Conception d'Applications Interactives

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

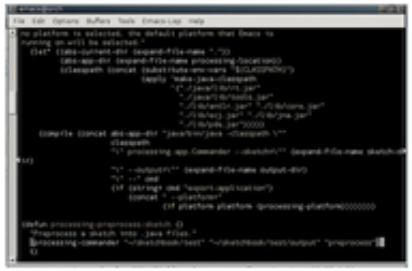
Alexis NEDELEC

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

Interfaces Homme-Machine

Interagir avec un ordinateur

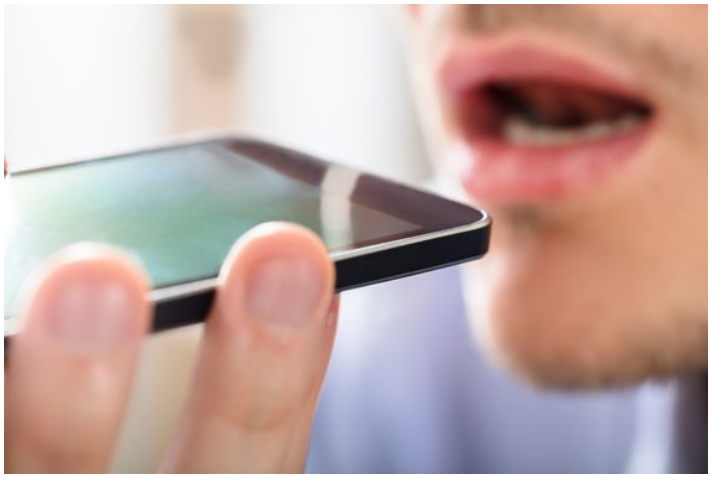
- CLI (Command Line Interface) : interaction clavier
- GUI (Graphical User Interface) : interaction souris-clavier
- NUI (Natural User Interface) : interaction tactile, capteurs

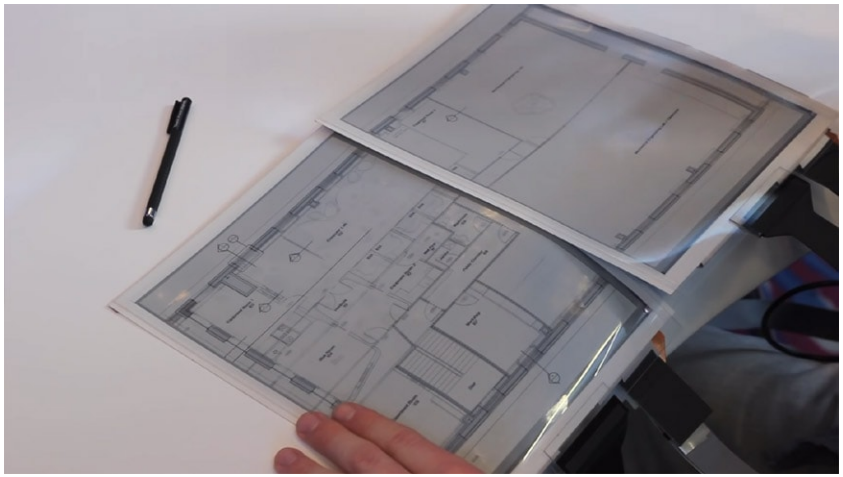


Interfaces Homme-Machine

Interagir avec un ordinateur

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





Interfaces Homme-Machine

Objectifs du cours

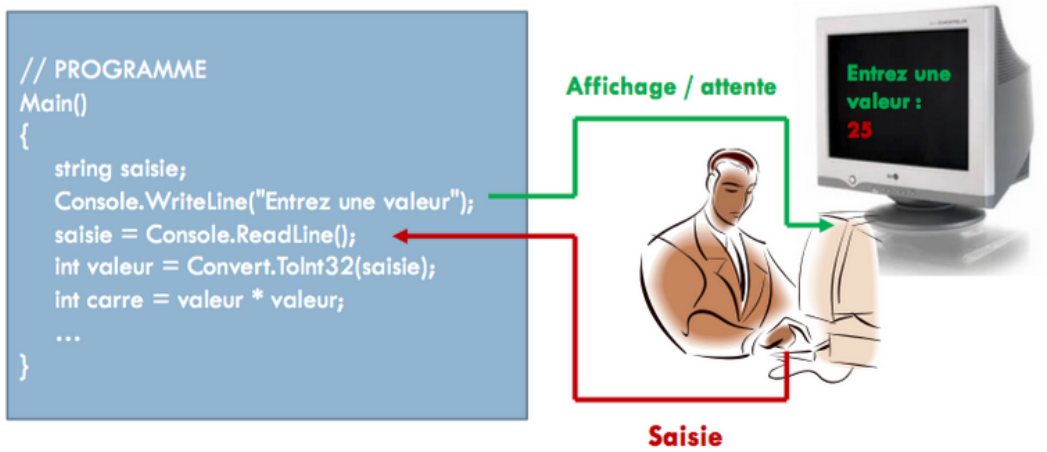
Savoir développer des IHM avec une bibliothèque de composants

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

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

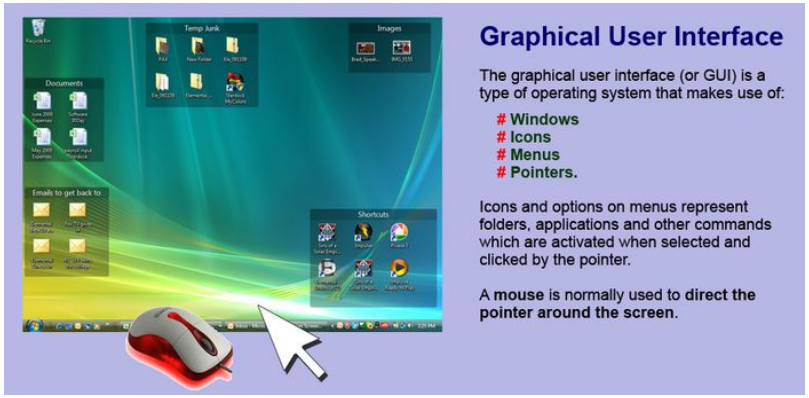
1. initialisation
   - modules externes, ouverture fichiers, connexion serveurs ...
   - création de composants graphiques
2. traitements de données par des fonctions réflexes (actions)
   - affichage de composants graphiques
   - liaison composant-événement-action
   - attente d'action utilisateur, 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

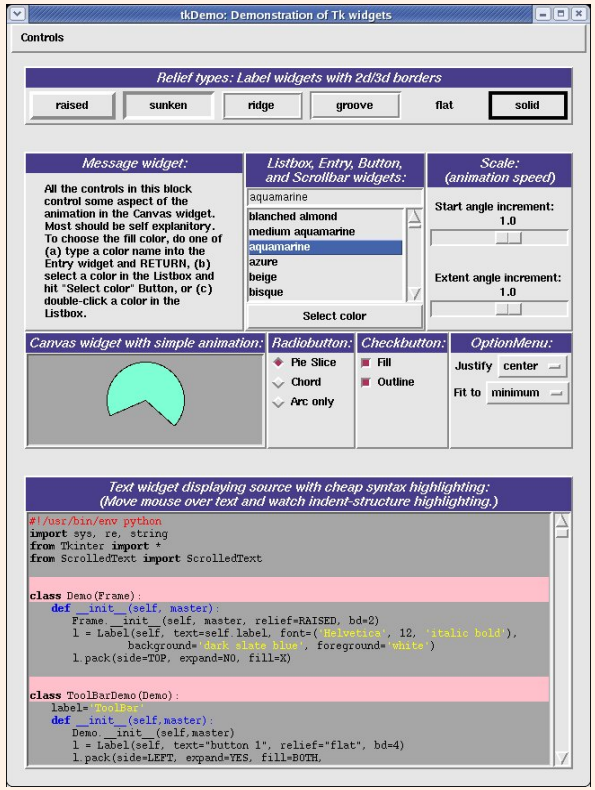
- Java : AWT,SWT,Swing,JavaFX,...,JGoodies, QtJambi ...
- C,C++ : Xlib, GTk, Qt, MFC, ...
- Python : TkInter, wxWidgets, PyQt, Pyside, Kivy,libavg...
- JavaScript : Angular, React, Vue.js, JQWidgets ...
- ...



OS, CLI, GUI, WIMP, Post-WIMP definitions

Python/TkInter

TkInter : Tk (de Tcl/Tk) pour python



Documentation python TkInter

Hello World

Création d'IHM

```
from Tkinter import Tk,Label,Button
root=Tk()
label_hello=Label(root,
                  text="Hello World !",fg="blue")
button_quit=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()`
- `label_hello=Label(root, ...)`
- `button_quit=Button(root, ...)`

Interaction sur un composant

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

Positionnement des composants

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

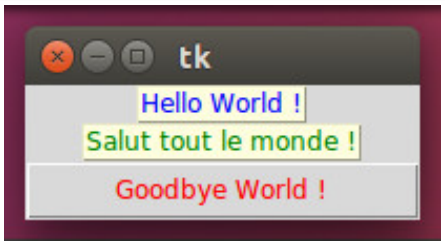
Entrée dans la boucle d'événements

- `root.mainloop()`

Hello World

Fichier de configuration d'options

```
from Tkinter import Tk,Label,Button
root=Tk()
root.option_readfile("hello.opt")  # widgets options
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()
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
```

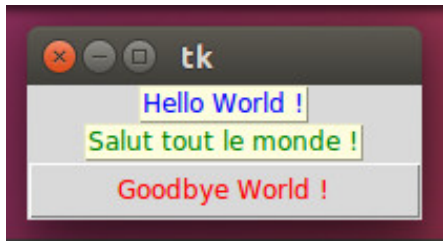
Accès aux widget pour fixer une valeur d'option

- 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

Programmation Orientée Objets

classe MainWindow

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



Programmation Orientée Objets

```
class MainWindow :
  def __init__(self,parent) :
    pass
  def gui(self) :
    pass
  def actions_binding(self) :
    pass
  def action_<name>(self,event) :
    pass
  def layout(self) :
    pass
```

Programmation Orientée Objets

Initialisation

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

Création des composants graphiques

```
def 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")
```

Programmation Orientée Objets

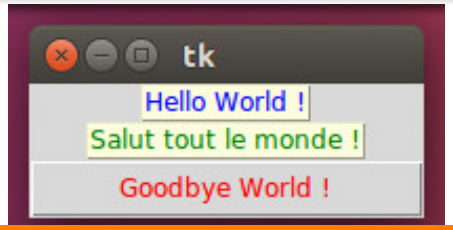
Liaison composant-événement-action

```
def actions_binding(self) :
  self.quit.bind("<Button-1>",self.action_quit)

def action_quit(self,event) :
  self.parent.destroy()
```

Positionnement des composants

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



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, TkInter ... entre autres !

Gestion d'événements

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

- option `command` : "click gauche", exécute la fonction associée

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

Liaison Composant-Événement-Action

- 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

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

Fonction réflexe : implémentation

```
def callback(event) :
  # an action needs information from :
  #  - user: get data from devices (mouse,keyboard ...)
  #  - widget: get or set widget options
  #  - the application: to manage shared data
```

Fonction réflexe : récupération d'informations

- liées à l'utilisateur (argument `event`)
- liées au composant graphique :
  - `event.widget` : on récupère le widget lié à l'événement
  - `configure()` : on peut fixer des valeurs aux options de widget
  - `cget()` : on peut récupérer une valeur d'option de widget
- liées à l'application (paramétrer la fonction réflexe)

Fenêtrage : coordonnées utilisateur

IHM : coordonnées du pointeur de souris

Affichage des coordonnées `(x,y)` sur un composant graphique :

- `Label` "Dans la Canvas" : position souris sur la zone de dessin (`Canvas`)
- `Label` "Hello World !" : position souris l'écran

Coordonnées `(x,y)` affichées :

- `event.x,event.y` : coordonnées relatives au widget
- `event.x_root,event.y_root` : relatives à l'écran

Fenêtrage : coordonnées utilisateur

Création des composants

```
def gui(self) :
  self.hello=tk.Label(self.parent,
                      text="Hello World !",fg="blue")
  self.canvas=tk.Canvas(self.parent,
                        width=200,height=150,
                        bg="light yellow")
  self.frame=tk.LabelFrame(self.parent,
                           text="Coordonnées")
  self.info=tk.Label(self.frame,
                     text="Dans la Canvas",fg="green")
  self.quit=tk.Button(self.parent,
                      text="Goodbye World",fg="red")
```

Fenêtrage : coordonnées utilisateur

Liaison Composant-Événement-Action

```
def actions_binding(self) :
  self.hello.bind("<Button-1>",
                  self.action_hello)
  self.canvas.bind("<Motion>",
                   self.action_canvas_move)
  self.canvas.bind("<Leave>",
                   self.action_canvas_leave)
  self.quit.bind("<Button-1>",
                 self.action_quit)
```

Fenêtrage : coordonnées utilisateur

Implémentation des actions

```
def action_hello(self,event) :
  event.widget.configure(text=
                         "X="+str(event.x_root)+\
                         ",Y="+str(event.y_root))
def action_canvas_move(self,event):
  self.info.configure(text=
                      "x="+str(event.x)+\
                      ",y="+str(event.y))
def action_canvas_leave(self,event):
  self.info.configure(text= "Dans la Canvas")

def action_quit(self,event) :
  self.parent.destroy()
```

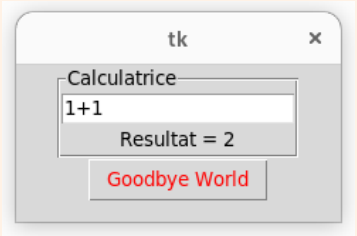
Fenêtrage : coordonnées utilisateur

Gestion de positionnement de composants

```
def layout(self) :
    self.hello.pack()
    self.canvas.pack()
    self.frame.pack()
    self.info.pack()
    self.quit.pack()
```

Traitement de données utilisateur

IHM : évaluer une expression mathématique



Utiliser un composant (`Entry`) pour :

- récupérer l'expression saisie par l'utilisateur (`entry.get()`)
- évaluer (`eval()`) l'expression lors de la validation de l'entrée
- afficher le résultat sur un composant (`Label`)

Traitement de données utilisateur

Création des composants

```
def gui(self) :
  self.frame=tk.LabelFrame(self.parent,
                           text="Calculatrice")
  self.entry=tk.Entry(self.frame)
  self.label=tk.Label(self.frame,
                      text="Resultat")
  self.quit=tk.Button(self.parent,
                      text="Goodbye World", fg="red")
```

Traitement de données utilisateur

Liaison Composant-Événement-Action

```
def actions_binding(self) :
  self.entry.bind("<Return>", self.action_entry)
  self.quit.bind("<Button-1>",self.action_quit)
```

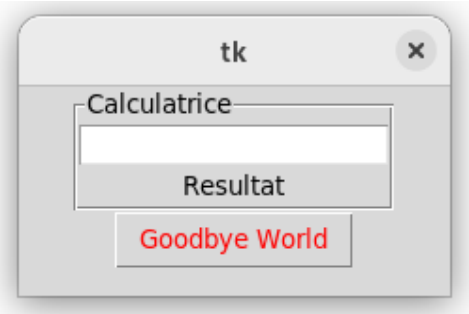
Implémentation des actions

```
def action_entry(self,event):
  self.label.configure(text="Resultat = "+\
                       str(eval(self.entry.get()))
def action_quit(self,event) :
  self.parent.destroy()
```

Traitement de données utilisateur

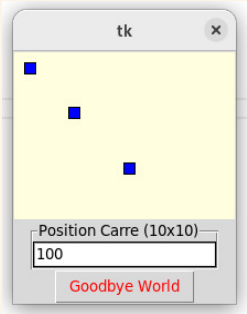
Gestion de positionnement de composants

```
def layout(self) :
    self.frame.pack()
    self.entry.pack()
    self.label.pack()
    self.quit.pack()
```



Communication entre composants

IHM : communication entre composants



Lorsqu'un utilisateur rentre une valeur dans un composant

- `self.entry.bind("<Return>",self.action_entry)`

L'action aura pour simple rôle d'émettre un événement :

- `event.widget.event_generate("<Control-Z>")`

Si un autre composant à l'écoute de cet événement :

- `self.parent.bind("<Control-Z>",self.action_parent)`

Il déclenchera alors l'action qui lui est associé.

Communication entre composants

Création des composants

```
def gui(self) :
  self.canvas=tk.Canvas(self.parent,
                        width=200,height=150,
                        bg="light yellow")
  self.frame=tk.LabelFrame(self.parent,
                           text="Position Carre")
  self.entry=tk.Entry(self.frame)
  self.quit=tk.Button(self.parent,
                      text="Goodbye World",fg="red")
```

Communication entre composants

Liaison Composant-Événement-Action

```
def actions_binding(self) :
  self.entry.bind("<Return>", self.action_entry)
  self.parent.bind("<Control-Z>", self.action_parent)
  self.quit.bind("<Button-1>",self.action_quit)
```

Implémentation des actions

```
def action_entry(self,event):
  event.widget.event_generate("<Control-Z>")
def action_parent(self,event):
  x=int(self.entry.get())
  self.canvas.create_rectangle(x,x,x+10,x+10,
                               fill="blue")
```

Traitement de données utilisateur

Gestion de positionnement de composants

```
def layout(self) :
    self.canvas.pack()
    self.frame.pack()
    self.entry.pack()
    self.quit.pack()
```

Positionnement de composants

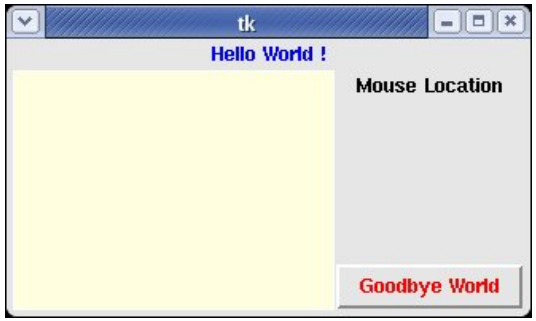
TkInter : Layout manager

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

Regroupement de composants : `Frame`

```
def gui(self,parent) :
  self.hello=tk.Label(self.parent,...)
  self.canvas=tk.Canvas(self.parent,...)
  self.frame=tk.LabelFrame(self.parent,...)
  self.info=tk.Label(self.frame,...)
  self.quit=tk.Button(self.parent,...)
```

Positionnement de composants



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

```
def layout(self) :
  self.hello.pack()
  self.canvas.pack(side="left")
  self.frame.pack(side="top")
  self.info.pack()
  self.quit.pack(side="bottom")
```

Positionnement de composants

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

```
self.hello.pack()
self.frame.pack(fill="both",expand=1)
self.canvas.pack(fill="both",expand=1)
self.label.pack()
self.quit.pack()
```

Positionnement de composants

grid() :agencement ligne/colonne

```
nom_label=tk.Label(parent,text="Nom :")
prenom_label=tk.Label(parent,text="Prenom :")
nom_entry=tk.Entry(parent)
prenom_entry=tk.Entry(parent)

nom_label.grid(row=0)
prenom_label.grid(row=1)
nom_entry.grid(row=0,column=1)
prenom_entry.grid(row=1,column=1)
```

Positionnement de composants

place() : positionnement géométrique

```
msg=tk.Message(parent,text="Place : \n \
          options de positionnement de widgets",
          justify="center",bg="yellow",relief="ridge")
okButton=tk.Button(root,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")
```

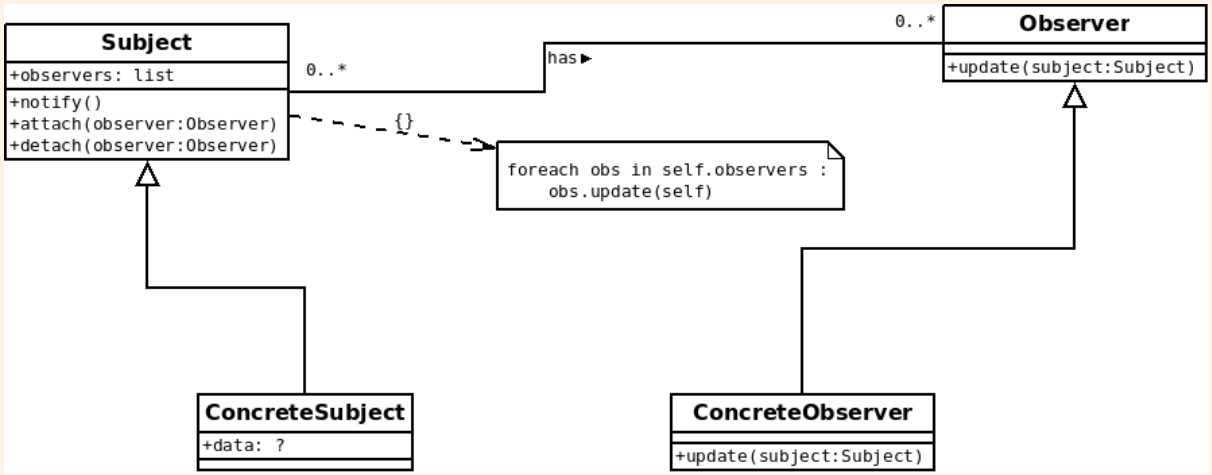
Patrons de conception

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



Suivant ce modèle de conception :

- un `Subject` notifie ses modifications de propriétés
- aux `observers` qui lui sont associés (attachés)
- chaque `Observer` fait alors sa mise à jour `update(subject)`

Les classes héritières devront au moins avoir :

- `ConcreteSubject` : des propriétés à modifier
- `ConcreteObserver` : une méthode de mise à jour

Modèle Observer

Subject : implémentation

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

- `notify()` : demande de mise à jour des observateurs

Toute méthode d'un `ConcreteSubject` modifiant ses propriétes (`data`) devra invoquer cette notification.

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

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())
```

Modèle Observer

Observer : mise à jour

```
class ConcreteSubject(Subject):
    def __init__(self):
        Subject.__init__(self)
        self.__data=0
    def get_data(self):
        return self.__data
    def increase(self):
        self.__data+=1
        self.notify()
if __name__ == "__main__":
    subject=ConcreteSubject()
    obs=ConcreteObserver()
    subject.attach(obs)
    subject.increase()        # obs.update()
```

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

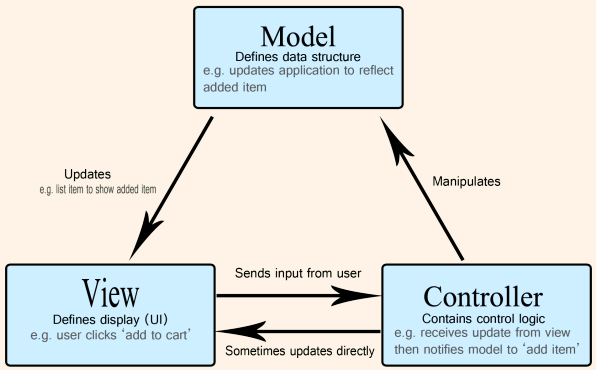
"MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility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

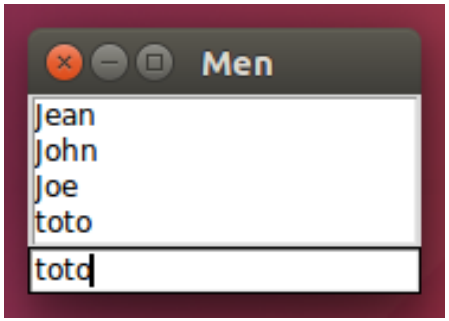


From MDN (Mozilla Developer Network)

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

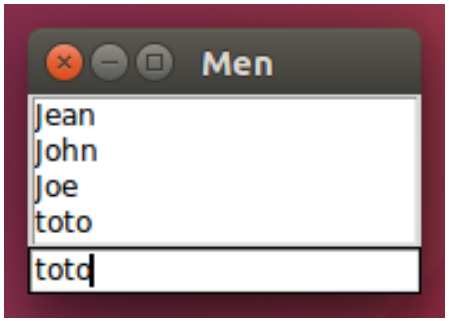
Contrôle du modèle : action utilisateur

```
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

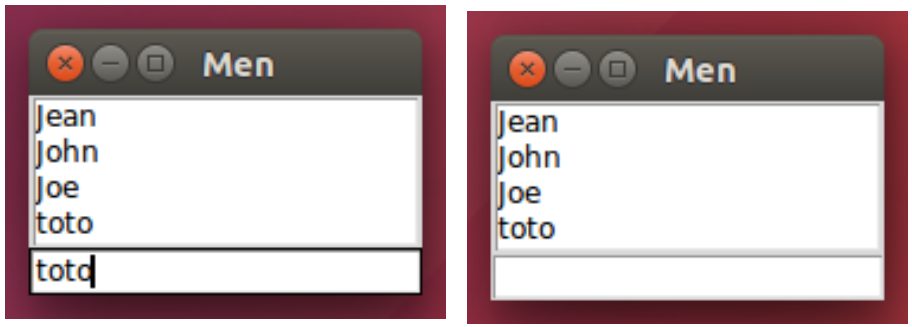
```
if __name__ == "__main__":
    root=tk.Tk()
    root.title("Men")
    names=["Jean", "John", "Joe"]
    model=Model(names)
    view=View(root)
    view.update(model)
    model.attach(view)
    ctrl=Controller(model,view)
```



Test IHM

Un modèle, des vues, des contrôleurs

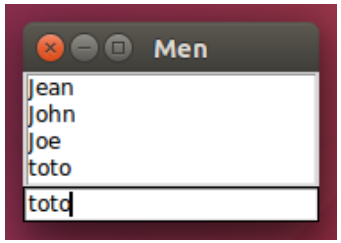
```
...
top=tk.Toplevel()
top.title("Men")
view=View(top)
view.update(model)
model.attach(view)
ctrl=Controller(model,view)
```

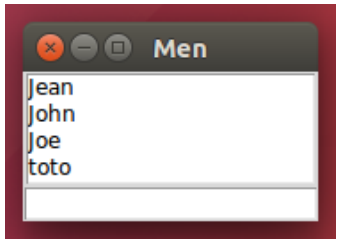


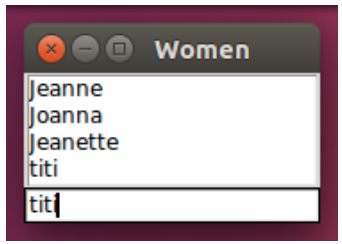
Test IHM

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)
```

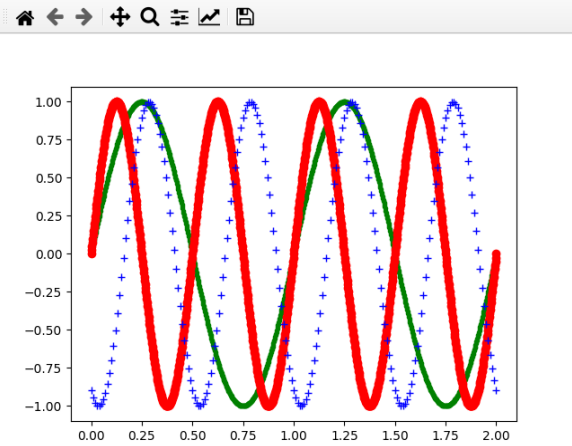


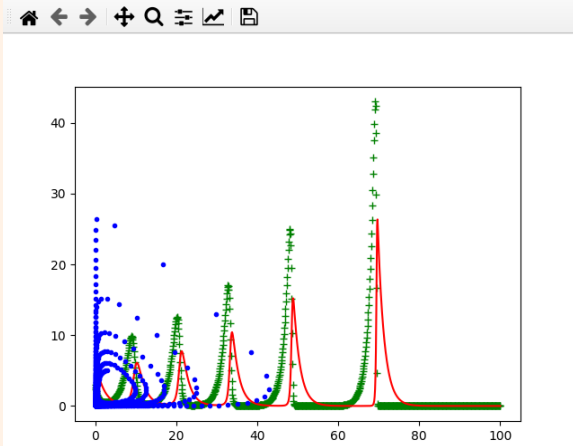


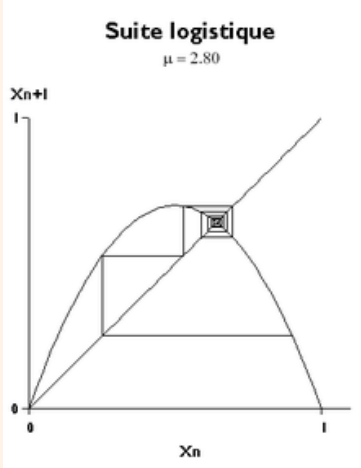


Labos : Générateur de signaux

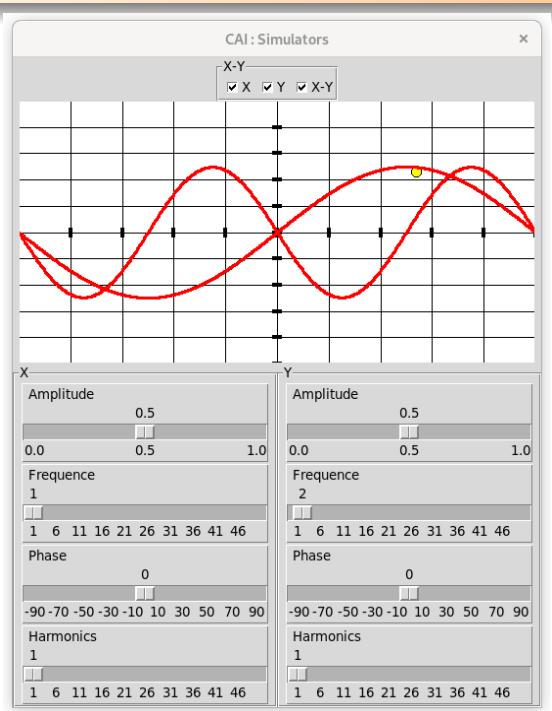
Modéliser, Visualiser et Contrôler des signaux

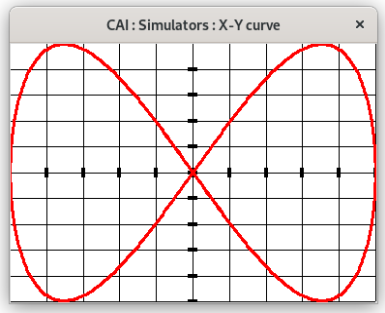






Labos : Générateur de signaux





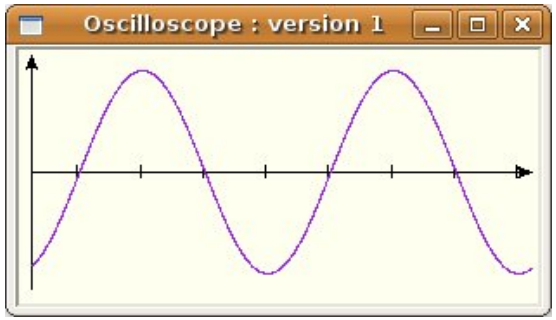
Modéliser, Visualiser et Contrôler des signaux

- contrôler tous les paramètres (amplitude,fréquence ...)
- charger, sauvegarder les paramètres des signaux
- sauvegarder les images représentées dans les vues
- animer un spot sur la trajectoire des courbes

Générateur de signaux

Vibration pure : mouvement vibratoire sinusoïdal

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



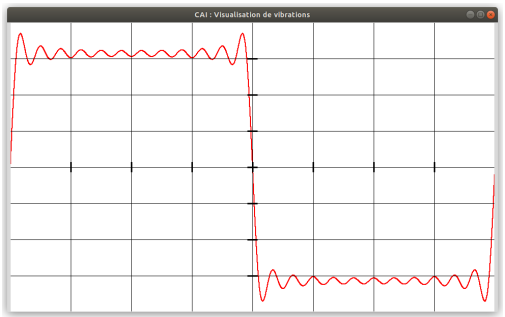
- e, t : élongation , temps
- a, f, ϕ : amplitude,fréquence,phase

Générateur de signaux

Vibration complexe : mouvement vibratoire avec harmoniques

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

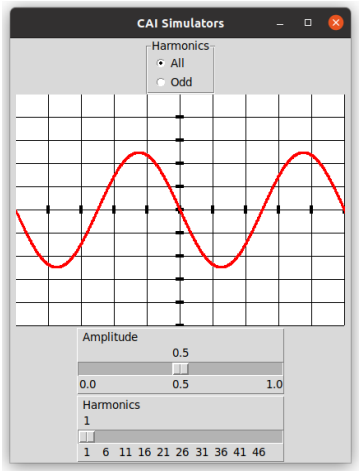
- e, t : élongation , temps
- a, f, ϕ : amplitude,fréquence,phase
- h : nombre d'harmoniques



Générateur de signaux

Tout dans la classe (`Generator`)

```
root=tk.Tk()
mw=Generator(root)
mw.generate()
mw.create_grid()
mw.plot_signal(mw.get_name(),mw.get_signal())
mw.packing()
root.mainloop()
```



Générateur de signaux

Tout dans la classe (Generator)

```
class Generator :
  def __init__(self,parent,bg="white",name="X",...):
    self.parent=parent
    self.name=name
    self.m,self.f,self.p=1.0,2.0,0.0
    self.harmonics=1
    ...
    self.gui()
    self.actions_binding()
#    self.action_<name>(...)
```

class Generator

Création des composants

```
def gui(self) :
  self.screen=tk.Canvas(self.parent,
                        bg=self.bg,
                        width=self.width,
                        height=self.height)
  self.frame=tk.LabelFrame(self.parent,text=self.name)
  self.var_mag=tk.IntVar()
  self.var_mag.set(1)
  self.scaleA=tk.Scale(self.frame,
                       variable=self.var_mag,
                       label="Amplitude",
                       orient="horizontal",length=250,
                       from_=0,to=5,tickinterval=1)
```

class Generator

liaisons des actions

```
  def actions_binding(self) :
    self.screen.bind("<Configure>",self.action_resize)
#    self.<item>.bind("<EventType>",self.action_<name>)
    self.scaleA.bind("<B1-Motion>",self.action_magnitude)
```

Implémentation des actions

```
  def action_resize(self, event):
    self.width = event.width
    self.height = event.height
    self.screen.delete("grid")
    self.create_grid()
    self.plot_signal(self.name,self.signal)
#  def self.action_<name>(self,*args) :
#    pass
```

class Generator

Implémentation des actions

```
def action_magnitude(self,event):
  if  self.m != self.var_mag.get() :
    self.m=self.var_mag.get()
    self.update()
```

Mise à jour

```
def update(self):
  self.generate()
  if self.signal :
    self.plot_signal(self.name,self.signal)
```

Design Pattern : `Observer.update(self,subject)`

class Generator

Calcul d'une élongation au temps t

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

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

class Generator

Calcul du signal sur un échantillon (self.samples)

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

class Generator

Visualisation de signal

```
def plot_signal(self,name="X",signal=[],color="red"):
  if signal and len(signal)>1:
    width,height=self.width,self.height
    if self.screen.find_withtag(name) :
      self.screen.delete(name)
    plot=[(x*width,(height/self.units)*y+height/2) \
                  for (x,y) in signal]
    self.canvas.create_line(plot,\
                            fill=color,smooth=1,\
                            width=3,tags=name)
```

class Generator

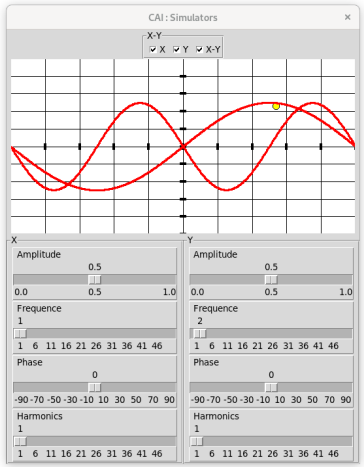
Grille de visualisation

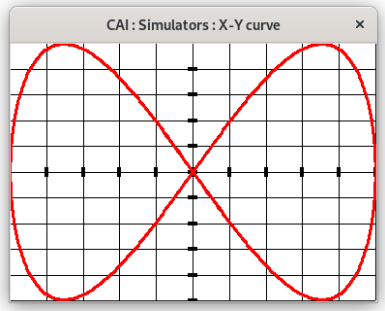
```
def create_grid(self):
  width,height=self.width,self.height
  tiles=self.tiles
  tile_x=width/tiles
  for t in range(1,tiles+1): # lignes verticales
    x=t*tile_x
    self.canvas.create_line(x,0,
                            x,height,
                            tags="grid")
    self.canvas.create_line(x,height/2-5,
                            x,height/2+5,
                            width=4,tags="grid")
```

class Generator

Grille de visualisation

```
tile_y=height/tiles
for t in range(1,tiles+1): # lignes horizontales
  y=t*tile_y
  self.canvas.create_line(0,y,width,y,tags="grid")
  self.canvas.create_line(width/2-5,y,width/2+5,y,
                          width=4,tags="grid")
```





Modèle MVC

Modéliser, Visualiser et Contrôler des signaux

Séparation en trois classes (`Generator`,`Screen`,`Controller`)

- plusieurs `Generator` à afficher sur un `Screen`
- plusieurs `Screen` pour visualiser un même `Generator`
- des `Controller` sur chaque `Generator`

Organisation de l'application

- `observer.py` : patron de conception (`Subject`,`Observer`)
- `generator.py` : **M**odèle de signal (héritage `Subject`)
- `screen.py` : **V**isualisation de signaux (héritage `Observer`)
- `controls.py` : **C**ontrôle de modèles de signaux
- `main.py` : intégration dans l'application finale

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)

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