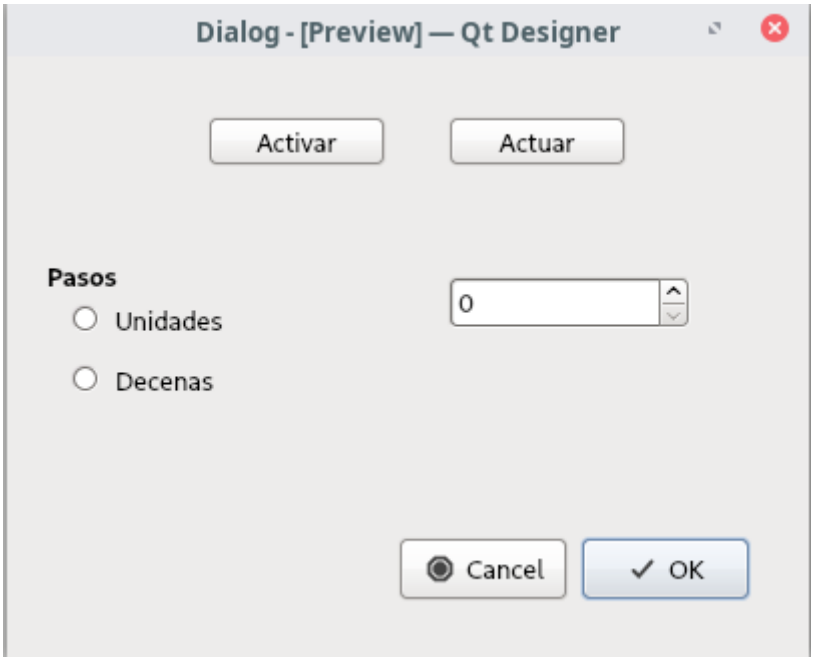
Designer.

Designer es una herramienta que asiste en la creación de interfaces gráficos y simplifica el diseño de los diálogos. Vamos a aprender con un ejemplo como el que puedes ver en esta captura



El resultado del trabajo hecho con el designer es **proporcionar una clase.** Esta clase que obtendremos con el designer, resolverá por sí mismo toda la complejidad de la creación y despliegue de los componentes de interfaz.

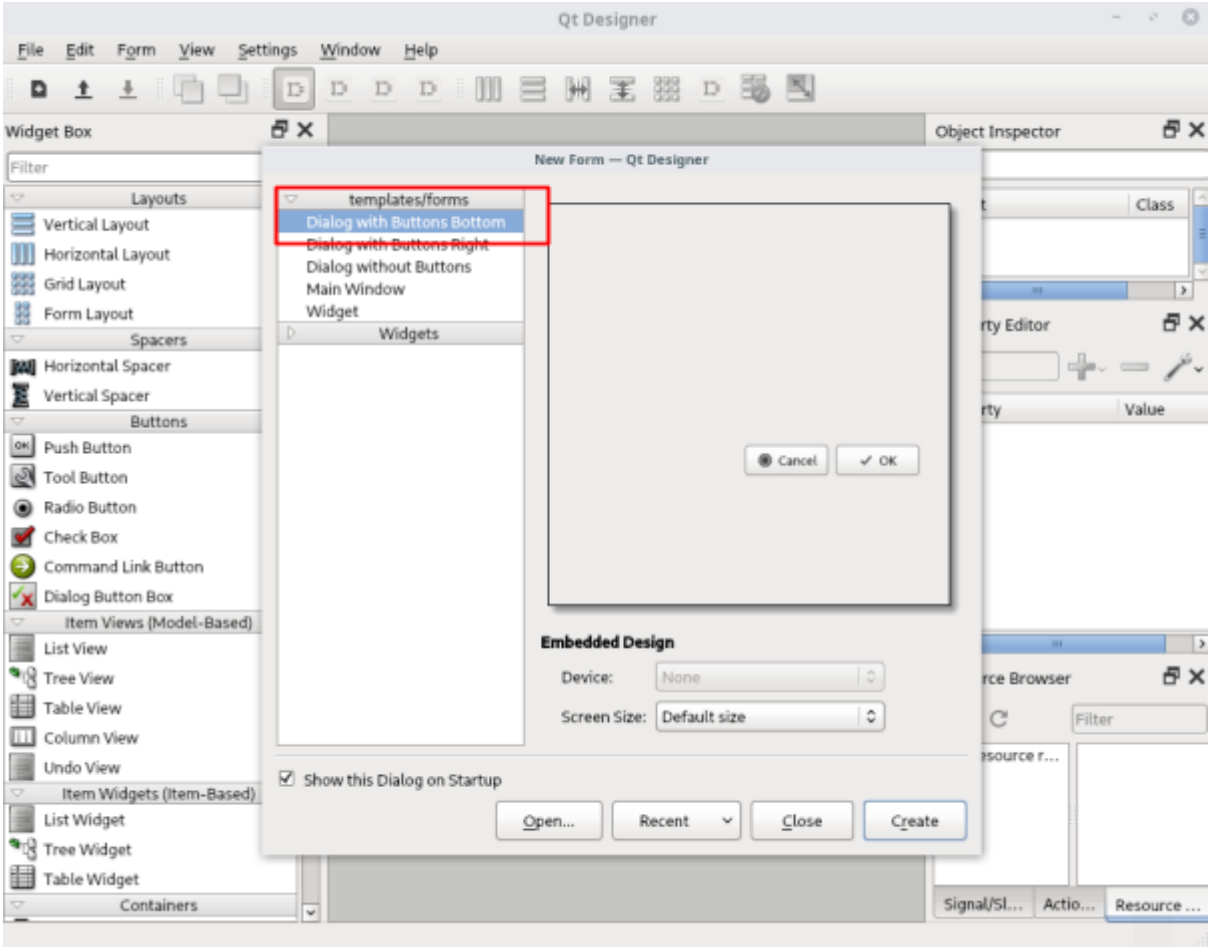
Es decir, nos evita tener que declarar, crear, configurar y colocar los botones, etiquetas, layouts, etc.

Recuerda: usando designer, lo que terminamos obteniendo es una clase de la que heredar y extender.

En este ejemplo, vamos a llamar a la clase: "**`DialogoDesigner`**" y a los ficheros que crearemos **`dialogodesigner.XXX`**. Este nombre nos acompañará todo el ejemplo

Crear y Diseñar un diálogo

Cuando abrimos el designer por primera, vez se nos muestra un diálogo de selección de "tipo de Formulario" que deseamos crear. De todas las opciones elegiremos la de "Dialog with Buttons Bottom" (que será con la que haremos el 73% del curso)

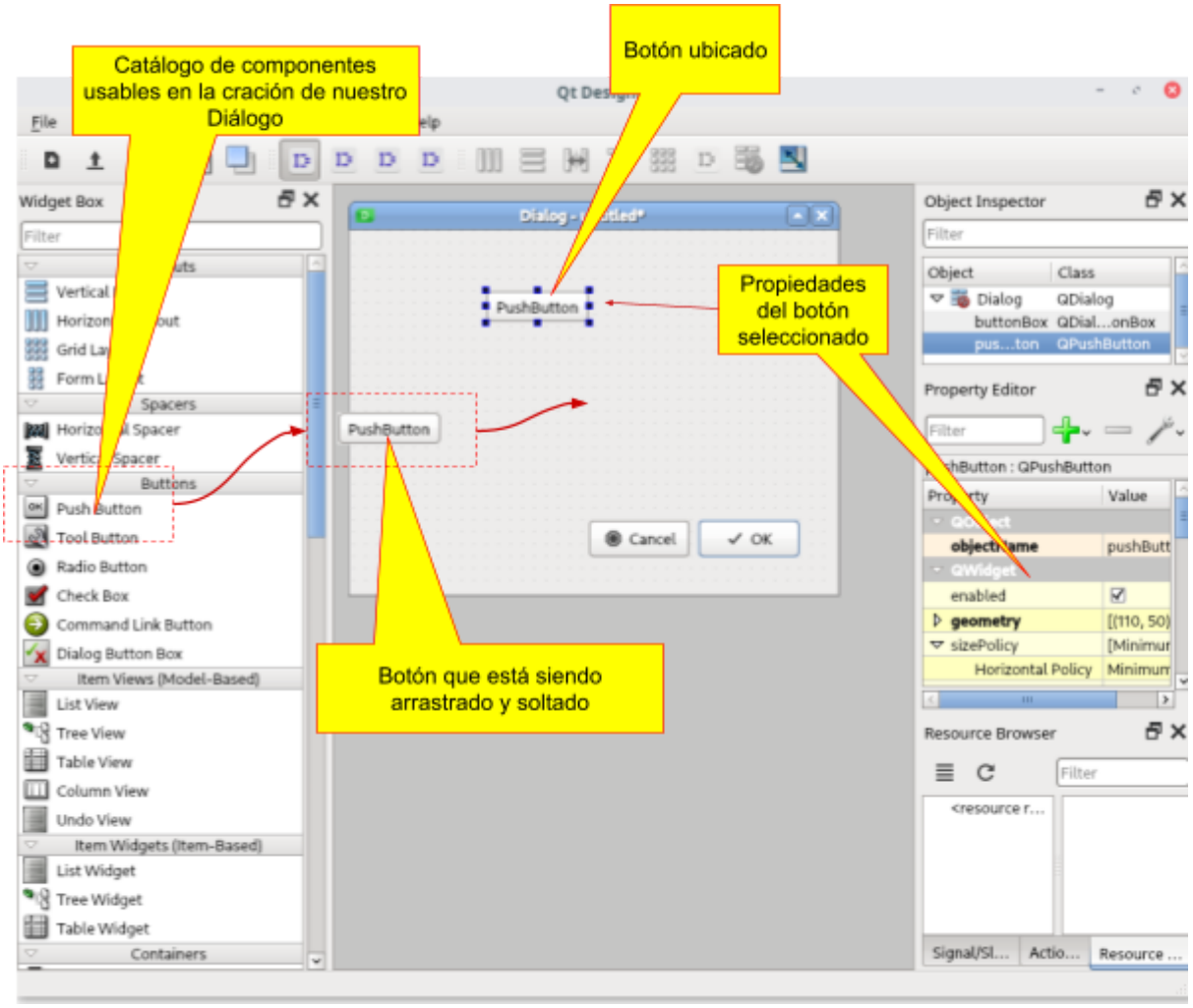


Al elegir "Dialog" estamos indicando, en el fondo, que vamos a hacer un QDialog. Es decir, que heredaremos de QDialog, como hemos hecho durante el curso.

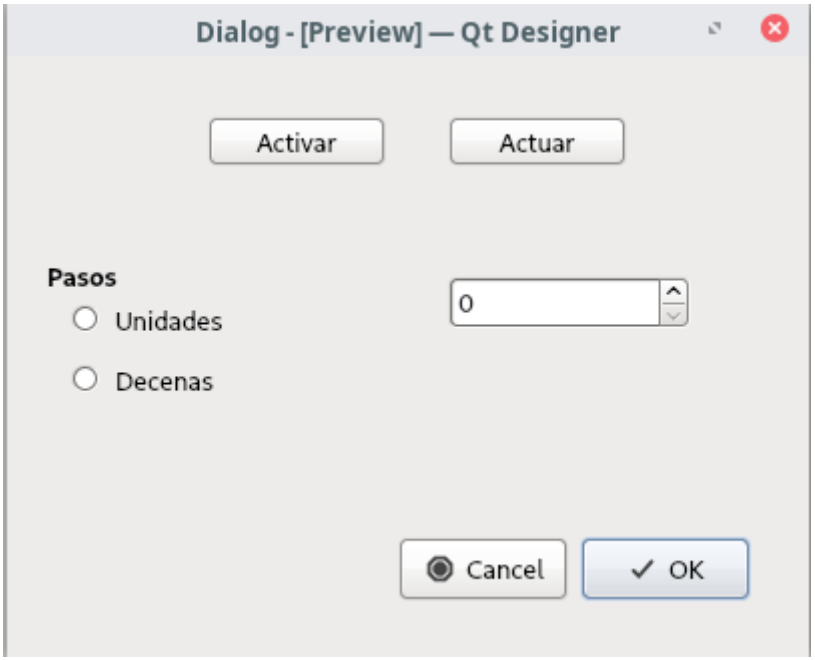
Seguidamente el aspecto de la ventana descubre tres partes importantes. Éstas son:

- Parte central: Aparece **el formulario que creamos** con los componentes que hayamos puesto. Esta será el área de trabajo principal, donde coloquemos nuestro diseño
- Parte izquierda: Un **catálogo de widgets disponibles** para colocar en nuestro Diálogo. Eso se consigue arrastrando el componente deseado y soltándolo encima del Diálogo
- Parte derecha. Cada componente ubicado en el diálogo y el diálogo mismo tienen muchas **propiedades y atributos** que pueden ser configurados en esta parte.

Cuando seleccionamos un elemento, la parte derecha muestra la información de dicho elemento.

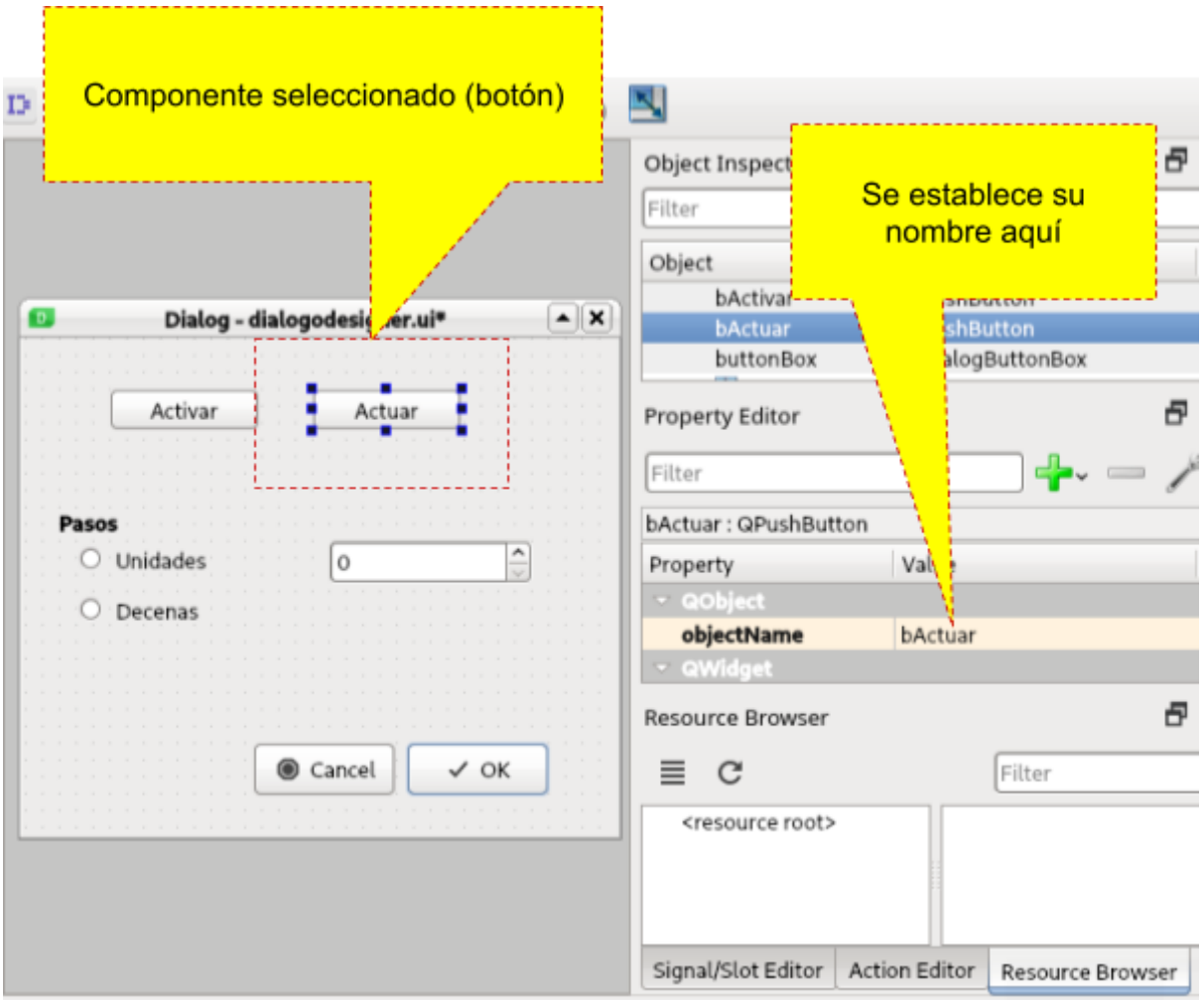


Ahora puedes arrastrar y soltar los componentes del diálogo y completar esta parte para obtener el diseño deseado que se vuelve a mostrar:



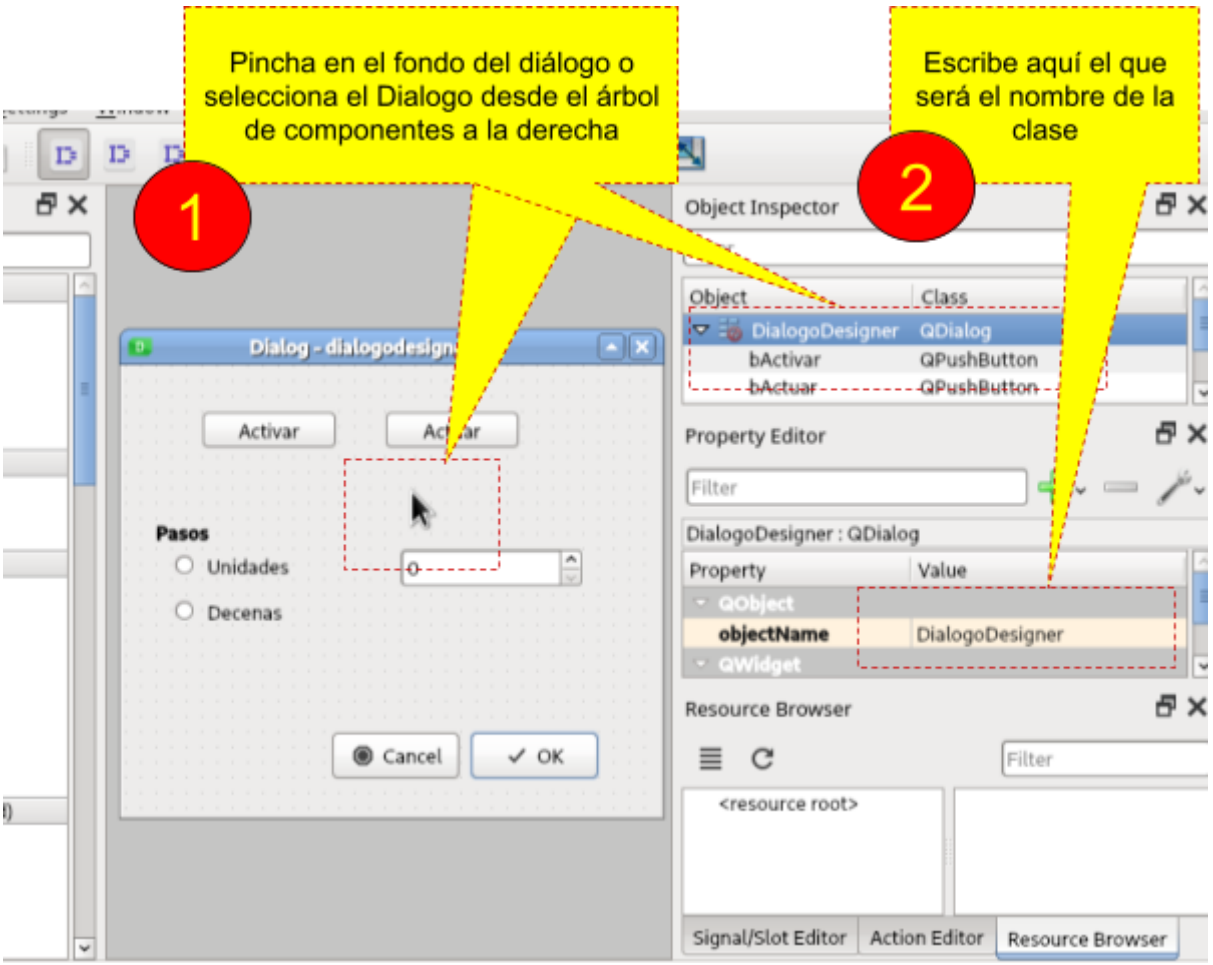
Identificar los componentes

Algunos de los elementos ubicados en el diálogo serán manipulados desde nuestro código más tarde. Pero ¡ El propio diálogo también será manipulado después como una nueva clase ! Por ello, es absolutamente necesario que pongas nombre a todos los elementos que serán usados más tarde. Esto se consigue seleccionando un componente y completando el campo "objectName" como muestra la siguiente captura:



Los nombres de los componentes serán más tarde nombres de atributos de la clase creada.

Establece adecuadamente un nombre para el diálogo tal como se muestra en la siguiente captura



El nombre del **formulario entero** terminará siendo el nombre de la clase creada en nuestro caso `DialogoDesigner.`

El nombre del formulario es el único que termina siendo tratado distinto al resto.

Cuando todo esté nombrado, ya puedes guardar el trabajo de forma normal ("Guardar Como"). Esto generará un fichero con la extensión ".ui". Para evitar confusiones y hacer las cosas como debe ser, limpias y aseadas, llama al fichero que vas guardar con el mismo nombre que la clase (con minúsculas o mayúsculas es tu elección).

En nuestro ejemplo, el fichero a guardar es: `dialogodesigner.ui`

Obtención de la clase

Ya hemos terminado con el designer, y volvemos al terminal

El fichero ".ui" "dialogodesigner.ui" es el punto de partida desde el cual se obtiene una clase en C++ de la que tú heredarás. Para ello, este fichero debe ser tenido en cuenta por el

proceso de compilación de qt (hay que volver a ejecutar una vez: `qmake -project; ... qmake...`). La compilación generará un fichero con extensión .h llamado ui_NOMBREDELFICHEROUI.h . En nuestro caso "**`ui_dialogodesigner.h`**"

> Recuerda, el fichero **`ui_dialogodesigner.h`** debe ser creado automáticamente al compilar

Una vez creado, este fichero .h, puede examinarse para aprender cosas sobre él y sobre Qt, pero no es necesario y en este tutorial no se incluye un detalle completo. Tan sólo debes recordar los siguientes hechos:

- Se ha creado una clase llamada igual que el Formulario. En nuestro ejemplo DialogoDesigner
- Esta clase está dentro de un nombre de espacios (namespace) llamado Ui (tranquilo, esto no tiene mucha importancia)
- La clase creada no hereda de nada. Quizá esperases que heredase de QDialog (que es lo que hemos elegido al principio en el designer)
- La clase creada tiene un método llamado `setupUi()` que será de suma importancia para nosotros (pero muy fácil de usar)
- Este fichero creado NO debe ser modificado.

Aprovechamiento de la clase.

La idea clave es :

> Crear una **nueva clase** y **heredar** de la que obtenemos con el fichero ui_dialogodesigner.h

Si quieres, puedes familiarizarte con algunas partes del fichero ui_dialogodesigner.h creado y que ayudan a entender lo que sigue después: Fíjate lo que ocurre en este fichero:

- se declara una clase `Ui_DialogoDesigner`, y después se declara otra (la que importa) más abajo llamada `Ui::DialogoDesigner` (esto está remarcado así)
- Se declaran lso componentes puestos en el designer (están parcialmente borrado por legibilidad) (remarcado)
- Se crea un método llamado **`setupUi(QDialog *)`** que está parcialmente borrado por legibilidad, en él se crean todos los componentes y se disponen arreglo al diseño hecho en el designer (remarcado)

```
#ifndef UI_DIALOGODESIGNER_H
#define UI_DIALOGODESIGNER_H

#include <QtCore/QVariant>
...

QT_BEGIN_NAMESPACE

class Ui_DialogoDesigner
{
public:
      QDialogButtonBox *buttonBox;
      QComboBox *cmbMateriales;
      QGroupBox *groupBox;
      ...

      void setupUi(QDialog *DialogoDesigner)
      {
      if (DialogoDesigner->objectName().isEmpty())

DialogoDesigner->setObjectName(QString::fromUtf8("DialogoDesigner"
));
      DialogoDesigner->resize(400, 300);
      buttonBox = new QDialogButtonBox(DialogoDesigner);
      buttonBox->setObjectName(QString::fromUtf8("buttonBox"));
      buttonBox->setGeometry(QRect(30, 240, 341, 32));
      buttonBox->setOrientation(Qt::Horizontal);
      butto...
      } // setupUi

      void retranslateUi(QDialog *DialogoDesigner)
      {

DialogoDesigner->setWindowTitle(QCoreApplication::translate("Dialo
goDesigner", "Dialog", nullptr));
      cmbMateriales->setItemText(0,
QCoreApplication::translate("DialogoDesigner", "Madera",
nullptr));
      cmbMateriales->setItemText(1,
QCoreApplication::translate("DialogoDesigner", "Metal", nullptr));
      ...
```

```
      } // retranslateUi

};

namespace Ui {
      class DialogoDesigner: public Ui_DialogoDesigner {};
} // namespace Ui

QT_END_NAMESPACE

#endif // UI_DIALOGODESIGNER_H
```

Este fichero anterior NO puedes modificarlo, pero no viene mal haberlo visto y revisarlo en cualquier momento si algo no te funciona para ver qué nombres estás dando a cada cosa. (Nota ; Durante el examen la cagarás porque olvidarás ponerle nombre a algo y al final venir a este fichero puede hacerte ver dónde la has cagado y salvarte el culo)

Ahora **vamos ya a crear nuestra clase definitiva**. Crearemos dos archivos de cabecera .h y código .cpp igual que ya hemos hecho para otros diálogos cuando no usábamos el designer..

Para seguir manteniendo el orden y aseo necesarios para no cagarla en el futuro (o en medio del examen), dale el mismo nombre de la clase DialogoDesigner y también al fichero que vas a crear: **`dialogodesigner.cpp`** y **`dialogodesigner.h`**

Empecemos por el fichero de cabecera `dialogodesigner.h`, (el siguiente ejemplo, supone la base de cualquier actividad futura al usar designer.)


```
#ifndef DIALOGODESIGNER_H
#define DIALOGODESIGNER_H
#include <QDialog>
#include "ui_dialogodesigner.h"

class DialogoDesigner : public QDialog, public Ui::DialogoDesigner

{
Q_OBJECT

public:
    DialogoDesigner(QWidget *parent = NULL);

};
```

```
#endif
```

Cosas a observar del fichero anterior:

- Estás creando una clase llamada **`DialogoDesigner`**. Es el mismo nombre que usaste en el designer para el formulario.
- Esta clase va a ser un diálogo, por ello **heredamos de `QDialog`**
- Esta clase va a aprovechar lo hecho desde el designer: Por ello **hereda de `Ui::DialogoDesigner`**
- El "`Ui::`" que aparece antes de "`DialogoDesigner`" viene determinado por el namespace visto en `dialogodesigner.h`. Esto no es importante ahora
- Creamos un constructor como siempre, y seguimos imitando el constructor de la clase `QDialog`

Heredar de dos clases es algo posible en C++ y útil en casos como éste. Imagina que alguien inventa un objeto que hereda del árbol del café y de una cafetera a la vez. ¿Qué obtendríamos? Pues algo así como una planta que saca café bebible caliente. O por ejemplo ¿Qué es un político en España? Una cosa que ha heredado de un vampiro y de una persona que habla convincentemente, aprovecha las dos características heredadas.

Nuestra clase es un QDialog y es también aquello que se obtenga desde la clase producida desde el Designer. Combina características de ambas y eso es lo que podremos aprovechar ahora a continuación.

Veámos el código del constructor, que es donde está la magia.

dialogodesigner.cpp:

```
#include "dialogodesigner.h"

DialogoDesigner::DialogoDesigner(QWidget * parent) :
QDialog(parent) {
    setupUi(this);

}
```

El constructor hace una llamada a `setupUi(this)`, que es el método heredado desde el designer, y le pasa ¡¡¡ this !!!. ¿Qué cojones pasa ahí?

Recuerda, esta clase que estás creando es un QDialog, y el método requiere que le pases un QDialog, por ello , ME PASO A MÍ MISMO COMO PARÁMETRO DE ESTE MÉTODO MÍO porque yo soy del tipo del argumento requerido por mi propio método!!

Es como si una cafetera la pudieses conectar a un árbol que produce café, y en nuestro engendro que combina cafetera y árbol del café, ese engendro se conectase a sí mismo para producir café

¿¡Será Gran Aixó?!

Este método, como hemos visto, crea todos los componentes y los pone arreglo al diseño de designer. y el resultado es que tenemos la posibilidad ahora de hacer cualquier cosa con los componentes, pero visibles y en su sitio ya están.

Aviso futuro: ¿Sabes? La vas a cagar muchas veces, y la vas a cagar porque no pondrás en el constructor la llamada a `setupUi(this);` en la puñetera primera línea de todas. Este método CREA los componentes (sí, ¡ con `new` !). Si se te ocurre usar los componentes (algún QPushButton por ejemplo) antes de crearlo, el programa fallará porque los punteros declarados en ui_dialogodesigner.h no están inicializados, porque se te ha olvidado poner `setupUi(this);` . Cada vez que la cagas así, Lucifer mata a un gatito y se lo lleva a sufrir al infierno.

Tú no quieres que los gatitos sufran , pues entonces, pon `setupUi(this)` en la primera línea del constructor

Ahora, la forma de dar comportamiento al diálogo puede ser ya creando conexiones señales y slots, crear nuevos métodos, etc. etc. etc. Por ejemplo, observa cómo hacemos una conexión de señales y slots entre los botones creados

```
#include "dialogodesigner.h"

DialogoDesigner::DialogoDesigner(QWidget * parent) :
QDialog(parent) {
    setupUi(this);

    connect(bActivar,SIGNAL(clicked()),
            bVerificar,SLOT(hide()));

    connect(rBpresion,SIGNAL(clicked(bool)),
            bVerificar,SLOT(setHidden(bool)));
    connect(rBTemperatura,SIGNAL(clicked(bool)),
            bVerificar,SLOT(show()));

}
```

Y por supuesto, mostrar el diálogo no tiene secreto, mira main.cpp:

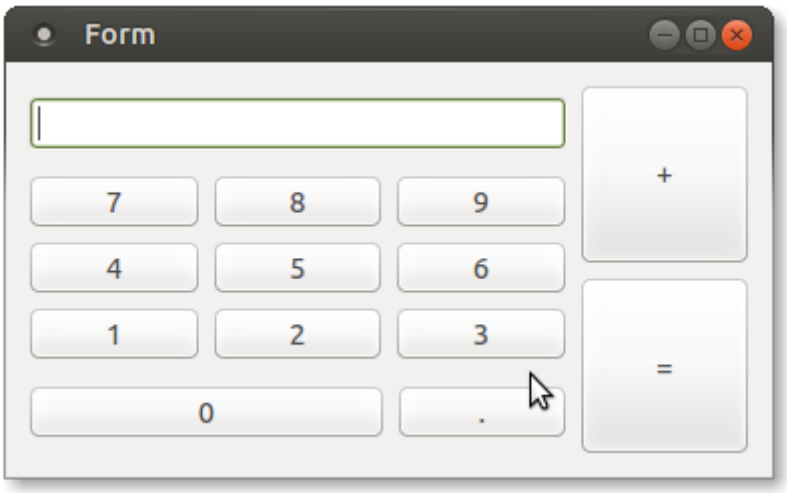
```
#include <QApplication>
#include "dialogodesigner.h"


 int main(int argc, char *argv[])
 {
      QApplication app(argc, argv);
         DialogoDesigner * dialogoDesigner = new DialogoDesigner();
         dialogoDesigner->show();

      return app.exec();
 }
```

En este punto podrías hacer tus propios slots con tu propio código y conectarlos a cualquier botón u otro elemento para adaptar el comportamiento del diálogo a lo que se pida. Esto ya lo has hecho antes.

Calculadora

Objetivo. Diseñar la funcionalidad básica de una calculadora como la siguiente.

Este ejercicio está poco explicado en su mayor parte, pero tiene una novedad que sí se explica con más profundidad (que es bastante complicada y puede postponerse en el curso). Sirve de práctica y de aprendizaje.

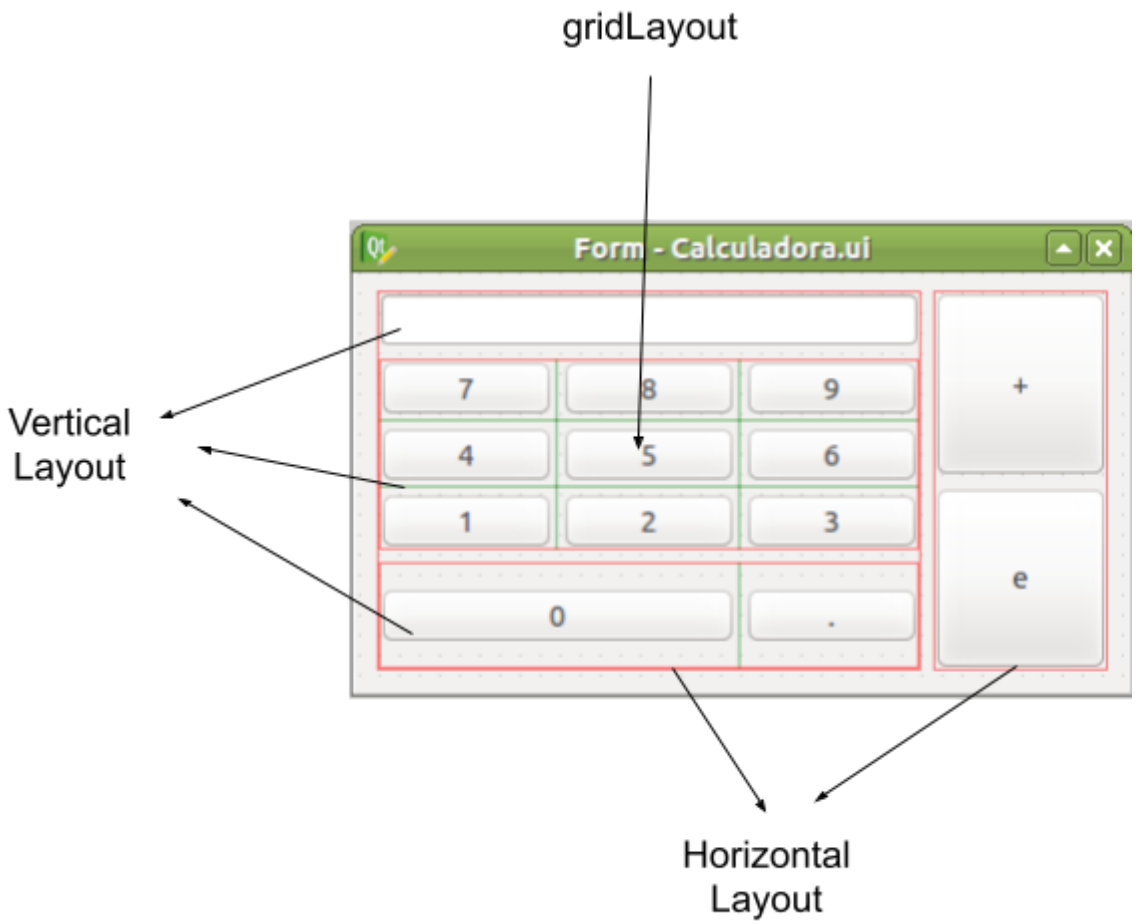


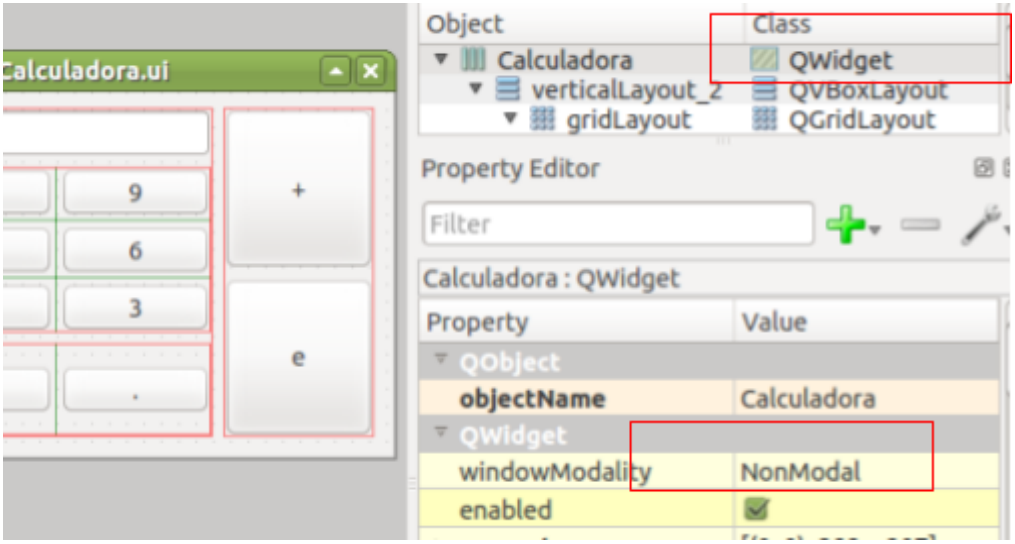
main.cpp no cambia:

```
#include <QApplication>
#include "calculadora.h"
```

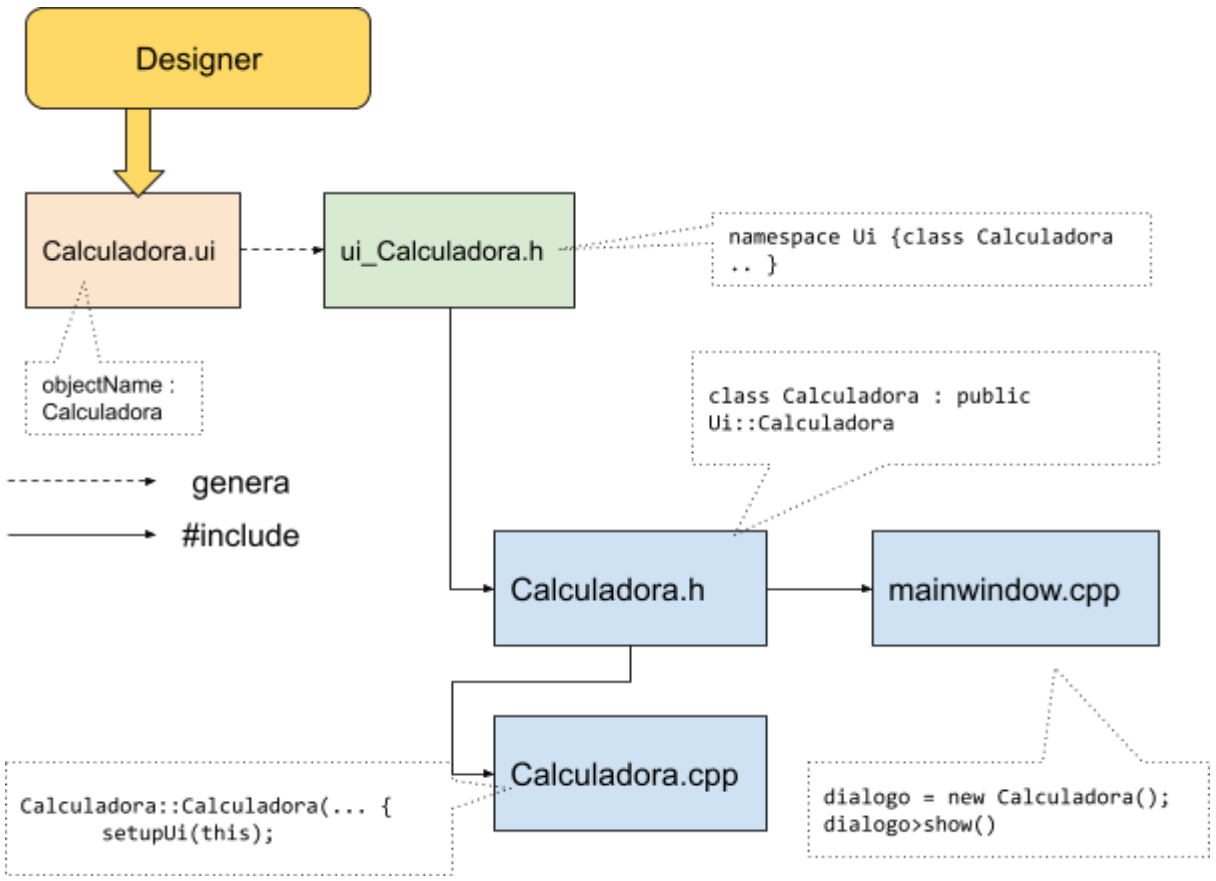
```
int main(int argc, char *argv[])
{
     QApplication app(argc, argv);
        Calculadora * calculadora = new Calculadora();
        calculadora->show();

     return app.exec();
}
```

La disposición de los elementos viene marcada por los siguientes layouts

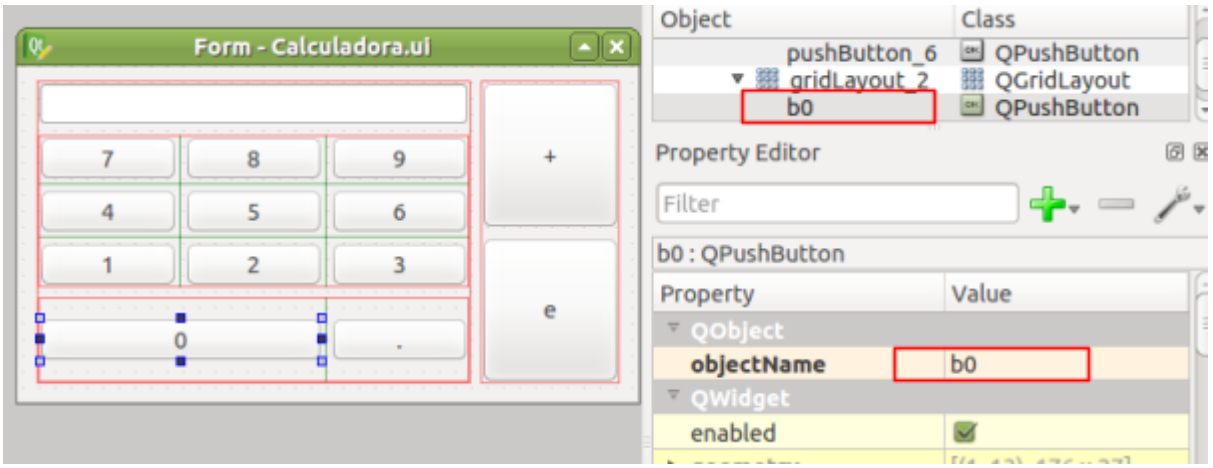




Repasemos el trabajo a hacer y los ficheros implicados:



Hay que nombrar los botones, todos, pero concretamente los numéricos vamos a hacerlos así: "b0", "b1", etc. Esto será útil después.



Siendo así, en el fichero ui_calculadora.h, el botón será declarado de la siguiente manera:

```
    QPushButton *b0;
```

El "display" puede ser un QlineEdit como el ejemplo, o un QLCDisplay, que es más chulo y admite establecer directamente enteros

Declaración de la clase

```
#ifndef CALCULADORA_H
#define CALCULADORA_H

#include <QDialog>
#include "ui_Calculadora.h"


class Calculadora : public QWidget, public Ui::Calculadora {
Q_OBJECT
public:
    Calculadora(QWidget *parent = 0);
private slots:
    void ponCero();
    void ponUno();
};

#endif
```

Implementación de la clase:

```
#include "calculadora.h"

Calculadora::Calculadora(QWidget * parent) : QWidget(parent){

    setupUi(this);
    connect(b0,SIGNAL(clicked()),this,SLOT(ponCero()));
    connect(b1,SIGNAL(clicked()),this,SLOT(ponUno()));
}


void Calculadora::ponCero(){
    display->setText(display->text() + "0");
}


void Calculadora::ponUno(){
    display->setText(display->text() + "1");
}
```

main.cp

```
#include <QApplication>

#include "calculadora.h"

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

Actuando ante un botón

Vamos a tomar una aproximaciòn más elegante para dar comportamiento a todos los botones que tienen un número. En vez de hacer un slot por cada botón, vamos a hacer

un slot para todos los botones.

Empezaremos actuando ante un botón, declaramos en la clase un nuevo slot

calculadora.h

```
public slots:
	void slotDigitos();
```

implemento algo mínimamente, después ampliaré:

```
void Calculadora::slotDigitos(){
	display->setText(display->text() + QString("1"));
}
```

conecto para probar al botón 1

```
connect(b1,SIGNAL(clicked()),this,SLOT(slotDigitos()));
```

Funciona sólo el botón 1, ¡¡ Evidentemente !!

Demasiado rollo hacer muchos slots, vamos a hacer sólo uno y que todos los botones estén conectados a él

```
connect(b1,SIGNAL(clicked()),this,SLOT(slotDigitos()));
connect(b2,SIGNAL(clicked()),this,SLOT(slotDigitos()));
connect(b3,SIGNAL(clicked()),this,SLOT(slotDigitos()));
```

¿Pillas la intención de las anteriores líneas? Todos los botones redigirán al mismo slot.

El efecto es el mismo para todos los botones. Parece algo estúpido, hasta que nos planteamos una cuestión ¿Es posible saber **dentro del slot** qué botón ha sido pulsado y es el culpable de haber lanzado la señal?

¡Sí! Existe una forma de preguntar a Qt quién ha sido el objeto que ha activado una señal por la que nos estamos ejecutando (como slot que somos)

primero, explicar herencia y problema dle programador del método sender() en determinar el día qu elo programó qué devolver.


```
void Calculadora::slotDigitos(){
	QObject *culpable = sender();
```

ejemplo, llamada que deja un número de teléfono pero no deja quién, usas el número para buscar en la agenda quién es el que ha llamado. aquí tienes la dirección del culpable y una "lista" de posibles culpables (b1,b2, b3)


```
void Calculadora::slotDigitos(){

	QObject *culpable = sender();

	if (culpable == b1 )
	display->setText(display->text() + QString("1"));
```

```
if (culpable == b2 )
display->setText(display->text() + QString("2"));

if (culpable == b3 )
display->setText(display->text() + QString("3"));
```

La conclusión importante -repetimos-, es que todos los botones comparte el mismo slot y por ello hemos escrito la siguiente conexión señales-slots:

```
connect(b1,SIGNAL(clicked()),this,SLOT(slotDigitos()));
connect(b2,SIGNAL(clicked()),this,SLOT(slotDigitos()));
connect(b3,SIGNAL(clicked()),this,SLOT(slotDigitos()));
```

Usando el texto de los botones para acortar el método.

El código anterior funciona, pero hay que hacer lo mismo para cada botón y sólo cambia el texto que se añade al display, que ¡Además! es el que muestra el propio botón. Este texto puede recuperarse con la propiedad `text()` de `QAbstractButton`. Podríamos concebir algo como ésto:

```
void Calculadora::slotDigitos(){
      QObject *culpable = sender();
      display->setText(display->text() + culpable->text());
      ...
```

Sin embargo, al compilar aparece un error desalentador pero claro

```
calculadora.cpp: In member function ‘void Calculadora::slotDigitos()’:
calculadora.cpp:14:50: error: ‘class QObject’ has no member named ‘text’
      display->setText(display->text() + culpable->text());
```

`sender` devuelve un QObject, que no tiene implementado ningún método text(),...

El compilador, no traga. porque para él, lo importante no es que haya sido un botón el que ha lanzado la señal, sino que la variable "culpable" es de tipo `QObject *` . Y ese tipo no tiene el método `text()`

Vamos a intentar lo siguiente (recoger un QpushButton) :

```
void Calculadora::slotDigitos(){
      QPushButton *culpable = sender();
      display->setText(display->text() + culpable->text());
```

Compilamos y da otro error:

	salida del compilador

```
calculadora.cpp: In member function ‘void Calculadora::slotDigitos()’:

calculadora.cpp:12:35: error: invalid conversion from ‘QObject*’ to
‘QPushButton*’ [-fpermissive]

      QPushButton *culpable = sender();
```

^

conversión inválida. "Todo coche es vehículo, pero todo vehículo no es coche".

El compilador no se fía de que lo que devuelve sender() ("QObject") pueda ser siempre un QPushButton. Al revés sí que habría transigido porque un QPushButton siempre es también un QObject

Hay que "pasar por encima del compilador" y forzar la conversión dado que sabemos que el valor retornado ahí por sender() va a ser un QPushButton en realidad (ya que los botones son los únicos que pueden activar el slot)

En **C** podíamos "prototipar" así:

```
void Calculadora::slotDigitos(){

      QPushButton *culpable = (QPushButton*) sender() );
      display->setText(display->text() + culpable->text());
```

con **C++** se añade un par de opciones más recomendables:

```
QPushButton *culpable = static_cast<QPushButton*>( sender() ) ;
QPushButton *culpable = dynamic_cast<QPushButton*>( sender() ) ;
```

con **Qt** disponemos de otra opción en exclusiva para QObject y descendientes :

```
QPushButton *culpable = qobject_cast<QPushButton*>( sender() ) ;
```

Ésta línea es la que finalmente hace el trabajo rápido, "culpable" ahora es de tipo QPushButton*

```
display->setText(display->text() + culpable->text());
```

Anexo Calculadora OPCIONAL

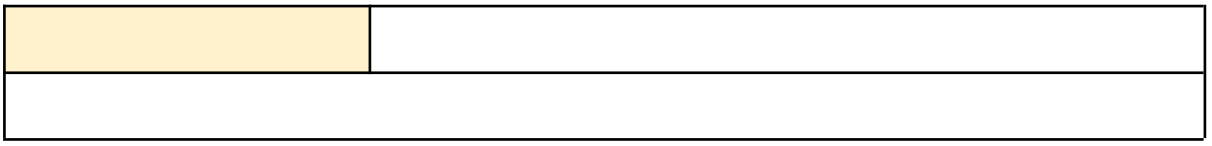
Hemos conectado las señales de todos los botones al mismo slot de la siguiente manera:

```
connect(b1,SIGNAL(clicked()),this,SLOT(slotDigitos()));
connect(b2,SIGNAL(clicked()),this,SLOT(slotDigitos()));
connect(b3,SIGNAL(clicked()),this,SLOT(slotDigitos()));
```

Pero para evitar que ésto sean demasiadas líneas cuando hay varios botones, podemos recuperar todos los botones de la ventana e iterar sobre ellos de la siguiente manera

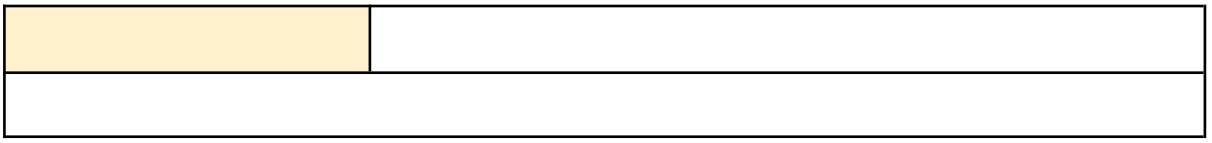
```
QList<QPushButton*> lista= this->findChildren<QPushButton*>();

     foreach (QPushButton * boton, lista ) {
          connect(boton,SIGNAL(clicked()),
                  this, SLOT(slotBoton()));
     }
```

Sin embargo, el código anterior, establecería conexiones de todos los botones al slotBoton(), incluso de aquellos botones que no deseamos esa conexión, La solución pasa por poner a los botones en un Layout, y modificar el bucle para seleccionar sólo los que pertenezcan al layout:



```
foreach(QWidget *w ,
   layoutDigitos->parentWidget()->
   findChildren<QPushButton*>() )
      {
        if (layoutDigitos->indexOf(w) >= 0){
            QPushButton *b = qobject_cast<QPushButton*>(w);
            connect(  b,    SIGNAL(clicked()),
                      this, SLOT(slotDigito()));
            }

      }
```

Mejor usando esta manera:

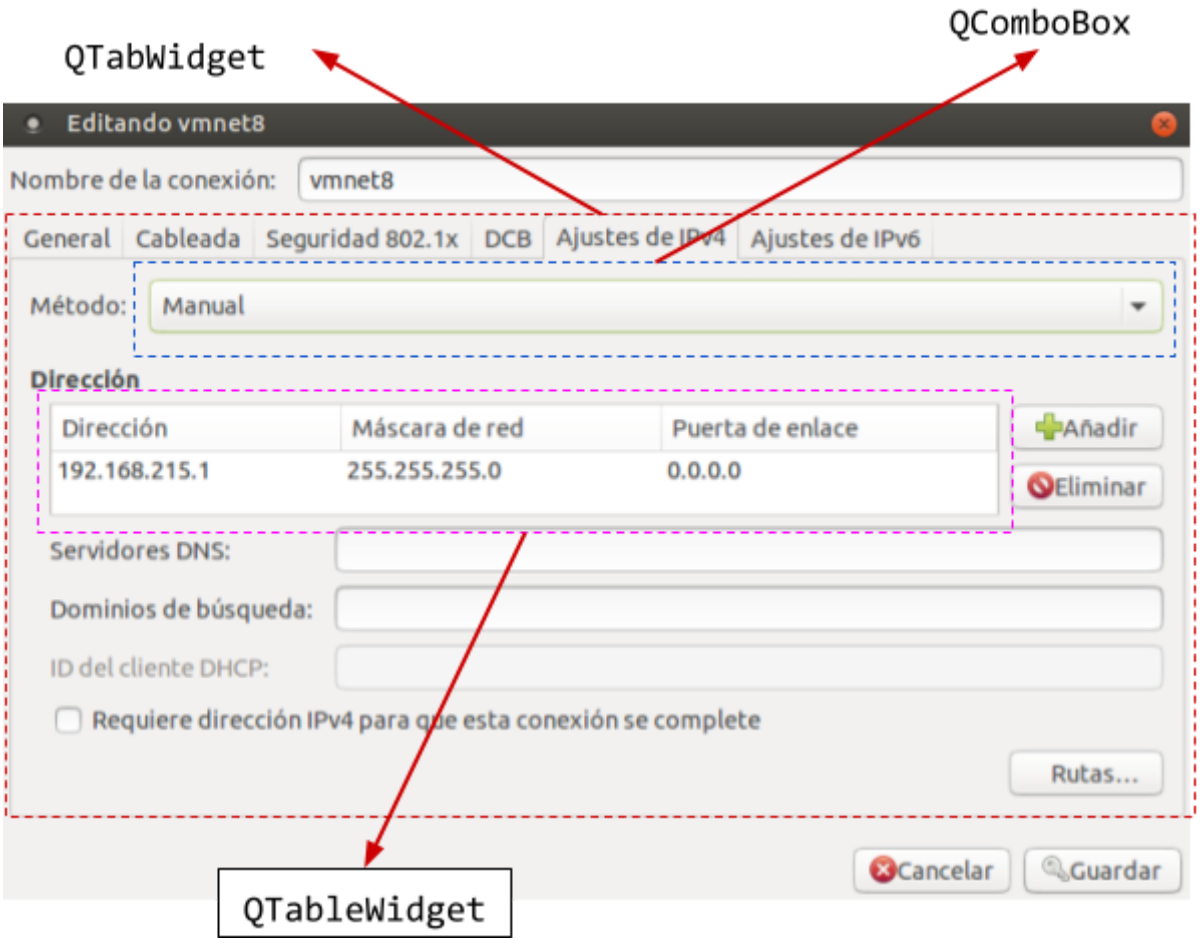


```
 for (int i=0; i < gridDigitos->count(); i++){
   QLayoutItem * item = layoutDigitos->itemAt(i);
   QWidget *widgetBoton = item->widget();
   QPushButton *boton ;
   boton = qobject_cast<QPushButton*>(widgetBoton);
   boton->installEventFilter(this);
   connect(boton, SIGNAL(clicked()),
           this,       SLOT(slotDigito()));

 }
```

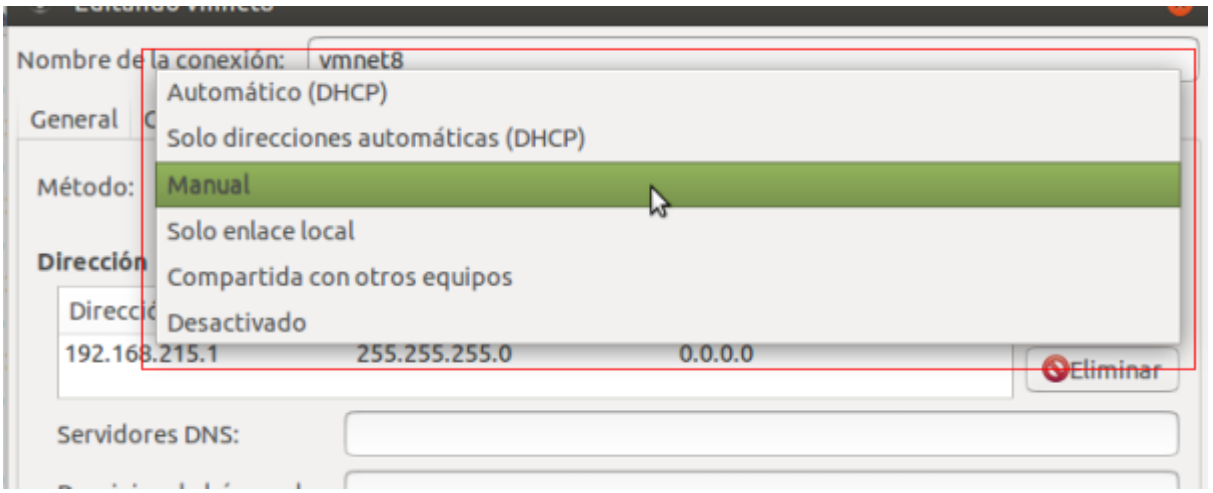
Diálogo Red

Ejercicio sólo de Designer

Diseñar un formulario como el siguiente

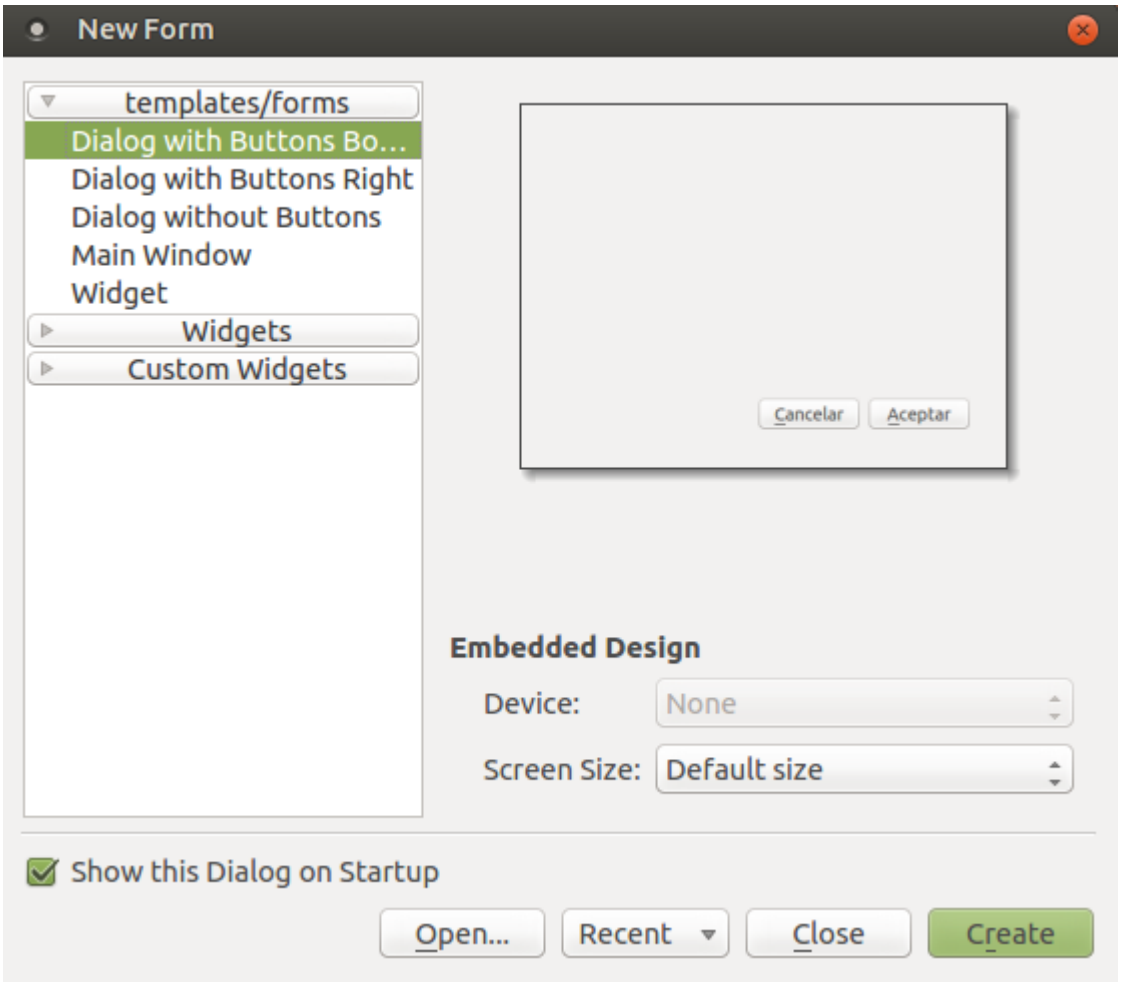


Un QComboBox es una lista desplegable que muestra una serie de entradas o cadenas de texto cuando se despliega con el ratón, como en la siguiente imagen:

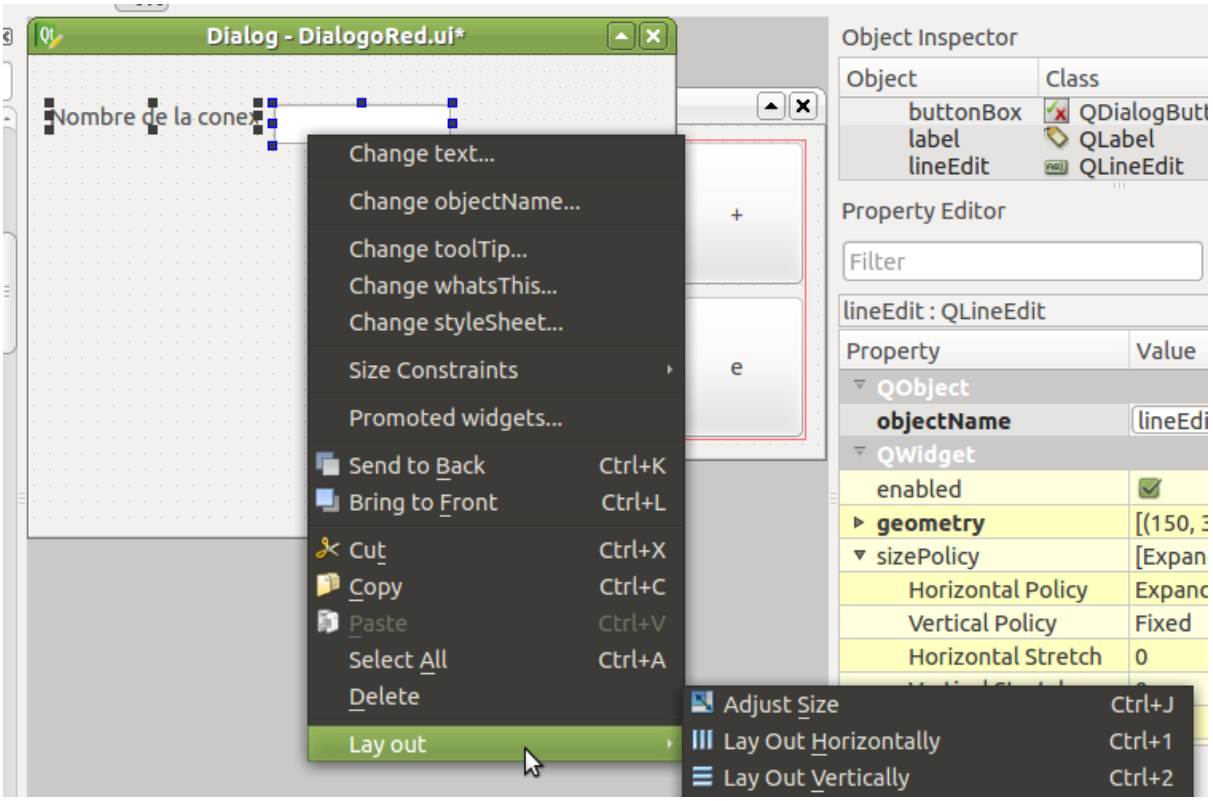


Empezamos.

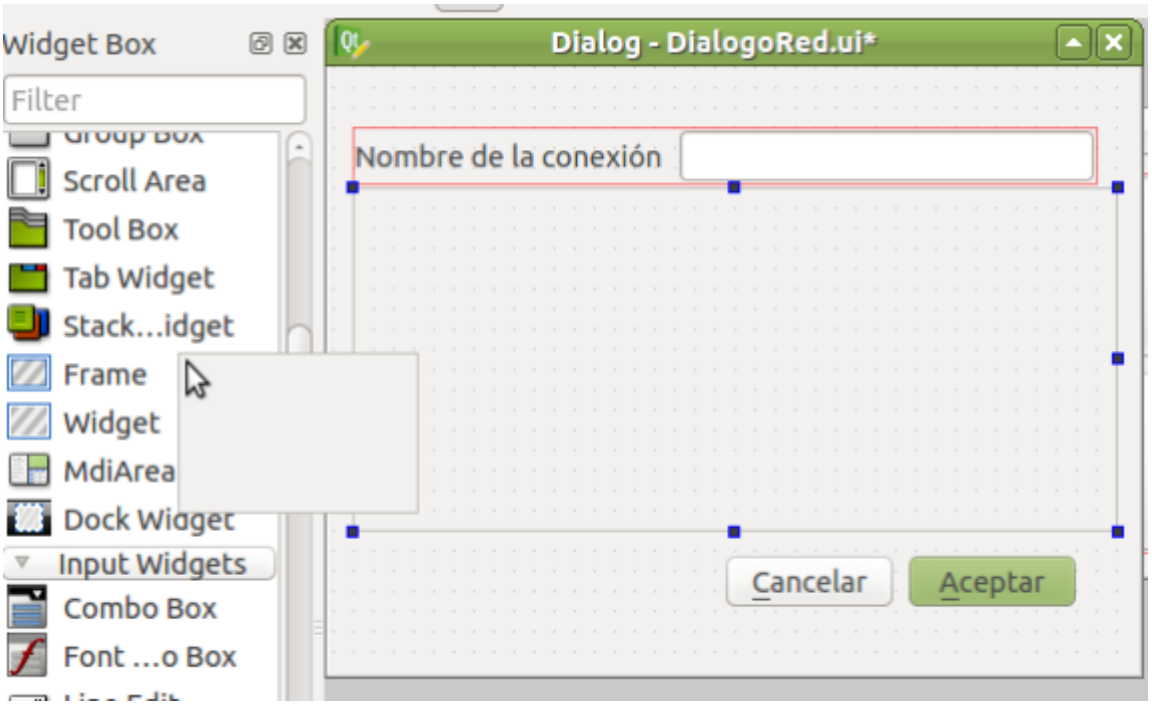
Es un Diálogo con botones en la parte inferior.



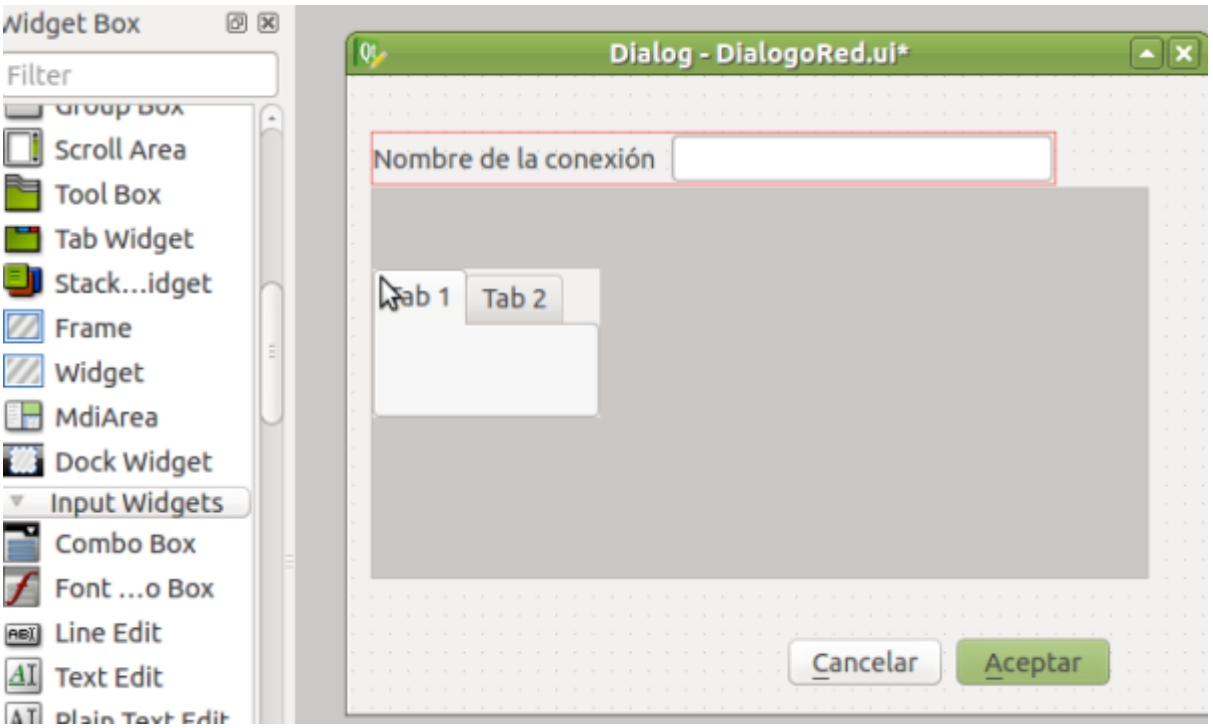
Ponemos los componentes superiores (QLabel y QLineEdit), y los metemos dentro de un layout Horizontal.



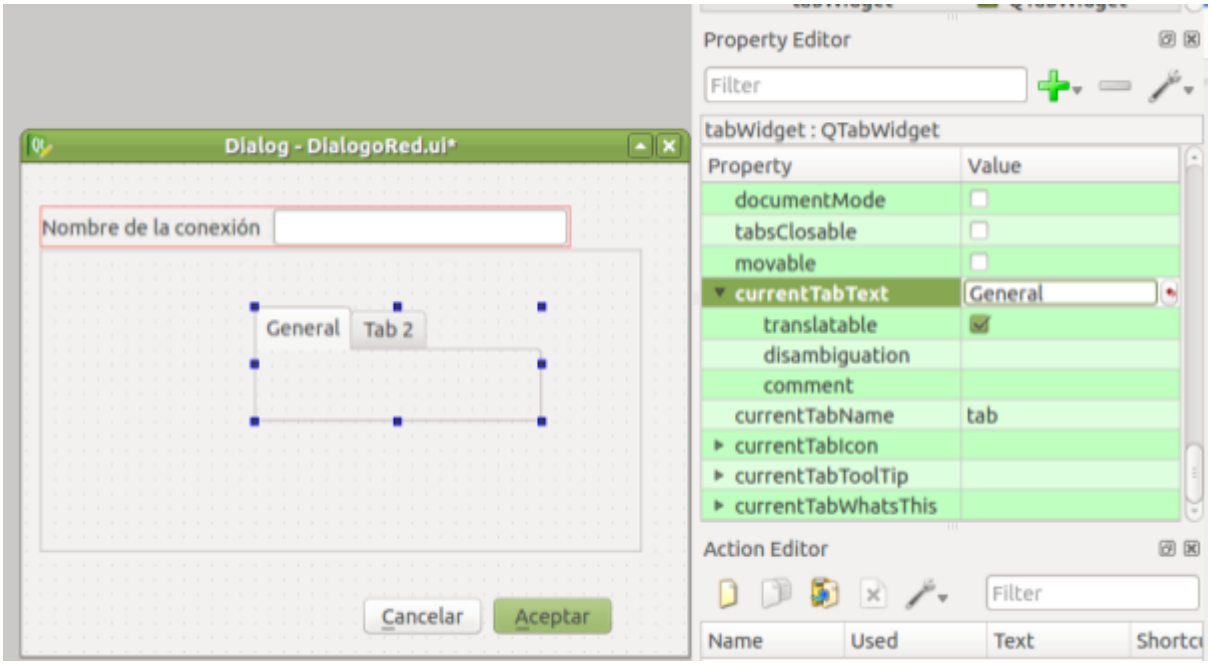
Podríamos seguir colocando widgets, pero la ventana a imitar parece tener un marco en la parte central de la misma, vamos a imitarlo usando para ello el componente QFrame.



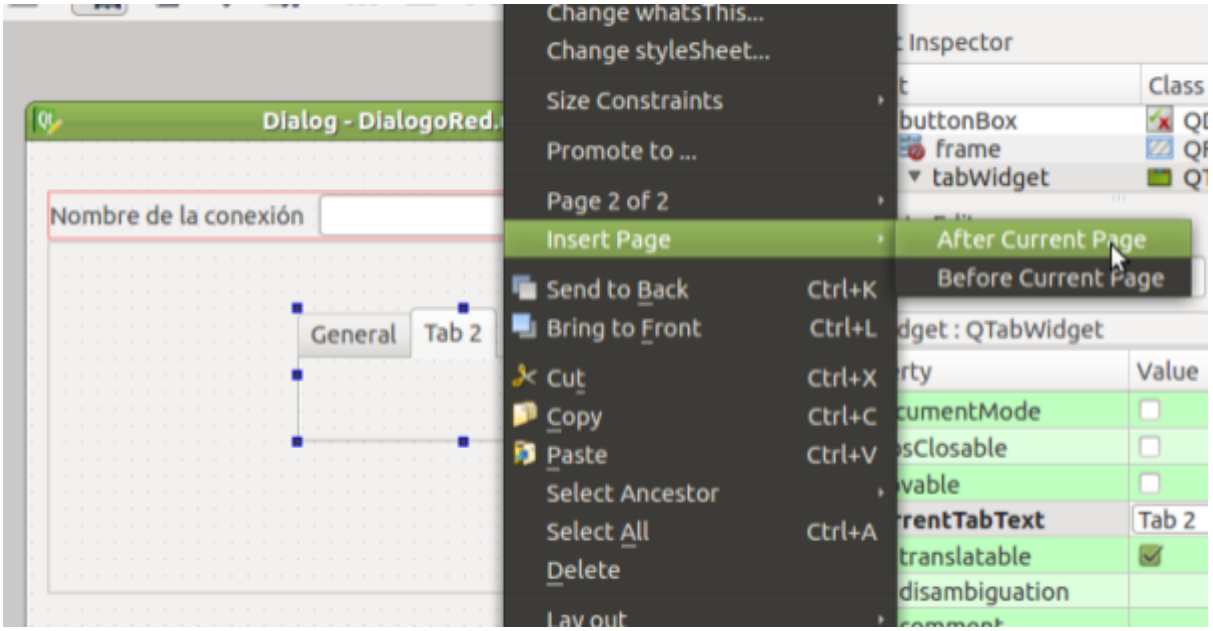
Ponemos el tabWidget



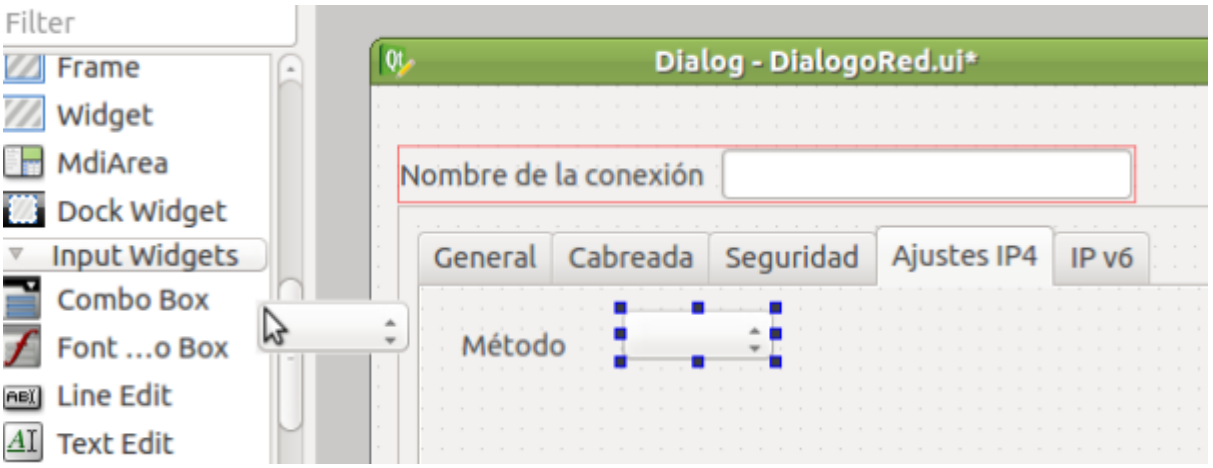
Ahora vamos a cambiar el nombre de una pestaña (o tab). Para ello, has de modificar la propiedad CurrentTabText del componente QTabWidget, cambiando entre las diferentes pestañas que existen (de momento sólo hay dos, que establece por defecto)



Añadimos una página y las que haga falta pinchando en el menú contextual.

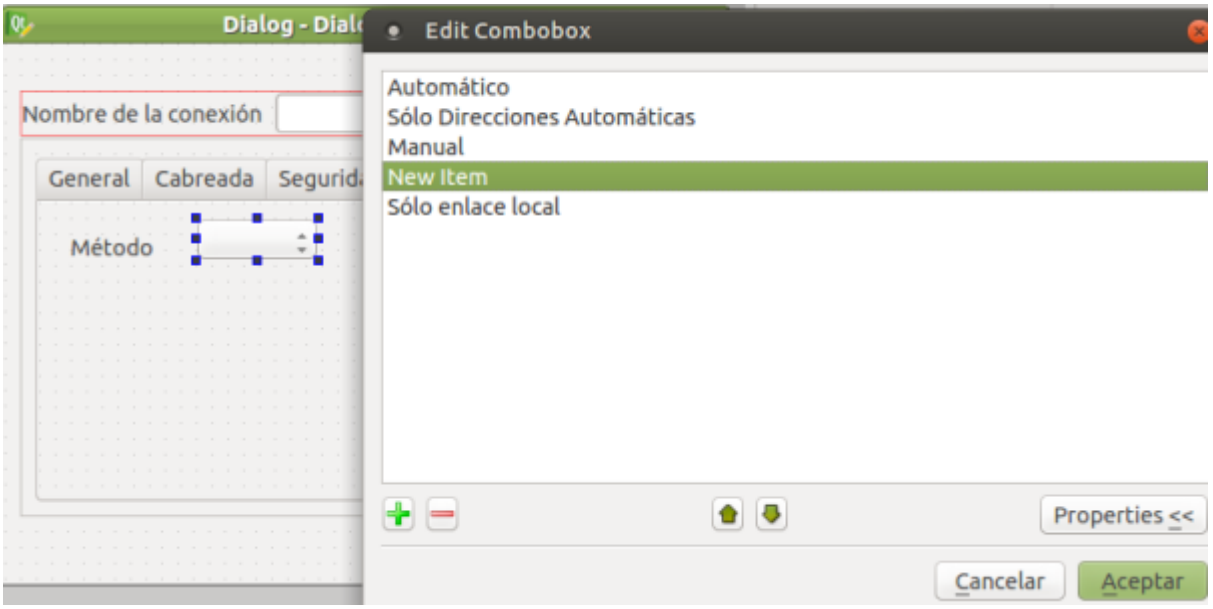


Con el QTabWidget completo, elegimos el de la pestaña que vamos a rellenar, y empezamos poniendo un QLabel y un QComboBox

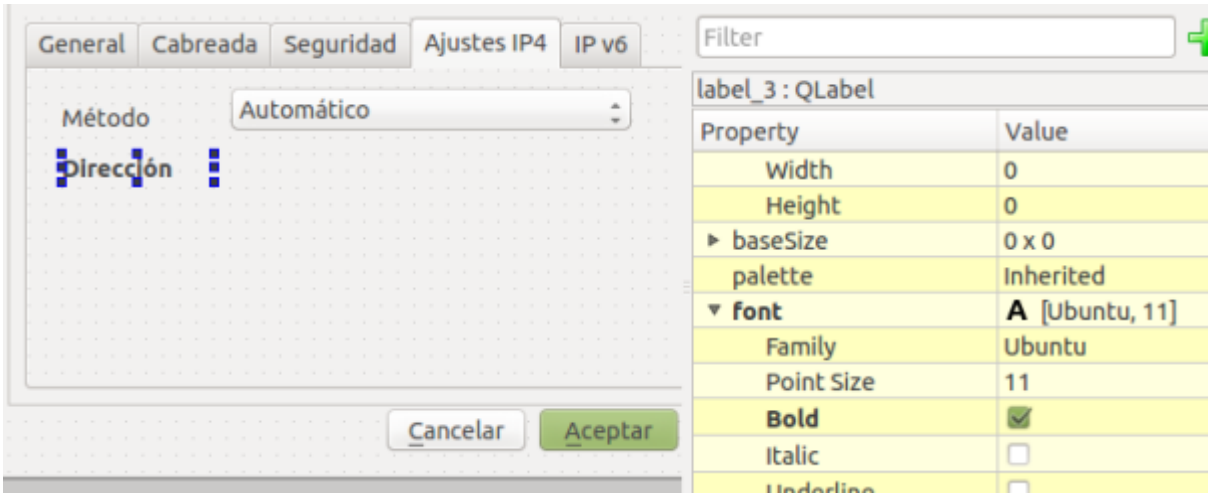


El comboBox puede tener varios valores y generalmente éstos se establecen durante el funcionamiento del programa. Pero en este caso, dado que sabemos cuáles son los textos

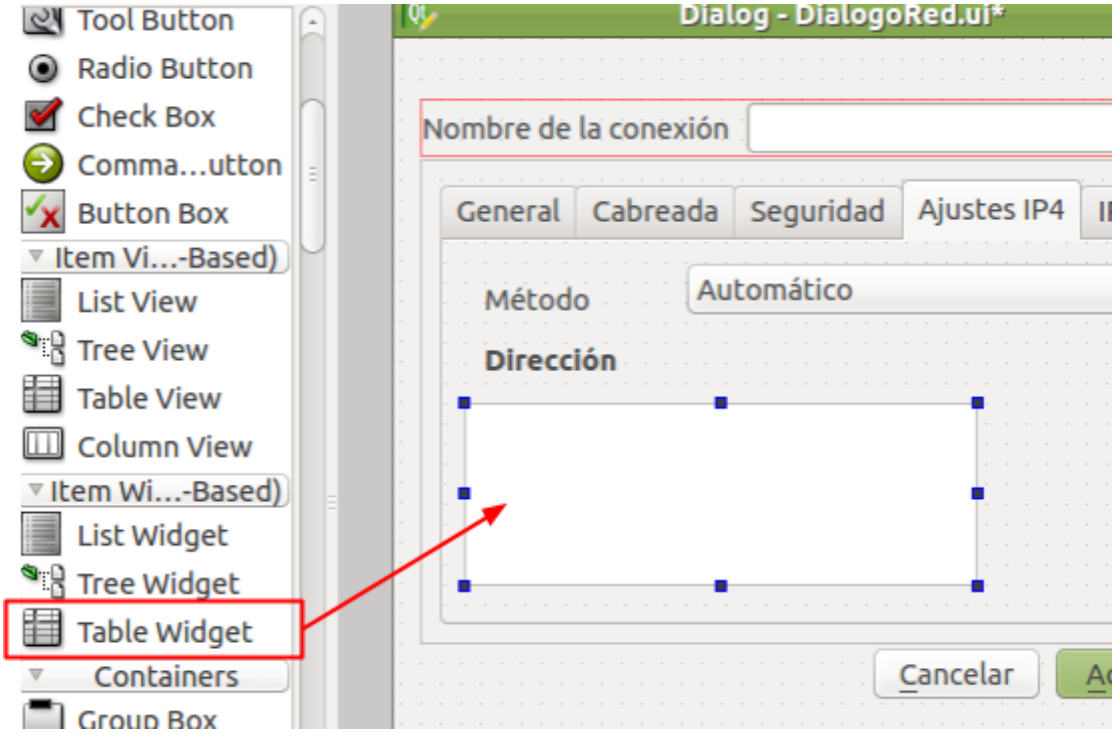
que va a mostrar, podemos aprovecharnos y usar un editor de elementos del combobox y escribir ya los valores que mostrará.



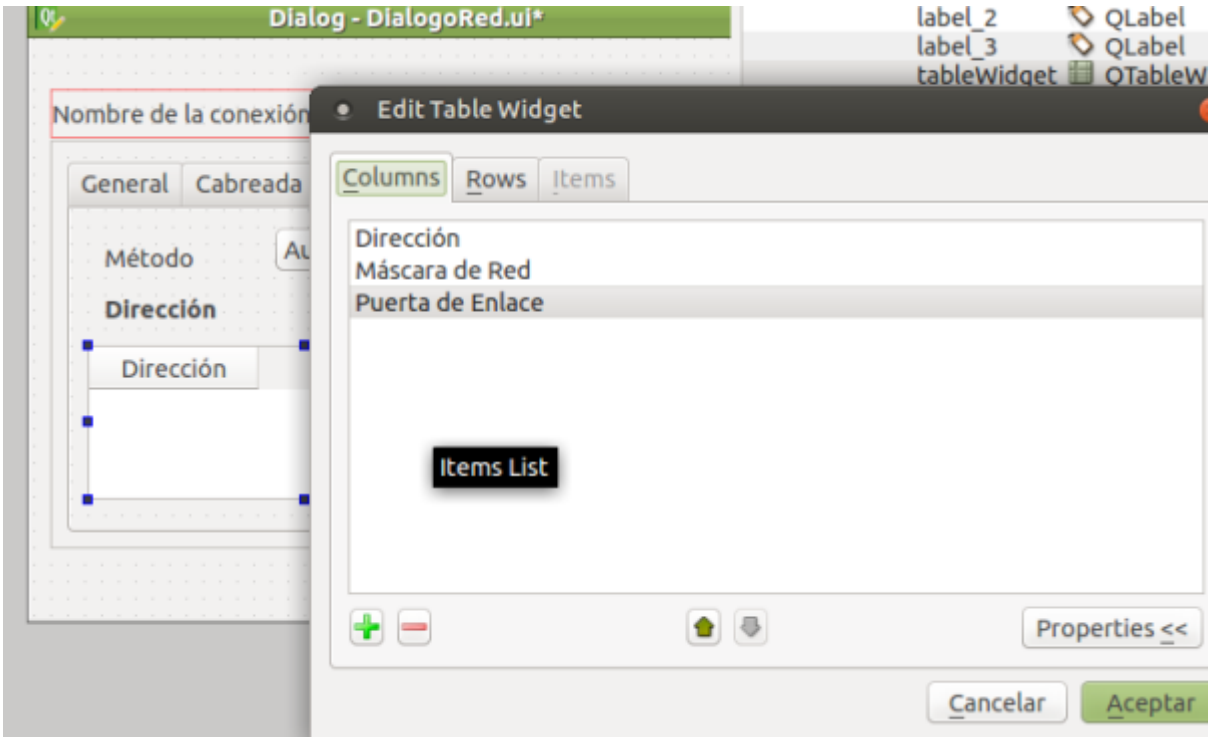
Seguimos poniendo la etiqueta. Hemos de cambiar la propiedad "Bold" de la fuente de ella para que salga en negrita



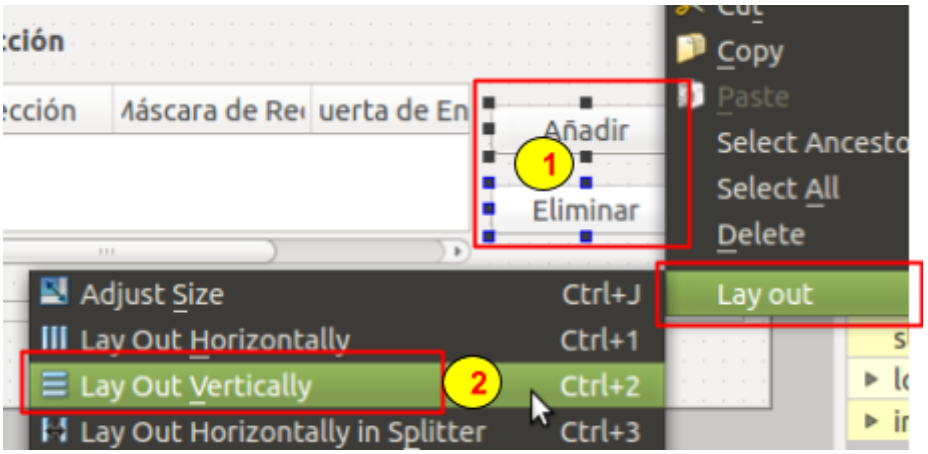
Añadimos el QTableWidget. Este componente será la tabla, que mostrará la cabezar y cada una de las direcciones IP



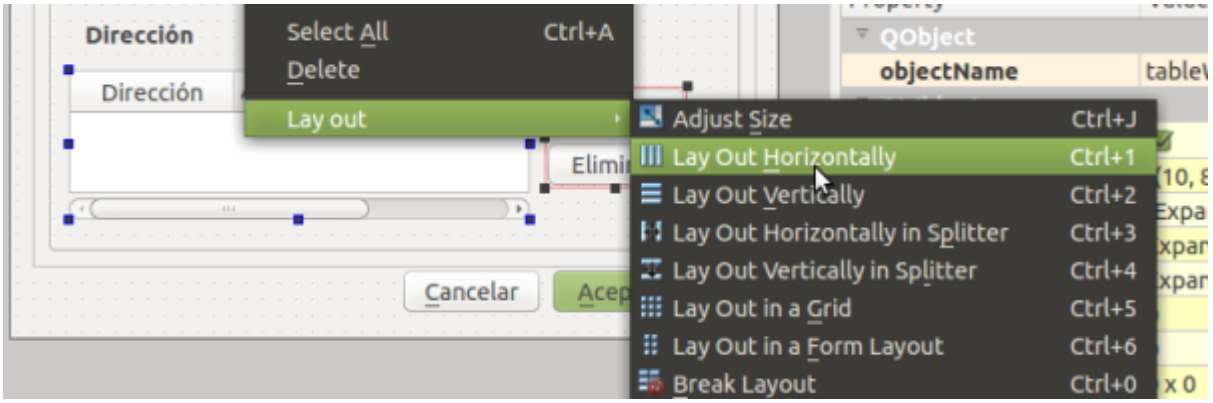
Editar columnas



Ponemos dos botones a la derecha de la tabla y los arreglamos en vertical.

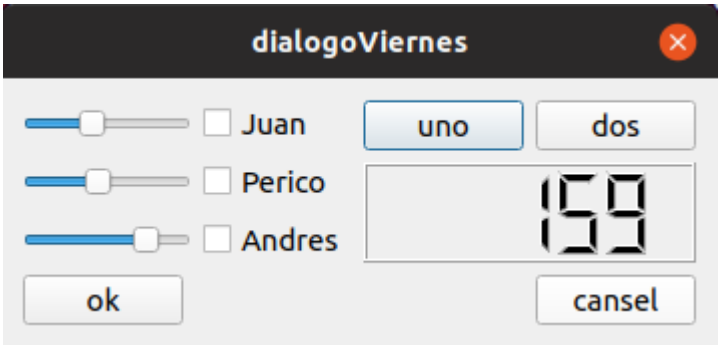


Ahora ponemos los dos botones que ya están enlazados y el `QTableWidget` horizontal



Anexo.

Ejercicio diálogo complejo (llamado aquí DialogoViernes)



Enunciado : El Display LCD Mostrará la suma de los valores de los sliders.

```
#ifndef DIALOGOVIERNES_H
#define DIALOGOVIERNES_H

#endif
```

```
#ifndef DIALOGOVIERNES_H
#define DIALOGOVIERNES_H

#include <QDialog>

class DialogoViernes : public QDialog {




};

#endif
```

```
#ifndef DIALOGOVIERNES_H
#define DIALOGOVIERNES_H

#include <QDialog>

class DialogoViernes : public QDialog {

public:
    DialogoViernes(QWidget *parent = NULL);

};

#endif
```

```
#ifndef DIALOGOVIERNES_H
#define DIALOGOVIERNES_H

#include <QDialog>

class DialogoViernes : public QDialog {

public:
    DialogoViernes(QWidget *parent = NULL);

    QPushButton *bOK, *bCancel,*b1, *b2;
    QSlider *slArriba, *slCentro, *slAbajo;
```

```
    QLCDNumber *lcd;
    QCheckBox *chArriba, *chCentro, *chAbajo
};
#endif
```

faltan to los includes

```
#ifndef DIALOGOVIERNES_H
#define DIALOGOVIERNES_H

#include <QDialog>

#include <QPushButton>
#include <QSlider>
#include <QLCDNumber>
#include <QCheckBox>

class DialogoViernes : public QDialog {

public:
    DialogoViernes(QWidget *parent = NULL);

    QPushButton *bOK, *bCancel,*b1, *b2;
    QSlider *slArriba, *slCentro, *slAbajo;
    QLCDNumber *lcd;
    QCheckBox *chArriba, *chCentro, *chAbajo;
};
#endif
```

Vamos con el .cpp

```
#include "dialogoViernes.h"


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



}
```

```
#include "dialogoViernes.h"


DialogoViernes::DialogoViernes(QWidget *parent ): QDialog(parent)
{
    bOK     = new QPushButton("ok");
    bCancel    = new QPushButton("cansel");
    b1     = new QPushButton("uno");
    b2     = new QPushButton("dos");

    slArriba = new QSlider();
    slCentro = new QSlider();
    slAbajo  = new QSlider();

    lcd       = new QLCDNumber;
    chArriba  = new QCheckBox();
    chCentro  = new QCheckBox();
    chAbajo   = new QCheckBox();

}
```

```
```

```
```

Solución final

.h

```
#ifndef DIALOGOVIERNES_H
#define DIALOGOVIERNES_H

#include <QDialog>

#include <QPushButton>
#include <QSlider>
```

```
#include <QLCDNumber>
#include <QCheckBox>

class DialogoViernes : public QDialog {


    Q_OBJECT
public:
    DialogoViernes(QWidget *parent = NULL);

    QPushButton *bOK, *bCancel,*b1, *b2;
    QSlider *slArriba, *slCentro, *slAbajo;
    QLCDNumber *lcd;
    QCheckBox *chArriba, *chCentro, *chAbajo;

public slots:
    void establecerLCD(int);

};
#endif
```

.cpp

```
#include <QHBoxLayout>
#include <QVBoxLayout>
#include "dialogoViernes.h"


DialogoViernes::DialogoViernes(QWidget *parent ): QDialog(parent)
{
    bOK     = new QPushButton("ok");
    bCancel    = new QPushButton("cansel");
    b1     = new QPushButton("uno");
    b2     = new QPushButton("dos");

    slArriba = new QSlider(Qt::Horizontal);
    slCentro = new QSlider(Qt::Horizontal);
    slAbajo  = new QSlider(Qt::Horizontal);

    lcd       = new QLCDNumber;
    chArriba  = new QCheckBox("Juan");
    chCentro  = new QCheckBox("Perico");
    chAbajo   = new QCheckBox("Andres");

    QVBoxLayout *mainLayout = new QVBoxLayout();
```

```
    QHBoxLayout *topLayout = new QHBoxLayout();
    QHBoxLayout *bottomLayout = new QHBoxLayout();
    QVBoxLayout *rightLayout = new QVBoxLayout();
    QVBoxLayout *leftLayout = new QVBoxLayout();

    QHBoxLayout *lArriba= new QHBoxLayout();
    QHBoxLayout *lCentro= new QHBoxLayout();
    QHBoxLayout *lAbajo= new QHBoxLayout();

    QHBoxLayout *topLeftButtonsLayout = new QHBoxLayout();

    topLeftButtonsLayout->addWidget(b1);
    topLeftButtonsLayout->addWidget(b2);
    rightLayout->addLayout(topLeftButtonsLayout);
    rightLayout->addWidget(lcd);

    lArriba->addWidget(slArriba); lArriba->addWidget(chArriba);
    lCentro->addWidget(slCentro); lCentro->addWidget(chCentro);
    lAbajo->addWidget(slAbajo); lAbajo->addWidget(chAbajo);

    lArriba->addStretch();
    lCentro->addStretch();
    lAbajo->addStretch();
    leftLayout->addLayout(lArriba);
    leftLayout->addLayout(lCentro);
    leftLayout->addLayout(lAbajo);

    bottomLayout->addWidget(bOK);
    bottomLayout->addStretch();
    bottomLayout->addWidget(bCancel);

    topLayout->addLayout(leftLayout);
    topLayout->addLayout(rightLayout);
    mainLayout->addLayout(topLayout);
    mainLayout->addLayout(bottomLayout);
    setLayout(mainLayout);

    connect(slArriba,SIGNAL(valueChanged(int)),
    this ,SLOT(establecerLCD(int)));


    connect(slCentro,SIGNAL(valueChanged(int)),
    this ,SLOT(establecerLCD(int)));

    connect(slAbajo,SIGNAL(valueChanged(int)),
    this ,SLOT(establecerLCD(int)));

}
```

```
void DialogoViernes::establecerLCD(int cantidad){

    int suma = slArriba->value() + slCentro->value() +
       slAbajo->value();

    lcd->display(suma);
}
```