CIP de FP BATOI

DESENROTLLAMENT D'INTERFÍCIES
2N DAM

S.A.2: Creació i us de components.

Professor: Iván Martos

S.A.2: Creació i us de components

Contenido

En aquesta segona unitat anem a treballar diferents maneres de crear interfícies gràfiques. En aquesta unitat farem ús de diferents llenguatges de programació, diferents frameworks i IDEs.

Començarem amb una breu introducció a la creació d'interfícies gràfiques utilitzant el llenguatge de programació **Python** i la biblioteca específica per a Python anomenada **PySide** que permet crear interfícies gràfiques de Qt.

1. Programari necessari per a les pràctiques

En aquesta part, anem a necessitar disposar del següent programari:

- **Python (https://www.python.org)**
- **Visual Studio Code (https://code.visualstudio.com) .**

Una vegada hem instal·lat el programari, cal instal·lar les pertinents extensions al Visual Studio Code. Per una banda instal·larem **l'extensió de Python** per al Visual Studio Code en l'apartat d'Extensions del MarketPlace del Visual Studio Code. També instal·larem **l'extensió QT for Python**. Després des del propi Terminal del Visual Studio Code instal·larem el PySide 6 seguint les següents comandes: **pip install pyside6**. Abans d'instal·lar Pyside, hem d'assegurar-se que ja tenim instal·lat Python al nostre equip.

Respecte a la documentació i suport oficial destaquem:

- PySide6: https://pypi.org/project/PySide6/
- Documentació Qt6: https://doc.qt.io
- Qt for Python: https://doc.qt.io/qtforpython-6/

Cal tindre en compte que tenim entorns de desenvolupament més complets com son **Qt Creator** (https://www.qt.io/product/development-tools) i també Qt Designer, però atenent al temps anirem a programar directament amb Python des del Visual Studio Code.

2. Aplicació base (QAplication)

Estructura bàsica d'un programa a PySide usant el component base de Qt anomenat QWidget, un widget buit:

```
from PySide6.QtWidgets import QApplication, QWidget

# Creamos una aplicación para gestionar la interfaz
app = QApplication()

# Creamos un widget para generar la ventana
window = QWidget()

# Mostramos la ventana, se encuentra oculta por defecto
window.show()

# Iniciamos el bucle del programa
app.exec()
```

QApplication: És el nucli d'un programa a Qt, és necessari per gestionar el bucle de l'aplicació, encarregat de gestionar totes les interaccions amb la interfície gràfica d'usuari.

Una aplicació requereix com a mínim un widget per mostrar alguna cosa en pantalla. Tots els widgets que hereten de QWidget es poden visualitzar com a finestres en si mateixos:

```
from PySide6.QtWidgets import QApplication, QPushButton

app = QApplication(sys.argv)

# Esta vez la ventana la maneja un widget de tipo botón
window = QPushButton("Hola mundo")
window.show()

app.exec()
```

Com podeu observar això no és gaire útil, ja que tota la finestra és el botó en si mateix.

Per sort Qt ens ofereix un widget capaç de gestionar finestres amb multitud de funcionalitats, es diu QMainWindow, el widget per gestionar finestres principals.

```
from PySide6.QtWidgets import QApplication, QMainWindow

app = QApplication()

# Ahora la ventana la gestiona el widget de ventana principal
window = QMainWindow()

# Damos un título a la ventana principal
window.setWindowTitle("Hola mundo")

window.show()

app.exec()
```

Aparentment tenim el mateix que en utilitzar un QWidget, però aquesta finestra principal ens permet assignar un widget per ocupar el seu espai central:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton

app = QApplication()
window = QMainWindow()
window.setWindowTitle("Hola mundo")

# Guardamos el botón en una variable
button = QPushButton("Soy un botón")
# Establecemos el botón como widget central de la ventana principal
window.setCentralWidget(button)

window.show()
app.exec()
```

3. Classes, superclasses i subclasses

A la propera lliçó transformarem el codi del nostre programa a classes i objectes. Per fer-ho, necessitarem treballar amb classes i l'herència de classes, així que repassem una mica els conceptes bàsics.

3.1 Classes

Per a comprendre con es creen les classes en Python anem a fer una correspondència amb Java, per exemple, una classe en Java podria ser:

```
public class Persona {
   // Atributos
   private String nombre;
   private int edad;
   private String ciudad;

   // Constructor
   public Persona(String nombre, int edad, String ciudad) {
      this.nombre = nombre;
      this.edad = edad;
      this.ciudad = ciudad;
   }

   // Método
   public void saludar() {
      System.out.println("Hola, soy " + nombre + ", tengo " + edad + " años y vivo en " +
ciudad + ".");
   }

   // Método principal para probar
   public static void main(String[] args) {
      Persona persona1 = new Persona("Iván", 40, "Alcoi");
      persona1.saludar();
   }
}
```

El mateix exemple en Python seria el següent:

```
class Persona:
   def __init__(self, nombre, edad, ciudad):
      self.nombre = nombre
      self.edad = edad
      self.ciudad = ciudad

   def saludar(self):
```

print(f"Hola, soy {self.nombre}, tengo {self.edad} años y vivo en {self.ciudad}.")

# Crear objeto
persona1 = Persona("Iván", 40, "Alcoi")
persona1.saludar()

Explicació:

- No es declaren tipus de dades (Python és dinàmic).
- __init__ és el constructor (s'executa en crear l'objecte, rep com a primer paràmetre self, que fa referència a si mateix).
- self equival a this a Java.
- No es fa servir new en instància de l'objecte.
- No cal un main(): el codi es pot executar directament.

3.2 Superclasses i subclasses.

A Python, quan una classe (anomenada subclasse) hereda d'una altra (anomenada superclasse) obté tot el seu comportament.

```
class Madre:
    def __init__(self):
        print(f"Soy Madre")

class Hijo(Madre):
    pass


hijo = Hijo()
```

La subclasse pot sobreescriure els mètodes de la superclasse per a realitzar les vostres pròpies accions:

```
class Madre:
    def __init__(self):
        print(f"Soy Madre")

class Hijo(Madre):
    def __init__(self):
        print(f"Soy Hijo")


hijo = Hijo()
```

Ara bé, de vegades potser ens interessa no sobreescriure completament, sinó estendre una funcionalitat de la superclasse. Quan necessitem aquest comportament podem fer ús de la funció super(), un accés directe a la superclasse:

```
class Madre:
```

```
                def __init__(self):
        print(f"Soy Madre")

class Hijo(Madre):
    def __init__(self):
        super().__init__()
        print(f"Soy Hijo")



hijo = Hijo()
```

Queda clar aleshores que també és possible estendre el comportament d'una superclasse sense sobreescriure'l si fem ús de l'accessor super().

Què passa amb l'herència múltiple? Com es comportaria super() si hereda de més d'una superclasse?

```
class Madre:
    def __init__(self):
        print(f"Soy Madre")


class Padre:
    def __init__(self):
        print(f"Soy Padre")


class Hijo(Madre, Padre):
    def __init__(self):
        super().__init__()
        print(f"Soy Hijo")


hijo = Hijo()
```

En aquest cas, se segueix la lògica de la prioritat d'herència, tenint més prioritat la classe d'esquerra, i per tant super() és l'accessor de Mare.

Si volguéssim estendre el comportament del pare, en lloc de super() utilitzarem el seu nom:

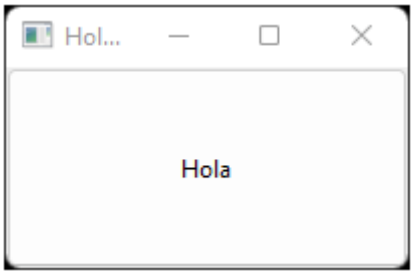
```
class Hijo(Madre, Padre):
    def __init__(self):
        Padre.__init__(self)
        print(f"Soy Hijo")
```

Per arrissar el ris res no ens impediria estendre el funcionament tant de la mare com del pare:

```
class Hijo(Madre, Padre):
```

```
        def __init__(self):
    Madre.__init__(self)
    Padre.__init__(self)
    print(f"Soy Hijo")
```

4. Primera aplicació fent us de POO

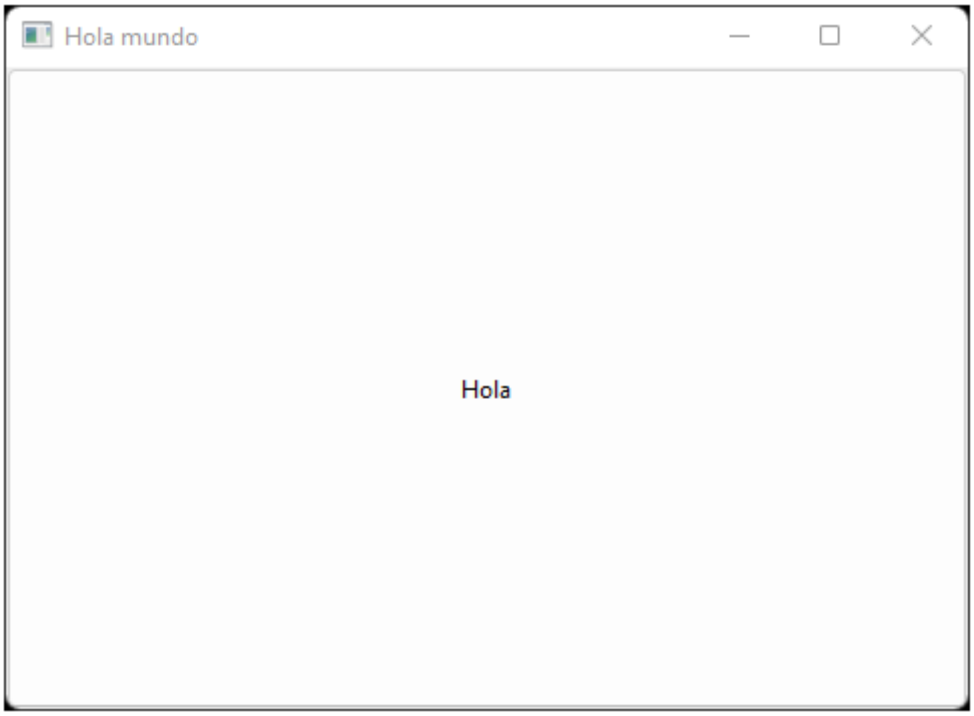



```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
import sys


class MainWindow(QMainWindow):

    """
    Creamos nuestra propia clase MainWindow heredando de QMainWindow
    """

    # Creamos el constructor de la clase
    def __init__(self):

        # Con super recogemos el comportamiento de QMainWindow
        super().__init__()

        # Damos un título al programa
        self.setWindowTitle("Hola mundo")

        # Guardamos el botón en una variable

      button = QPushButton("Hola")

        # Establecemos el botón como widget central de la ventana
principal
        self.setCentralWidget(button)


# "Ejecuta el código que hay dentro solo si este archivo se ejecuta
directamente, no si se importa desde otro modulo o archivo (Si
importas el archivo desde otro (import main) → no se ejecuta ese
bloque). Esto permite que se ejecute todo el código como programa
principal, o sólo la parte de arriba del código como módulo
reutilizable sin que se ejecute automáticamente la parte final.
"
```

```
if __name__ == "__main__":
    # Creamos la aplicación
    app = QApplication(sys.argv)
    # Creamos nuestra ventana principal
    window = MainWindow()
    # Mostramos la ventana
    window.show()
    # Iniciamos el bucle del programa
    app.exec()
```

La clau és estendre el funcionament del constructor de QMainWindow, ja que en executar super().__init__() heretem el seu comportament. Des d'aquest moment la nostra classe MainWindow es capaç d'adquir els mètodes heretats de QMainWindow com ara setWindowTitle i setCentralWidget.

5. Tamany dels widgets (QSize)



A Qt hi ha un objecte anomenat QSize que podem assignar als widgets per controlar el seu tamany. Pren una amplada i una alçada en píxels i es pot establir com a mida mínima, màxima o fixa per a un widget:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
from PySide6.QtCore import QSize # Nuevo
```

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Hola mundo")
        button = QPushButton("Hola")
        self.setCentralWidget(button)

        # Tamaño mínimo de la ventana
        self.setMinimumSize(QSize(480, 320))
        # Tamaño máximo de la ventana
        self.setMaximumSize(QSize(480, 320))

        # Tamaño fijo de la ventana
        self.setFixedSize(QSize(480, 320))


if __name__ == "__main__":
    app = QApplication()
    window = MainWindow()
    window.show()
    app.exec()
```

Alternativament, si no volem complicar-nos la vida podem utilitzar el mètode resize de la finestra:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Hola mundo")
        button = QPushButton("Hola")
        self.setCentralWidget(button)

        # Tamaño inicial de la ventana, permite redimensionar.
        self.resize(480, 320)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    app.exec()
```

6. Senyals i receptors (Signals i Slots)

Tenim un botó al nostre programa però no fa res.

A Qt per afegir una funcionalitat a un Widget necessitem connectar-lo a una acció, cosa que s'aconsegueix mitjançant senyals i slots.

Els senyals són notificacions emeses pels widgets quan passa alguna cosa. Per exemple, un botó envia un senyal de "botó polsat" quan un usuari hi fa clic.

Doncs bé, de poc serveix que un widget enviï un senyal si ningú n'és conscient. Per a aquest propòsit existeixen els receptors, coneguts com a slots:

```
# Definimos un receptor para conectar la señal clicked a un método
button.clicked.connect(self.boton_clicado)

def boton_clicado(self):
    print("¡Me has clicado!")
```

Vegem altres senyals dels botons per practicar:

```
# Pulsación y liberación
button.pressed.connect(self.boton_pulsado)
button.released.connect(self.boton_liberado)

def boton_pulsado(self):
    print("¡Me has pulsado!")

def boton_liberado(self):
    print("¡Me has liberado!")



# Señal para controlar el botón como un alternador true/false
button.setCheckable(True)
#La señal clicked de un botón checkable envía un argumento booleano
#(True o False) indicando su estado actual.
button.clicked.connect(self.boton_alternador)


def boton_alternador(self, valor):
    print("¿Alternado?", valor)
```

Amb això us podeu fer una idea de com els senyals estan pendents de tot el que passa sobre els widgets per permetre'ns actuar en conseqüència.

7. Components manipulables

Per acabar aquesta introducció veurem com manipular un widget.

Si volem accedir a un widget des d'un mètode és tan senzill com emmagatzemar un accés a aquest widget a la pròpia instància, és a dir, hem de generar un punter al mateix widget:

```
# generem punter al button per a actualitzar el valor del botó en cada
moment
self.button = button
```

Una volta anomenat el punter, podem fer referència a qualsevol instància d'un widget per modificar-la a voluntat:

```
def boton_alternador(self, valor):
    if valor:
        self.button.setText("Estoy activado")

    else:
        self.button.setText("Estoy desactivado")
```

Fem un darrer exemple usant un altre component:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QLineEdit  #
editado
from PySide6.QtCore import QSize

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Hola mundo")
        self.setMinimumSize(QSize(480, 320))

        # widget input de texto
        texto = QLineEdit()
        # capturamos la señal de texto cambiado
        texto.textChanged.connect(self.texto_modificado)

        # establecemos el widget central
        self.setCentralWidget(texto)

        # creamos el puntero
        self.texto = texto

    def texto_modificado(self):
        # recuperasmo el texto del input
        texto_recuperado = self.texto.text()
```

```
        # modificamos el título de la ventana al vuelo
        self.setWindowTitle(texto_recuperado)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Respecte als esdeveniments:

- Cada interacció de l'usuari amb la interfície, per exemple, un clic de ratolí, una pulsació de tecla... generarà un esdeveniment. Aquest esdeveniment serà afegit a una cua d'esdeveniments (**event queue**) per a ser gestionat.

- El bucle d'esdeveniments (**event loop**), que és un bucle infinit, comprovarà en cada iteració si hi ha esdeveniments pendents de ser gestionats. En cas de que sí, l'esdeveniment serà gestionat pel gestor d'esdeveniments (event handler) que executarà el seu "manejador".

- El bucle d'esdeveniments serà gestionat per l'objecte **QApplication** i llançarà a l'executar el mètode **exec()** d'aquest.

8. Credits

Aquests apunts són un material de suport per als alumnes de DAM i han sigut desenvolupats per Herctor Docs i publicats sota una Llicència CC BY 4.0. Estos apunts han sigut modificats i adaptats per Iván Martos.