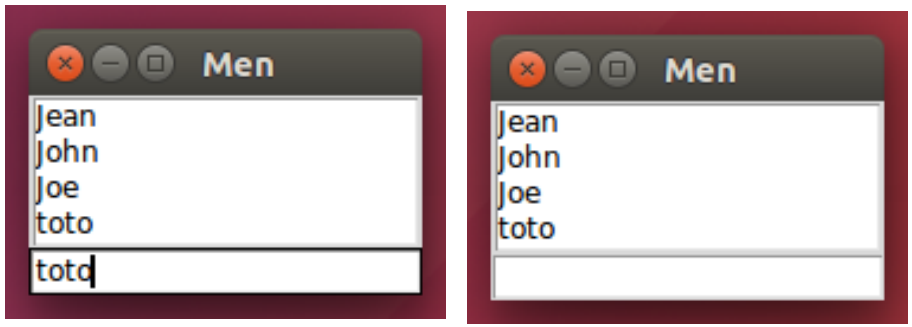
```
if __name__ == "__main__":
    root=tk.Tk()
    root.title("Men")
    names=["Jean", "John", "Joe"]
    model=Model(names)
    view=View(root)
    view.update(model)
    model.attach(view)
    ctrl=Controller(model,view)
```



Test IHM

Un modèle, des vues, des contrôleurs

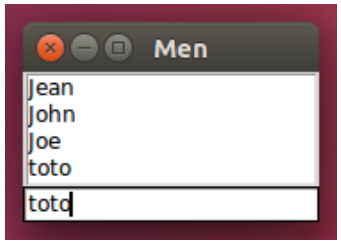
```
...
top=tk.Toplevel()
top.title("Men")
view=View(top)
view.update(model)
model.attach(view)
ctrl=Controller(model,view)
```

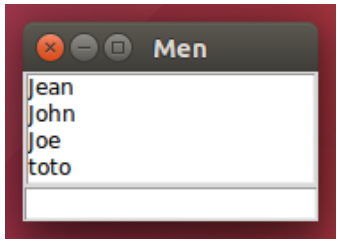


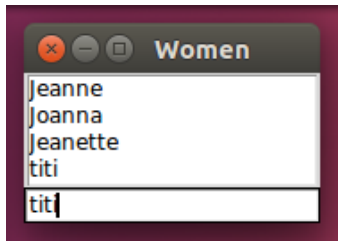
Test IHM

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)
```

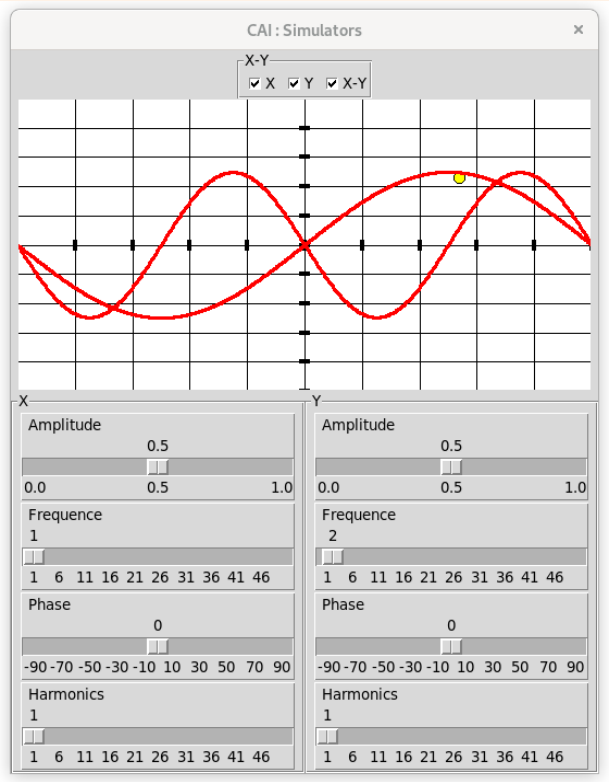


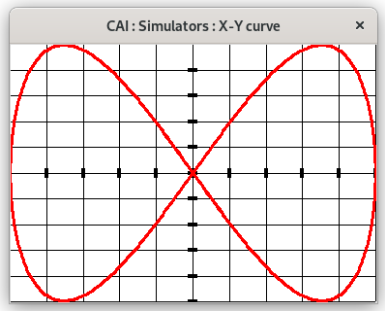




Application : Oscilloscope

Modéliser, Visualiser et Contrôler des signaux





Application : Oscilloscope

Visualiser et contrôler des signaux

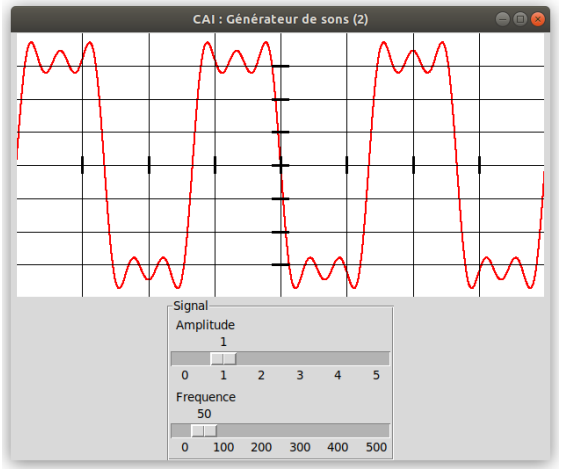
```
model=Generator()                    # générateur de signal
root=tk.Tk()
view=Screen(root)                    # écran d'oscilloscope
view.layout()                        # positionnement écran
model.attach(view)
model.generate()
control=Controls(model,view) # réglages d'oscilloscope
control.layout()                     # positionnement réglages
...
root.mainloop()
```

Générateur de signal

Mouvement vibratoire harmonique

$$e = \sum_{h=1}^{n} \frac{a}{h} \, sin(2\pi(f.h)t + p)$$

- e : élongation à l'instant t
- a, f, p : amplitude, fréquence, phase
- h : nombre d'harmoniques $(h > 0)$



Générateur de signal

Le modèle (Subject) à observer

```
class Generator(Subject) :
  def __init__(self,name="X",mag=1.0,freq=1.0,phase=0) :
    Subject.__init__(self)
    self.name=name
    self.mag,self.freq,self.phase=mag,freq,phase
    self.harmonics=1
    self.signal=[]
    self.samples=100
    ....
```

En plus des caractéristiques du signal, on lui donnera un nom (`self.name`) et on mémorisera dans une liste (`self.signal`) les élongations sur un nombre d'échantillons (`self.samples`)

Générateur de signal

Generator : calcul d'élongation à un instant t

```
def vibration(self,t):
  m,f,p=self.mag,self.freq,self.phase
  harmo=int(self.harmonics)
  sigma=0.0
  for h in range(1,harmo+1) :
    sigma=sigma + (m/h)*sin(2*pi*(f*h)*t+p)
  return sigma
```

$$e = \sum_{h=1}^{n} \frac{a}{h}\ sin(2\pi(f.h)t + p)$$

Générateur de signal

Generator : générer le signal

```
def generate(self,period=1):
  del self.signal[0:]
  samples=range(int(self.samples)+1)
  psamples = period/self.samples
  for t in samples :
    self.signal.append (
        [t*psamples,self.vibration(t*psamples)]
                        )
  self.notify()           # to update observers
  return self.signal
```

- stockage dans une liste (`self.signal.append()`)
- des élongations (`self.vibration()`) d'échantillon (`samples`)
- demande de mise à jour des observateurs (`self.notify()`)

Visualisation de signal

La visualisation (Observer) du modèle

```
class Screen(Observer) :
  def __init__(self,parent,
               bg="white",width=600,height=300):
    Observer.__init__(self)
    self.parent, self.bg=parent,bg
    self.width,self.height=width,height
    self.canvas,self.tiles=None,4
    self.create_gui()
    self.actions_binding()
```

MVC : **V**isualisation (`Screen`) du **M**odèle (`Generator`) :

- sur l'écran (`self.canvas`) de l'`Observer`
- avec un nombre de "carreaux" (`self.tiles`) fixés

Visualisation de signal

L'observation du modèle

```
def create_gui(self) :
  self.canvas=tk.Canvas(self.parent,bg=self.bg,
                        width=self.width,
                        height=self.height)
  self.create_grid()
def actions_binding(self) :
  self.canvas.bind("<Configure>",self.resize)
```

Création des composants, gestion d'événements :

- création de la grille (`create_grid()`)
- sur l'écran (`self.canvas`)
- reconfiguration de l'écran (`resize()`)
- en cas redimensionnement (événement : `"<Configure>"`)

Code de la méthode `create_grid()` consultable en annexe (p.56)

Visualisation de signal

L'observation du modèle

```
def resize(self,event):
    self.width,self.height=event.width,event.height
    self.canvas.delete("grid")
    self.create_grid()
    self.canvas.delete("signal")
    self.plot_signal()
```

En cas de redimensionnement de la fenêtre :

- récupérer les nouvelles dimensions
- effacer la grille existante
- créer une grille avec les nouvelles dimensions
- effacer les données (le signal) affichées
- ré-afficher le signal avec les nouvelles dimensions

Visualisation de signal

L'observation du modèle

```
def update(self,subject):
  self.signal=subject.get_signal()
  if self.signal :
    if self.signal_id :
      self.canvas.delete("signal")
    self.plot_signal()
```

Mise à jour de l'observation selon le modèle `Observer` :

- accéder aux données du modèle (`subject.get_signal()`)
- si elles sont déjà visualisées (`if self.signal_id`),
- les effacer (`self.canvas.delete("signal")`)
- pour visualiser les nouvelles données (`plot_signal()`)

Code de la méthode `plot_signal()` consultable en annexe (p.54)

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)
- mise en œuvre des patrons de conception (Observer,MVC)

Annexes : Oscilloscope

Dimensionnement du signal à l'écran

```
def plot_signal(self):
  w,h=self.width,self.height
  if self.signal and len(self.signal) > 1:
    plots= [
      (x*w,h/2-h*y/self.tiles)
      for (x,y) in self.signal
          ]
```

Création d'une liste (`plots`) contenant les coordonnées `(x,y)` de chaque point du signal (`self.signal`) redimensionnées en fonction de la largeur,hauteur de l'écran `(w,h)` et du nombre de carreaux (`self.tiles`) sur l'écran.

Annexes : Oscilloscope

Dimensionnement du signal à l'écran

```
    self.signal_id=self.screen.create_line(
                                plots,
                                fill=self.signal_color,
                                smooth=1,width=3,
                                tags="signal"
    )
  return self.signal_id
```

On utilise la liste des points redimensionnés (`plots`) pour créer la courbe sur l'écran (`self.screen.create_line`).
On remarquera le tag associé à la ligne (`tags="signal"`).
Le nom associé aux objets "taggés" permet de les manipuler dans une `Canvas`. Ce qui nous servira, en cas de modification, à les effacer, avant de les redessiner

Annexes : Oscilloscope

Création de la grille d'oscilloscope

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

Création de la grille d'oscilloscope

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

Création des lignes horizontales sur l'écran
On remarquera également le tag ("grid") associé aux lignes

Références

Bibliographie

- Gérard Swinnen : “Apprendre à programmer avec Python 3” (2012)
- Guido van Rossum : “Tutoriel Python” `https://bugs.python.org/file47781/Tutorial_EDIT.pdf`
- 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)

Références

Adresses “au Net”

- https://inforef.be/swi/python.htm
- https://docs.python.org/fr/3/library/tk.html
- https://wiki.python.org/moin/TkInter
- https://www.jchr.be/python/tkinter.htm
- https://www.thomaspietrzak.com/teaching/IHM
- https://developer.mozilla.org/en-US/docs/Glossary/MVC

Pour (in)formation

https://www.access-it.fr/formation/formation-python-concevoir-des-interfaces-graphiques