Programming of Interactive Systems

Anastasia.Bezerianos@lri.fr

Week 6 :
a. System structure and Toolkits

Anastasia.Bezerianos@lri.fr

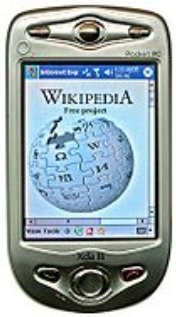
(part of this class is based on previous classes from Anastasia,
and of T. Tsandilas, S. Huot, M. Beaudouin-Lafon, N.Roussel, O.Chapuis)

graphical interfaces

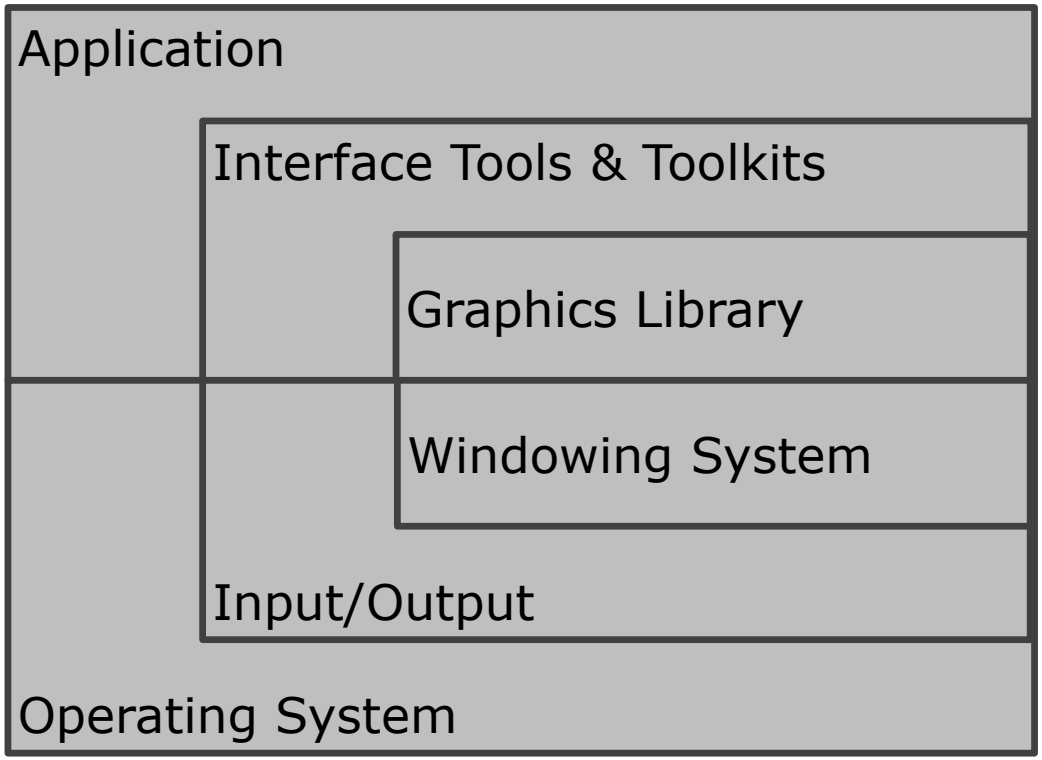
GUIs: input is specified w.r.t. output

Input peripherals specify commands at specific locations on the screen (*pointing*), where specific objects are drown by the system.

Familiar behavior from physical world



software layers



Applications/Communication (MacApp)

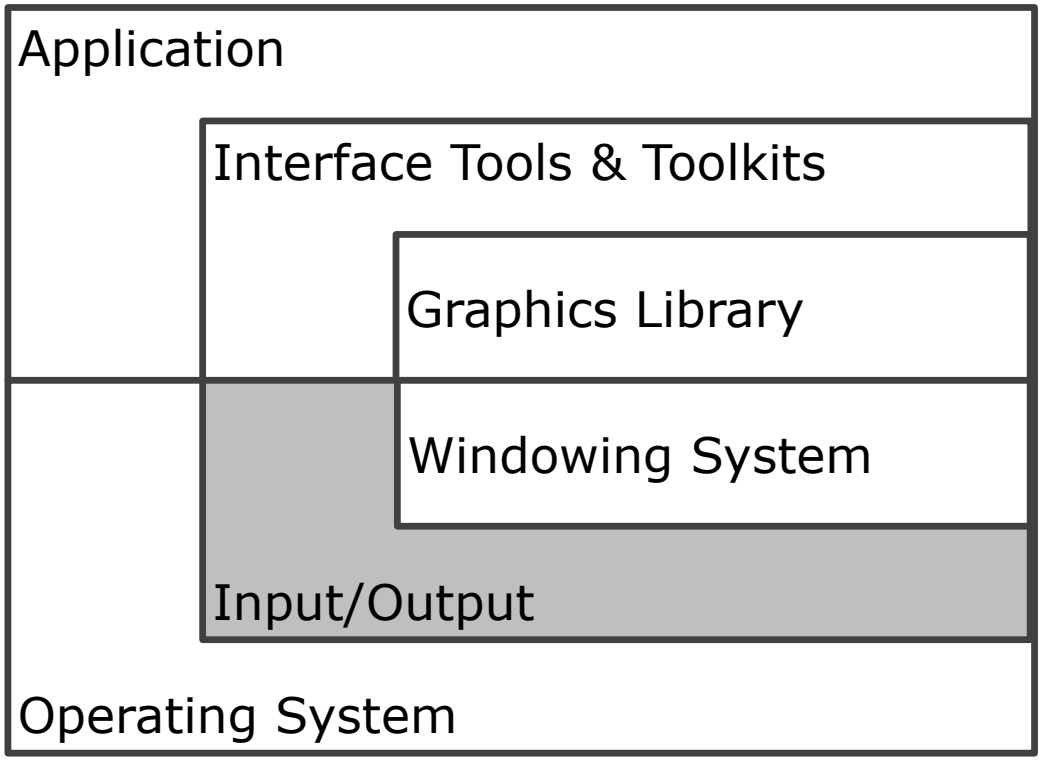
Builders, Java Swing, JavaFX, Qt, GTK+, MFC, Cocoa

GDI+, Quartz, GTK+/Xlib, OpenGL

X Windows (+KDE or GNU)

Windows, Mac OS, Unix, Linux, Android, iOS, WindowsCE

software layers



input/output peripherals

input: where we give commands

output: where the system shows its state

interactivity vs. computing

closed systems (computation):

- read input, compute, produce result
- final state (end of computation)

open systems (interaction):

- events/changes caused by environment
- infinite loop, non-deterministic

problem

- we learn to program algorithms (computational)
- most languages (C/C++, Java, Lisp, Scheme, Pascal, Fortran, ...) designed for algorithmic computations, not interactive systems

problem

treating input/output during computation (interrupting computation) …

- write instructions (`print, put, send,…`) to send data to output peripherals
- read instructions (`read, get, receive,…`) to read the state or state changes of input peripherals

problem

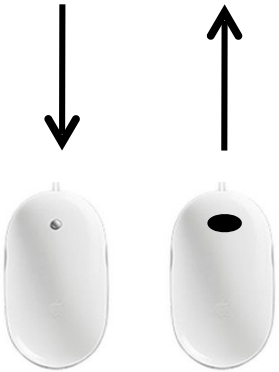
to program IS in algorithmic/computational form:

```
two buttons B1 and B2
finish <- false
while not finish do
          button <- waitClick ()
                    //interruption, blocked comp.
          if button
                B1 : print « Hello World »
                B2 : finish <- true
          end
end
```

managing input

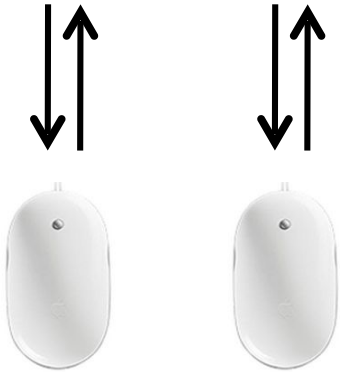
Querying

Query & wait
1 per. at a time



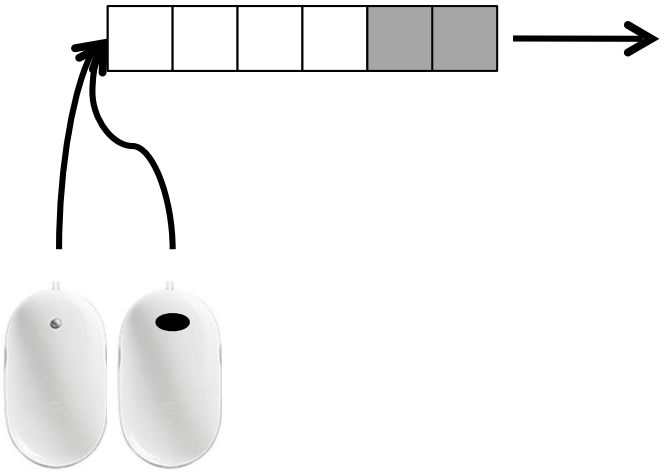
Polling

Active wait
Polling in sequence
CPU cost



Events

Wait queue

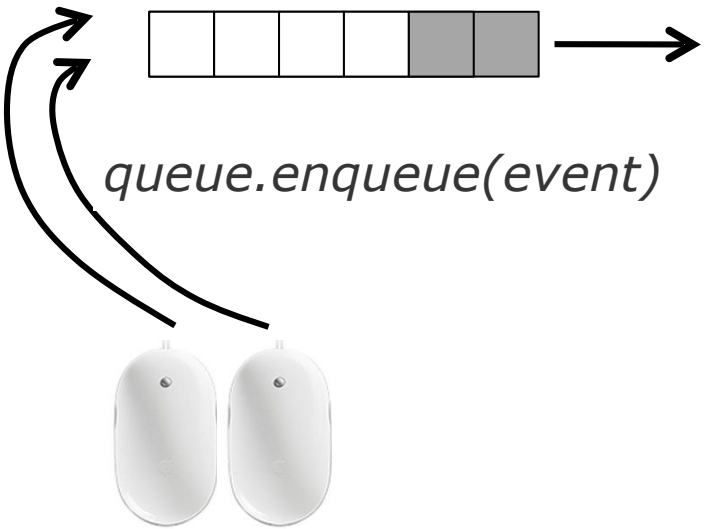


event based (driven) programming

```
while active
  if queue is not empty
     event <- queue.dequeue()
     source <- findSource(event)
     source.processEvent(event)
  end if
end while
```

Source :
Mouse Click

event (waiting) queue



queue.enqueue(event)

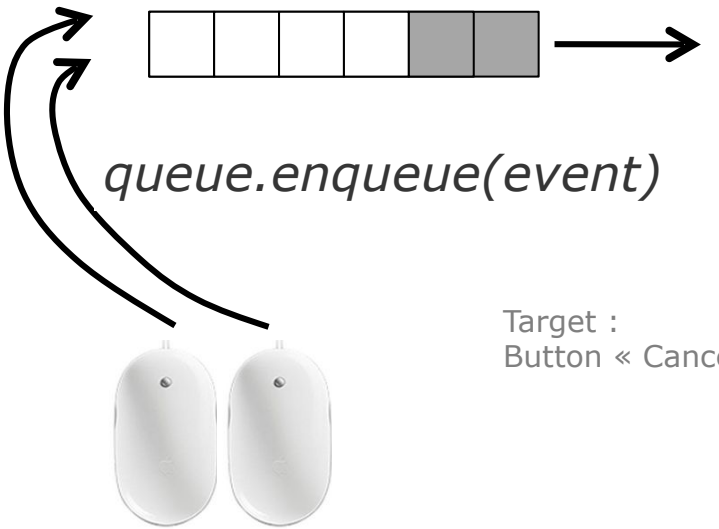
Animations : « clock » source of events
« tick » -> event -> animation progression

event based (driven) programming

```
while active
  if queue is not empty
     event <- queue.dequeue()
     source <- findSource(event)
     source.processEvent(event)
  end if
end while
```

Source :
Mouse Click

event (waiting) queue



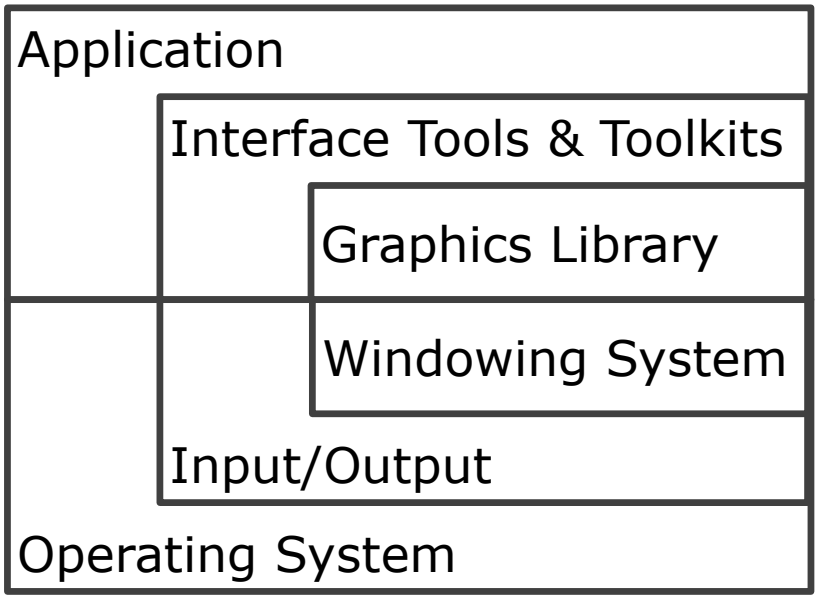
queue.enqueue(event)

```
processEvent(event)
   target <- FindTarget (event)
   if (target ≠ NULL)
    target.processEvent(event)
```

Target :
Button « Cancel »

Animations : « clock » source of events
« tick » -> event -> animation progression

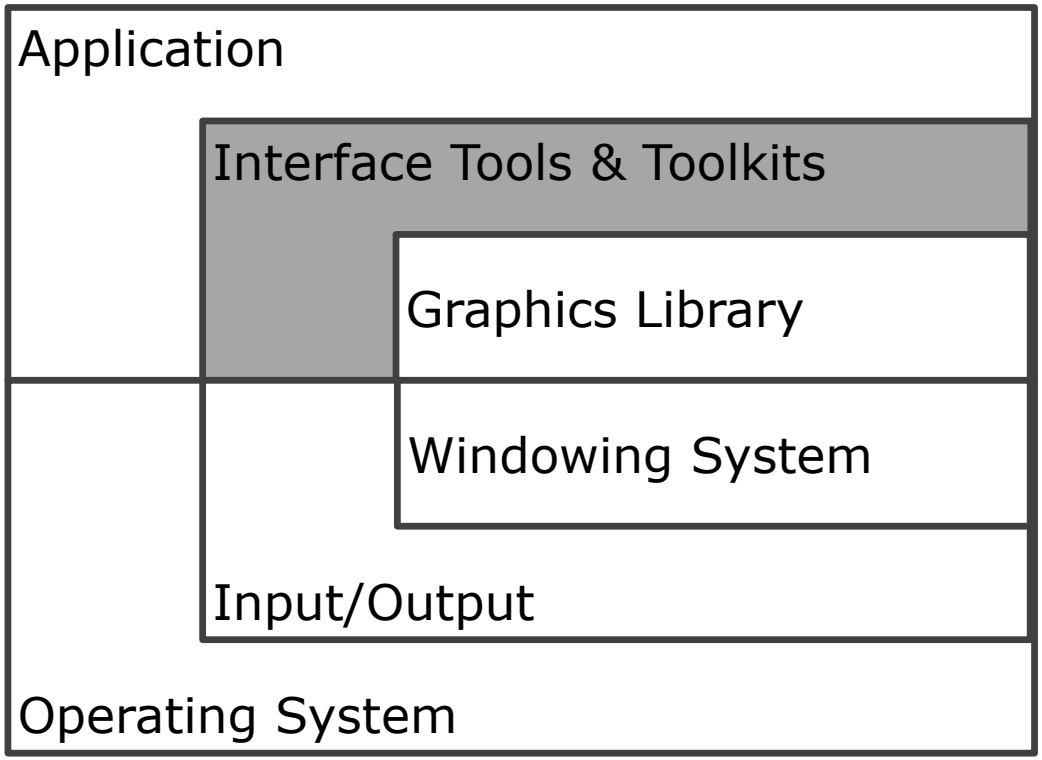
event handling



Lower layers fill-up the queue

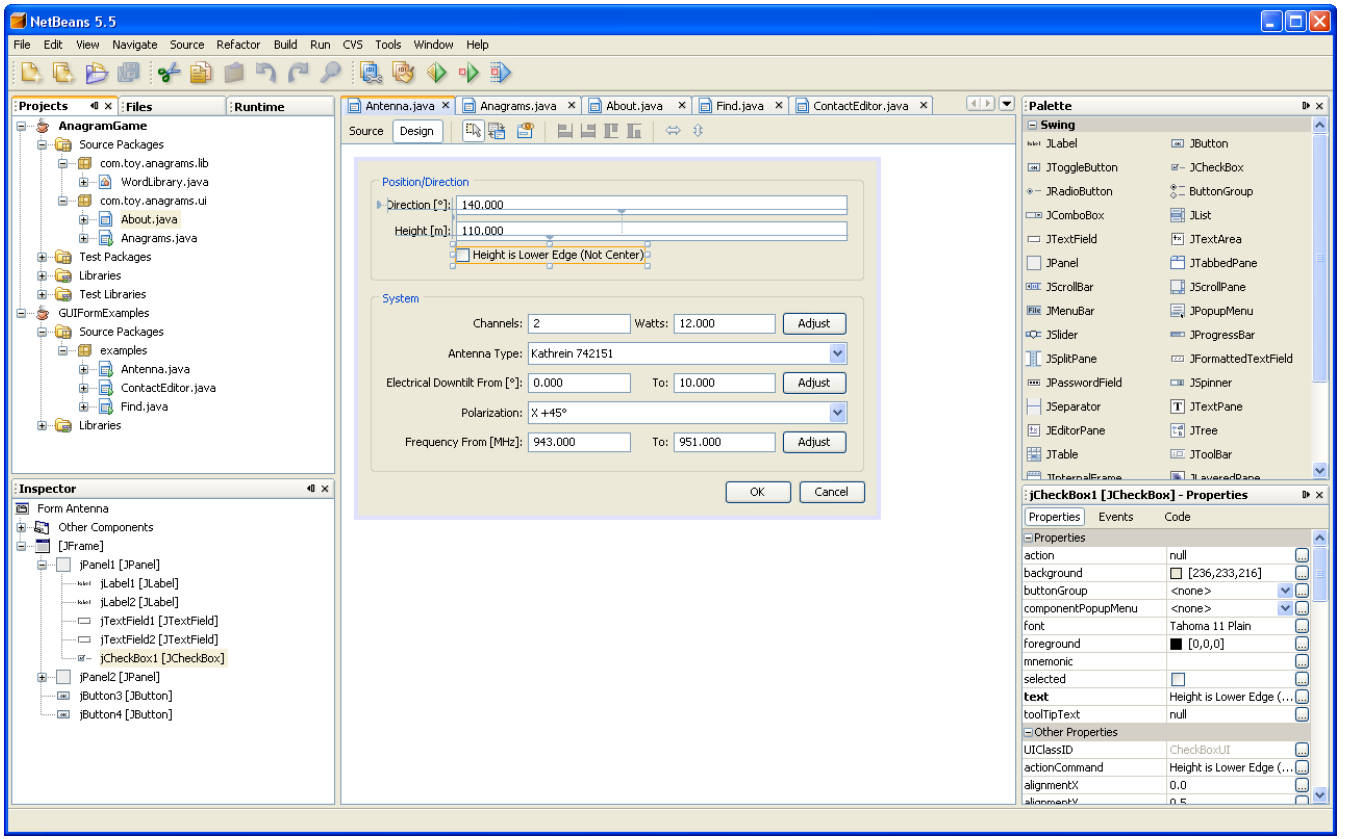
Upper layers de-queue and treat events

software layers



We focused on a specific Tookit (JavaFX),
but there are several comment characteristics …

interface builders



Examples :

SceneBuilder (JavaFX),
MS Visual Studio (C++, C#, etc.), NetBeans (Java), Interface Builder (ObjectiveC), Android Layout Editor

interface builders

can be used to

- create prototypes (but attention it looks real)
- get the « look » right
- be part of final product

- design is fast
- modest technical training needed
- can write user manuals from it

But: still need to program (and clean code …)

interface toolkits

libraries of interactive objects (« widgets », e.g. buttons) that we use to construct interfaces

functions to help programming of GUIs and handle input events

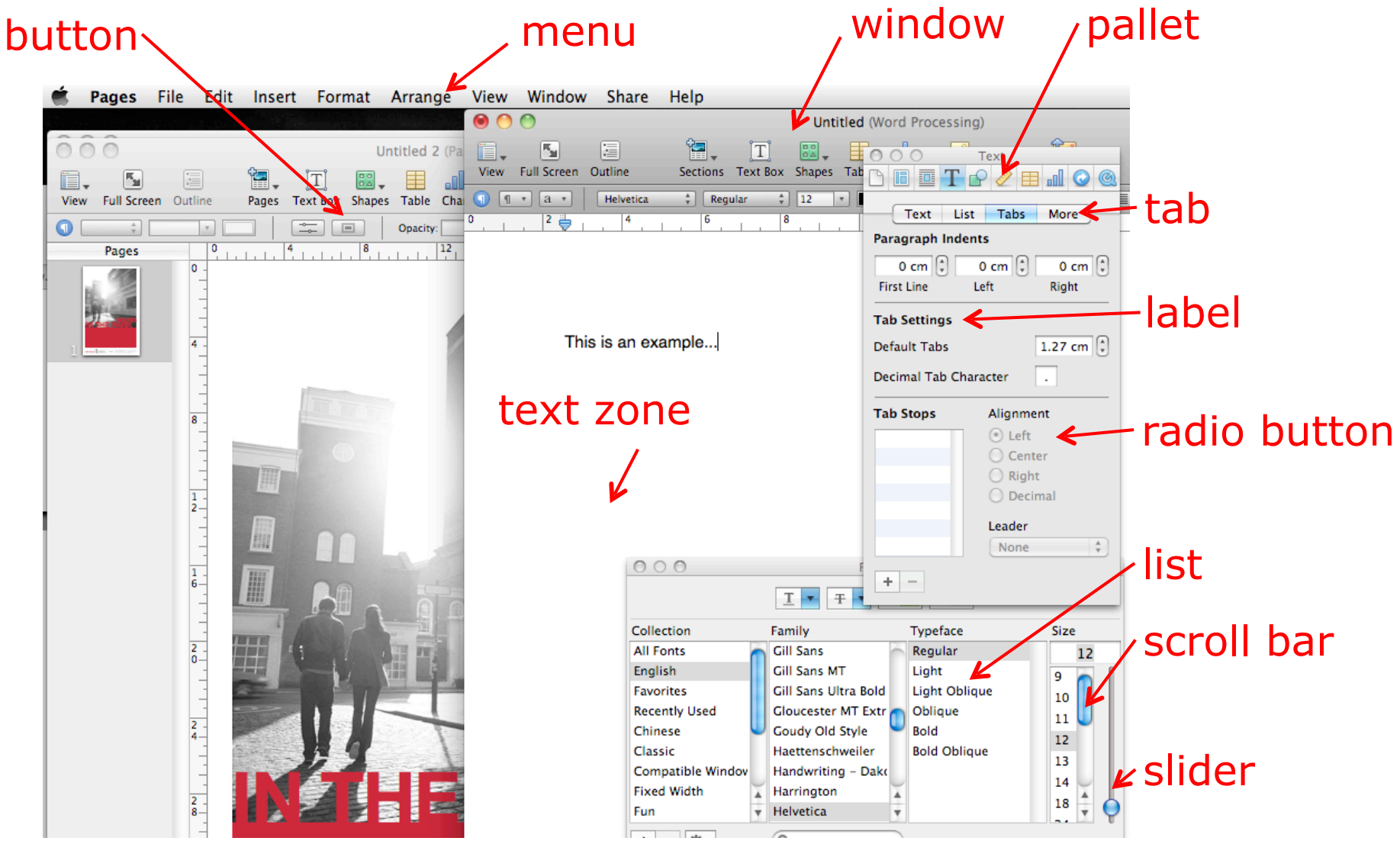
interface toolkits

Toolkit	Platform	Language
Qt	multiplatform	C++
GTK+	multiplatform	C
MFC later WTL	Windows	C++
WPF (subset of WTL)	Windows	(any .Net language)
FLTK	multiplatform	C++
AWT / Swing / JavaFX	multiplatform	Java
Android	Android	Java
iOS	iOS/ WatchOS	Objective C / Swift
Cocoa	MacOs	Objective C / Swift
Gnustep	Linux, Windows	Objective C
Motif	Linux	C
JQuery UI	Web	javascript

Problem with toolkits? ….

treating events

« widgets » (window gadget)



facets of a widget

presentation

- appearance

behavior

- reaction to user actions

interface with the application:
notification of state changes

Button:
border with text inside
« pressing » or « releasing » animation when clicked
call function when the button is clicked

facets of a widget

presentation

- appearance

behavior

- reaction to user actions

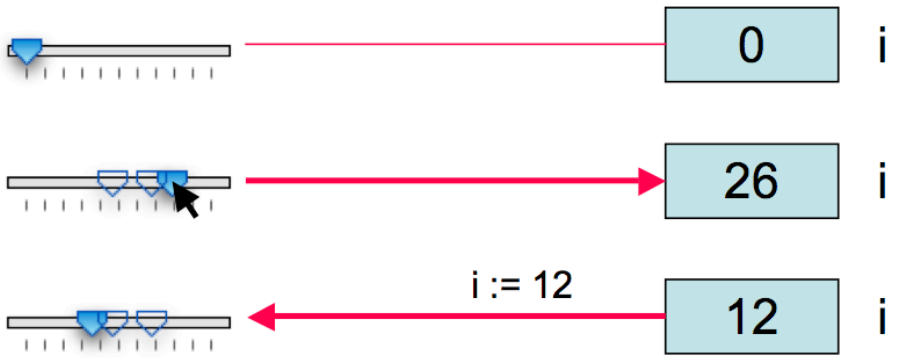
interface with the application:
notification of state changes

- active/linked/wrapped variables (Tcl/Tk)
- event dispatching (Qt)
- callback functions (Swing/JavaFX)

1. variable wrappers (active variables)

two-way link between a state variable of a widget and another application variable

(in Tcl/Tk referred to as *tracing, related to binding in JavaFX*)




```
main (){
    int i = 0;
    …
    /* widget */
    nc = CreateSlider (…);
    /* active var */
    SetIntegerActiveVariable (nc, &i);
    …
}
```

problems

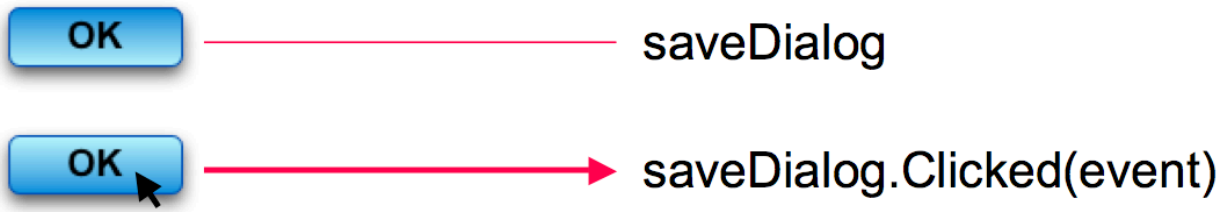
- limited to simple types
- return link can be costly if automatic
- errors when links are updated by programmers

2. event dispatching

widgets act as input peripherals and send events when their state changes

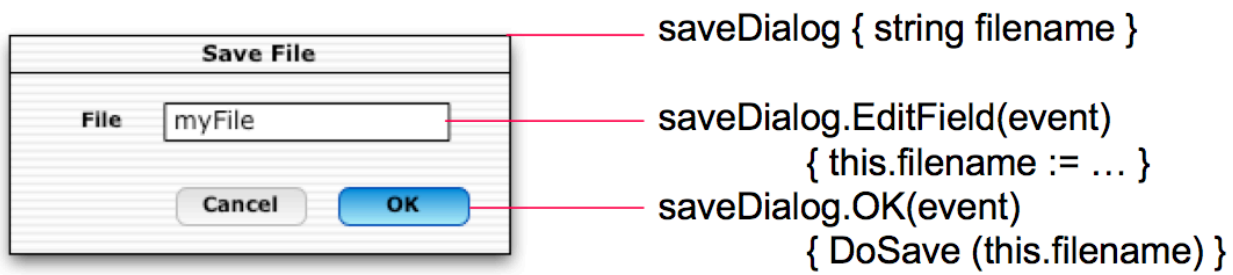
a "while" loop reads and treats events

associate an object to a widget, and its methods to changes in the widget state



(Unity, Qt, ...)

2. event dispatching

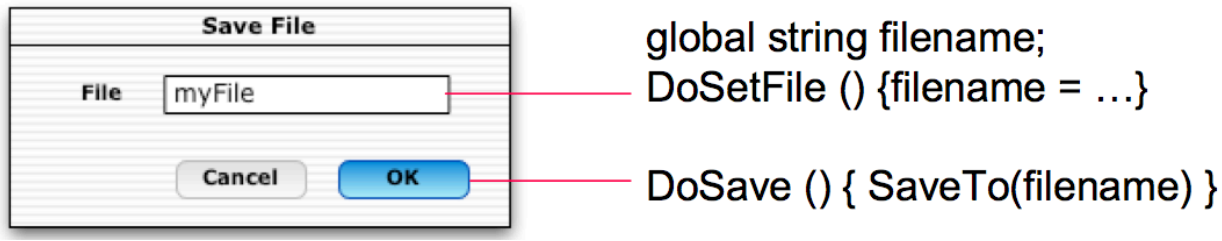


- event sending and treatment
- better encapsulation (inside widget class)
- but when similar behaviors exist …

3. callback functions

Registration at widget creation

Call at widget activation



3. callback functions

Problem: spaghetti of callbacks

Sharing a state between multiple callbacks by:

- global variables: widgets check them
  - too many in real applications
- widget trees: callback functions are called with a reference to the widget that called it (visible in the same tree)
  - Fragile if we change the structure of the UI, does not deal with other data not associated to widgets (e.g. filename)
- token passing: data passed with the callback function call

3. callback functions

```
/* callback function */
void DoSave (Widget w, void* data) {
     /* retrieve file name */
      filename = (char**) data;
     /* call an application function */
     SaveTo (filename);
     /* close the dialog */
     CloseWindow (getParent(getParent(w)));
}

/* main program */
main () {
     /* variable with file name */
     char* filename = "";
     …
     /* create a widget and assosiate a callback */
     ok = CreateButton (....);
     RegisterCallback (ok, DoSave, (void*) &filename);
     …
     /* event manager loop */
     MainLoop ();
}
```

Callbacks registered once

token

Events generated here, treated by calling appropriate callback

event listeners (Java)

Listeners/Handlers
are a variation of callbacks in Java:

methods of type `addListener` or `setOn…` that do not specify a callback function but an object, the **listener/handler**

when a widget changes state, it triggers a predefined method of the **listener** object (e.g. `actionPerformed() or handle()`)

input treatment

interface with the application:
notification of state changes

- active/linked/wrapped variables
- event dispatching
- callback functions / handlers

Callbacks/listeners still produce the most reusable (but potentially most complex) code

interface toolkits

event-action model

- can lead to errors (e.g., forgotten events)
- difficult to extend (e.g., add hover events)
- complex code

=> Finite State Machine and Hierarchical SM

hard to do things the toolkit was not designed for

e.g. multi-device input, multi-screen applications, advanced interaction techniques (CrossY)