

# Programmation des systèmes interactifs

1<sup>ère</sup> école aéronautique européenne

Anke Brock (C113), Jérémie Garcia (C107), Catherine Letondal (C115),  
Mathieu Magnaudet (C103), Célia Picard (C109), Nicolas Saporito (C105)

[prenom.nom@enac.fr](mailto:prenom.nom@enac.fr)

# Positionnement dans le cours de programmation IENAC



- Introduction, cours et TPI à 6,
- Manipulation de textes et fichiers
- Approfondissements
- **Programmation des systèmes interactifs: 12h**
  - Algorithmique
  - Calculs numériques
  - Projet de programmation

# Pourquoi apprendre à programmer des *IHM* ?

- Une façon ludique de pratiquer la programmation,
- introduction de notions réutilisables dans d'autres domaines (programmation système, réseau...),
- de nombreux projets de votre scolarité nécessitent de contrôler et visualiser des calculs; beaucoup de projets de fin d'études comportent du prototypage et du développement d'applications interactives,
- vous serez des utilisateurs d'*IHM* plus avertis !

J'entends et j'oublie, je vois et je me souviens, je fais et je comprends.  
Proverbe chinois

semestre	cours associés
5	Introduction à la programmation + Approfondissement en programmation de systèmes interactifs
6	Ergonomie et facteurs humains + Ingénierie du besoin et des exigences
7	Conception orientée objet + Programmation orientée objet ( <i>OPS, minSITA</i> ), BE Image radar ( <i>minSITA</i> ), <i>PIR</i>
8	Conception d'IHM ( <i>SITA/AVI</i> ), Conception orientée objet + Programmation orientée objet + Programmation événementielle + Projet Java ( <i>SITA, PIR</i> )
9	<i>PIR, minSITA</i> + projets de synthèse ( <i>SITA, AVI, Master IHM ou IATSED</i> )
10	<b>Très probablement</b> votre PFE



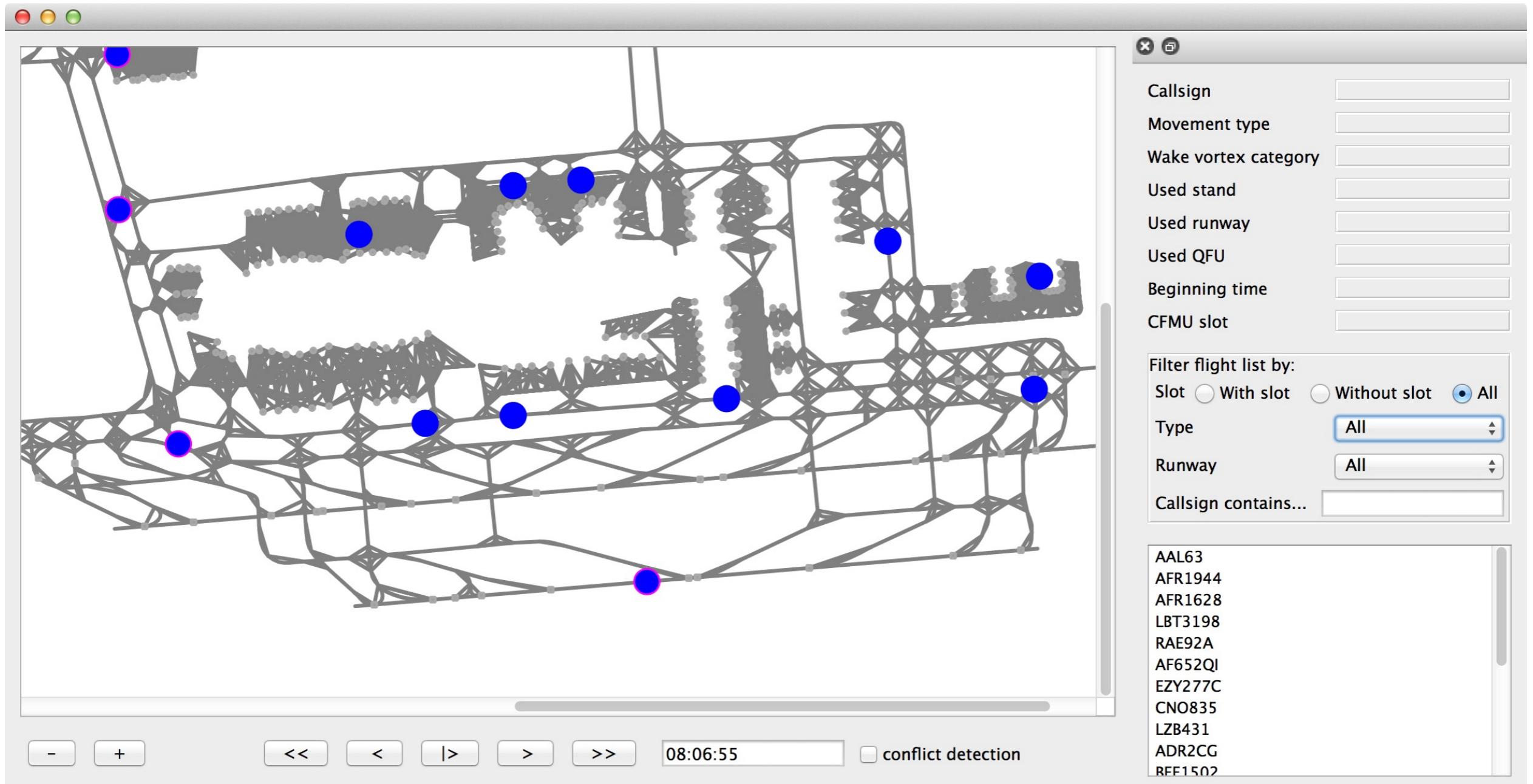
# Objectifs cours + TP

- Savoir utiliser l'**API** de la bibliothèque *Qt* pour programmer des **IHM**,
- savoir décrire le modèle d'exécution d'une application interactive (flot d'exécution vs flot de programmation, mécanismes d'affichage),
- savoir utiliser et assembler des **widgets**,
- savoir structurer le code d'une application réactive, utiliser les **signaux**, programmer des interactions de **manipulation directe** (prendre, glisser, déposer).



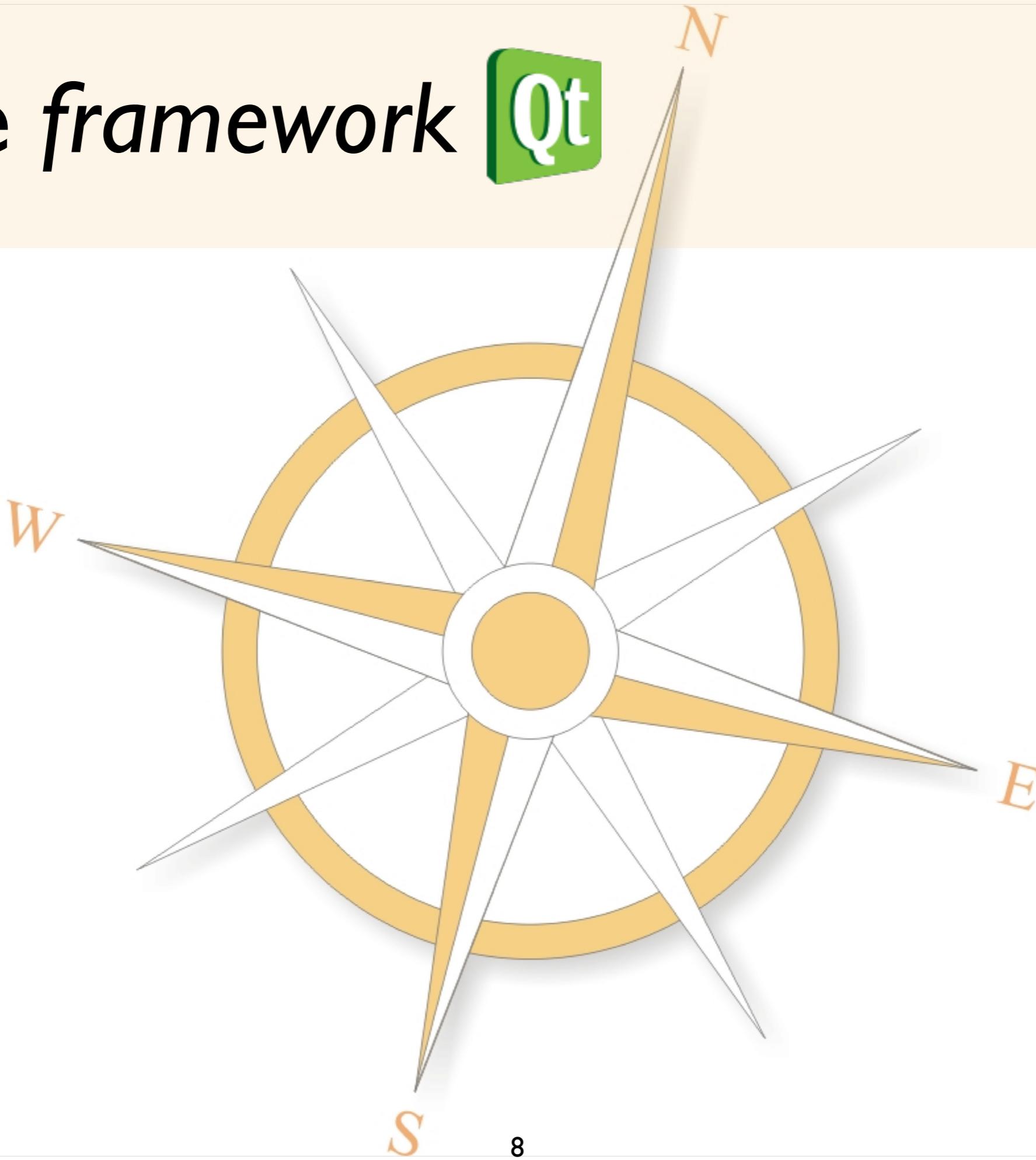
# Déroulement

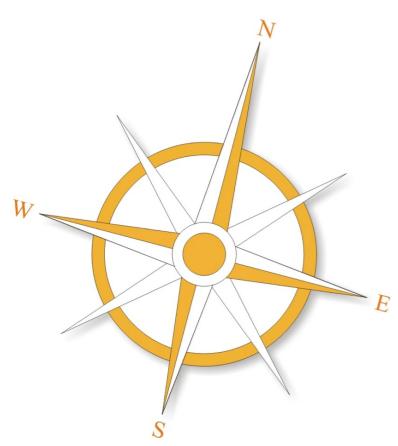
1. Présentation du *framework Qt*
2. Identification des notions de *flot d'exécution* dans un programme fourni, premières modifications du programme
3. Programmation *WIMP*, utilisation de *widgets* avec Qt et Qt Designer
4. Mécanismes d'affichage
5. Structuration d'une application interactive, programmation d'une interaction à *manipulation directe*



# Approche outside in, par la pratique

# I - Le framework





# I.I - Le framework





# Fiche d'identité

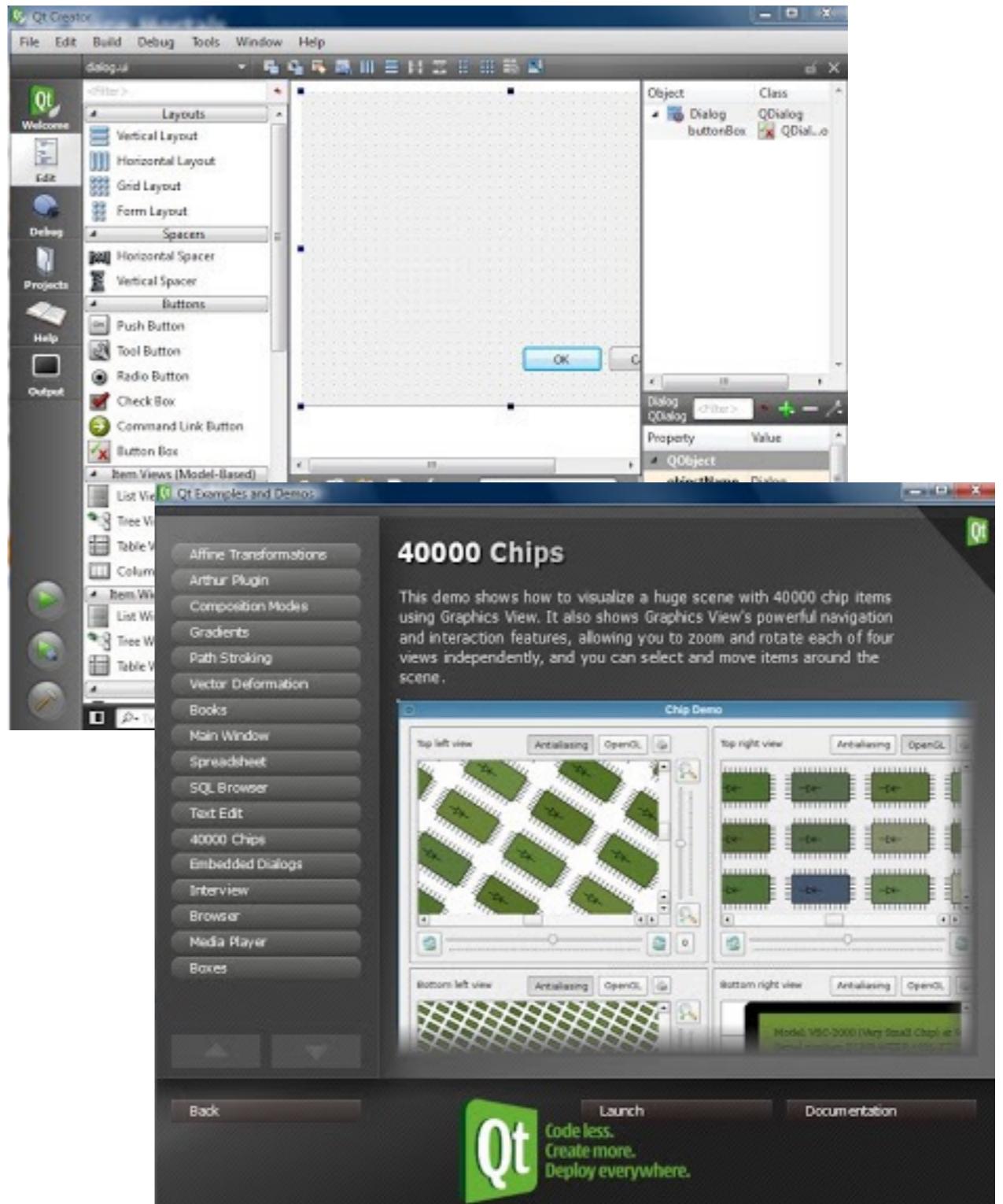
- Une des boites à outils (*toolkit, framework*) les plus populaires pour le développement d'applications interactives, projet débuté en 1991,
- très utilisée pour le desktop (Linux, MacOS, Windows...), sur mobile (iOS, android), ainsi que dans l'industrie aéronautique (prototypage, applications support),
- logiciel libre (GPL v3 et LGPL v2.1) ou commercial, multi plate-formes (Linux, Windows, MacOSX, Android...),
- interface de programmation d'applications (**API**) dans plusieurs langages (C++ natif, mais aussi Python, Java, Objective-C...).

<https://www.qt.io>

# Quelques outils associés

Framework Qt 5 à installer

- **Qt Assistant** (documentation),
- **Qt Designer** construction graphiques d'interfaces,
- **qtdemo** exemples de réalisations,
- QML + Javascript et **QtQuick**,
- et votre **IDE** préféré...  
(Environnement de développement intégré)



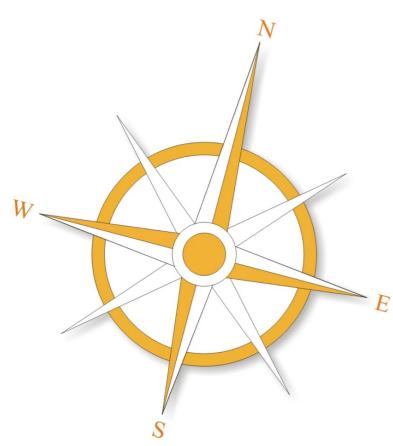
# Comment utiliser Qt avec python ?

- Renseigner l'interpréteur, si le système est bien installé :

```
from PyQt5.QtCore import ...
from PyQt5.QtGui import ...
from PyQt5.QtWidgets import ...
```

(les *IDE* peuvent en général faire les imports automatiquement)

- créer une `app = QApplication()` qui va gérer la transmission des événements et les mécanismes d'affichage,
- créer un ou plusieurs widgets et les montrer,
- entrer dans la boucle principale `app.exec()`



## I.2 - Qu'est ce qu'un *widget* ?

**Widget** = objet graphique interactif semi-autonome

Exemples :

le bouton (QPushButton)



la barre de défilement  
(QScrollBar)



**trois facettes :**

présentation, comportement, lien avec application

# Présentation

Cancel

OK

ou « look »

# Présentation

*mac mavericks*

Cancel

OK

*win8*

OK

Cancel

*gnome/ubuntu*

Cancel

OK

La présentation dépend généralement de l'environnement d'exécution mais est paramétrable dans certaines toolkit (dont Qt).

# Comportement



Essayez par vous-même :  
Que se passe-t-il si le curseur sort du widget avec le bouton de la souris toujours enfoncé ?

# Comportement



Essayez par vous-même :

Quel est le comportement de la scrollbar quand le curseur dépasse les limites de la scrollbar ? Et va loin de la scrollbar ?

# Comportement



ou « **feel** » (d'où la notion de « look and feel »)

# Comportement



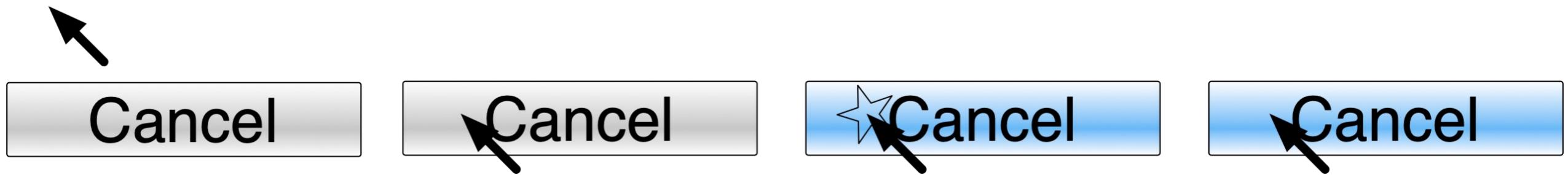
Widget = objet graphique interactif **semi-autonome**

Le comportement est déjà implémenté pour vous programmeurs (en tant qu'assembleur de widgets), et fonctionne (presque) tout seul.

Note : « comportement » est parfois appelé « logique » chez certains industriels

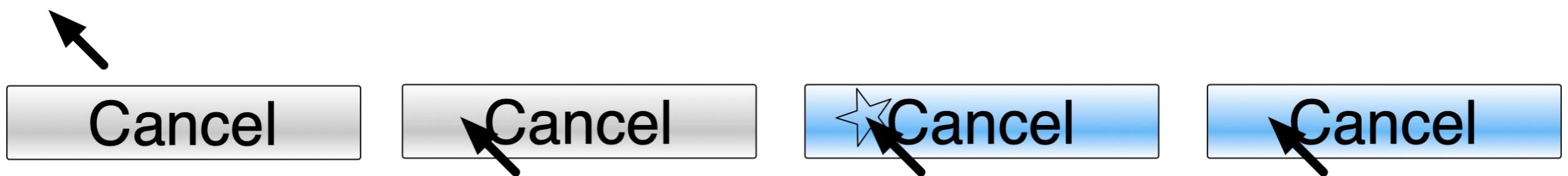
Le comportement est **peu voire pas** paramétrable dans les toolkits (dont Qt).

# Lien avec l'application



Essayez par vous-même :  
Que se passe-t-il quand je relâche le bouton de la souris  
sur le widget bouton ?

# Lien avec l'application



Widget = objet graphique interactif **semi-autonome**

Le widget fonctionne tout seul, mais il faut qu'il **contrôle** l'application (déclenchement d'une action dans le cas du bouton).

Le lien avec l'application est **par essence** paramétrable dans les toolkits (avec Qt, on utilise la notion de « **signal** » pour implémenter ce lien).



# QCM

Qu'est-ce que c'est un widget ?

1. Un environnement de développement pour concevoir des interfaces
2. Un élément de l'interface avec un comportement associé que l'utilisateur déclenche
3. Un élément statique de l'interface qui affiche l'état du système
4. Une action déclenchée par le système sans intervention par l'utilisateur



# QCM

En général, qu'est-ce qui doit toujours être paramétré dans un widget ?

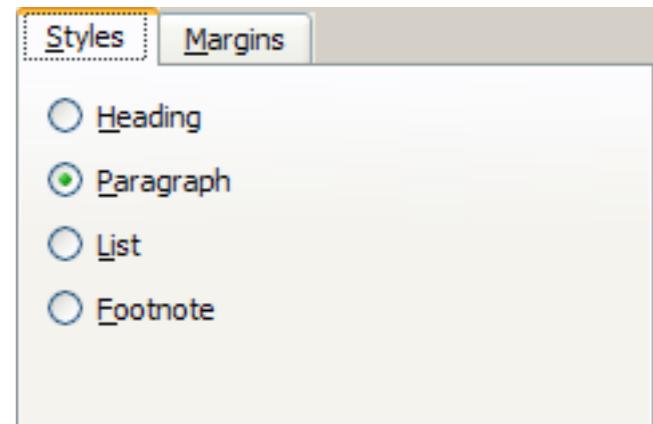
1. Un comportement
2. Une présentation
3. Un lien avec l'application

# Widgets : en résumé

Widget = objet graphique interactif semi-autonome, avec trois facettes :

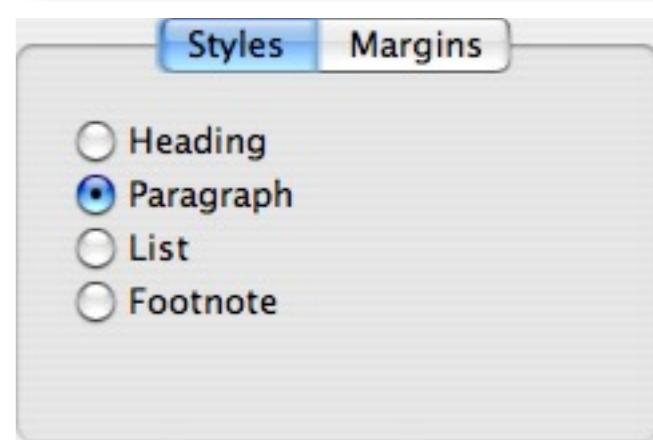
- **présentation** : aspect graphique

- plus ou moins paramétrable (ex : ressources Xt ou thèmes GTK)
- «*look*» du «*look and feel*»



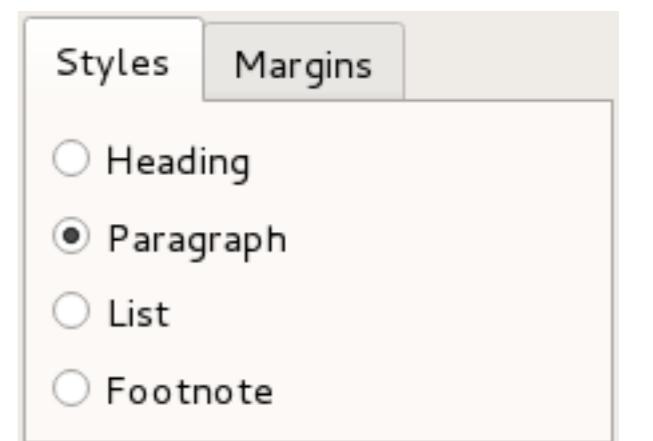
- **comportement** (autonome) : réactions aux actions de l'utilisateur

- généralement, peu ou pas paramétrable
- «*feel*» du «*look and feel*»

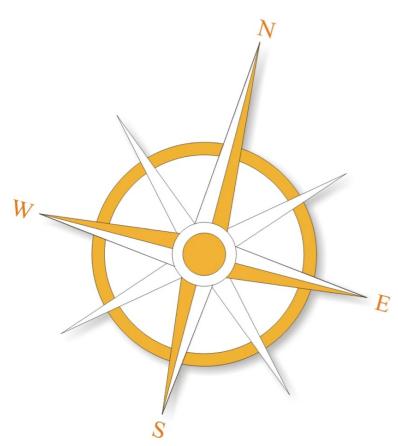


- **interface** d'application : lien avec l'application, notification des changements d'état

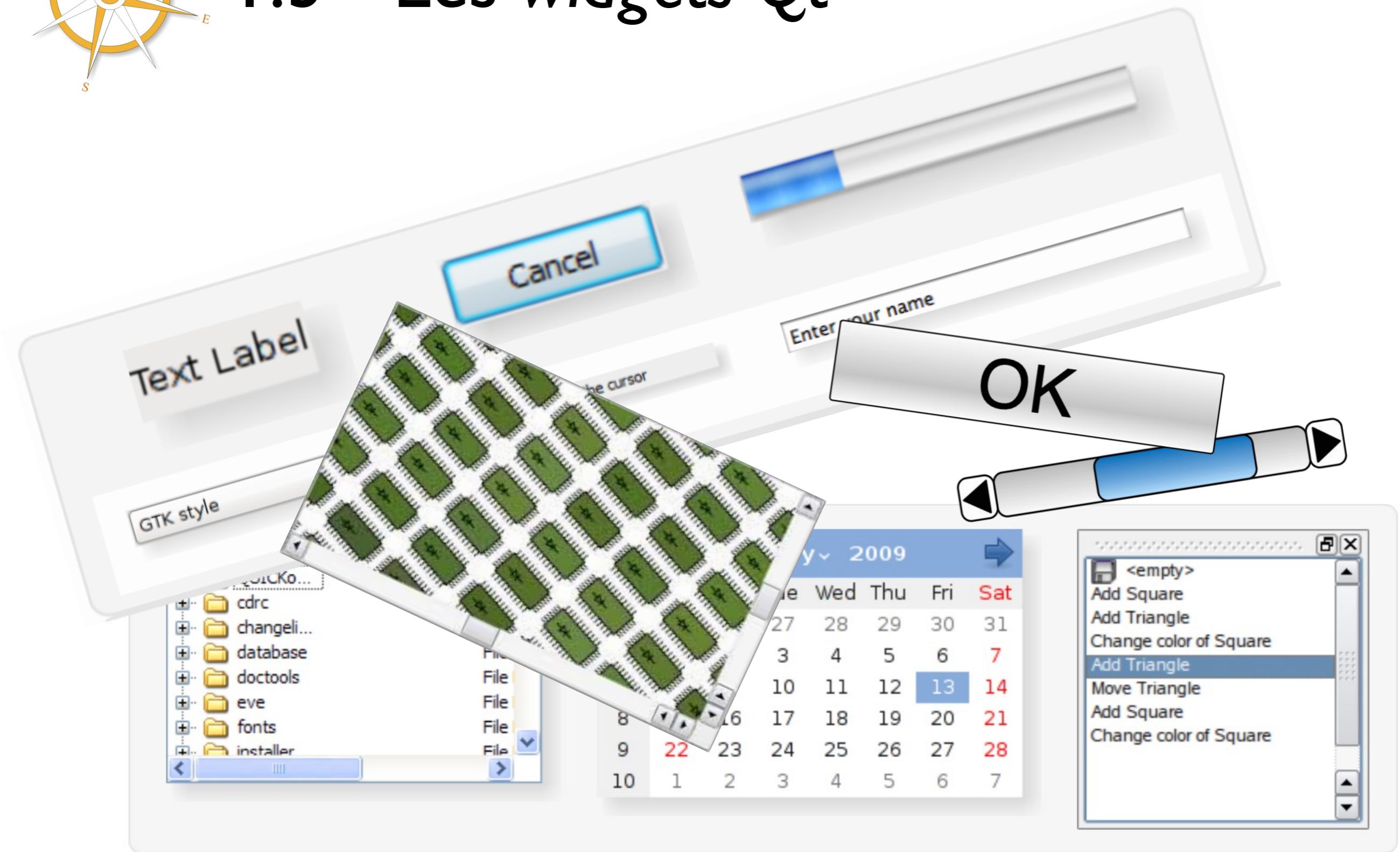
- obligatoirement paramétrable, programmable !



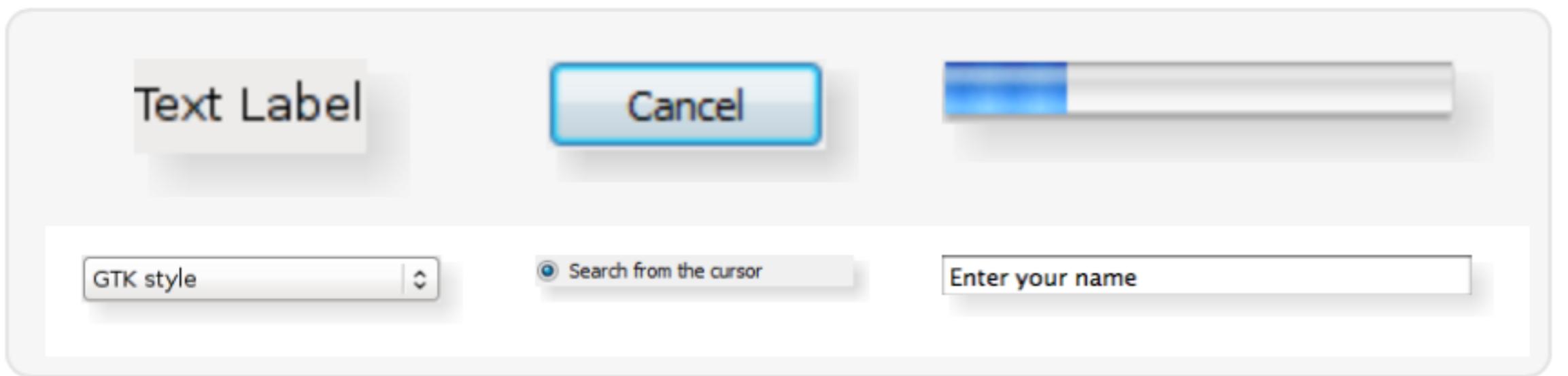
**Vous en connaissez des widgets ?**



# I.3 - Les widgets Qt

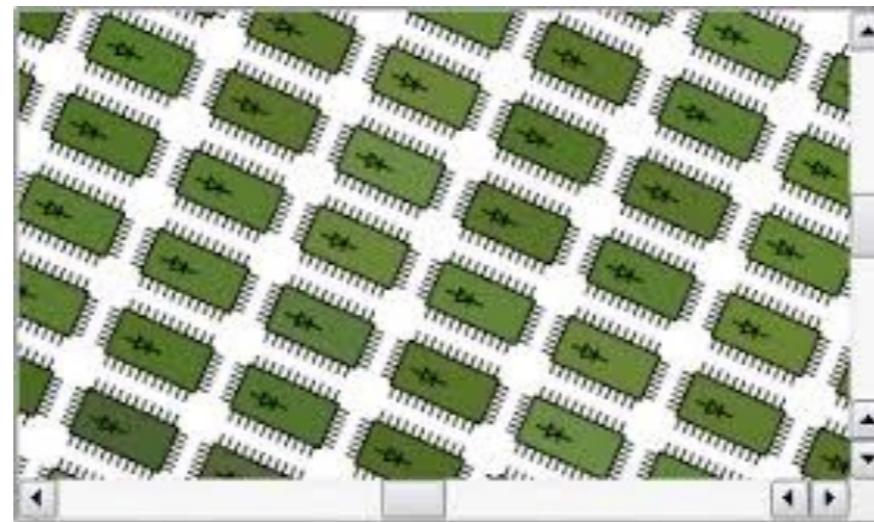


# Widgets basiques : contrôles



Bouton, zones de texte, barre de progression, barre de défilement, slider, combo box...

# Containers

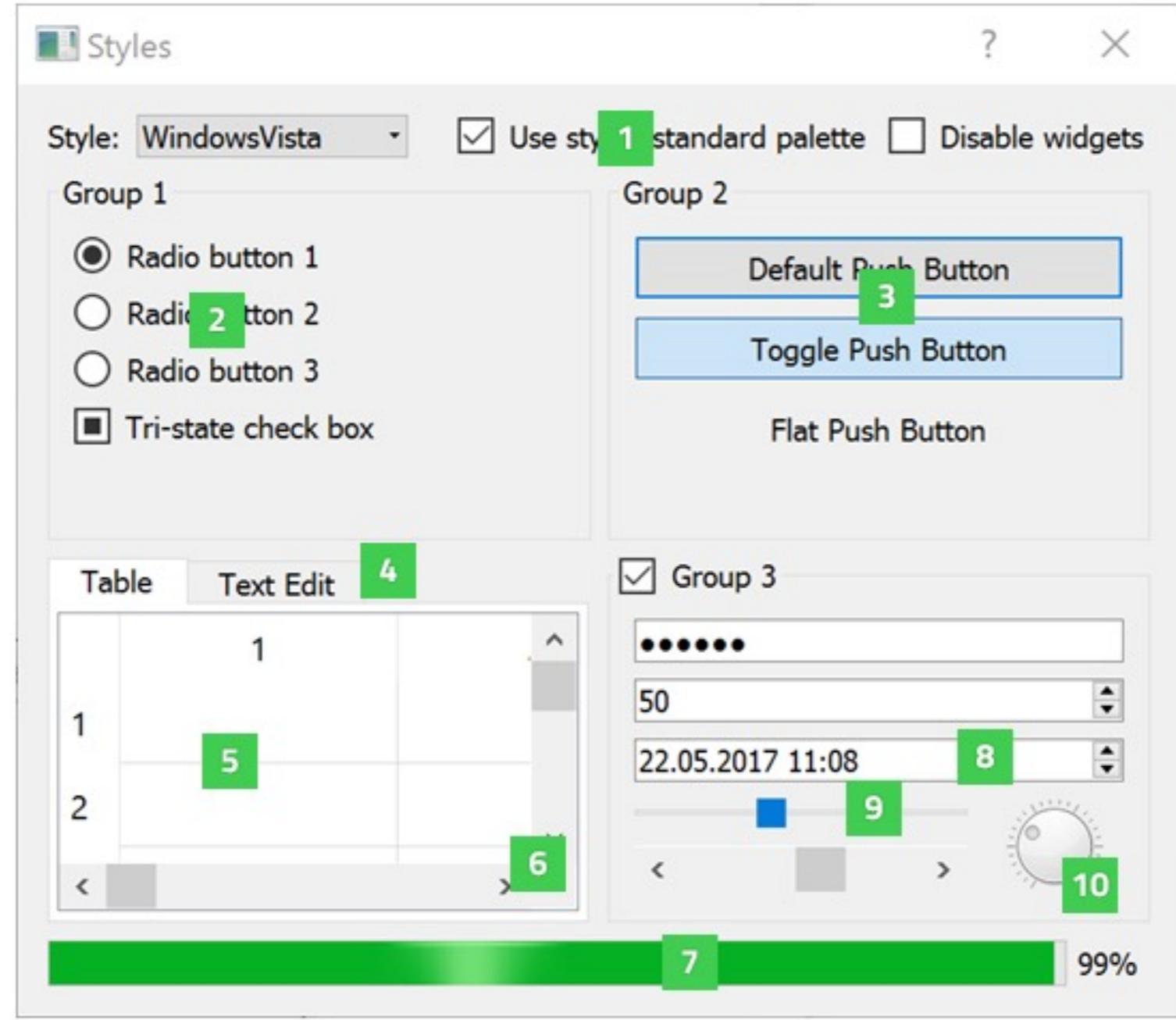


- Un *container* est un widget qui contient des enfants : fenêtre, frame, zone scrollable...  
Un container dispose en général d'un *layout* (une contrainte sur le positionnement de ses enfants : ligne, colonne, grille...),
- Il y a aussi des containers spécifiques pour dessiner : zone de dessin, item classique (rectangles, traits), item fait maison.

# « Advanced » widgets

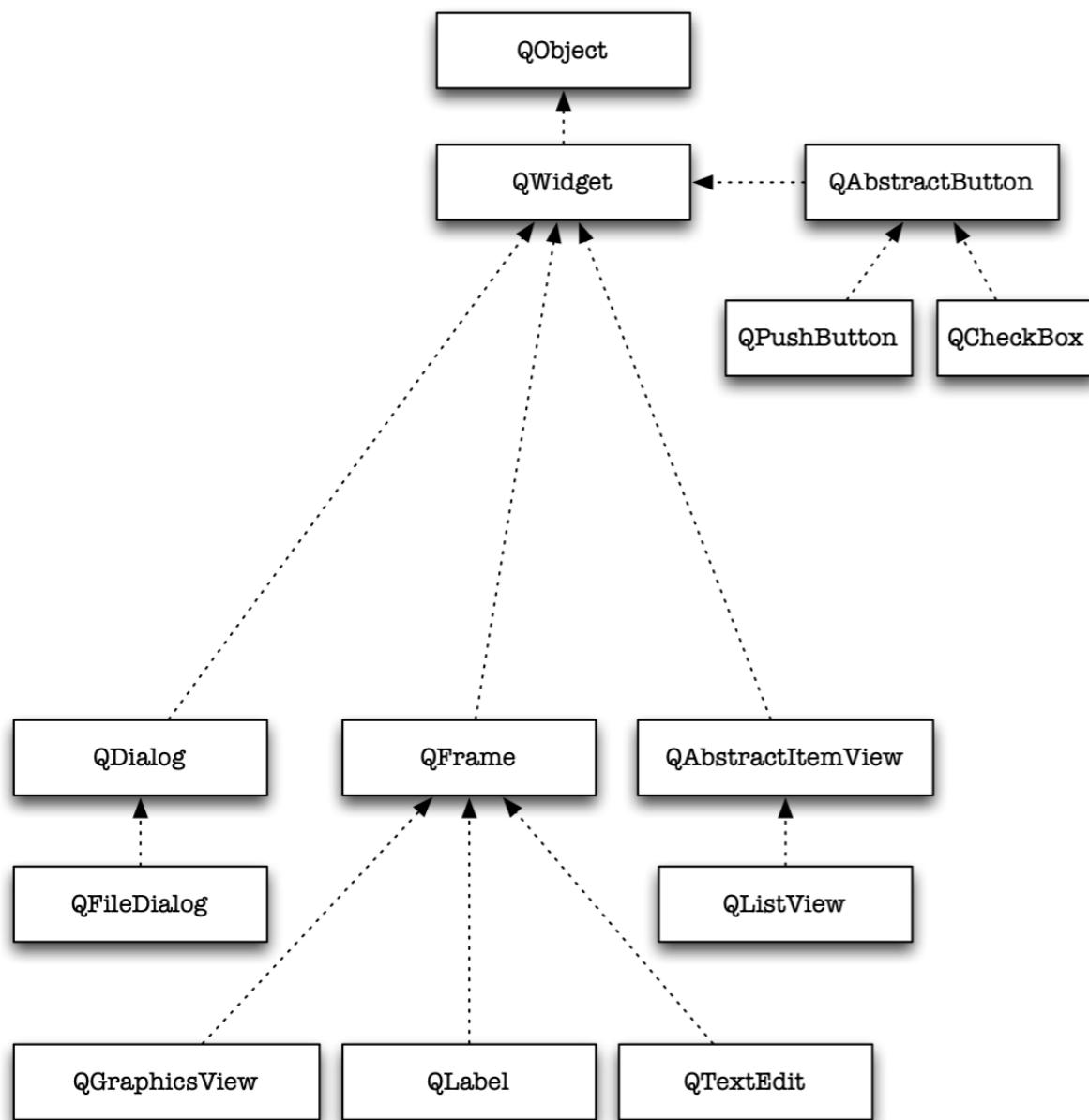


Liste, calendrier, arborescence...



- › `QCheckBox` (1) provides a checkbox with a text label.
- › `QRadioButton` (2) provides a radio button with a text or pixmap label.
- › `QPushButton` (3) provides a command button.
- › `QTabWidget` (4) provides a stack of tabbed widgets.
- › `QTableWidget` (5) provides a classic item-based table view.
- › `QScrollBar` (6) provides a vertical or horizontal scroll bar.
- › `QProgressBar` (7) provides a horizontal progress bar.
- › `QDateTimeEdit` (8) provides a widget for editing dates and times.
- › `QSlider` (9) provides a vertical or horizontal slider.
- › `QDial` (10) provides a rounded range control (like a speedometer or potentiometer).

# Arborescence de classe des widgets *Qt*



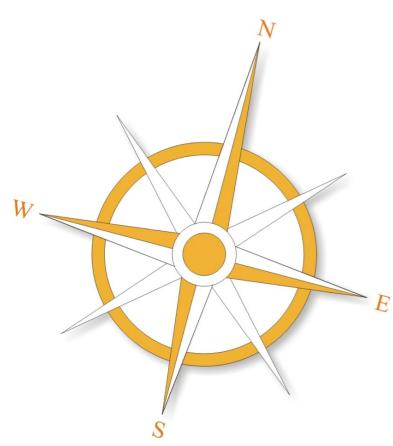
Hiérarchie de classes :  
Tous les widgets *Qt* sont des  
(ou héritent de) **QWidget**

<https://doc.qt.io/qt-5/gallery.html>

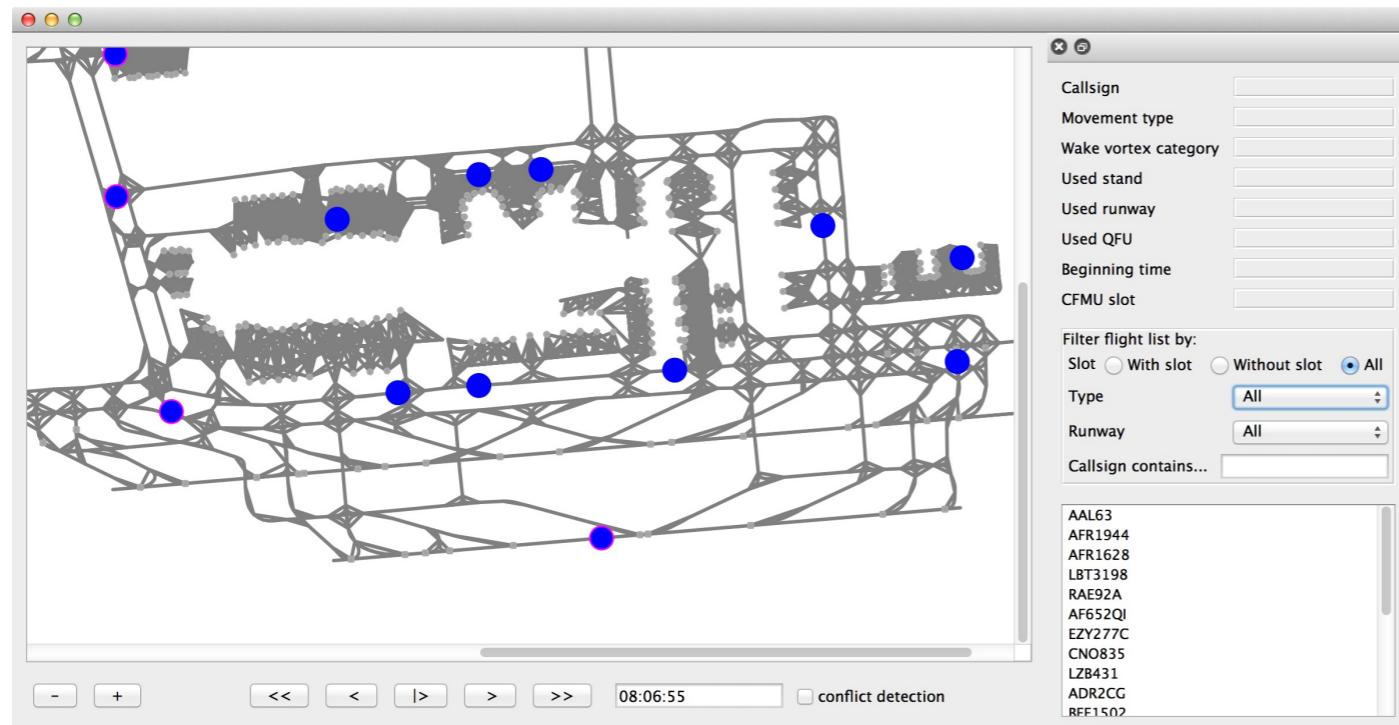


32

et il y en a beaucoup ...



# I.4 - Application WIMP : assemblage de widgets



*Window  
Icon  
Menu  
Pointing Device*

Assemblage de widgets imbriqués les uns dans les autres  
à la main, en programmant, ou bien en utilisant un  
builder d'interface (cf partie 3)

# Comment ajouter un widget ?

- Créer une instance du widget, par ex :

```
my_widget = QPushButton("blabla")
```

- Si l'application ne comporte que ce widget, il suffit de le montrer (méthode show) pour qu'il soit intégré à la fenêtre de l'application :

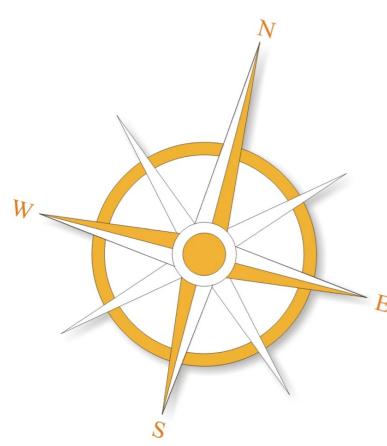
```
my_widget.show()
```

- Si le widget doit être intégré à un parent, il faut l'ajouter au layout de celui-ci (méthode addWidget), par ex :

```
the_layout = QVBoxLayout()
the_parent = QWidget()
the_parent.setLayout(the_layout)
the_layout.addWidget(my_widget)
```

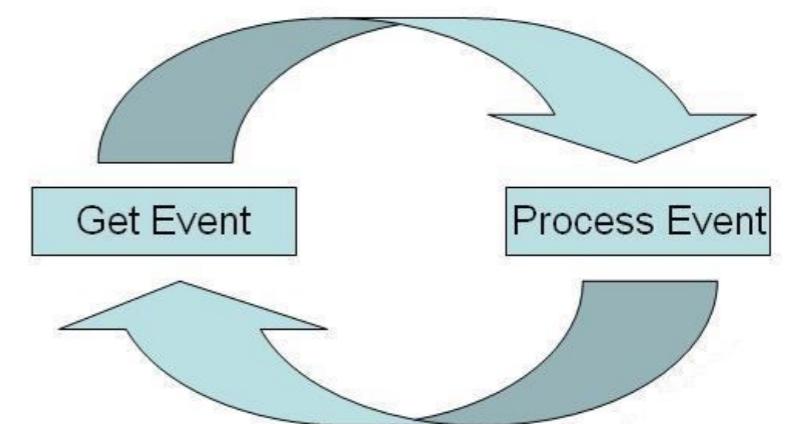
## 2 - Flot d'exécution d'un programme interactif

- Fonctionnement d'un programme non interactif vs interactif
- Conséquence sur la programmation des interactions en Qt



## 2. I - Flot d'exécution

- Dialogue conversationnel à l'initiative de la machine : contrôle par la machine, l'utilisateur réagit
- Système interactif : contrôle par l'utilisateur, la **machine réagit** aux actions de l'utilisateur :  
Le programme **attend** des **événements** de la part des utilisateurs, du système, du réseau...



# Flot d'exécution d'un programme **non**-interactif

ordre des instructions...

... dans le fichier .py

... à l'exécution

```
x=5+3  
print(x)  
print('au revoir')
```

**x=5+3**

**print(x)**

**print('au revoir')**

**Flot de programmation**

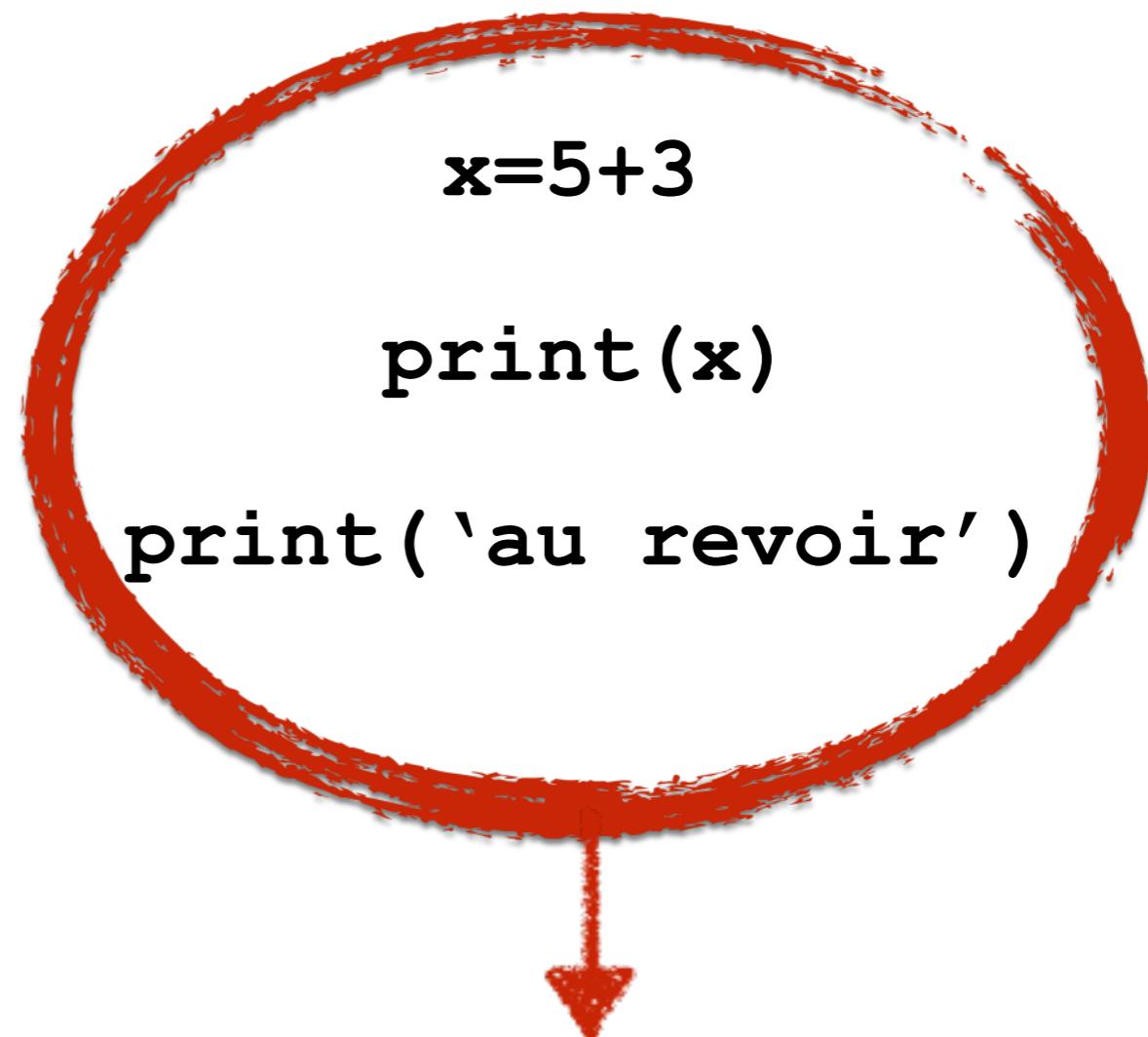
# Flot d'exécution d'un programme **non**-interactif

ordre des instructions...

... dans le fichier .py

... à l'exécution

```
x=5+3  
print(x)  
print('au revoir')
```



**Flot d'exécution**

# Flot d'exécution d'un programme **non**-interactif

Flot de programmation

```
x=5+3  
print(x)  
print('au revoir')
```

Flot d'exécution

**x=5+3**

**print(x)**

**print('au revoir')**

Ici c'est facile :

Le flot d'exécution est **le même** que l'ordre des instructions dans le fichier.

Il est donc totalement prévisible.

# Flot d'exécution d'un programme **non**-interactif

Flot de programmation

```
def maFonction2():
    print('coucou')

def maFonction1():
    maFonction2()

maFonction1()
```

Flot d'exécution

- 1) maFonction2 puis  
maFonction1 puis print
- 2) maFonction2 puis print  
puis maFonction1
- 3) maFonction1 puis  
maFonction2 puis print
- 4) maFonction1 puis print  
puis maFonction2



# Flot d'exécution d'un programme **non**-interactif

Flot de programmation

```
def maFonction2():
    print('coucou')
```

```
def maFonction1():
    maFonction2()
```

```
maFonction1()
```

Flot d'exécution

maFonction1

maFonction2

coucou



# Flot d'exécution d'un programme non-interactif

# Flot de programmation

```
def maFonction2():
    print('coucou')

def maFonction1():
    maFonction2()

maFonction1()
```

## Flot d'exécution

• maFonction1

maFonction2

coucou

Un peu plus difficile « à suivre » :

le **flux d'exécution** n'est **pas** le même que l'ordre des instructions dans le fichier...

Mais le flot d'exécution est toujours prévisible.

# Dialogue conversationnel à initiative machine

Flot de programmation

```
def myinput():
    x = input('saisie:')
    return x != 'q'

while myinput():
    print('merci')

print ('fini')
```

Flot d'exécution



# Dialogue conversationnel à initiative machine

Flot de programmation

```
def myinput():
    x = input('saisie:')
    return x != 'q'

while myinput():
    print('merci')

print ('fini')
```

Flot d'exécution

myinput  
saisie:0  
merci

myinput  
saisie:a  
merci

myinput  
saisie:q  
fini

# Dialogue conversationnel à initiative machine

## Flot de programmation

```
def myinput():
    x = input('saisie:')
    return x != 'q'

while myinput():
    print('merci')

print ('fini')
```

## Flot d'exécution

```
myinput
saisie:0
merci

myinput
saisie:a
merci

myinput
saisie:q
fini
```

Le flot d'exécution n'est **pas** le même que l'ordre des instructions dans le fichier.

Il n'est plus prévisible.

Description proche d'un programme non interactif, mais interaction peu confortable pour l'utilisateur (la machine a le contrôle)

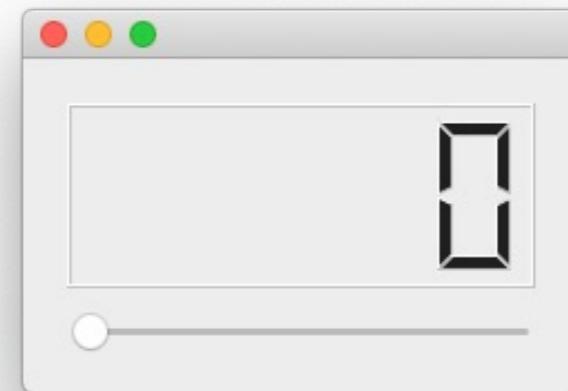


# Flot de description d'un programme interactif

```
import sys
from PyQt5.QtWidgets import QApplication, QLCDNumber, QSlider, QVBoxLayout, QWidget
from PyQt5.QtCore import Qt

def changed(event):
    print("1")

    print("2")
app = QApplication(sys.argv)
container = QWidget()
lcd = QLCDNumber()
sld = QSlider(Qt.Horizontal)
vbox = QVBoxLayout()
vbox.addWidget(lcd)
vbox.addWidget(sld)
container.setLayout(vbox)
sld.valueChanged.connect(lcd.display)
sld.valueChanged.connect(changed)
container.setGeometry(300, 300, 250, 150)
container.show()
print("3")
app.exec()
print("4")
```



cours1.py

**RUN!**



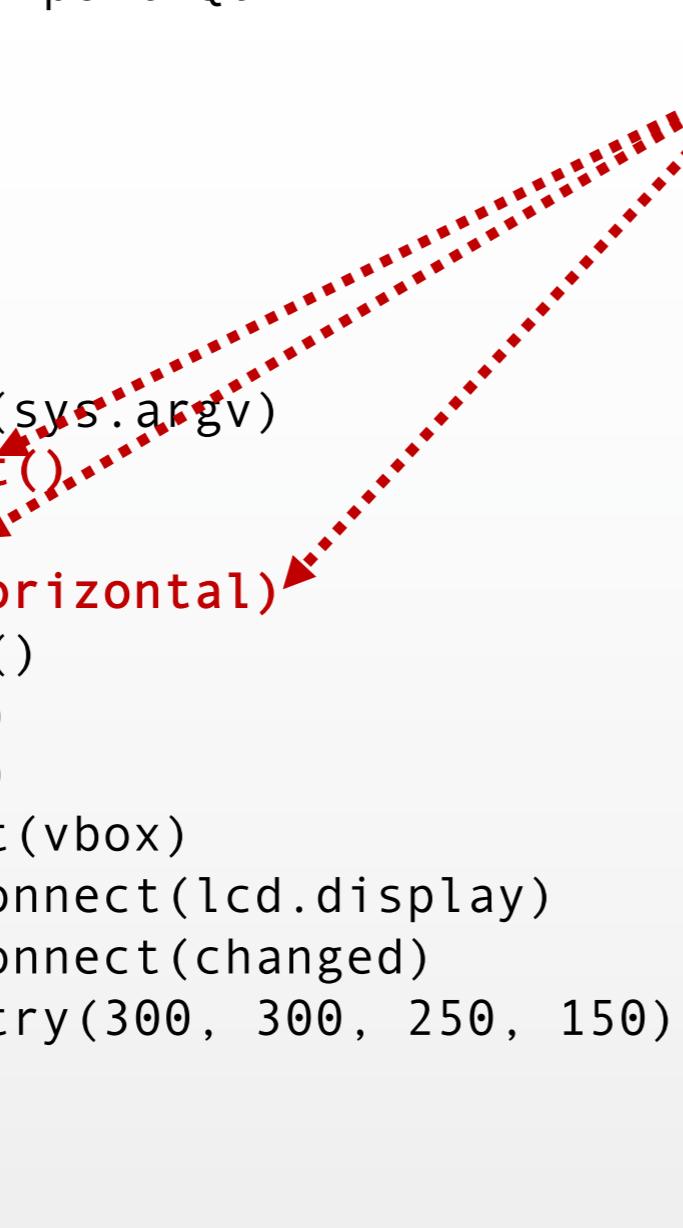
# Flot de description d'un programme interactif

```
import sys
from PyQt5.QtWidgets import QApplication, QLCDNumber, QSlider, QVBoxLayout, QWidget
from PyQt5.QtCore import Qt

def changed(event):
    print("1")

    print("2")
    app = QApplication(sys.argv)
container = QWidget()
lcd = QLCDNumber()
sld = QSlider(Qt.Horizontal)
vbox = QVBoxLayout()
vbox.addWidget(lcd)
vbox.addWidget(sld)
container.setLayout(vbox)
sld.valueChanged.connect(lcd.display)
sld.valueChanged.connect(changed)
container.setGeometry(300, 300, 250, 150)
container.show()
print("3")
app.exec()
print("4")
```

**print "2"**  
**crée plein de widgets**





# Flot de description d'un programme interactif

```
import sys
from PyQt5.QtWidgets import QApplication, QLCDNumber, QSlider, QVBoxLayout, QWidget
from PyQt5.QtCore import Qt

def changed(event):
    print("1")

    print("2")
    app = QApplication(sys.argv)
    container = QWidget()
    lcd = QLCDNumber()
    sld = QSlider(Qt.Horizontal)
    vbox = QVBoxLayout()
    vbox.addWidget(lcd)
    vbox.addWidget(sld)
    container.setLayout(vbox)
    sld.valueChanged.connect(lcd.display)
    sld.valueChanged.connect(changed)
    container.setGeometry(300, 300, 250, 150)
    container.show()
    print("3")
    app.exec()
    print("4")
```

**print "2"**  
**crée plein de widgets**  
**prépare les abonnements**



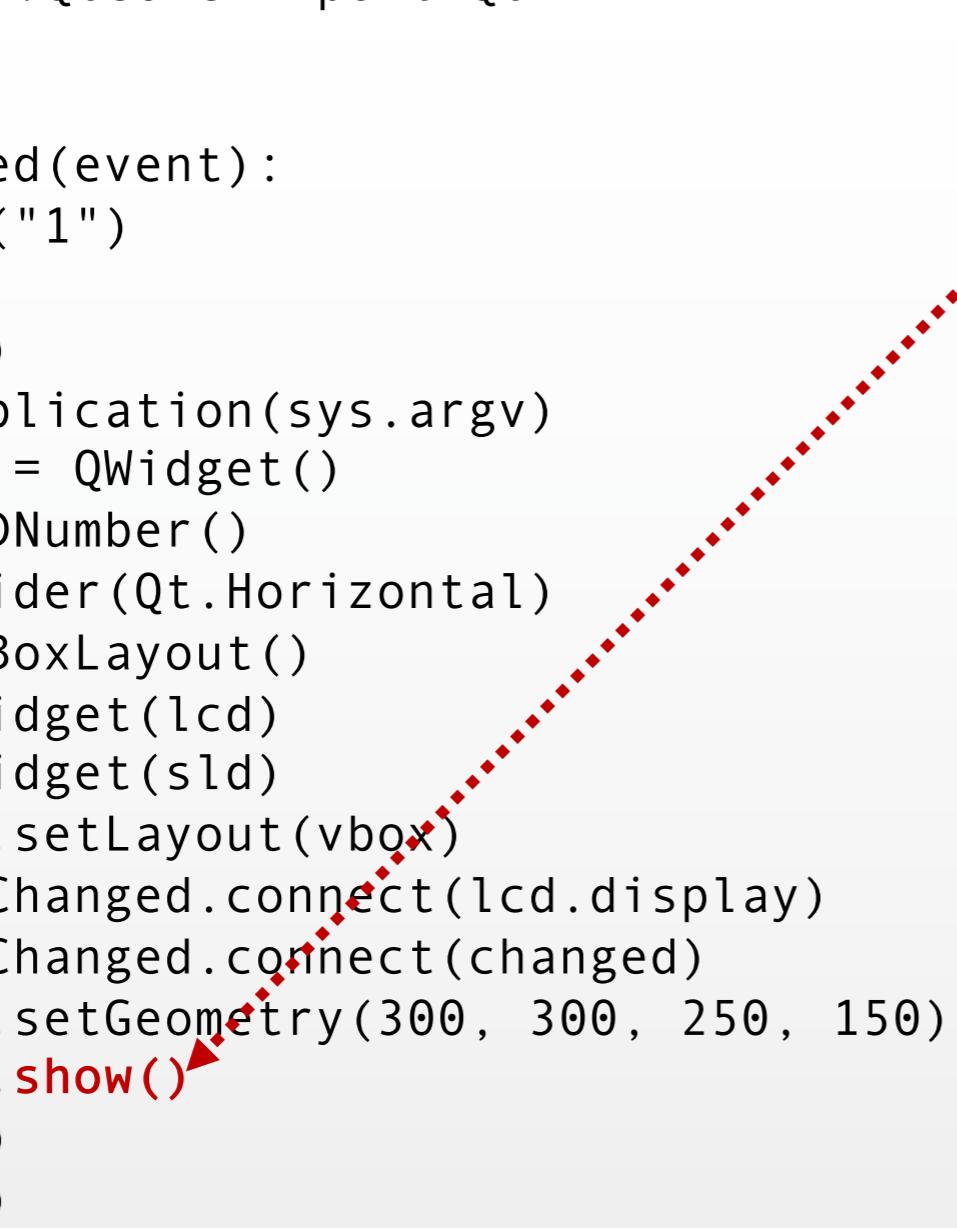
# Flot de description d'un programme interactif

```
import sys
from PyQt5.QtWidgets import QApplication, QLCDNumber, QSlider, QVBoxLayout, QWidget
from PyQt5.QtCore import Qt

def changed(event):
    print("1")

    print("2")
    app = QApplication(sys.argv)
    container = QWidget()
    lcd = QLCDNumber()
    sld = QSlider(Qt.Horizontal)
    vbox = QVBoxLayout()
    vbox.addWidget(lcd)
    vbox.addWidget(sld)
    container.setLayout(vbox)
    sld.valueChanged.connect(lcd.display)
    sld.valueChanged.connect(changed)
    container.setGeometry(300, 300, 250, 150)
container.show()
    print("3")
    app.exec()
    print("4")
```

**print "2"**  
**crée plein de widgets**  
**prépare les abonnements**  
**montre la fenêtre**





# Flot de description d'un programme interactif

```
import sys
from PyQt5.QtWidgets import QApplication, QLCDNumber, QSlider, QVBoxLayout, QWidget
from PyQt5.QtCore import Qt

def changed(event):
    print("1")

    print("2")
    app = QApplication(sys.argv)
    container = QWidget()
    lcd = QLCDNumber()
    sld = QSlider(Qt.Horizontal)
    vbox = QVBoxLayout()
    vbox.addWidget(lcd)
    vbox.addWidget(sld)
    container.setLayout(vbox)
    sld.valueChanged.connect(lcd.display)
    sld.valueChanged.connect(changed)
    container.setGeometry(300, 300, 250, 150)
    container.show()
    print("3")
app.exec()
    print("4")
```

**print "2"**  
**crée plein de widgets**  
**prépare les abonnements**  
**montre la fenêtre**

**print "3"**  
**mainloop : app.exec()**  
**réagit aux actions**  
**comportement standard des widgets**  
**+ slots connectés (fonction changed)**



# Flot de description d'un programme interactif

```
import sys
from PyQt5.QtWidgets import QApplication, QLCDNumber, QSlider, QVBoxLayout, QWidget
from PyQt5.QtCore import Qt

def changed(event):
    print("1")

    print("2")
    app = QApplication(sys.argv)
    container = QWidget()
    lcd = QLCDNumber()
    sld = QSlider(Qt.Horizontal)
    vbox = QVBoxLayout()
    vbox.addWidget(lcd)
    vbox.addWidget(sld)
    container.setLayout(vbox)
    sld.valueChanged.connect(lcd.display)
    sld.valueChanged.connect(changed)
    container.setGeometry(300, 300, 250, 150)
    container.show()
    print("3")
    app.exec()
print("4")
```

**print "2"**  
**crée plein de widgets**  
**prépare les abonnements**  
**montre la fenêtre**

**print "3"**  
**mainloop : app.exec()**  
**réagit aux actions**  
**comportement standard des widgets**  
**+ slots connectés (fonction changed)**

**print "4"**  
**quand on ferme la fenêtre**





# Flot de **description** d'un programme interactif

```
import sys
from PyQt5.QtWidgets import QApplication, QLCDNumber, QSlider, QVBoxLayout, QWidget
from PyQt5.QtCore import Qt

def changed(event):
    print("1")

    print("2")
    app = QApplication(sys.argv)
    container = QWidget()
    lcd = QLCDNumber()
    sld = QSlider(Qt.Horizontal)
    vbox = QVBoxLayout()
    vbox.addWidget(lcd)
    vbox.addWidget(sld)
    container.setLayout(vbox)
    sld.valueChanged.connect(lcd.display)
    sld.valueChanged.connect(changed)
    container.setGeometry(300, 300, 250, 150)
    container.show()
    print("3")
    app.exec()
    print("4")
```

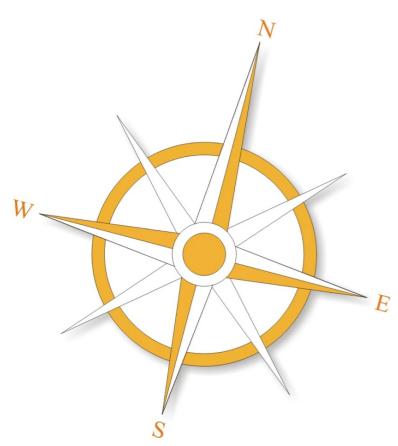
**print "2"**  
**crée plein de widgets**  
**prépare les abonnements**  
**montre la fenêtre**

**print "3"**  
**mainloop : app.exec()**  
**réagit aux actions**  
**comportement standard des widgets**  
**+ slots connectés (fonction changed)**

**print "4"**  
**quand on ferme la fenêtre**

Le flot d'exécution n'est **pas le même** que l'ordre des instructions dans le fichier.

Le flot d'exécution est **prévisible** entre le point d'entrée et mainloop, mais **dépend** des actions de l'utilisateur après...



## 2.2 - Programmer des interactions



# Programmer une interaction

Programmer une interaction consiste à *décrire la réaction de l'application à un événement* donné.

L'*événement* peut être une action de l'utilisateur (clic souris, touche enfoncée...) ou un message depuis un autre composant logiciel (timer, réseau...).

Un composant logiciel peut *s'abonner à un événement*, ce qui lui permettra de réagir chaque fois que l'événement surviendra.



# Le *signal*

Pour programmer les interactions au niveau des widgets, Qt fournit le système *signal/slot* :

- Un *signal* est un objet permettant de notifier un événement donné. Il convoie parfois des informations supplémentaires (*dimension* du signal).
- Chaque widget est capable d'émettre un certain nombre de signaux.

Ex :

Un QSlider émet (entre autres) le signal `valueChanged` pour signaler que sa valeur a été modifiée (par une action de l'utilisateur ou une simple ligne de code à un moment de l'exécution).

Le signal `valueChanged` convoie avec lui la nouvelle valeur du slider.

# Le *slot* et la connexion *signal/slot*

- Pour pouvoir réagir à un signal précis émis par un widget donné, il faut s'y abonner en « connectant » le signal à un *slot* (une fonction qui décrit la réaction souhaitée), grâce à la méthode `connect`
- Lorsque le widget émet le signal, le slot est alors automatiquement appelé et exécute donc la réaction qui y est décrite.

cours1.py

```
def changed(event):  
    print("1")  
  
sld.valueChanged.connect(changed)  
sld.valueChanged.connect(lcd.display)
```

Slot

Abonnement (connexion)

Le slot peut aussi être une méthode d'un autre widget :

Ici celle qui permet de modifier l'affichage du QLCDNumber, qui affiche du coup l'information convoyée par le signal `valueChanged` (la nouvelle valeur). On a lié les deux widgets.

# Fonction vs. appel de fonction

Attention : le paramètre de la méthode connect est une **fonction**, pas un **appel de fonction**...

On ne peut donc pas passer de paramètre explicitement de cette manière.

```
[...]
def func():
    print("toto")

button = QPushButton("Call func")
button.clicked.connect(func())
[...]                                         button.clicked.connect(func)
```

# Fonction *lambda*

Si on veut passer des paramètres il faut utiliser une lambda :

```
[...]
def func(param):
    print(param)

button = QPushButton("Call func")

my_param = "toto"
button.clicked.connect(lambda:func(my_param))
[...]
```



# QCM

```
[...]
def func(value):
    print ("func has been called with value:", value )
[...]

slider = QSlider()
slider.valueChanged.connect(???)
```

**Que ne peut-on PAS mettre à la place de ???**

- 1) func
- 2) func(slider.value())
- 3) lambda v: print ("func has been called with value ", v)
- 4) print



```
import sys
from PyQt5.QtWidgets import QApplication, QPushButton

def affiche(texte):
    print('hello %s' % texte)

appli = QApplication(sys.argv)
button = QPushButton("press me")
button.clicked.connect(lambda: affiche("toto"))
button.show()
appli.exec()
print("fin")
```

cours2.py

## Création du widget button

**Abonnement** du bouton au signal “clicked”, sur le slot (affiche), avec comme paramètre un pointeur vers la chaîne de caractères “toto”.

## Qu'est-ce qui se passe quand on clique sur le bouton ?

- 1) affiche "press me"
- 2) affiche "hello toto"
- 3) affiche le bouton
- 4) ferme la fenêtre



```
import sys
from PyQt5.QtWidgets import QApplication, QPushButton

def affiche(texte):
    print('hello %s' % texte)

appli = QApplication(sys.argv)
button = QPushButton("press me")
button.clicked.connect(lambda: affiche("toto"))
button.show()
appli.exec()
print("fin")
```

cours2.py

## Création du widget button

**Abonnement** du bouton au signal “clicked”, sur le slot (affiche), avec comme paramètre un pointeur vers la chaîne de caractères “toto”.

## Qu'est-ce qui se passe quand on ferme la fenêtre ?

- 1) affiche "press me"
- 2) affiche "hello toto"
- 3) ne fait rien
- 4) affiche "fin"



```
class Film:  
    def __init__(self, titre, version):  
        self.titre = titre  
        self.version = version  
  
films = [Film('Fast and Furious', 9), Film('xXx', 3)]  
  
app = QApplication(sys.argv)  
  
container = QWidget()  
box = QHBoxLayout()  
container.setLayout(box)  
  
label_annonce = QLabel('Vin Diesel joue dans')  
box.addWidget(label_annonce)  
  
label_titre = QLabel(films[0].titre + " " + str(films[0].version))  
box.addWidget(label_titre)  
  
button = QPushButton("Et c'est tout ?")  
box.addWidget(button)  
  
container.show()  
app.exec()
```

**À chaque clic, incrémenter le numéro de version du film et l'afficher...**

- visibilité des variables ?
- localité ?

# Comment réagir à un signal ?

- Chaque widget est capable d'émettre un certain nombre de signaux (ex : valueChanged pour un QSlider)
- Pour pouvoir réagir à un signal précis émis par un widget donné, il faut s'y abonner grâce à la méthode connect :

```
my_widget.SIGNAL.connect(SLOT)
```

où SIGNAL est un signal que le widget peut émettre,  
et SLOT est une fonction (pas le résultat d'un appel de fonction ; on peut utiliser lambda : si on souhaite passer des paramètres).

Lorsque le widget émet le signal, le slot est alors automatiquement appelé avec comme paramètre la dimension du signal (la nouvelle valeur dans le cas du slider) ou des paramètres additionnels (si lambda :)

# Cycle de vie simplifié d'un widget

- **Créé** (allocation de ressources), et **paramétré** via l'API du widget  
`my_widget=QPushButton("titi")`
- **paramétré** via des appels de méthodes sur l'objet :  
`my_widget.setText("tutu")`
- **parenté**: ajouté à un container, pour contrôler l'imbrication, la géométrie
- **montré** ou **caché**: il est montré par défaut si son parent l'est:  
`my_widget.show()`, `my_widget.hide()`
- **abonné** à un ou plusieurs signaux  
`sld.valueChanged.connect(lcd.display)` pour définir le comportement de l'application
- **réagit** aux événements utilisateur au sein de la boucle principale `app.exec()` (fonctionnement par défaut du widget et *signaux/slots*, tout est géré par l'objet `QApplication`)
- **détruit**: (explicitement ou) automatiquement par le *garbage collector*  
`my_widget.destroy()`

# TPI

## **Exercice I.I : Flot d'exécution**



# TPI

**Exercice 1.2 : Flot d'exécution**  
Questions diapos suivantes...



# Exo I.2

Que se passera-t-il si on modifie le programme comme suit :

en rajoutant dans la fonction ligne 7 la séquence d'instruction de boucle infinie  
while True: pass

Et maintenant testez !

# Exo I.2

Que se passera-t-il si on modifie le programme comme suit :

en commentant au moyen d'un dièse la ligne 15  
# button.show()

Et maintenant testez !

# Exo 1.2

Que se passera-t-il si on modifie le programme comme suit :

en commentant la ligne 17

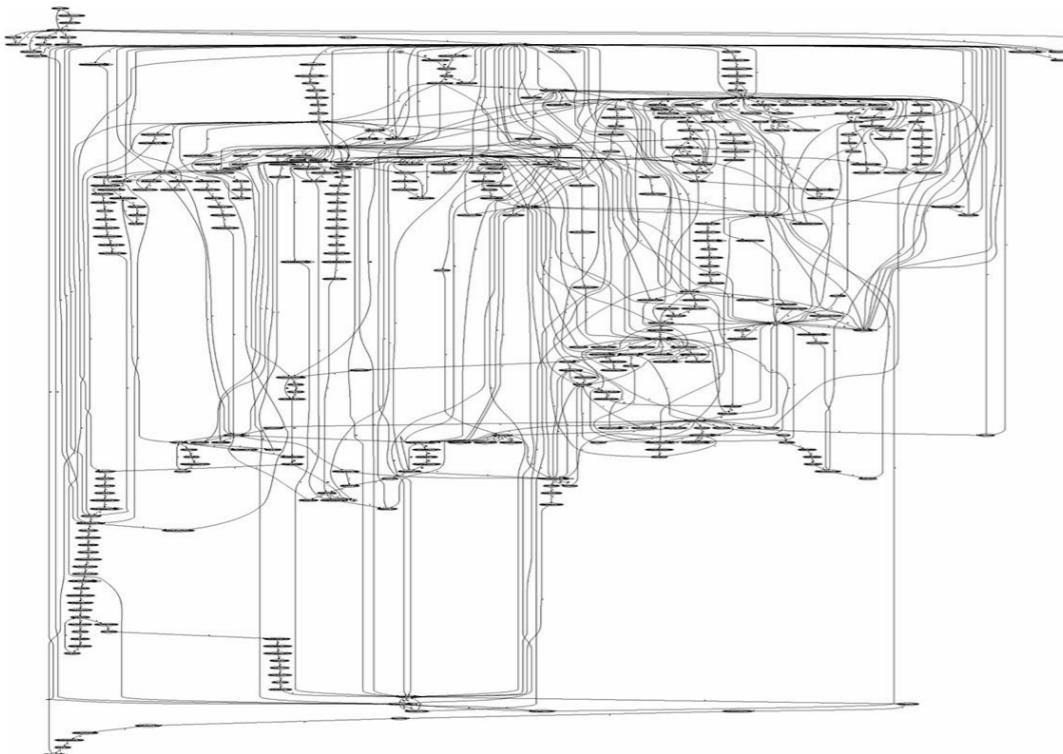
```
# app.exec()
```

Et maintenant testez !

# Pourquoi c'est si compliqué ? (1/2)

## Constat : **code spaghetti**

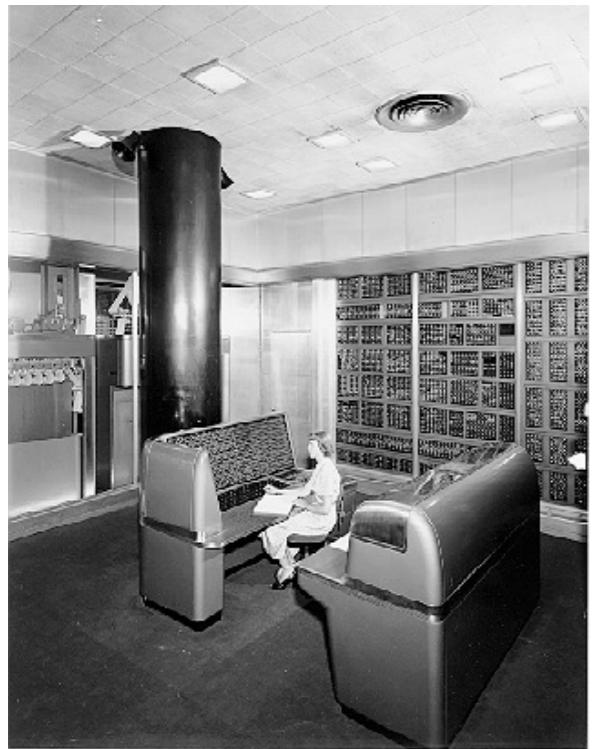
On définit le **comportement** (*print /*) quelque part dans le code, on fait des abonnements **aux actions utilisateur** ailleurs, la logique de contrôle du dialogue est répartie dans le source et dans les comportements par défaut des widgets.



In spaghetti code, the relations between the pieces of code are so tangled that it is nearly impossible to add or change something without unpredictably breaking something somewhere else.

# Pourquoi c'est si compliqué ? (2/2)

Les outils d'aujourd'hui héritent des modèles d'interaction et de programmation adaptés aux outils d'hier.



IBM SSEC, 1948

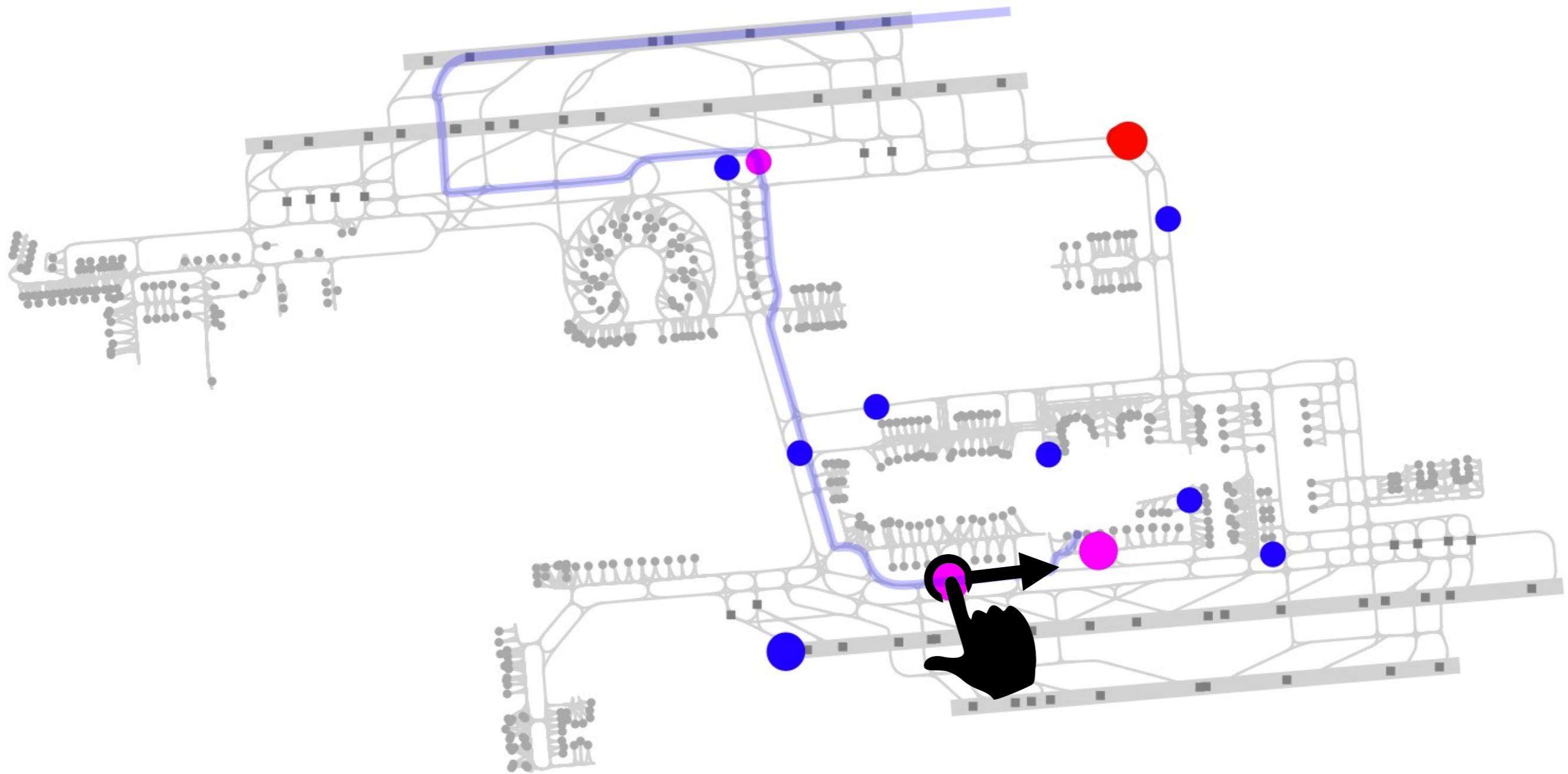
```
Terminal — bash — 80x24
afric:~/ens conversy$ cd ..
afric:~/ens conversy$ ls
Applications/ Public/ ens/ rsync-save.py@
Desktop/ Sites@ mail/ rsync-www.py@
Documents/ Temporary Items/ mindterm/
Library/ admin/ misc/ share/
Movies/ archives/ mnt/ src/
Music/ build/ public_html/ src-ext/
Pictures/ data/ recherche/ tmp/
afric:~/ens conversy$ ls -l rsync*
lrwxr-xr-x 1 root sconvers 18 Jan 11 2007 rsync-save.py@ -> misc/rsync-save
.py
lrwxr-xr-x 1 root sconvers 17 Jan 11 2007 rsync-www.py@ -> misc/rsync-www.p
y
lrwxr-xr-x 1 root sconvers 13 Jan 11 2007 rsync.py@ -> misc/rsync.py
afric:~/ens conversy$ cd ens
afric:~/ens conversy$ ls
COO/ IHM avance/ old/ tecvisu/
CP/ algo/ perception/ visu/
Collecticiels/ autres/ projet-etudiants/
IG/ intro-ihm/ proto/
afric:~/ens conversy$
```

interface à « ligne de commande »

Entrer le nom: Einstein  
Entrer le prénom: Albert  
Entrer l'âge: \_

paradigme d'interaction « dialogue conversationnel »

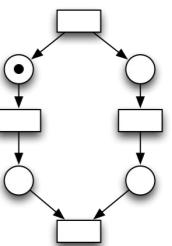
# Paradigme de *manipulation directe*



# Ça pourrait être plus simple !

Il existe d'autres approches...

- langages réactifs (esterel, *functional reactive programming*, valeurs actives), où on définit des dépendances dynamiques : `rect.x = zoom * mouse.x + pan.x`
- langages et notations à plusieurs dimensions (réseaux de Petri, *statecharts*, machines à états hiérarchiques)

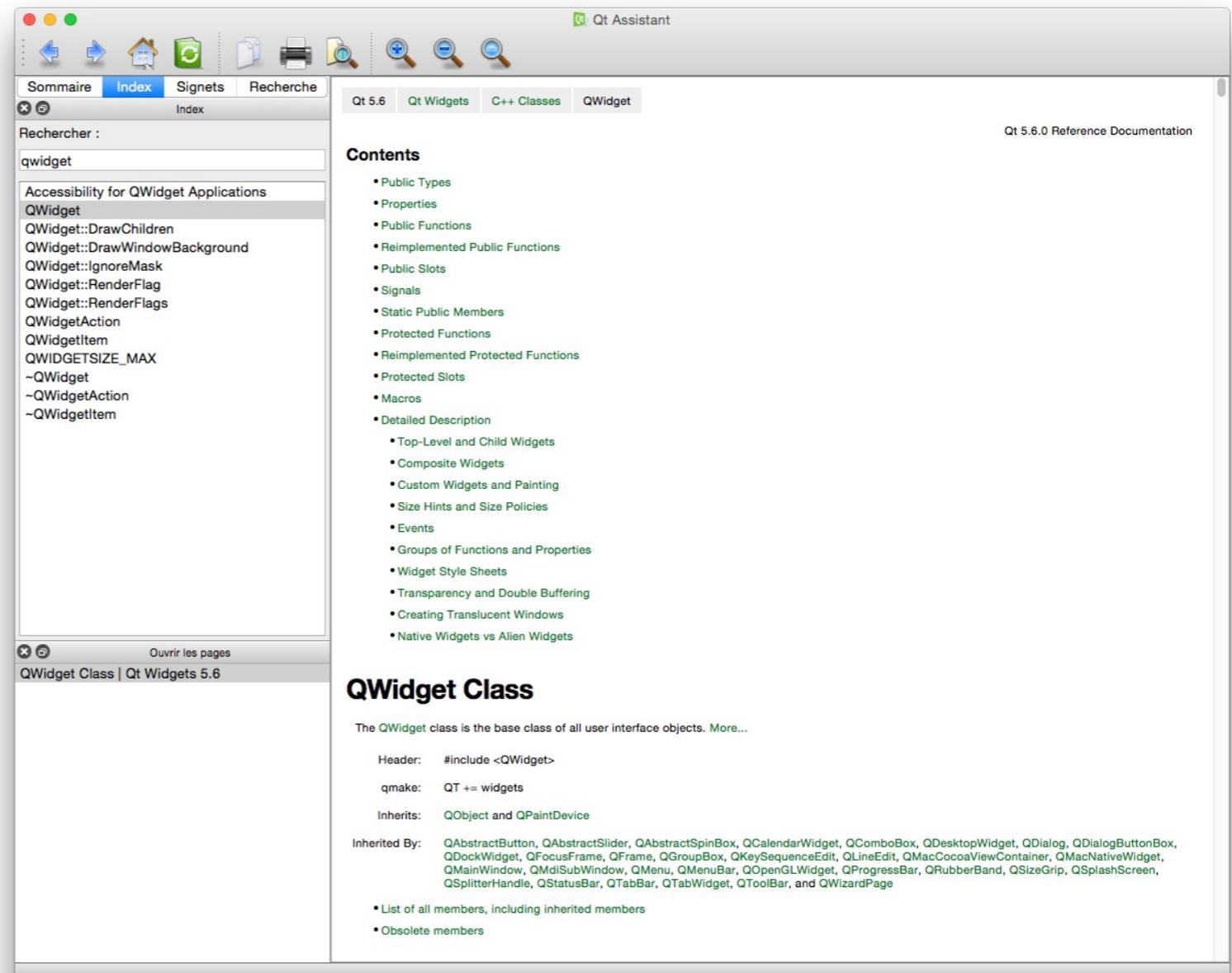


...mais leur diffusion est encore restreinte.

Heureusement, l'approche objet permet de maintenir une certaine localité des données et traitements.

# Documentation de l'API

*Qt Assistant (pour C++) :*  
index,  
recherche,  
navigation hypertexte



Doc Qt/C++ : <https://doc.qt.io/archives/qt-5.12/classes.html>

Doc Qt/PySide : <https://doc.qt.io/qtforpython-5/contents.html>

Doc Qt/PyQt : <https://www.riverbankcomputing.com/static/Docs/PyQt5/sip-classes.html>

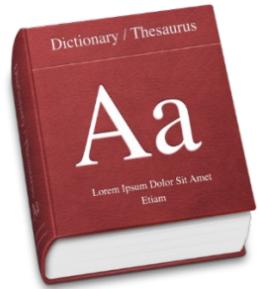
# TPI

**Exercice 2 :** Simulateur Airport  
Organisation des widgets (classes, objets...),  
interactions, ajouter un widget



# 3 - Programmation WIMP avec *Qt Designer*

- Utilisation d'un Graphical User Interface (**GUI**) Builder
- Notion de gestion de la géométrie, de *containers*,
- Gestion des **signaux** et **slots**

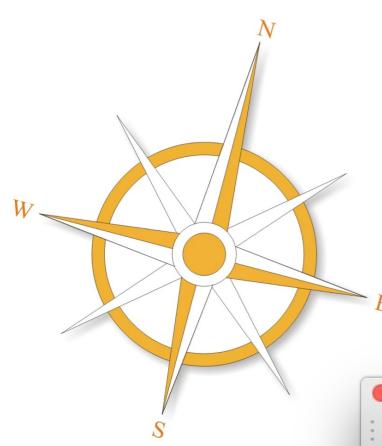


# Définition *GUI builder*

Un *GUI builder* est une application qui permet de construire une IHM en disposant visuellement les widgets. Il est souvent lié à un langage de programmation et une *GUI toolkit* particulière.

Avantages :

- vitesse de réalisation,
- apprentissage rapide des paramètres (reconnaissance plutôt que mémorisation, manipulation directe vs commandes complexes),



# 3. I - Qt Designer

The screenshot shows the Qt Designer interface with the following components:

- Toolbars:** Standard file operations (New, Open, Save) and layout tools (Vertical Layout, Horizontal Layout, Grid Layout, Form Layout, Spacers, Buttons, Item Views, Item Widgets, Containers, Input Widgets).
- Central View:** A window titled "Form - ui\_colorchooser.ui" containing three horizontal sliders labeled "Red", "Green", and "Blue". Each slider has a spin box to its right. To the right of the sliders is a black square preview area.
- Inspector d'objet (Object Inspector):** Lists the objects in the form:
  - colorchooser (QWidget)
  - verticalLayout\_2 (QVBoxLayout)
  - horizontalLayoutRed (QHBoxLayout)
  - labelRed (QLabel)
  - spinBoxRed (QSpinBox)
  - horizontalLayoutGreen (QHBoxLayout)
  - horizontalSliderGreen (QSlider)
  - labelGreen (QLabel)
  - spinBoxGreen (QSpinBox)
  - horizontalLayoutBlue (QHBoxLayout)
  - horizontalSliderBlue (QSlider)
  - labelBlue (QLabel)
- Éditeur de propriétés (Properties Editor):** Shows properties for the "colorchooser" object, including:
  - objectName: colorchooser
  - windowModality: NonModal
  - enabled: checked
  - geometry: [(0, 0), 506 x 124]
  - sizePolicy: [Fixed, Fixed, 0, 0]
    - Politique horizontale: Fixed
    - Politique verticale: Fixed
    - Éirement horizontal: 0
    - Éirement vertical: 0
  - minimumSize: 0 x 0
  - Largeur: 0
- Éditeur de signaux et slots (Signals and Slots Editor):** Shows connections between spin boxes and sliders:

Émetteur	Signal	Recepteur	Slot
spinBoxRed	valueChanged(int)	horizontalSliderRed	setValue(int)
spinBoxGreen	valueChanged(int)	horizontalSliderGreen	setValue(int)
spinBoxBlue	valueChanged(int)	horizontalSliderBlue	setValue(int)
horizontalSliderRed	valueChanged(int)	spinBoxRed	setValue(int)
horizontalSliderGreen	valueChanged(int)	spinBoxGreen	setValue(int)
horizontalSliderBlue	valueChanged(int)	spinBoxBlue	setValue(int)

# *Qt Designer*

*Qt Designer* est l'outil de l'API *Qt* pour construire des *IHM*. Il permet :

- d'assembler des widgets par *drag-and-drop*,
- de prévisualiser (Control+R) un widget sans avoir à écrire de code pour créer l'application (solution utilisable pour faire du prototypage de surface),
- mais aussi de programmer des interactions de base sans écrire de code non plus.

# TP2

**Exercice I : Construire un widget avec Qt Designer**



# Qt Designer : Comment générer du code python ?

A partir de la description du widget créé dans *Qt Designer* (fichier *.ui*), il faut générer du code utilisable dans le langage de notre choix depuis un outil en ligne de commande (fourni avec l'API *Qt* pour ce langage).

Pour *PyQt5* il s'agit de l'utilitaire en ligne de commande *pyuic5* :

```
pyuic5 -x nom_fichier.ui -o nom_fichier_python_généré.py
```

Options à retenir : -h, --help

- x
- o

Manuel *Qt Designer / PyQt5* <http://pyqt.sourceforge.net/Docs/PyQt5/designer.html>

Paramétrer VS Code pour *PyQt* <https://e-campus.enac.fr/moodle/mod/page/view.php?id=133058>

# Qt Designer : Comment utiliser le code généré ?

- Importer la classe générée dans l'espace de nom,
- créer une instance de la classe générée (`widget_QtDesigner`),
- créer une instance de QWidget (`widget_parent`) dans laquelle votre classe générée sera affichée,
- paramétriser le composant créé sous Qt Designer en appelant sa méthode `setupUi` :

```
widget_QtDesigner.setupUi(widget_parent)
```

- intégrer tout cela dans votre application, entre la création de la QApplication et la boucle principale (exemple de code à la fin du fichier généré avec l'option -x).

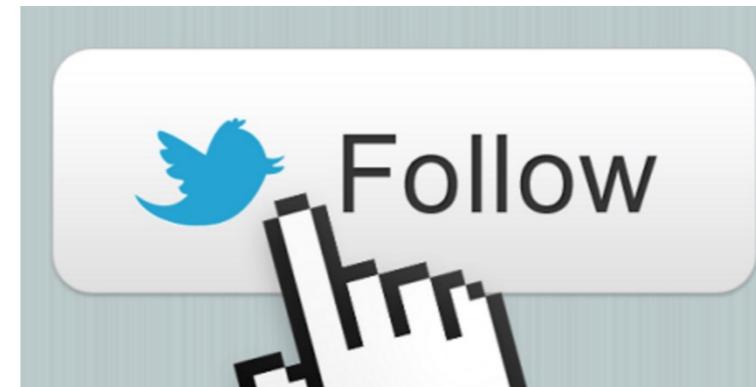
# TP2

**Exercice 2 :** Intégrer la classe générée à un programme



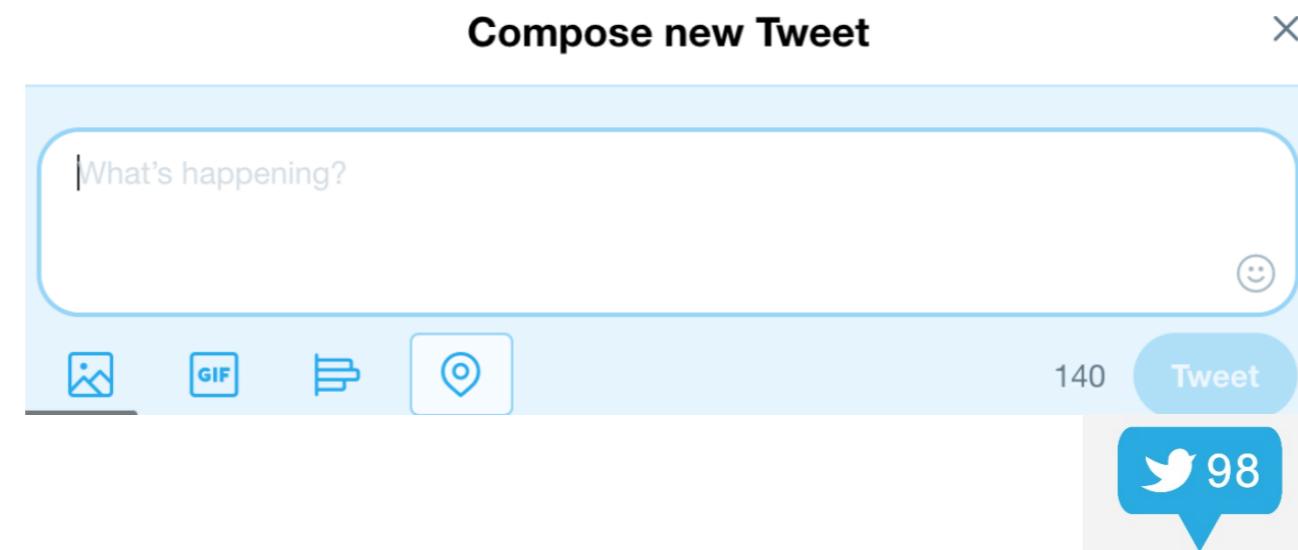
# La nécessité d'émettre un signal

N'importe qui peut s'abonner au signal



tweet.connect(my\_slot)

Au click le signal est émis, tous les abonnés le reçoivent



CLICK 

tweet.emit()

# Comment créer un signal custom ?

- Une classe héritant de QObject (donc n'importe quel QWidget) peut déclarer un signal (variable de classe, pas d'instance) :

```
my_signal = pyqtSignal(TYPE)
```

où **TYPE** est le type d'une information qu'on décide de passer à notre nouveau signal. Il peut être un type natif de python, un nom de classe, un dict, une list, ou rien du tout (pas d'info).

- Le signal peut ensuite être émis à volonté et convoyer toute info du **TYPE** déclaré à la création du signal :

```
my_signal.emit(my_info)
```

- Et on peut y réagir en s'abonnant de façon classique :

```
my_signal.connect(...)
```

# TP2

**Exercice 3 : Créer un signal custom**



# TP2

**Exercice 4 : Afficher les informations d'un vol**



# Difficultés d'intégration

- Bien nommer les objets correspondant aux différents widgets depuis *Qt Designer*, pour pouvoir y faire facilement référence depuis la méthode de mise à jour dans votre code :  
`my_inspector.label_callsign.setText(flight.callsign)`
- Couplage entre des objets très lointains : l'objet AircraftItem qui réagit à l'action utilisateur (`mousePressEvent`), et l'objet Inspector créé depuis le main qui doit y réagir aussi
- Éléments de solution :
  - faire en sorte que l'AircraftItem connaisse l'inspecteur (couplage fort, via la RadarView et main.py),
  - ou réduire le couplage en utilisant un signal custom qui permet de faire le lien depuis main.py...



# 3.2 - Conclusion GUI Builders

The screenshot shows the NetBeans IDE interface. The title bar indicates the project is "VAPS XT - [edit] - PFD\_XT\_Flightsim \*". The menu bar includes File, Edit, View, Operations, Arrange, Version Control, Settings, Run, State Chart, Code Gen, Window, Help. The toolbar has various icons for file operations. The Projects panel shows a "Source Packages" node containing "examples" with files "Antenna.java", "ContactEditor.java", and "Find.java", and a "Libraries" node with "Swing Layout Extensions - swing-layout-1.0.4.jar" and "JDK 1.6 (Default)". The Navigator panel lists components under "Form Antenna" such as JFrame, JPanel1, JPanel2, and several JLabels, JTextFields, and JButtons. The Inspector panel shows the current component selected. The main editor area displays the "Antenna.java" code in "Design" mode. A configuration dialog is open, titled "Position/Direction", with fields for "Direction [°]: 140.000", "Height [m]: 110.000", and a checked option "Height is Lower Edge (Not Center)". Below this is a "System" section with fields for "Channels: 2", "Watts: 12.000", "Antenna Type: Kathrein 742151", "Electrical Downtilt From [°]: 0.000", "To: 10.000", "Polarization: X +45°", and "Frequency From [MHz]: 943.000", "To: 951.000". The dialog has OK and Cancel buttons. To the right of the editor is a "Palette" window listing Swing Containers (Panel, Split Pane, Tool Bar, Internal Frame) and Swing Controls (Label, Toggle Button, Radio Button, Combo Box, Text Field, Scroll Bar, Progress Bar, Password Field) with their corresponding icons. The "Properties" panel for "Antenna.java" shows the file's properties: Name (Antenna), Extension (java), All Files (C:\Users\ruth\Docum...), File Size (16856), Modification Time (Jun 7, 2010 5:22:20 PM), Classpaths, Compile Classpath (C:\Program Files\NetB...), Runtime Classpath (C:\Program Files\NetB...), and Boot Classpath (C:\Program Files\Java...). The status bar at the bottom shows "Customerwindow" and "REALbasic - REAL SOFTWARE".

NetBeans Technologies

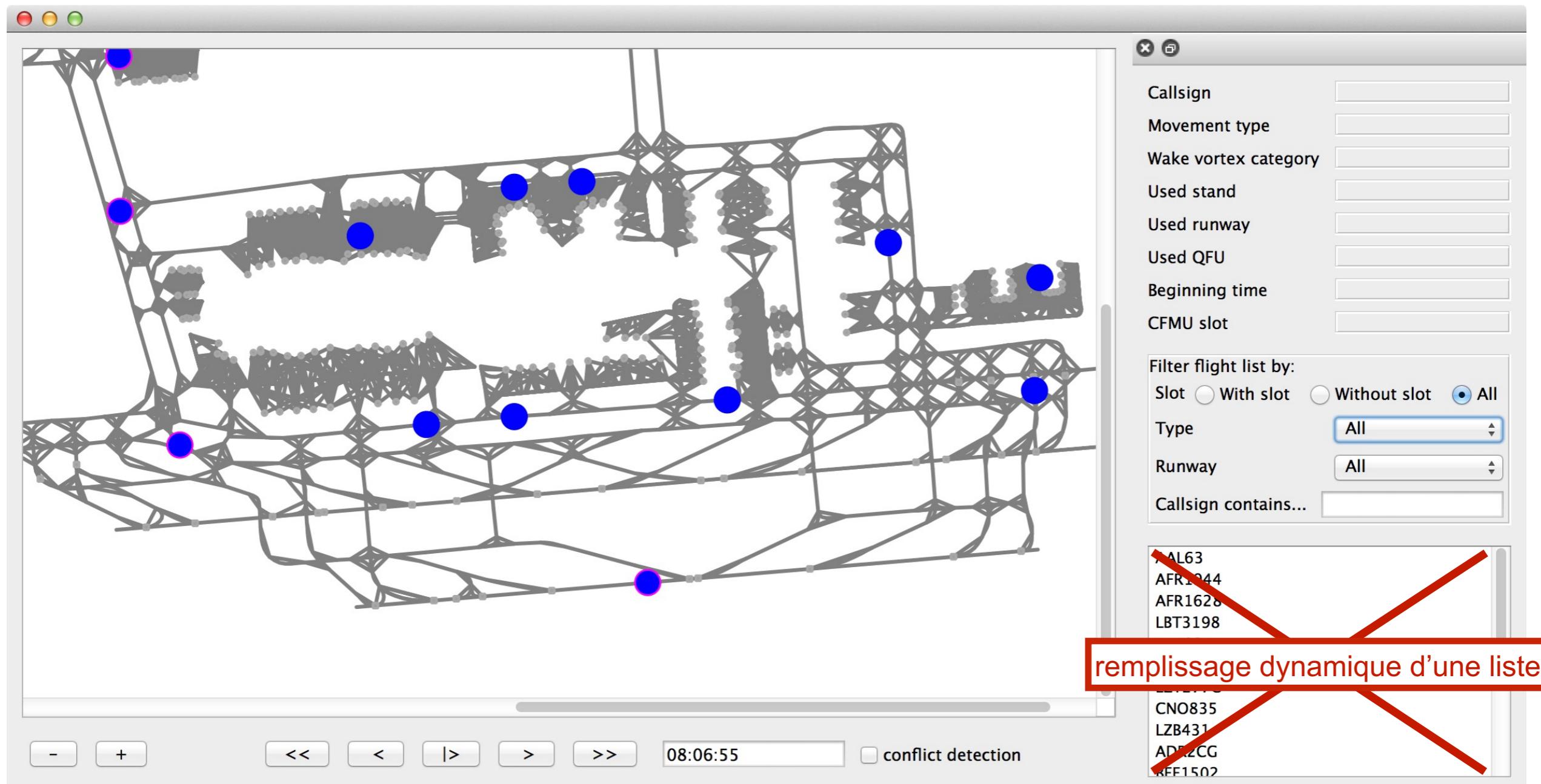
# *Qt Creator*

*Qt Creator* intègre *Qt Designer* pour le langage C++.

En plus du fichier `.ui`, il génère le code C++ automatiquement et l'utilise pour prévisualiser les widgets.

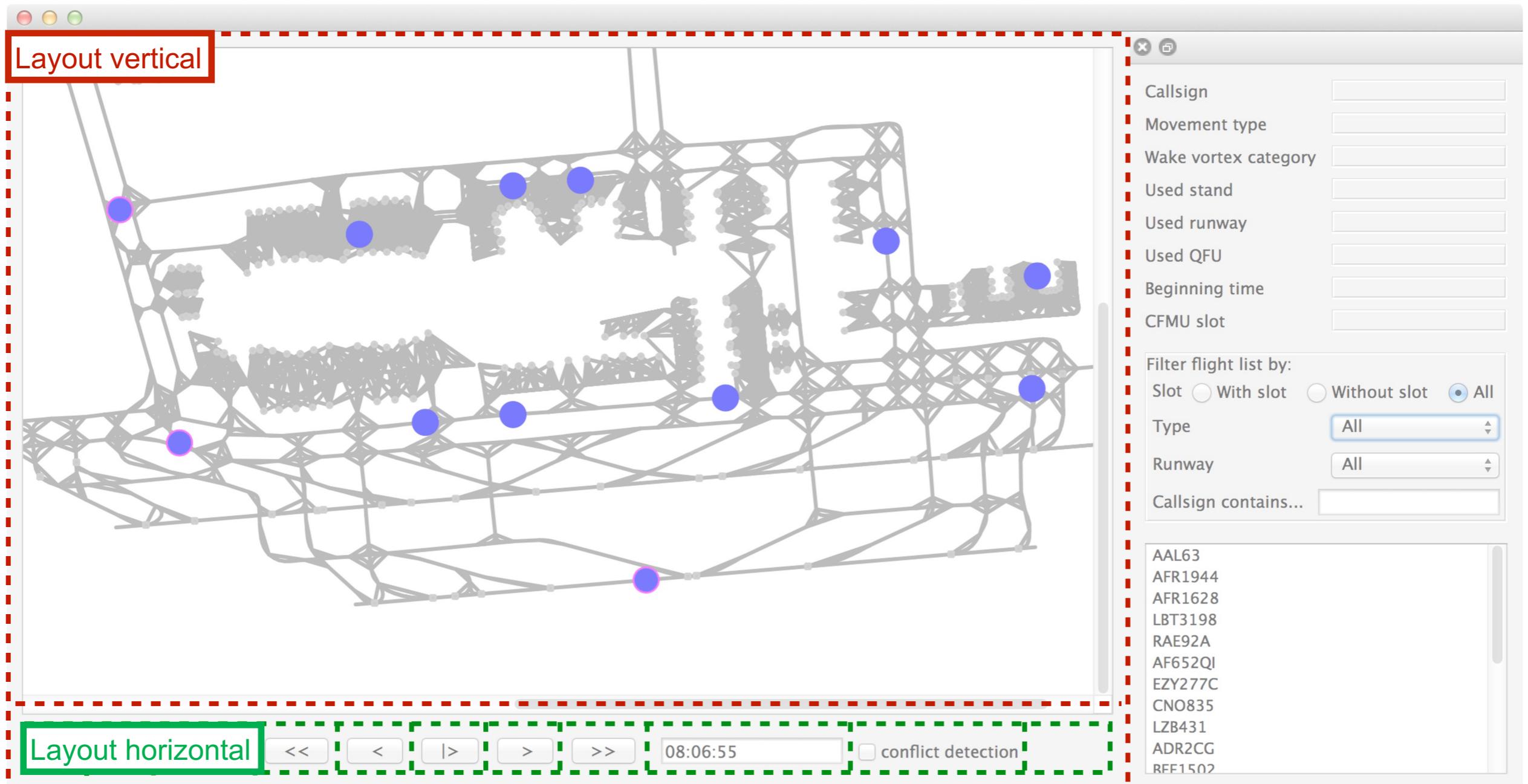
Mais on peut toujours générer du code dans tous les langages ayant une API Qt à partir du fichier `.ui`

# GUI Builder : les inconvénients



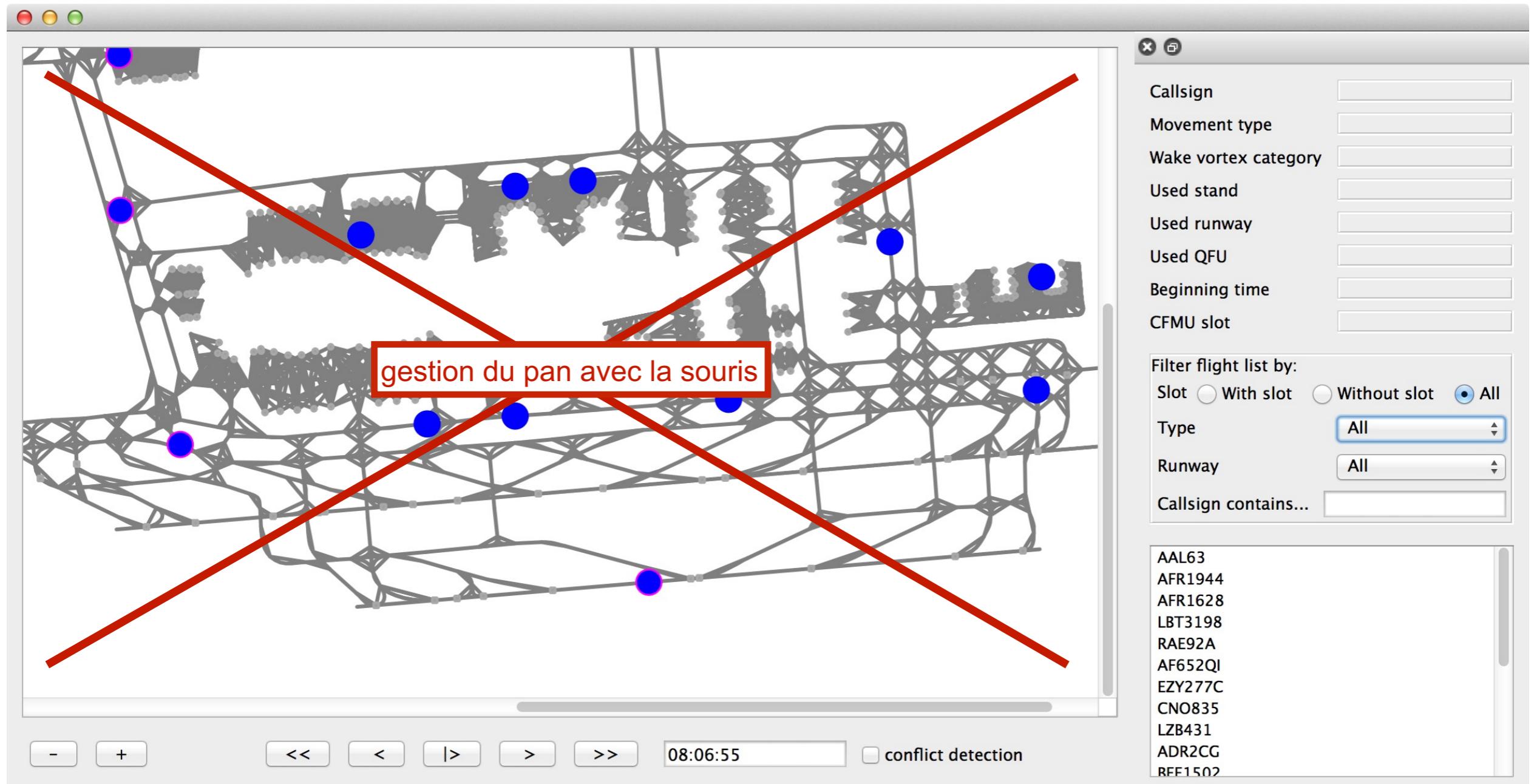
Interface figée : OK  
Interface dynamique : NOK !

# GUI Builder : les inconvénients



Contraintes de placement pas si simples à représenter de manière graphique (layout manager)

# GUI Builder : les inconvénients



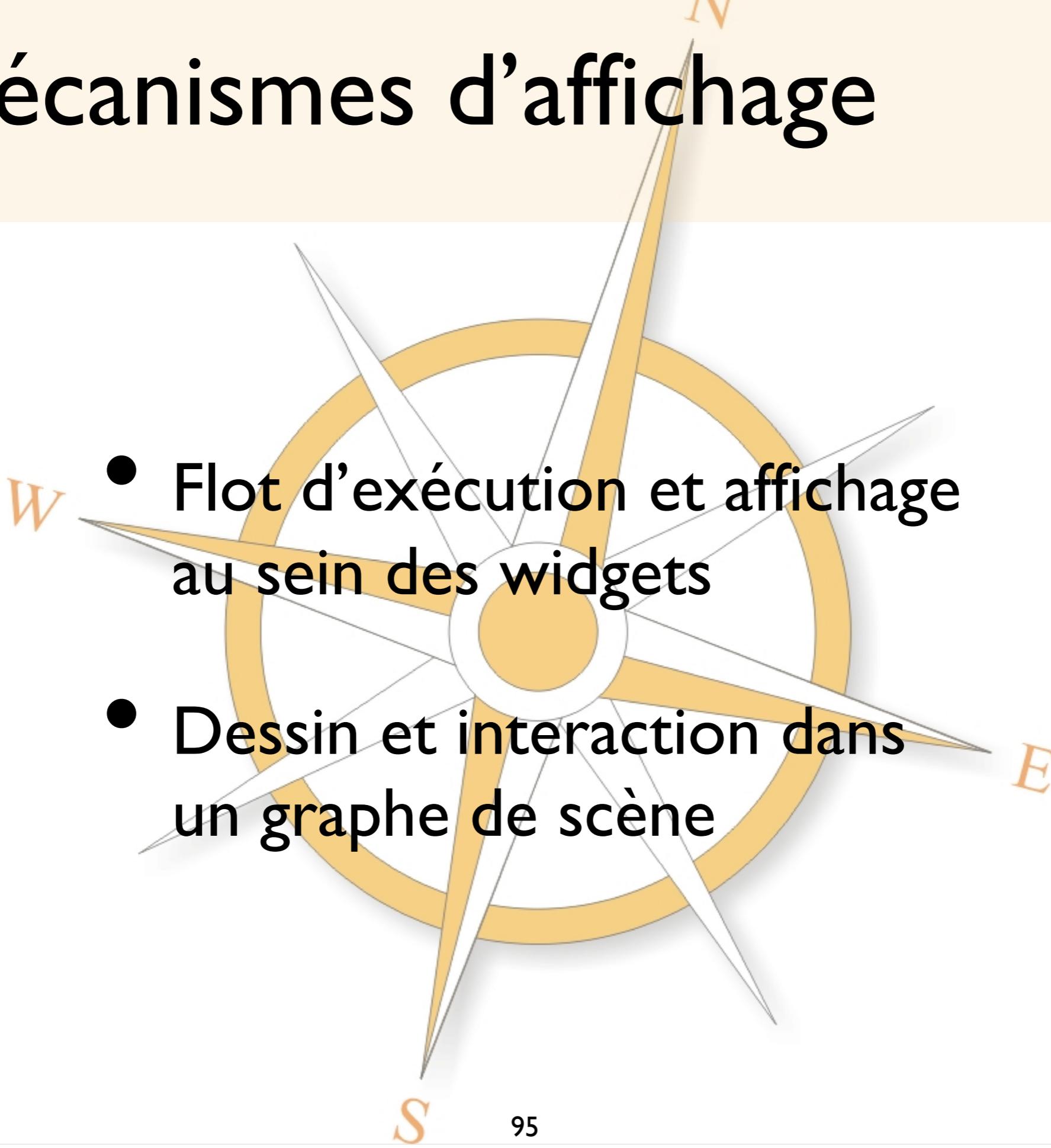
Peu de support pour des interactions autres que WIMP

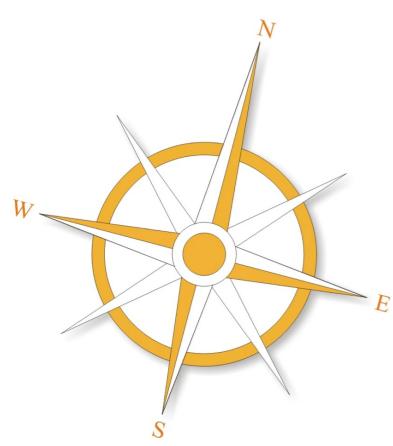
# *GUI Builder* : les inconvénients

Enfin, il reste à faire le **lien avec l'application** :

Les réactions aux actions de l'utilisateur doivent être codées dès lors qu'elles sont un peu complexes...

# 4 - Mécanismes d'affichage

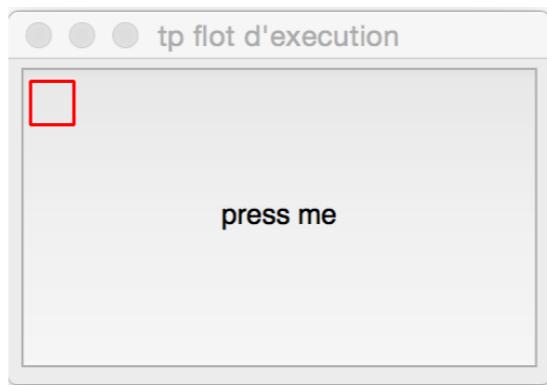




## 4. I - Flot d'exécution et affichage au sein des widgets

Les widgets standards disposent d'un mécanisme d'affichage, défini dans leur méthode paintEvent.

# Comment dessiner au sein d'un widget ?



- Pour personnaliser le dessin d'un widget il faut créer une classe héritant du widget (par ex QPushButton) et y redéfinir sa méthode paintEvent,
- pour dessiner « par dessus » le dessin habituel il faut d'abord faire appel au paintEvent de la super classe (super().paintEvent(event))...
- et utiliser ensuite un QPainter avec un contexte graphique (QPen, QBrush, QColor...) et des primitives graphiques (des méthodes pour dessiner des formes dans le QPainter).

# Cycle de vie d'un widget (rappel)

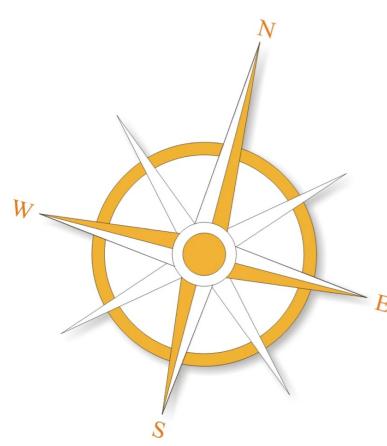
- **Créé** (allocation de ressources), et **paramétré** via l'API du widget  
`my_widget=QPushButton("titi")`
- **paramétré** via des appels de méthodes sur l'objet :  
`my_widget.setText("tutu")`
- **parenté**: ajouté à un container, pour contrôler l'imbrication, la géométrie
- **montré** ou **caché**: il est montré par défaut si son parent l'est:  
`my_widget.show()`, `my_widget.hide()`
- **abonné** à un ou plusieurs signaux  
`sld.valueChanged.connect(lcd.display)` pour définir le comportement de l'application
- **réagit** aux événements utilisateur au sein de la boucle principale `app.exec()` (fonctionnement par défaut du widget et *signaux/slots*, tout est géré par l'objet `QApplication`)
- **détruit**: (explicitement ou) automatiquement par le *garbage collector*  
`my_widget.destroy()`

et aussi le réaffichage !

# TP3

**Exercice I : Flot d'exécution et affichage au sein des widgets**





## 4.2 - Dessin et interaction dans un graphe de scène

- Qt dispose d'une classe permettant de décrire une scène graphique (`QGraphicsScene`),
- on peuple cette scène avec des items graphiques interactifs (`QGraphicsItem` et ses sous-classes),
- et on associe la scène avec une ou plusieurs vues (`QGraphicsView`) pour la visualiser et interagir avec ses éléments,
- les interactions ne se font pas à l'aide de signaux mais de méthodes déjà définies dans les items graphiques.

# Comment dessiner dans un graphe de scène ?

- Instancier une scène (QGraphicsScene) :

```
scene = QGraphicsScene()
```

- créer des items en utilisant les méthodes de la scène (addEllipse, addText, addWidget ...),

- ou en instanciant des objets héritant de QGraphicsItem et en les ajoutant à la scène, par ex :

```
text_item = QGraphicsSimpleTextItem("Coucou !")  
scene.addItem(text_item)
```

- ou en créant des items maison qui héritent de la classe voulue (dans le simu airport, AircraftItem hérite de GraphicsEllipseItem),

- paramètrer les items (couleur, fond...) avec un QPen, un QBrush...

- associer la scène avec une ou plusieurs vues (QGraphicsView) pour la visualiser et interagir avec :

```
view = QGraphicsView(scene)
```

# Comment interagir dans un graphe de scène ?

- Les `QGraphicsItem` et `QGraphicsView` ne sont pas sensibles aux signaux,
- ils disposent à la place d'un certain nombre de méthodes (`mousePressEvent`, `mouseReleaseEvent`, `wheelEvent...`) destinées à être redéfinies si on veut réagir aux événements correspondants,
- on peut faire une affectation de méthode, par ex :

```
def change_text(self, event):
    # do something

my_item.mousePressEvent = change_text
```
- ou on peut créer une nouvelle classe héritant de l'item voulu et redéfinissant les méthodes intéressantes (ex : dans le simu airport, `AircraftItem` hérite de `QGraphicsEllipseItem` et redéfinit `mousePressEvent` pour pouvoir réagir aux press sur lui-même)

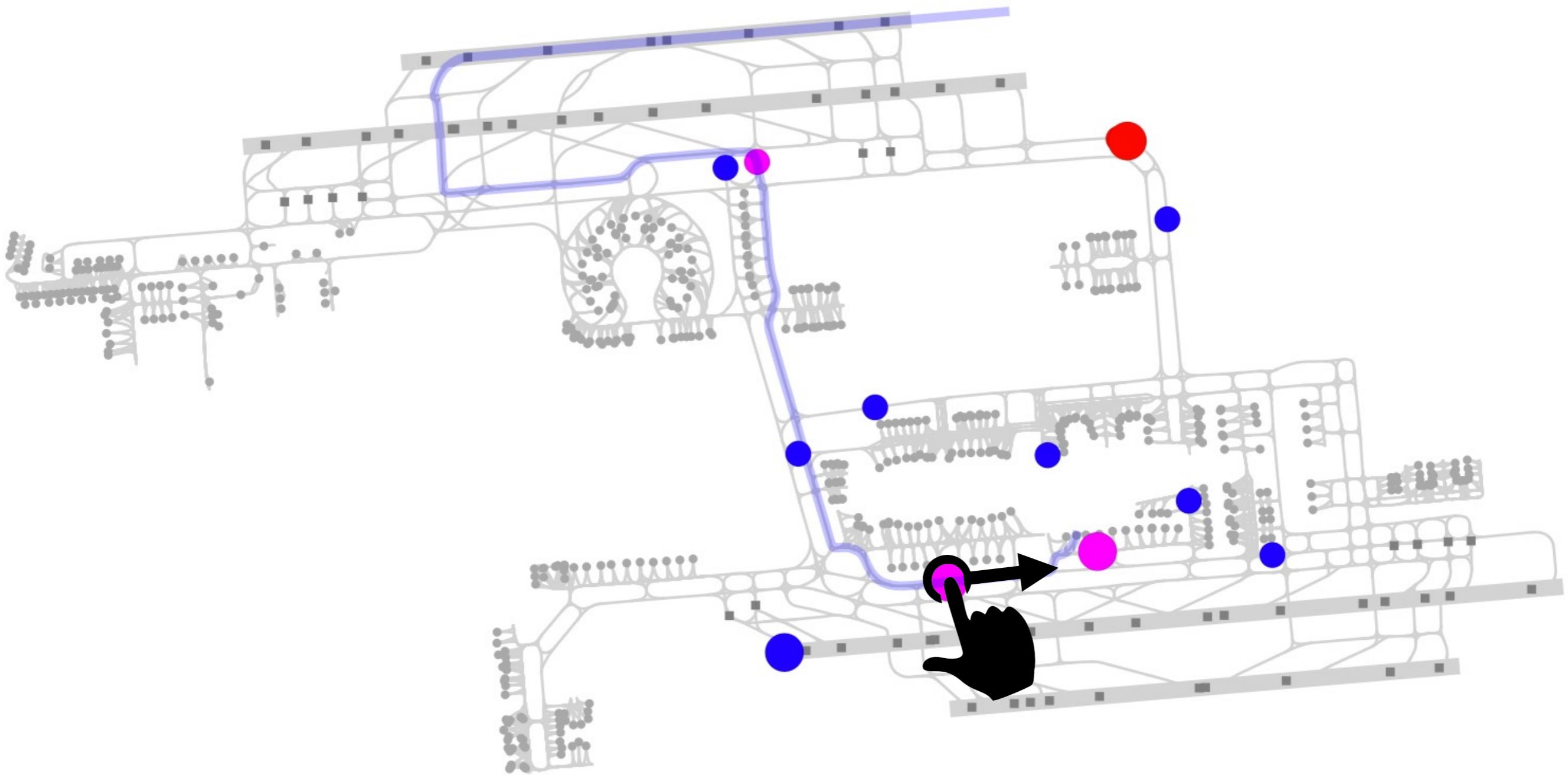
# TP3

**Exercice 2 :** Dessin et interaction dans un graphe de scène

**Exercice 3 :** Exploitation du graphe de scène dans le simu Airport



# 5 - Manipulation directe



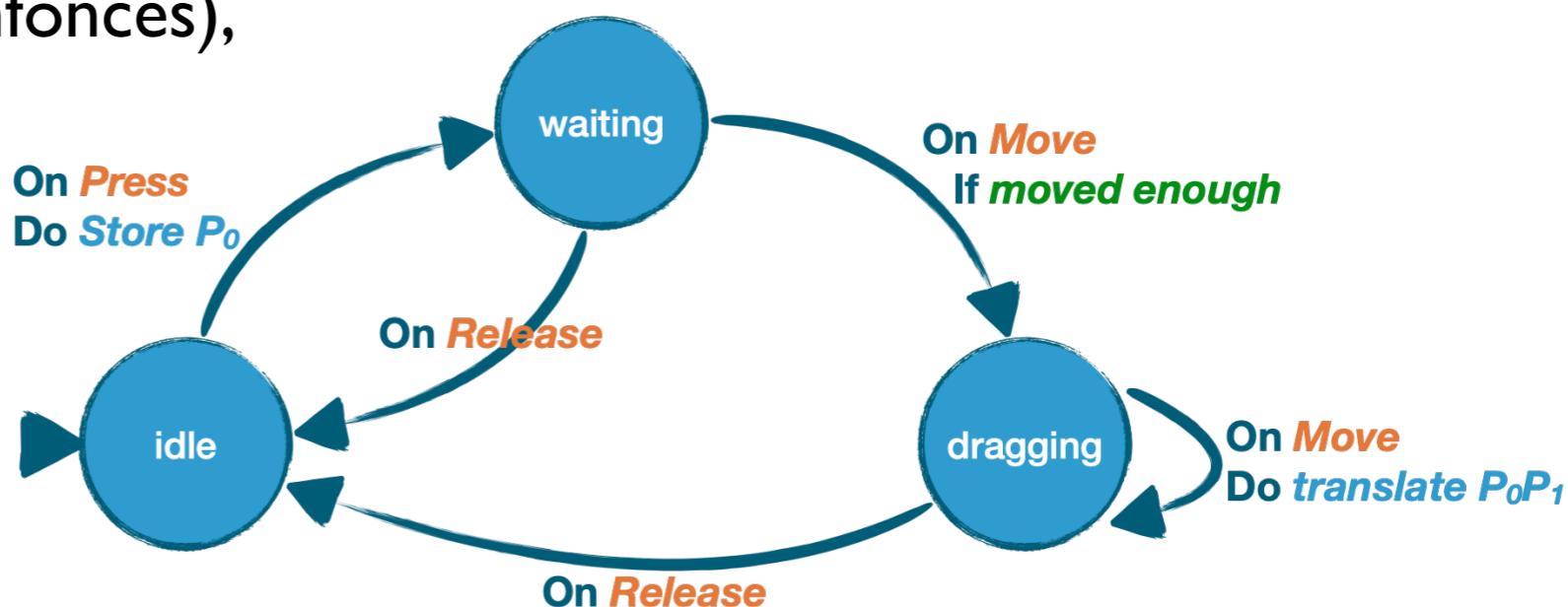
# Opérations d'un graphe de scène

- Gestion de groupes, du Z-order, stockage des items dans une structure optimisée,
- calcul des intersections, de la zone englobante (*bounding box*),
- gestion des transformations, fonctions de transformation entre coordonnées vue et coordonnées scène,
- mécanisme de gestion d'événement sur les items.

Les opérations du graphe de scène permettent de mettre facilement en place des interaction de *manipulation directe* des éléments graphiques.

# Etats et événements

- Au sein d'un objet (héritant de `QGraphicsItem`), créer un ensemble de variables d'instance permet de stocker des caractéristiques graphiques définissant un **état** (`setPen`, `setBrush`, `setPos...`),
- programmer des transitions d'un état à un autre en réaction à des événements (en réécrivant la méthode pré définie associée, par ex `mousePressEvent`), et en exploitant les dimensions de l'événement (par ex les événements souris fournissent au moins la position de la souris et les boutons enfoncés),
- idéalement, utiliser un automate à états finis pour décrire les interactions complexes.



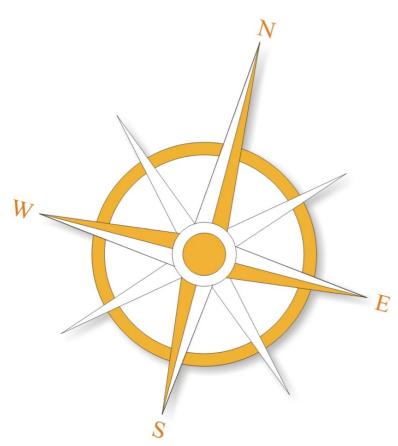
# TP4

Manipulation directe :  
Programmation d'une interaction « prendre, glisser,  
déposer »

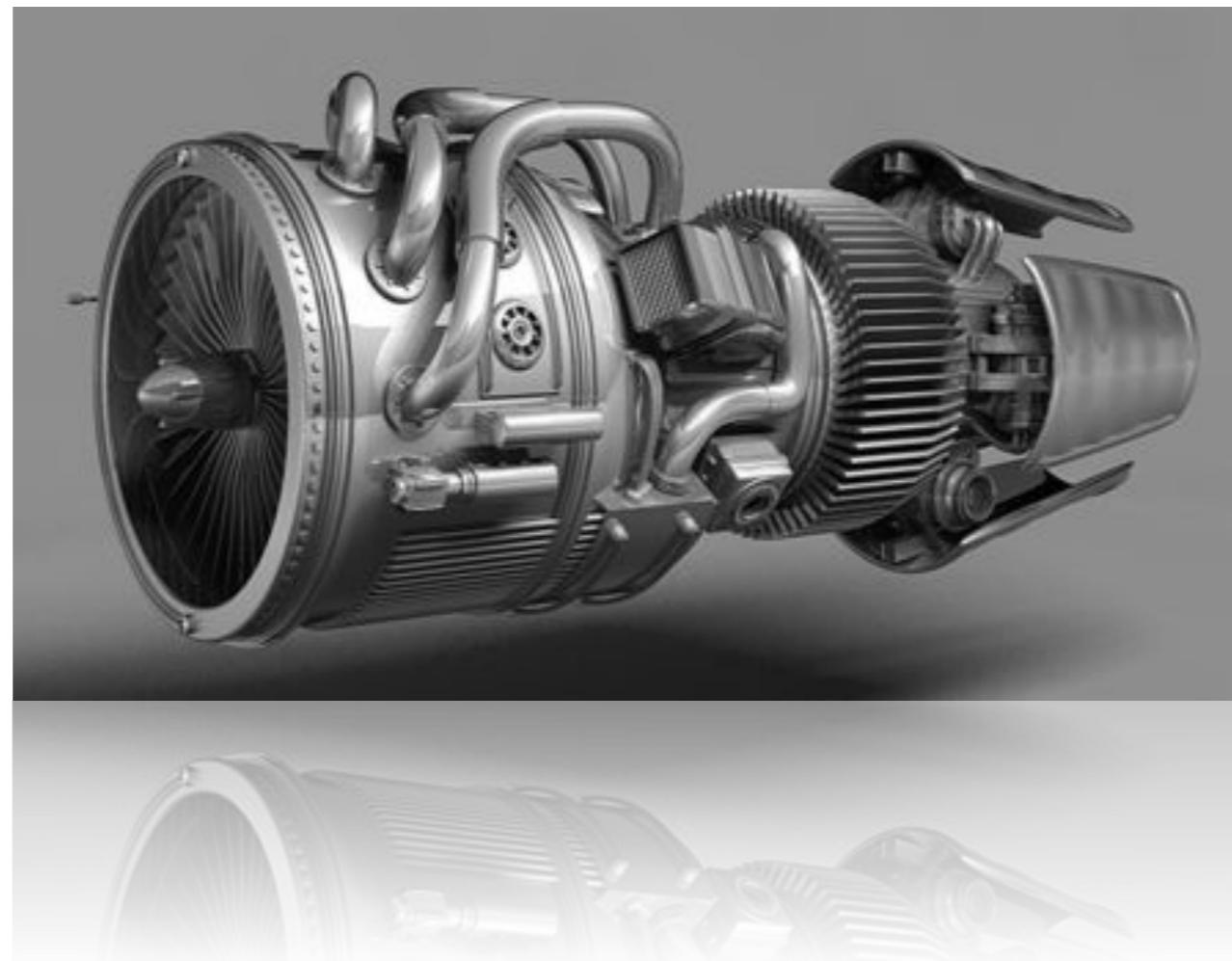


# 6 - Rappels

- Rappels sur le **W** **flot d'exécution**
- Rappels sur les **N** **mécanismes d'affichage, dessin et interaction dans un graphe de scène** **E**



# 6. I - Rappels sur le frottement d'exécution



# Comment utiliser Qt avec python ?

- Renseigner l'interpréteur, si le système est bien installé :

```
from PyQt5.QtCore import ...
from PyQt5.QtGui import ...
from PyQt5.QtWidgets import ...
```

(les *IDE* peuvent en général faire les imports automatiquement)

- créer une `app = QApplication()` qui va gérer la transmission des événements et les mécanismes d'affichage,
- créer un ou plusieurs widgets et les montrer,
- entrer dans la boucle principale `app.exec()`

# Comment ajouter un widget ?

- Créer une instance du widget, par ex :

```
my_widget = QPushButton("blabla")
```

- Si l'application ne comporte que ce widget, il suffit de le montrer (méthode show) pour qu'il soit intégré à la fenêtre de l'application :

```
my_widget.show()
```

- Si le widget doit être intégré à un parent, il faut l'ajouter au layout de celui-ci (méthode addWidget), par ex :

```
the_layout = QVBoxLayout()
the_parent = QWidget()
the_parent.setLayout(the_layout)
the_layout.addWidget(my_widget)
```

# Comment réagir à un signal ?

- Chaque widget est capable d'émettre un certain nombre de signaux (ex : valueChanged pour un QSlider)
- Pour pouvoir réagir à un signal précis émis par un widget donné, il faut s'y abonner grâce à la méthode connect :

```
my_widget.SIGNAL.connect(SLOT)
```

où SIGNAL est un signal que le widget peut émettre, et SLOT est une fonction (pas le résultat d'un appel de fonction ; on peut utiliser lambda : si on souhaite passer des paramètres).

Lorsque le widget émet le signal, le slot est alors automatiquement appelé avec comme paramètre la dimension du signal (la nouvelle valeur dans le cas du slider) ou des paramètres additionnels (si lambda :)



# QCM Signaux/slots

```
button = QPushButton("CLICK")
button.clicked.connect(myFunc)
```



Qu'est-ce qui se passe ?

1. On affiche « CLICK » sur la console
2. myFunc est appelée
3. Le bouton apparaît cliqué à l'écran
4. myFunc attend des signaux de bouton mais n'est pas appelée

# Qt Designer : Comment générer du code python ?

A partir de la description du widget créé dans *Qt Designer* (fichier *.ui*), il faut générer du code utilisable dans le langage de notre choix depuis un outil en ligne de commande (fourni avec l'API *Qt* pour ce langage).

Pour *PyQt5* il s'agit de l'utilitaire en ligne de commande *pyuic5* :

```
pyuic5 -x nom_fichier.ui -o nom_fichier_python_généré.py
```

Options à retenir : -h, --help

- x
- o

Voir TP2

Manuel *Qt Designer / PyQt5* <http://pyqt.sourceforge.net/Docs/PyQt5/designer.html>

Paramétrer VS Code pour *PyQt* <https://e-campus.enac.fr/moodle/mod/page/view.php?id=133058>

# Qt Designer : Comment utiliser le code généré ?

- Importer la classe générée dans l'espace de nom,
- créer une instance de la classe générée (`widget_QtDesigner`),
- créer une instance de QWidget (`widget_parent`) dans laquelle votre classe générée sera affichée,
- paramétriser le composant créé sous Qt Designer en appelant sa méthode `setupUi` :

```
widget_QtDesigner.setupUi(widget_parent)
```

- intégrer tout cela dans votre application, entre la création de la QApplication et la boucle principale (exemple de code à la fin du fichier généré avec l'option -x).

Voir TP2

# Comment créer un signal custom ?

- Une classe héritant de QObject (donc n'importe quel QWidget) peut déclarer un signal (variable de classe, pas d'instance) :

```
my_signal = pyqtSignal(TYPE)
```

où **TYPE** est le type d'une information qu'on décide de passer à notre nouveau signal. Il peut être un type natif de python, un nom de classe, un dict, une list, ou rien du tout (pas d'info).

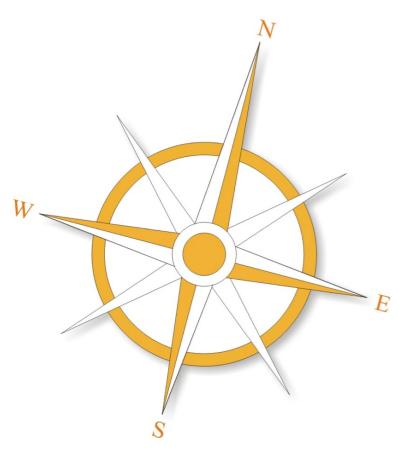
- Le signal peut ensuite être émis à volonté et convoyer toute info du **TYPE** déclaré à la création du signal :

```
my_signal.emit(my_info)
```

- Et on peut y réagir en s'abonnant de façon classique :

```
my_signal.connect(...)
```

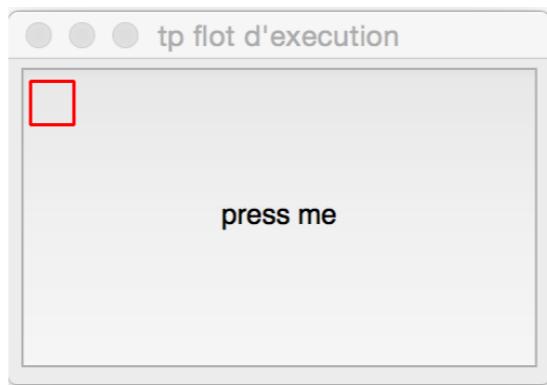
Voir TP2 et RadarView, utilisation de signaux custom pour baisser le couplage entre les différents objets.



## 6.2 - Rappels sur les mécanismes d'affichage



# Comment dessiner au sein d'un widget ?



- Pour personnaliser le dessin d'un widget il faut créer une classe héritant du widget (par ex QPushButton) et y redéfinir sa méthode paintEvent,
- pour dessiner « par dessus » le dessin habituel il faut d'abord faire appel au paintEvent de la super classe (super().paintEvent(event))...
- et utiliser ensuite un QPainter avec un contexte graphique (QPen, QBrush, QColor...) et des primitives graphiques (des méthodes pour dessiner des formes dans le QPainter).

Voir TP3

# Comment dessiner dans un graphe de scène ?

- Instancier une scène (QGraphicsScene) :

```
scene = QGraphicsScene()
```

- créer des items en utilisant les méthodes de la scène (addEllipse, addText, addWidget ...),

- ou en instanciant des objets héritant de QGraphicsItem et en les ajoutant à la scène, par ex :

```
text_item = QGraphicsSimpleTextItem("Coucou !")  
scene.addItem(text_item)
```

- ou en créant des items maison qui héritent de la classe voulue (dans le simu airport, AircraftItem hérite de GraphicsEllipseItem),

- paramètrer les items (couleur, fond...) avec un QPen, un QBrush...

- associer la scène avec une ou plusieurs vues (QGraphicsView) pour la visualiser et interagir avec :

```
view = QGraphicsView(scene)
```

Voir TP3

# Comment interagir dans un graphe de scène ?

- Les `QGraphicsItem` et `QGraphicsView` ne sont pas sensibles aux signaux,
- ils disposent à la place d'un certain nombre de méthodes (`mousePressEvent`, `mouseReleaseEvent`, `wheelEvent...`) destinées à être redéfinies si on veut réagir aux événements correspondants,
- on peut faire une affectation de méthode, par ex :

```
def change_text(self, event):
    # do something

my_item.mousePressEvent = change_text
```
- ou on peut créer une nouvelle classe héritant de l'item voulu et redéfinissant les méthodes intéressantes (ex : dans le simu airport, `AircraftItem` hérite de `QGraphicsEllipseItem` et redéfinit `mousePressEvent` pour pouvoir réagir aux press sur lui-même)

Voir TP3

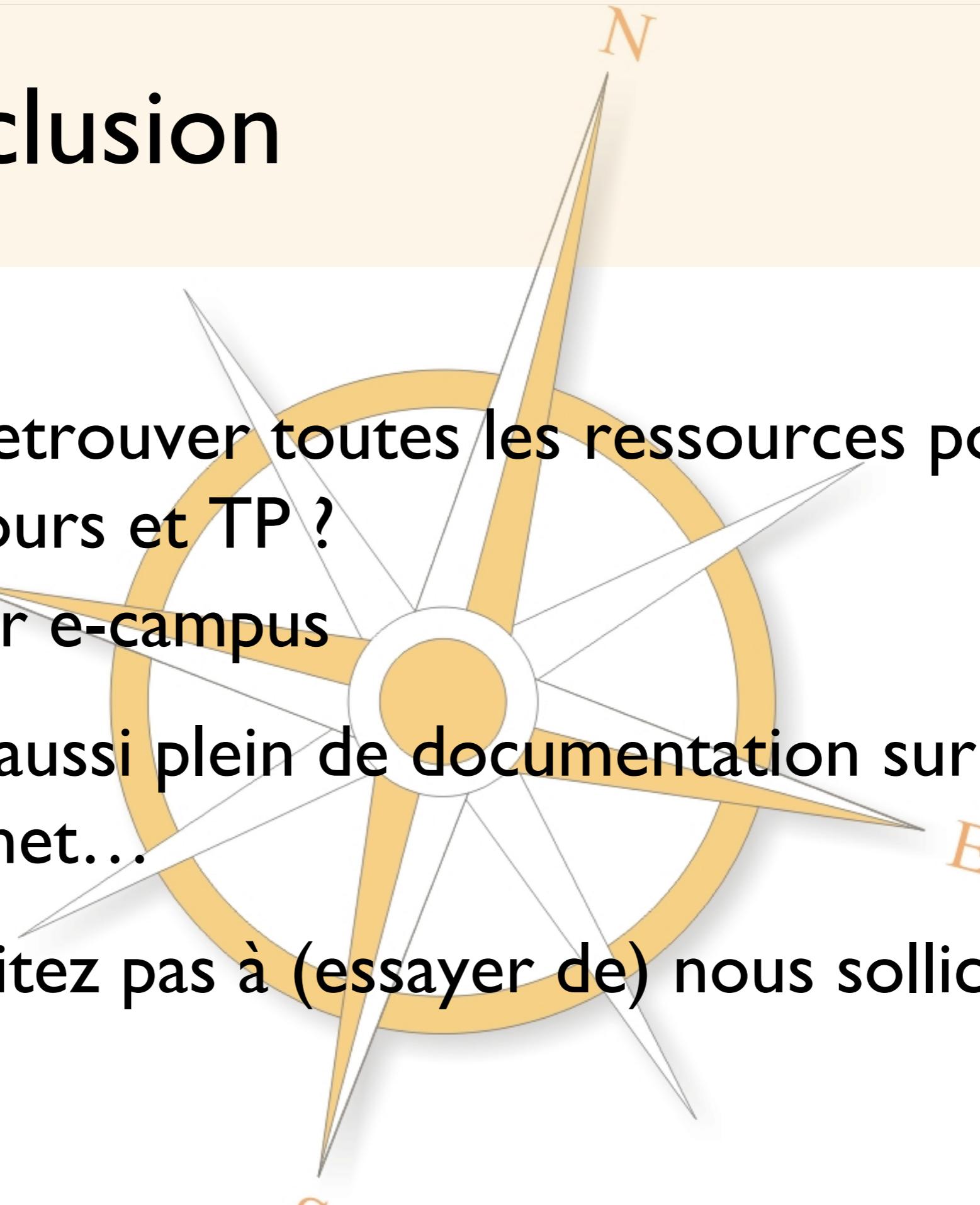


# QCM Graphe de scène

Pour afficher un rond rouge il faut :

1. Une fenêtre, une vue, un item
2. Une fenêtre, une scène, un item
3. Une fenêtre, une scène, un item, une vue
4. Une fenêtre, un layout, une scène, un item

# 7 - Conclusion

- 
- Où retrouver toutes les ressources pour les cours et TP ?  
→ sur e-campus
  - mais aussi plein de documentation sur internet...
  - n'hésitez pas à (essayer de) nous solliciter !



# Retour sur les objectifs

- Savoir utiliser l'**API** de la bibliothèque *Qt* pour programmer des **IHM**,
- savoir décrire le modèle d'exécution d'une application interactive (flot d'exécution vs flot de programmation, mécanismes d'affichage),
- savoir utiliser et assembler des **widgets**,
- savoir structurer le code d'une application réactive, utiliser les **signaux**, programmer des interactions de **manipulation directe** (prendre, glisser, déposer).



# QCM

Une API c'est ?

1. une développeuse contente,
2. du code obscur,
3. une documentation,
4. des classes, méthodes, constantes et fonctions pour utiliser une bibliothèque.



# QCM

Qu'est-ce que c'est un widget ?

1. Un environnement de développement pour concevoir des interfaces
2. Un élément de l'interface avec un comportement associé que l'utilisateur déclenche
3. Un élément statique de l'interface qui affiche l'état du système
4. Une action déclenchée par le système sans intervention par l'utilisateur