Editor de textos. Explicación completa

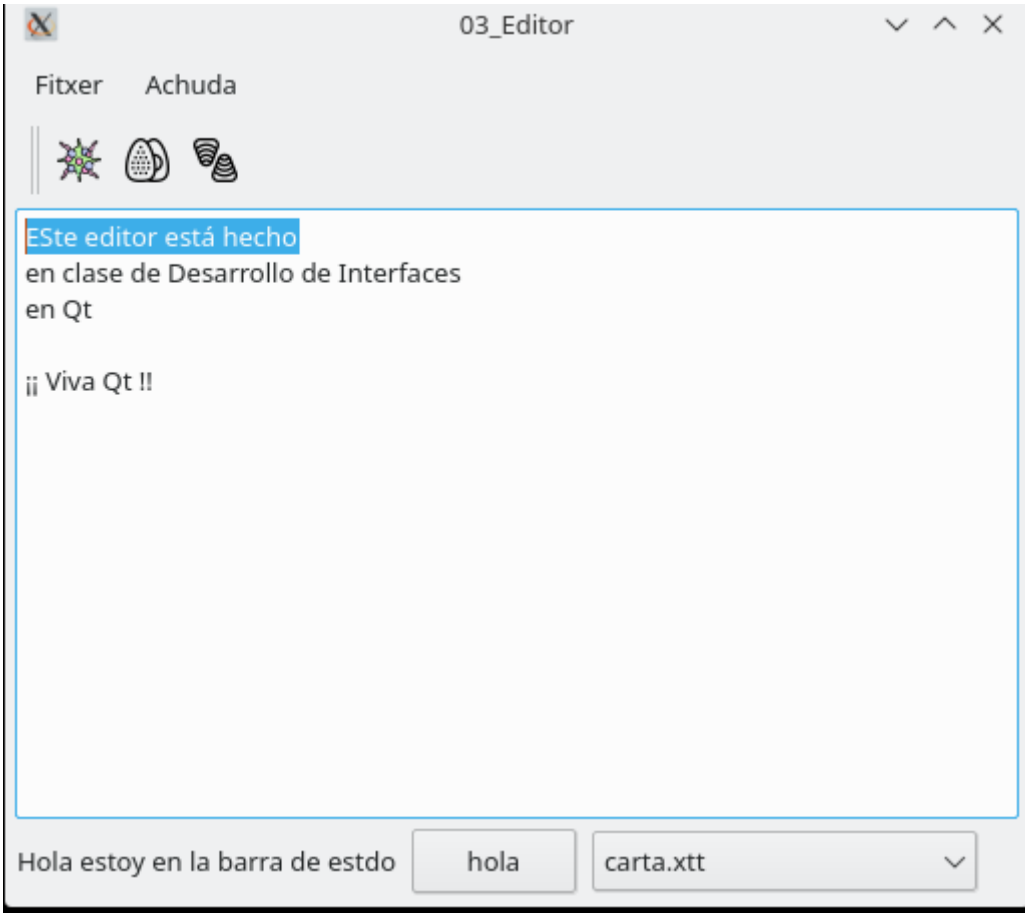
El siguiente script prepara la compilación cada vez que hay un cambio en los ficheros del proyecto.

script "haz.sh"

```
#!/bin/bash
make clean
rm *.pro
qmake -project
fichero=$( ls *.pro )
echo "QT += widgets" >> $fichero
qmake
make
```

Objetivo a largo plazo

Hacer un editor de texto como éste:



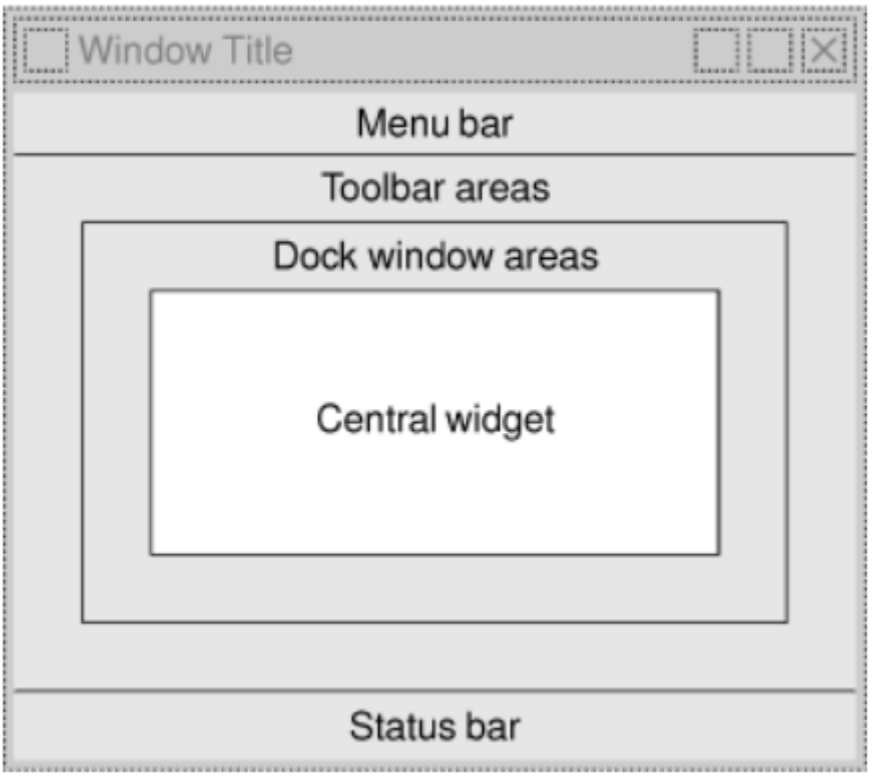
Ventana Principal

Como ya se ha hecho antes, el trabajo consiste en **crear e implementar una clase que herede**, extienda y amplíe de otra clase base que proporciona la funcionalidad básica. En casos anteriores, para hacer diálogos, hemos extendido `QDialog` , pero no en este ejemplo. Ahora necesitamos más… llamaremos a nuestra clase **`VentanaPrincipal`**. y no será un QDialog.

`QMainWindow` es una clase que hereda de `QWidget` y por tanto es un elemento gráfico. Es semejante a QDialog en el sentido en que muestra una ventana dentro de la cual se pueden insertar otros widgets mediante layouts.

La diferencia que destaca a `QMainWindow` es que está pensada para ser la **ventana principal** de una aplicación estándar y por ello dispone de elementos adicionales que facilitan la creación de aplicaciones de escritorio. En concreto dispone de áreas o zonas reservadas para menús, barras de herramientas, barras de estado y áreas de disposición de subventanas. También tiene slots y señales especiales .

Las zonas de una `QMainWindow` se organizan de la siguiente forma:



Widget en área central

El área central es la zona de la Ventana donde aparecerá el widget de trabajo principal. Aquí se suele insertar un widget complejo (habitualmente sólo uno). En nuestro caso vamos a poner un componente que es en sí un editor de textos (`QTextEdit`)

Ya tenemos lo necesario para iniciar la actividad y crear una Ventana principal mínima.

ventanaprincipal.h

```
#ifndef VENTANAPRINCIPAL_H
#define VENTANAPRINCIPAL_H

#include <QMainWindow>
#include <QTextEdit>

class VentanaPrincipal : public QMainWindow {
Q_OBJECT
public:
	VentanaPrincipal(QWidget * parent = 0, Qt::WindowFlags flags = 0);

private:
	QTextEdit *editorCentral;

};
```

```
#endif
```

ventanaprincipal.cpp

```
#include "ventanaprincipal.h"
#include <QStatusBar>
#include <QMenuBar>
#include <QMessageBox>

VentanaPrincipal::VentanaPrincipal(
      QWidget * parent ,
      Qt::WindowFlags flags ) : QMainWindow(parent,flags) {

      editorCentral = new QTextEdit(this);
      setCentralWidget(editorCentral);

    setWindowIcon(QIcon(":/images/icon.png"));
}
```

El constructor que creamos para nuestra clase, otra vez, lo hacemos mirando e imitando el constructor de la clase padre QMainWindow. Viendo la ayuda podemos determinar que nuestro constructor será el siguiente:

```
VentanaPrincipal(QWidget * parent = 0, Qt::WindowFlags flags = 0);
```

main.cpp

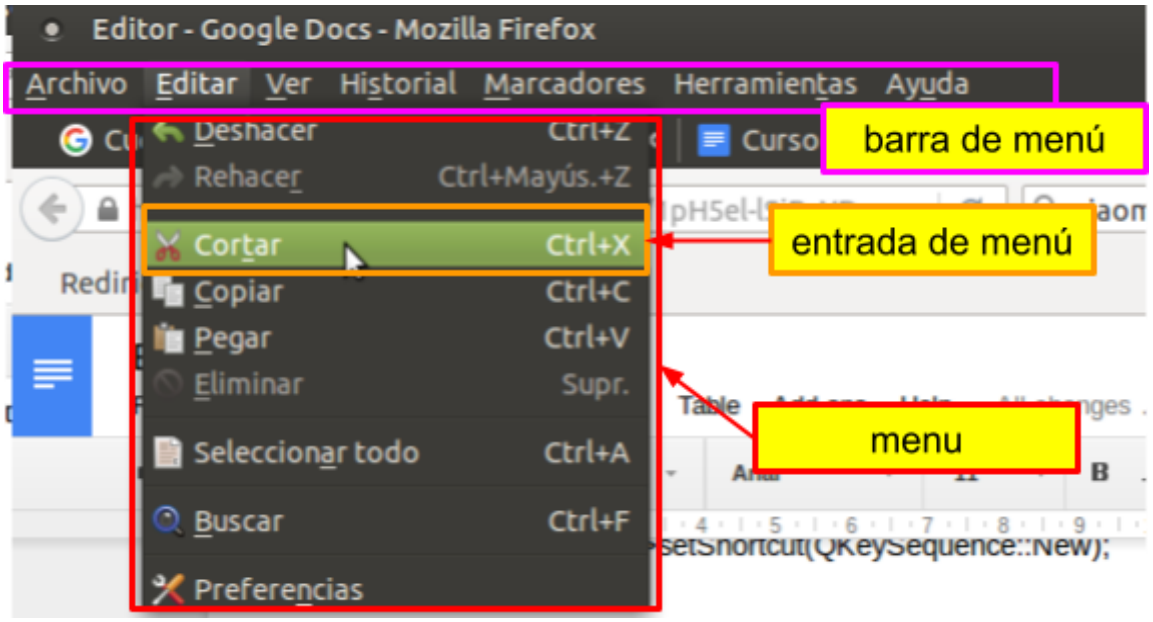
Este fichero no cambia significativamente respecto al resto de actividades. Se crea un objeto de tipo `VentanaPrincipal`, se muestra y se inicia Qt (app.exec()).

```
#include <QApplication>

#include "ventanaprincipal.h"

int main(int argc, char *argv[])
{
      QApplication app(argc, argv);
      VentanaPrincipal *finestra = new VentanaPrincipal;
      finestra->show();
      return app.exec();
}
```

Menús

La primera característica de QMainWindow la que nos aprovecharemos son los menús,

Un menú es una lista de opciones elegibles por un usuario para que el programa realice alguna operación. En una aplicación hay varios menús que se despliegan desde una barra en la parte superior. Llamamos barra de menú a esa franja. Cada menú está compuesto de diferentes líneas que son "entradas de menú".



QAction

Si observas las aplicaciones, verás que muchas de las acciones que pretendes conseguir con los menús de la barra de menús, se pueden conseguir de formas alternativas:

- Con una combinación de teclas. Por ejemplo, la opción buscar de los editores de texto se lanza con "F3" también
- En otro menú diferente al de la barra de menús.
- Con un botón de alguna barra de herramientas,
- automáticamente debido a algún evento, por ejemplo guardar documento modificado al intentar salir

En Qt, se introduce el concepto de QAction que representa alguna acción que puede ser lanzada de varias maneras, siendo una de ellas el menú. Por tanto, **para hacer menús, hay que hacer QActions.** Las `QActions` terminarán siendo las *entradas de menú*.

Los Elementos de un `QAction` son los necesarios para que aparezcan en un menú y sea fácil crear los atajos de teclado.

- texto (que aparecerá en el menú)
- imagen

- atajo
- "Tip" o descripción emergente

Lo bueno de las `QAction` es que cuando el proceso está en marcha y se activan (por ejemplo, haciendo click en el menú), se genera una señal y tan sólo hemos de conectarla a un slot para lograr una respuesta por parte de programa. En definitiva queremos: "Que cuando el usuario haga click en el menú, se ejecute un pedazo de código que hemos previsto"

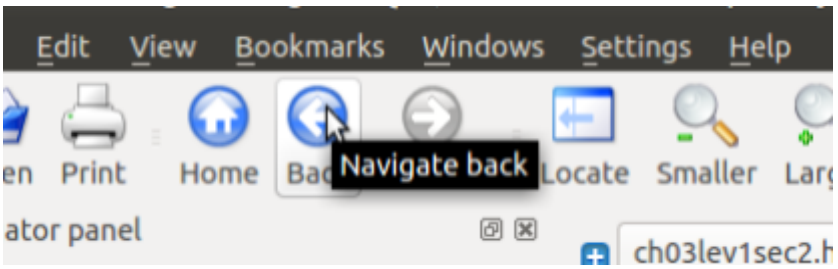
Todas las QAction tienen la señal `triggered()` como la señal a emitir cuando son activadas.

Con todo ello, la inicialización de una QAction se puede ver en el código siguiente

`ventanaprincipal.cpp`	**creación de una QAction o entrada de menú**

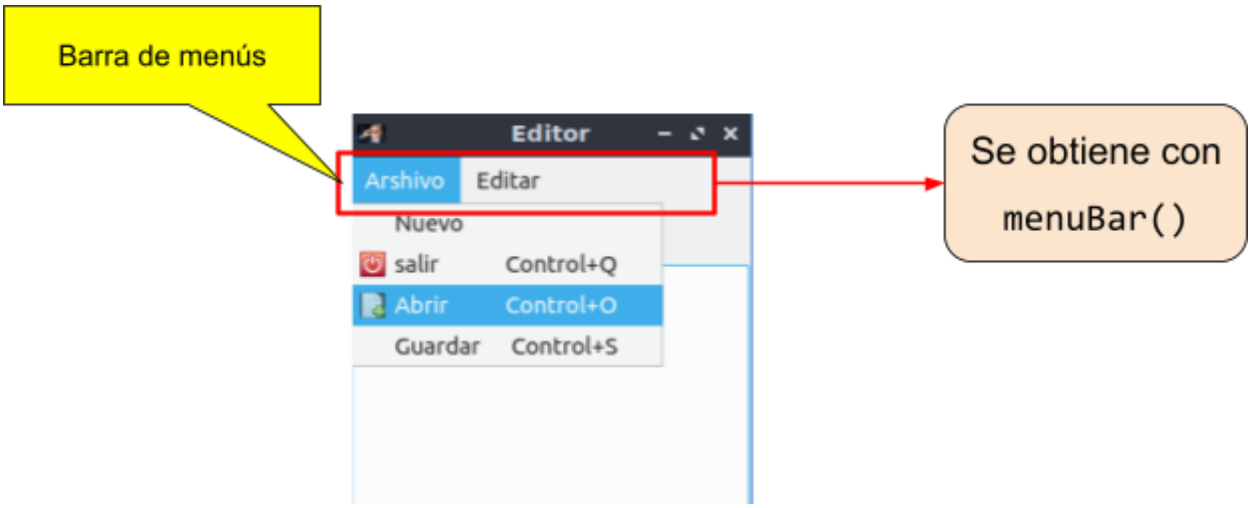
```
newAction = new QAction(tr("&New"), this);
newAction->setIcon(QIcon(":/images/new.png"));
newAction->setShortcut(QKeySequence::New);
newAction->setStatusTip(tr("Crear un nuevo documento"));
connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
```

Los tips son pequeñas descripciones emergentes que aparecen cuando el ratón se detiene encima de un botón:

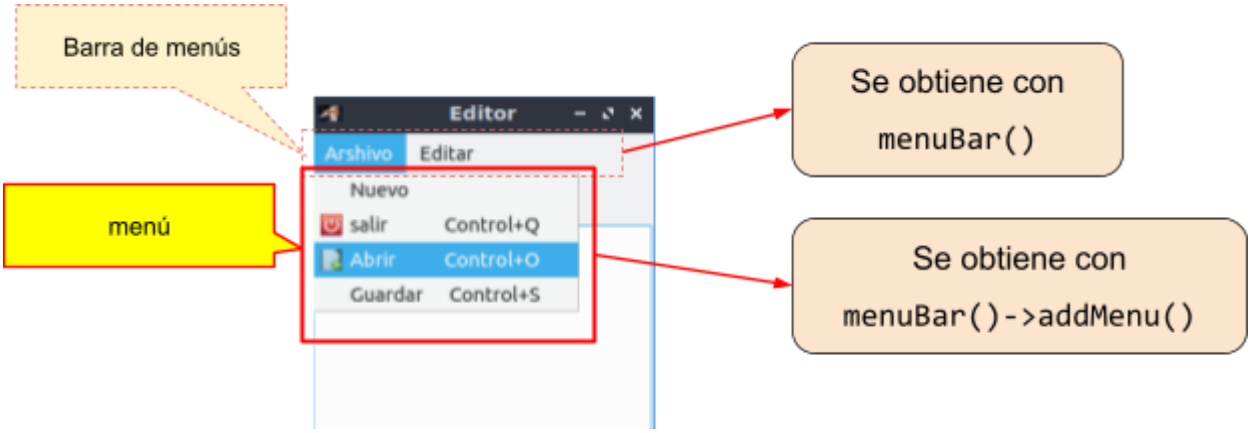


Creación de menus.

Una ventana de tipo `QMainWindow` ya tiene una barra de menús. El caso es que si no se ha puesto ningún menú, no es visible (¿Para qué mostrar algo que está vacío?) pero existir… existe. Y hay un método que te devuelve el puntero a dicha barra de menú (método menuBar(). Este método se puede usar directamente para poner un nuevo menú, que es automáticamente creado y devuelto.



Cuando se tiene la referencia de la barra de menús de la aplicación, se puede crear un nuevo menú



Obtener la barra de menú es tan sencillo como:

```
QMenuBar *barraPrincipal = menuBar();
```

Ahora añadir un menú nuevo es fácil… ¡ lo hace la barra de menú por nosotros y tan sólo hemos de recoger el puntero devuelto!

```
QMenu * fileMenu;
fileMenu = menuBar()->addMenu("Fishero");
```

En el siguiente código puedes ver cómo crear un menú y obtener el puntero para seguir manipulándolo y poniendo entradas de menú. (Ignora parcialmente lo referente a QAction en el código de abajo)

```
QMenu * fileMenu;                          //preparamos varirable
...
QAction * newAction = new QAction("Nuevo...");   // creamos QAction
fileMenu = menuBar()->addMenu(tr("&File"));      //creamos menú!!
fileMenu->addAction(newAction);                  //añadimos entrada al
                                                 // menú anteriore
```

Ahora que ya hemos visto lo básico sobre los menús vamos a ampliar el ejercicio haciendo todo el detalle para crear realmente un menú y nuestra primera entrada de menú funcional.

Un primer Menú. "Salir"

Para añadir un menú, es clave crear una QAction (que será realmente la entrada del menú). Los pasos son los siguientes

Declaramos en ventanaprincipal.h una nueva QAction (un puntero de momento)

```
QAction *accionSalir;
```

y por tanto , en ese fichero:

```
#include <QAction>
```

(necesitarás más, pero ya vendrán los otros includes)

en ventanaprincipal.cpp añadimos el siguiente código que pasamos a explicar:

ventanaprincipal.cpp

```
 accionSalir = new QAction(tr("Salir"));
    accionSalir->setIcon(QIcon(":/images/icon.png"));
    accionSalir->setShortcut(tr("Ctrl+Q"));
//tambien   accionSalir->setShortcut(QKeySequence::Close);
    accionSalir->setStatusTip(tr("apagar y marcharme a casa"));
    connect(accionSalir,SIGNAL(triggered()), this, SLOT(close()));
```

Explicación de los pasos

1. `accionSalir = new QAction(tr("Salir"))`: Creamos la acción (hasta ahora sólo teníamos el puntero). A la acción le pasamos un texto, en este caso "Salir"

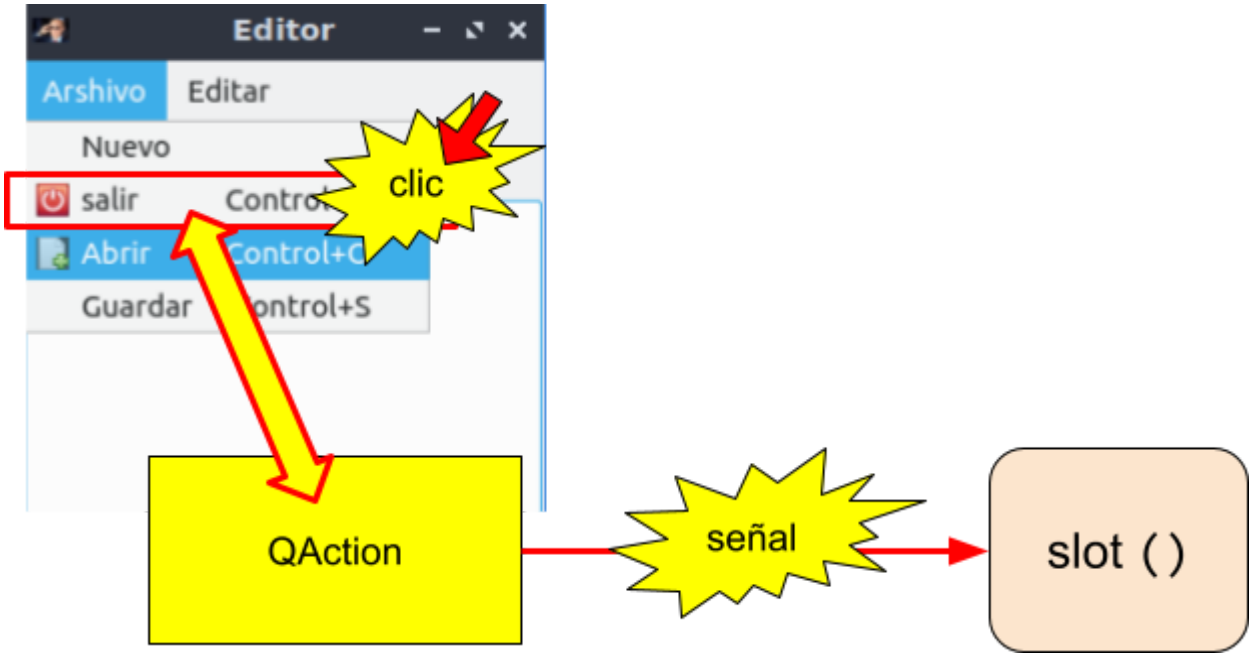
2. `accionSalir->setIcon(QIcon(":/images/icon.png"))`: Añadimos un icono para que se muestre junto al texto (tanto en el menú como en otros lugares donde aparecerá la QAction

3. `accionSalir->setShortcut(tr("Ctrl+Q"));` Algunas acciones pueden ser invocadas mediante atajos de teclado. Éstos se pueden establecer con este método, el atajo se puede especificar de varias maneras, aquí con un QString "Ctrl+Q"

4. `accionSalir->setStatusTip(tr("apagar y marcharme a casa"))`: La Acción aparecerá como elemento del menú, pero también podrá hacerlo de otras formas que veremos más adelante. En algunas, al pasar el ratón por encima del menú o boton de la acción, aparecerá un consejo emergente (o "tip") que aquí especificamos

5. `connect(accionSalir,SIGNAL(triggered()), this, SLOT(close()))`: Esta es la parte principal del establecimiento del menú. Aquí indicamos que al ser activada la QAction (en este caso mediante el clic en el menú), se ejecute un slot llamado close() d

   El slot "close()" ya está hecho y pertenece a la clase QMainWindow… de la cual heredamos



Ya tenemos la QAction creada, ahora la usaremos al crear el menú

Para ello, debemos acceder a la barr de menús y crearnos un menú. Esto se consigue de la siguiente manera:

```
menuBar()->addMenu(tr("Archivo"));
```

menuBar() devuelve un puntero a la barra de menús de la aplicación, que resulta ser un puntero a un objeto de tipo QMenuBar. Este objeto puede ser manipulado para añadir un menú

nuevo. El método devuelve un puntero al menú creado. Esto podemos verlo en la ayuda correspondiente a este método que se muestra aquí:

QMenu * QMenu::addMenu (const QString & *title*)

Por lo que la forma correcta de usarlo es algo así:

```
QMenu * menuSalir;
menuSalir = menuBar()->addMenu(tr("Archivo"));
```

y por ello hay que añadir lo siguiente en ventanaprincipal.cpp (o .h)

```
#include <QMenu>
```

pero también !!!

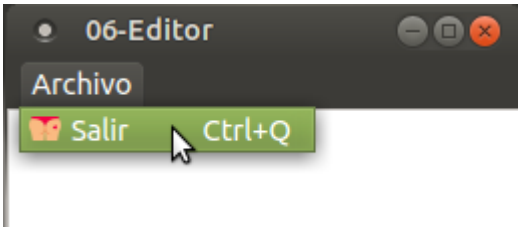
```
#include <QMenuBar>
```

Finalmente, con el menú creado, añadimos la nueva entrada de menú o QAction.

```
menuSalir->addAction(accionSalir);
```

```
#include "ventanaprincipal.h"
#include <QStatusBar>
#include <QMenuBar>
#include <QMessageBox>

VentanaPrincipal::VentanaPrincipal(
      QWidget * parent ,
      Qt::WindowFlags flags ) : QMainWindow(parent,flags) {

      editorCentral = new QTextEdit(this);
      setCentralWidget(editorCentral);

    setWindowIcon(QIcon(":/images/icon.png"));

    accionSalir = new QAction(tr("Salir"),this);
    accionSalir->setIcon(QIcon(":/images/icon.png"));
    accionSalir->setShortcut(tr("Ctrl+Q"));
    accionSalir->setStatusTip(tr("apagar y marcharme a casa"));
    connect(accionSalir,SIGNAL(triggered()), this, SLOT(close()));

    // QMenu * menuSalir
    menuSalir = menuBar()->addMenu(tr("Archivo"));
    menuSalir->addAction(accionSalir);

}
```

```
#ifndef VENTANAPRINCIPAL_H
#define VENTANAPRINCIPAL_H

#include <QMainWindow>
#include <QTextEdit>
#include <QMenu>
#include <QAction>
#include <QMenuBar>

class VentanaPrincipal : public QMainWindow {
Q_OBJECT
public:
      VentanaPrincipal(QWidget * parent = 0, Qt::WindowFlags flags = 0);

private:
      QTextEdit *editorCentral;

    QMenu * menuSalir ;
    QAction *accionSalir;
};

#endif
```



Poblar los menús adecuadamente

Para no enmarranar el código en el constructor, vamos a añadir a la clase dos métodos auxiliares donde se recogerá el código que cree las QAction por una parte y los menús por otra

```
createActions();
createMenus();
```

son métodos normales, auxiliares (privados preferentemente)

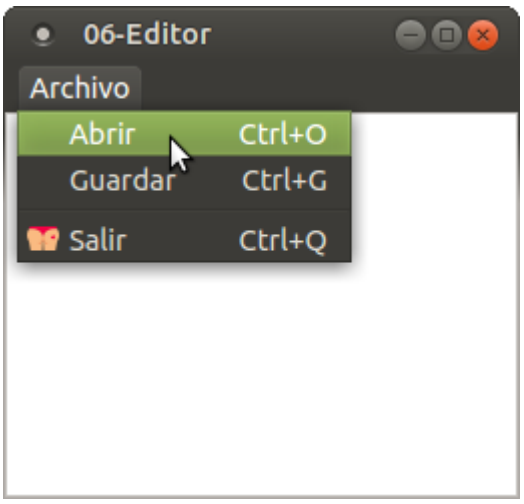
createActions() tendrá en el futuro el código de la creación de acciones y la conexión de sus señales a diferentes slots

```
void VentanaPrincipal::createActions() {
```

```
    accionSalir = new QAction(tr("Salir"),this);
    accionSalir->setIcon(QIcon(":/images/icon.png"));
    accionSalir->setShortcut(tr("Ctrl+Q"));
    accionSalir->setStatusTip(tr("apagar y marcharme a casa"));
    connect(accionSalir,SIGNAL(triggered()), this, SLOT(close()));

    accionAbrir = new QAction(tr("Abrir"),this);
    accionAbrir->setShortcut(tr("Ctrl+O"));
    accionAbrir->setStatusTip(tr("Abrir Archivo"));

    accionGuardar = new QAction(tr("Guardar"),this);
//    accionGuardar->setIcon(QIcon(":/images/icon.png"));
    accionGuardar->setShortcut(tr("Ctrl+G"));
    accionGuardar->setStatusTip(tr("Guardar Archivo"));

    connect(accionSalir, SIGNAL(triggered()),this, SLOT(close()));
    connect(accionNuevo,SIGNAL(triggered()),this, SLOT(slotNuevo()));

}
```

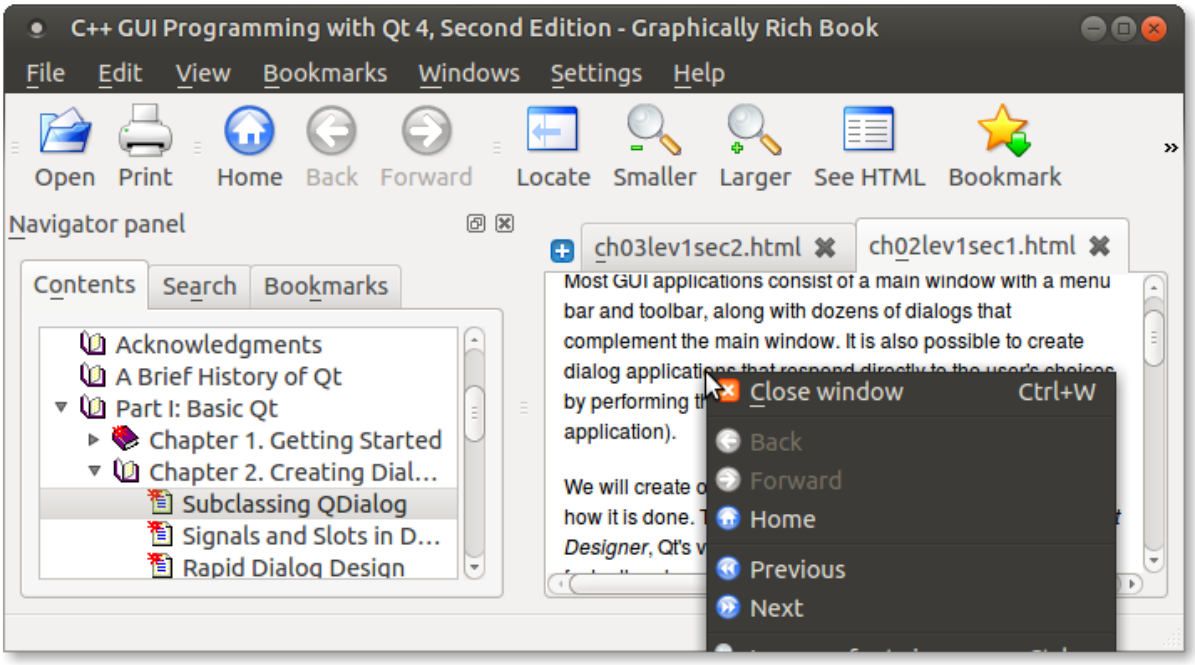
createMenus() hará lo propio con los menús. NO olvides que las QActions hay que crearlas ANTES de los menús y deben ser Punteros a QActions **declarados en la clase**.

```
void VentanaPrincipal::createMenus(){
    menuSalir = menuBar()->addMenu(tr("Archivo"));
    menuSalir->addAction(accionAbrir);
    menuSalir->addAction(accionGuardar);
    menuSalir->addSeparator();
    menuSalir->addAction(accionSalir);
}
```

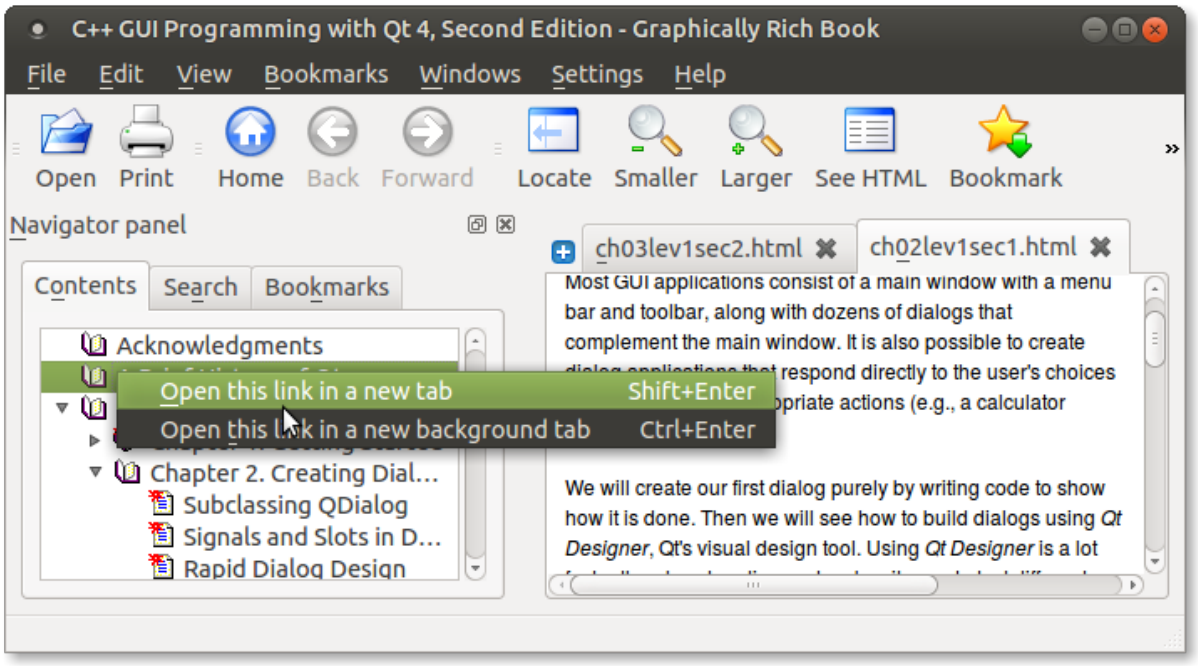


Menú contextual

El menú contextual es un menú. Mira estos ejemplos



que cambia con el contexto



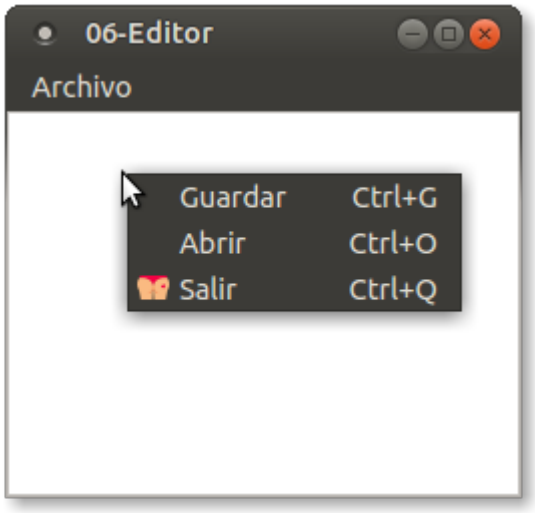
Lo importante es que a todos los widgets (o casi) se les puede añadir un menú contextual particular, pero realmente es un menú más que está construido a partir de QAction.

Para seguir teniendo el editor bien organizado, vamos a crear un nuevo método auxiliar de nuestra clase VentanaPrincipal, donde gestionaremos los menús contextuales. Llamémosle createContextMenu(); Observa en el código cómo añadimos un menú contextual al editor central

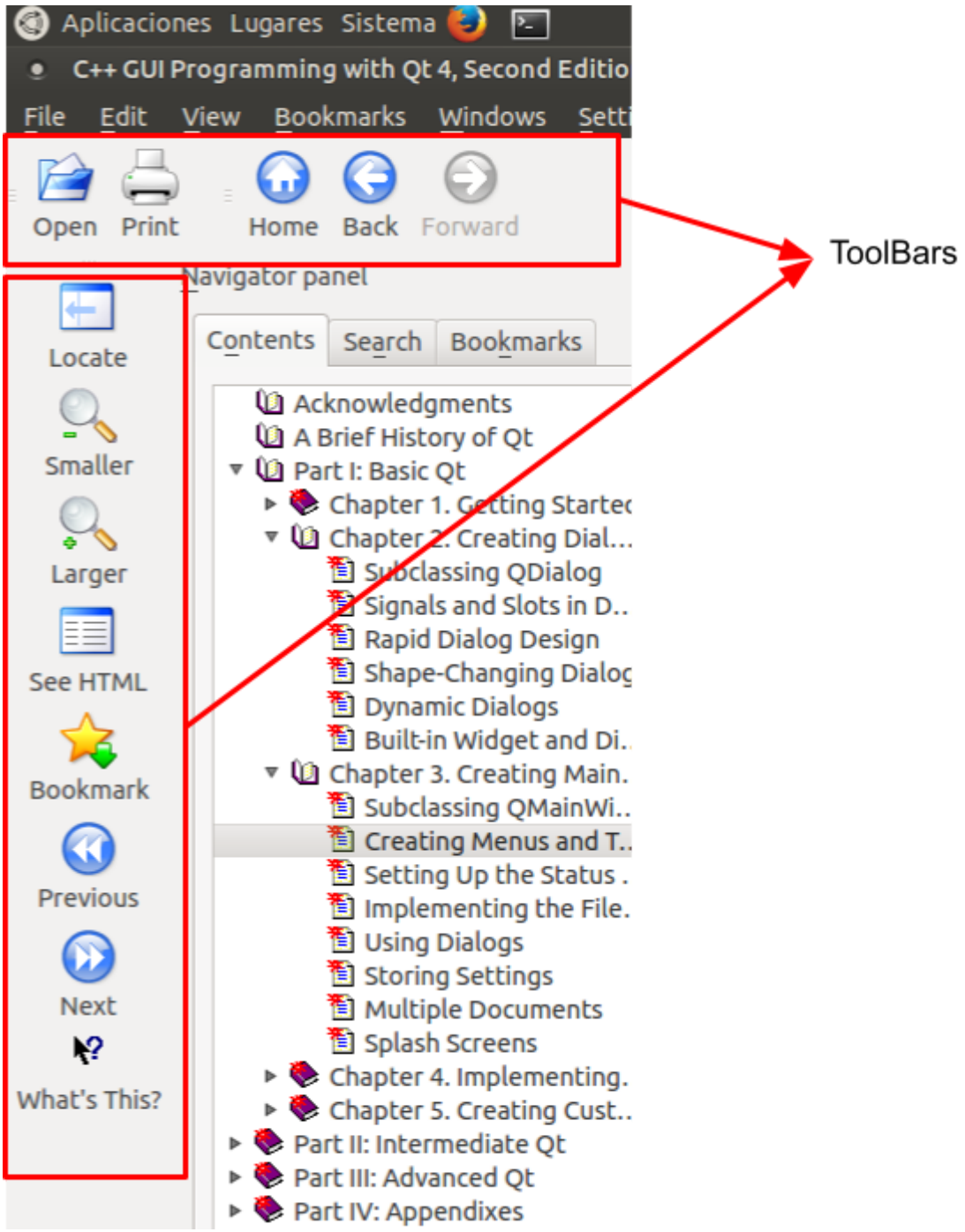
```
void VentanaPrincipal::createContextMenu(){

    editorCentral->addAction(accionGuardar);
    editorCentral->addAction(accionAbrir);
    editorCentral->addAction(accionSalir);
    editorCentral->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

Este componente tiene su propio menú contextual, hemos de avisar de que queremos nuestro propio menú contextual.

No olvides llamar a ese método desde el constructor.



Barras de herramientas



LA barra de herramientas ha de crearse. la ventana viene preparada con el siguiente método:

QToolBar * QMainWindow::addToolBar (const QString & *title*)

para verlo bien, vamos a crear dos barras

por tanto:

```
barraArchivo = addToolBar("archivo");
barraSalir = addToolBar("salir");
```

barraArchivo y barraSalir son QToolBar *

en .h:

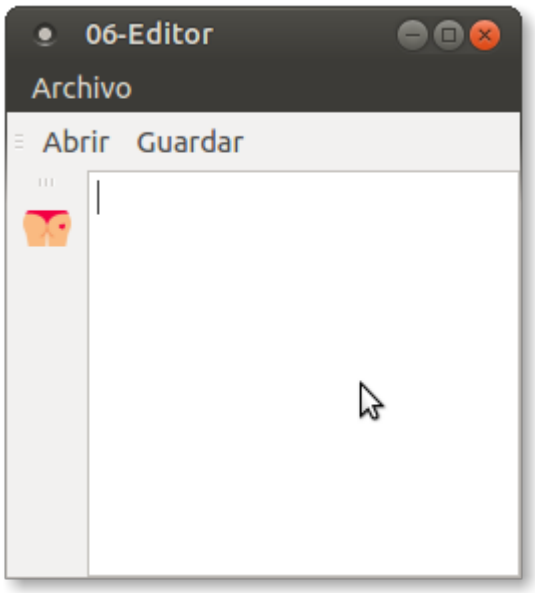
```
QToolbar *barraSalir, * barraArchivo;
```


no olvidar el #include

Ahora hay que añadir… ¡Actions!

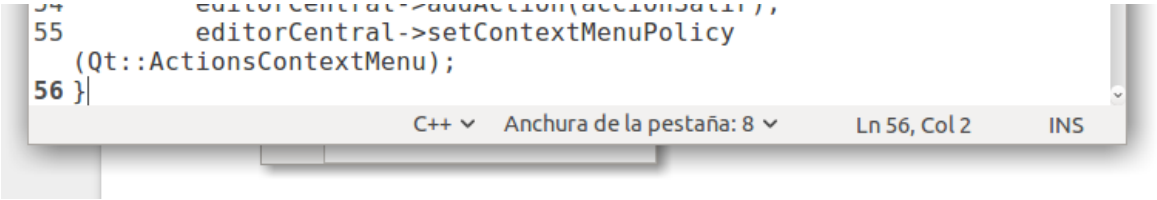
```
barraArchivo->addAction(accionAbrir);
barraArchivo->addAction(accionGuardar);

barraSalir->addAction(accionSalir);
```

y llamar al método createToolBars() desde el constructor.

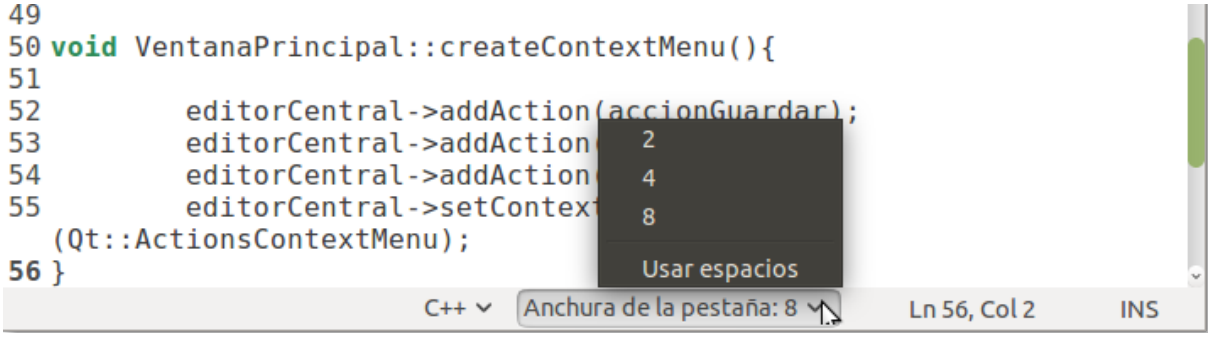


Barra de estado

La barra de estado aparece debajo de la zona central y muestra información cambiante sobre el estado de la aplicación o widget central. En un editor, suele mostrar la línea y posición del cursor, el modo de escritura (insertar, sobreescribir), etc.



A veces, la barra de estado incluye elementos de entrada , como lineEdits, botones o desplegables:



Por lo que la barra es en sí un widget

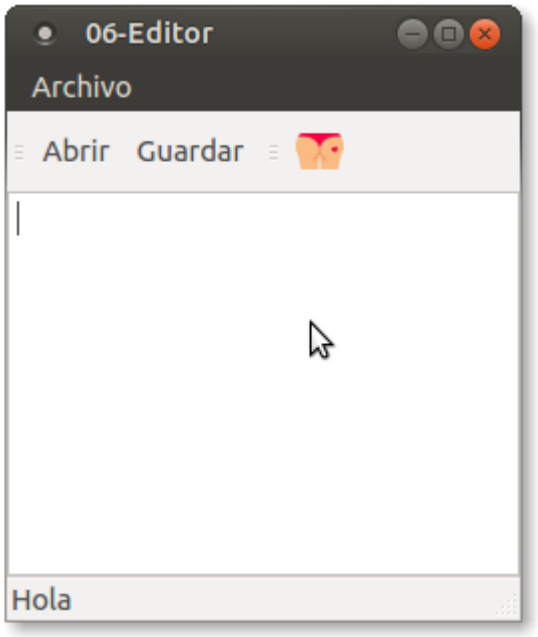
La barra de estado ya existe cuando creas QMainWindow (igual que la barra de menú) pero no se ve si no se meten dentro widgets.

La idea básica es crear componentes, meterlos en la barra de estado y manipularlos después para reflejar cambios

ventanaprincipal.cpp ventanaprincipal.h	**nuevo método createStatusBar para crear barra de estado**

```
      /*en ventanaprincipal.h */ QLabel *etiquetaEstado;
      ...
void VentanPrincipal::createStatusBar(){
          etiquetaEstado = new QLabel("Hola");
          statusBar()->addWidget(etiquetaEstado);
}

...

      etiquetaEstado->setText("Documento Guardado");
```

Con el ejemplo anterior, (si no se ejecuta la última línea), el efecto será el siguiente;



Actualizar la información de la barra de estado.

La barra de estado siempre muestra una información que puede cambiar (¡Esa es su función!). Pero los cambios que hayan de mostrarse allí, los hemos de reflejar nosotros. Ella sola no actualiza nada. Vamos a verlo con el siguiente ejercicio:

La barra de estado mostrará el número de líneas que tiene el texto

Nos enfrentamos a un problema de señales, slots, componentes y clases desconocidas.

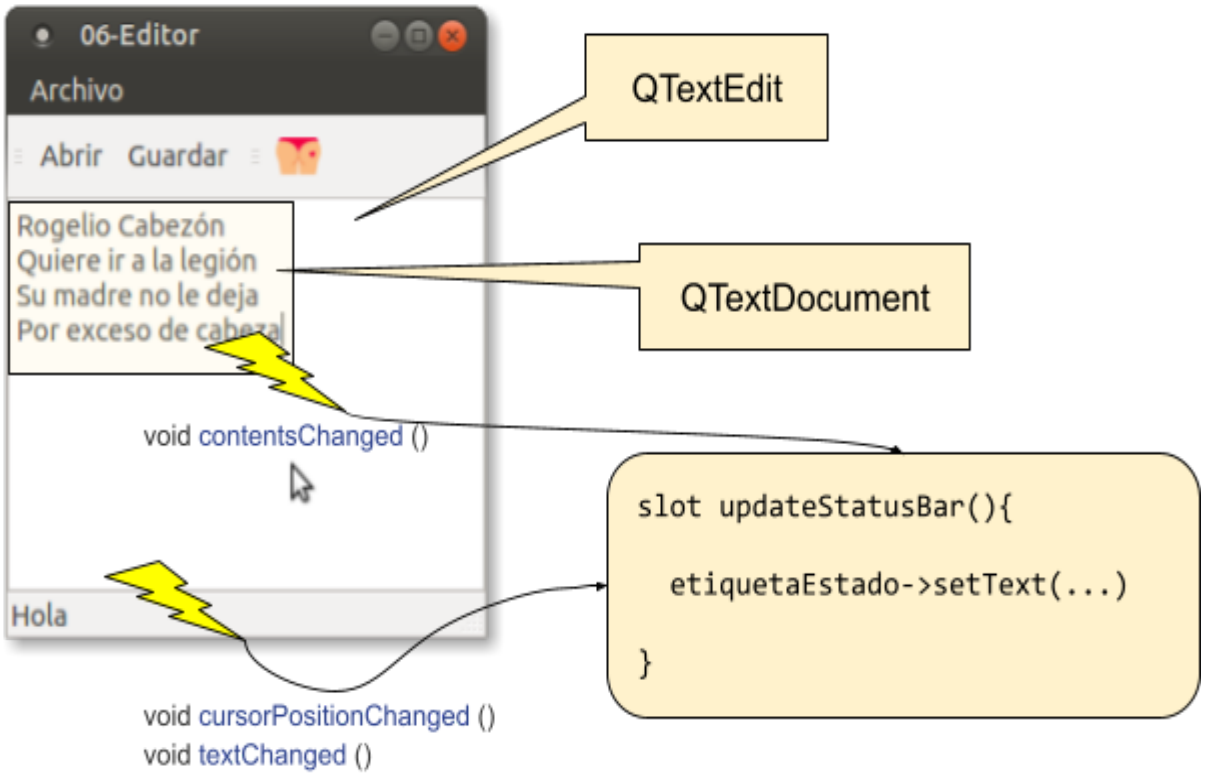
La idea es:

> Cada vez que el usuario escriba o modifique algo en el documento/texto, se debe ejecutar una función (slot) que calcule y actualice la información en la barra de estado.

"El usuario escriba o modifique el texto"... es claramente un evento que ocurre en el editor Central. Debermos saber si existe una señal asociada al editor central que se dispare cada vez que el usuario escribe algo (o elimina)... La hay, pero ¡ No la vamos a usar !

Hay que saber que el componente QTextEdit sólo representa a la parte visible de un editor de texto. El texto mismo, que es la información que contiene va gestionado por un objeto interno de tipo QTextDocument.

,,,,



Los cambios en el editorCentral, son también cambios en el documento interno. Podemos usar sus señales para reacccionar a cambios en el texto

Usaremos una señal del objeto QTextDocument interno

```
QTextDocument * doc = editorCentral->document();
connect(    doc,
            SIGNAL(contentsChanged()),
            this,
             SLOT(slotActualizarBarraEstado()));
```


Ahora el slot que es llamado cada vez que cambia algo:

```
void VentanaPrincipal::slotActualizarBarraEstado(){
    QTextDocument * doc = editorCentral->document();
    QString textoAMostrar("Total lineas: ");
    textoAMostrar = textoAMostrar + QString::number(doc->lineCount());
    etiquetaEstado->setText(textoAMostrar);
```

```
}
```


La alternativa a usar el connect anterior, con su señal es :

```
connect(editorCentral,SIGNAL(textChanged()),
      this, SLOT(slotActualizarBarraEstado()));
```

Opción "Crear Nuevo Documento"< esta sección debe rehacerse , pasito a pasito. sin anticipar … como ahora ocurre>

La opción de nuevo eliminará el documento actual y creará un nuevo documento. Realmente lo único que debemos hacer es eliminar el contenido del documento actual. Esto se consigue simplemente con la ejecución de la siguiente línea:

```
editorCentral->document()->clear();
```

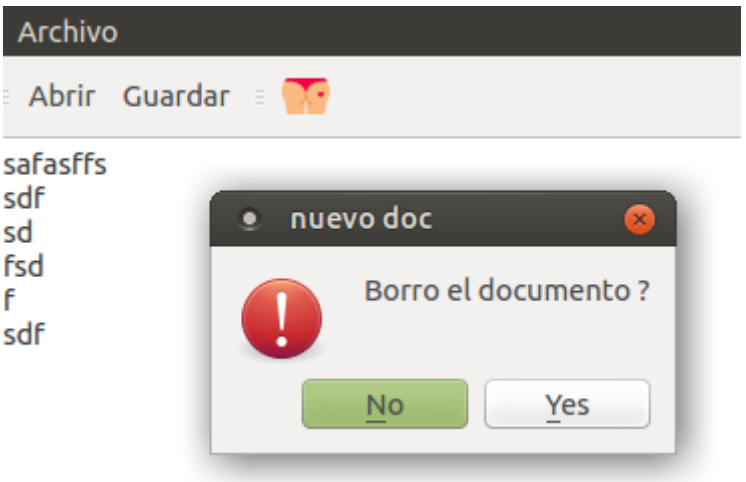
Se puede conseguir esto teniendo la nueva QAction y estableciendo la siguiente conexión en el constructor de la ventana principal:

```
connect(accionNuevo,SIGNAL(triggered()),editorCentral,SLOT(clear()));
```

Preguntar al usuario si está seguro

Cuando se elija la opción del menú "Nuevo", Antes de proceder al borrado, hay que ver si podemos borrar el contenido anterior. Esto impide que usemos la opción anterior conectando directamente la QAction con el slot clear() del editorCentral

Ahora necesitamos un slot propio, que primero muestre un diálogo muy concreto que pregunta al usuario si seguir adelante. Y si el usuario elige "Yes" entonces elimine el contenido.



Este tipo de preguntas es muy **frecuente** durante la interacción con un usuario. Son diálogos muy simples que siempre se componen de los siguientes elementos:

Existen 4 tipos de diálogo que corresponden con 4 intenciones

- Informar al usuario de algo crítico que plantea la finalización del programa
- Advertir al usuario de algo problemático o arriesgado pero no crítico
- Informar al usuario de algo no problemático
- Preguntar al usuario

Para hacer estos diálogos, se ha creado una clase que incorpora todo lo necesario para rápidamente y sin complicaciones mostrar los diálogos **y recoger la respuesta que el usuario elija** con los botones.

La clase es **QMessageBox**

Por ejemplo el diálogo anterior se crea simplemente haciendo una llamada a un método de clase (static) de la classeQMessageBox:

```
QMessageBox::warning(this, tr("nuevo doc"),
            tr("Borro el documento?"),
```

```
                    QMessageBox::Yes | QMessageBox::No);
```

Fíjate que los 4 elementos de estos diálogos están ahí, en la llamada anterior:

Selección de la pregunta

Si vemos la ayuda de QMessageBox e indagamos sobre

```
enum QMessageBox::StandardButton
```

Veremos entre otros los siguientes valores

`QMessageBox::Cancel`	`0x00400000`	A "Cancel" button defined with the RejectRole.
`QMessageBox::Yes`	`0x00004000`	A "Yes" button defined with the YesRole.
`QMessageBox::No`	`0x00010000`	A "No" button defined with the NoRole.

Recuerda: La siguiente llamada

```
QMessageBox::warning(...,
      QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel);
```

Permite configurar en ese último argumento qué botones se van a mostrar. Consulta la ayuda para ver todos los botones disponibles. Estos botones tienen un código que puede combinarse con la operación OR . De forma que el resultado es un número…

0x00414000

Que es descomponible por QT en los tres valores usados para componer ese argumento

El código anterior mostrará el MessageBox, pero en nuestro programa debemos recibir/recuperar/averiguar la elección hecha por el usuario. **La misma función invocada devuelve** un número que corresponde con el código del botón pulsado. Este valor se puede usar para actuar a continuación

```
int r = QMessageBox::warning(this, tr("Spreadsheet"),
            tr("The document has been modified.\n"
                  "Do you want to save your changes?"),
                  QMessageBox::Yes | QMessageBox::No
                  | QMessageBox::Cancel);
```

Con el QMessageBox ocurre que… se abre la ventana, el usuario pincha un botón y se cierra. ¿Cuál ha pinchado? Es lo que la llamada devuelve, en el ejemplo se copia el valor en la variable r, que puede ser estudiada y se toma decisiones en función de su valor. A partir el ejemplo anterior, observa cómo se trata la respuesta en el siguiente ejemplo:

```
if (r == QMessageBox::Yes) {
            return save();
} else if (r == QMessageBox::Cancel) {
            return false;
}
```

Por tanto, para implementar lo que tenemos entre manos, hemos de :

1. Crear una nueva acción (accionNuevo)
2. Conectarla a un nuevo slot (slotNuevo())
3. Colocarla en el menú y toolbar

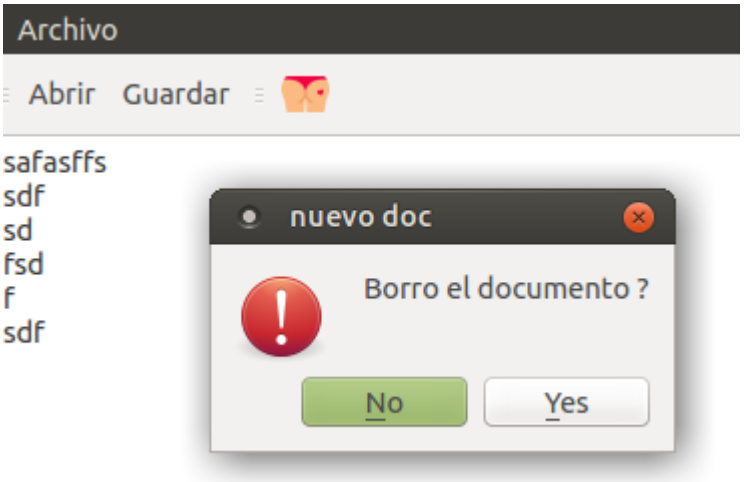
Ya en el slot

4. Mostrar un messageBox con dos botones "seguir" o "sí", y "cancelar" o "no"
5. recoger el valor devuelto
6. Actuar en consecuencia.

Vemos aquí el slot que te ayudará a comprender el uso y funcionamiento de QMessageBox:

ventanprincipa.cpp	

```
void VentanaPrincipal::slotNuevo(){

    int respuesta = QMessageBox::warning(this,
                    tr("nuevo doc"),
                    tr("Borro el documento ?"),
                    QMessageBox::Yes | QMessageBox::No);

    if (respuesta == QMessageBox::Yes)
       editorCentral->document()->clear();
}
```

Proteger ante cambios

La pregunta anterior no tiene sentido hacerla normalmente si el documento ha sido recién guardado o si no se ha escrito nada. Un editor pregunta si ve que hay cambios por guardar. Vamos a empezar a implementar la lógica de esto. El programa no preguntará si no hace falta tener tanta precaución.

Hace falta:

- Tener un atributo bool documentoModificado de tipo bool que almacenará true si el documento no está guardado en el disco tal cual está enl a pantalla
- Inicializar documentoModificado a false en el constructor

- Cambiarlo a true cuando hay una modificación en el texto escrito (el usuario cambia algo en el texto)
- cambiarlo a false cuando guardamos el documento.

```
class VentanaPrincipal : public QMainWindow {
Q_OBJECT
....
    bool documentoModificado;
```

```
VentanaPrincipal::VentanaPrincipal( ...) {
      documentoModificado = false;
```

¿Qué señal usar ? … textChanged… que ya la tenemos conectada al slot slotAcualizarBarra… REUSAMOS ESTE SLOT

```
void VentanaPrincipal::slotActualizarBarra(){
    QTextDocument * documento =
editorCentral->document();

    documentoModificado = true;
    int numLineas;
```

```
void VentanaPrincipal::slotNuevo(){

    if (!documentoModificado) {
       editorCentral->document()->clear();
       documentoModificado = false;
       return;
    }
```

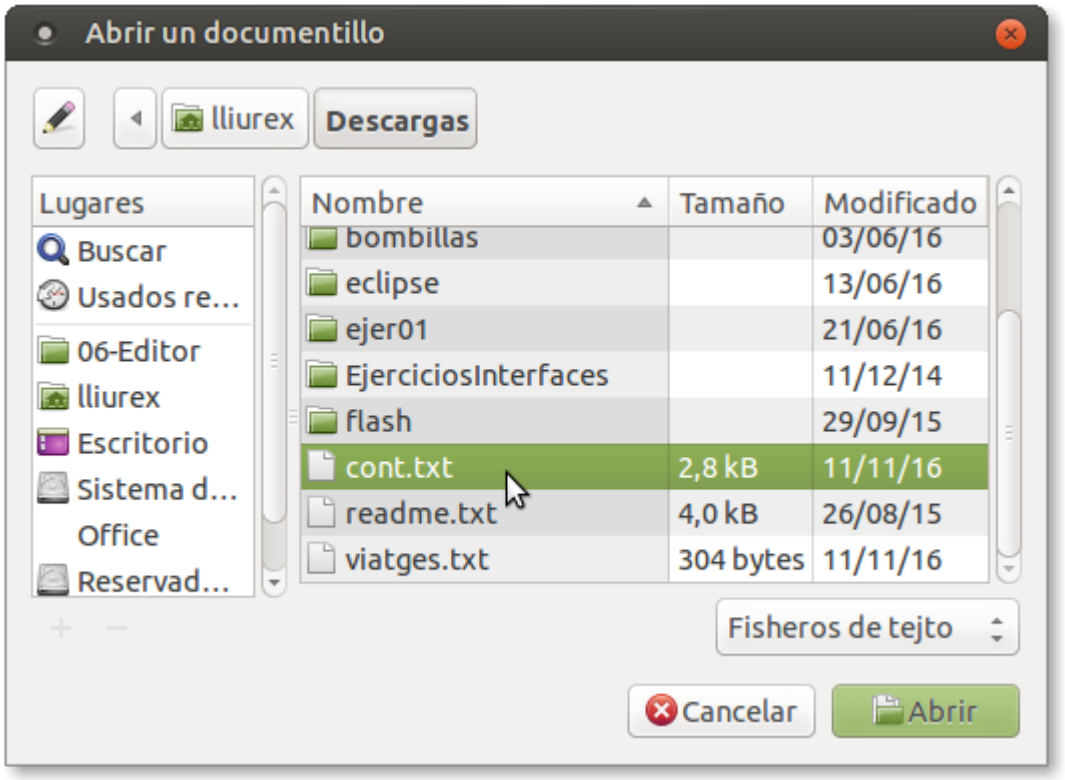
Abrir y Guardar

Para abrir documento, primero debemos crear todo lo habitual ya :

- QActions
- Añadirlas al menús
- Connect de QAction con slots

- Nuevo Slot

En el slot, lo primero que haremos será mostrar el siguiente diálogo para solicitar al usuario el nombre del archivo. Esto se consigue muy fácilmente con la clase `QFileDialog`.



Observa el código del slot para mostrar el diálogo anterior.

ventanaprincipal.cpp	slotAbrir

```
void VentanaPrincipal::slotAbrir()
{
   QString fileName = QFileDialog::getOpenFileName(this,
                                    tr("Abrir un documentillo"), ".",
                                    tr("Fisheros de tejto (*.txt)"));
   if (!fileName.isEmpty())
            abrirFichero(fileName);

}
```

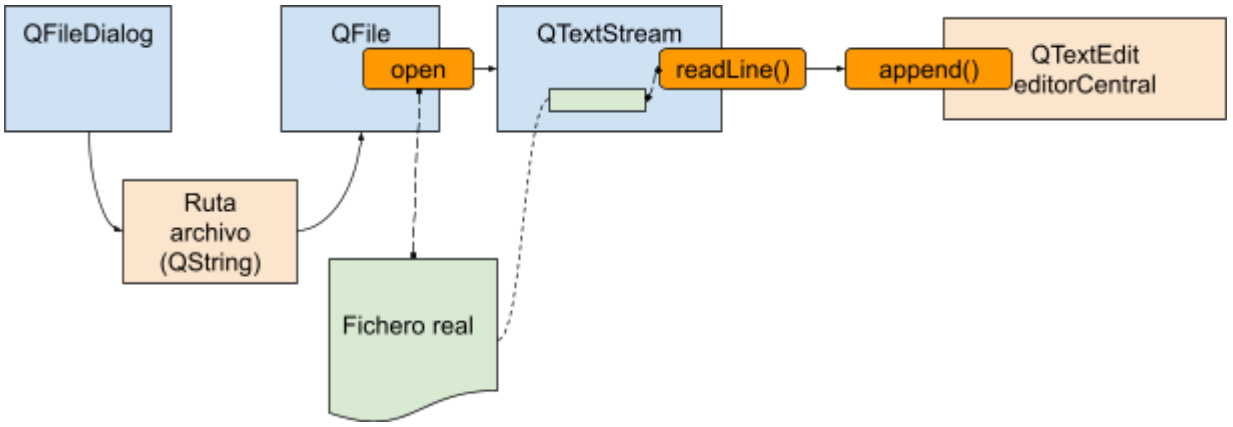
¡Observa que el **diálogo es modal** ! Y al finalizar su ejecución **devuelve la ruta de un archivo**.

La selección de extensiones de ficheros admisibles se puede, por ejemplo, hacer así:

```
tr("Spreadsheet files (*.sp)\n"
```

```
    "Comma-separated values files (*.csv)\n"
    "Lotus 1-2-3 files (*.wk1 *.wks)")
```

Este slot sólo lanza el diálogo y recoge la selección del usuario en la cadena fileName. La carga real de los datos se realiza en el método "abrirFichero", que recoge la ruta del archivo en string. La carga de un fichero es una acción compleja que requiere varios pasos (de hecho hay un módulo enteramente dedicado a cosas como ésta). En cada paso se crea un objeto nuevo a partir del anterior. La idea es la siguiente



1. Con el **QFileDialog** logramos obtener la ruta de un archivo. (la metemos en un QString).
2. Con la **ruta**, creamos un objeto de Tipo **QFile**, que representa al fichero.
3. Con el objeto **QFile** creamos otro objeto de tipo **QTextStream** que representa a un flujo de datos de tipo texto (leer y escribir en este objeto representa leer y escribir en el fichero)
4. Leemos del objeto QTextStream y escribimos las QString leídas en el editorCentral (tiene un método para añadir texto "append(QString)"

Observa el siguiente método donde se aprecian los pasos anteriores y cómo se van creando los objetos unos a partir de otros hasta finalmente leer datos.

ventanaprincipal.cpp	abrirFichero(QString)

```
bool VentanaPrincipal::abrirFichero(QString nombreFichero) {
      editorCentral->document()->clear();
      QFile fichero(nombreFichero);
      if (!fichero.open(QIODevice::ReadOnly)) {
                  QMessageBox::warning(this, tr("Editor"),
                              tr("Cannot read file %1:\n%2.")
                              .arg(fichero.fileName())
                              .arg(fichero.errorString()));
      return false;
      }
```

```
        QTextStream stream(&fichero);

        while (!stream.atEnd()){
                    editorCentral->append(stream.readLine());
        }
        return true;
}
```

Guardar.

Para guardar el documento, además de toda la parafernalia necesaria para las nuevas entradas de menú, añadiremos dos nuevos métodos .<explicar el "Como" en el guardarComo")

```
        void slotGuardarComo();

        bool guardarFichero(QString nombreFichero) ;
```

El SlotGuardarComo es muy similar, tan sólo preguntamos el nombre del archivo, pero ¡ Ojo ! hay que usar get**Save**FileName

ventanaprincipal.cpp

```
void VentanaPrincipal::slotGuardarComo()

   QString fileName = QFileDialog::getSaveFileName(this,
                                    tr("Ajuardar el fishero"), ".",
                                    tr("Fisheros de tejto (*.txt)"));
   if (!fileName.isEmpty())
            guardarFichero(fileName);
}
```

Es preciso hacer `QFileDialog::getSaveFileName` para que deje escribir un fichero que no existe. Creamos el objeto fichero a continuación

```
        QFile fichero(nombreFichero);
```

Y lo abrimos. El fichero es pa escribir no pá leer, por lo que en la llamada pasamos una constante que indica que el fichero es para escribir.

```
        fichero.open(QIODevice::WriteOnly)
```

Seguimos…. ¿Cómo leer o capturar el texto del qTextEdit y pasarlo a QString?

El editor Central es sólo el componente visual, el documento y su texto están "dentro" en un elemento de tipo QTextDocument. Obtenemos el mismo de la siguiente manera:

```
QTextDocument *documento;
documento = editorCentral->document();
```

El texto que hay en su interior, para nosotros, está organizado en párrafos o líneas. (Piensa que un salto de línea equivale al final de un párrafo). Qt llama a cada línea "bloque" de texto, y un QTextDocument tiene varios.

Tan sólo tenemos que ir, uno por uno, obteniendo los bloques de texto . El siguiente bloque realiza eso mismo. Fíjate en los dos métodos que usamos para orquestar el recorrido

```
QTextDocument *documento = editorCentral->document();
for(int i=0; i< documento->blockCount();i++){
   QString cadenaL;
   cadena = documento->findBlockByNumber(i).text();
}
```

Para finalmente realizar la escritura, Creamos un flujo de texto a partir del fichero, al igual que antes (pero es un flujo de escritura)

ventanaprincipal.cpp	escribirFichero

```
QTextStream flujo (&fichero);

for(int i=0; i< editorCentral->document()->blockCount();i++){
   QString cadena;
   cadena =
editorCentral->document()->findBlockByNumber(i).text();
   flujo << cadena << '\n' << Qt::flush ;
}
```

Guardar **al salir**, al abrir, o al "nuevo Documento"

Esta explicación es más importante de lo que parece.

En un editor normal, cuando el usuario decide crear un nuevo documento, abrir uno existente desde un archivo o cerrar, el programa sabe si el documento actual ha sido modificado y advierte al usuario que la acción que solicita va a causar que se pierdan los datos modificados del documento actual.

Por ejemplo, el slot de nuevo documento:

```
void VentanaPrincipal::slotNuevo(){
       editorCentral->document()->clear();
}
```

Si el slot es llamado, se elimina el contenido del documento actual, incluso si no ha sido guardado, se pierden las modificaciones. Ocurre lo mismo con abrir o al salir del programa.

Para solucionar esto hay que aplicar ciertos elementos:

- Saber en cada momento si el documento está modificado respecto la última vez que se guardó
- Mantener, por tanto, un atributo de la clase de tipo bool que indicará si el documento está modificado.
- Cada vez que se guarda el documento con éxito se debe "marcar" el documento como guardado o no modificado.
- Cada vez que el usuario escribe algo, se debe marcar el documento como "modificado"

Veámos pues la historia de este atributo


```
class VentanaPrincipal : public QMainWindow {
private:
      ...
      bool documentoModificado;
      ...
};
```

En el constructor la inicialización es clara:

```
documentoModificado = false;
```

Ahora, hay que asegurarse que cada vez que cambia el texto, se actualiza este valor… Esto supone trabajar con la señal textChanged() de editorCentral, luego hay que conectarlo a un slot (por ejemplo en el constructor)

```
connect(editorCentral,SIGNAL(textChanged()),this,slotModificado());
```

y hay que hacer el slot

ventanaprincipalcpp	

```
      void VentanaPrincipal::slotModificado(){
          documentoModificado = true;
```

```
    }
```

Y también, al guardar, en el método guardarFichero(String), hay que actualizar el estado del documento.

ventanaprincipal.cpp

```
    bool VentanaPrincipal::guardarFichero(QString nombreFichero)
    {
     ...
        documentoModificado=false;
         return true;
    }
```

Podríamos empezar a cambiar ya algunos slots. Por ejemplo, el slotNuevo podría ser ahora:

ventanaprincipal.cpp **slotNuevo**

```
void VentanaPrincipal::slotNuevo(){
      if (! documentoModificado)
        editorCentral->document()->clear();
      else
      QMessageBox::warning(this, tr("Editor"),
              tr(" No puedo crear nuevo fichero porque está modificado
el anterior. Guarda antes");
}
```

Pero esto es muy incómodo y no es lo habitual. Lo habitual es preguntar : "¿seguir sin guardar?", "¿guardar antes de continuar?" o "abortar y no seguir". Esta pregunta, además, se hace igual en tres situaciones si hay modificaciones pendientes de guardar:

- Al querer salir del programa (por ejemplo, pulsando el aspa)
- Al querer abrir un documento
- Al querer crear un nuevo documento

Por ello, vamos a crear un nuevo método llamado `quieresContinuar()`;

ventanaprincipal.cpp	quieresContinuar()

```
bool VentanaPrincipal::quieresContinuar(){

    if (documentoModificado) {
       int respuesta = QMessageBox ( " ¿Quieres guardar majete...
       if (respuesta == YES) { guardarFichero(..) ; return true; }
       else return false;
    }
    return true; // no había nada modificado
}
```

Ahora el slot de Nuevo puede reescrbirse bien fácilmente (y también el de "Abrir", pero dejamos el asunto de "cerrar la aplicación" para más tarde)


```
void VentanaPrincipal::slotNuevo(){
            if (quieresContinuar())
             editorCentral->document()->clear();
}
```

Volviendo al método `quieresContinuar()` observemos que el usuario puede elegir la opción de guardar antes de continuar. No es habitual preguntar al usuario el nombre del fichero si el documento ya fue guardado con ese nombre antes. en estos casos se procede directamente a guardar con el nombre del fichero que ya tenía antes.

Además, en el menú habitualmente hay dos opciones para guardar:

- guardar
- guardarComo

La primera opción no está implementada y vamos a implementarla de paso

Nuestro programa no sabe hacer otra cosa que preguntar el nombre cada vez con el slot slotGuardarComo de a continuación:

Recordatorio:

```
void VentanaPrincipal::slotGuardarComo(){
   QString fileName = QFileDialog::getSaveFileName(this,
                        tr("Ajuardar el fishero"), ".",
```

```
                        tr("Fisheros de tejto (*.txt)"));
   if (!fileName.isEmpty())   guardarFichero(fileName);
}
```

¿Has visto que en un editor normal, cuando ya has guardado el documento, puedes pinchar a "Guardar" y no te vuelve a preguntar por el nombre del archivo? Nuestro editor debe recordar el nombre del archivo del documento en edición. Y el editor lo estamos programando nosotros así que tendremos que prever esta circunstancia.

Así mismo, vamos a añadir un nuevo slot para guardar un fichero sin preguntar por el nombre. Aquí están lasnuevas adiciones a la clase VentanaPrincipal .

```
class VentanaPrincipal : public QMainWindow {
      ...
private:
      bool documentoModificado;
      QString rutaFicheroActual;
private slots:
      bool slotGuardar();
};
```

Este atributo rutaFicheroActual, se debe inicializar vacío:

- Al arrancar el programa
- Al crear un Nuevo Documento

En ambos casos hay que hacer lo siguiente:

```
      rutaFicheroActual.clear();
```

Ahora en el método quieresContinuar, según la respuesta del usuario vamos a continuar por distintos derroteros ya que puede estar el documento modificado o no, y el nombre de archivo para nuestro documento establecido o no… eso lo gestionará slotGuardar()

```
bool VentanaPrincipal::quieresContinuar(){

 if (documentoModificado) {
```

```
      int r = QMessageBox::warning(this, tr("Editor"),
                        tr("The document has been modified.\n"
                        "Do you want to save your changes?"),
                        QMessageBox::Yes | QMessageBox::No
                        | QMessageBox::Cancel);

      if (r == QMessageBox::Yes)  // veamos si el usuario quiere guardar
            return slotGuardar();
      else if (r == QMessageBox::Cancel)  // el usuario no quiere seguir
            return false;
      /* else */
      return true; // el usuario ha pulsado "NO guardar" quiere salir YA

      } // no había modificaciones
      return true;
}
```

Si el slot slotGuardar() es llamado lo primero que habría que hacerse es mirar si el documento actual tiene ya un fichero asociado y recuperar su ruta. Para ello habíamos creado un atributo QString rutaFicheroActual; que ahora hay que consultar

ventanaprincipal.cpp

```
void VentanaPrincipal::slotGuardar(){
      if( ! rutaFicheroActual.isEmpty() )
             guardarFichero(rutaFicheroActual);
}
```

Pero. ¿Y si no hay un nombre de fichero asociado? Pues no queda más remedio que preguntarlo… igual que hacíamos con el slot guardarComo()

ventanaprincipal.cpp

```
void VentanaPrincipal::slotGuardar()
{
   if( ! rutaFicheroActual.isEmpty() )
         guardarFichero(rutaFicheroActual);
   else
         slotGuardarComo() ;
}
```

Por supuesto. Cada vez que se Guarda un fichero o se abre un fichero hay que anotarse el nombre de fichero usado. Por ejemplo al guardar el archivo

ventanaprincipal.cpp

```
bool VentanaPrincipal::guardarFichero(QString nombreFichero) {
        ...
        documentoModificado=false;
        rutaFicheroActual = nombreFichero;
        return true;
}
```

¡¡ Pero también al abrir un archivo !! Esto no está documentado, pero hay que hacerlo.

Cerrar la aplicación con modificaciones.

Aquí hay una explicación muy importante del curso, no es una sección más.

Objetivo:

> Hay muchos motivos por los que puede cerrarse la ventana de la aplicación finalizándola. Sea como sea hemos de verificar que la ventana no se cierra con un documento modificado y no se le pregunta al usuario qué hacer, como hemos hecho antes.
>
> Realmente, deseamos "interceptar" el cierre de la ventana, y preguntar antes de que realmente ocurra ese cierre. Y según lo que conteste el usuario, ¡Evitar que se cierre la ventana!.

¿cómo lo hacemos ?

Este es un **problema universal, muy habitual** y que se soluciona fácilmente, pero de una forma no vista hasta ahora. Vamos a hacer un rodeo y dar una explicación larga que sirve para muchos diferentes problemas similares.

Lee con calma la explicación, hacerlo será rentable.

Observa la documentación de la clase `QWidget` (clase madre de `QMainWindow`). busca un método llamado `closeEvent()`, busca el detalle de ese método y verás cómo está definido:

```
void QWidget::closeEvent ( QCloseEvent * event ) [virtual protected]
```

o también

```
virtual void closeEvent(QCloseEvent * event )
```

¿Qué significa ese **`virtual`**? Para entenderlo bien, hay que dejar claras las cosas que NO cambian por ese "virtual" concretamente:

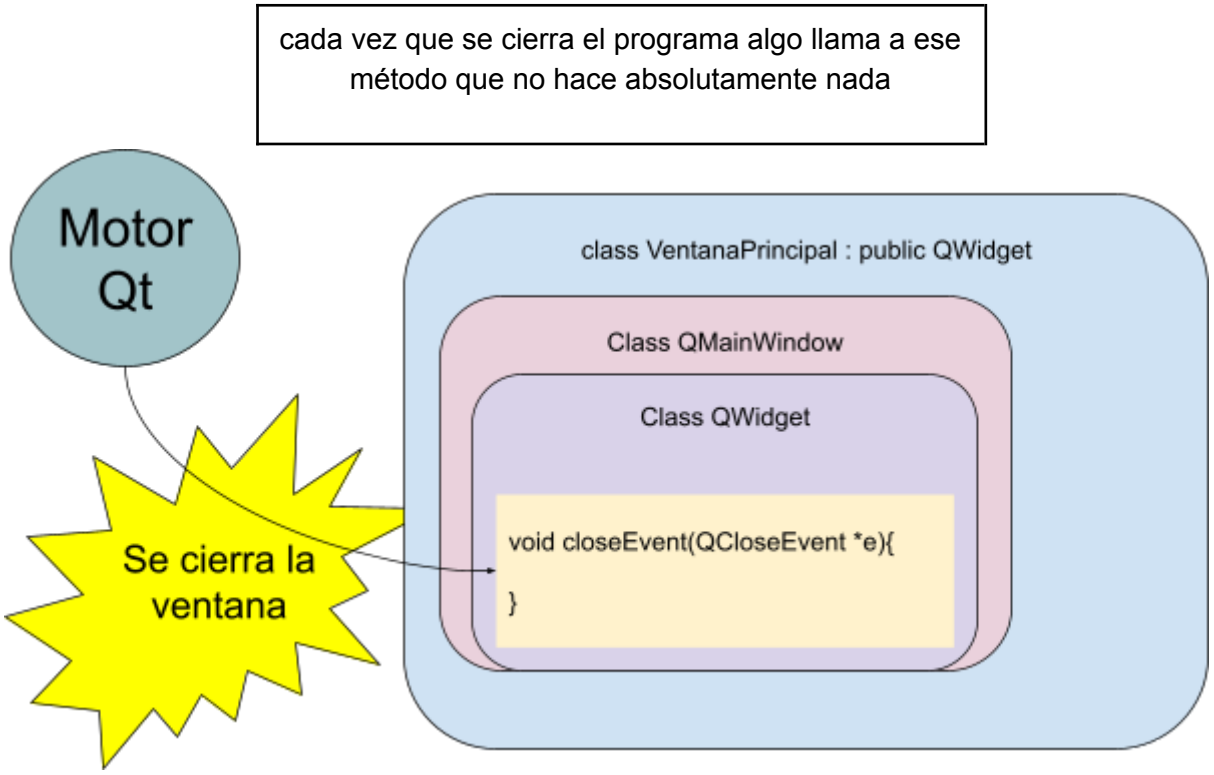
- `closeEvent` sigue siendo un método implementado en la clase QWidget
- `closeEvent` sigue pudiendo ser llamado (pero nosotros no lo llamamos)
- `closeEvent` no hace nada importante tal como ya está implementado en la clase `QWidget`

(ojo, hay una versión de "virtual" que es más radical y no cumple las afirmaciones anteriores)

Además hay un hecho oculto pero ahora importantísimo:

- El motor Qt (el "omnipresente" o "qt") **llama a ese método actualmente** cada vez que se cierra la ventana.
- Ese método está implementado en alguna clase padre, quizá en qwidget
- Ese método NO HACE NADA tal como está implementado.

Conclusión:



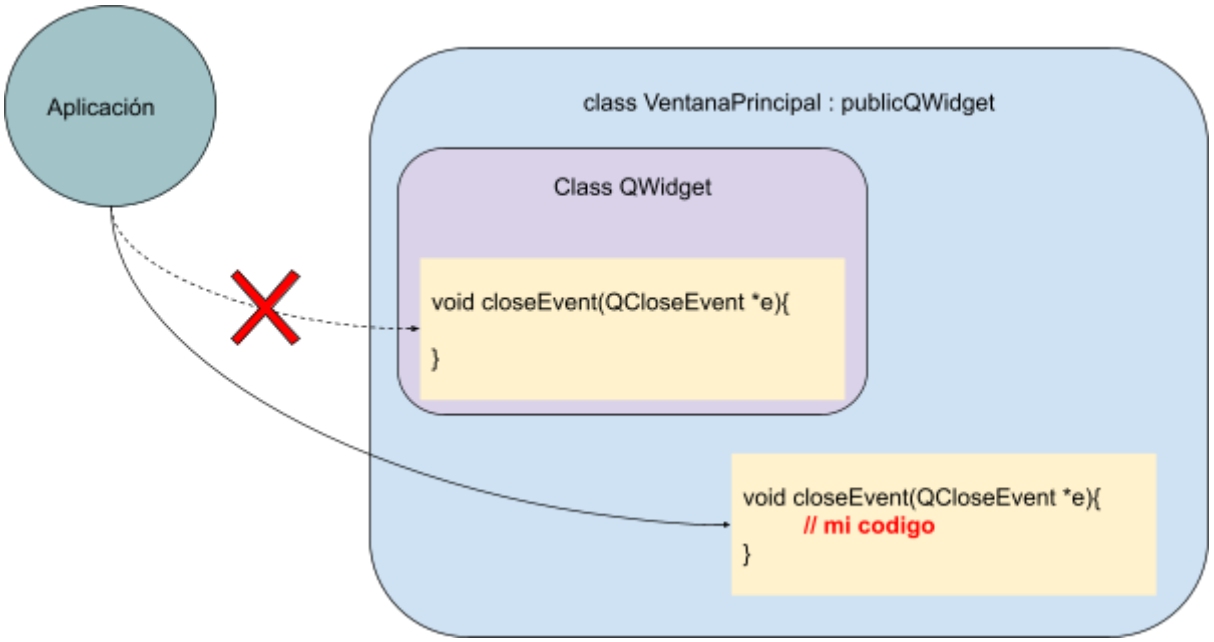
¿No sientes que es uno de esos momentos que algo te parece soberanamente absurdo pero que al final tendrá mucho sentido? Así es.

La gracia de la palabra **virtual**, es que **el programador puede volver a implementar el método** y en dicho caso, en vez de ser llamado el método que ya venía implementado en la clase QWidget, **¡¡ Qt llamará al nuevo método !!** al que el programador ha hecho en la clase

VentanaPrincipal. Es algo así como implementar un método que anula el anterior, siendo esta implementación opcional.

Recuerda: Este método es llamado cada vez que se pretende cerrar la ventana

Si tú implementas ese método, se invoca el tuyo (cuando se cierre la ventana). De alguna forma esto es similar a una conexión señal y slot sólo que *ya está establecida la señal y el connect… y tú … ya … si quieres implementas el slot* .



Método `virtual` (ver assistant);

ventanaprincipal.cpp	

```
void VentanaPrincipal::closeEvent(QCloseEvent * event){

      if (quieresContinuar()) {
            //    writeSettings();
            event->accept();
      } else {
            event->ignore();
      }
}
```

Observa dos cosas:

- Este método es llamado cuando **se pretende** cerrar el diálogo (el usuario pincha en el aspa del marco del diálogo y Qt se da prisa en llamar a éste método)

- Nosotros podemos **decidir aceptar o denegar** que se cierre el diálogo.

De ahí esos event->accept() o event->reject(). Ahí estamos aceptando lo que pretende el evento o no. Si no lo aceptamos, el diálogo no se debería cerrar.

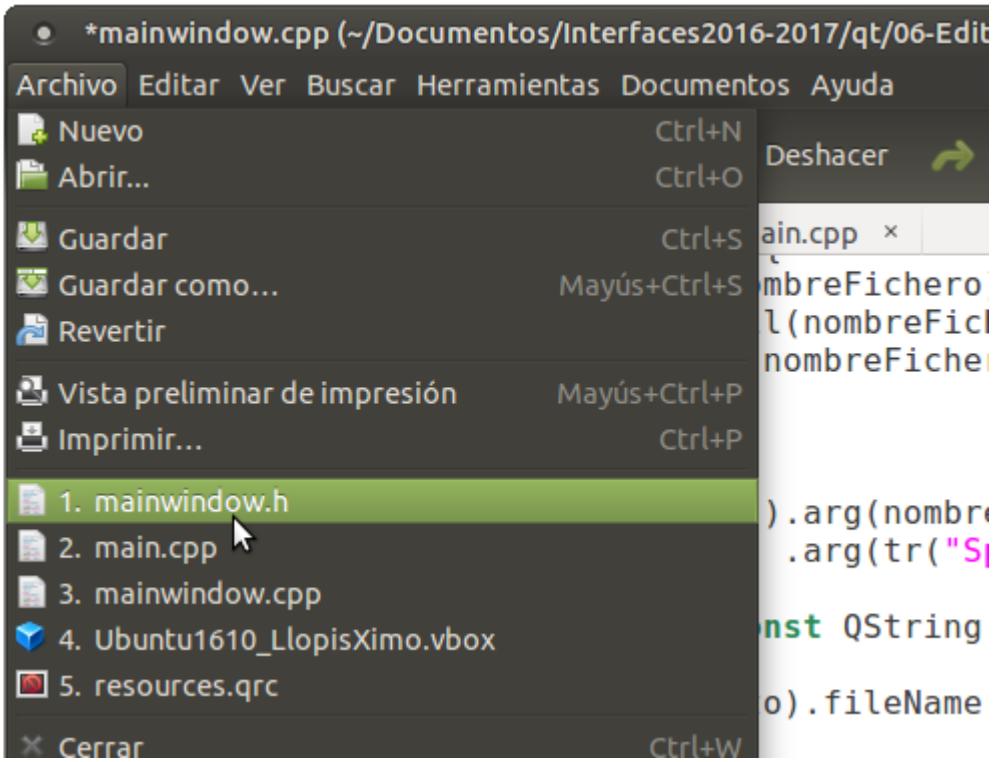
el slot para crear un nuevo documento:

```
void VentanaPrincipal::slotNuevo(){

    if (!quieresContinuar()) return;

    editorCentral->document()->clear();
    rutaFicheroActual.clear();
    documentoModificado = false;
}
```

Mantener un menú con la lista de ficheros recientes

> Esta sección no es de las más importantes (como la anterior de métodos virtuales), pero es bastante compleja y difícil.

Objetivo: Que el programa recuerde los últimos 5 ficheros abiertos, los muestre en el menú "Archivo" y permita abrirlos rápidamente simplemente seleccionando la entrada de menú correspondiente. Observa la captura de ejemplo (tomada de uneditor que implementa esta funcionalidad)



Cada vez que se abra un fichero o se guarde, se está conociendo el nombre de un fichero cuyo documento está abierto o ha sido abierto. Es en ese momento cuando hay que regenerar o actualizar esa lista de entradas de menú.

Lista de nombres

Recordemos que existe un atributo que almacena la ruta del fichero **actual**.

ventanaprincipal.h	

```
QString rutaFicheroActual;
```

Esta ruta ya se actualiza al "guardar como" o al "abrir"

ventanaprincipal.cpp	guardarFichero

```
bool VentanaPrincipal::guardarFichero(QString nombreFichero) {
        ...
        documentoModificado=false;
        rutaFicheroActual = nombreFichero;
}
```

Como ahora vamos a recordar varios archivos anteriores, este tinglao está basado en una lista de nombres de fichero:

```
QStringList ficherosRecientes;
```

`ficherosRecientes`, será una lista de Strings que contendrá el nombre completo de los archivos que hayan sido abiertos.

`ficherosRecientes[0]`	`"/home/user/carta.txt"`
`ficherosRecientes[1]`	`"/home/user/factura.txt"`
`ficherosRecientes[2]`	`"/home/user/poema.txt"`
`...`	`...`

Cada vez que se abra o guarde un fichero, hay que verificar la lista anterior y actualizarla. Eso es lo que hace el siguiente método

ventanaprincipal.cpp

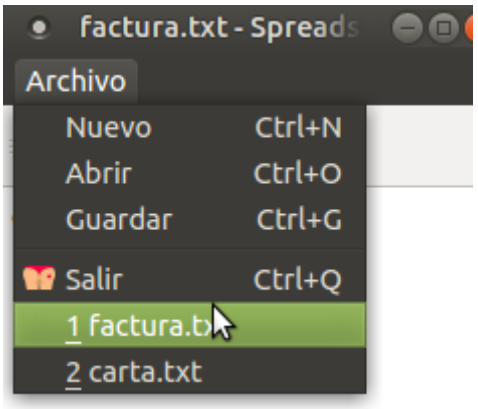
```
void VentanaPrincipal::establecerFicheroActual(const QString
&nombreFichero)
{
      ficherosRecientes.removeAll(nombreFichero);
      ficherosRecientes.prepend(nombreFichero);

      actualizarActionsFicheros();// explicación más adelante
    }
```

`removeAll(nombreFichero)` quita cualquier cadena existente que coincida con "nombreFichero" y vuelve a añadir, al principio de la lista, esta cadena `(prepend(nombreFichero))`. Así, el fichero aparecerá el primero de la lista aunque ya estuviese,

Quedan cosas por hacer, pero la actualización de la lista de nombres de fichero ya está y el resto se ejecutará llamando a otra función "`actualizarActionsFicheros()`"

Actualizar el menú

¿Por qué antes se llama a `actualizarActionsFicheros()` ? Porque no está todo hecho. La lista `ficherosRecientes` es algo interno de la clase, pero hay que actualizar el menú a partir de ella



Vamos a recorrer la lista para verificar que todos los nombres se corresponden con ficheros existentes. Quizá el usuario ha borrado un fichero y lo primero que hay que hacer es quitarlo de la lista porque ya no existe. El método empieza así:

ventaanapirncipa.cpp

```
void VentanaPrincipal::actualizarActionsFicheros()
{
      QMutableStringListIterator i(ficherosRecientes);
            while (i.hasNext()) {
            if (!QFile::exists(i.next()))
                  i.remove();
      }
```

El método `QFile::exists(QString)` devuelve false si un fichero con la ruta pasada no existe

Seguimos... rehacer todas las acciones. ¿Varias acciones? ¿ Cuántas acciones? ¿Cómo se declaran?

```
QAction * accionesFicherosRecientes[MAX_RECENT_FILES];
```

MAX_RECENT_FILES es una constante que indica el número máximo de archivos que se recordarán. La puedes declarar de alguna de las siguietnes dos formas:

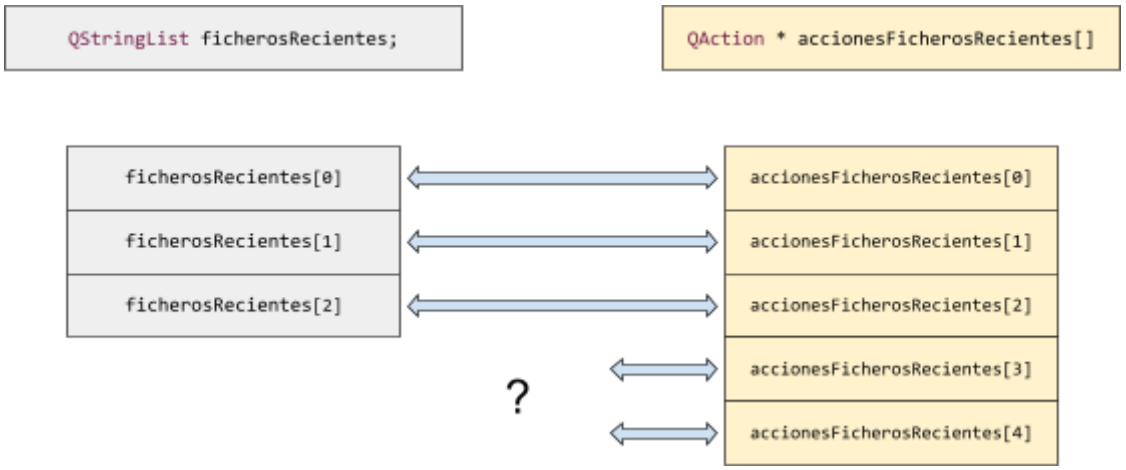
Al principio del fichero ventanaprincipal.h :

```
#define MAX_RECENT_FILES 5
```

Dentro de la clase VentanaPrincipa (preferible)l:

```
const static int MAX_RECENT_FILES = 5;
```

Las acciones serán unas visibles con los nombres de ficheros, y otras ocultas en caso de no haber una lista tan larga de nombres. Es decir, la lista de nombres puede tener, por ejemplo, sólo 3 ficheros... ¡Ojo que se hace todo en un bucle for!



ventanaprincipal.cpp	constructor o inicialización

```
for (int j = 0; j < MaximoFicherosRecientes; ++j) {
   if (j < ficherosRecientes.count()) {
        QString text = tr("&%1 %2")
                    .arg(j + 1)
                    .arg(nombreCorto(ficherosRecientes[j]));
        accionesFicherosRecientes[j]->setText(text);
        accionesFicherosRecientes[j]->setData(ficherosRecientes[j]);
        accionesFicherosRecientes[j]->setVisible(true);
      } else {
        accionesFicherosRecientes[j]->setVisible(false);
    }
}
```

(seguramente tú habrías puesto primero todas las accionesFicherosRecientes con setVisible(false) y después habrías hecho visibles sólo aquellas necesarias.

¿Qué es nombreCorto() ?

ventanaPrincipal.cpp	metodo auxiliar para obtener el nombre corto de un fichero

```
QString VentanaPrincipal::nombreCorto(const QString
&nombreCompleto)
{
	return QFileInfo(nombreCompleto).fileName();
}
```

Datos cualesquiera dentro de QActions

¿Qué es setData()?

En un buen coche, hay un posavasos, una guantera para gafas de sol, otra para el movil, asientos para personas, etc. pero, por si acaso, siempre hay un maletero para cualquier cosa que haga falta un dia cualquiera llevar. Ese maletero se ha pensado para "*cualquier cosa que necesite el dueño*".

de la misma manera, la clase `QAction` tiene un nombre, un icono, un tip, pero además puede llevar dentro un dato como recurso para necesidades especiales. Como aquí todo es muy diñamico y los nombres de ficheros y menús cambian, vamos a meter en cada qAction el nombre del fichero entero usando `setData()`. Ya veremos después que gracias a este "último recurso desesperado" podemos hacer funcionar el tinglao. Ojo! porque debido a esto, hay dos lugares donde se guardan las rutas completas de los ficheros:

1. En la `QStringList` ficherosRecientes
2. En las Diferentes `QActions`, dentro de `setData()`

Recordatorio de otras cosas que hay que actualizar

en `VentanaPrincipal.h`, hacemos includes y declaraciones de nuevos atributos y de los tres nuevos metodos


```
	#include <QStringList>
...
	QStringList ficherosRecientes;
	enum { MaximoFicherosRecientes = 5 };
	QAction *accionesFicherosRecientes[MaximoFicherosRecientes];
```

```
void establecerFicheroActual(const QString &nombreFichero);
    QString nombreCorto(const QString &nombreCompleto);
    void actualizarActionsFicheros();
...
}
```

Debemos actualizar los puntos donde actualizábamos el nombre del fichero actual. Hay que cambiar cosas ya hechas


```
bool VentanaPrincipal::guardarFichero(QString nombreFichero) {
    ...
    documentoModificado=false;
    rutaFicheroActual = nombreFichero;
    establecerFicheroActual(nombreFichero);
    return true;
}
```

en Abrir Fichero:


```
bool VentanaPrincipal::abrirFichero(QString nombreFichero) {

    documentoModificado = false;
    rutaFicheroActual = nombreFichero;
        establecerFicheroActual(nombreFichero);

    return true;
}
```

[falta terminar esta parte mostrando la carga la ruta del fichero mediante setData en el QAction correspondiente]


```
QStringListIterator ii(ficherosRecientes);
   int indice=0;
   while (ii.hasNext() && indice< MAX_RECENT_FILES) {
     QString ruta = ii.next();
     QString nombreCorto = QFileInfo(ruta).fileName();
     accionesFicherosRecientes[indice]->setText(nombreCorto);
     accionesFicherosRecientes[indice]->setVisible(true);

     accionesFicherosRecientes[indice]->setData(QVariant(ruta));



          editorCentral->append(ruta);
     indice++;
   }
```

Inicialización de lista de acciones y establecimiento del slot

Importante porque tenemos un vector de punteros y hay que crear los objetos:

```
      for (int i = 0; i < MaximoFicherosRecientes; ++i) {
                  accionesFicherosRecientes[i] = new QAction(this);
                  accionesFicherosRecientes[i]->setVisible(false);

            }
```

Pero ¡Son QActions! que se deben conectar de su señal y slot. ¿Cómo si todavía no sé que ficheros son los que deberé abrir, ¡Eso cambia durante la ejecución del programa!

Si hiciese 5 slots, y cambio después la constante MaximoFicherosRecientes a 10 por ejemplo, la fastidiaré, sólo podría llamar a uno de esos 5 slots. De momento lo único que puedo hacer es **conectar todas las señales al mismo slot**.

ventanaprincipal.cpp	
`for (int i = 0; i < MaximoFicherosRecientes; ++i) {` `  accionesFicherosRecientes[i] = new QAction(this);`	

```
            accionesFicherosRecientes[i]->setVisible(false);
            connect(accionesFicherosRecientes[i], SIGNAL(triggered()),
                this, SLOT(abrirFicheroReciente()));
    }
```

No hay que olvidar meter las QActions en el menú ¡O no se verá nada!

```
    for (int i = 0; i < MaximoFicherosRecientes; ++i)
        menuSalir->addAction(accionesFicherosRecientes[i]);
```

¡¡Ahora puedes probar ya todo esto !!

[falta documentar la actualización delas qactions cuando se carga un fichero, esto es: como al cargar un fichero se cambia el texto del menú]

Un slot, múltiples cargas!

Recordemos que todas las QAction de accionesFicherosRecientes están todas ellas conectadas al mismo slot

Recordatorio:

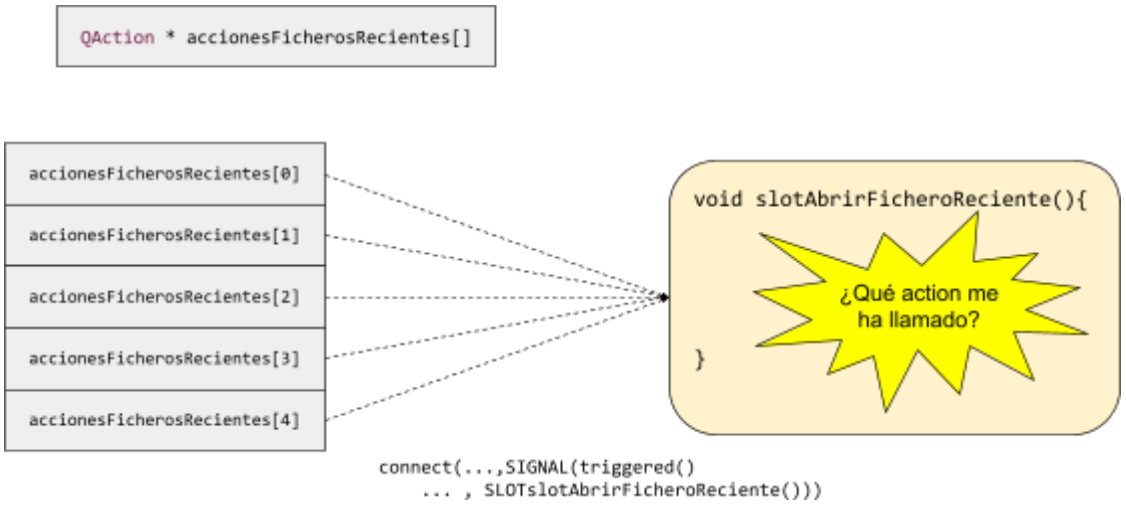
```
    for (int i = 0; i < MaximoFicherosRecientes; ++i) {
        accionesFicherosRecientes[i] = new QAction(this);
        accionesFicherosRecientes[i]->setVisible(false);
        connect(accionesFicherosRecientes[i], SIGNAL(triggered()),
                this, SLOT(slotAbrirFicheroReciente()));
    }
```

Este slot es un poco raro, ya que hay varios QActions con la misma señal que están conectados a él. Es decir, cuando se ejecute el slot, la señal que lo ha desencadenado habrá podido venir de diferentes objetos. Antes de continuar, vamos a sacar en claro lo más importante del slot: debe averiguar el nombre del archivo que el usuario ha seleccionado y llamar al método abrirFichero(QString nombreFichero) para abrirlo.


```
void VentanaPrincipal::slotAbrirFicheroReciente(){
    QString rutaFichero;
    rutaFichero = averiguar_nombre_fichero_elegido_por_usuario_desde_menu()
    abrirFichero(rutaFichero);
}
```

El caso es que el usuario no hace click en el nombre de ningún fichero, sino en una entrada de menú, (una QAction realmente). Vamos a replantear nuestras intenciones: el método anterior (o slot) quiere descubrir qué QAction ha sido la que ha enviado la señal.

Estamos de suerte, porque la aplicación Qt (el omnipresente) que mueve los hilos detrás del escenario, ofrece la posibilidad de descubrir -dentro de un slot- qué objeto ha sido el que ha provocado la señal que ha desencadenado la ejecución del slot




```
QObject * QObject::sender () const [protected]
```

Todas las clases disponen, por herencia de este método, que devuelve un puntero al componente que ha disparado su señal, activando la ejecución del slot. Podemos saber concretamente qué QAction es la culpable así

```
QObject * accionCulpable = sender():
```

Recuerda que dentro de una QAction, habíamos metido un string que contenía la ruta del fichero a abrir. Si ya tengo la QAction averiguada, ¿Cómo recupero el QString ? Recordemos que se inserta con setData la ruta de un archivo en la QAction


```
QAction * unaAccion = new QAction(...:
unaAccion->setData(QString("/home/pepe/...");
...
QString rutaFichero = unaAccion->data();
```

Ahora, al reaccionar desde el slot a una QAction, podemos rescatar o recuperar esa ruta de archivo. Observa la siguiente propuesta de slot

```
1    void VentanaPrincipal::slotAbrirFicheroReciente(){
2        QString rutaFichero;
3        QObject * accionCulpable = sender():
4        QString rutaFichero = accionCulpables->data();
5        abrirFichero(rutaFichero);
     }
```

En la línea 3 descubrimos la QAction que ha sido disparada (esto indica qué entrada de menú se ha activado). En la siguiente línea (4) , a partir de esa acción, recuperamos la ruta del archivo asociado a esa entrada de menú. En la línea 5, finalmente llamamos a un método para abrir el archivo, pasándole la ruta

El problema es que el compilador te descubre un problema muy muy claro: accionCulpable esta del tipo QObject porque la documentación es bastante clara en cuanto al tipo de dato devuelto por `QObject::sender()`. Resulta ser un `QObject *`. Y por eso hemos declarado:

```
QObject * accionCulpable
```

Compilador no está de acuerdo, e indica que la clase QObject no tiene el método data(); Y tiene toda la razón. Probemos a cambiar el tipo a QAction. Observa el siguiente código y prúebalo:

ventanaprincipa.cpp	

```
void VentanaPrincipal::slotAbrirFicheroReciente(){
    QString rutaFichero;
    QAction * accionCulpable = sender();
    QString rutaFichero = accionCulpables->data();
```

```
    abrirFichero(rutaFichero);
}
```

Tampoco funciona. Ahora el compilador protesta en la línea

```
    QAction * accionCulpable = sender();
```

Y otra vez con razón. El compilador no está seguro que el objeto devuelto por sender() sea un QAction*. En la ayuda dice que es un QObject * y no todos los QObject * pueden pasar por QAction *... aunque ¡ Tú estás seguro de ello, porque a este slot sólo hay conectadas señales de cinco QActions !

Hay que **obligar al compilador** a aceptar nuestros deseos porque estamos seguros que todos los QObject * devueltos por sender() sólo pueden ser realmente QActions *. Esto lo logramos haciendo un "casting", esto es: obligar al compilador a convertir tipos de datos

```
QObject * accionCulpable = sender();
QAction *accionLanzada = qobject_cast<QAction*>( accionCulpable );
```

Con ello completamos el método y ya funciona

Resumen

Resumen de cambios para la funcionalidad del menú de "ficheros recientes"

- Nueva QStringList "`ficherosRecientes`"
- Nuevo método para actualizar la lista `ficherosRecientes` con el nombre del fichero actual: "`establecerFicheroActual(QString)`"
- El método `establecerFicheroActual(QString)` es invocado desde `abrirFichero(QString)` y `guardarFichero(QString)`
- Nuevo vector `QAction* accionesFicherosRecientes[]`
- Inicializar `accionesFicherosRecientes` a `QActions` sin texto o vacías e invisibles
- Conectar todas las QAction de `accionesFicherosRecientes` al mismo slot "`slotAbrirFicheroReciente()`", y ...
- Crear dicho slot que es invocado desde señales de accionesFicherosRecientes. Este slot Se llama `slotAbrirFicheroReciente()`

- Crear entradas de menú con los elementos de `accionesFicherosRecientes` (son invisibles recién inicializados)
- Crear un método para mantener la lista de QActions de accionesFicherosRecientes al día: método `actualizarActionsFicheros();`. Este método se invoca cada vez que se guarda o abre un fichero… al igual que `establecerFicheroActual(QString).` por tanto, este método es invocado desde el final del método anterior `establecerFicheroActual(QString)`
- En el slot `slotAbrirFicheroReciente()`
    - descubrir cada vez que se ejecuta qué QAction ha sido la causante de que se termine ejecutando el slot.
    - Averiguar cuál es el string interno que lleva la QAction. Usarlo como nombre de fichero
    - llamar a `abrirFichero(QString)` pasándole el nombre del fichero anterior.

Diálogos

El programa hasta ahora sólo tiene una ventana principal y algún diálogo prefabricado (como el de abrir un fichero). Habitualmente todos los programas muestran diálogos en algún momento de su ejecución (por ejemplo el diálogo para buscar una palabra en un texto). Aunque ya sabemos hacer diálogos propios, no hemos practicado la situación siguiente:

1. El programa arranca y muestra la ventana principal
2. En algún momento el usuario, mediante el menú por ejemplo, invoca la aparición de algún diálogo.
3. El diálogo se crea (si es necesario) y se muestra.
4. El diálogo muestra o recibe datos desde la ventana principal.
5. El usuario manipula el diálogo, provocando cambios que se propagan a la ventana principal. Estos cambios en la ventana principal pueden manifestarse
    a. Inmediatamente que se cambia algún componente en el diálogo
    b. Al cerrar el diálogo pulsando "aceptar" o "aplicar"
6. El diálogo debe ser cerrado y quizá recoger el resultado de su ejecución

El aspecto más problemático y confuso al que vamos a enfrentarnos no es en sí los diálogos, sino el intercambio o compartición de datos entre la ventana principal y los diálogos. Primero aprenderemos técnicas que permiten intercambiar datos de cualquier forma posible y después haremos elecciones que sean adecuadas para la mantenibilidad, poco acoplamiento y alta cohesión de los datos.

Incorporar un diálogo al programa

Los diálogos se han de crear independientemente de la ventana principal. Básicamente hay que heredar de la clase QDialog, añadir los componentes y comportamiento deseado y, finalmente, entregar un fichero .h y otro .cpp que implementan el diálogo. Es posible también crear el diálogo usando Designer para facilitar el diseño de la interfaz.

Aquí vamos a tomar un diálogo que hicimos anteriormente llamado FindDialog y que está disponible mediante dos ficheros (que usaremos como ejemplo)

- finddialog.cpp
- finddialog.h

EL código está disponible más abajo

Si no tienes estos ficheros, de momento, haz un diálogo (con el designer) con algún componente menor (un par de botones) y usa ese diálogo para la primera parte de este apartado (mostrar el diálogo desde un menú)

Pasos generales:

Es muy importante que retengas bien estos pasos porque habitualmente siempre deberás seguirlos para crear un diálogo dentro de una aplicación. La siguiente tabla es un **recordatorio-chuletario** para repasar los pasos necesarios para que aparezca un nuevo diálogo cuando se elige una entrada de menú. Después se van a explicar y detallar estos pasos.

Pasos	**Lugar o destino**	**Observaciones**
Crear el diálogo con el **Designer** (si es el caso)		
Usar el Designer para diseñar un Diálogo	Generar un fichero .ui findDialog.ui	Hay que rehacer el proyecto con `qmake -project` y ... Al compilar se generará un fichero `ui_findDialog.h`

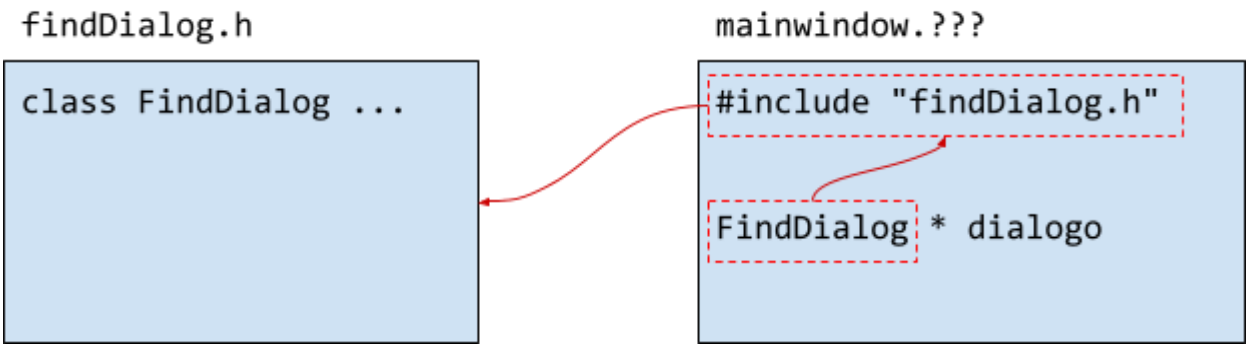
Rehacer el proyecto por haber añadido un fichero nuevo
`qmake -project`
`echo "QT += widgets" >>...`
`qmake`
	fichero .pro del proyecto	Alternativamente usa el script "haz.sh"

Preparar ficheros de cabecera e implementación de la clase.		
Crear los ficheros .h y .cpp del diálogo y programar la clase **`findDialog.h`** **`findDialog.cpp`**	carpeta del proyecto	`#include "ui_finddialog.h"`
`#incluir "findDialog.h"`	fichero ".h" de la ventana principal (ventanaprincipal.h)	

Preparar menús para la ventana principal		
Declarar **`QAction * accBuscar`**	`ventanaprincipal.h`	
Crear QAction **`QAction *accBuscar;`**	ventanaprincipal.cpp (constructor o método incializador)	`accBuscar = new QAction("..`
Añadir la QAction al menú	ventanaprincipal.cpp (constructor o método incializador)	`menu->addAction(...`
Conectar QAction con slot	ventanaprincipal.cpp (constructor o método incializador)	El slot deberás declararlo y crearlo más adelante
Preparar slot asociado a la entrada del menú		
Declarar slot void slotBuscar(...)	`ventanaprincipal.h`	sin argumentos porque la señal triggered de la acción no tiene argumentos.
Implementar slot	`ventanaprincipal.cpp`	nuevo método. no olvides `"MainWindow::"`
Crear y mostrar diálogo		
Declarar un puntero al diálogo	en ventanaprincipal.h	Si el diálogo se va a reutilizar o no es modal
	en el slot	Si el diálogo es modal y se destruye al cerrar
Inicializar el diálogo	Inicialmente en el constructor de mainwindow a NULL	poco habitual pero posible
	en el slot	Es lo habitual

#incluir el diálogo

El nuevo menú tendrá un fichero de cabecera, por ejemplo: `findDialog.h`. Este diálogo tiene la declaración de la clase del diálogo y esa declaración necesita ser conocida por cualquier otro fichero que quiera usar la clase del diálogo.



En ventanaprincipal.h añadimos #include "findDialog.h". En este momento, estamos ampliando nuestro proyecto con más clases, más archivos .h y .cpp. Hay que regenerar el proyecto

Preparar el menú

Añadimos la nueva acción que se convertirá en una entrada de menú nueva

ventanaprincipal.h

```
QAction *accionBuscar;
```

En ventanaprincipal.cpp, creamos todo lo necesario para mostrar el menú:

ventanaprincipal.cpp

```
void VentanaPrincipal::createActions() {
    ...
        accionBuscar = new QAction(tr("Buscar"),this);
        accionBuscar->setShortcut(tr("Ctrl+F"));
        accionBuscar->setStatusTip(tr("Buscar"));
        connect(accionBuscar,SIGNAL(trigger()),
            this, SLOT(slotDialogoBuscar()));
```

Observa cómo queda conectada la señal de la QAction al slot slotDialogoBuscar

Creamos ahora el menú si hace falta (la alternativa es meter la QAction en un menú ya creado)

declaramos (en ventanaprincipal.h, pero no necesariamente)

```
QMenu * menuSalir, *menuEditar ;
```

Y ponemos la QAction dentro del menú

```
void VentanaPrincipal::createMenus(){
    ....
    menuEditar = menuBar()->addMenu(tr("Editar"));
    menuEditar->addAction(accionBuscar);
}
```

Crear el slot

Este slot se ejecutará cuando se seleccione la nueva entrada del menú.

ventanaprincipal.h	Declaración

```
private slots:
     ...
     void slotDialogoBuscar();
```

en ventanaprincipal.cpp lo implementamos (de momento vacío)

ventanaprincipal.cpp	Implementación

```
void VentanaPrincipal::slotDialogoBuscar(){ }
```

Ya compila, aparece el menú y se activa el slot anterior (vacío) cuando el usuario selecciona esa entrada de menú. Hemos de completar el menú y lograr que se muestre el diálogo.

Mostrar el diálogo

Crear y mostrar. Dos maneras:

1. Modal
2. No modal

Con el diálogo **modal**, se muestra el diálogo, y el programa (o el motor de qt) detiene la ejecución de la ventana principal hasta que el diálogo se cierra

Por contra, con un diálogo **no modal**, se muestra el diálogo, y el programa (o el motor de qt) no detiene la ejecución de la ventana principal, de forma que se puede interactuar tanto con la ventana como con el diálogo creado.

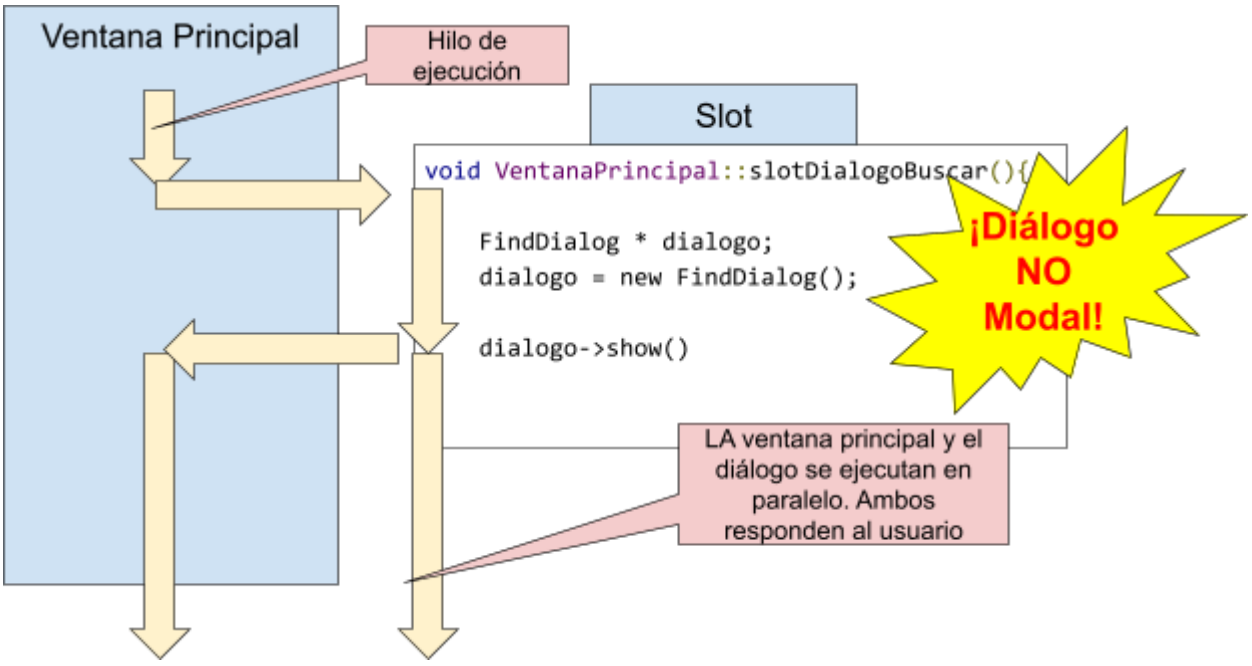
Durante el curso, los QMessageBox que ya hemos usado son ejemplos de diálogos **modales**.

Diálogo No Modal

Vamos a tratar de entender el funcionamiento de los diálogos no modales. Supongamos que en el slot asociado al menú, se despliega un diálogo no modal. Para ello invocamos el método show() sobre un diálogo

```
void VentanaPrincipal::slotDialogoBuscar(){
      FindDialog * dialogo = new FindDialog();

      dialogo->show()
} // el diálogo está creado y funciona pero el método éste ya ha terminado
```

show() mostrará el diálogo creado por la pantalla, y el programa seguirá inmediatamente con la ejecución de la siguiente instrucción. Cuando el slot termina, el diálogo se queda visible y en ejecución, y la ventana principal sigue igualmente con su ejecución



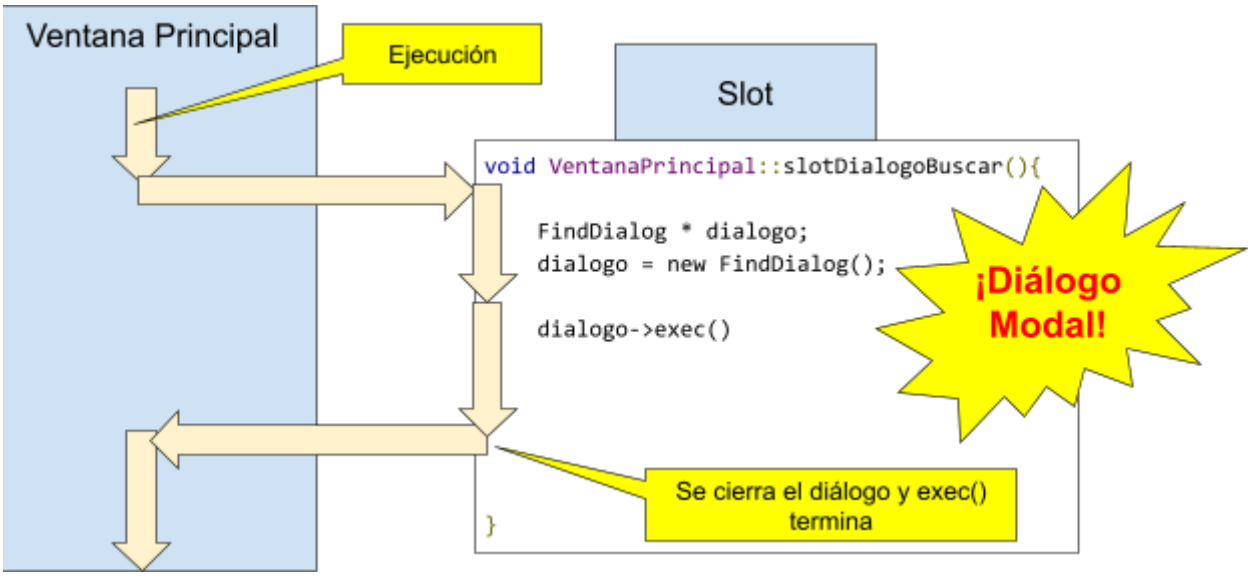
Dialogo Modal

El diálogo modal sólo difiere en el nombre del método usado para mostrar el diálogo. Todo lo demás (en cuanto a programación y preparación) es igual

	Versión modal

```
void VentanaPrincipal::slotDialogoBuscar(){
      FindDialog * dialogo = new FindDialog();

      dialogo->exec() // de aquí no se sale hasta que el usuario
      cierra el diálogo

}
```

La diferencia es que el programa se detiene en la línea "dialogo->exec()" mientras el diálogo sea visible, y se continuará la ejecución cuando se cierre.



Ya hemos percibido esta forma de mostrar y gestionar diálogos. En concreto con QMessageBox. Allí, hasta que no se cerraba el mensaje emergente, el programa no continuaba su ejecución, de forma que podíamos recoger la respuesta.

	Ejercicios anteriores

```
void VentanaPrincipal::slotNuevo(){

    int respuesta = QMessageBox::warning(this,
                    tr("nuevo doc"),
                    tr("Borro el documento ?"),
                    QMessageBox::Yes | QMessageBox::No);

    if (respuesta == QMessageBox::Yes)
       editorCentral->document()->clear();
```

En el ejemplo anterior, el mensaje emerge en la línea resaltada, y la variable respuesta obtiene el valor cuando se cierra el mensaje. Este es un comportamiento modal del diálogo emergente

Observa la siguiente consecuencia de los diálogos modales y no modales.

> Los diálogos modales se pueden crear con una variable local, porque mientras se ejecuta la función exec() la variable y el objeto existen, y la función donde han sido creados no ha finalizado.
> Con los diálogos no modales debemos usar objetos creados con new y preferentemente guardar sus referencias con punteros, ya que los diálogos creados siguen existiendo después de que finalice la ejecución de las funciones donde han sido creados.

Por tanto, las siguientes posibilidades son ambas correctas (a falta de limpiar la memoria después)

```
void VentanaPrincipal::slotDialogoBuscar(){
      FindDialog * dialogo = new FindDialog();

      dialogo->exec()
}
```

```
void VentanaPrincipal::slotDialogoBuscar(){
      FindDialog  dialogo;
      dialogo.exec() ;
}
```

Observa que en el primer caso, el objeto diálogo creado con new sigue creado después de finalizar el slot y aunque se destruya el puntero `dialogo`. En el segundo caso cuando la variable diálogo se destruya se destruye también el diálogo, el diálogo ya habrá sido cerrado antes.

Crear el diálogo

¿Cómo se llama la clase del objeto que vamos a crear? Miremos el fichero "finddialog.h"

… " class FindDialog : public QDialog"

Como es no modal, hay que almacenar un puntero porque se va a crear con new, (podría ser local, pero haríamos un memory leak)

ventanaprincipal.h	Dentro de la declaración de la clase VentanaPrincipal

```
FindDialog * dialogoBuscar;
```

Luego hay que declarar un puntero en ventanaprincipal.h

Luego hay que #include "finddialog.h"

el slot queda así:

```
void VentanaPrincipal::slotDialogoBuscar(){
 /* crear */
  dialogoBuscar = new FindDialog();
  /* mostrar */
}
```

(funciona no modal porque POR DEFECTO el modo es no modal)

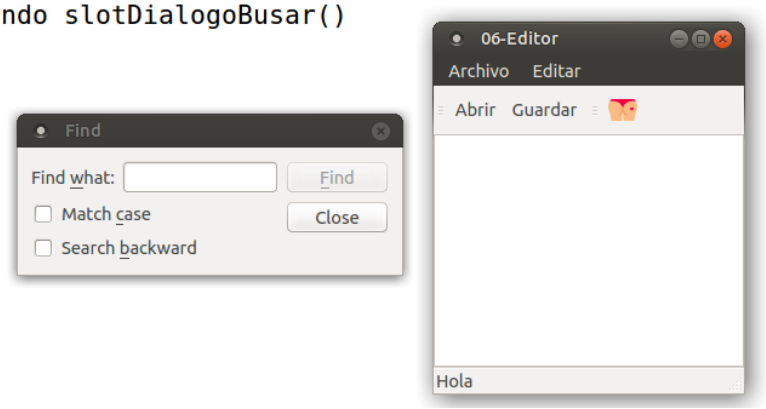
Para poder realizar experimentos que muestren la diferencia entre modal y no modal, vamos a mostrar mensaejs de texto por el terminal y veremos la diferencia. añade la siguiente línea en el lugar propuesto

```
void VentanaPrincipal::slotDialogoBuscar(){
    /* crear */
    dialogoBuscar = new FindDialog();
```

```
    /* mostrar */
    dialogoBuscar->show();
    qDebug() << "Finalizando slotDialogoBusar()"<< endl;
}
```

(haz "#include <QDebug>" en ventanaprincipal.cpp)

```
lliurex@HiDi-Tec:~/Documentos/Interfaces2016-2017/qt/06-Editor$ ./06-Editor
Finalizando slotDialogoBusar()
```



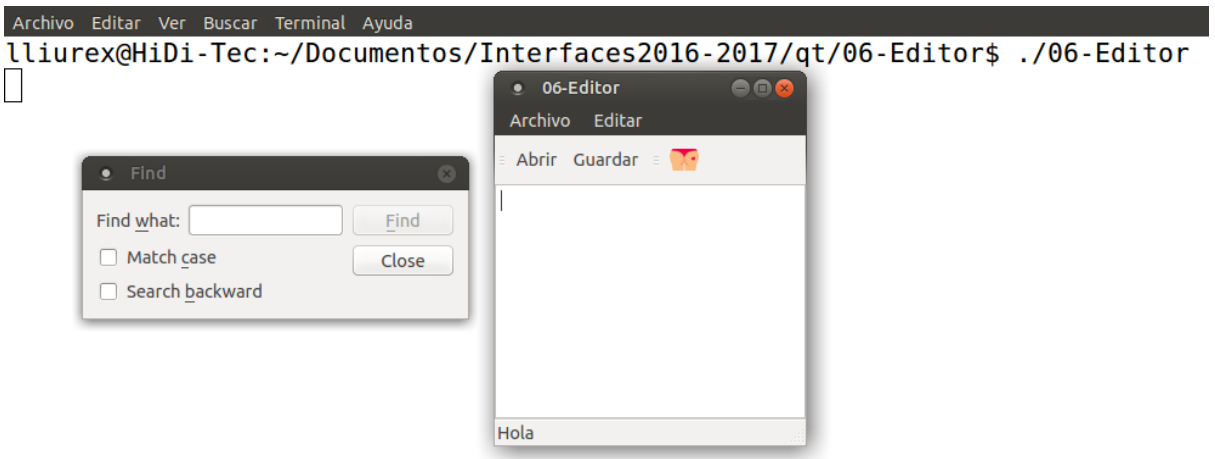
observa que el diálogo funciona (reacciona a introducir texto en el campo de texto). Pero ¡ La ventana principal sigue funcionando totalmente ! Puedes escribir texto

Diálogo no modal.

en Qt se puede activar o lanzar de varias maneras . Usaremos el método exec()

```
void VentanaPrincipal::slotDialogoBuscar(){
    /* crear */
    dialogoBuscar = new FindDialog();
    /* mostrar */
    dialogoBuscar->exec();
    qDebug() << "Finalizando slotDialogoBusar()"<< endl;
}
```

Observa el efecto, no se ejecuta la línea "qDebug()<< … " hasta que no cierras el diálogo. Por ello, cuando finaliza el método, se podría y debería destruir el diálogo. Si alcanzas el estado de la siguiente imagen podrás ver no sólo lo que aquí se indica, sino que ¡ La ventana principal no reacciona a pulsaciones !



La destrucción del diálogo se hace así

ventanaprincipal.cpp

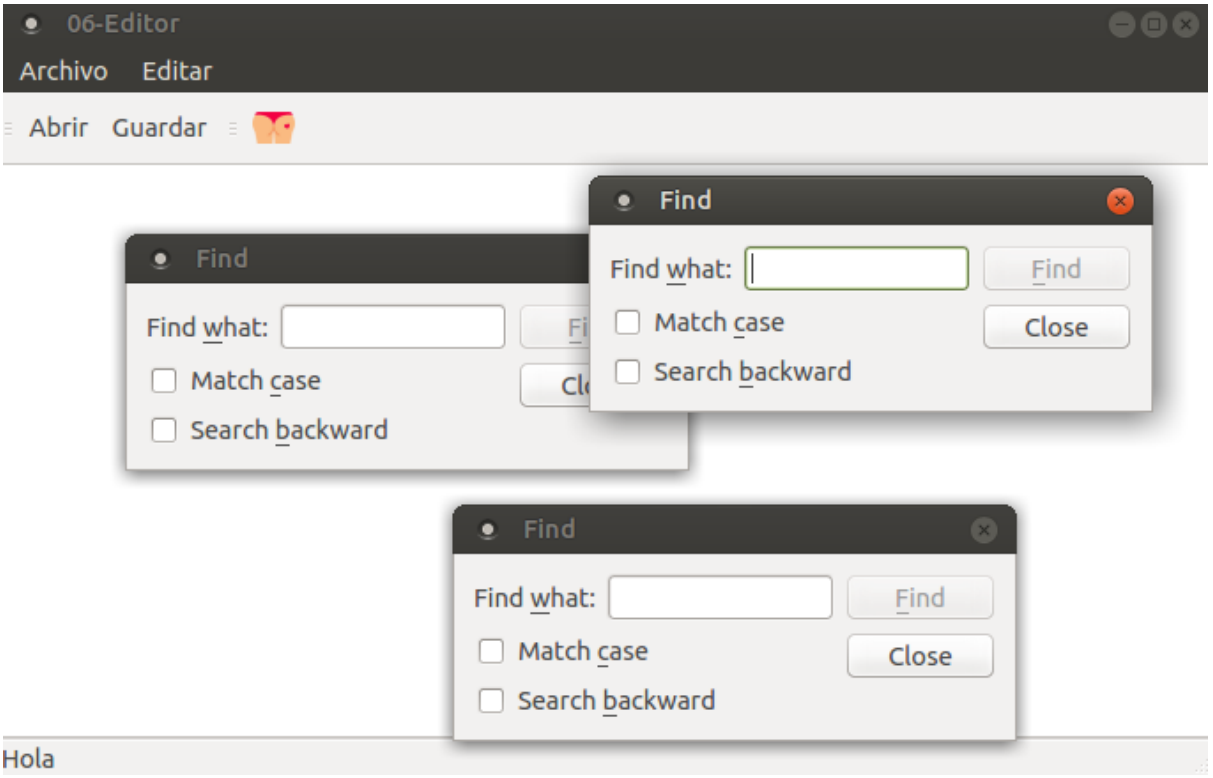
```
void VentanaPrincipal::slotDialogoBuscar(){
    /* crear */
    dialogoBuscar = new FindDialog();
    /* mostrar */
    dialogoBuscar->exec();
    qDebug() << "Finalizando slotDialogoBusar()"<< endl;
    delete dialogoBuscar;
}
```

Vida del diálogo

vamos a restaurar la No modalidad de nuestro diálogo (y por tanto hemos de evitar borrar algo que va a perdurar) y vamos a activar repetidas veces el menú. observa lo que ocurre

```
void VentanaPrincipal::slotDialogoBuscar(){
```

```
    /* crear */
    dialogoBuscar = new FindDialog();
    /* mostrar */
    dialogoBuscar->show();
    qDebug() << "Finalizando slotDialogoBusar()"<< endl;
}
```



Si tenemos la precaución de cerrar un diálogo antes de abrir el siguiente, la cagamos igualmente, porque la tercera vez que hayamos pasado por el slot, habremos creado tres diálogos diferentes (dos de los cuales están vivos pero ocultos)

Deberíamos

- No crear más diálogos cuando ya hay uno creado
- No crear diálogos que no podamos destruir

Nuestra solución va a ser la siguiente:

1. Al arrancar el programa no creamos ningún diálogo

2. Si el usuario usa el menú para buscar y desencadena la ejecución del slot buscar, creamos irremediablemente un objeto FindDialog y lo mostramos.
3. Cuando el usuario cierra el diálogo, éste no se destruye, tan sólo se oculta pero sigue creado.
4. Cuando el usuario vuelve a activar el slot, deberemos comprobar si el diálogo ya fue creado y entonces no volverlo a crear, sólo mostrarlo

La clave está en el slot, hay que saber si el atributo dialogoBuscar ya referencia a un objeto ya creado o todavía no. y ello pasa por inicializar el atributo en el inicio del programa a un valor conocido que permita descubrirlo. Dado que es un puntero, vamos a inicializarlo a NULL. Esto, en un atributo normal (atributo de instancia), siempre lo hacemos en el constructor, nunca en la declaración.

ventanaprincipal.cpp

```
VentanaPrincipal::VentanaPrincipal(..) {}

      dialogoBuscar = NULL
```

Por supuesto, este atributo tendrá "vida" durante toda la existencia de la propia ventana y por ello estará declarado en la clase

ventanaprincipa.h

```
class VentanaPrincipal : public QMainWindow {
...
public:
   FindDialogo *dialogoBuscar
```

Ahora en el slot, usamos el valor de inicialización para saber si ya existe un diálogo o no.

ventanaprincipal.cpp

```
void VentanaPrincipal::slotDialogoBuscar(){
    /* crear... si hace falta */
    if (dialogoBuscar == NULL )
       dialogoBuscar = new FindDialog();
    /* mostrar en cualquier caso */
    dialogoBuscar->show();
    qDebug() << "Finalizando slotDialogoBusar()"<< endl;
}
```

Interacción diálogo <-> ventana principal

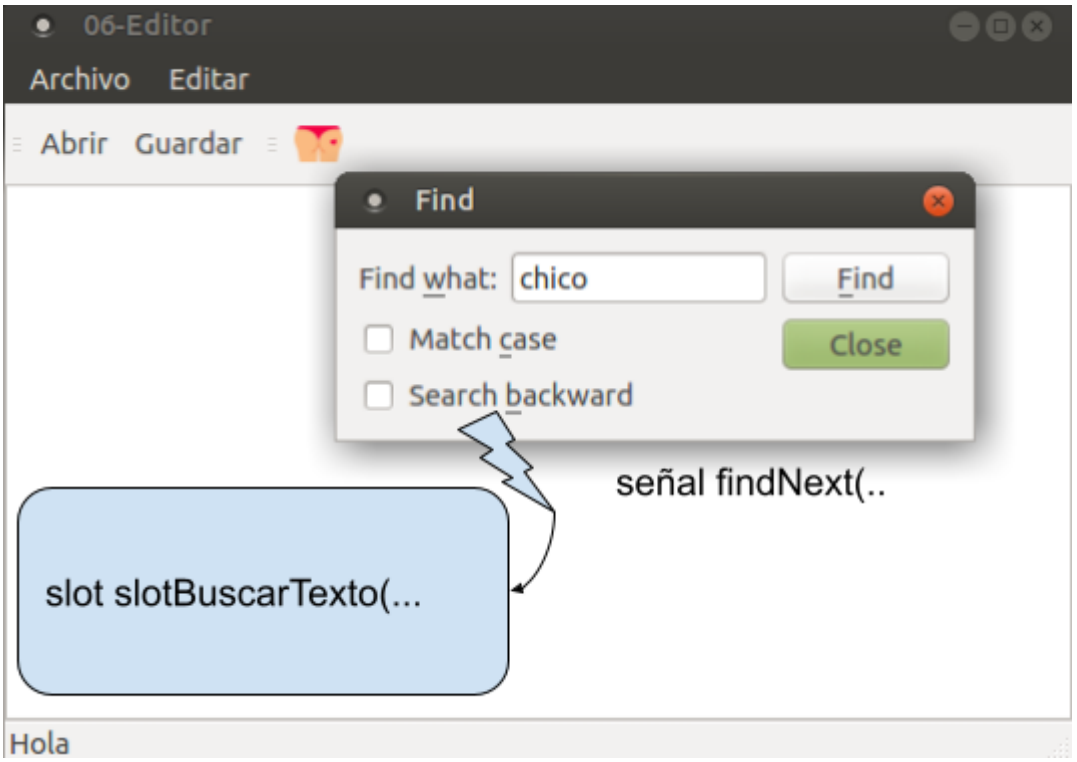
El diálogo "buscar" va a estar abierto y visible, mientras la ventana principal sigue activa y usable. Además, si se manipula el diálogo para realizar búsquedas, la ventana principal tiene que realizar efectivamente esas búsquedas. Dicho de otra forma, el diálogo y la ventana deben intercomunicarse.

Habrá muchas situaciones diferentes y necesidades. cada una a su vez puede tener varias soluciones imperfectas. En este caso el diálogo buscar debe "ordenar" a la ventana que realice cierta búsqueda.

Señales en el diálogo

Una de las maneras de interactuar es mediante señales que un diálogo emite. En el ejemplo que estamos siguiendo, el diálogo emitía dos señales destinadas a realizar una búsqueda de cadenas. En el ejercicio donde se hizo y probó el diálogo, estas señales nos se usaban porque no se podía conectar a nada.

Pero aquí vamos a conectar la señal del diálogo a un slot de la ventana principal (que es donde deben cambiar ciertas cosas)



hay que crear un nuevo slot y realizar la conexión de la señal del diálogo a este slot

```
void slotBuscarTexto();
```

Esta declaración tiene un problema. Observa la declaración de la señal (o el lugar desde el que se hace el emit)

finddialog.h

```
signals:
   void findNext(const QString &str, Qt::CaseSensitivity cs);
   void findPrevious(const QString &str, Qt::CaseSensitivity cs);
```

Las señales llevan argumentos porque junto con ellas se transmiten datos a los slot a los que estén conectadas. Estos datos importan (son el texto que se está buscando), y por tanto el slot debe recoger lo que la señal transmite. Ello marca los argumentos del slot… tan fácil como copiar

```
const QString &str, Qt::CaseSensitivity cs
```

y pegarlo para dejar la declaración (e implementación) así:

```
void slotBuscarTexto(const QString &str, Qt::CaseSensitivity cs);
```

Conexión de una señal de diálogo con slot de VentanaPrincipal.

Date cuenta de que no podemos realizar un connect de una señal de un objeto que no existe. Es decir, en el constructor o en los métodos de la inicialización de la ventana principal, el diálogo no existe y no se puede establecer la conexión. Sólo cuando exista el diálogo podremos realizar el connect, que además, debe hacerse sólo una vez. El lugar adecuado es justo después de crear el diálogo

ventanaprincipal.cpp

```
void VentanaPrincipal::slotDialogoBuscar(){
        /* crear */
   if (dialogoBuscar == NULL ){
       dialogoBuscar = new FindDialog();
       connect(
          dialogoBuscar,
          SIGNAL(findNext(const QString &, Qt::CaseSensitivity ),
```

```
        this,
        SLOT(slotBuscarTexto(const QString &, Qt::CaseSensitivity
    )));
        }
        /* mostrar */
        dialogoBuscar->show();
        qDebug() << "Finalizando slotDialogoBusar()"<< endl;
}
```

Ya funciona la conexión. Ahora hay que investigar en el assitant cómo buscar algo en un texto.. Ya está hecho, sólo hay que encontrar el método adecuado , bien de la clase QTextEdit o QTextDocument. En QTextEdit encontramos:

```
bool QTextEdit::find (  const QString & exp,
                        QTextDocument::FindFlags options = 0 )
```

Finds the next occurrence of the string, exp, using the given options. Returns true if exp was found and changes the cursor to select the match; otherwise returns false.

Así pues, éste es el método al que hay que llamar para que haga el trabajo "sucio".

ventanaprincipal.cpp	**slot para buscar texto**

```
void VentanaPrincipal::slotBuscarTexto(const QString &str,
Qt::CaseSensitivity cs){
        editorCentral->find(str);
}
```

De momento, aquí ignoramos el valor del parámetro `cs`.

Si se quiere respetar el parámetro cs para hacer una búsqueda sensible a mayúsculas y minúsculas:

ventanaprincipal.cpp	

```
void VentanaPrincipal::slotBuscarTexto(const QString &str,
Qt::CaseSensitivity cs){
    QTextDocument::FindFlags flags =
    QTextDocument::FindCaseSensitively;
    if (cs == Qt::CaseInsensitive)
            flags = flags & ! QTextDocument::FindCaseSensitively;
        editorCentral->find(str, flags);
```

```
    }
```

Falta conectar la otra señal del diálogo (findPrevious) a otro slot que debería tener un nombre slotBuscarTextoAtras… y por tanto éste que hemos hecho debería tener un nombre slotBuscarTextoAdelante()

Código FindDialog

Aquí tienes la clase FindDialog usada en el ejercicio anterior

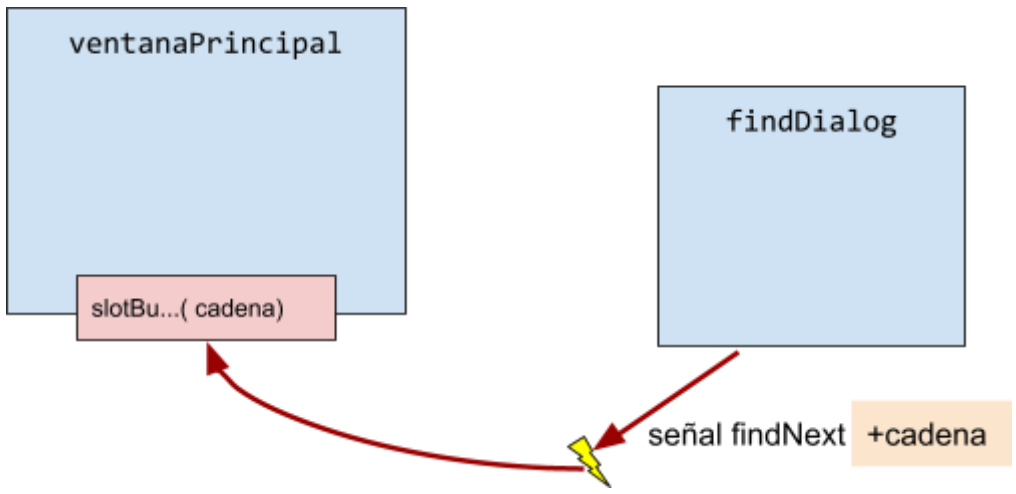
finddialog.cpp

```
#include <QtGui>
#include <QDebug>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QGridLayout>
#include "finddialog.h"

FindDialog::FindDialog(QWidget *parent) : QDialog(parent)

 {

    label = new QLabel(tr("Find &what:"));
    lineEdit = new QLineEdit;
      label->setBuddy(lineEdit);

    caseCheckBox = new QCheckBox(tr("Match &case"));
    backwardCheckBox = new QCheckBox(tr("Search &backward"));
    findButton = new QPushButton(tr("&Find"));
    findButton->setDefault(true);
    findButton->setEnabled(false);
    closeButton = new QPushButton(tr("Close"));

    connect(lineEdit, SIGNAL(textChanged(const QString &)),
      this, SLOT(enableFindButton(const QString &)));
    connect(findButton, SIGNAL(clicked()),
      this, SLOT(findClicked()));
    connect(closeButton, SIGNAL(clicked()),
      this, SLOT(close()));
```

```
    QHBoxLayout *topLeftLayout = new QHBoxLayout;
    topLeftLayout->addWidget(label);
    topLeftLayout->addWidget(lineEdit);
    QVBoxLayout *leftLayout = new QVBoxLayout;
    leftLayout->addLayout(topLeftLayout);
    leftLayout->addWidget(caseCheckBox);
    leftLayout->addWidget(backwardCheckBox);

       QVBoxLayout *rightLayout = new QVBoxLayout;
       rightLayout->addWidget(findButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();

    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);


    setLayout(mainLayout);

    setWindowTitle(tr("Find"));
    setFixedHeight(sizeHint().height());
}


 void FindDialog::findClicked()
 {
     QString text = lineEdit->text();
     Qt::CaseSensitivity cs =
          caseCheckBox->isChecked() ? Qt::CaseSensitive
                                    : Qt::CaseInsensitive;
     if (backwardCheckBox->isChecked()) {
     emit findPrevious(text, cs);
     } else {
     emit findNext(text, cs);
     }
 }

 void FindDialog::enableFindButton(const QString &text)
 {
     bool estaVacia;
     estaVacia = text.isEmpty();
     bool hayAlgoDeTexto = !estaVacia;
     findButton->setEnabled(hayAlgoDeTexto);
 }
```

finddialog.h

```
#ifndef FINDDIALOG_H
#define FINDDIALOG_H
#include <QCheckBox>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
#include <QDialog>

class FindDialog : public QDialog
{
      Q_OBJECT

 public:
      FindDialog(QWidget *parent = 0);

      QLabel *label;
      QLineEdit *lineEdit;
      QCheckBox *caseCheckBox;
      QCheckBox *backwardCheckBox;
      QPushButton *findButton;
      QPushButton *closeButton;


signals:
      void findNext(const QString &str, Qt::CaseSensitivity cs);
      void findPrevious(const QString &str, Qt::CaseSensitivity cs);

 private slots:
      void findClicked();
      void enableFindButton(const QString &text);

 };
#endif
```



Ejercicio ampliación

simulación de examen

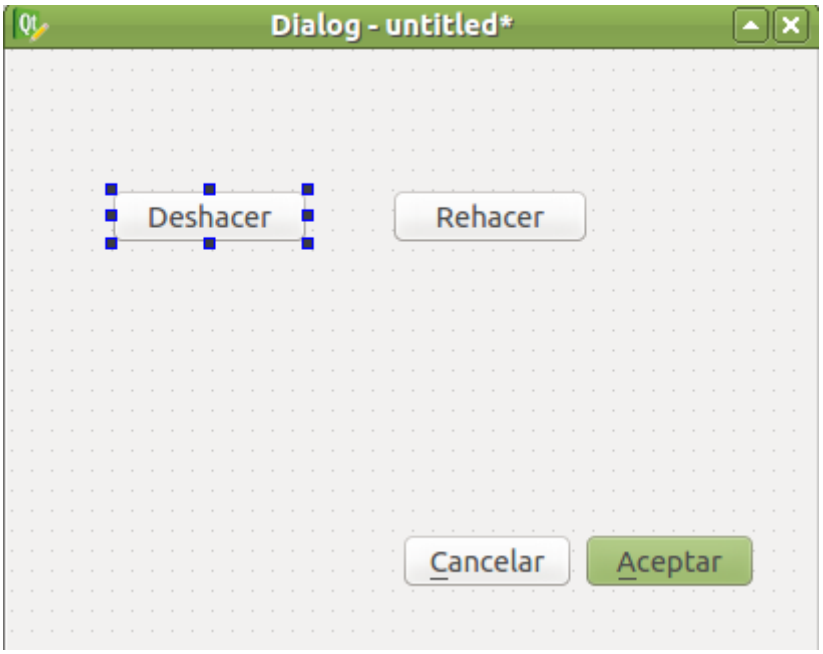
Añadir un menú que al ser lanzado desplegará un diálogo con dos botones. Un botón mostrará "DesHacer" y el otro "ReHacer"

El botón de deshacer, deshará los cambios hechos en el texto igual que ocurre cuando se pulsa la tecla [ctrl +z], mientras que el botón de rehacer equivale a "redo" o [ctrl + may + z].

Los botones deben deshabilitarse cuando no hay posibilidad de deshacer o rehacer. Los dos botones estarán conectados al mismo slot.

solución

Con el designer, crear este diálogo llamado DialogoDeshacer



Importante! Poner nombres a :

- El botón de Dehacer: **`botonDeshacer`**
- El botón de Rehacer : **`botonRehacer`**
- Al diálogo entero : "**`DialogoDeshacer`**"

Guardamos como `DialogoDeshacer.ui` y regeneramos el proyecto.

Creamos los esqueletos de la nueva clase DialogoDeshacer

DialogoDeshacer.h

```
#ifndef DIALOGODESHACER_H
#define DIALOGODESHACER_H
#include "ui_dialogoDeshacer.h"

 class DialogoDeshacer : public QDialog
{
      Q_OBJECT

 public:
      DialogoDeshacer(QWidget *parent = 0);
};
#endif
```

DialogoDeshacer.cpp

```
#include "DialogoDeshacer.h"

 DialogoDeshacer::DialogoDeshacer(QWidget *parent)
      : QDialog(parent)
  {

}
```

… mas preparativos, Hay que crear el menú, Qaction, slots y demás elementos necesarios para permitir al usuario elegir la opción de abrir el diálogo desde la ventan principal

en ventanaprincipal.h

```
QAction * accionDialogoDeshacer;

void slotAbrirDialogoDeshacer();
```

en ventanaprincipal.cpp, en el constructor:

```
dialogoDeshacer = NULL;
```

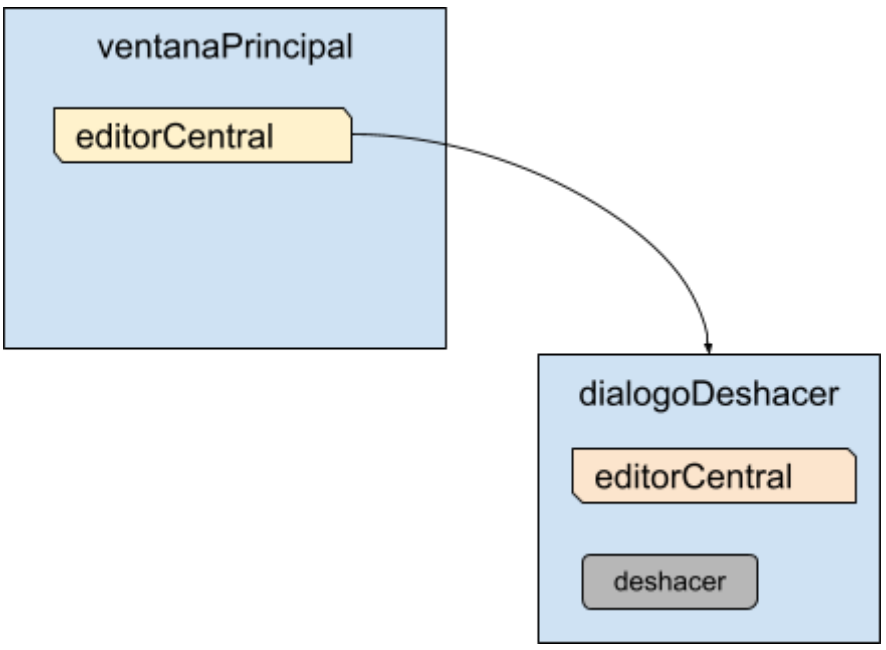
la creación de la acción y su conexión

```
      accionDialogoDeshacer = new QAction("Deshacer",this);
      accionDialogoDeshacer->setIcon(QIcon("./images/abrir.png"));
      accionDialogoDeshacer->setStatusTip("Deshacer y rehacer");
      accionDialogoDeshacer->setToolTip("Deshacer y rehacer");

      connect(accionDialogoDeshacer,SIGNAL(triggered()),
            this,SLOT(slotDialogoDeshacer()));
```

el slot en MainWindow

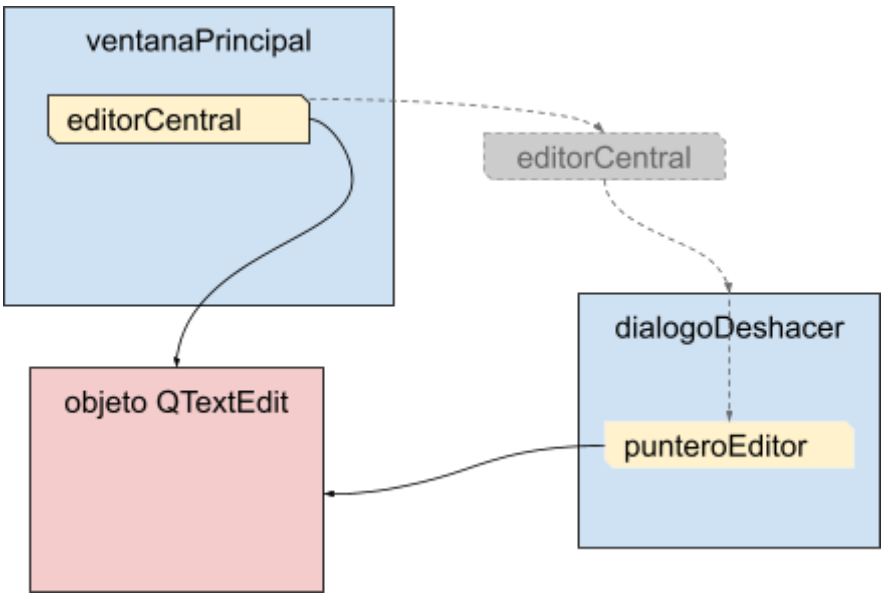
```
void VentanaPrincipal::slotDialogoDeshacer(){

    if (dialogoDeshacer == NULL ) {
       dialogoDeshacer = new DialogoDeshacer(this);
    }

    dialogoDeshacer->show();
}
```

Con esto ya tenemos la posibilidad de lanzar el diálogo de una forma convencional. El problema es que **desde el diálogo se debe acceder al editor central** porque vamos a actuar sobre él.



Más real… Tanto la ventana principal como el diálogo debe tener un puntero al mismo objeto QTextEdit que es el editorCentral creado.

QTextEdit * editorCentral



Este puntero se pasa o comunica al dialogoDeshacer desde la ventana principal preferentemente en el constructor, por tanto el constructor de dialogoDeshacer

```
DialogoDeshacer( QTextEdit *, QWidget * parent = 0);
```

y su implementación. observa también la conexión del botón con su slot que es el siguiente

```
void DialogoDeshacer::slotBotonDeshacer(){

    punteroATextEdit->undo();

}
```

este punteroATextEdit está declarado en la clase DialogoDeshacer

```
class DialogoDeshacer :  public QDialog, public Ui::DialogoDeshacer {
Q_OBJECT
public:
    DialogoDeshacer( QTextEdit *, QWidget * parent = 0);
```

```
    QTextEdit *punteroATextEdit;
public slots:
    void slotBotonDeshacer();
    void slotBotonRehacer();

};
```

```

```

creamos la action y la conectamos

Ejercicio resuelto de selección de colores_:

Enunciado por hacer:

Cambios

Nuevo diálogo en el designer y ficheros DElegircolor.h y .cpp

DElegirColor.h (Ponemos ya dos slots para los botones y un puntero al editor central)

```
#ifndef _DELEGIRCOLOR_H
#define _DELEGIRCOLOR_H

#include "ui_DElegirColor.h"
#include <QTextEdit>

class DElegirColor :  public QDialog, public Ui::DElegirColor {
Q_OBJECT
public:
    DElegirColor( QTextEdit *, QWidget * parent = 0);

    QTextEdit *punteroATextEdit;
public slots:
    void slotBotonElegir();
    void slotBotonProbar();

};
```

```
#endif
```

DElegirColor.cpp

```
#include "DElegirColor.h"
#include <QColorDialog>
#include <QPalette>

DElegirColor::DElegirColor(
    QTextEdit *pEditor, QWidget * parent) : QDialog(parent){
    setupUi(this);

    punteroATextEdit = pEditor;

    connect(botonElegirColor,SIGNAL(clicked()),
            this,SLOT(slotBotonElegir()));
    connect(botonProbar,SIGNAL(clicked()),
            this,SLOT(slotBotonProbar()));
}

void DElegirColor::slotBotonElegir(){}


void DElegirColor::slotBotonProbar(){ }
```

Los pasos principales en Ventana principal están aquí resumidos (sólo para un diálogo):

```
#include "DElegirColor.h"

      void slotDElegirColor();


      QAction * accionElegirColor;

      DElegirColor * dElegirColor;
```

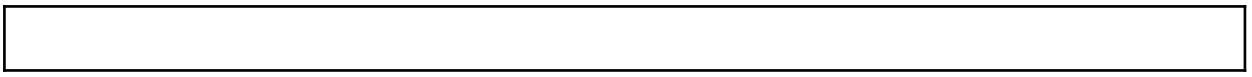
En Constructor y creadores de acciones y menúsVentana Cpp

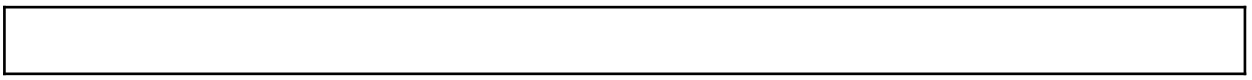
El slot en Ventana principal

```
void VentanaPrincipal::slotDElegirColor(){

    if (dElegirColor == NULL ) {
       dElegirColor = new DElegirColor(editorCentral, this);
    }

    dElegirColor->show();
}
```

Ejercicios ampliación

1. almacenar en unfichero de configuración la visibilidad de los menús, de forma que se pueda especificar que un menú no aparezca visible
2. Mostrar unfichero para ele gir el color de fondo y primer plano





v