DESKTOP GUI

HOW GUI APPLICATIONS WORK?

GUI loop

HOW GUI APPLICATIONS WORK?

GUI loop

```
while True:
```

HOW GUI APPLICATIONS WORK?

GUI loop

```
while True:
  # are some keys pressed? has the mouse moved?
  events = get_events(keys, mouse)
  if events:
    handle_events(events)
```

HOW GUI APPLICATIONS WORK?

GUI loop

```
while True:
  # are some keys pressed? has the mouse moved?
  events = get_events(keys, mouse)
  if events:
    handle_events(events)
  else:
    sleep()
```

HOW GUI APPLICATIONS WORK?

GUI loop

```
while True:
  # are some keys pressed? has the mouse moved?
  events = get_events(keys, mouse)
  if events:
    handle_events(events)
  else:
    sleep()
```

- almost the same as in games

HOW GUI APPLICATIONS WORK?

GUI loop

```python
while True:
  # are some keys pressed? has the mouse moved?
  events = get_events(keys, mouse)
  if events:
    handle_events(events)
  else:
    sleep()
```

- almost the same as in games
- GUI is only redrawn after an event happens (keyboard, mouse, network, timer, ...)

HOW GUI APPLICATIONS WORK?

GUI loop

```
while True:
  # are some keys pressed? has the mouse moved?
  events = get_events(keys, mouse)
  if events:
    handle_events(events)
  else:
    sleep()
```

- almost the same as in games
- GUI is only redrawn after an event happens (keyboard, mouse, network, timer, ...)
- without events the application sleeps

HOW IS THE UI DESIGNED?

HOW IS THE UI DESIGNED?

- GUI apps are composed of a hierarchy of UI elements (called widgets, views, components, ...)

HOW IS THE UI DESIGNED?

- GUI apps are composed of a hierarchy of UI elements (called widgets, views, components, ...)
- Together they form a tree of widgets with parent/child relationships

HOW IS THE UI DESIGNED?

- GUI apps are composed of a hierarchy of UI elements (called widgets, views, components, ...)
- Together they form a tree of widgets with parent/child relationships
- Composite design pattern

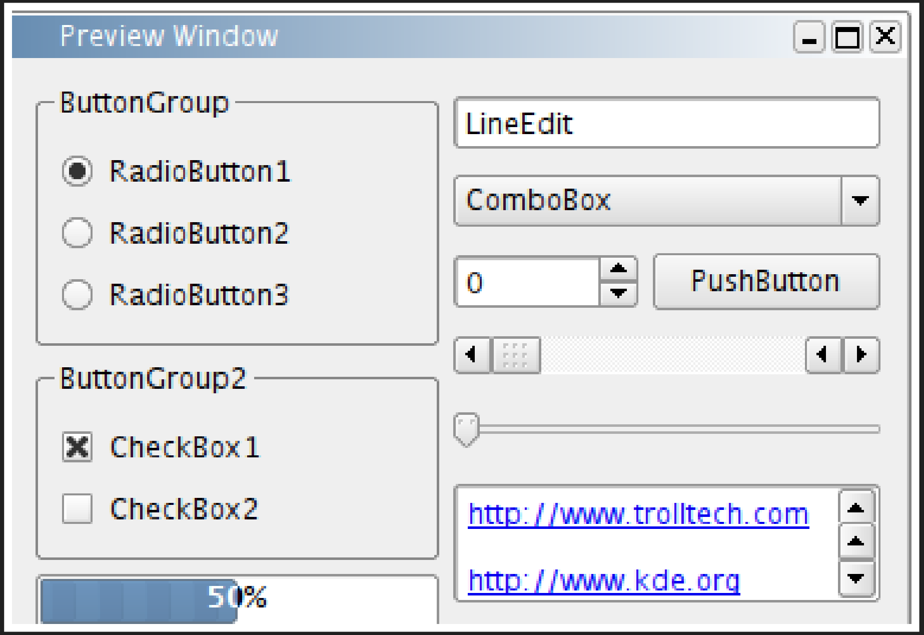
HOW IS THE UI DESIGNED?

- GUI apps are composed of a hierarchy of UI elements (called widgets, views, components, ...)
- Together they form a tree of widgets with parent/child relationships
- Composite design pattern
- Sometimes the hierarchy is specified with XML files or UI designer app

HOW THE HIERARCHY LOOKS IN (PSEUDO)CODE?

```
window = Window()
row = Row()                           # widget container
row.add(Button("Click me!"))  # add widgets to container
row.add(Label("Hello world"))
window.add(row)
```

TYPICAL WIDGETS



- Button
- Label
- TextInput
- Checkbox/Radio
- ComboBox
- ...

USER INTERACTION

USER INTERACTION

- callbacks (functions) are bound to specific events

USER INTERACTION

- callbacks (functions) are bound to specific events
- click on a button, typing a character into a textbox, mouse move, etc.

USER INTERACTION

- callbacks (functions) are bound to specific events
- click on a button, typing a character into a textbox, mouse move, etc.
- when an event happens, control goes from the GUI loop to the callback

USER INTERACTION

- callbacks (functions) are bound to specific events
- click on a button, typing a character into a textbox, mouse move, etc.
- when an event happens, control goes from the GUI loop to the callback
- the callback should NOT block, otherwise the whole UI will freeze

USER INTERACTION

- callbacks (functions) are bound to specific events
- click on a button, typing a character into a textbox, mouse move, etc.
- when an event happens, control goes from the GUI loop to the callback
- the callback should NOT block, otherwise the whole UI will freeze

```python
def on_button_click():
  print("button was clicked")
```

USER INTERACTION

- callbacks (functions) are bound to specific events
- click on a button, typing a character into a textbox, mouse move, etc.
- when an event happens, control goes from the GUI loop to the callback
- the callback should NOT block, otherwise the whole UI will freeze

```
def on_button_click():
  print("button was clicked")

app = Window()
btn = Button("Click me!")
btn.onclick.set_callback(on_button_click)
app.add(btn)
app.mainloop()
```

GUI FRAMEWORKS

Functionality:

GUI FRAMEWORKS

Functionality:

- Window/dialog management (open, close, resize, ...)

GUI FRAMEWORKS

Functionality:

- Window/dialog management (open, close, resize, ...)
- Input management (keyboard and mouse events)

GUI FRAMEWORKS

Functionality:

- Window/dialog management (open, close, resize, ...)
- Input management (keyboard and mouse events)
- 2D drawing (text, images, lines, rectangles, shadows, ...)

GUI FRAMEWORKS

Functionality:

- Window/dialog management (open, close, resize, ...)
- Input management (keyboard and mouse events)
- 2D drawing (text, images, lines, rectangles, shadows, ...)
- Common widget implementation

GUI FRAMEWORKS

Functionality:

- Window/dialog management (open, close, resize, ...)
- Input management (keyboard and mouse events)
- 2D drawing (text, images, lines, rectangles, shadows, ...)
- Common widget implementation
- Should be multi-platform

GUI FRAMEWORKS

Functionality:

- Window/dialog management (open, close, resize, ...)
- Input management (keyboard and mouse events)
- 2D drawing (text, images, lines, rectangles, shadows, ...)
- Common widget implementation
- Should be multi-platform
- You don't want to implement this yourself

GUI FRAMEWORKS

- **Qt** (C++, Python, cross-platform)
- GTK (C++, Python, cross-platform)
- WxWidgets (C++, Python, cross-platform)
- Tcl/Tk (Python, Perl, Ruby, cross-platform)
- WinForms/WPF (C#, Windows)
- AWT/Swing/SWT/JavaFX (Java, cross-platform)
- Cocoa (Objective C, OS X/iOS)
- ...

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE
- Google Earth

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE
- Google Earth
- Skype

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE
- Google Earth
- Skype
- Source engine tools

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE
- Google Earth
- Skype
- Source engine tools
- TeamSpeak

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE
- Google Earth
- Skype
- Source engine tools
- TeamSpeak
- TeamViewer

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE
- Google Earth
- Skype
- Source engine tools
- TeamSpeak
- TeamViewer
- VLC player

QT

- multi-platform
- C++, Python
- easy to use, well designed

Programs built with Qt:

- Adobe Photoshop Album, Elements
- Autodesk Maya
- Battle.net client
- CryEngine editor
- EAGLE
- Google Earth
- Skype
- Source engine tools
- TeamSpeak
- TeamViewer
- VLC player
- and many others...

- We will use Qt 5 with Python binding PyQt5

```
$ pip install PyQt5
```

MINIMAL QT APP

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

app = QApplication()
button = QPushButton("Hi!")
button.show()   # tells the button to be visible
app.exec_()     # start the GUI loop
```

LAYOUTS

LAYOUTS

- absolute positioning (place widget at pixel (113, 67)) doesn't scale

LAYOUTS

- absolute positioning (place widget at pixel (113, 67)) doesn't scale
- when adding widgets to the hierarchy, you want them to be positioned automatically

LAYOUTS

- absolute positioning (place widget at pixel (113, 67)) doesn't scale
- when adding widgets to the hierarchy, you want them to be positioned automatically
- Qt has many automatic layouts: row, column, grid, etc.

LAYOUTS

- absolute positioning (place widget at pixel (113, 67)) doesn't scale
- when adding widgets to the hierarchy, you want them to be positioned automatically
- Qt has many automatic layouts: row, column, grid, etc.

```
# horizontal box (row), shortcut QHBoxLayout
layout = QBoxLayout(QBoxLayout.LeftToRight)
```

LAYOUTS

- absolute positioning (place widget at pixel (113, 67)) doesn't scale
- when adding widgets to the hierarchy, you want them to be positioned automatically
- Qt has many automatic layouts: row, column, grid, etc.

```
# horizontal box (row), shortcut QHBoxLayout
layout = QBoxLayout(QBoxLayout.LeftToRight)
# vertical box (column), shortcut QVBoxLayout
# layout = QBoxLayout(QBoxLayout.TopToBottom)
```

LAYOUTS

- absolute positioning (place widget at pixel (113, 67)) doesn't scale
- when adding widgets to the hierarchy, you want them to be positioned automatically
- Qt has many automatic layouts: row, column, grid, etc.

```
# horizontal box (row), shortcut QHBoxLayout
layout = QBoxLayout(QBoxLayout.LeftToRight)
# vertical box (column), shortcut QVBoxLayout
# layout = QBoxLayout(QBoxLayout.TopToBottom)

layout.addWidget(QPushButton("Btn 1"))
layout.addWidget(QPushButton("Btn 2"))
```

LAYOUTS

- absolute positioning (place widget at pixel (113, 67)) doesn't scale
- when adding widgets to the hierarchy, you want them to be positioned automatically
- Qt has many automatic layouts: row, column, grid, etc.

```
# horizontal box (row), shortcut QHBoxLayout
layout = QBoxLayout(QBoxLayout.LeftToRight)
# vertical box (column), shortcut QVBoxLayout
# layout = QBoxLayout(QBoxLayout.TopToBottom)

layout.addWidget(QPushButton("Btn 1"))
layout.addWidget(QPushButton("Btn 2"))

window = QWidget()
window.setLayout(layout)
window.show()
```

EVENT HANDLING

EVENT HANDLING

- widgets have events (signals) that are connected to callbacks (slots)

EVENT HANDLING

- widgets have events (signals) that are connected to callbacks (slots)

```
btn = QPushButton("Btn 1")
btn.clicked.connect(lambda event: print(event))
```

EVENT HANDLING

- widgets have events (signals) that are connected to callbacks (slots)

```
btn = QPushButton("Btn 1")
btn.clicked.connect(lambda event: print(event))
```

Own events

EVENT HANDLING

- widgets have events (signals) that are connected to callbacks (slots)

```
btn = QPushButton("Btn 1")
btn.clicked.connect(lambda event: print(event))
```

Own events

EVENT HANDLING

- widgets have events (signals) that are connected to callbacks (slots)

```
btn = QPushButton("Btn 1")
btn.clicked.connect(lambda event: print(event))
```

Own events

```
class MyWidget(QWidget):
  MyEvent = pyqtSignal(int, int)

  def fn():
    self.MyEvent.emit(1, 2)
```

EVENT HANDLING

- widgets have events (signals) that are connected to callbacks (slots)

```
btn = QPushButton("Btn 1")
btn.clicked.connect(lambda event: print(event))
```

Own events

```
class MyWidget(QWidget):
  MyEvent = pyqtSignal(int, int)

  def fn():
    self.MyEvent.emit(1, 2)

widget = MyWidget()
widget.MyEvent.connect(lambda x, y: print(x + y))
widget.fn()  # prints 3
```

UI REFRESH

UI REFRESH

- unlike in games, the UI is not refreshed X times per second

UI REFRESH

- unlike in games, the UI is not refreshed X times per second
- when the app state changes, you have to manually redraw the changed (or all) widgets

UI REFRESH

- unlike in games, the UI is not refreshed X times per second
- when the app state changes, you have to manually redraw the changed (or all) widgets

```
widget.repaint()  # redraw the widget
```

UI REFRESH

- unlike in games, the UI is not refreshed X times per second
- when the app state changes, you have to manually redraw the changed (or all) widgets

```
widget.repaint()  # redraw the widget
```

Observer pattern is very useful here, widgets listen to changes in the app state:

UI REFRESH

- unlike in games, the UI is not refreshed X times per second
- when the app state changes, you have to manually redraw the changed (or all) widgets

```
widget.repaint()  # redraw the widget
```

Observer pattern is very useful here, widgets listen to changes in the app state:

UI REFRESH

- unlike in games, the UI is not refreshed X times per second
- when the app state changes, you have to manually redraw the changed (or all) widgets

```
widget.repaint()  # redraw the widget
```

Observer pattern is very useful here, widgets listen to changes in the app state:

```
class LikeCountDisplay(QWidget):
  def __init__(self, state):
    state.on_change.set_listener(self.update)

  def update(self):
    self.label.setText(state.get_likes())
```

UI REFRESH

- unlike in games, the UI is not refreshed X times per second
- when the app state changes, you have to manually redraw the changed (or all) widgets

```
widget.repaint()  # redraw the widget
```

Observer pattern is very useful here, widgets listen to changes in the app state:

```
class LikeCountDisplay(QWidget):
  def __init__(self, state):
    state.on_change.set_listener(self.update)

  def update(self):
    self.label.setText(state.get_likes())


counter = LikeCounter(tweet)
widget = LikeCountDisplay(counter)
...                    # sometime later
counter.add_like()     # widget is refreshed automatically
```

2D DRAWING

2D DRAWING

```
class MyWidget(QWidget):
  def paintEvent(self, *args, **kwargs):
    painter = QPainter(self)

    # pen is used for drawing (rectangle edges)
    painter.setPen(QColor.fromRgb(255, 0, 0))
    painter.drawRect(x1, y1, width, height)

    # brush is used for filling (rectangle area)
    painter.setBrush(QColor("red"))
    painter.fillRect(x, y, width, height)
```

2D DRAWING

```
class MyWidget(QWidget):
  def paintEvent(self, *args, **kwargs):
    painter = QPainter(self)

    # pen is used for drawing (rectangle edges)
    painter.setPen(QColor.fromRgb(255, 0, 0))
    painter.drawRect(x1, y1, width, height)

    # brush is used for filling (rectangle area)
    painter.setBrush(QColor("red"))
    painter.fillRect(x, y, width, height)
```

- often you need to change 2D drawing attributes (color, line width)
- save/restore keeps the changes local so they don't affect the rest of the drawing code
- ideal candidate for a context managet (**with** construct)

2D DRAWING

```
class MyWidget(QWidget):
  def paintEvent(self, *args, **kwargs):
    painter = QPainter(self)

    # pen is used for drawing (rectangle edges)
    painter.setPen(QColor.fromRgb(255, 0, 0))
    painter.drawRect(x1, y1, width, height)

    # brush is used for filling (rectangle area)
    painter.setBrush(QColor("red"))
    painter.fillRect(x, y, width, height)
```

- often you need to change 2D drawing attributes (color, line width)
- save/restore keeps the changes local so they don't affect the rest of the drawing code
- ideal candidate for a context managet (**with** construct)

```
painter.save()    # save painter attributes to an internal sta
painter.setPen(QPen(QColor.fromRgb(255, 0, 0))) # set red pen
painter.drawLine(x1, y1, x2, y2)  # draw red line
painter.restore() # revert to the original state
```

UI/LOGIC SEPARATION

What's wrong with this code?

UI/LOGIC SEPARATION

What's wrong with this code?

```
class GameBoard(QWidget):
  def mousePressEvent(self, event):
    x = event.x()
    y = event.y()
    cell = self.board[x][y]
    if cell == Empty:
      self.board[x][y] = Cross
    if self.check_win():
      print("game ended")

  def check_win():
    ...
```

UI/LOGIC SEPARATION

What's wrong with this code?

```
class GameBoard(QWidget):
  def mousePressEvent(self, event):
    x = event.x()
    y = event.y()
    cell = self.board[x][y]
    if cell == Empty:
      self.board[x][y] = Cross
    if self.check_win():
      print("game ended")

  def check_win():
    ...
```

App (game) logic is combined with UI code!

UI/LOGIC SEPARATION

What's wrong with this code?

```
class GameBoard(QWidget):                    # game logic is bound to U
  def mousePressEvent(self, event): # input is bound to mouse
    x = event.x()
    y = event.y()
    cell = self.board[x][y]
    if cell == Empty:
      self.board[x][y] = Cross
    if self.check_win():
      print("game ended")

  def check_win():
    ...
```

App (game) logic is combined with UI code!

This makes the game code hard to modify and test:

This makes the game code hard to modify and test:

- want to run game as a multiplayer server without UI? rewrite

This makes the game code hard to modify and test:

- want to run game as a multiplayer server without UI? rewrite
- want to test the game code? rewrite

This makes the game code hard to modify and test:

- want to run game as a multiplayer server without UI? rewrite
- want to test the game code? rewrite
- want to support multiple UI outputs (console, web, desktop GUI)? rewrite

This makes the game code hard to modify and test:

- want to run game as a multiplayer server without UI? rewrite
- want to test the game code? rewrite
- want to support multiple UI outputs (console, web, desktop GUI)? rewrite
- want to change the GUI framework? rewrite

This makes the game code hard to modify and test:

- want to run game as a multiplayer server without UI? rewrite
- want to test the game code? rewrite
- want to support multiple UI outputs (console, web, desktop GUI)? rewrite
- want to change the GUI framework? rewrite
- want to change the game code? must touch the UI code

Solution

- separate UI code and logic (domain) code
- the game code should not know anything about the UI
- dependency inversion - observer pattern again very useful

Solution

- separate UI code and logic (domain) code
- the game code should not know anything about the UI
- dependency inversion - observer pattern again very useful

Solution

- separate UI code and logic (domain) code
- the game code should not know anything about the UI
- dependency inversion - observer pattern again very useful

```
class Game: # in separate file, knows nothing about the UI
  def move(self, x, y): pass
  def set_on_move_listener(self, listener): pass
```

Solution

- separate UI code and logic (domain) code
- the game code should not know anything about the UI
- dependency inversion - observer pattern again very useful

```
class Game: # in separate file, knows nothing about the UI
  def move(self, x, y): pass
  def set_on_move_listener(self, listener): pass

class GameBoard(QWidget):
  def __init__(self, game):
    self.game = game
    self.game.set_on_move_listener(lambda: self.redraw())
  def mousePressEvent(self, event):
    x = event.x()
    y = event.y()
    self.game.move(x, y)
    if self.game.check_win():
      print("game ended")
```