

Si no sabes poner el .py del QTDesigner apaga y vamonos a la mierda.

Una vez que llegamos a este punto, vamos a escribir el código mínimo necesario para lanzar la ventana. En el fichero main.py codificamos lo siguiente:

Analicemos el código:



```
1 from ventana import *
2 import sys
3
4
5 class Main(QtWidgets.QMainWindow):
6
7     def __init__(self):
8         super(Main, self).__init__()
9         self.ui = Ui_MainWindow()
10        self.ui.setupUi(self)
11
12
13 if __name__ == '__main__':
14     app = QtWidgets.QApplication([])
15     window = Main()
16     window.show()
17     sys.exit(app.exec())
18
```

1.Dentro del script main.py empezaremos importando todo lo que hay en el script ventana.py y que utilizaremos posteriormente.

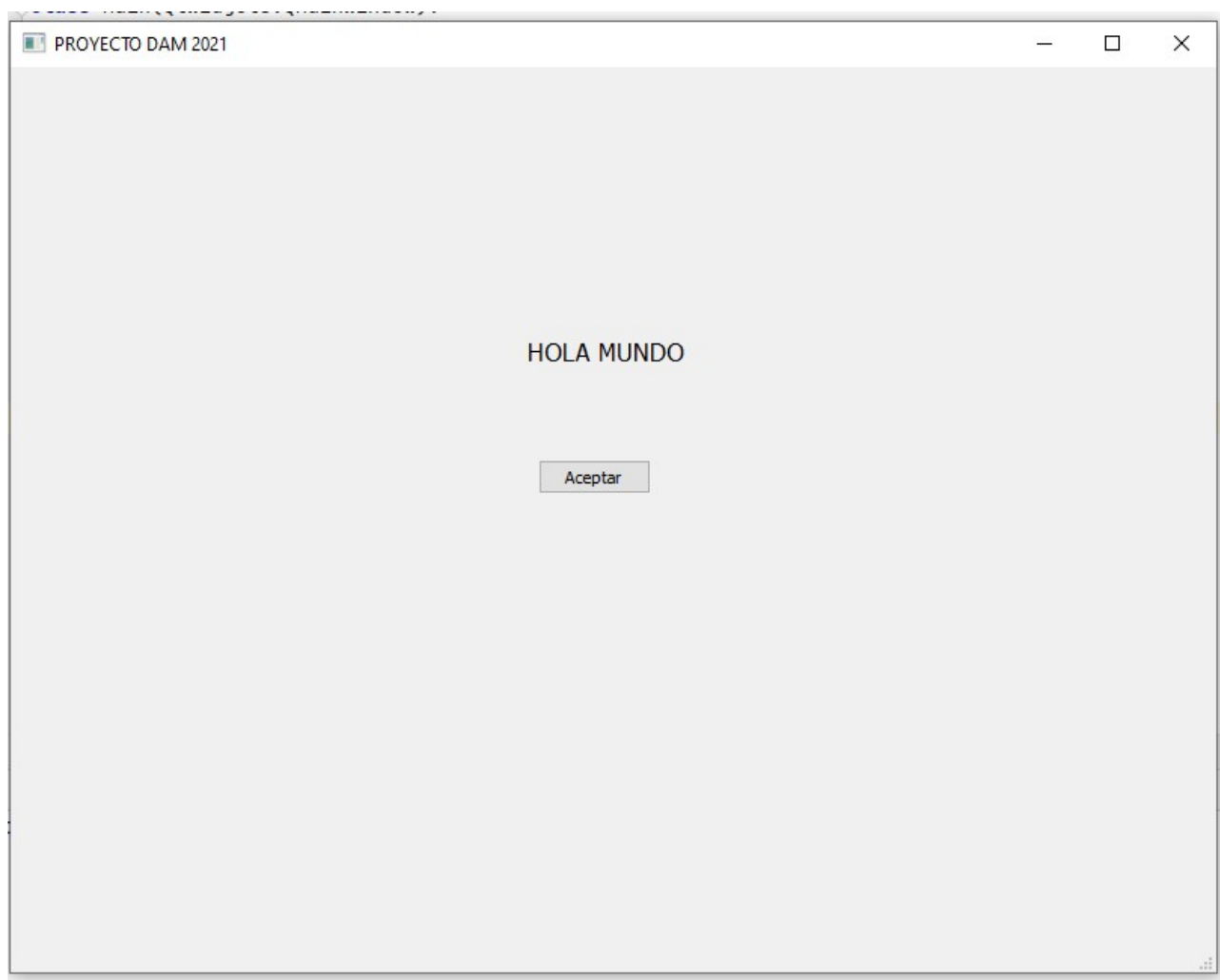
2.A continuación vamos a crear una nueva clase que representará nuestra ventana principal (MainWindow), pero no la crearemos desde cero, vamos a heredar del Widget QMainWindow de PyQt que encontraremos en el módulo QtWidgets. Este ha sido importado previamente en el script ventana.py y por lo que ya está cargado en memoria. Se renderiza la estructura y para ello sobreescribiremos el constructor y llamaremos al método setupUi encargado de generar la interfaz. Como se puede observar se utiliza herencia, sobreescritura de métodos y otros conceptos de POO.

3.En el tercer bloque con la comprobación de la primera línea estamos añadiendo una cláusula por la que el contenido del if sólo se ejecutará al llamar el propio script en la terminal. Sirve básicamente para evitar ejecuciones duplicadas en caso de que importe el fichero ventana.py en otro script. A continuación creamos una aplicación de PyQt con el widget QApplication, al que le pasaremos una lista vacía. Normalmente aquí se pasarían argumentos de la línea de

comandos, pero en las aplicaciones gráficas no se suelen usar. Sólo asignaremos su ejecución a una variable `app` que nos permitirá controlar el bucle del programa.

A continuación crearemos una instancia de nuestra `MainWindow` y la mostraremos con su método `show()`. Finalmente pondremos en marcha el bucle del programa con `app.exec_()`. Si no ponemos esta última línea el programa se iniciará, pero se cerrará inmediatamente.

El resultado sería:



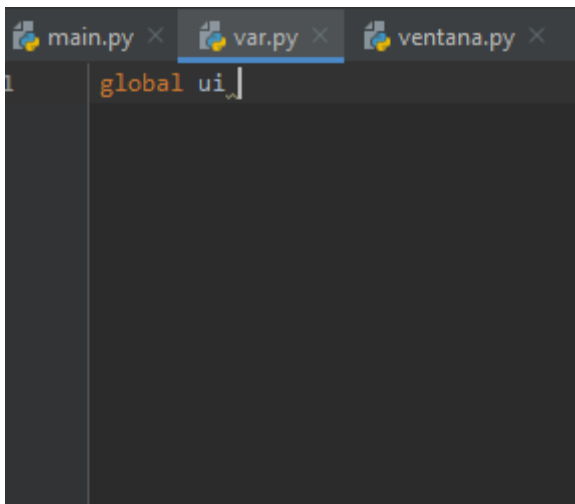
La **programación orientada a eventos (POE)** es un paradigma de programación en el que tanto la estructura como la ejecución de los programas están determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen (Wikipedia).

Una vez lanzado el programa su continuación estará bloqueada hasta que se produzca algún evento. Cuando uno de esos eventos se inicia solo se ejecutará el código asociado al mismo. Dicha ejecución dependerá del **lenguaje de programación** utilizado, del **sistema operativo** donde esté instalada la aplicación y del propio **código** creado por el programador.

La POE es la base de las interfaces gráficas de usuario y que han permitido acercar de forma intuitiva el software al usuario inexperto, así como un aumento de la productividad en muchas tareas que realizamos con los ordenadores. Hoy en día, la POE está presente en casi todas las aplicaciones que utilizamos.

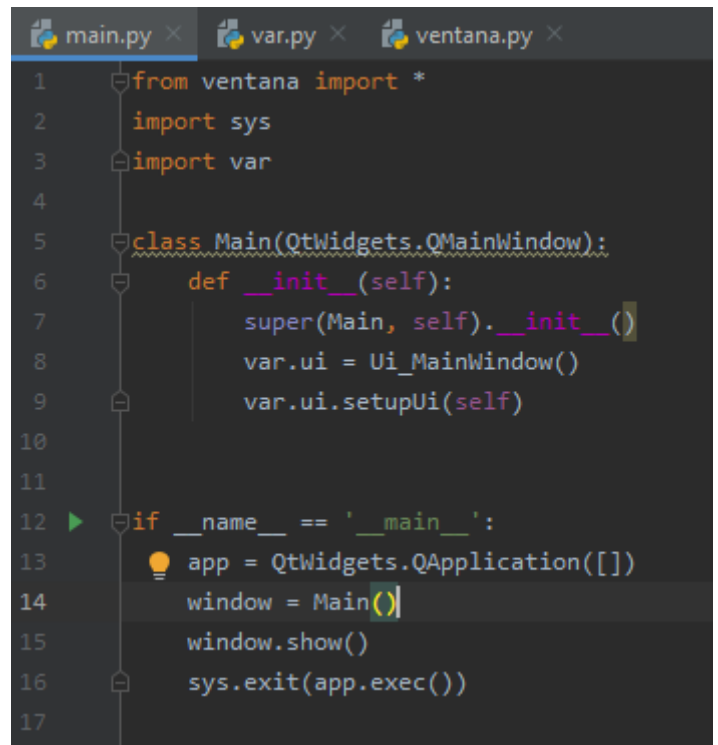
En el ejemplo con el que trabajamos hasta ahora vimos que al pulsar el botón el programa no hace nada. Es obvio, el botón no tiene asociado ningún código que genere evento alguno. Para entender el mecanismo, veamos un ejemplo muy simple, al pulsar el botón el texto del *label* o etiqueta cambiará por otro.

Antes de seguir, vamos realizar algunos cambios en el ejemplo que estamos desarrollando. En primer lugar, vamos a crear un fichero llamado **var.py** donde iremos almacenando las variables que vayamos utilizando. Esto es útil para cuando tengamos que acceder a dichas variables situadas en ficheros diferentes. Nos quedaría de lo forma siguiente.



```
1 global ui
```

Declaramos la variable **ui**



```
1 from ventana import *
2 import sys
3 import var
4
5 class Main(QtWidgets.QMainWindow):
6     def __init__(self):
7         super(Main, self).__init__()
8         var.ui = Ui_MainWindow()
9         var.ui.setupUi(self)
10
11
12 if __name__ == '__main__':
13     app = QtWidgets.QApplication([])
14     window = Main()
15     window.show()
16     sys.exit(app.exec())
17
```

Importamos el módulo y cambiamos las referencias de **ui**

En segundo lugar, creamos otro fichero que contiene los módulos que gestionan los eventos, al que llamaremos **events.py**. Dependiendo de la complejidad del proyecto, si este tuviese muchos eventos, y suele ser lo habitual, sería necesario crear nuevos ficheros que almacenasen el código de los eventos agrupándolos por categorías, por ejemplo, eventos relacionados con los clientes, eventos relacionados con servicios o artículos y así sucesivamente. De momento, vamos a crear uno solo.

```
main.py x events.py x var.py x ventana.py x
import var

class Eventos():

    def Saludo():
        try:
            var.ui.lblHola.setText('Has pulsado el Botón')
        except Exception as error:
            print('Error: %s ' % str(error))
```

```
main.py x events.py x var.py x ventana.py x
1 from ventana import *
2 import sys
3 import var,events
4
5 class Main(QtWidgets.QMainWindow):
6     def __init__(self):
7         super(Main, self).__init__()
8         var.ui = Ui_MainWindow()
9         var.ui.setupUi(self)
10        ...
11        conexión con los eventos
12        ...
13        var.ui.btnAceptar.clicked.connect(events.Eventos.Saludo)
14
15
16 if __name__ == '__main__':
17     app = QtWidgets.QApplication([])
```

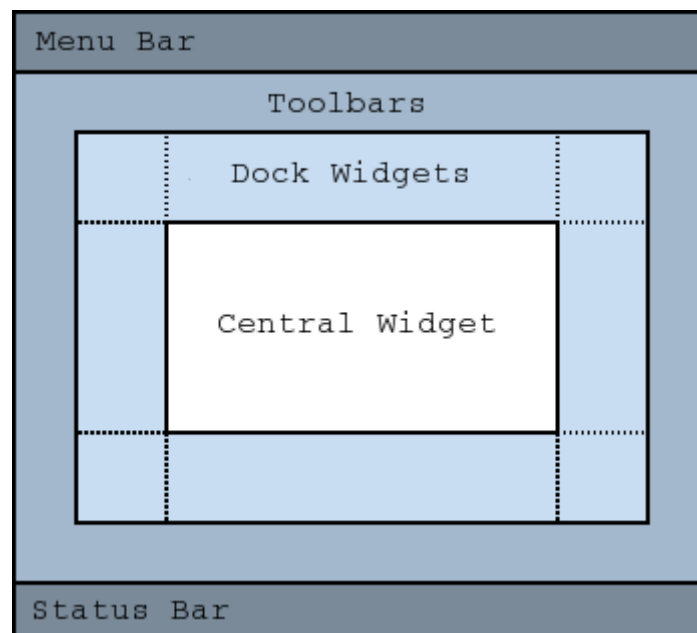
En la primera imagen vemos el contenido del fichero **events.py** donde observamos que aparece el nombre del label o etiqueta ¿de donde sale eso? Cuando diseñamos con Qt-Designer este fue el nombre que elegimos para el widget. Dicho nombre se mantiene cuando realizamos la conversión del fichero **.ui** a **.py**, solo tenemos que consultar el código de **ventana.py**. Además, en el fichero **main.py** tenemos la sentencia que nos carga en la variable **var.ui** la ventana principal que contiene todos los widgets. Recordemos que es una estructura "arbórea" que se va construyendo a partir de un fichero XML.

En la segunda imagen, observamos la importación de fichero **events.py**. De la misma forma que con la etiqueta lblHola, también accedemos al control btnAceptar. Este botón, que como se ha dicho, es un control, utiliza el método clicked para conectarse con el fichero events.py que contiene la clase Eventos que a us vez contiene la función Saludo. La idea que subyace es intuitiva y es la base de la POE. Este mecanismo básico es igual en todos los lenguajes de programación asociados a IDE que diseñan interfaces gráficas de usuario. Al ejecutar este simple ejemplo y pulsando el botón observaremos como el contenido de la etiqueta cambia.

•Profundizando en el diseño

Esta actividad solo tenía por objetivo ver el funcionamiento de la POE a grandes rasgos. Sin embargo, debemos tener en cuenta que este módulo tiene como fin principal el diseño de interfaces de acuerdo a las **reglas de usabilidad** que se ven en la teoría. Para ello vamos a rediseñar totalmente la

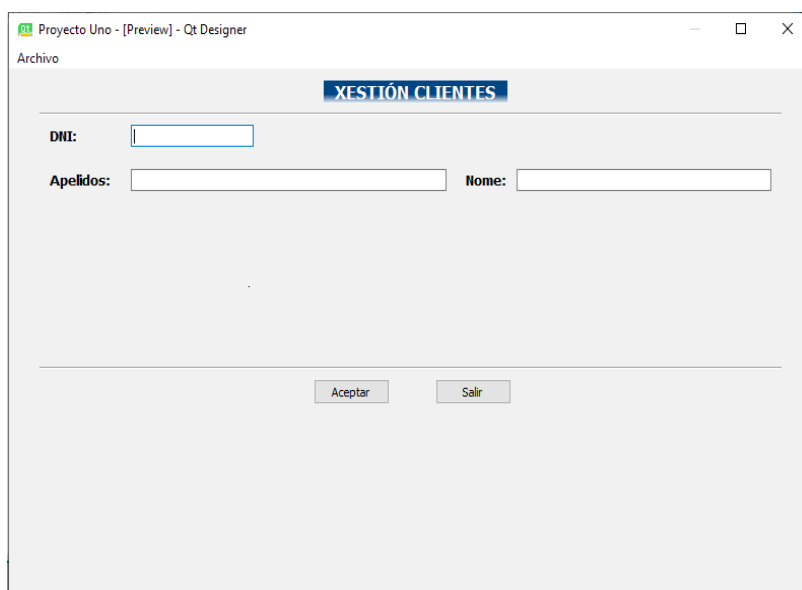
ventana adecuándola a dichas reglas. Observemos la siguiente imagen. Es la disposición típica de una ventana de un aplicación:



Menubar: contiene los menús Archivo, Ver, Acerca de...

- **Toolbars:** contiene la barra de iconos para acceso rápido a las principales funciones
- **Dock Widgets:** no siempre aparece. En este [enlace](#) podemos ver su objetivo. Apenas lo utilizaremos.
- **Central Widgets:** donde se sitúan los widgets de la aplicación. Para **activarlo, una vez tenga widgets en su interior**, pulsamos **Ctrl+1**. Sin embargo, es mejor dejarlo al principio sin activar para poder tener libertad en la disposición de los widgets.

Intentemos crear la siguiente ventana.



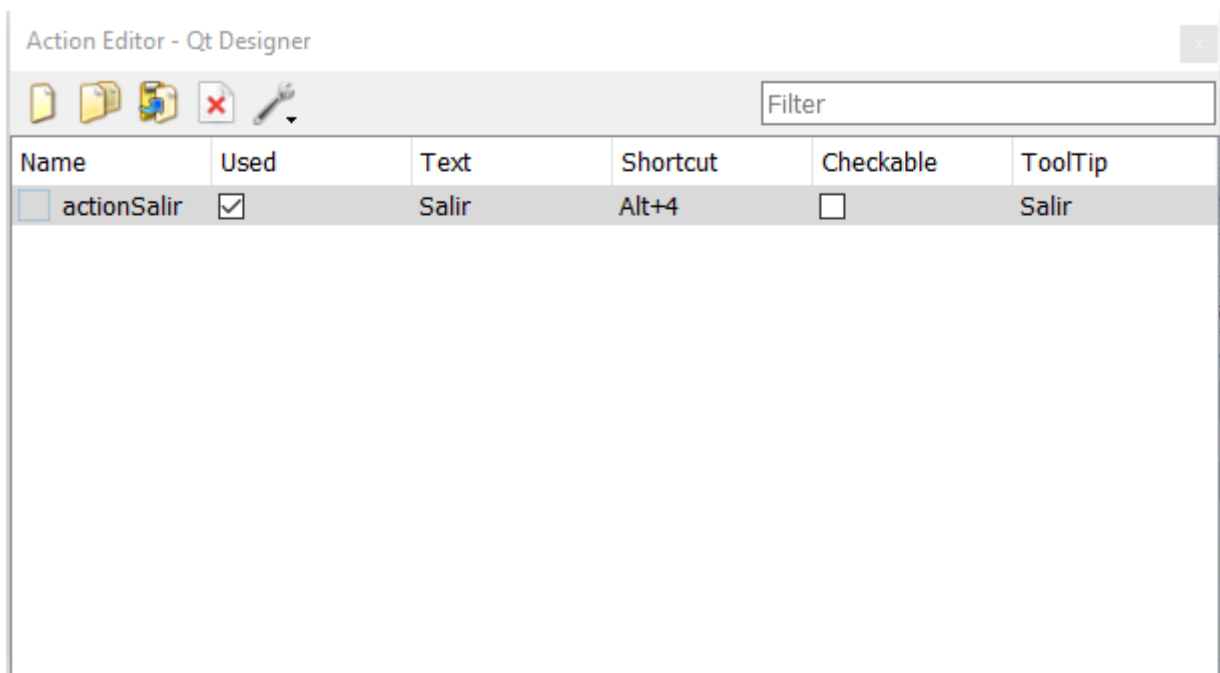
Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
btnAceptar	QPushButton
btnSalir	QPushButton
cmbProv	QComboBox
editApel	QLineEdit
editDir	QLineEdit
editDni	QLineEdit
editNome	QLineEdit
horlaySex	QHBoxLayout
rbtFem	QRadioButton
rbtMasc	QRadioButton
horlayPago	QHBoxLayout
chkEfectivo	QCheckBox
chkTarjeta	QCheckBox
chkTransf	QCheckBox
lblApel	QLabel
lblDir	QLabel
lblDni	QLabel
lblMetpago	QLabel
lblNome	QLabel
lblProv	QLabel
lblSexo	QLabel
lblTitCli	QLabel
lblValido	QLabel
lineInf	Line
lineSup	Line
menubar	QMenuBar
menuArchivo	QMenu
actionSalir	QAction
statusBar	QStatusBar

Los label o etiquetas no tienen ningún secreto. Son un elemento que nos permite información.

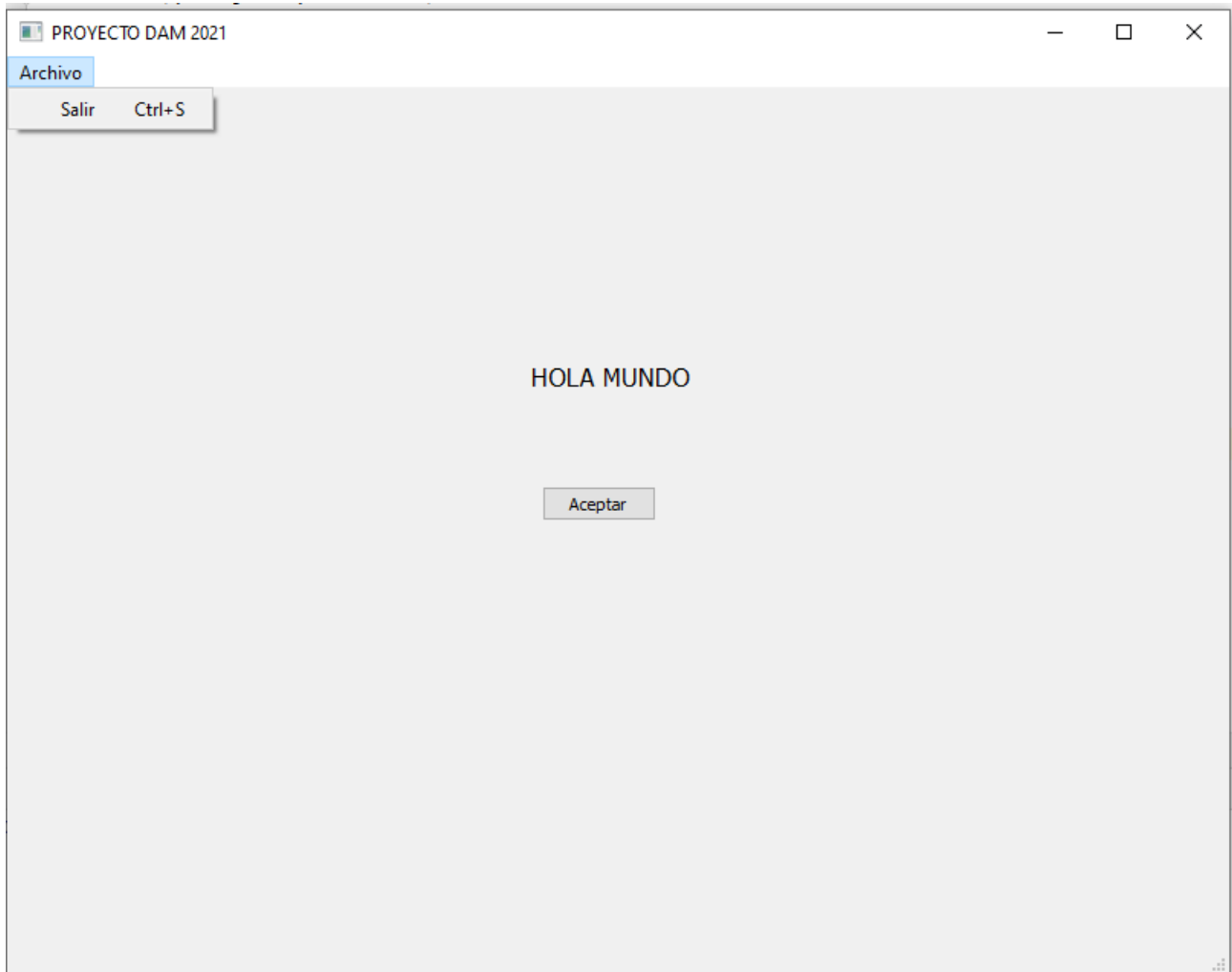
El cuadro de texto que usaremos el **LineEdit**. Es el widget input más básico y el que se utiliza en la mayoría de los casos. Sin embargo. QtDesigner tiene algún tipo más:

- TextEdit**: a diferencia del anterior es multilínea a modo de editor de texto básico.
- Plain TextEdit**: es igual que el anterior pero mejora el formateo del texto.

Para realizar esta miniaplicación configuraremos propiedades como ***width, height, stylesheet y textarea*** en las etiquetas y los botones. Además tocamos el menubar tal como se indica en la imagen. En dicho menú vamos a incluir la opción Salir con una combinación de teclas, por ejemplo, ***Crtl+S***. Para ello vamos a la opción **View -> Action Editor**, y hacemos las modificaciones de la imagen. Además modificamos un par de líneas del código (ver imagen derecha).



Quedando algo así:



Ya solo queda probar la salida de la aplicación desde la Barra de Menús y desde el símbolo X de la ventana añadiendo los códigos siguientes:

```
main.py x events.py x var.py x ventana.py x
1 from ventana import *
2 import sys, var, events
3
4
5 class Main(QtWidgets.QMainWindow):
6
7     def __init__(self):
8         super(Main, self).__init__()
9         var.ui = Ui_MainWindow()
10        var.ui.setupUi(self)
11        var.ui.actionSalir.triggered.connect(events.Eventos.Salir)
12
13
14 if __name__ == '__main__':
15     app = QtWidgets.QApplication([])
16     window = Main()
17     window.show()
18     sys.exit(app.exec())
19
```

```
main.py x events.py x var.py x ventana.py x
1 import sys
2
3
4 class Eventos:
5
6     def Salir():
7         try:
8             sys.exit()
9         except Exception as error:
10            print("Error %s: " % str(error))
11
```

Para finalizar esta unidad, demoslo a la aplicación una utilidad adicional. Como vemos en la ventana la aplicación nos pide el DNI. Pues bien, la idea es que el valor que nos pasen sea controlado por la aplicación para ver si es correcto.

Empecemos por el IU (ver las imágenes más abajo). Colocaremos una etiqueta a la derecha de la caja de texto del DNI, usando el tipo de letra **Forte**, tamaño 14 y sin texto alguno. Ya veremos posteriormente como lo usamos.

En el código de la aplicación en primer lugar, vamos a crear un nuevo fichero llamado **clients.py** que almacenará aquellas funciones que estén relacionadas con la gestión de los clientes. En el mismo añadimos el siguiente código.

```
main.py x events.py x clients.py x var.py x ventana.py x
1
2 class Clients():
3     def validarDNI(dni):
4         """
5         código tomado de la web perezmartin.es
6         """
7
8         try:
9             tabla = "TRWAGMYFPDXBNJZSQVHLCKE" # letras del dni, esta lista estándar
10            dig_ext = "XYZ" # tabla letras extranjero
11            reemp_dig_ext = {'X': '0', 'Y': '1', 'Z': '2'} # letras que identifican extranjero cambiadas por nº
12            numeros = "1234567890"
13            dni = dni.upper() # pasa letras a mayúsculas
14            if len(dni) == 9: # el dni debe tener 9 caracteres
15                dig_control = dni[8] # tomo la letra
16                dni = dni[:8] # tomo el número que son los 8 primeros
17                if dni[0] in dig_ext: # comprueba que es extranjero y si es cambia por número
18                    dni = dni.replace(dni[0], reemp_dig_ext[dni[0]])
19                return len(dni) == len([n for n in dni if n in numeros]) and tabla[int(dni) % 23] == dig_control
20                # devuelve true si se dan las 2 condiciones y sale del módulo o sino false y sale del módulo
21            return False # sino devuelve falso
22        except:
23            print('error en la aplicación')
24            return None
```


El paso siguiente es asociar el evento que queremos utilizar con esta función. La idea es que cuando el cursor salga de la caja de texto del dni, se ejecute el código expuesto. Para ello añadimos unas líneas al módulo `main.py` y `events.py` (o `clients.py`).

```
#var.ui.btnAceptar.clicked.connect(events.Eventos.AltaCli)
var.ui.btnSalir.clicked.connect(events.Eventos.Salir)
var.ui.actionSalir.triggered.connect(events.Eventos.Salir)
var.ui.editDni.editingFinished.connect(events.Eventos.validoDNI)|

name == ' main ':
app = QtWidgets.QApplication([])
```

```
'''
eventos clientes
'''

def validoDNI():
    try:
        dni = var.ui.editDni.text()
        if clients.Clients.validarDNI(dni):
            var.ui.lblValido.setStyleSheet('QLabel {color: green;}')
            var.ui.lblValido.setText('V')
            var.ui.editDni.setText(dni.upper())
        else:
            var.ui.lblValido.setStyleSheet('QLabel {color: red;}')
            var.ui.lblValido.setText('X')
            var.ui.editDni.setText(dni.upper())

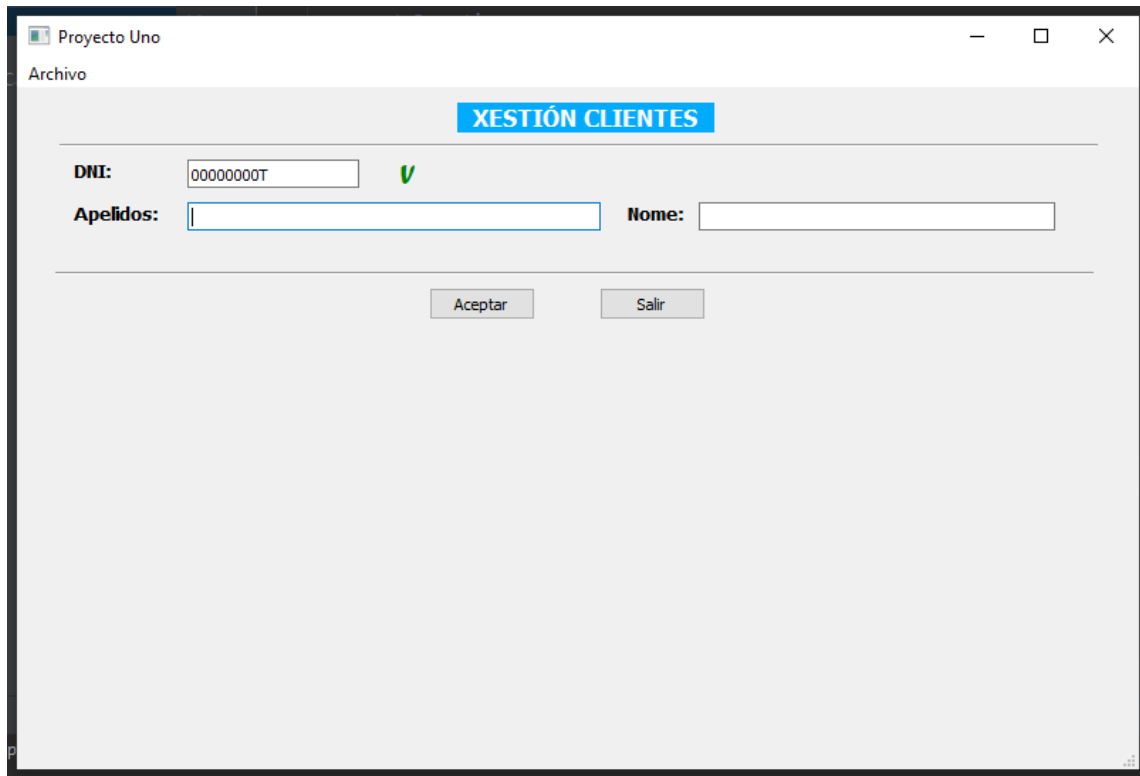
    except Exception as error:
        print('Error: %s ' % str(error))
```

main.py: conectamos el evento **editingFinished** editDni para que cuando el cursor salga de la caja de texto del deni se ejecute **validoDNI** en **events.py**

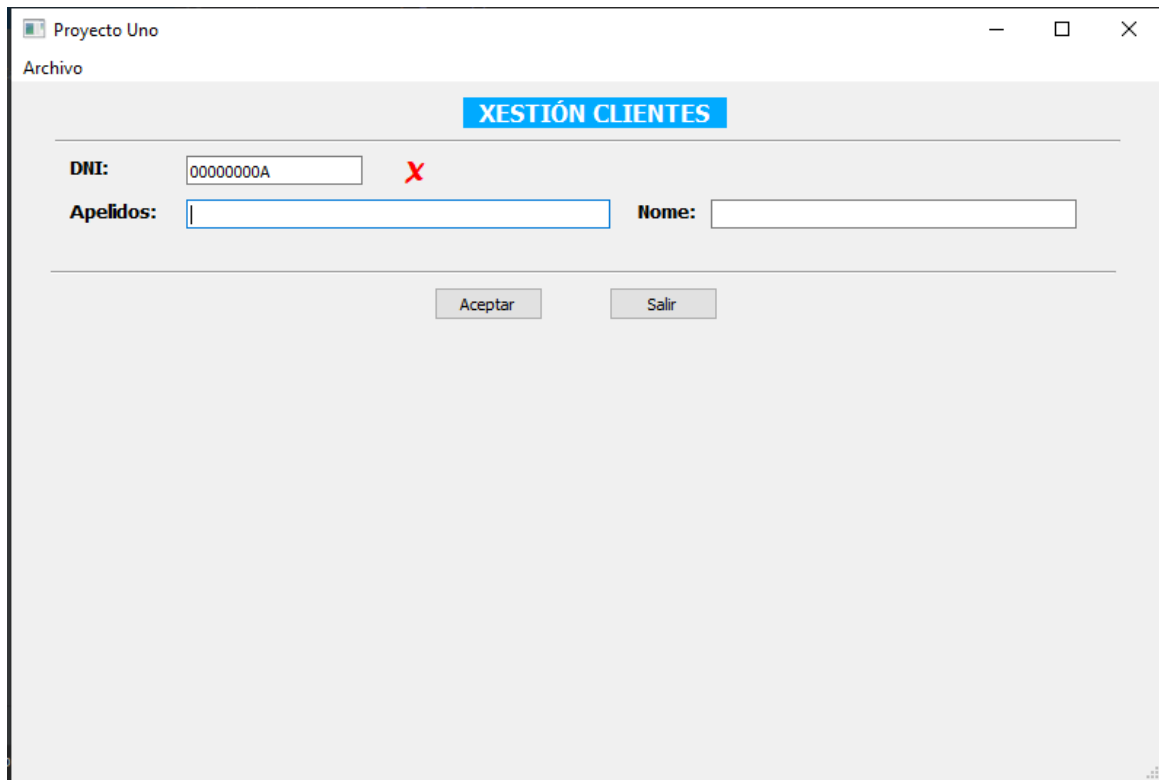
- events.py** (o en un módulo **clients.py**): este evento primero coge el valor **dni** introducido en la caja de texto con la función **.text()** y se lo pasamos a la función **validarDNI** en **clients.py**. Esta función nos devuelve **True** o **False**. Si nos devuelve True escribiremos una V en color verde (estilos), si nos devuelve False escribiremos una X en color

rojo. En ambos casos, además lo pone en mayúscula con la función **setText** y **upper()**.

Lo vemos el resultado en los dos casos:



The screenshot shows a window titled 'Proyecto Uno' with a menu bar containing 'Archivo'. The main content area has a title bar 'XESTIÓN CLIENTES'. Below this, there are three input fields: 'DNI:', 'Apellidos:', and 'Nome:'. The 'DNI:' field contains the text '00000000T' and has a green checkmark icon to its right. The 'Apellidos:' and 'Nome:' fields are empty. At the bottom of the form, there are two buttons: 'Aceptar' and 'Salir'.



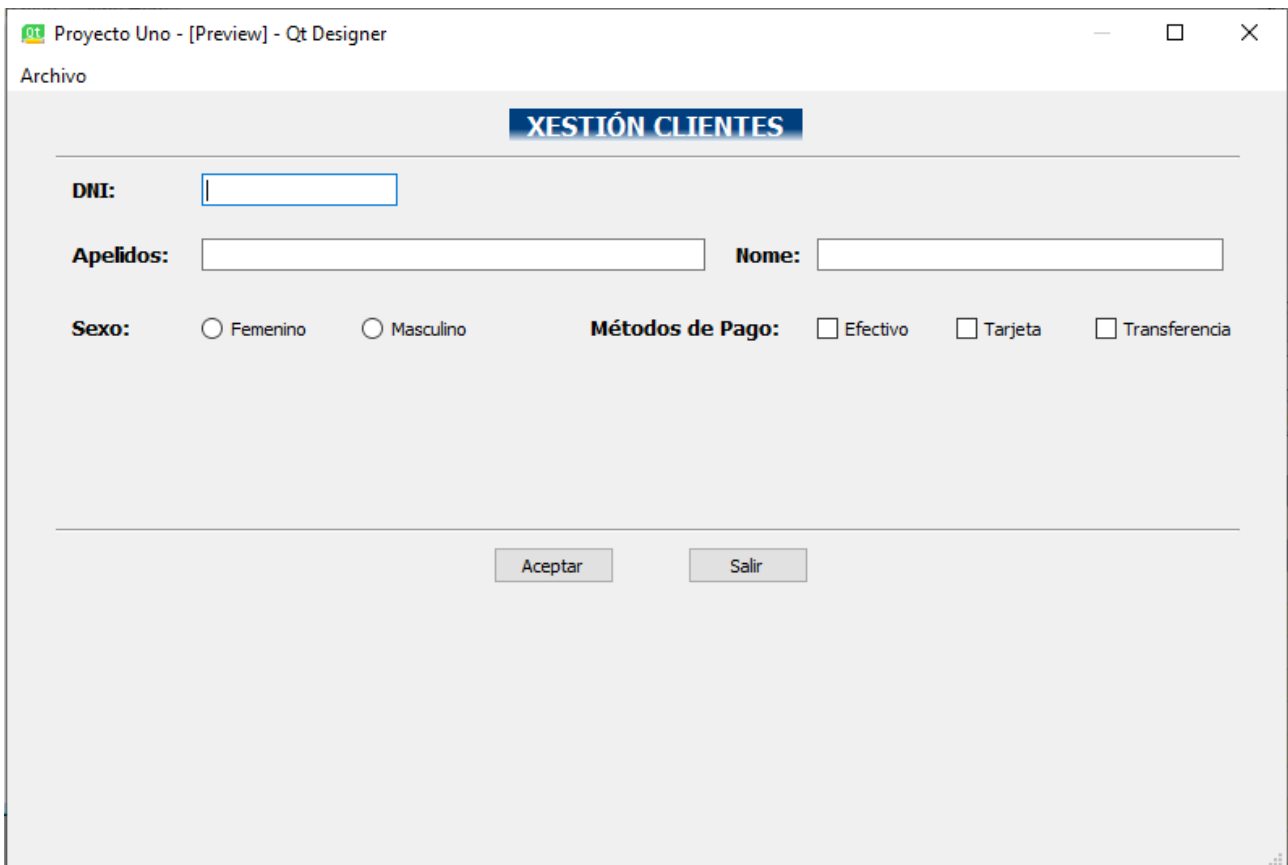
The screenshot shows the same window as above, but with the 'DNI:' field containing the text '00000000A'. A red 'X' icon is now to the right of the field, indicating an invalid entry. The 'Apellidos:' and 'Nome:' fields remain empty, and the 'Aceptar' and 'Salir' buttons are still at the bottom.

Hasta aquí la unidad tres. En la siguiente vamos a introducirnos en el uso de radiobutton, checkbox y combobox.

RadioButton, Checkbox, Combobox

Los **RadioButton** y los **Checkbox** son dos de los widgets más utilizados en los formularios. En primer lugar se diferencian por su forma, los primeros **son pequeños círculos** y los segundos por su **pequeños cuadrados**. Pero su diferencia fundamental es la función, mientras los **RadioButton** los utilizamos cuando queremos **seleccionar un solo elemento entre varios** elementos de un grupo, los **Checkbox** se utilizan cuando **necesitamos tener uno o más elementos** pueden ser seleccionados. Su respuesta siempre es una variable booleano.

Veamos el siguiente ejemplo,



The screenshot shows a Qt Designer window titled 'Proyecto Uno - [Preview] - Qt Designer'. The form is titled 'GESTIÓN CLIENTES' and contains the following fields and controls:

- DNI:** A text input field.
- Apellidos:** A text input field.
- Nombre:** A text input field.
- Sexo:** Two radio buttons labeled 'Femenino' and 'Masculino'.
- Métodos de Pago:** Three checkboxes labeled 'Efectivo', 'Tarjeta', and 'Transferencia'.
- Buttons:** Two buttons at the bottom labeled 'Aceptar' and 'Salir'.

un cliente puede ser varón o mujer, pero no ambos, pero si puede utilizar diferentes tipos de métodos de pago según su deseo.

Algunas ideas sobre este diseño:

- utilizar layouts horizontales para agruparlos
- usad nombres intuitivos: rbtFem, rbtMasc, chkEfec, chkTarj, chkTrans

Veamos las líneas de código que incluyen los eventos más importantes para el manejo de estos widgets.

En el caso de los radiobutton.

```
conexión eventos botones
'''

#var.ui.btnAceptar.clicked.connect(events.Eventos.AltaCli)
var.ui.btnSalir.clicked.connect(events.Eventos.Salir)
var.ui.actionSalir.triggered.connect(events.Eventos.Salir)
var.rbtsex = (var.ui.rbtFem, var.ui.rbtMasc)
for i in var.rbtsex:
    i.toggled.connect(events.Eventos.selSexo)

'''

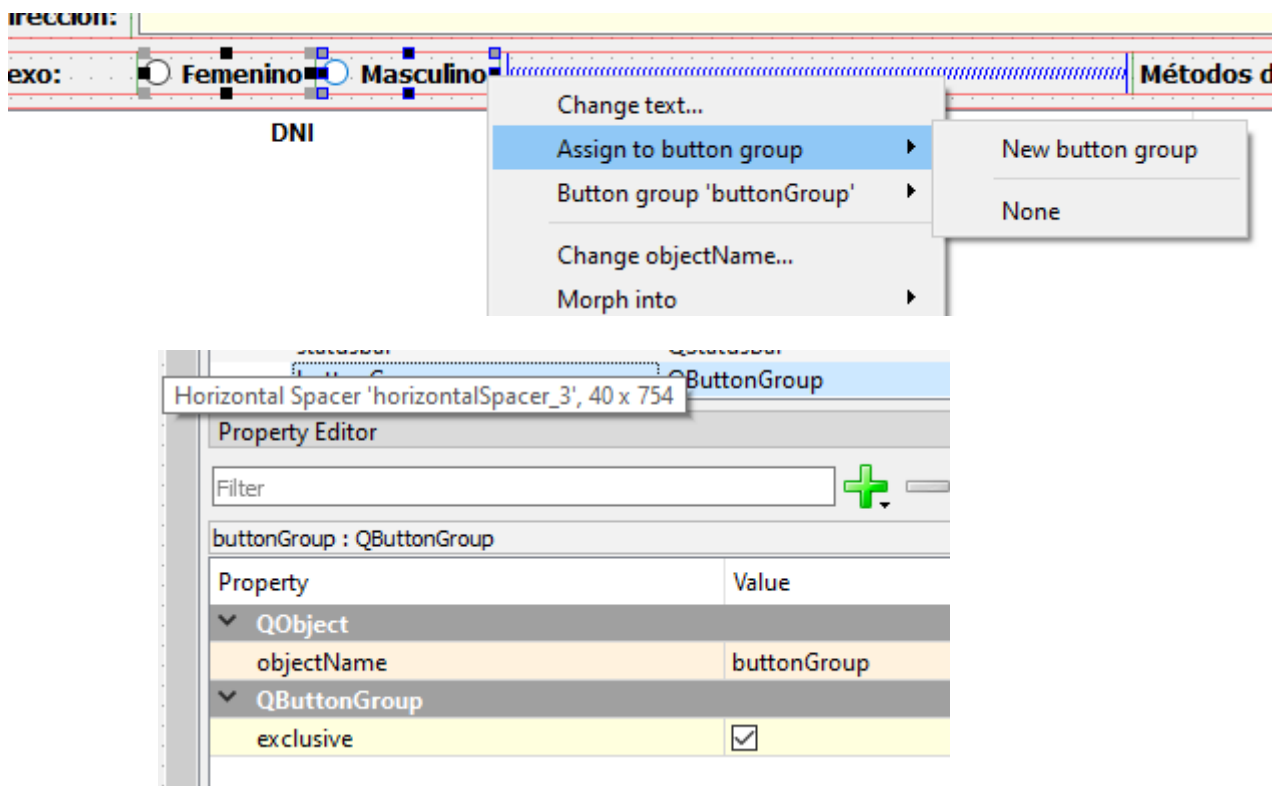
conexión eventos input widgets
'''
```

toggled es el evento que debemos llamar

```
def selSexo():
    try:
        if var.ui.rbtFem.isChecked():
            var.ui.lblPrueba.setText('femenino')
        if var.ui.rbtMasc.isChecked():
            var.ui.lblPrueba.setText('masculino')
    except Exception as error:
        print('Error: %s ' % str(error))
```

isChecked es el método

Para facilitar el manejo de los **radiobutton** se agrupan en los **QButtonGroup**. Es widget, pero que no aparece desde PyQt4 en la tabla. Para ello nos situamos en el diseño marcamos los dos **radiobutton** y haciendo click con el botón derecho los añadimos a un **QButtonGroup**.



En el caso de los checkbox.

```
for i in var.rbtsex:
    i.toggled.connect(events.Eventos.selSexo)
var.chkpago = (var.ui.chkEfectivo, var.ui.chkTarjeta, var.ui.chkTransf)
for i in var.chkpago:
    i.stateChanged.connect(events.Eventos.selPago)
...
```

State changed es el evento

```
def selPago():
    try:
        if var.ui.chkEfectivo.isChecked():
            print('pagas con efectivo')
        if var.ui.chkTarjeta.isChecked():
            print('pagas con tarjeta')
        if var.ui.chkTransf.isChecked():
            print('pagas con transferencia')
    except Exception as error:
        print('Error: %s ' % str(error))
```

isChecked es el método

El tercer widget, y uno de los más utilizados, es el **combobox o cuadro combinado**. Tradicionalmente, es una **combinación entre una lista desplegable y un cuadro de texto**, que permite al usuario introducir cualquier valor directamente en el teclado o, alternativamente, seleccionar un valor de la lista. El término "lista desplegable" se utiliza a veces para significar "cuadro combinado", aunque no todos los lenguajes lo consideran de la misma forma.

Esta primera aproximación al cuadro combinado será muy simple. En aplicaciones más desarrolladas los cuadros combinados suelen cargar sus datos desde bases de datos. Por ejemplo, cuando seleccionas una provincia en un cuadro combinado se lanza un evento que carga todos los municipios de esa provincia en un cuadro combinado.

Modifiquemos nuestra IU para obtener el siguiente resultado:

Object	Class
▼ MainWindow	QMainWindow
▼ centralwidget	QWidget
btnAceptar	QPushButton
btnSalir	QPushButton
cmbProv	QComboBox
editApel	QLineEdit
editDir	QLineEdit
editDni	QLineEdit
editNome	QLineEdit
▼ horlaySex	QHBoxLayout
rbtFem	QRadioButton
rbtMasc	QRadioButton
▼ horlayPago	QHBoxLayout
chkEfectivo	QCheckBox
chkTarjeta	QCheckBox
chkTransf	QCheckBox
lblApel	QLabel
lblDir	QLabel
lblDni	QLabel
lblMetpago	QLabel
lblNome	QLabel
lblProv	QLabel
lblSexo	QLabel
lblTitCli	QLabel
lblValido	QLabel
lineInf	Line
lineSup	Line
▼ menubar	QMenuBar
▼ menuArchivo	QMenu
actionSalir	QAction
statusBar	QStatusBar

A la izquierda observamos los cambios de la interfaz gráfica la inclusión de una caja de texto para introducir la dirección (obviamos cajas de texto para número de vivienda, piso, bloque, etc... por simplificar). Además supondremos que será en Galicia con lo que incluiremos en el cuadro combinado de la derecha solamente las provincias gallegas.

A la derecha un esquema de los widgets introducidos hasta el momento. Obsérvese el nombre intuitivo que se pretende dar en cada caso.

Ahora trabajemos con el código, concretamente con el combobox. En las siguientes imágenes vemos los códigos necesarios:

```
i.stateChanged.connect(clients.Clients.selPago)

'''
conexión eventos input widgets y otras funciones
'''

clients.Clients.cargarProv()
var.ui.cmbProv.activated[str].connect(clients.Clients.selProv)
var.ui.editDni.editingFinished.connect(events.Eventos.validoDNI)

if __name__ == '__main__':
```

Líneas de código añadidos en main.py

```
def cargarProv():
    '''
    esta solución es provisional en su momento lo haremos de otra forma
    cargando los registros desde una base de datos
    '''
    try:
        prov = ['A Coruña', 'Lugo', 'Ourense', 'Pontevedra']
        for i in prov:
            var.ui.cmbProv.addItem(i)
    except Exception as error:
        print('Error: %s' % str(error))
```

Función cargarProv

```
def selProv(prov):
    try:
        print('Has seleccionado la provincia de ', prov)
        return prov
    except Exception as error:
        print('Error: %s ' % str(error))
```

Función selProv

En el fichero **main.py** añadimos dos líneas:

- la primera **clients.Clients.cargarProv()** es una función que se lanza cuando arranca la aplicación. Su función es cargar el cuadro combinado o combobox.
- la segunda llama al evento **activated** este evento tiene como función cargar en una variable el valor seleccionado del combobox, de ahí que haya una tupla con un **string**.

En el fichero **clients.py** tenemos dos funciones:

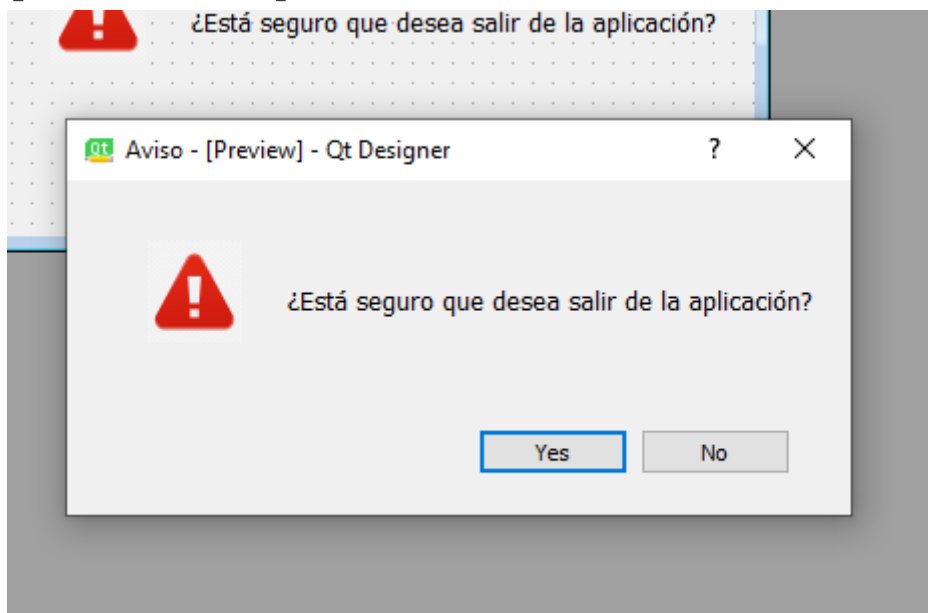
- la primera función **cargarProv** es encarga de cargar en el combobox el listado de provincias. Como ya se indica es una situación provisional. Más adelante veremos como cargar funciones pero desde la base de datos. Por ejemplo, las provincias de toda España sin necesidad de tener que cargar una a una en una tupla o lista.
- la segunda función **selProv** nos devuelve la provincia seleccionada.

Hasta aquí hemos visto tres de los widgets más comunes en las interfaces gráficas de usuario. Indicar que hemos establecido los conceptos básicos de su funcionamiento, en especial, del combobox, control que nos dará más juego cuando veamos las bases de datos.

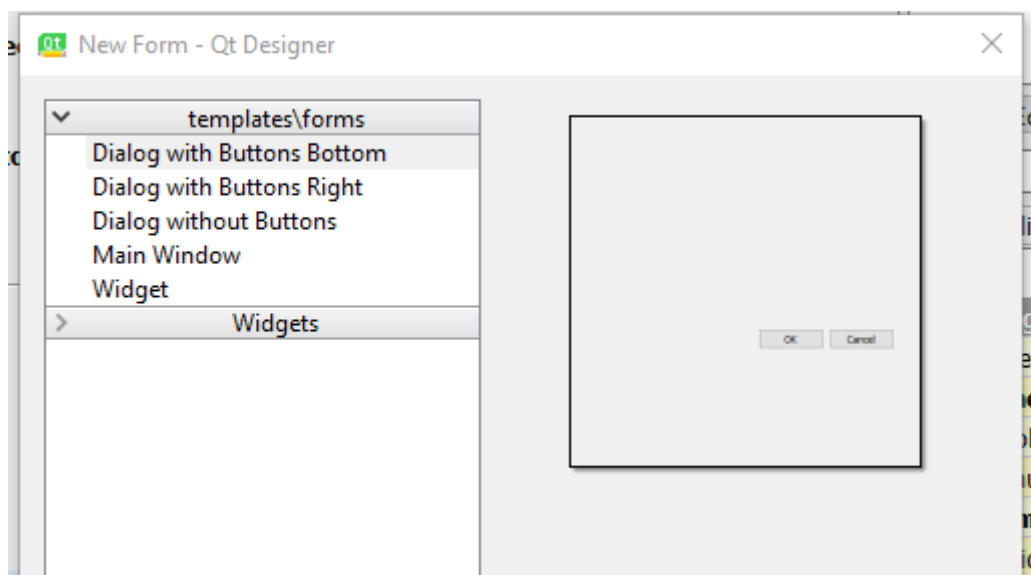
Ventanas modales y no-modales

Las **ventanas modales** son ventanas o cuadros de diálogo que aparecen sobre la página, bloqueando todas las funciones para concentrar el foco en una acción particular. Esta es su característica diferenciadora, le piden al usuario a realizar una acción. Se suele identificar cuando el fondo de la ventana primaria se oscurece. En caso contrario se denominan **no-modales**. Ventanas del tipo: **pop up**, **pop over** y **light box**, son derivadas de estas..

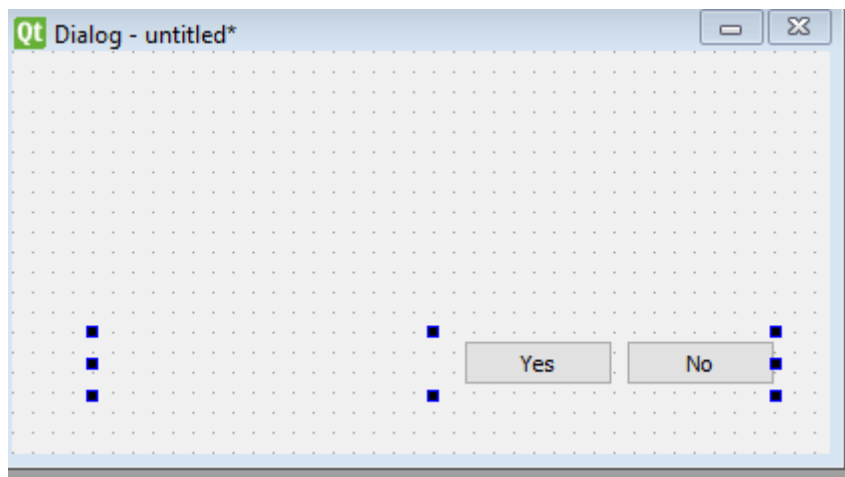
En nuestra sencilla aplicación, vamos a generar una ventana modal que nos pregunte previamente si queremos salir de la misma.



Los pasos necesarios son los siguientes:



Lanzamos una ventana de diálogo

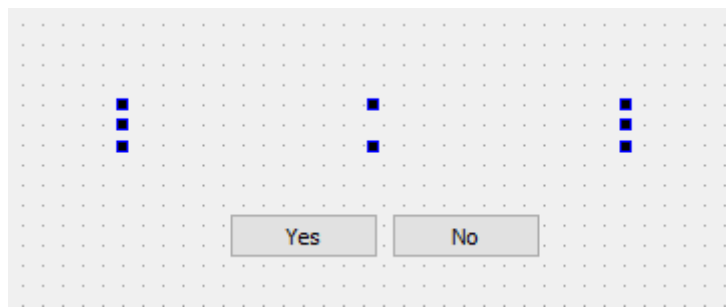


El Widget de los botones es un **button dialogbox**.

Permite contener una serie de botones

> inputMethodHints	ImhNone
▼ QDialogButtonBox	
orientation	Horizontal
▼ standardButtons	Yes No
NoButton	<input type="checkbox"/>
Ok	<input type="checkbox"/>
Save	<input type="checkbox"/>
SaveAll	<input type="checkbox"/>
Open	<input type="checkbox"/>
Yes	<input checked="" type="checkbox"/>
YesToAll	<input type="checkbox"/>
No	<input checked="" type="checkbox"/>
NoToAll	<input type="checkbox"/>
Abort	<input type="checkbox"/>
Retry	<input type="checkbox"/>
Ignore	<input type="checkbox"/>
Close	<input type="checkbox"/>
Cancel	<input type="checkbox"/>
Discard	<input type="checkbox"/>
Help	<input type="checkbox"/>
Apply	<input type="checkbox"/>
Reset	<input type="checkbox"/>
RestoreDefaults	<input type="checkbox"/>
centerButtons	<input checked="" type="checkbox"/>

Algunas propiedades del button Dialogbox

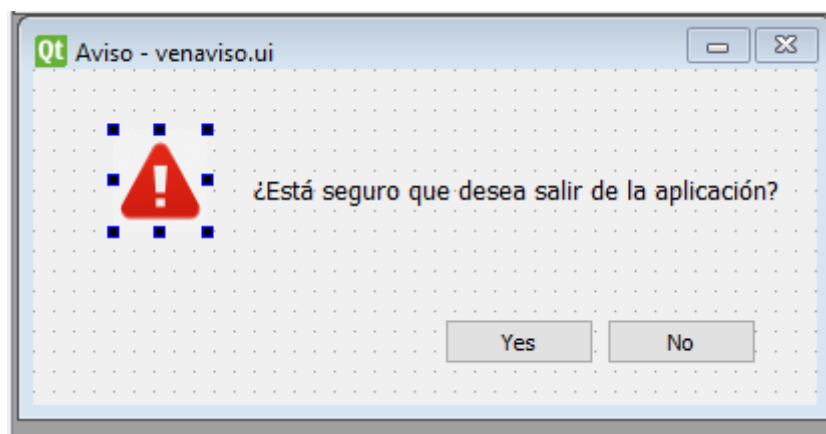


El label que contendrá el texto

Object Inspector	
Object	Class
▼ dlgAviso	QDialog
btnboxSalir	QDialogButtonBox
label	QLabel
lblMensaje	QLabel

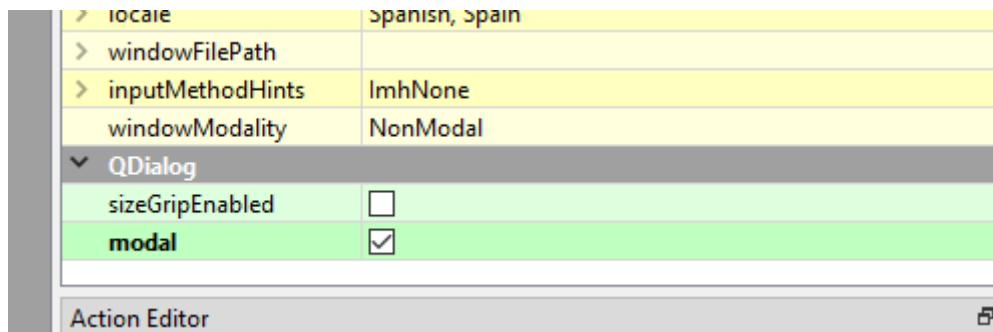
Property Editor	
Filter	
dlgAviso : QDialog	
Property	Value
> sizePolicy	[Preferred, Preferred, 0, 0]
> minimumSize	0 x 0
▼ maximumSize	16777215 x 16777215
Width	16777215
Height	16777215
> sizeIncrement	0 x 0
> baseSize	0 x 0
palette	Inherited
> font	A [MS Shell Dlg 2, 8]
cursor	Arrow
mouseTracking	<input type="checkbox"/>
tabletTracking	<input type="checkbox"/>
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	<input type="checkbox"/>
> windowTitle	Aviso
> windowIcon	
windowOpacity	1.000000
> toolTip	
toolTipDuration	-1

Propiedades de la ventana dialogo



La imagen debe de ir en un label o etiqueta.

No nos olvidemos marcar modal, al final de las propiedades del editor.

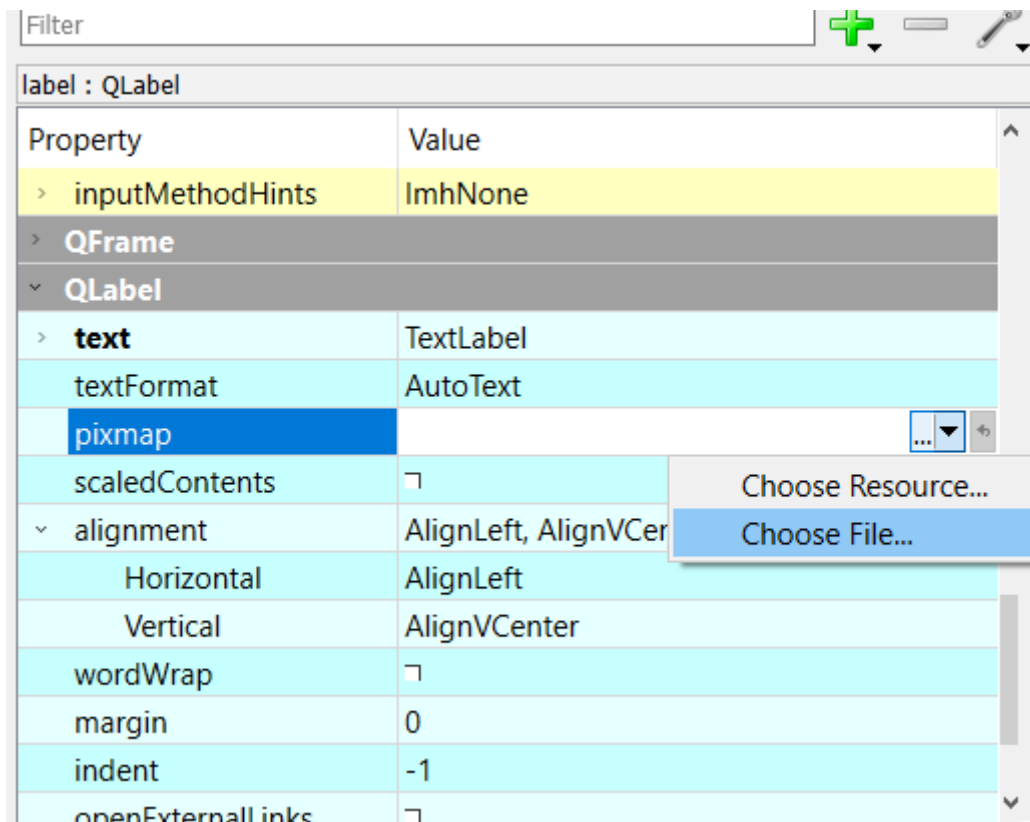


Nos vamos a parar en la inserción de la imagen de warning en la **ventana modal**, ya que requiere una serie de pasos. Esto nos permitirá en el futuro introducir imágenes en la interfaz.

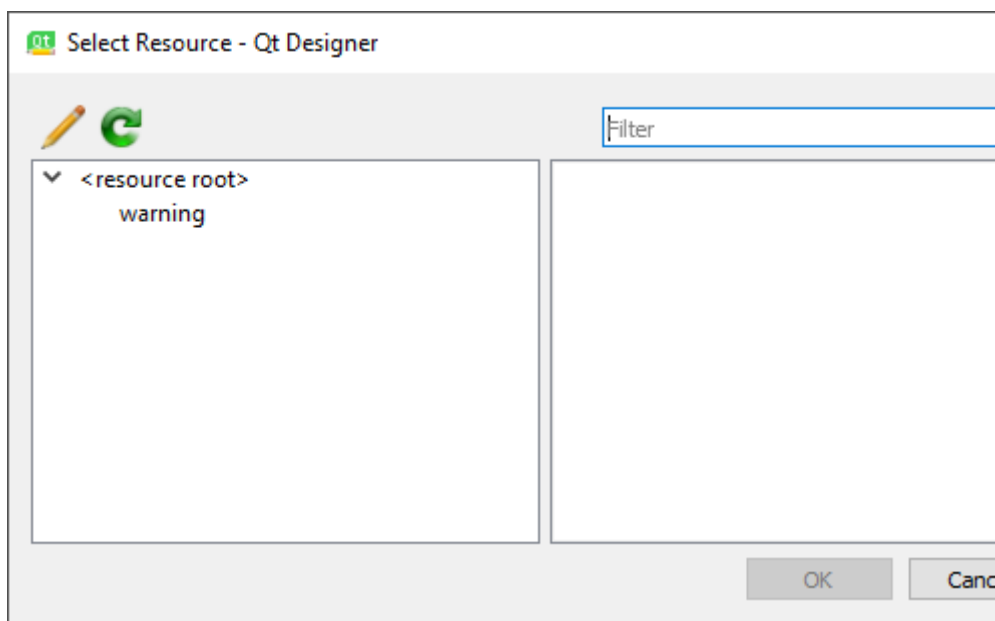
En primer lugar, hacemos lo siguiente:

1. Creamos una carpeta llamada **img** en el directorio de la aplicación
2. Descargamos una imagen similar a la del ejemplo y escalamos su tamaño a 36x36 pixeles guardándola en el directorio anterior, aunque se puede obviar este paso, ya que Qt-Designer escala las imágenes, aunque haciéndolo descargamos el peso de la aplicación en su distribución.

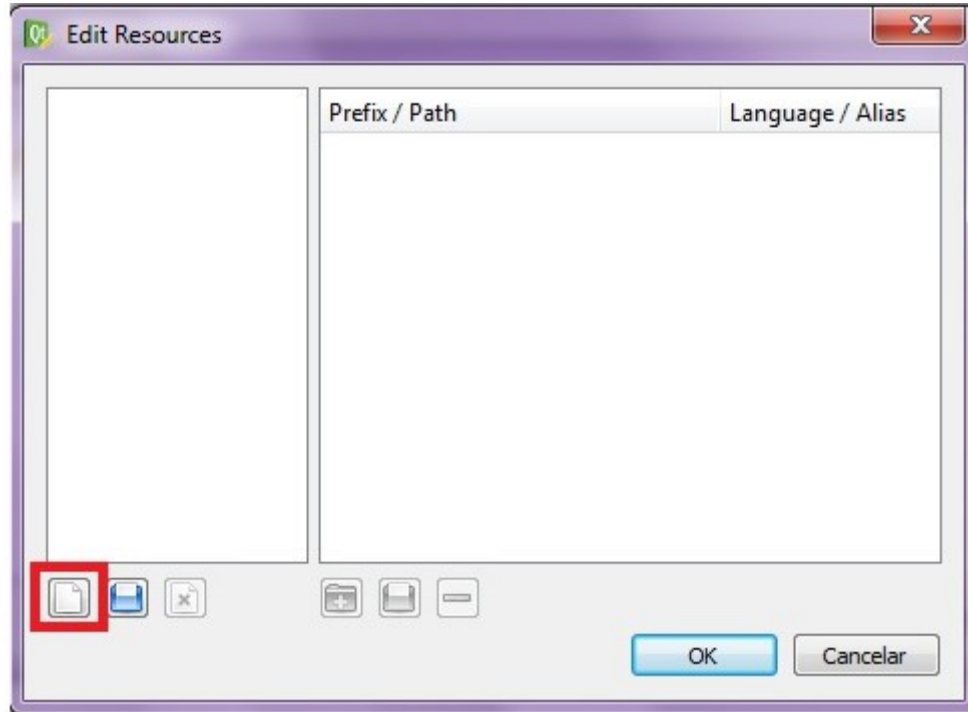
En segundo lugar, desde *Qt-Designer* seguimos las indicaciones de las imágenes que se muestran:



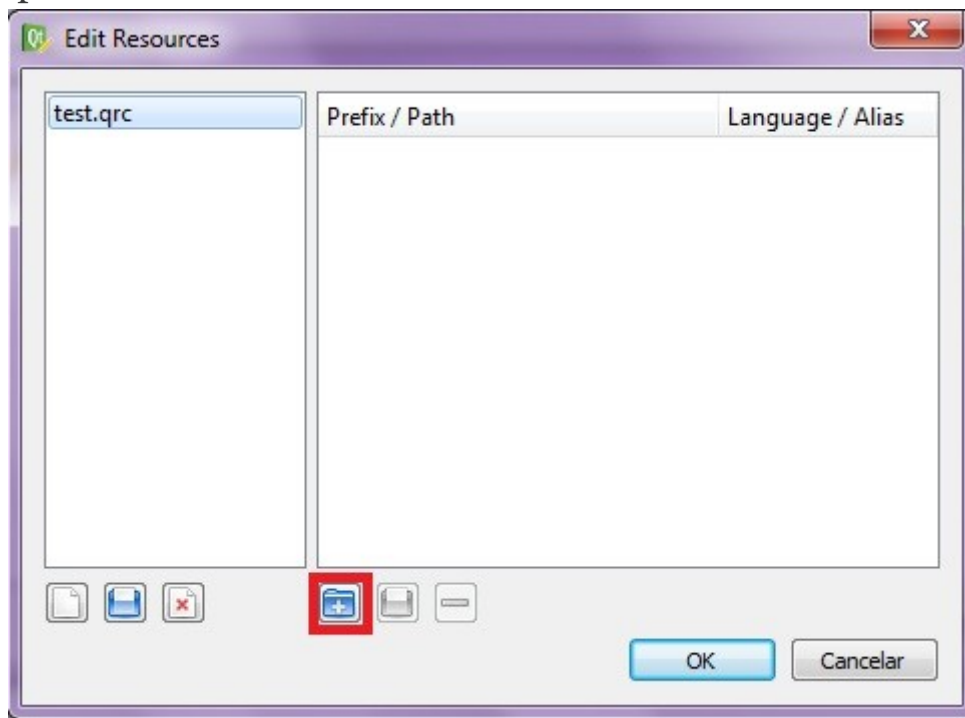
Pixmap en Label, donde configuraremos la carga de la imagen con **Choose File**



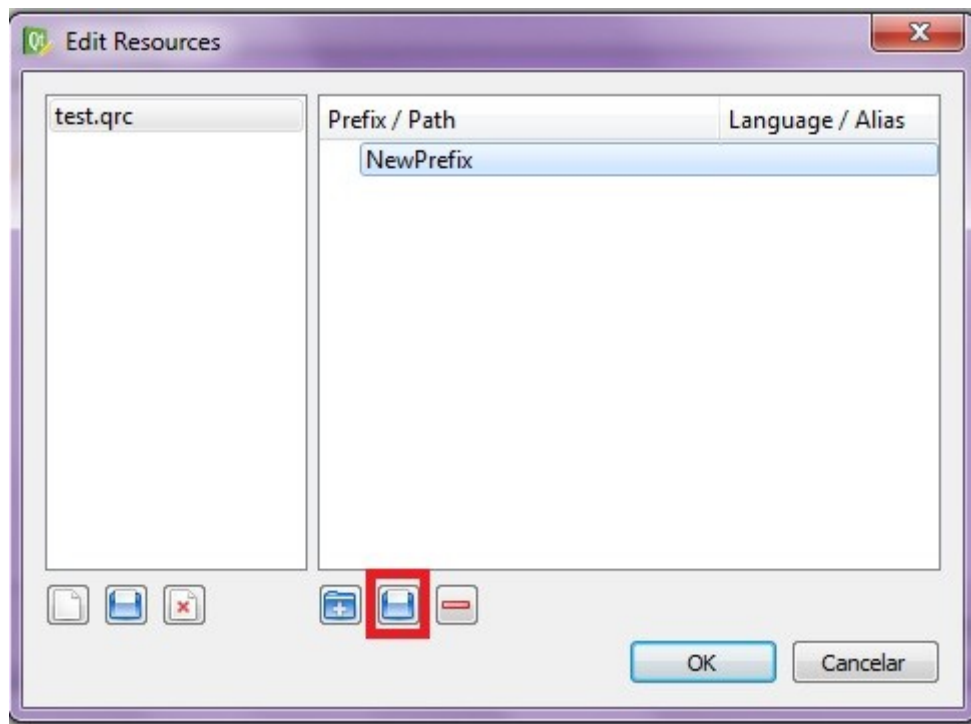
Luego en **Choose Resource...** pulsamos en el **lápiz** con lo que se nos muestra la imagen siguiente



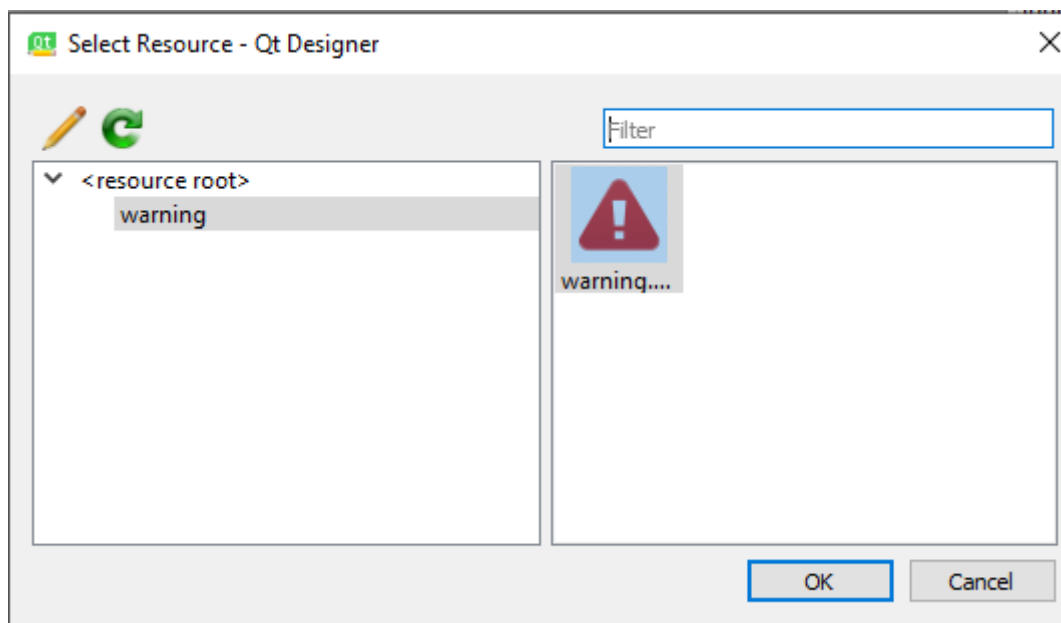
Pulsamos **new resource file** y dentro de **img** le ponemos un nombre.



Añadimos el **prefix** mecanimo para hacer referencia **de la imagen**.



Subimos la imagen descargada de la Red
y pulsamos a Ok



Resultado final que se debe obtener vemos
el fichero de la imagen y el **src** al que se asocia

Muy importante, debemos marcar **scaledContents** para que la imagen se adapta al tamaño del label.

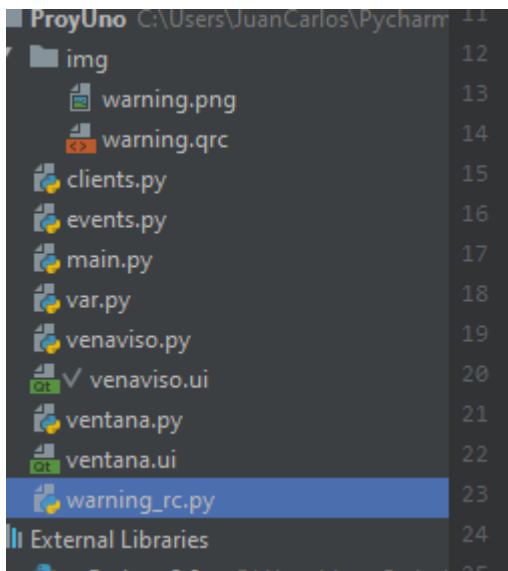
Llegados a este punto, pasamos al código. En este caso, lo que pretendemos es que cuando pulsemos Salir, la aplicación nos pregunte antes si deseamos

hacerlo, y si confirmamos salimos de la aplicación y, en caso contrario, volvemos a la situación inicial. Debemos compilar la ***ventanasalir.ui*** a ***.py***

En primer lugar tenemos que convertir el **recurso .qrc en .py**. Con PyQt tenemos que convertir todo fichero que forme parte del proyecto en un fichero **python**.

```
C:\Users\JuanCarlos\PycharmProjects\ProyUno\img>pyrcc5 warning.qrc -o warning_rc.py  
C:\Users\JuanCarlos\PycharmProjects\ProyUno\img>
```

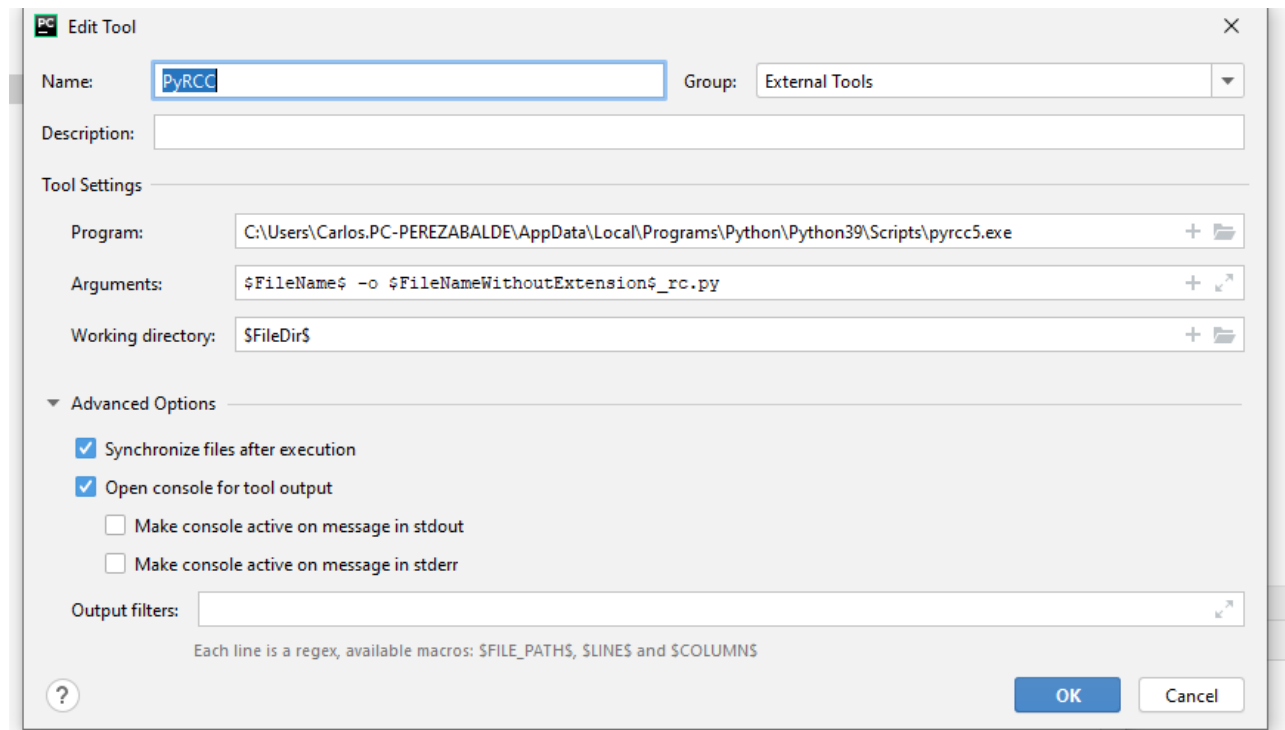
La conversión a fichero python



El fichero python creado

Muy importante, el fichero generado debe encontrarse con el resto de los ficheros Python, **en la misma ruta**.

Si queremos automatizar la compilación **.py** del fichero como una **External Tools** dejo la configuración de **File -> Settings**



Ahora nos queda incluir el código y conectar los botones de la ventana de diálogo al mismo. Empecemos mostrando el código y comentándolo:

```
class DialogSalir(QtWidgets.QDialog):
    def __init__(self):
        super(DialogSalir, self).__init__()
        var.avisosalir = Ui_dlgSalir()
        var.avisosalir.setupUi(self)
        var.avisosalir.btnBoxSalir.button(QtWidgets.QDialogButtonBox.Yes).clicked.connect(events.Eventos.Salir)
        #var.avisosalir.btnBoxSalir.button(QtWidgets.QDialogButtonBox.No).clicked.connect(events.Eventos.Salir)
        #no es necesario no quiero que haga nada
```

```
def Salir(event):
    """
    Módulo para cerrar el programa
    :return:
    """
    try:
        var.avisosalir.show()
        if var.avisosalir.exec_():
            sys.exit()
        else:
            var.avisosalir.hide()
            event.ignore() #necesario para que ignore X de la ventana
    except Exception as error:
        print('Error %s' % str(error))
```

En el fichero **eventos.py** modificamos el método **Salir** adaptándolo para que llame a la ventana **dialog** tal como se muestra en la imagen.

En el fichero **main.py** hemos hecho varias modificaciones.

Incluimos la clase **DialogSalir** para instanciar la ventana diseñada, importando además el fichero de la misma **venaviso.py** en la clase principal.

Configuramos **QAction** para que la ventana **dialog** de Salir sea llamada cuando **pulsamos el icono X** de la principal. Son las líneas:

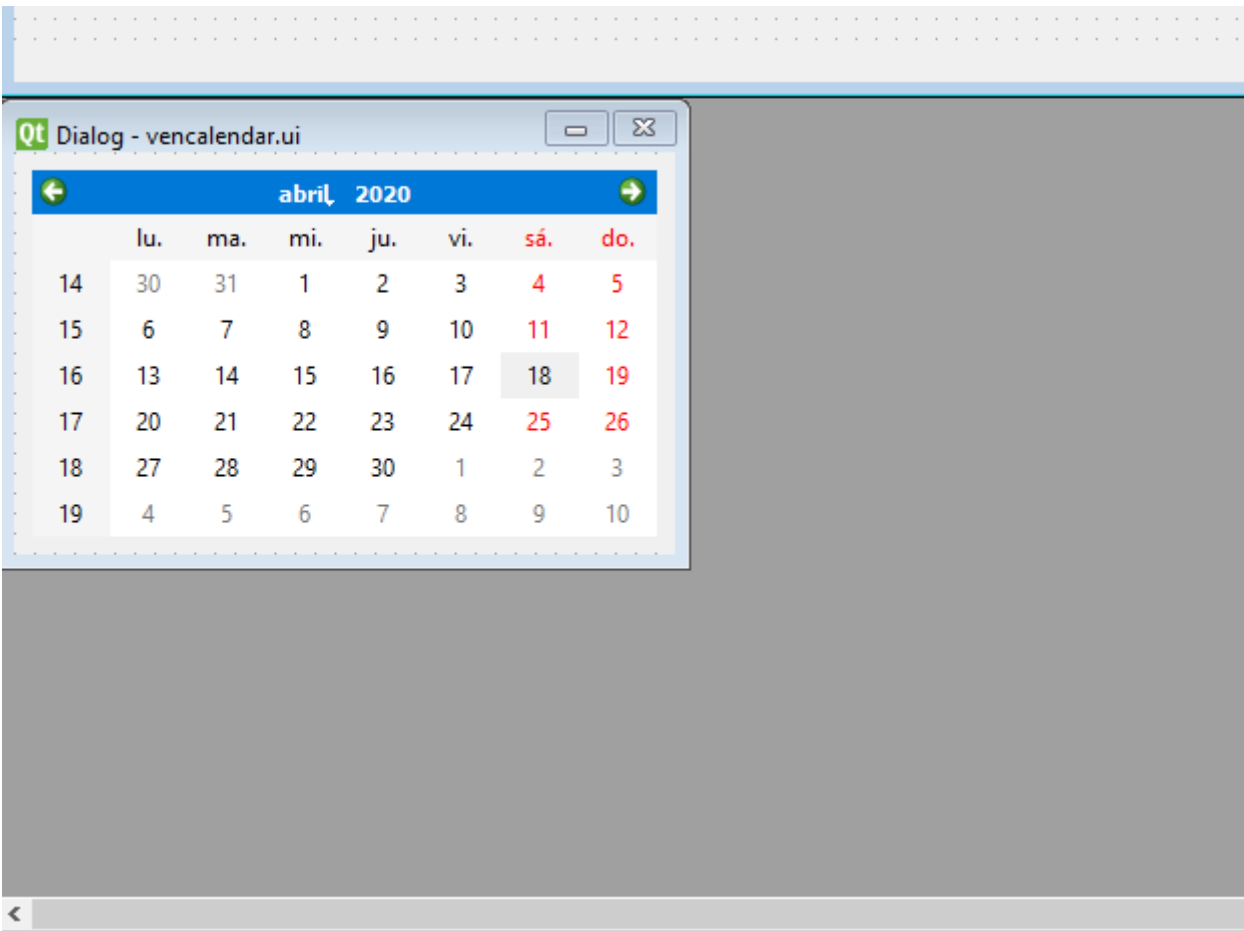
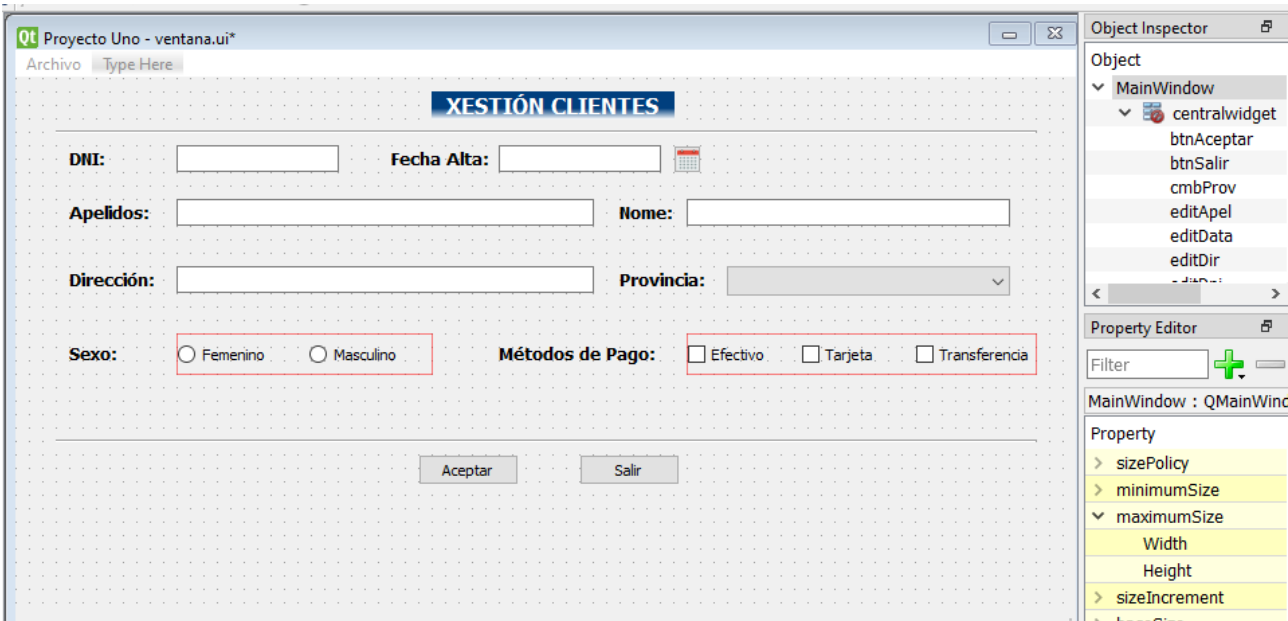
```
var.dlgcalendar = DialogCalendar()  
QtWidgets.QAction(self).triggered.connect(self.close) #debe cargarse al principio para
```

A veces es mejor suprimir la anterior línea (en algún caso interfiere en el siguiente módulo. Tuve que comentarlo en el ejemplo de clase). Ver explicación siguiente.

```
def closeEvent(self, event):  
    events.Eventos.Salir(event)
```

El evento QAction ya está configurado por defecto en la clase QWidget. Sin embargo, tal como hemos hecho, podemos sobrescribir el método manejador del evento, por ejemplo, nuestra ventana extiende la clase QWidget, esta utiliza el método **closeEvent(self, event)** para responder al evento que se produce cuando el usuario hace clic sobre el botón de cerrar (X), la acción es cerrar la ventana, podemos sobrescribir este método para que tenga un comportamiento diferente, como es el caso, preguntar al usuario si realmente desea cerrar la aplicación.

Vamos a continuación a trabajar en un segundo widget muy común en muchas aplicaciones, que es el *calendario o calendar widget*. Observemos las siguientes imágenes:



En la primera imagen se observa una nueva **caja de texto y un botón** que en vez de texto tiene la imagen de un calendario. Para ello descargamos un icono de calendario de Internet y en la propiedad del botón **icon -> choose file** cargamos en el botón dicha imagen. Esa imagen se guardará en la carpeta **img** del proyecto.

Este botón tiene por función lanzar la ventana **venCalendar** encargada de contener el **widget Calendar**. Esta ventana es del tipo **dialog without buttons**. En este caso no es necesario preguntar al usuario si quiere cerrar la ventana ya que no es un evento muy necesario. En cuanto a su función esta es, seleccionar una fecha, la de alta del cliente en este caso, y cargarla en el cuadro de texto que hay a su izquierda.

Establezcamos el evento con el código asociado.

Declaramos en primer lugar la clase **DialogCalendar** en el main.py que nos permite lanzar la ventana del calendario. Además, empezamos a trabajar con la librería **datetime** de Python. El método **setSelectedDate** nos permite seleccionar una fecha y el evento **cargaFecha** tiene como función cargar la fecha en la caja de texto de alta del cliente.

```
class DialogCalendar(QtWidgets.QDialog):
    def __init__(self):
        super(DialogCalendar, self).__init__()
        var.dlgcalendar = Ui_dlgCalendar()
        var.dlgcalendar.setupUi(self)
        diaactual = datetime.now().day
        mesactual = datetime.now().month
        anoactual = datetime.now().year
        var.dlgcalendar.Calendar.setSelectedDate((QtCore.QDate(anoactual,mesactual,diaactual)))
        var.dlgcalendar.Calendar.clicked.connect(clients.Cientes.cargarFecha)
```

En la clase main.py no nos olvidemos de instanciar la clase anterior y declarar el evento **abrirCalendar** que lanzará la ventana del calendario:

```
class Main(QtWidgets.QMainWindow):
    def __init__(self):
        super(Main, self).__init__()
        var.ui = Ui_venPrincipal()
        var.ui.setupUi(self)
        var.avisosalir = DialogSalir()
        var.dlgcalendar = DialogCalendar()
```

Instanciamos ambas clases

```

'''
conexion de eventos con los objetos
estamos conectando el código con la interfaz gráfico
'''

QtWidgets.QAction(self).triggered.connect(self.close)
var.ui.btnSalir.clicked.connect(events.Eventos.Salir)
var.ui.actionSalir.triggered.connect(events.Eventos.Salir)
var.ui.editDni.editingFinished.connect(clients.Clientes.validoDni)
var.ui.btnCalendar.clicked.connect(clients.Clientes.abrirCalendar)
for i in var.rbtsex:

```

Conectamos el Botón con la función **abrirCalendar**

Ahora nos vamos al módulo *clients.py* que será donde situamos los módulos **abrirCalendar** y *cargaFecha* que sería más o menos de la forma siguiente. Este módulo es ejecutado cuando se instancia *dlgCalendar*

```

'''
Abrir la ventana calendario
'''

def abrirCalendar():
    try:
        var.dlgcalendar.show()
    except Exception as error:
        print('Error: %s ' % str(error))

'''
Este módulo se ejecuta cuando clickeamos en un día del calendar, es decir, clicked.connect de calendar
'''

def cargarFecha(qDate):
    try:
        data = ('{0}/{1}/{2}'.format(qDate.day(), qDate.month(), qDate.year()))
        var.ui.editClialta.setText(str(data))
        var.dlgcalendar.hide()
    except Exception as error:
        print('Error cargar fecha: %s ' % str(error))

```

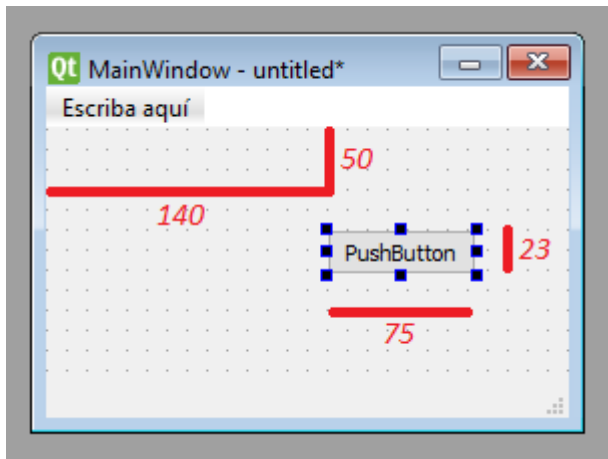
Hasta aquí un repaso básico de los widgets más utilizados en la mayoría de las aplicaciones de gestión. En la actividad siguiente nos centraremos en el concepto de layout o disposición.

Layouts

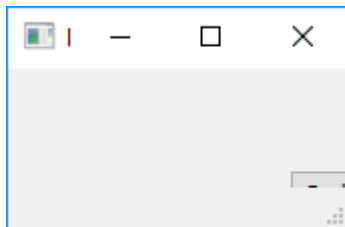
Los **Layouts** son los elementos sobre los cuales se sustentan los diferentes componentes de la **interfaz de usuario**, y controlan la **distribución, la posición y las dimensiones** de dichos componentes.

El manejo de los layouts en QT - Designer no es tarea sencilla y, en realidad, tiene algo de "ensayo prueba-error". Ya sea una ventana principal *QMainWindow*, un *QWidget* o un *QDialog*, en todos necesitaremos

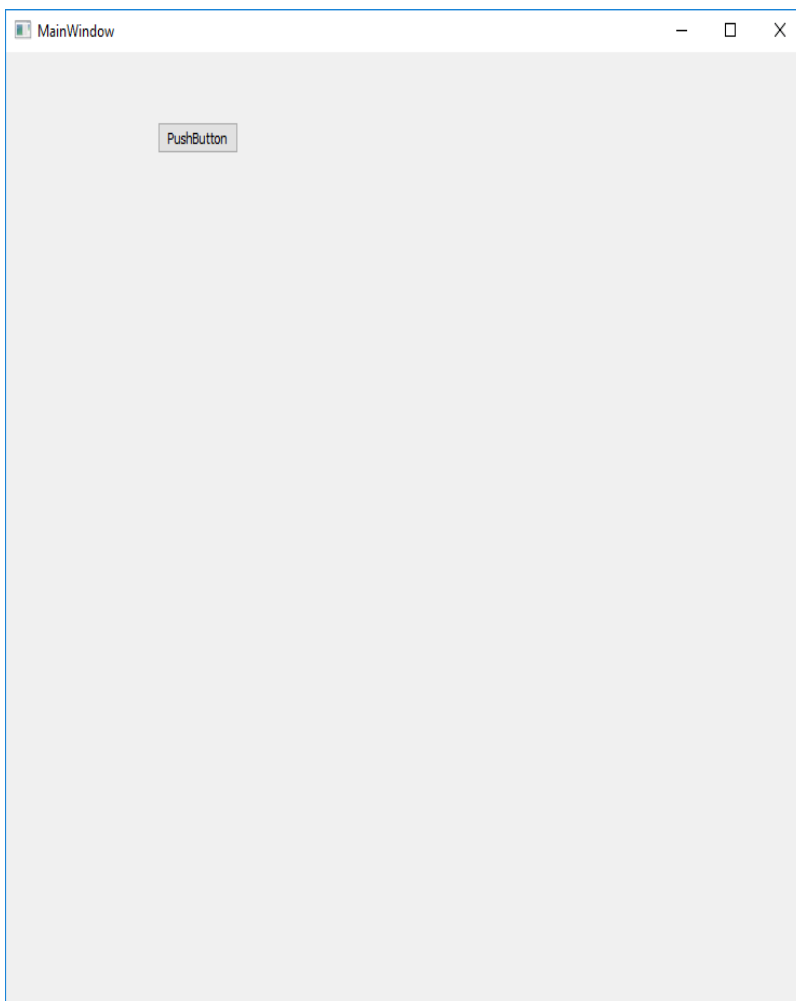
jugar con la disposición de los elementos si queremos crear un diseño atractivo, y sobre todo usable, para el programa.



Podemos usar la distribución mediante coordenadas. Si vemos el fichero .ui vemos las coordenadas:
`self.pushButton.setGeometry(
QtCore.QRect(140, 50, 75,
23))`



El problema de utilizar esta forma es que está “enlazada” a las coordenadas exactas de la ventana. Si la ventana fuera más pequeña no se vería el *PushButton*:

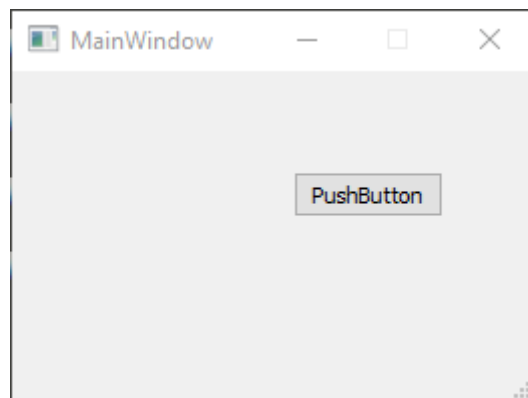


Y si fuera más grande quedaría con el mismo margen respecto a la esquina superior izquierda y se vería mal:

Por la tanto, una solución sería bloquear la ventana para que no se pueda redimensionar. Desde el diseñador se puede hacer estableciendo las propiedades sizePolicy de la QMainWindow; minimimSize y maximumSize manualmente:

▼ minimumSize	261 x 165
Ancho	261
Alto	165
▼ maximumSize	261 x 165
Ancho	261
Alto	165

De esa forma nos aparecerá bloqueado el botón de maximizar y tampoco podremos redimensionar:



Alternativamente, si inicializamos desde el código podemos hacer lo mismo con la siguiente línea en el constructor de la QMainWindow:

```
self.setFixedSize(261,165)
```

Como hemos visto, hacer un programa con los widgets posicionados por coordenada no es una buena opción. Además hay que tener en cuenta que, dada la naturaleza multiplataforma de Qt, el tamaño de los componentes puede variar entre sistemas operativos. Un botón que se ve bien en Windows puede ser mayor en Linux, o más pequeño en MAC OS o viceversa.

Para asegurarnos de que un programa se verá perfecto independientemente del sistema tenemos dos opciones:

- **Sobreescribir el diseño de los widgets genéricos del sistema** por los nuestros propios, así siempre se verá igual.
- **O bien ajustar automáticamente el tamaño de la ventana y su contenido utilizando layouts.** Es quizás la opción más lógica y técnica.

Distribución de widgets con layouts

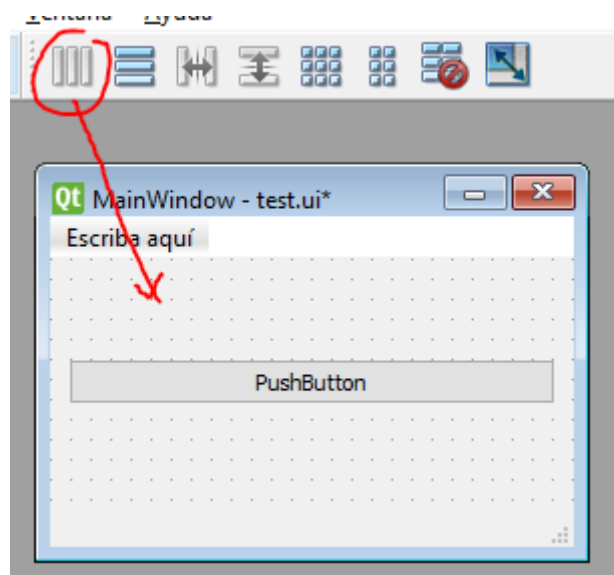
Cuando diseñamos una ventana principal con Qt Designer, veremos que éstas contienen un objeto llamado **centralwidget** de tipo **QWidget**:

Objeto	Clase
MainWindow	QMainWindow
centralwidget	QWidget
pushButton	QPushButton
menubar	QMenuBar
statusbar	QStatusBar

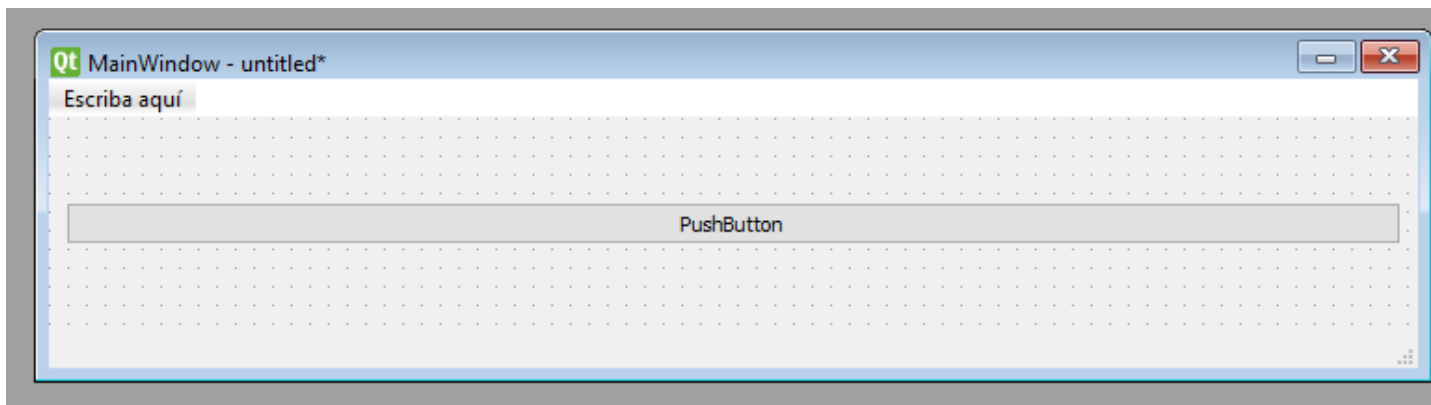
El objetivo de este widget es controlar la disposición de su contenido, y tenemos varias opciones disponibles a través de los botones de la parte superior, sólo disponibles si tenemos por lo menos un componente (como un botón) dentro de la ventana:



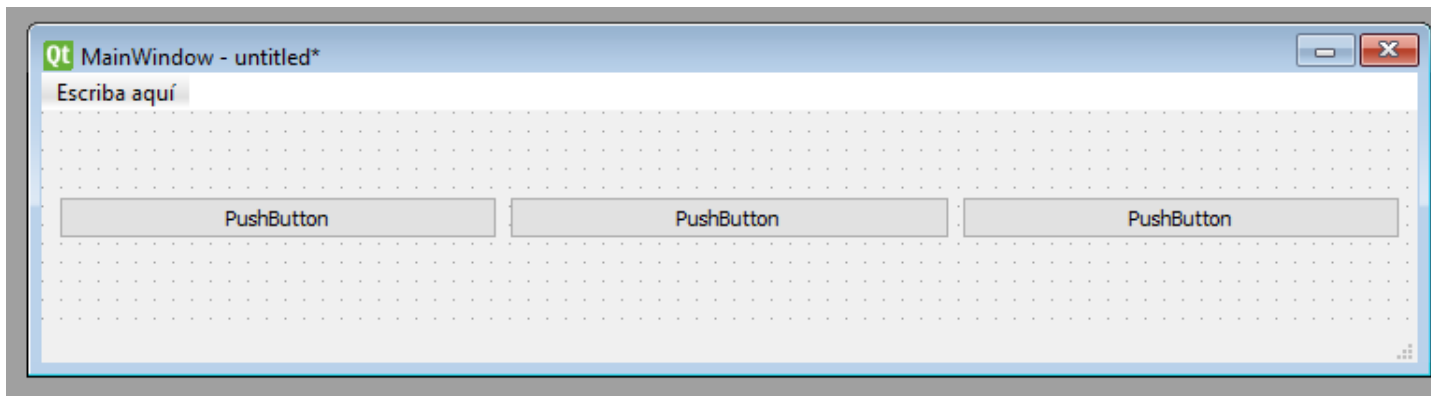
Tal como resalto en la imagen, por defecto está marcada la opción de “Romper la distribución” (en gris deshabilitado) en la barra de herramientas o iconos superior. Esto habilita el posicionamiento libre por coordenadas. Pero si cambiamos a una “**Distribución horizontal**” veremos que empiezan a cambiar cosas:



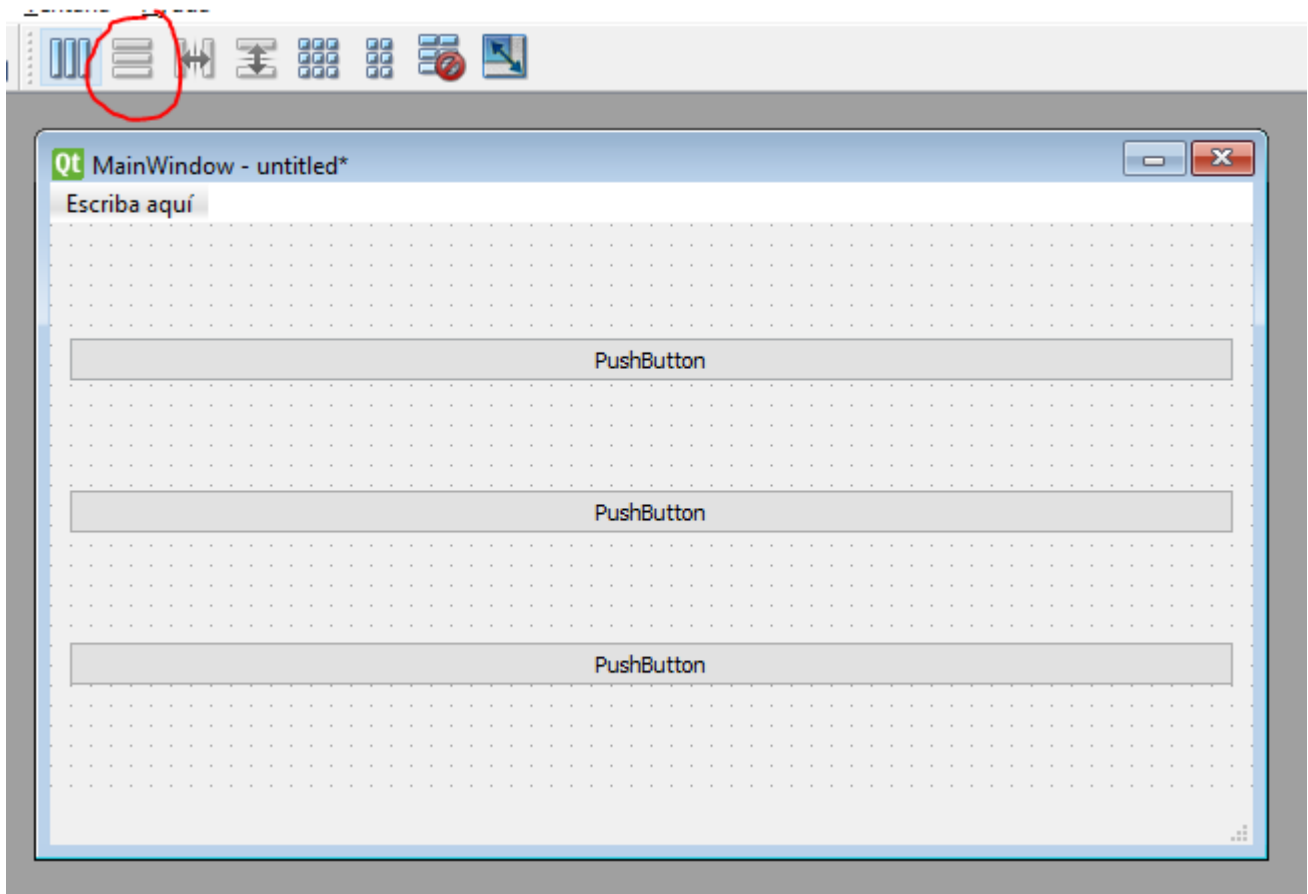
Al cambiar la distribución a horizontal, lo que haremos es que los elementos se posicionen dentro de la ventana ocupando todo el ancho. Si tenemos un elemento éste ocupa el 100% del ancho independientemente del tamaño de la ventana:



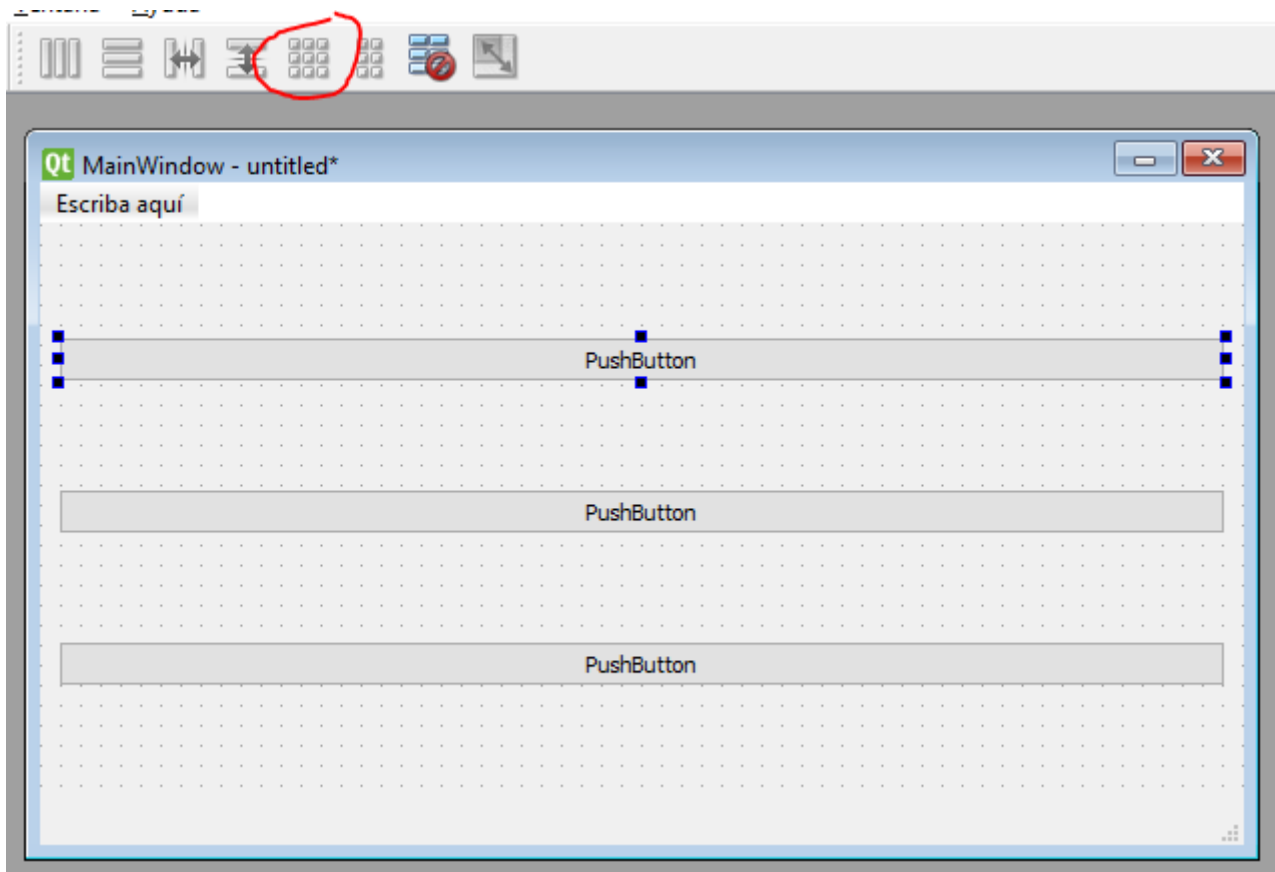
El punto está en que si intentamos añadir varios widgets, éstos siempre **se posicionarán horizontalmente**, dividiendo el espacio total entre ellos:



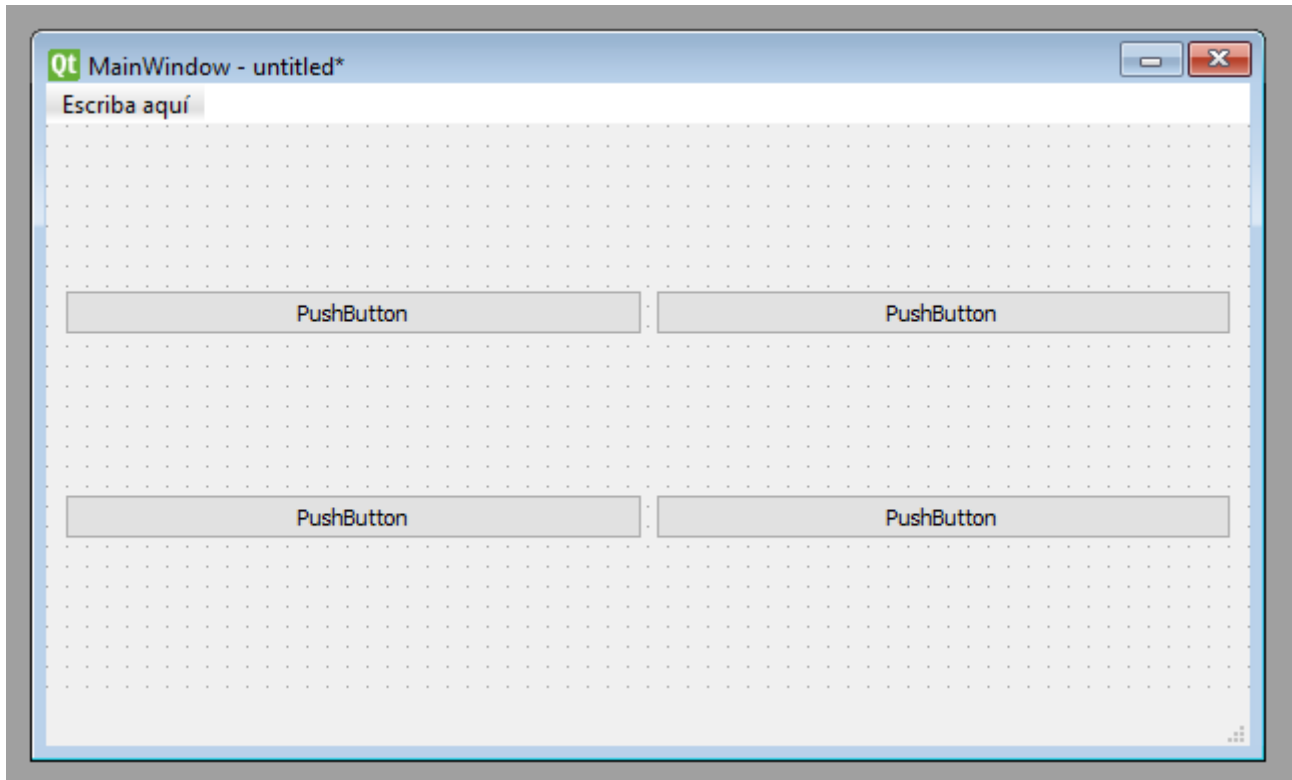
Ahora, si cambiásemos a una “**Disposición vertical**” en lugar de horizontal, tendríamos **el mismo efecto pero verticalmente**, con un solo widget por fila dividiéndose el espacio vertical entre todos ellos, independientemente del ancho y alto de la ventana:



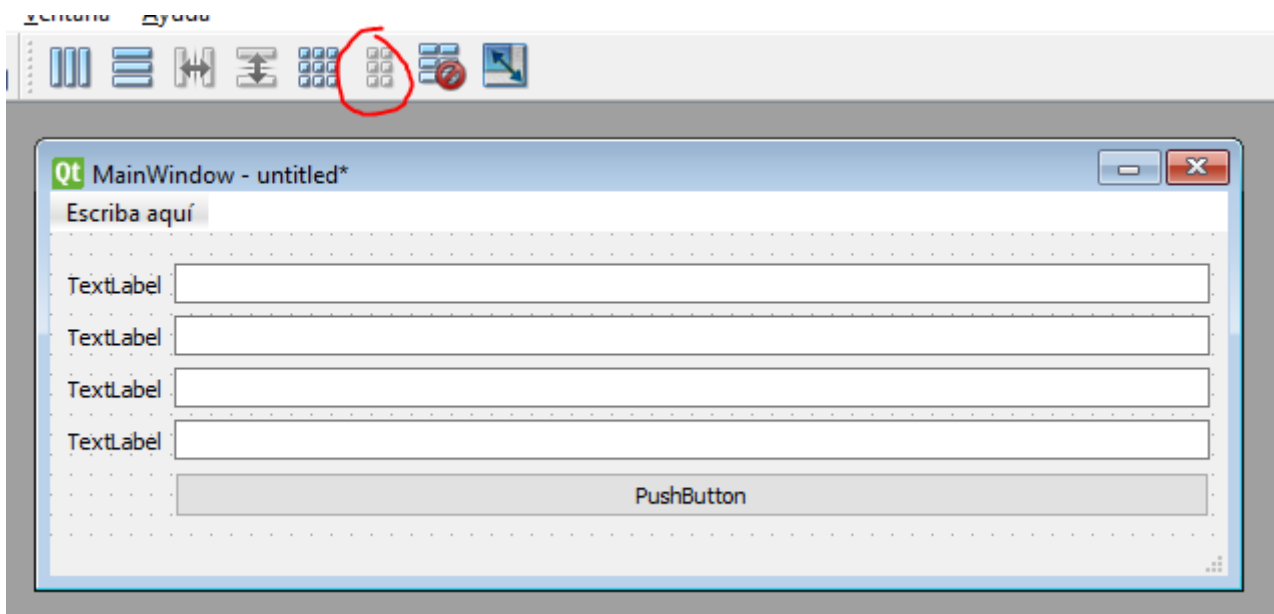
Otra opción interesante es la “**Distribución en cuadrícula**”:



Si la activamos no ocurrirá nada, pero nos permitirá **posicionar los elementos en una especie de tabla o rejilla**. Si bien la disposición horizontal simulaba una fila y la vertical una columna, con esta podemos establecer diferentes filas y columnas arrastrando los componentes a las “celdas” donde queramos ponerlos:



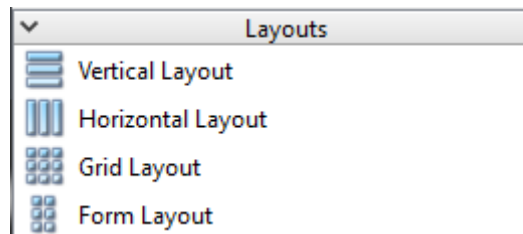
Por último también está la opción de utilizar una “**Distribución de formulario**”, cuyo propósito como el nombre indica, es construir dos espacios. Uno más pequeño a la izquierda para las etiquetas (Labels) y otro a la derecha para los campos de texto (LineEdit):



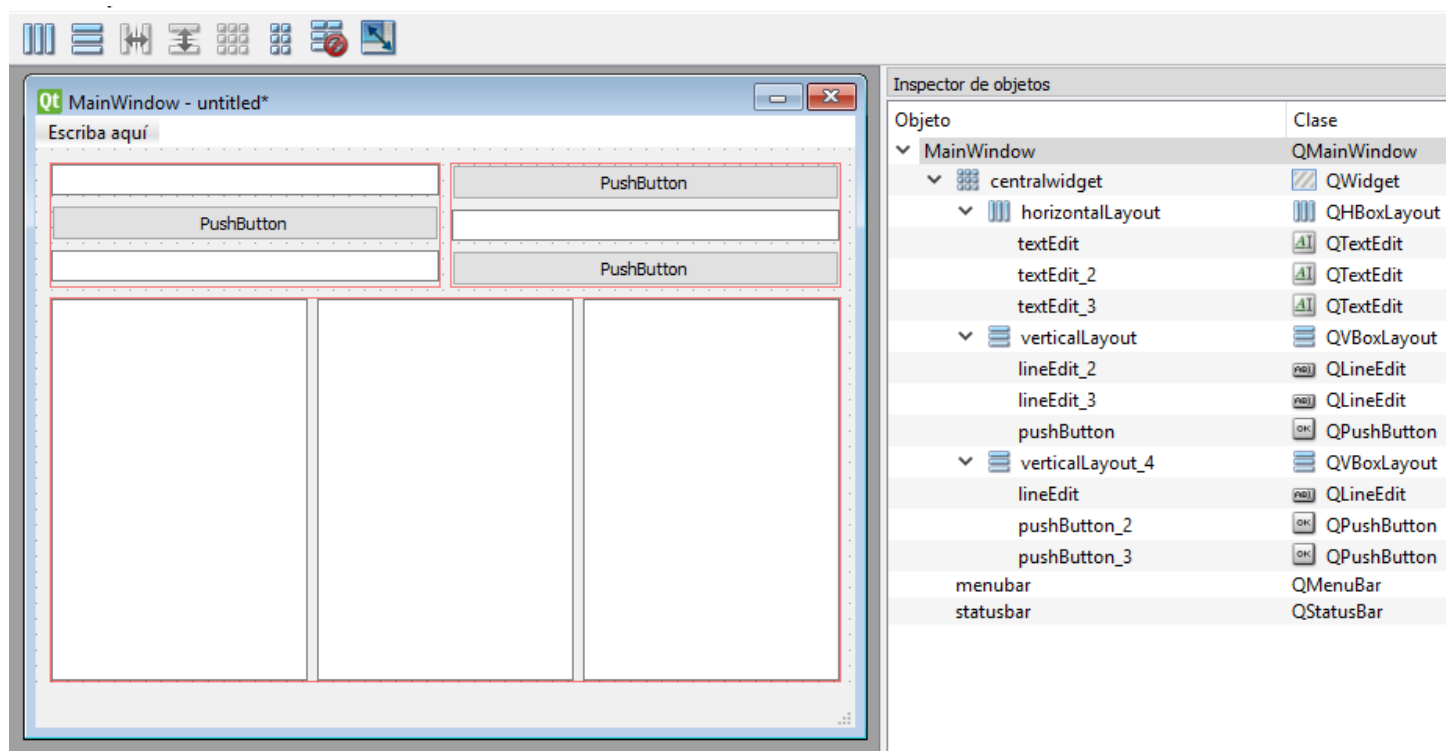
Mezclando diferentes distribuciones

Como hemos visto las QMainWindow tienen ese centralwidget base, pero también podemos crear nuestras propias estructuras introduciendo **layouts dentro de otros layouts**.

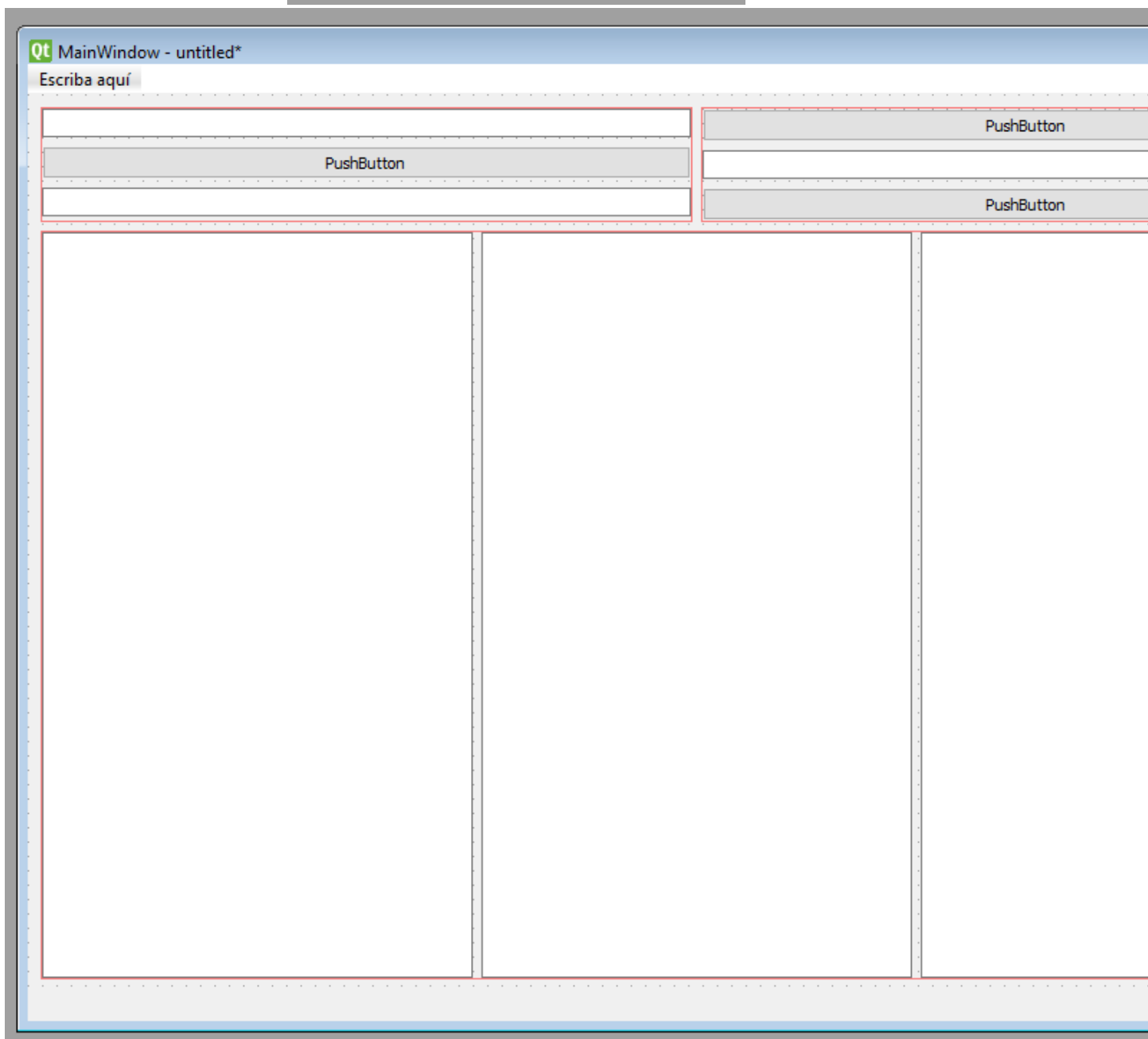
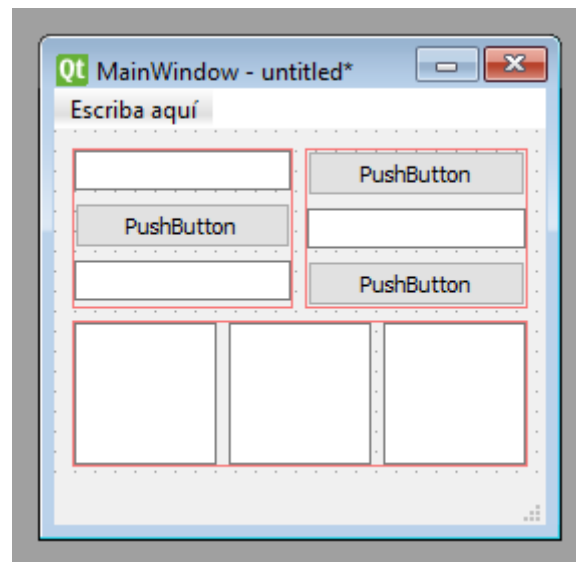
Tenemos las 4 formas mencionadas también disponibles como widgets independientes:



Podemos ponerlos dentro del centralwidget de una QMainWindow y en QWidget y QDialogs, ya que ellos carecen de centralwidget por defecto. Un ejemplo podría ser el siguiente:



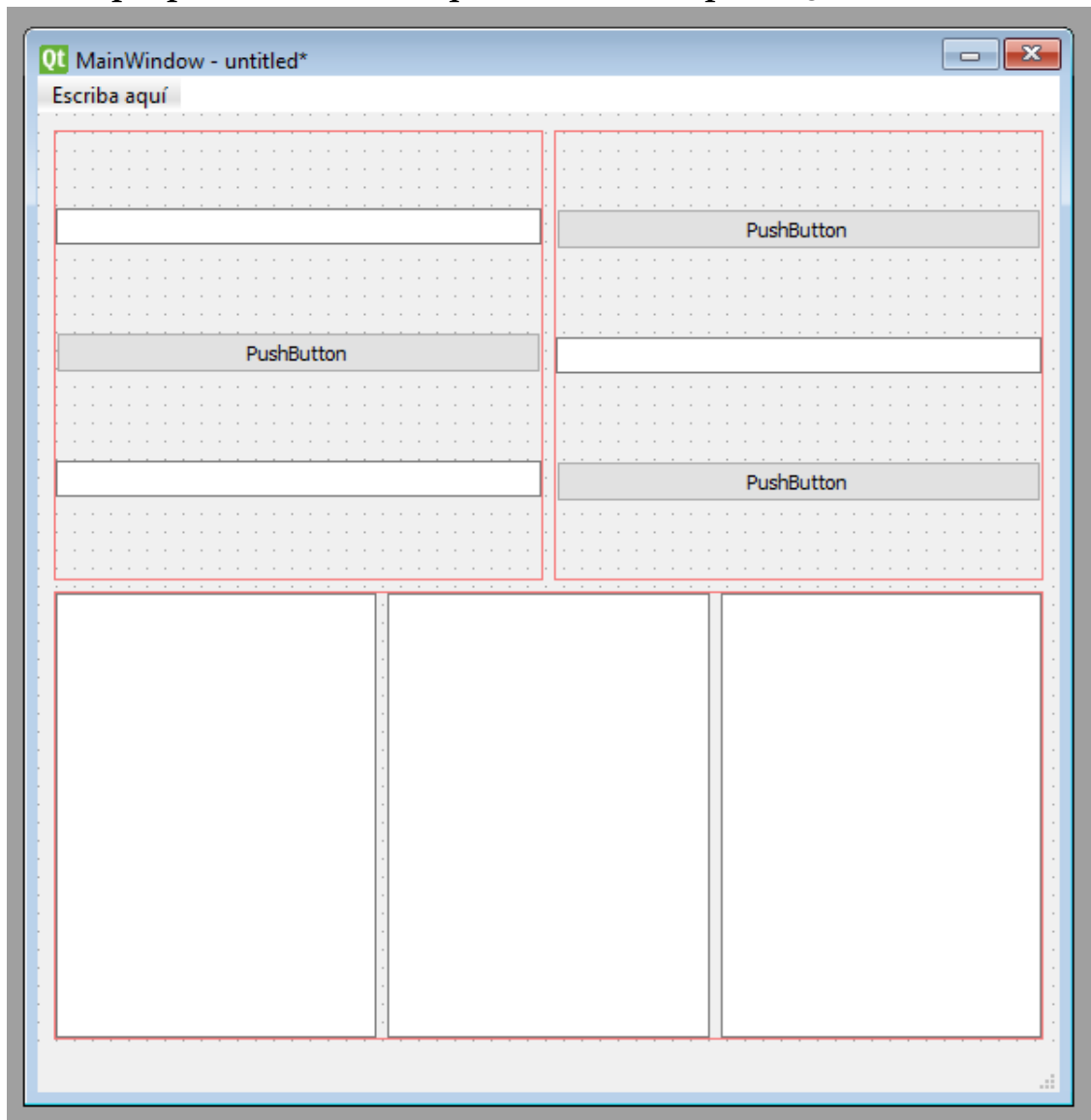
Este diseño parte de un centralwidget con una disposición en cuadrícula que contiene dos distribuciones verticales en la primera fila, y una horizontal abajo que ocupa toda la segunda fila. Como se ha comentado antes, lo bueno es que estos elementos **se adaptarán automáticamente** al tamaño de la ventana:



Además podemos mezclar propiedades específicas de cada layout para ir todavía más allá, como por ejemplo dividir el tamaño de las filas uniformemente en una distribución en cuadrícula:

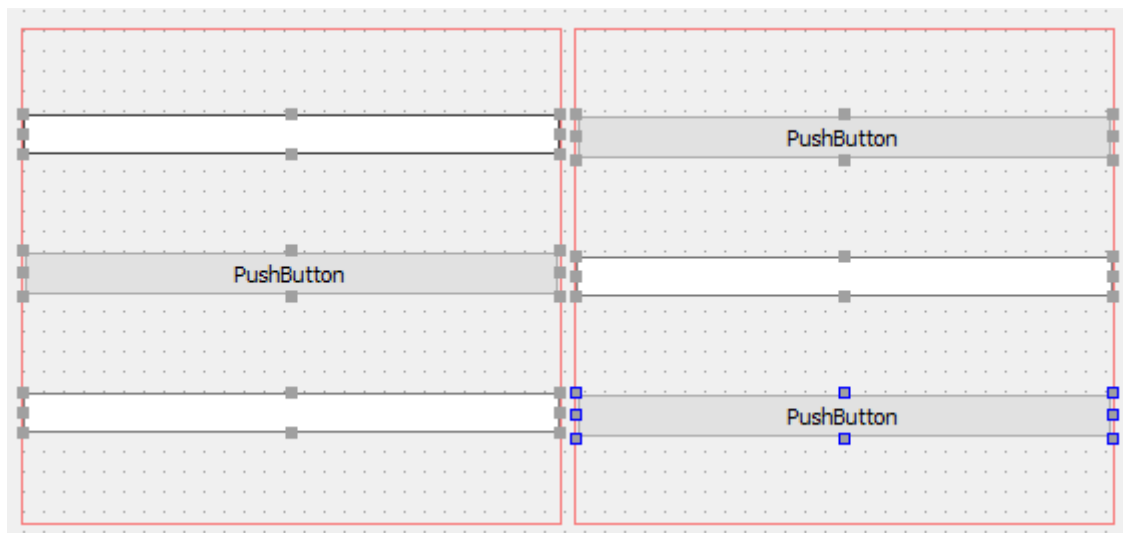
`layoutRowStretch` 50,50

Esta propiedad establece que cada fila ocupa un 50% de la altura.



Como se puede observar, ahora los campos de texto inferiores sólo ocupan la mitad de la altura.

Jugando podríamos hacer que todos los componentes superiores se expandan para ocupar el espacio cambiando sus Políticas a Expanding en su `sizePolicy`. Podríamos seleccionarlos todos con Control presionado:

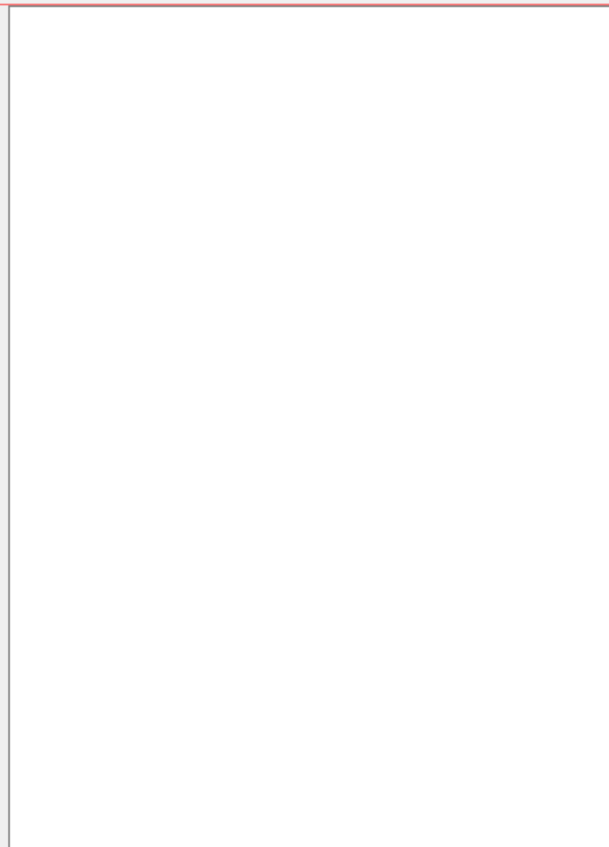
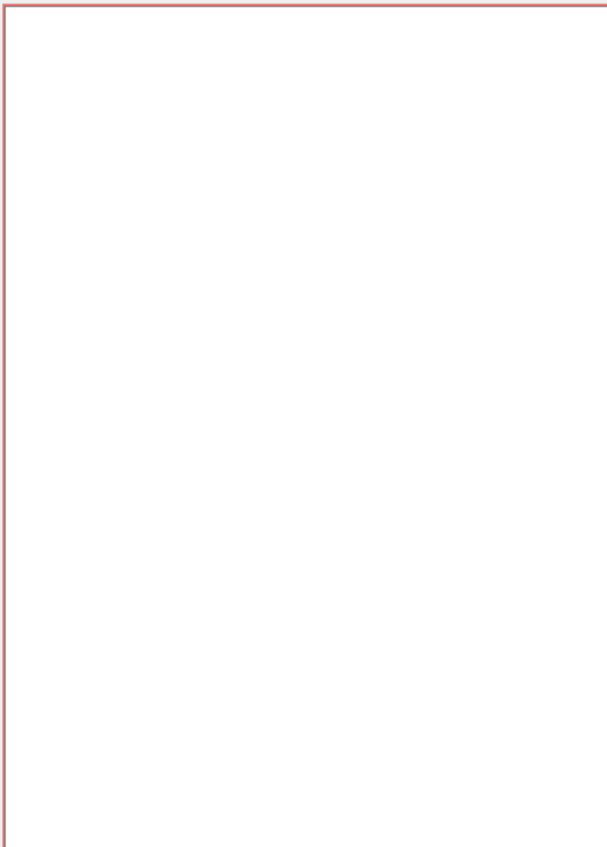
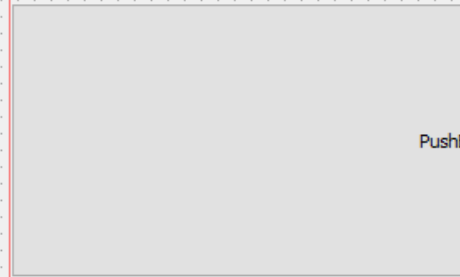
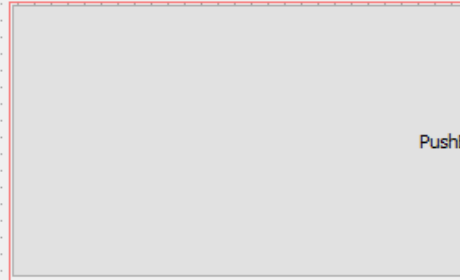
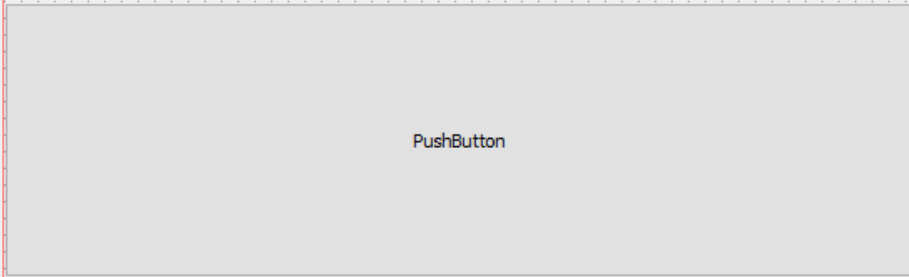


Y luego cambiar esa propiedad en bloque:

Editor de propiedades	
sizePo	
pushButton_3 : QPushButton	
Propiedad	Valor
▼ QWidget	
▼ sizePolicy	[Expanding, Expanding, 0, 0]
Política horizontal	Expanding
Política vertical	Expanding
Ajuste horizontal	0
Ajuste vertical	0

Y este sería el resultado:

Escriba aquí



Sin duda todo, y una vez más insistir, que se basa en experimentar, en este y otros diseñadores gráficos de Iu, probar mediante ensayo prueba-error hasta que encontremos la forma que nos interesa.

Views y Paneles

El **Modelo – Vista – Controlador (MVC)** es un patrón arquitectónico utilizado para desarrollar interfaces de usuario que divide una aplicación en tres partes interconectadas permitiendo separar la representación interna de los datos de cómo se presenta y acepta la información del usuario.

El patrón de diseño MVC tiene tres componentes principales:

- El **modelo** contiene la estructura de datos con la que está trabajando la aplicación.
- La **vista** es cualquier representación de información que se muestra al usuario, ya sea gráfica o de tablas. Se permiten múltiples vistas del mismo modelo de datos.
- El **controlador** acepta la entrada del usuario, transformándola en comandos para modificar el modelo o la vista.

Las dos partes son esencialmente responsables de:

- El modelo almacena los datos, o una referencia a ellos, y devuelve registros de rangos individuales o individuales, y metadatos asociados o instrucciones de visualización.
- La vista solicita datos del modelo y muestra lo que se devuelve en el widget

```
def selSexo(self):
    try:
        global sex
        if var.ui.rbtFem.isChecked():
            sex = 'Mujer'
        if var.ui.rbtMasc.isChecked():
            sex = 'Hombre'

    except Exception as error:
        print('Error: %s ' % str(error))
```

En este módulo lo que hemos hecho es declarar una variable sex de carácter global para poder acceder a ella desde otros módulos de la misma clase. Lo veremos en el módulo *showClients*.

```
def selPago():
    try:
        if var.ui.chkEfectivo.isChecked():
            #print('Pagas con efectivo')
            var.pay.append('Efectivo')
        if var.ui.chkTarjeta.isChecked():
            #print('Pagas con tarjeta')
            var.pay.append('Tarjeta')
        if var.ui.chkTransf.isChecked():
            #print('Pagas con transferencia')
            var.pay.append('Transferencia')
    except Exception as error:
        print('Error: %s ' % str(error))
```

El manejo de los checkbox es algo más complicado. Cada vez que se checkea un valor se llama a este módulo con lo que se duplican o incluso triplican en este caso los valores elegidos.

Eso lo resolveremos en el módulo *showClients*. Por otro lado, hemos creado una variable *var.pay* para almacenar los métodos de pago elegidos por los clientes.

```
def selProv(prov):
    try:
        global vpro
        vpro = prov
    except Exception as error:
        print('Error: %s ' % str(error))
```

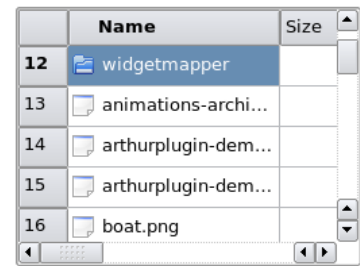
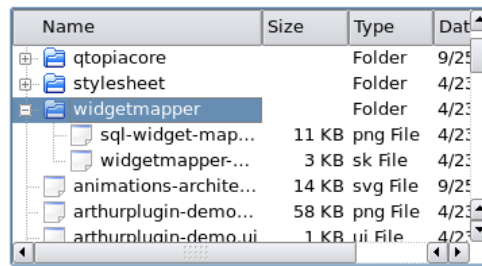
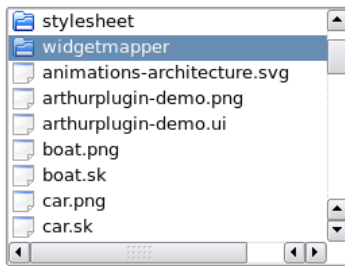
Las modificaciones realizadas aquí son mínimas pero necesarias para acceder a la provincia seleccionada.

Es posible que la distinción entre la vista y el controlador se vuelva un poco difusa. PyQt acepta eventos de entrada del usuario (a través del sistema operativo) y los delega a los widgets que actúan como controladores para manejar dichos eventos además de manejar la presentación del estado actual al usuario, es decir, actuando en el modo Vista.

El modelo actúa como la interfaz entre el almacén de datos y el **ViewController**. El Modelo contiene los datos (o una referencia a ellos) y presenta estos datos a través de una API estandarizada que las Vistas luego toman y presentan al usuario. Las vistas múltiples pueden compartir los mismos datos, presentándolos de formas completamente diferentes. Se puede usar cualquier "almacén de datos" para su modelo, incluida, por ejemplo, una lista o diccionario estándar de Python, o una base de datos.

Todos los item models están basados en la clase `QAbstractItemModel`. `QAbstractItemModel` proporciona una interfaz para datos que es lo suficientemente flexible como para manejar vistas que representan datos en forma de tablas, listas y árboles. Sin embargo, al implementar nuevos modelos para estructuras de datos tipo lista y tabla, las clases `QAbstractListModel` y `QAbstractTableModel` son más útiles porque proporcionan implementaciones predeterminadas más legibles a través de los widgets **`QListView`**, **`QTreeView`** y **`QTableView`**.

En la siguiente imagen ejemplo podemos ver a la izquierda **`QListView`**, en el centro **`QTreeView`** y a la derecha, el más común para la representación del contenido de una base de datos, el **`QTableModel`**.



El comportamiento predeterminado de las vistas estándar que se muestran debería ser suficiente para la mayoría de las aplicaciones. Proporcionan servicios de edición básicos y se pueden personalizar para satisfacer las necesidades de interfaces de usuario más especializadas y complejas.

A continuación vamos a revisar los códigos necesarios para que cuando demos de alta un cliente, usuario, producto, se muestre en un **QTableModel** los valores que hemos cargado.

En primer lugar, añadimos dicho widget a nuestro formulario ejemplo, quedando algo parecido a esto:

Proyecto Uno - [Preview] - Qt Designer

Archivo

XESTIÓN CLIENTES

DNI: Fecha Alta:

Apellidos: Nombre:

Dirección: Provincia:

Sexo: ☐ Femenino ☐ Masculino Métodos de Pago: ☐ Efectivo ☐ Tarjeta ☐ Transferencia

Empecemos ahora con el código.

En primer lugar preparamos la lista que contendrá todos los datos. No se va exponer todo el código, solo aquel que hemos modificado. Tendremos dos funciones `showClients`, `selProov` y `cleanClients`, ambos dentro del fichero ***Cientes.py***. También vamos a realizar modificaciones en `selPago` y `selSex`. Empecemos por estas últimas.

Ya hemos modificado los módulos necesarios para cargar los datos de cada cliente. Vamos mostrar el módulo `showClients` en dos partes. La primera nos prepara la lista de los datos del cliente para que luego la segunda parte del código nos lo cargue en el view.

```
def showClients():
    try:
        #Preparamos el registro
        newcli = []
        client = [var.ui.editDni, var.ui.editApel, var.ui.editNome, var.ui.editData, var.ui.editDir]
        for i in client:
            newcli.append(i.text())
        newcli.append(vpro)
        #elimina duplicados
        var.pay = set(var.pay)
        for j in var.pay:
            newcli.append(j)
        newcli.append(sex)
        print(newcli)
    except Exception as error:
        print('Error: %s ' % str(error))
```

La lista `newcli` será la encargada de almacenar los valores. El primer **for es autoexplicativo**, carga todo el contenido de todos los `lineEdit`. Después añadimos el valor de la provincia que, al ser una variable global accedemos sin problema a ella desde cualquier función de la clase.

El código que necesita alguna explicación es la selección de los pagos que puede elegir el cliente. Cada vez que cliqueamos un **checkbox** este carga todos los marcados, el nuevo, más los anteriores con lo que nos aparecerían duplicados debido al `for` del `main.py`. De ahí la función python: **`var.pay = set(var.pay)`** que elimina los duplicados, y finalmente, a través de otra variable global cargamos el sexo del cliente. Esta primera parte lo que hace es prepararnos la lista con todos los valores que ha introducido el usuario de la aplicación.

A continuación volvemos a QtDesigner para incluir una **QTableWidget**, donde solo mostraremos el *DNI*, *Apellidos* y *Nombre*. En una tabla de gestión de este tipo no conviene cargar muchos datos ya que se convierte ilegible. Basta con los datos básicos para localizar la persona o producto o servicio que en este caso, podría ser con un código y su descripción.

PyQt permite las **View Model-Based**, o basadas en un modelo, pero su manejo es a nivel de código lo que la hace más complicado frente a las *View Item-Based*, basadas en el registro que van a mostrar y que será el que utilicemos. En entornos gráficos como PyGtk habituales en Gnome-Linux, las View se basan en el modelo pero este se construye en el diseñador gráfico cuando se diseña el prototipo, a diferencia de PyQt que obliga a realizar el diseño a nivel de código.

Añadimos entonces en la ventana principal mediante el QtDesigner un **QTableWidget** estableciendo el modelo tal como se muestra en las imágenes.

```
def showClients():
    try:
        #Preparamos el registro
        newcli = []
        clitable = [] # serán los datos que carguemos en la tabla
        client = [var.ui.editDni, var.ui.editApel, var.ui.editNome, var.ui.editData, var.ui.editDir]
        k = 0
        for i in client:
            newcli.append(i.text())
            #carguemos los valores para la tabla que solo tiene tres DNI, apellidos y nome
            if k < 3:
                clitable.append(i.text())
                k += 1
        newcli.append(vpro)
        #elimina duplicados
        var.pay = set(var.pay)
        for j in var.pay:
            newcli.append(j)
        newcli.append(sex)
        print(newcli)
        print(clitable)
        row = 0 #posición de la fila, problema: coloca al último como primero en cada click
        column = 0 #posición de la columna
        var.ui.cliTable.insertRow(row) #insertamos una fila nueva con cada click de botón
        for registro in clitable:
            #la celda tiene una posición fila, columna y cargamos en ella el dato
            cell = QtWidgets.QTableWidgetItem(registro) #carga en cell cada dato de la lista
            var.ui.cliTable.setItem(row, column, cell) #lo escribe
            column += 1
        except Exception as error:
            print('Error: %s ' % str(error))
```

GESTIÓN CLIENTES

DNI: ✓
Fecha Alta:

Apellidos:
Nome:

Dirección:
Provincia: Pontevedra ▼

Sexo: ☒ Femenino ☐ Masculino
 Métodos de Pago: ☒ Efectivo ☒ Tarjeta ☐ Transferencia

	DNI	Apellidos	Nombre
1	00000000T	Castro Vilela	María
2	11111111H	García Sánchez	Luis

Aceptar
Salir

Lo último que nos queda es un aspecto muy útil, y productivo, en listados con gran cantidad de datos. Consiste en que una vez marquemos con el ratón un registro del listado todos los datos de ese registro se carguen en los widgets correspondientes. Como aún no manejamos una base de datos que almacene todos nos ceñiremos exclusivamente a los tres principales que hay en la tabla.

El siguiente al que se llama *cargarCliente* permite lo indicado anteriormente. Fijarse en el código que hay dentro del *if* que permite ahorrar unas líneas de código en una sola. Se suele conocer como comprensión en lista y con ello se consigue crear listas "sobre la marcha".

También destaca el concepto de ***enumerate*** en Python. Este recurso obtiene al **recorrer una lista no solo su valor sino su posición también en la misma.**

```
def cargarCliente(self):
    try:
        fila = var.ui.cliTable.selectedItems()
        client = [var.ui.editDni, var.ui.editApel, var.ui.editNome]
        if fila:
            fila = [dato.text() for dato in fila]
            print(fila)
            i = 0
            for i, dato in enumerate(client):
                dato.setText(fila[i])
    except Exception as error:
        print('Error: %s ' % str(error))
```


Este código no funciona si no activamos la siguiente señal de **QTableWidget** en el módulo principal. Una vez que se activa, está a la "escucha o *listener* " del evento correspondiente que en este caso es hacer click sobre la tabla.

```
i.stateChanged.connect(clients.Cientes.selPago)
var.ui.cmbProv.activated[atr].connect(clients.Cientes.selProv)
var.ui.tableCli.clicked.connect(clients.Cientes.cargarCli)
var.ui.tableCli.setSelectionBehavior(QtWidgets.QTableWidget.SelectRows)
...
Llamada a módulos iniciales
...
```

Por último, otro código muy habitual es el contenido en el módulo *limpiardatos*. Como su nombre indica lo que hace que una vez se llama limpia los valores que hay en los entry o desactiva los otros widgets.

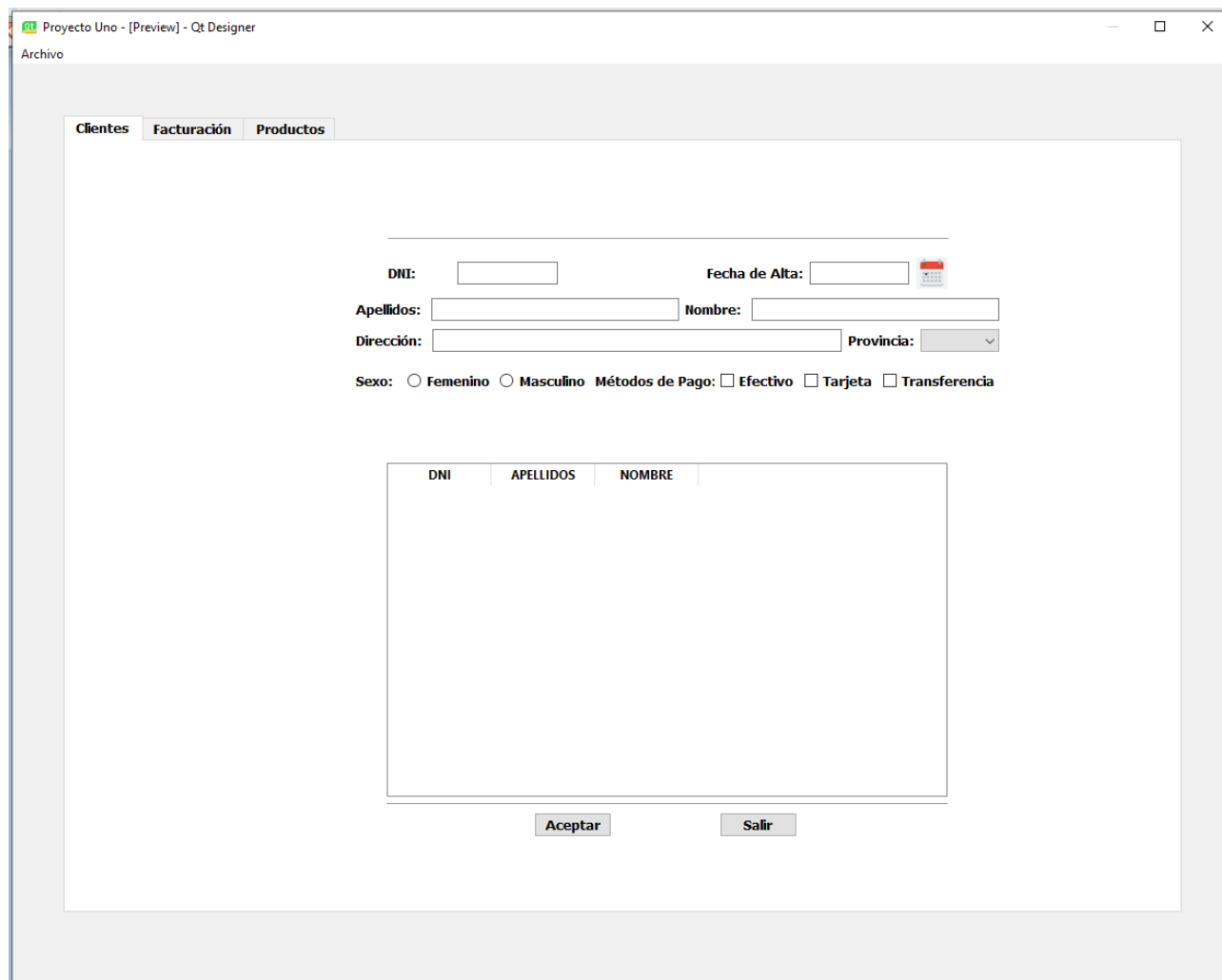
El código necesario, ya está comentado, sería el siguiente:

```
def limpiarCli(listaaeditCli, listaRbtsex, listaChkpay):
    """
    limpia los datos del formulario cliente
    :param listaRbtsex:
    :param listaChkpay:
    :return: none
    """
    try:
        for i in range(len(listaaeditCli)):
            listaaeditCli[i].setText('')
            var.ui.buttonGroup.setExclusive(False) #necesario para los radiobutton
        for dato in listaRbtsex:
            dato.setChecked(False)
        for data in listaChkpay:
            data.setChecked(False)
        var.ui.cmbProv.setCurrentIndex(0)
        var.ui.lblValidar.setText('')
    except Exception as error:
        print('Error cargar fecha: %s ' % str(error))
```


Paneles

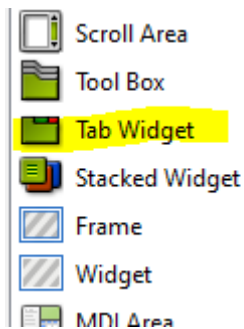
Los paneles es uno de los elementos que mejor permiten la ordenación de diferentes lógicas del negocio en una interfaz de usuario. Con el tiempo se ha ido tendiendo a evitar que el usuario haga el menos número de clicks posibles para llegar a un elemento de la interfaz y, sobre todo, eliminando el número de ventanas que hay que abrir y cerrar. Una aplicación que obligue a abrir un número elevado de ventanas para ejecutarla es una aplicación mal diseñada. Las ventanas que aparezcan, aparte de la principal, no deben ir más allá de ser simples avisos, mensajes o advertencias al usuario.

Para evitar todo ello se utilizan los **paneles o *QTabWidget***. Estos elementos permiten ir de un elemento a otro de la aplicación sin necesidad de abrir ventanas. Por ejemplo, para ir de clientes a facturación como en el ejemplo de la figura.

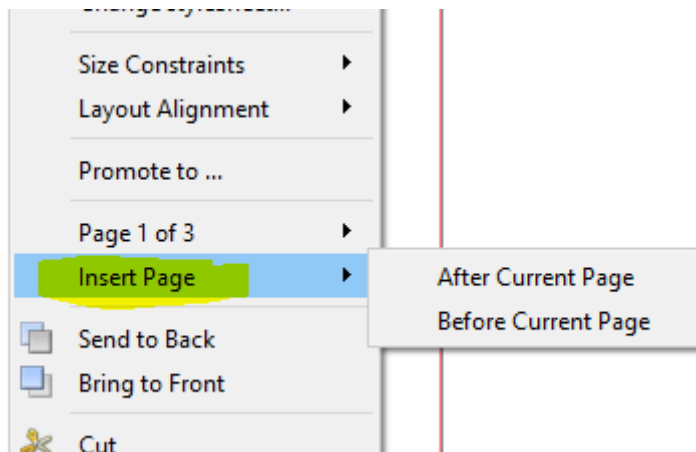


Como podemos observar para ir de un lugar a otro solo hay que clicar de una pestaña a otro.

Su manejo se entiende en las imágenes siguientes:



Es el widget TabWidget



Añadir paneles nuevos, botón derecho

movable	<input type="checkbox"/>
tabBarAutoHide	<input type="checkbox"/>
currentTabText	Cientes
translatable	<input checked="" type="checkbox"/>

Para cambiar el nombre a *currentTabText*

tabShape	Rounded
currentIndex	1
iconSize	16 x 16
elideMode	ElideNone
usesScrollButtons	<input checked="" type="checkbox"/>
documentMode	<input type="checkbox"/>
tabsClosable	<input type="checkbox"/>
movable	<input type="checkbox"/>
tabBarAutoHide	<input type="checkbox"/>
currentTabText	Facturación
translatable	<input checked="" type="checkbox"/>
disambiguation	
comment	
currentTabName	panelFac

Finalmente *currentIndex* nos indique que panel es el actual, empezando por el 0 y *currentTabName* su nombre

fichero .py cuando compilemos

Enlace a datos - I

PyQT 5 soporta la conexión con varios servidores *SQL* como: **SQLite**, MySQL, ODBC, y PostgreSQL, así como servidores *noSQL* como **MongoDB**. Para conectarnos a cualquier de estos servidores debemos instalar el driver correspondiente.

En nuestro caso usaremos dos servidores **SQLite** y **MongoDB**.

- **SQLite**

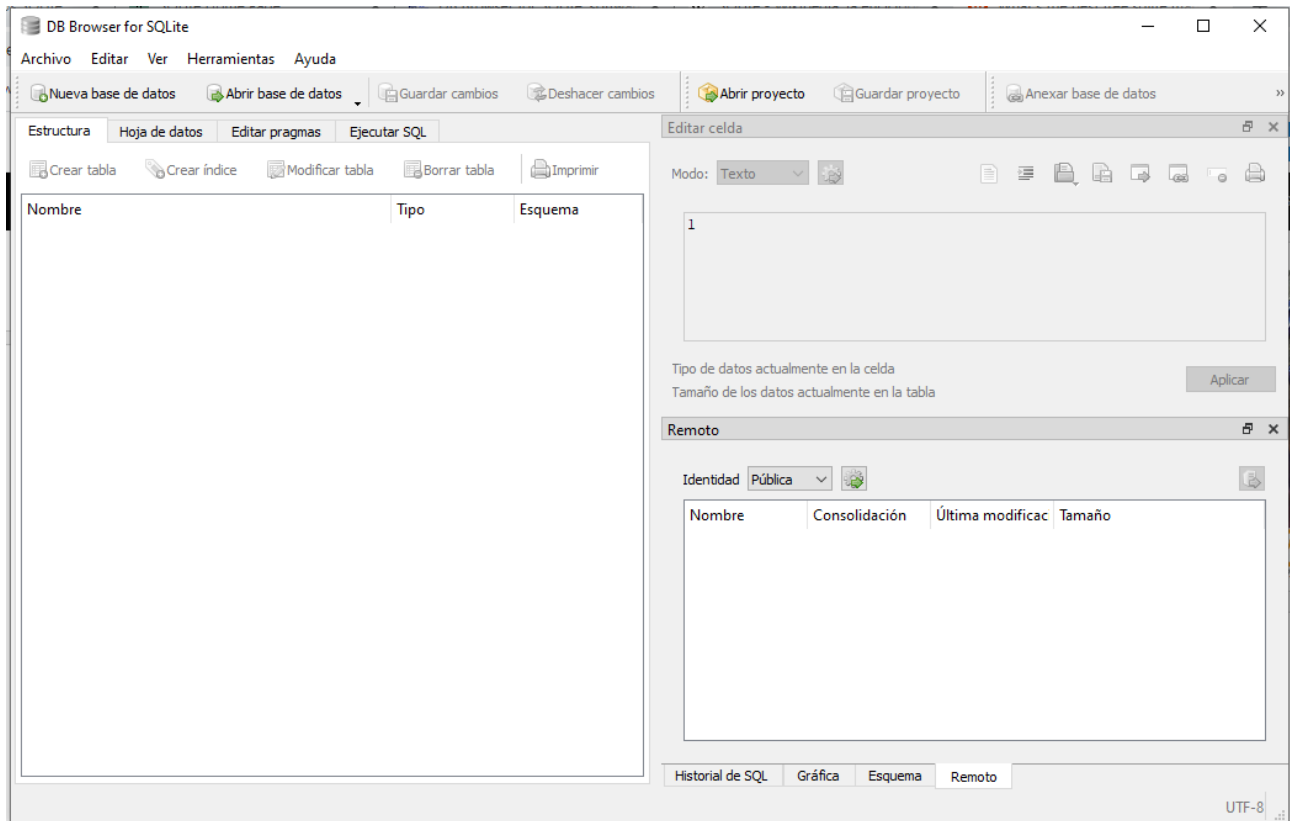
Es un sistema de base de datos relacional contenido en una biblioteca desarrollada en C que se integra en la aplicación a diferencia del resto y que se caracteriza por ser libre y rápido. Dicho de otro modo, tiene integrado el motor de base de datos SQL mientras los otros gestores el motor se alberga en un servidor independiente.

SQLite lee y escribe directamente en archivos de disco ordinarios, el contenido de una base de datos se almacena en un solo archivo de disco, lo que le proporciona una gran versatilidad y rendimiento incluso en entornos de poca memoria. Actualmente, la versión 3 permite archivos de hasta 2 Teras. Es el sistema gestor de base de datos implementado en las aplicaciones Android.

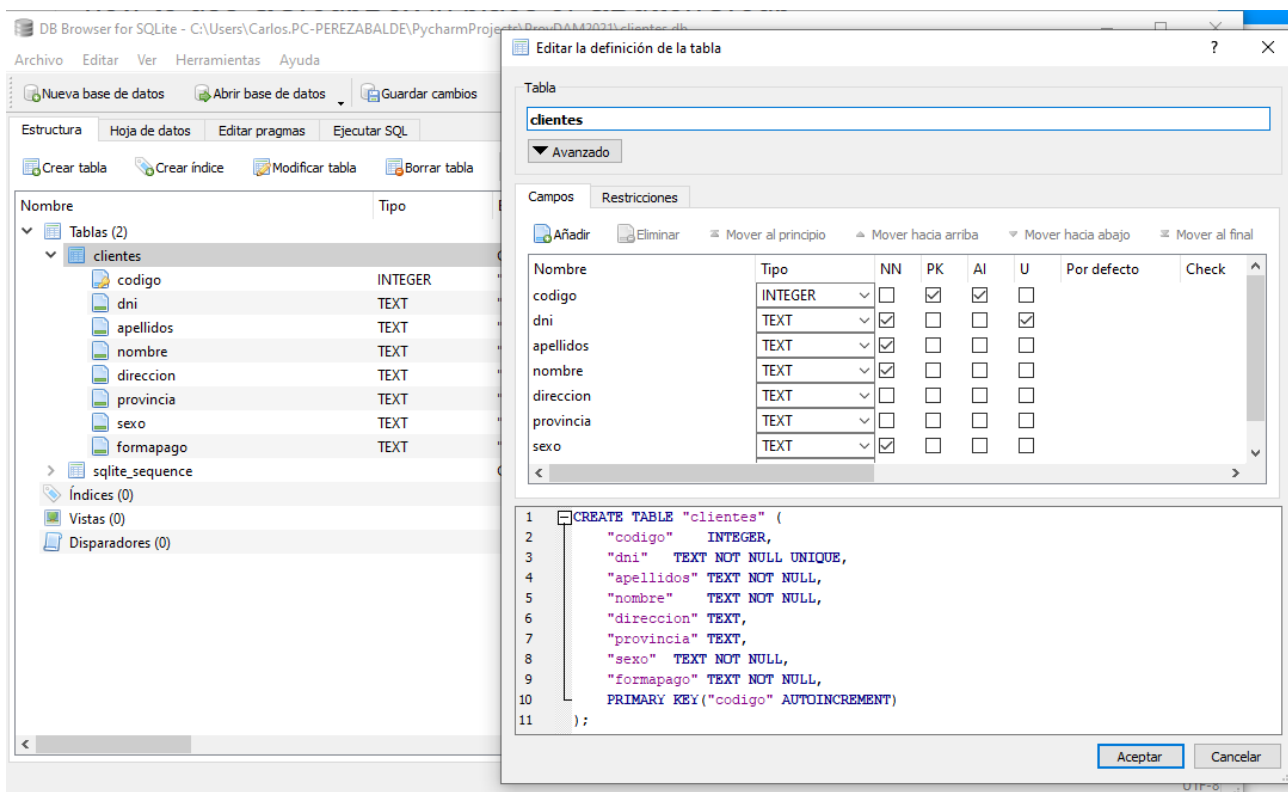
- **Instalación**

Su instalación en Windows es sencilla. Para ello vamos a la web <https://sqlitebrowser.org/dl/> y descargamos el instalador apropiado.

DBBrowser o SQLiteBrowser es una aplicación gratuita que facilita de forma sencilla e intuitiva la administración de bases de datos de forma gráfica. Existen otras opciones como SQLite Manager que es una extensión para Firefox o SQLiteStudio con alguna opción más. Si lanzamos DB Browser nos aparece la pantalla siguiente:



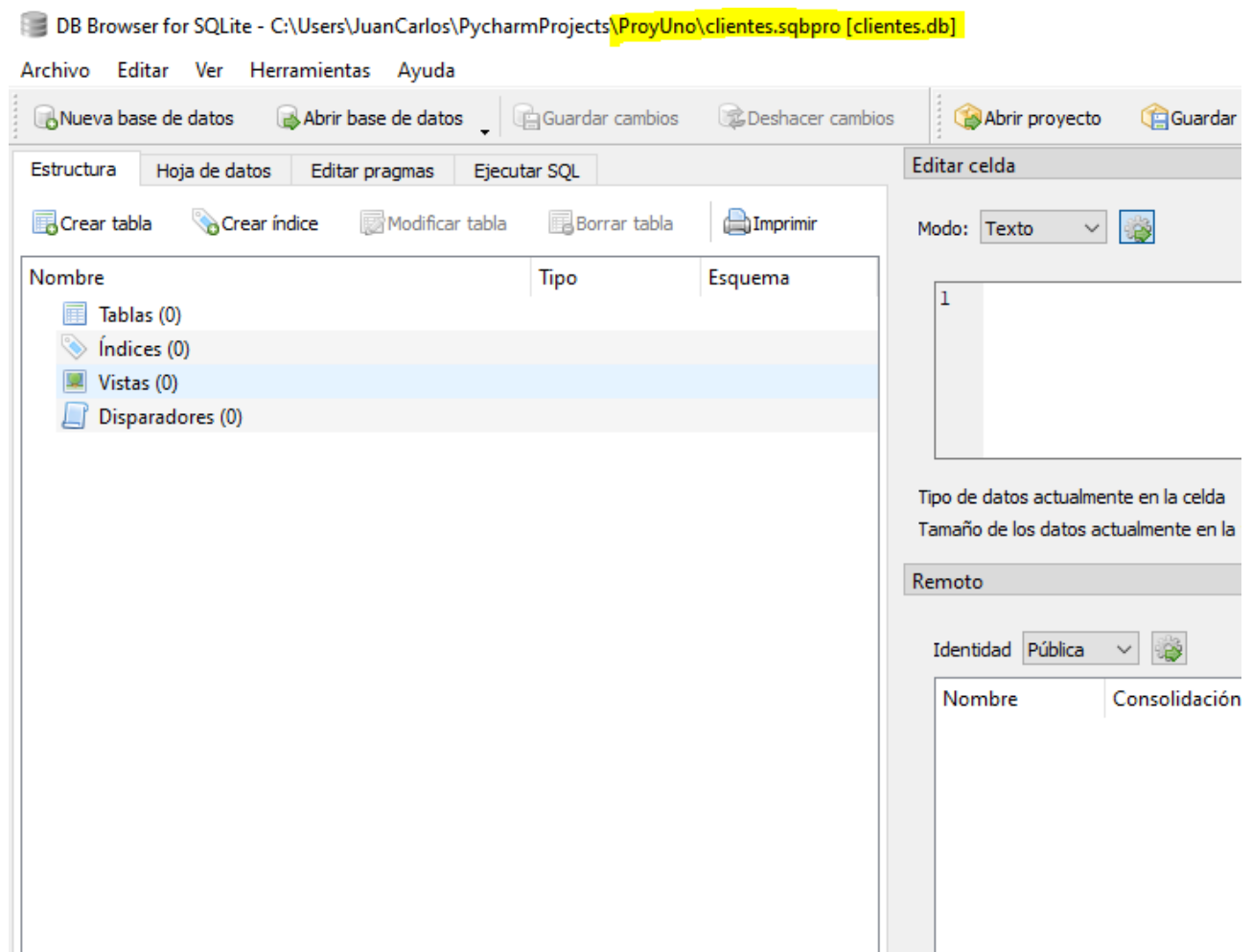
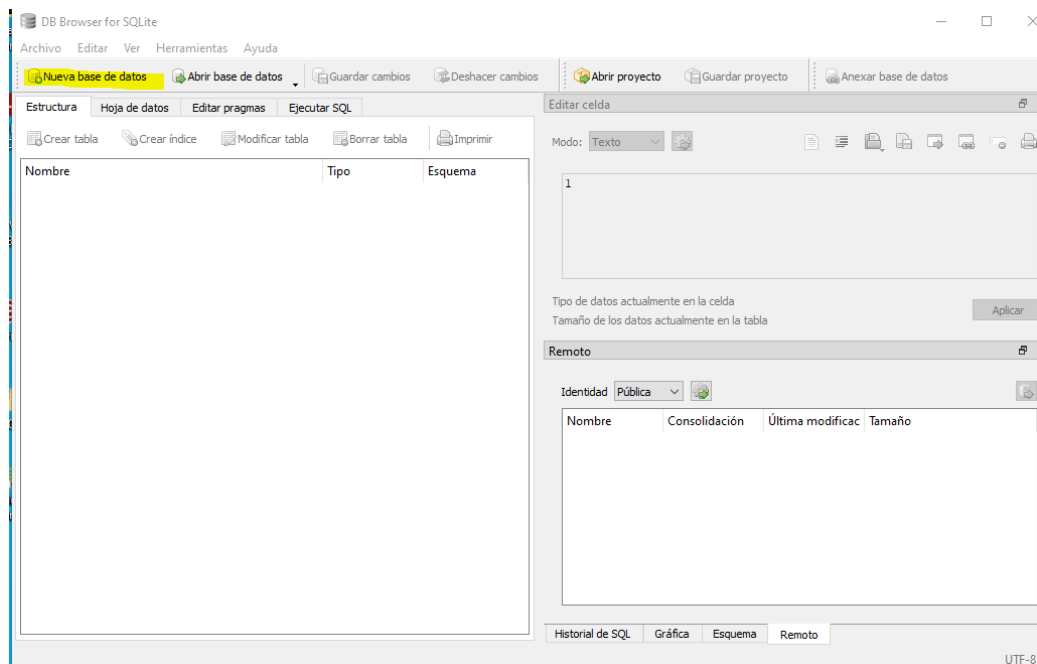
Aprovechamos y creamos la primera tabla de **clientes**. No nos olvidemos guardar los cambios.



Para establecer la conexión al servidor de bases de datos usaremos el método estático `QSqlDatabase.addDatabase('QSQLITE')` indicado el servidor que utilizaremos. En otros tipos de servidores tenemos: `QSQLITE`, `QMYSQL`, `QMYSQL3`, `QODBC`, `QODBC3`, `QPSQL`, y `QPSQL7`, y con el **método** `open()` para establecer la conexión y verificar la misma.

Por otro lado, en nuestro ejemplo, crearemos un fichero llamado *conexion.py*. Esta idea permite trasladar este fichero a otras aplicaciones simplemente cambiando el nombre de la base de datos.

En primer lugar, creamos la base de datos:



Para evitar rutas lo que haremos es guardar el fichero de la base de datos en el mismo directorio que la aplicación. En las aplicaciones no muy complejas con SQLite suele hacerse así para evitar complicaciones.

A continuación establecemos los códigos necesarios en *main.py*, *conexion.py* y *var.py*.

En *main.py* simplemente hacemos la llamada al módulo de conexión. No nos olvidemos importar el módulo en las llamadas a las librerías.

```
var.ui.Cachable.setSelectedBehavior(QtWidgets.QAbstractWidget.SelectRows)  
conexion.Conexion.db_connect(var.filebd)
```

Si nos fijamos, vemos una variable *var.filebd* que, claramente, hace referencia a la base de datos que hemos creado y que guardamos en el módulo *var.py*. No supone un adelanto especial con respecto a utilizar el nombre directamente. Pero, por ejemplo, podemos utilizar una base de pruebas y una vez finalizado el diseño y codificación del programa añadir una base "limpia". Otra opción sería, añadir este nombre asignándolo en el *main.py* a *filebd*, y este a su vez vacío en *var.py*. Así podríamos utilizar el programa para varias bases de datos diferentes.

```
dialogcat = None  
filebd = 'clientes.db'
```

Por último, nos queda el código principal, que es el módulo que se conecta a la base de datos y que sería:

El código es autoexplicativo. Simplemente observamos que permite crear una base de datos si no existiese *addDatabase*, pero en nuestro caso no sería necesario ya que está creada.

```
from PyQt5 import QtWidgets, QSql  
  
class Conexion():  
  
    def db_connect(filename):  
  
        db = QSql.QSqlDatabase.addDatabase('QSQLITE')  
        db.setDatabaseName(filename)  
        if not db.open():  
            QtWidgets.QMessageBox.critical(None, 'No se puede abrir la base de datos',  
                                           'No se puede establecer conexion.\n' 'Haz Click')  
            QtWidgets.QMessageBox.Cancel()  
            return False  
        else:  
            print('Conexión Establecida')  
        return True
```

Resultado:

```
Conexión establecida  
  
Process finished with exit code 0
```

•MongoDB

MongoDB es una base de datos **orientada a documentos**. Esto quiere decir que en lugar de guardar los datos en registros, **guarda los datos en documentos**. Estos documentos son almacenados **en BSON**, que es una **representación binaria de JSON**. Una de las diferencias más importantes con respecto a las bases de datos relacionales, **es que no es necesario seguir un esquema**. **Los documentos de una misma colección**, concepto similar a una tabla de una base de datos relacional, **pueden tener esquemas diferentes**. Es por ello que también se las conoce como **bases de datos NoSQL**.

Aunque este tipo de bases de datos están tomando mucha fuerza en aplicaciones que precisan el almacenamiento de información, no suelen ser recomendables en aquellas que precisen de **transacciones que precisen cumplir la propiedad ACID (atomicidad, consistencia, aislamiento y durabilidad)**, es decir aquellos procesos que precisen de varias operaciones (un ejemplo podría ser una transacción bancaria) o también, que precisen de estructuras **join**. Los primeros podrían llegar a simularse pero no es recomendable.

Empecemos por su instalación partiendo del hecho que en estamos en un máquina de 64bits:

```
Microsoft Windows [Version 10.0.17134.285]  
(c) 2018 Microsoft Corporation. All rights reserved.  
  
C:\WINDOWS\system32>wmic os get osarchitecture  
OSArchitecture  
64-bit  
  
C:\WINDOWS\system32>
```

En primer lugar vamos al siguiente enlace, [Download MongoDB on Windows](#) y descargamos.

Tras finalizar el paso siguiente es configurar el servidor de BBDD como un servicio. La mejor manera es utilizar una herramienta gráfica. El propio instalador trae MongoDB Compass

El código que nos permite conectarnos a la base de datos sería:

```

import pymongo, var

class Conexion():
    HOST = 'localhost'
    PORT = '27017'
    URI_CONNECTION = 'mongodb://' + HOST + ':' + PORT + '/'
    var.DATABASE = 'empresa'
    try:
        #var.client = pymongo.MongoClient(URI_CONNECTION)
        #var.client.server_info()
        print('OK -- Conectador al servidor %s' % HOST)
        print('Conexion base de datos establecida')
    except pymongo.errors.ServerSelectionTimeoutError as error:
        print('Error en la conexión MongoDB: %s' % error)
    except pymongo.errors.ConnectionFailure as error:
        print('No se puede conectar to MongoDB: %s' % error)

```

Como se puede observar, hay que instalar previamente la librería **pymongo**. Sin olvidarnos de llamarlo desde el módulo principal para abrir la conexión:

```

...
módulos conexion base datos
...

#conexion.Conexion.db_connect(var.filebd)
conexion.Conexion()

```

El resultado de que todo fue bien:

```

C:\Users\Carlos\AppData\Local\Programs\Python\Python38-32\python.exe C:/Us
OK -- Conectador al servidor localhost
conexion base de datos establecida

Process finished with exit code 0

```

Introducción

La palabra CRUD, siglas CRUD en programación es un acrónimo que hace referencia a:

- Create -> Crear. INSERT
- Read -> Leer. SELECT
- Update -> Actualizar. UPDATE
- Delete -> Borrar. DELETE

O, lo que es lo mismo, las operaciones que podemos realizar con una base de datos. Por cuestiones de tiempo, haremos solo referencia a SQLite, es decir, una base de datos SQL. Con MySQL, PostgreSQL o SQLServer por mencionar las más utilizadas, la única deferencia sería modificar la

conexión ya que, a diferencia de SQLite, son SGBD alojadas en servidores. Del resto las operaciones son similares. Hay muchos ejemplos en Internet para ver como se realiza la conexión

Con respecto a las bases de datos NoSQL, que han cogido mucha fuerza en los últimos años solo veremos la conexión ya analizada en la actividad anterior.

Empecemos pues.

•INSERTAR

Antes de comenzar vamos a realizar un par de cambios en los códigos anteriores que son necesarios.

El primero es el de la función **selPago()**.

Los cambios son necesarios para controlar que checks están marcados y que no. Además para evitar duplicidades si el usuario se dedicase a marcar y desmarcar repetidamente. Fijémonos además que devuelve una lista que contiene las formas de pago. Y por otro lado los hemos agrupado en un ButtonGroup tal como hicimos con los radiobutton.

```
def selPago():
    try:
        var.pay = []
        for i, data in enumerate(var.ui.grpbtnPay.buttons()):
            #agrupamos en QtDesigner los checkbox en un ButtonGroup
            if data.isChecked() and i == 0:
                var.pay.append('Efectivo')
            if data.isChecked() and i == 1:
                var.pay.append('Tarjeta')
            if data.isChecked() and i == 2:
                var.pay.append('Transferencia')
        #var.pay = set(var.pay)
        print(var.pay)
        return var.pay
    except Exception as error:
        print('Error: %s' % str(error))
```

Además necesitamos realizar un par de cambios más. En **showClientes** es donde recogemos esta función, **selPago()** que realice el testeo de los checkbox marcados, así como comprobar que los campos principales, nombre, apellidos y dni no están vacíos.


```

        k += 1
    newcli.append(vpro)
    var.pay2 = Clientes.selPago()
    newcli.append(var.sex)
    newcli.append(var.pay2)
    if client:
        #comprobamos que no esté vacío lo principal
        #aquí empieza como trabajar con la TableWidget
        row = 0
        column = 0
        var.ui.tableCli.insertRow(row)
        for registro in clitable:
            cell = QTableWidgetItem(registro)
            var.ui.tableCli.setItem(row, column, cell)
            column += 1

        conexion.Conexion.cargarCli(newcli)
    else:
        print('Faltan Datos')
        Clientes.limpiarCli(client, var.rbtsex, var.chkpago)
except Exception as error:
    print('Error cargar fecha: %s ' % str(error))

```

Vamos ahora a por las operaciones con la Base de Datos.

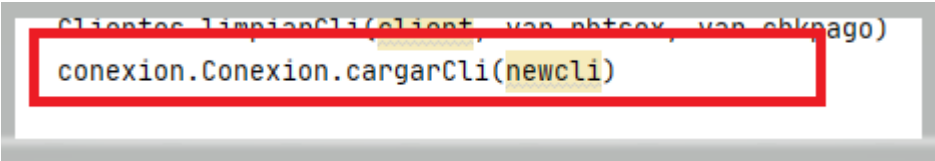
Hay muchas formas de insertar con el módulo QtSQL en Python. Elegimos la siguiente:

```

def cargarCli(cliente):
    query = QSqlQuery()
    query.prepare('insert into clientes (dni, apellidos, nombre, fechalta, direccion, pr
        'VALUES (:dni, :apellidos, :nombre, :fechalta, :direccion, :provincia, :
    query.bindValue(':dni', str(cliente[0]))
    query.bindValue(':apellidos', str(cliente[1]))
    query.bindValue(':nombre', str(cliente[2]))
    query.bindValue(':fechalta', str(cliente[3]))
    query.bindValue(':direccion', str(cliente[4]))
    query.bindValue(':provincia', str(cliente[5]))
    query.bindValue(':sexo', str(cliente[6]))
    # pagos = ' '.join(cliente[7]) si quisesemos un texto, pero nos viene mejor meterlo
    query.bindValue(':formasPago', str(cliente[7]))
    # print(pagos)
    if query.exec_():
        print("Inserción Correcta")
    else:
        print("Error: ", query.lastError().text())

```

Insistimos no nos olvidemos llamar a esta función desde **showClientes()**



```

Clientes limpiarCli(client, van_nhtsex, van_chkpago)
conexion.Conexion.cargarCli(newcli)

```

Convendría ahora hacer dos cambios por sentido común. El primer cambiar el nombre del botón *Aceptar* por el de **Alta Cliente** o simplemente **Alta**. El segundo cambio sería el nombre de la función *showClientes* por **altaClientes**, ya que es más intuitivo y relacionado con lo que veremos. Otra opción mantener ambas funciones de forma independiente.

•SELECT

Vamos a dividir este apartado en dos.

En primer lugar vamos a realizar una búsqueda general cuyo objetivo será que cuando iniciemos el programa se carguen los datos de todos los clientes en un listado. Este módulo lo llamaremos desde el principal. Su nombre será **mostrarClientes()**. En el fichero **conexion.py** generamos el módulo **mostrarClientes()**

```
main.py x var.py x clients.py x conexion.py x ventana.py x events.py x

def mostrarClientes(self):
    index = 0
    query = QSql.QSqlQuery()
    query.prepare('select dni, apellidos, nombre from clientes')
    if query.exec_():
        while query.next():
            dni = query.value(0)
            apellidos = query.value(1)
            nombre = query.value(2)
            var.ui.tableCli.setRowCount(index+1) # crea la fila y a continuación
            var.ui.tableCli.setItem(index,0, QTableWidgetItem(dni))
            var.ui.tableCli.setItem(index, 1, QTableWidgetItem(apellidos))
            var.ui.tableCli.setItem(index, 2, QTableWidgetItem(nombre))
            index += 1
        else:
            print("Error mostrar clientes: ", query.lastError().text())

# class Conexion():
```

Que llamamos desde main.py para que se ejecute al lanzar el programa.

```
main.py x var.py x clients.py x conexion.py x ventana.py x events.py x
52         i.toggled.connect(clients.Clientes.selSexo)
53     for i in var.chkpago:
54         i.stateChanged.connect(clients.Clientes.selPago)
55     var.ui.cmbProv.activated[str].connect(clients.Clientes.selProv)
56     var.ui.tableCli.clicked.connect(clients.Clientes.cargarCli)
57     var.ui.tableCli.setSelectionBehavior(QtWidgets.QTableWidget.SelectRows)
58     '''
59     Llamada a módulos iniciales
60     '''
61     events.Eventos.cargarProv()
62
63     '''
64     módulos conexion base datos
65     '''
66     conexion.Conexion.db_connect(var.filebd)
67     #conexion.Conexion()
68     conexion.Conexion.mostrarClientes(self)
69
70     def closeEvent(self, event):
71         if event:
72             events.Eventos.Salir(event)
```

Pero no solo eso si no que lo **lanzaremos también cada vez que insertemos un cliente para que se actualice la tabla** con todos los valores de la tabla clientes de la base de datos. Es decir,

```
main.py x var.py x clients.py x conexion.py x ventana.py x events.py x

    print('Conexión Establecida')
    return True

def cargarCli(cliente):
    query = QSqlQuery()
    query.prepare('insert into clientes (dni, apellidos, nombre, fe
                    'VALUES (:dni, :apellidos, :nombre, :fechalta, :dir
    query.bindValue(':dni', str(cliente[0]))
    query.bindValue(':apellidos', str(cliente[1]))
    query.bindValue(':nombre', str(cliente[2]))
    query.bindValue(':fechalta', str(cliente[3]))
    query.bindValue(':direccion', str(cliente[4]))
    query.bindValue(':provincia', str(cliente[5]))
    query.bindValue(':sexo', str(cliente[6]))
    # pagos = ' '.join(cliente[7]) si quisesemos un texto, pero no
    query.bindValue(':formas_pago', str(cliente[7]))
    # print(pagos)
    if query.exec_():
        print("Inserción Correcta")
        Conexion.mostrarClientes(self)
    else:
        print("Error: ", query.lastError().text())
```

•Eliminar

El código para un registro de la base de datos es muy sencillo. Tendremos que tener un elemento que o bien es la clave primaria, en nuestro caso el código, o bien podemos utilizar en este caso de personas el DNI ya que no deja de ser único para cada registro. El código sería:

En el módulo **clientes.py** estaría la función **bajaCliente** que llama a **bajaCli** del módulo **conexion.py**

```
main.py x var.py x clients.py x conexion.py x ventana.py x events.py x

def bajaCliente(self):
    """
    módulos para dar de baja un cliente
    :return:
    """
    try:
        dni = var.ui.editDni.text()
        conexion.Conexion.bajaCli(dni)
        conexion.Conexion.mostrarClientes(self)
        Clientes.limpiarCli()

    except Exception as error:
        print('Error cargar clientes: %s ' % str(error))
```

```
var.py x clients.py x conexion.py x ventana.py x events.py x

print("Error mostrar clientes: ", query.lastError().text())

def bajaCli(dni):
    """
    módulo para eliminar cliente. se llama desde fichero clientes.py
    :return None
    """
    query = QSql.QSqlQuery()
    query.prepare('delete from clientes where dni = :dni')
    query.bindValue(':dni', dni)
    if query.exec_():
        print('Baja cliente')
        var.ui.lblstatus.setText('Cliente con dni ' + dni + ' dado de baja')
    else:
        print("Error mostrar clientes: ", query.lastError().text())
```

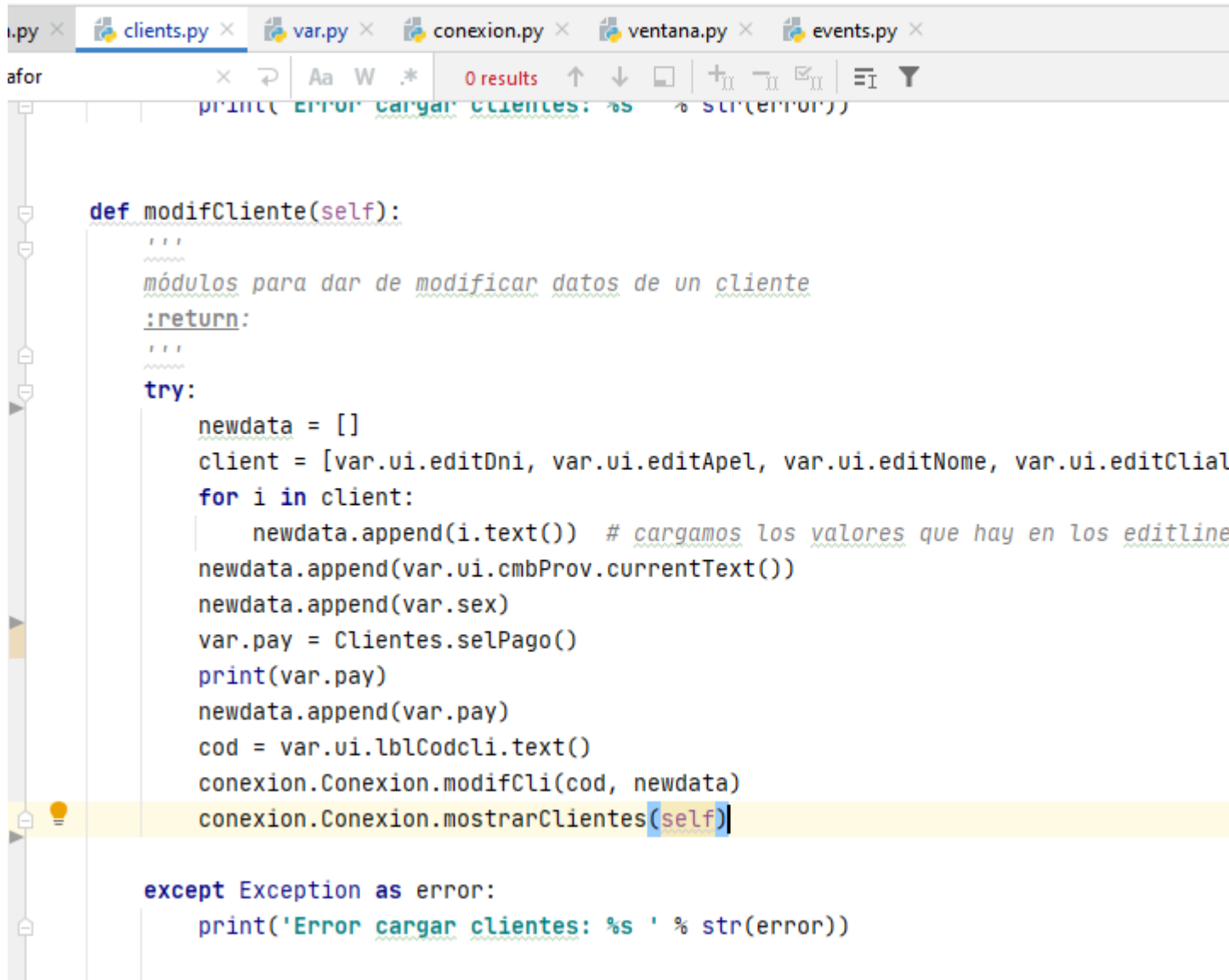
No podemos olvidarnos de llamar a **limpiarCli** para restaurar el formulario y recargar con **mostrarClientes** la lista con la actualización hecha. Todo ello desde **bajaCliente** del fichero **clientes.py**

Tampoco debemos olvidarnos de conectar el botón correspondiente del **main.py** con **bajaCliente**.

•Modificar

Nos queda modificar o update. Tengamos en cuenta que aquí podemos modificar todo menos la clave primaria o código del cliente. La forma más cómoda es mostrar los datos del cliente modificar todo o todos aquellos datos que deseamos y cargarlos todos aunque algunos no se hayan cambiado.

Los códigos sería, en **clientes.py**:

The image shows a code editor with several tabs at the top: 'clientes.py', 'var.py', 'conexion.py', 'ventana.py', and 'events.py'. The 'clientes.py' tab is active. The code in the editor is as follows:

```
def modifCliente(self):  
    """  
    módulos para dar de modificar datos de un cliente  
    :return:  
    """  
    try:  
        newdata = []  
        client = [var.ui.editDni, var.ui.editApel, var.ui.editNome, var.ui.editClia]  
        for i in client:  
            newdata.append(i.text()) # cargamos los valores que hay en los editline  
        newdata.append(var.ui.cmbProv.currentText())  
        newdata.append(var.sex)  
        var.pay = Clientes.selPago()  
        print(var.pay)  
        newdata.append(var.pay)  
        cod = var.ui.lblCodcli.text()  
        conexion.Conexion.modifCli(cod, newdata)  
        conexion.Conexion.mostrarClientes(self)  
  
    except Exception as error:  
        print('Error cargar clientes: %s ' % str(error))
```

Y el correspondiente a **modifCli** de **conexion.py**.


```
py x clients.py x var.py x conexion.py x ventana.py x events.py x

def modifCli(codigo, newdata):
    """
    modulo para modificar cliente. se llama desde fichero clientes.py
    :return None
    """
    query = QSql.QSqlQuery()
    codigo = int(codigo)
    query.prepare('update clientes set dni=:dni, apellidos=:apellidos, nombre=:nom
    'direccion=:direccion, provincia=:provincia, sexo=:sexo, formaspago=:formaspago')
    query.bindValue(':codigo', int(codigo))
    query.bindValue(':dni', str(newdata[0]))
    query.bindValue(':apellidos', str(newdata[1]))
    query.bindValue(':nombre', str(newdata[2]))
    query.bindValue(':fechaalta', str(newdata[3]))
    query.bindValue(':direccion', str(newdata[4]))
    query.bindValue(':provincia', str(newdata[5]))
    query.bindValue(':sexo', str(newdata[6]))
    query.bindValue(':formaspago', str(newdata[7]))
    if query.exec_():
        print('Cliente modificado')
        var.ui.lblstatus.setText('Cliente con dni ' + str(newdata[0]) + ' modificado')
    else:
        print("Error modificar cliente: ", query.lastError().text())
```

Nota.- Un aspecto a tener en cuenta que ha sucedido en este módulo es que el orden en que estaban los *checkbox* en **ventana.py**. En este caso el transformador *pyuic* no los colocaba en el mismo orden en que estaban en la interfaz gráfica diseñada con Designer, con lo que se "chekeaban" de forma diferente a lo buscado por el usuario al ejecutar el programa.

•Actividades de repaso

En la imagen vemos dos botones añadidos. El primero consiste en una lupa que nos permitiría **realizar una búsqueda** de un determinado cliente en la lista si fuese muy larga mostrando todos sus datos en el formulario y además a ese cliente solo en la tabla por si quisiésemos consultar, eliminar o modificar algo del mismo.

El segundo es un botón **reload o recargar** que nos volvería a recargar todos los clientes y limpiar el formulario.

Con lo expuesto hasta ahora se deja estas dos opciones para realiza de forma autónoma.

Clientes Facturación Productos

6

DNI:

00000000T



Apellidos: Gómez

Dirección: Calle Rúa, 23

Sexo:

☒ Femenino☐ Masculino

	DNI	APELLIDOS
1	11111111H	Soler
2	33333333P	Couto Mariño
3	44444444A	Solla
4	00000000T	Gómez
5	00000000A	Pepito
6	66666666Q	Gómez Julián
7	45245625W	Dosi
8	23123233E	Dato
9	23424324C	Romero

Grabar

Modificar

Eliminar