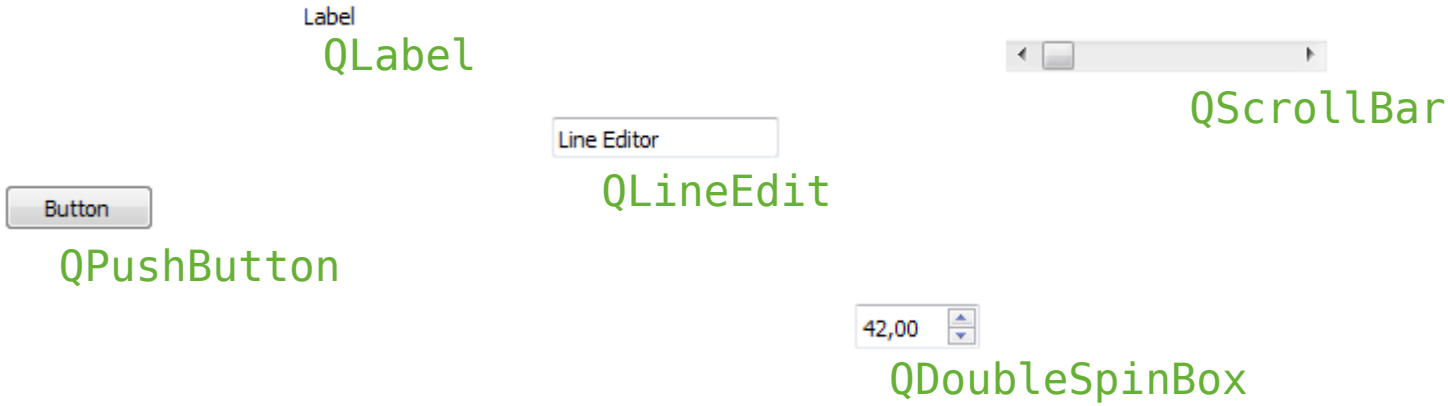
Qt in Education

Widgets and Layouts

NOKIA

User Interface Components

- User interfaces are built from individual widgets

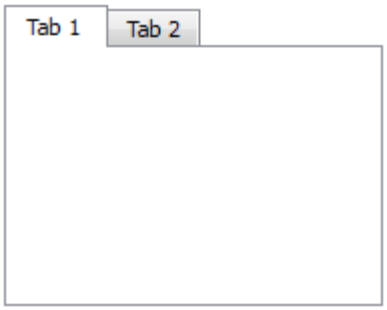


- 46 widgets in Designer
- 59+ direct descendants from `QWidget`

Widgets in Widgets

- Widgets are placed in hierarchies

`QGroupBox`



`QTabWidget`

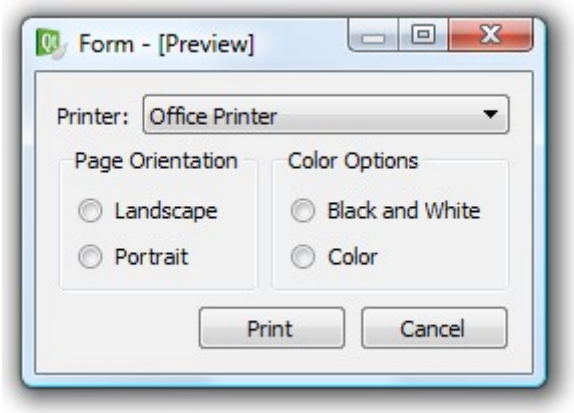
- Container classes provide visual structure...
- ...but also functional (e.g. `QRadioButton`)

Traits of a Widget

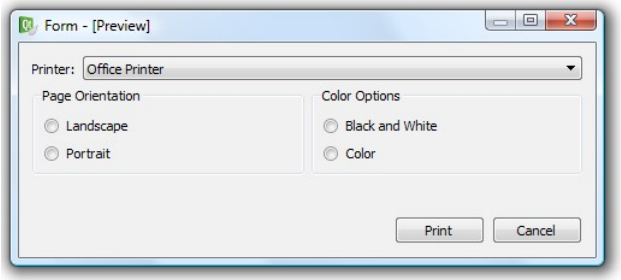
- Occupies a rectangular area of the screen
- Receives events from input devices
- Emits signals for “notable” changes

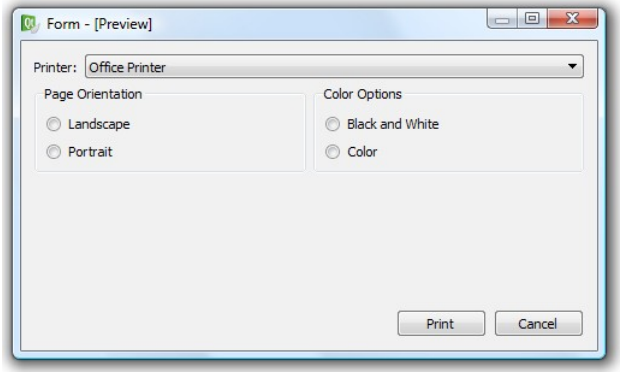
- Are structured in a hierarchy
- Can contain other widgets

An Example Dialog



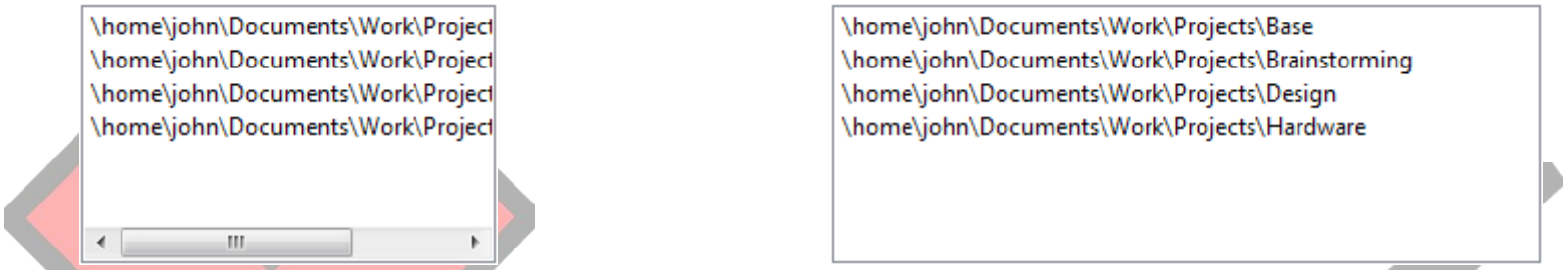
- Widgets are placed in layouts – to make the user interface elastic





Why is Elastic Good?

- Lets widgets adapt to contents



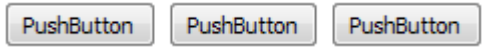
- Lets widgets adapt to translations

- Lets widgets adapt to user settings

Layouts

- There are several possible layouts available

QVBoxLayout



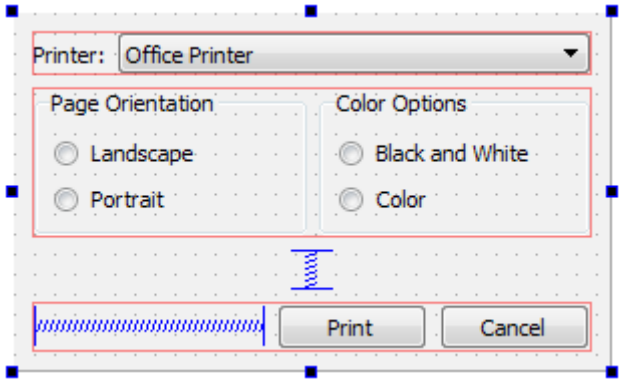
QHBoxLayout

QGridLayout

- Layouts and widgets “negotiate” for sizes and positions
- Spacer springs can be used to fill voids

An Example Dialog

- Dialogs are built from multiple layers of layouts and widgets

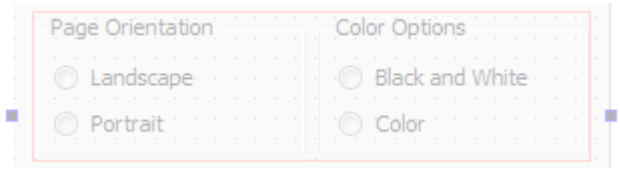


Object	Class
Form	QWidget
horizontalLayout	QHBoxLayout
label	QLabel
printerBox	QComboBox
horizontalLayout_2	QHBoxLayout
cancelButton	QPushButton
horizontalSpacer	Spacer
printButton	QPushButton
horizontalLayout_3	QHBoxLayout
groupBox	QGroupBox
landscapeButton	QRadioButton
portraitButton	QRadioButton
groupBox_2	QGroupBox
bwButton	QRadioButton
colorButton	QRadioButton
verticalSpacer	Spacer

Note that layouts are not parents of the widgets that they manage.

An Example Dialog

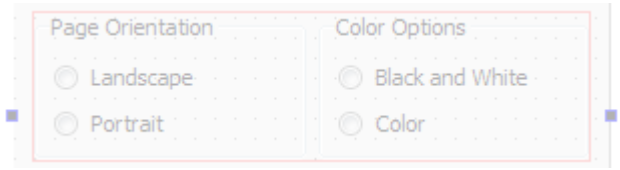
```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);

QHBoxLayout *groupLayout = new QHBoxLayout();

...

outerLayout->addLayout(groupLayout);

outerLayout->addSpacerItem(new QSpacerItem(...));

QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```



An Example Dialog

```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);
```


```
QHBoxLayout *groupLayout = new QHBoxLayout();

...

outerLayout->addLayout(groupLayout);
```

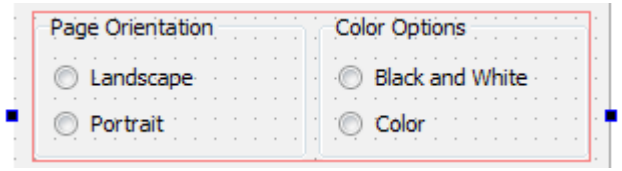



```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

```
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```

An Example Dialog

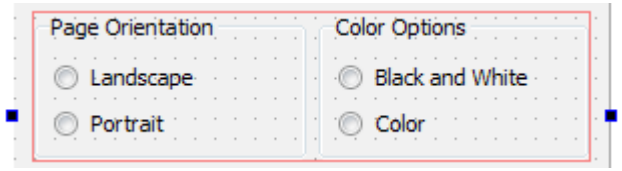
```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);


QHBoxLayout *groupLayout = new QHBoxLayout();


...


outerLayout->addLayout(groupLayout);


outerLayout->addSpacerItem(new QSpacerItem(...));


QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```



An Example Dialog

```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);
```


```
QHBoxLayout *groupLayout = new QHBoxLayout();

...

outerLayout->addLayout(groupLayout);
```

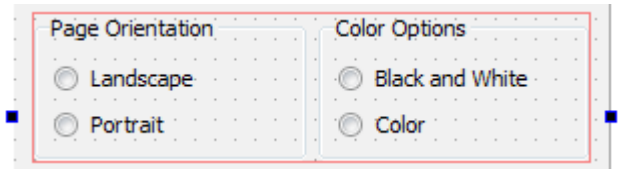



```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

```
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```

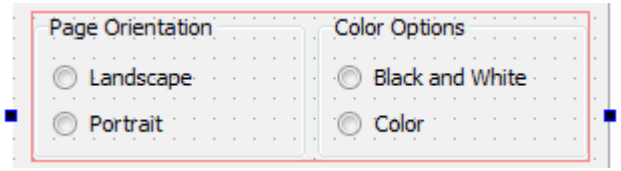
An Example Dialog

```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);


QHBoxLayout *groupLayout = new QHBoxLayout();


...


outerLayout->addLayout(groupLayout);


outerLayout->addSpacerItem(new QSpacerItem(...));


QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```

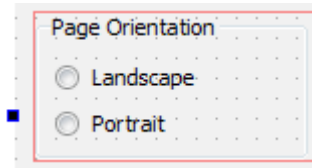


An Example Dialog

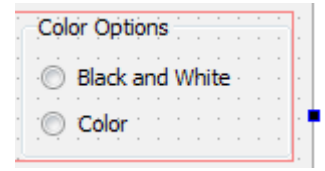
- Horizontal box, contains group boxes, contains vertical boxes, contains radio buttons




```
QHBoxLayout *groupLayout = new QHBoxLayout();

QGroupBox *orientationGroup = new QGroupBox();
QVBoxLayout *orientationLayout = new QVBoxLayout(orientationGroup);
orientationLayout->addWidget(new QRadioButton("Landscape"));
orientationLayout->addWidget(new QRadioButton("Portrait"));
groupLayout->addWidget(orientationGroup);
```




```
QGroupBox *colorGroup = new QGroupBox();
QVBoxLayout *colorLayout = new QVBoxLayout(colorGroup);
colorLayout->addWidget(new QRadioButton("Black and White"));
colorLayout->addWidget(new QRadioButton("Color"));
groupLayout->addWidget(colorGroup);
```



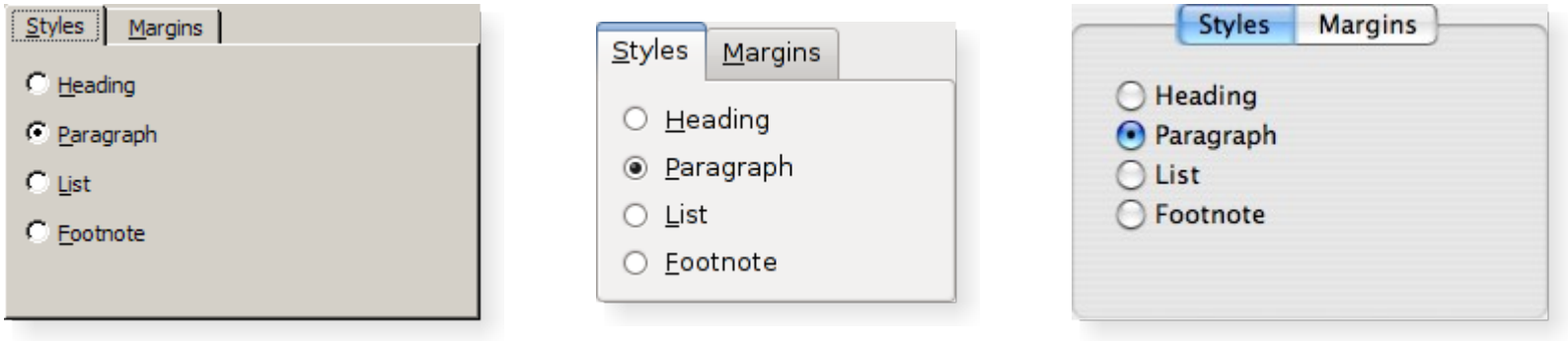
An Example Dialog

- You can build the same structure using Designer

Object	Class
Form	QWidget
horizontalLayout	QHBoxLayout
label	QLabel
printerBox	QComboBox
horizontalLayout_2	QHBoxLayout
cancelButton	QPushButton
horizontalSpacer	Spacer
printButton	QPushButton
horizontalLayout_3	QHBoxLayout
groupBox	QGroupBox
landscapeButton	QRadioButton
portraitButton	QRadioButton
groupBox_2	QGroupBox
bwButton	QRadioButton
colorButton	QRadioButton
verticalSpacer	Spacer

Cross Platform Styles

- Widgets are drawn using a platform specific style to ensure a native look

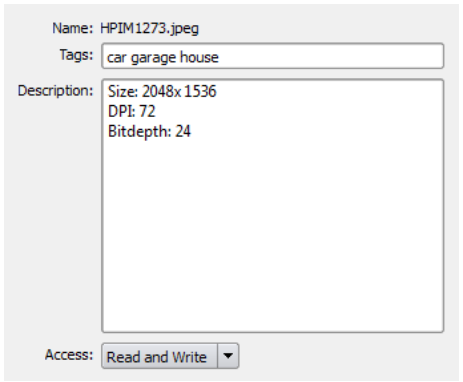


Cross Platform Issues

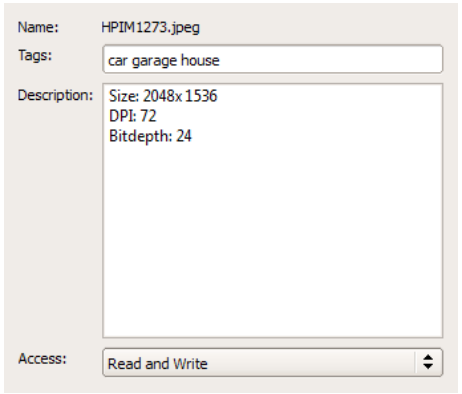
- Comparing user interfaces tells us that there is more to it than just changing the style of the widgets
  - Form layout
  - Dialog button ordering
  - Standard dialogs

Cross Platform Issues

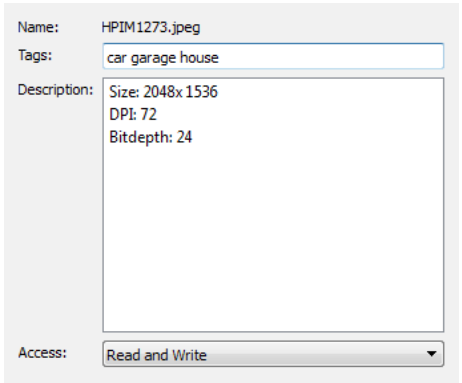
- Comparing user interfaces tells us that there is more to it than just changing the style of the widgets
  - **Form layout**
  - Dialog button ordering
  - Standard dialogs



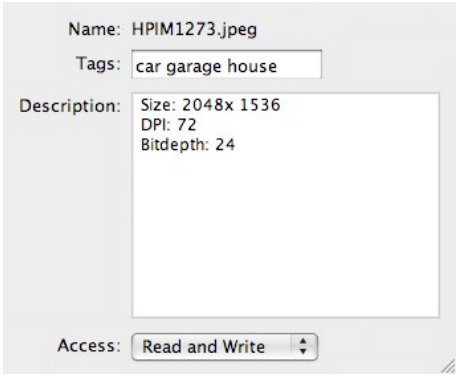
Plastique



ClearLooks



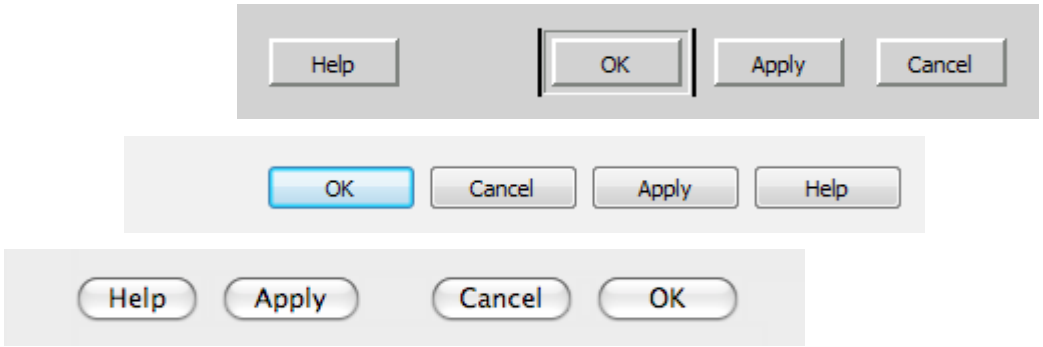
Windows



MacOS X

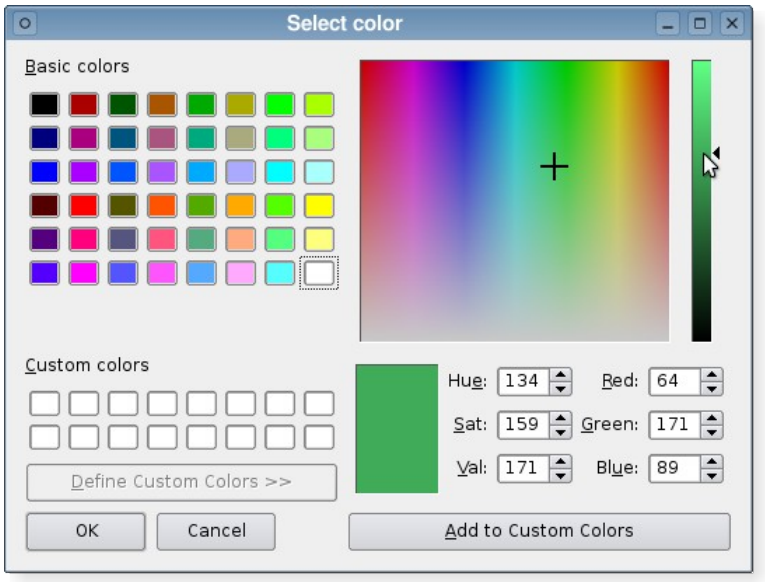
Cross Platform Issues

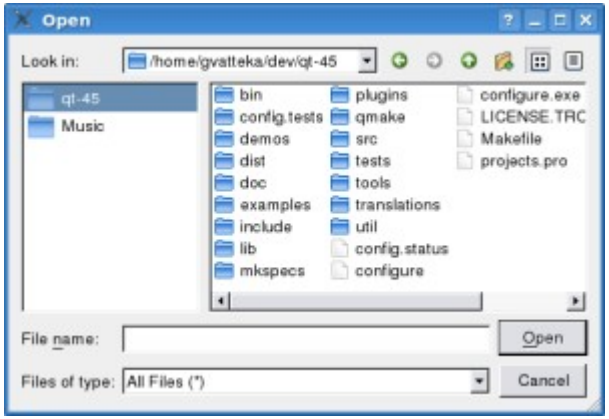
- Comparing user interfaces tells us that there is more to it than just changing the style of the widgets
  - Form layout
  - **Dialog button ordering**
  - Standard dialogs



Cross Platform Issues

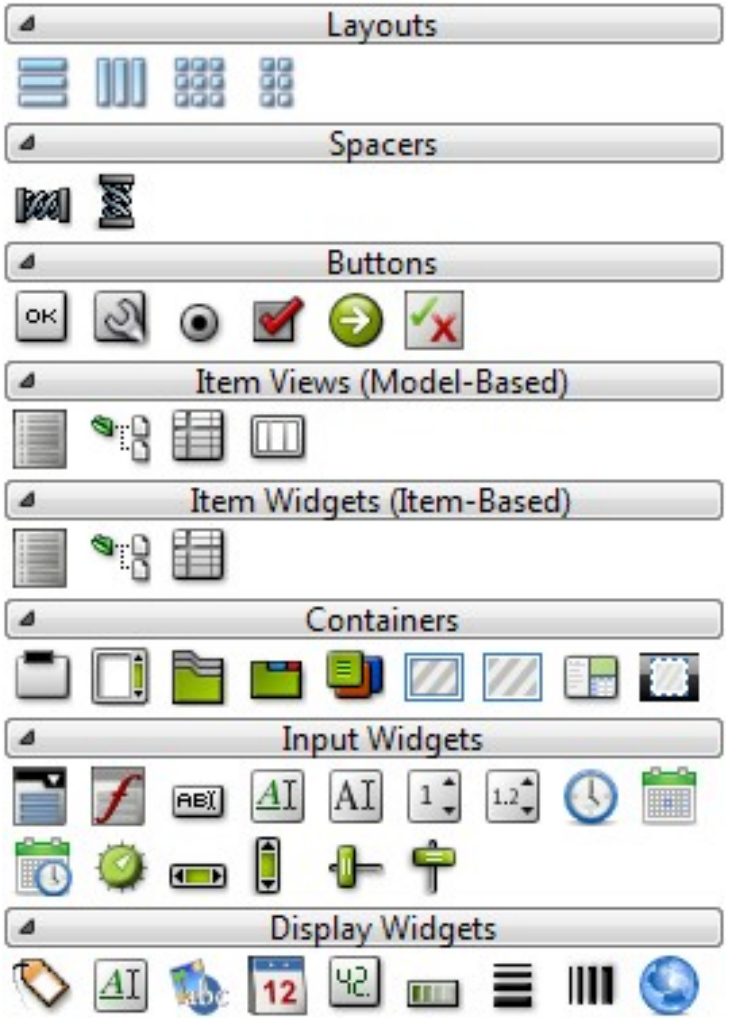
- Comparing user interfaces tells us that there is more to it than just changing the style of the widgets
  - Form layout
  - Dialog button ordering
  - **Standard dialogs**





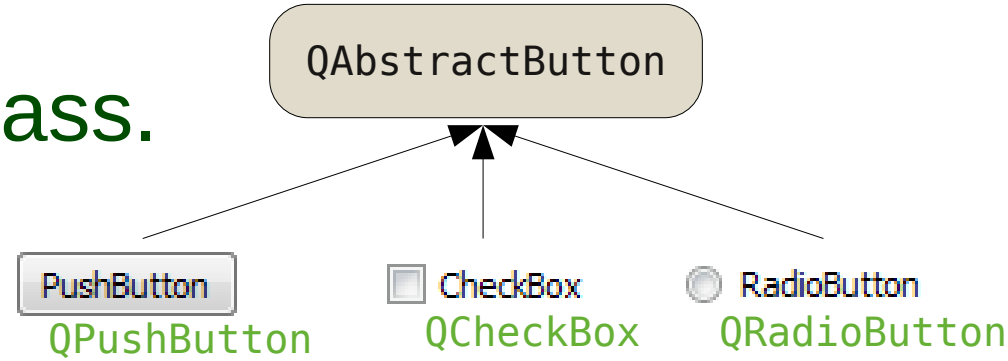
Common Widgets

- Qt contains numerous widgets for all common situations.
- Designer has a good overview of the widget groups



Common Widgets
Buttons

- All buttons inherit the `QAbstractButton` base class.



- Signals
  - `clicked()` - emitted when the button is clicked (button released).
  - `toggled(bool)` – emitted when the check state of the button is changed.

- Properties
  - `checkable` – true if the button can be checked. Makes a push button toggle.
  - `checked` – true when the button is checked.
  - text – the text of the button.
  - icon – an icon on the button (can be displayed together with text).

Common Widgets
Item Widgets

QListWidget

- `QListWidget` is used to show lists of items
- Adding items
  - `addItem(QString)` – appends an item to the end of the list
  - `insertItem(int row, QString)` – inserts an item at the specified row
- Selection
  - `selectedItems` – returns a list of `QListWidgetItem`s used `QListWidgetItem::text` to determine the text
- Signals
  - `itemSelectionChanged` – emitted when the selection is changed

QComboBox

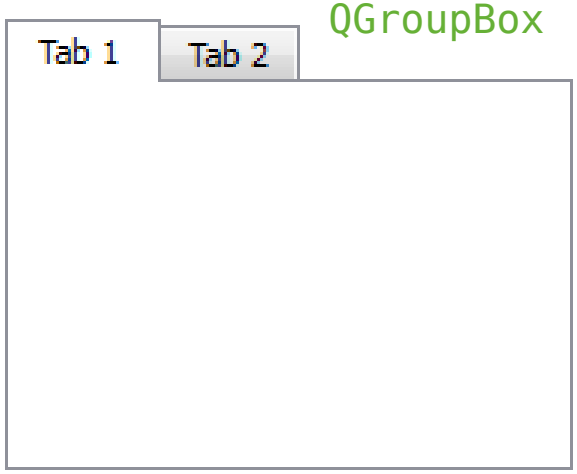
- `QComboBox` shows a list with a single selection in a more compact format.

Common Widgets
Containers

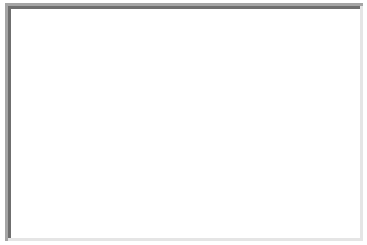
- Container widgets are used to structure the user interface
- They can be considered passive (not entirely true)
- A plain `QWidget` can be used as a container
- Designer: Place widgets in the container and apply a layout to the container
- Code: Create a layout for the container and add widgets to the layout

```
QGroupBox *box = new QGroupBox();
QVBoxLayout *layout = new QVBoxLayout(box);
layout->addWidget(...);
...
```

QGroupBox



QTabWidget



QFrame

Common Widgets
Input Widgets

- Use `QLineEdit` for single line text entries
- Signals:
  - `textChanged(QString)` - emitted when the text is altered
  - `editingFinished()` - emitted when the widget is left
  - `returnPressed()` - emitted when return is pressed
- Properties
  - `text` – the text of the widget
  - `maxLength` – limits the length of the input
  - `readOnly` – can be set to true to prevent editing (still allows copying)

`QLineEdit`

Common Widgets
Input Widgets

- Use `QTextEdit` or `QPlainTextEdit` for multi line text entries
- Signals
  - `textChanged()` - emitted when the text is altered
- Properties
  - `plainText` – unformatted text
  - `html` – HTML formatted text
  - `readOnly` – can be set to prevent editing

`QTextEdit`

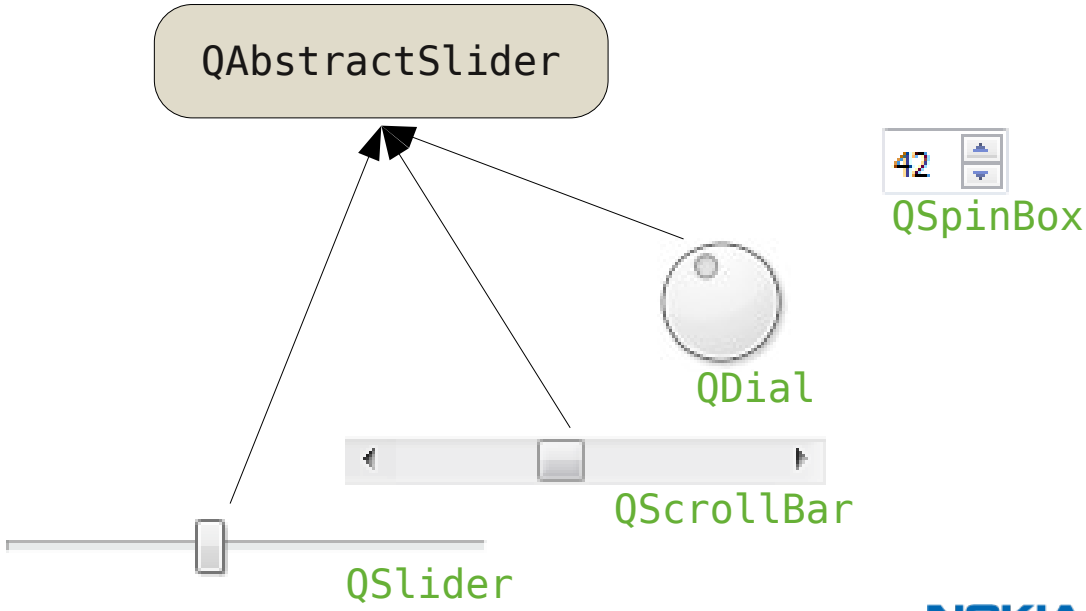
`QComboBox`

- `QComboBox` can be made editable through the editable property
- Signals
  - `editTextChanged(QString)` – emitted while the text is being edited
- Properties
  - `currentText` – the current text of the combo box

Common Widgets
Input Widgets

- There is a large choice of widgets for editing integer values
- There are more for doubles, time and dates

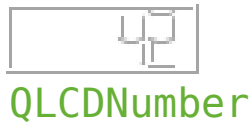
- Signals:
  - `valueChanged(int)` - emitted when the value is updated
- Properties
  - `value` – the current value
  - `maximum` – the maximum value
  - `minimum` – the minimum value



Common Widgets
Display Widgets

- The `QLabel` displays a text *or* a picture
- Properties
  - `text` – a text for the label
  - `pixmap` – a picture to show

- `QLCDNumber` is used to display integer values
- Properties
  - `intValue` – the value shown (set using `display(int)`)

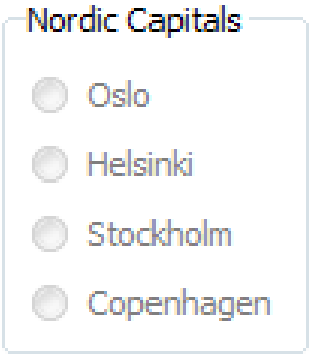


Common Widget Properties

- All widgets have a set of common properties inherited from the `QWidget` base class
- `enabled` – enable or disable user interaction

- `visible` – shown or not (alter with show and hide)

- These properties affect child widgets as well. For instance, enable or disable a container widget.



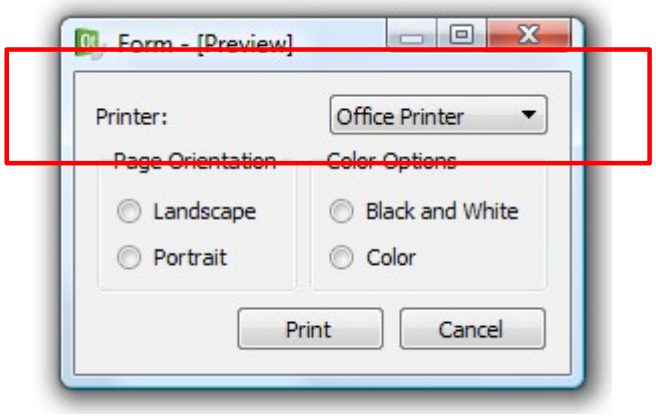
Break

Size Policies

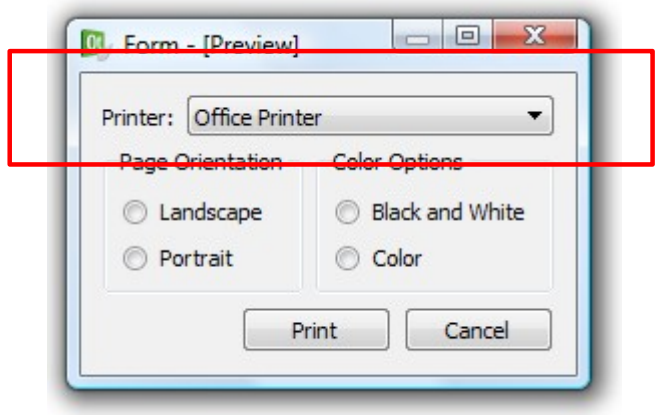
- Layout is a negotiation process between widgets and layouts
- Layouts bring structure
  - horizontal and vertical boxes
  - grid
- Widgets supply
  - size policies for each direction
  - minimum and maximum sizes

Size Policies

- The example was not complete!




```
printerList->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed)
```



Size Policies

- Each widget has a size hint that is combined with a policy for each direction
  - `Fixed` – the hint specifies the size of the widget
  - `Minimum` – the hint specifies the smallest possible size
  - `Maximum` – the hint specifies the largest possible size
  - `Preferred` – the hint specifies preferred, but not required
  - `Expanding` – as preferred, but wants to grow
  - `MinimumExpanding` – as minimum, but wants to grow
  - `Ignored` – the size hint is ignored, widget gets as much space as possible

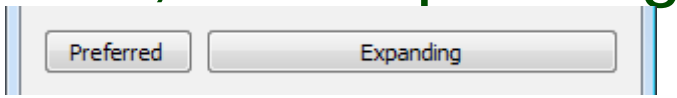
Size Policies

- Each widget has a size hint that is combined with a policy for each direction
  - `Fixed` – fixed to size hint
  - `Minimum` – can **grow**
  - `Maximum` – can **shrink**
  - `Preferred` – can **grow**, can **shrink**
  - `Expanding` – can **grow**, can **shrink**, *wants* to **grow**
  - `MinimumExpanding` – can **grow**, *wants* to **grow**
  - `Ignored` – the size hint is ignored, can **grow**, can **shrink**

What If?

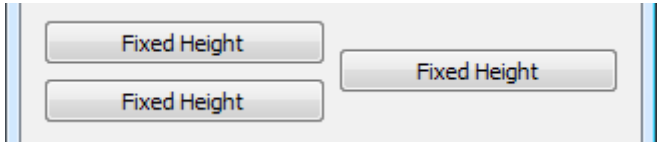
- Two preferred next to each other

- One preferred, one expanding



- Two expanding next to each other

- Not enough widget to fill the space (fixed)



More on Sizes

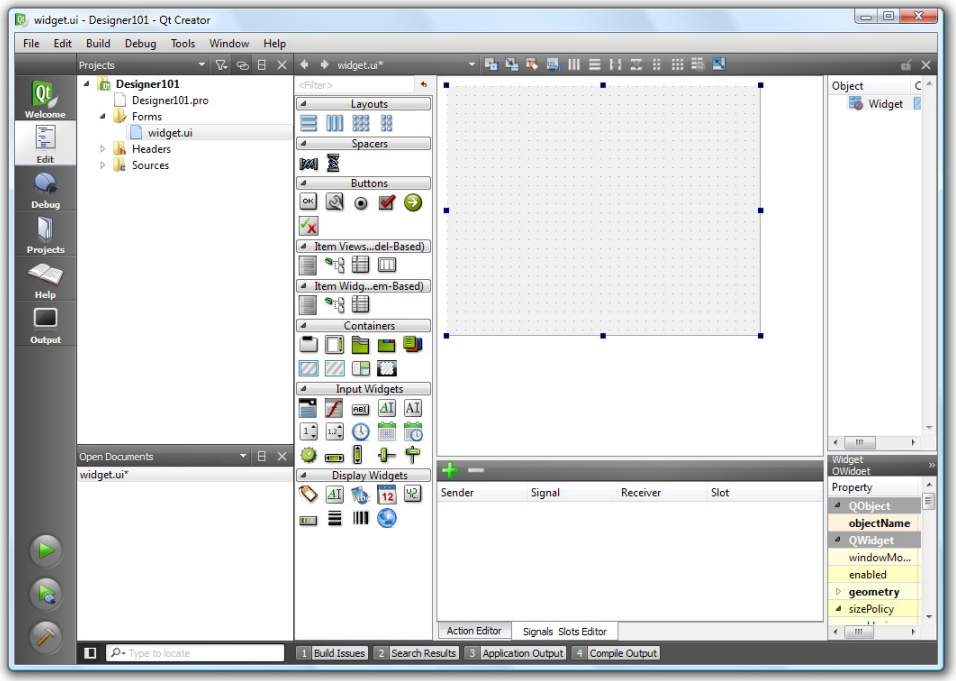
- Widget sizes can be further controlled using the properties for maximum and minimum size

- `maximumSize` – largest possible size
- `minimumSize` – smallest possible size

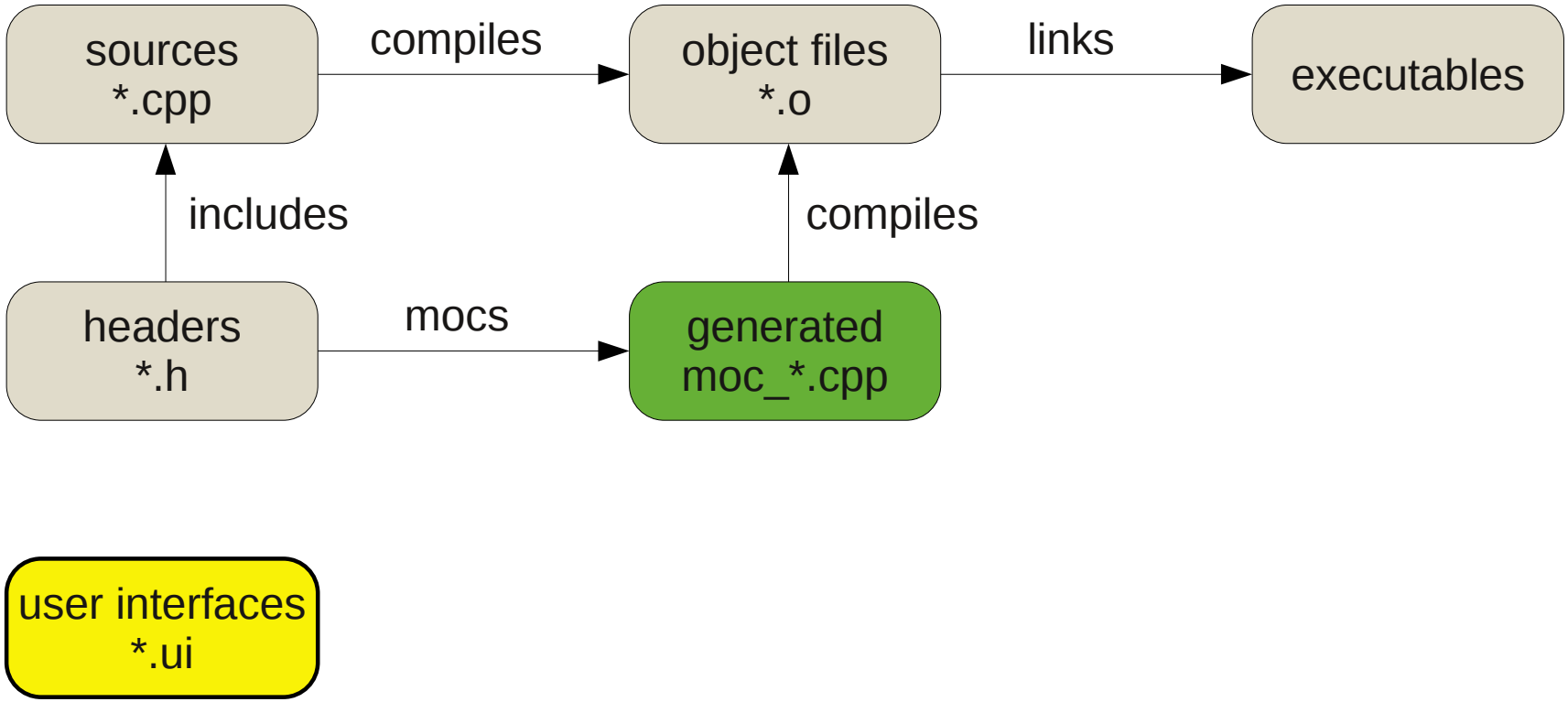
```
ui->pushButton->setMinimumSize(100, 150);
ui->pushButton->setMaximumHeight(250);
```

Introducing Designer

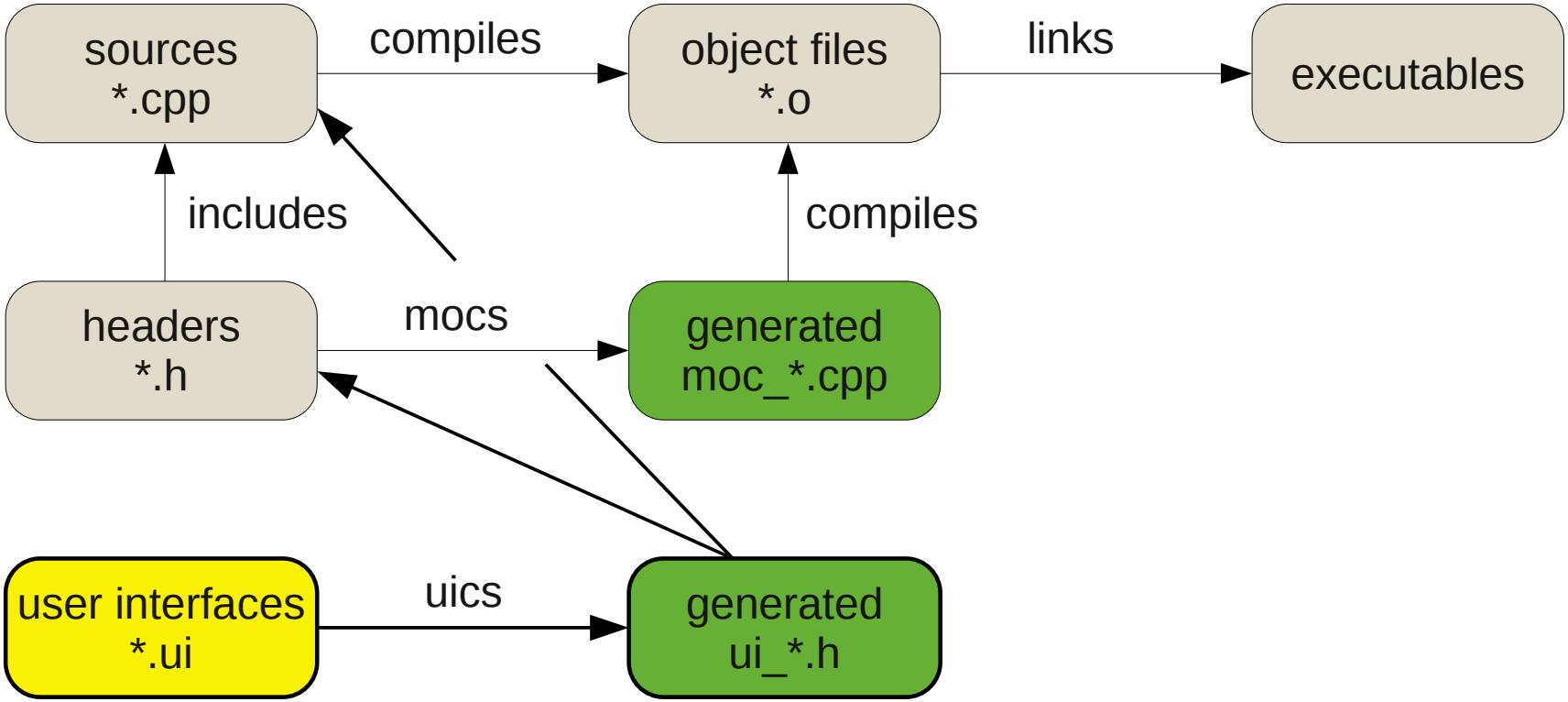
- Designer was historically a separate tool, but is now part of Qt Creator
- A visual editor for forms
- Drag-and-drop widgets
- Arrange layouts
- Make connections



Introducing Designer



Introducing Designer



Using the Code

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

namespace Ui {
    class Widget;
}

class Widget : public QWidget {
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
};

#endif // WIDGET_H
```

Forward declaration of the `Ui::Widget` class

A `Ui::Widget` pointer, `ui`, refers to all widgets

Basically a standard `QWidget` derived class

Using the Code

```
#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
}

Widget::~Widget()
{
    delete ui;
}
```

Calls `setupUi`, creating all the widgets as children to the given parent (`this`).

Instanciates the `Ui::Widget` class as `ui`

Deletes the `ui` object

Using Designer

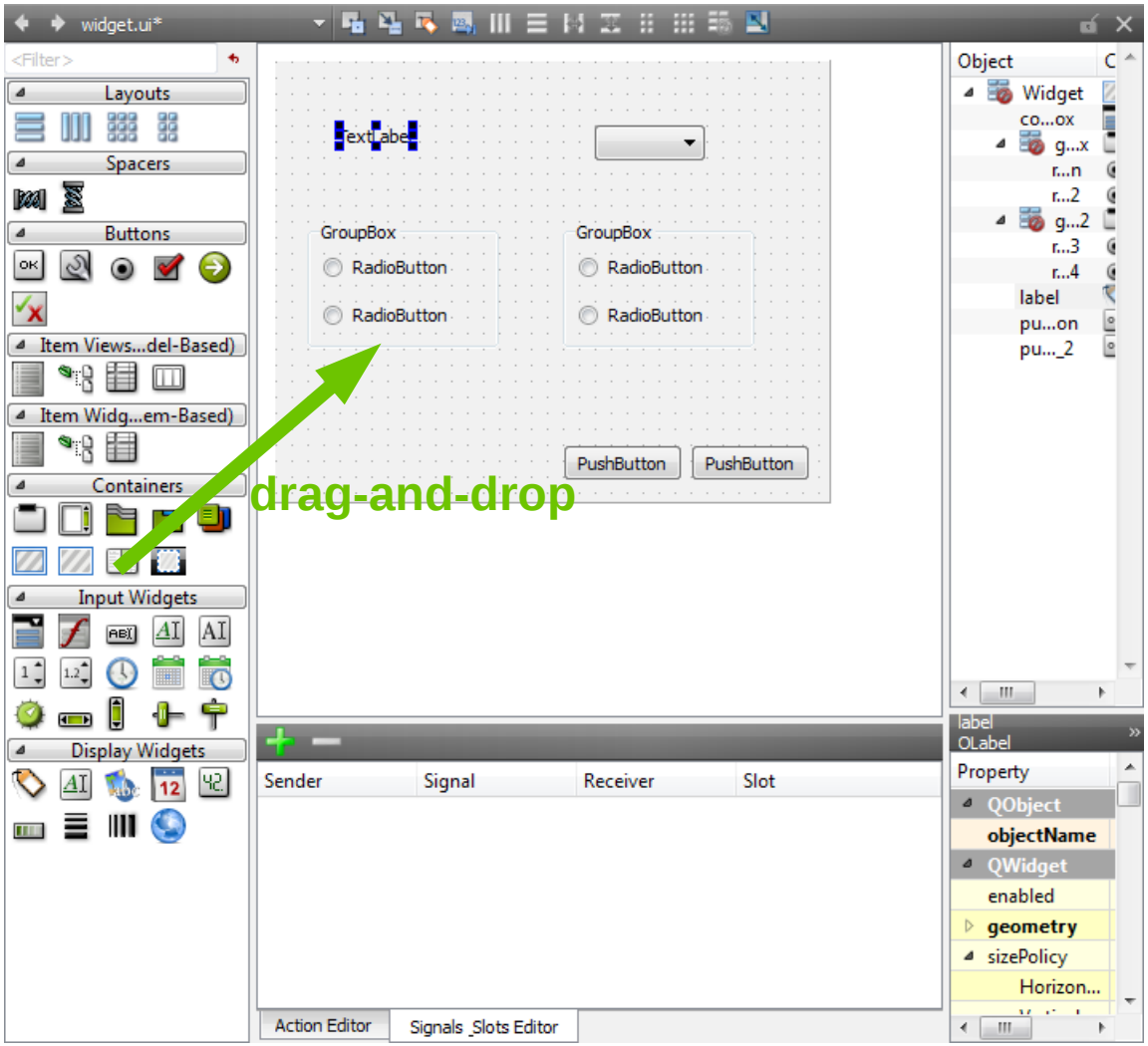
- Basic working order
  1. Place widgets roughly
  2. Apply layouts from the inside out, add spacers as needed
  3. Make connections
  4. Use from code

  - Throughout the process, alter and edit properties

- Practice makes perfect!

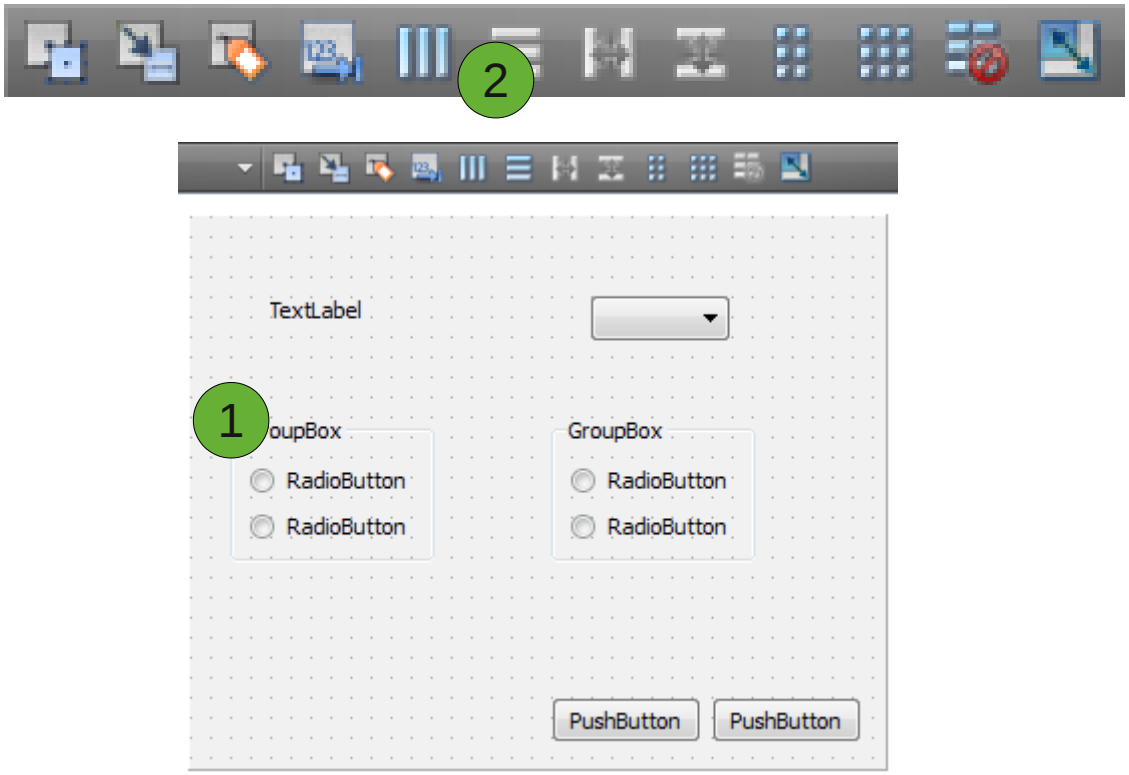
Using Designer

Place widgets roughly



Using Designer

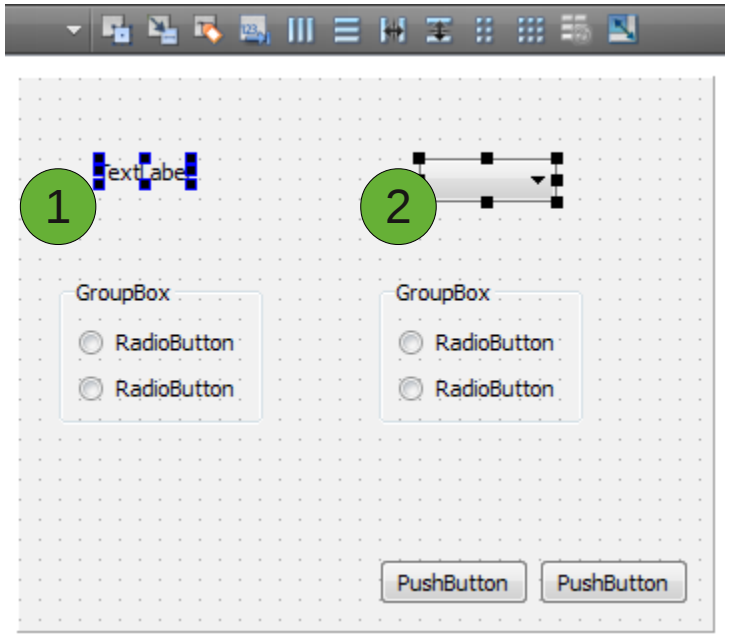
Apply layouts from the inside out, add spacers as needed



1. Select each group box, **2**. apply a vertical box layout

Using Designer

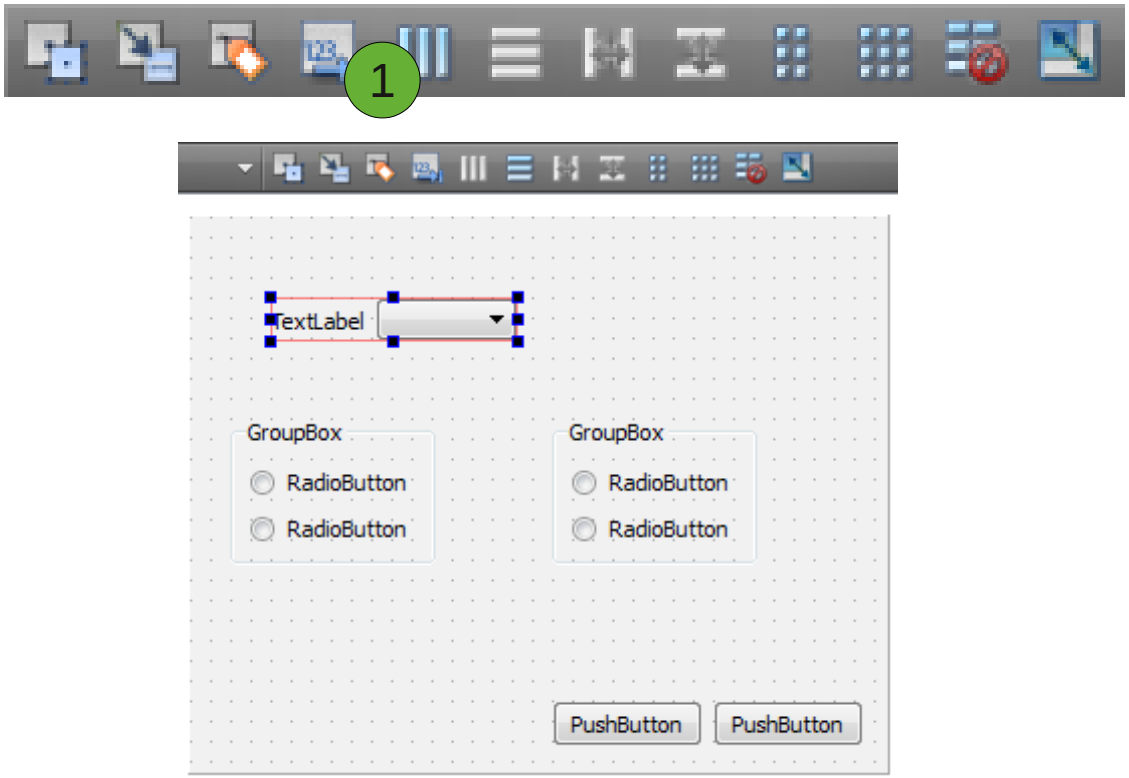
Apply layouts from the inside out, add spacers as needed



1. Select the label (click), **2**. Select the combobox (Ctrl+click)

Using Designer

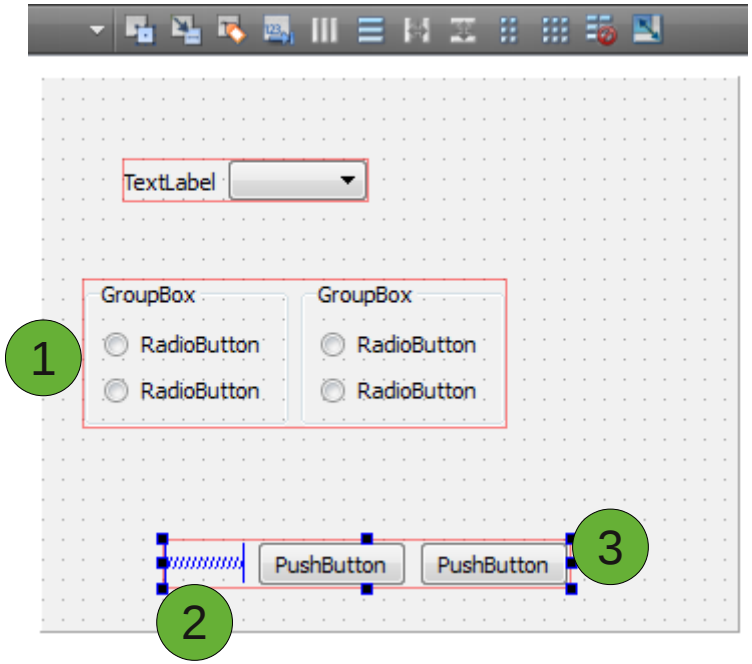
Apply layouts from the inside out, add spacers as needed



1. Apply a horizontal box layout

Using Designer

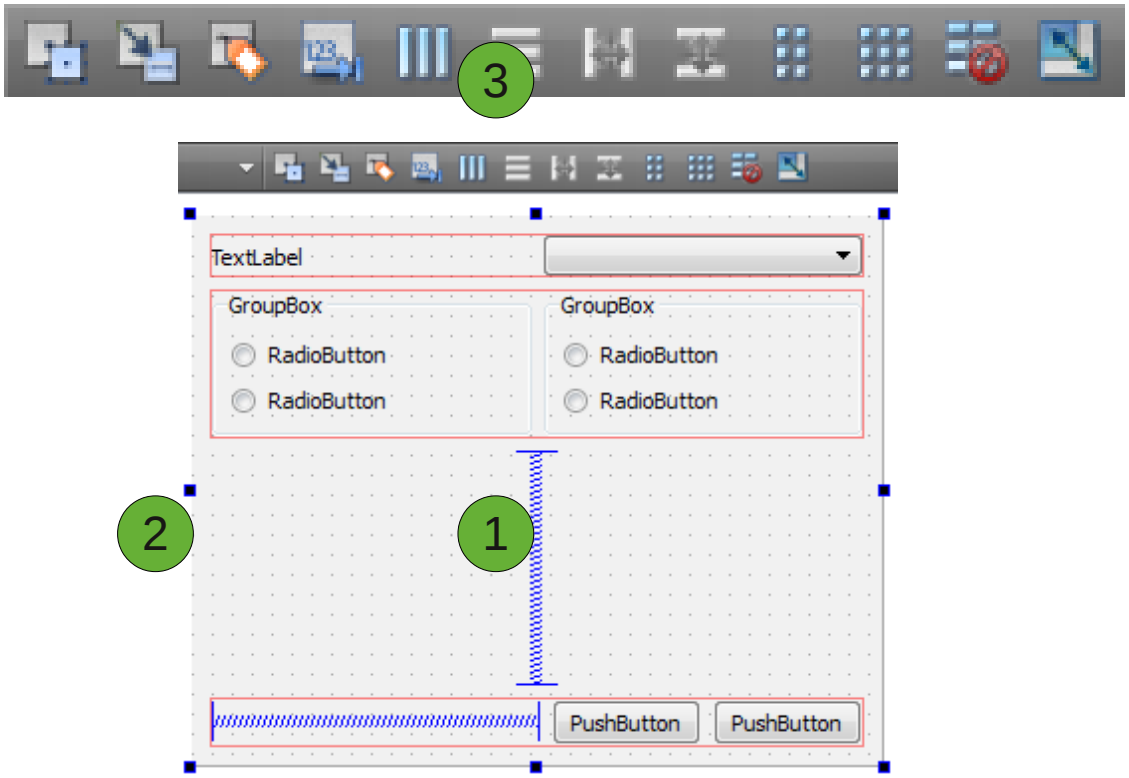
Apply layouts from the inside out, add spacers as needed



1. Select both group boxes and lay them out, **2**. add a horizontal spacer,
3. place the buttons and spacer in a layout

Using Designer

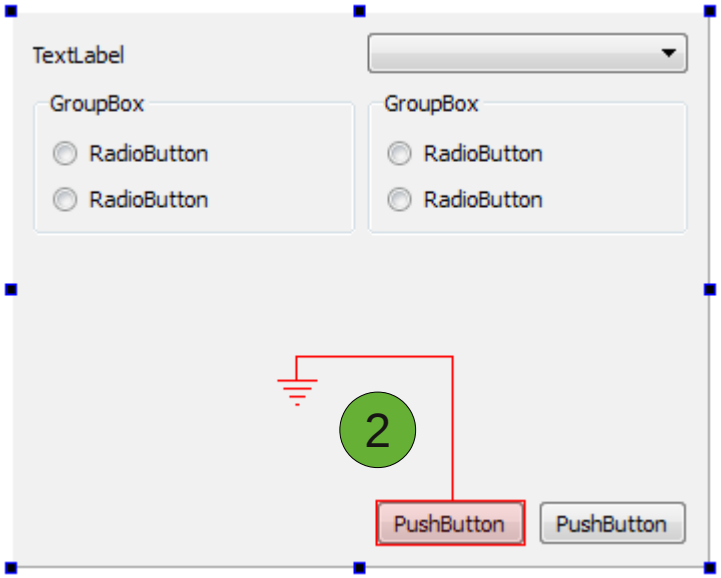
Apply layouts from the inside out, add spacers as needed

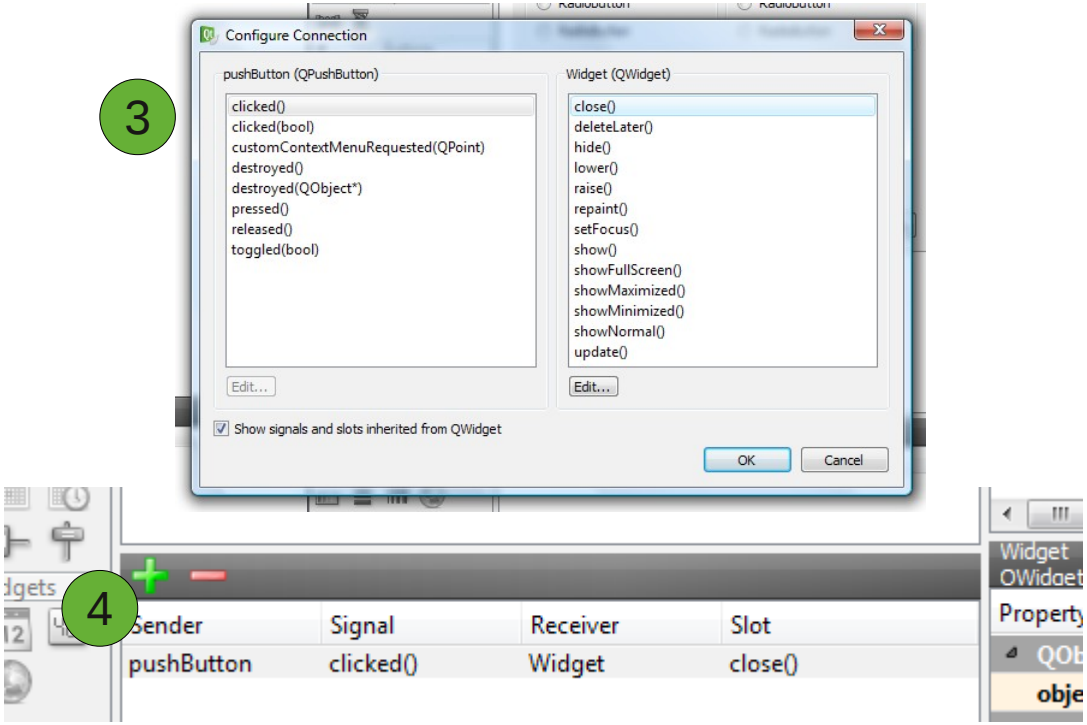


1. Add a vertical spacer, **2**. select the form itself, **3**. apply a vertical box layout

Using Designer

Make connections (between widgets)



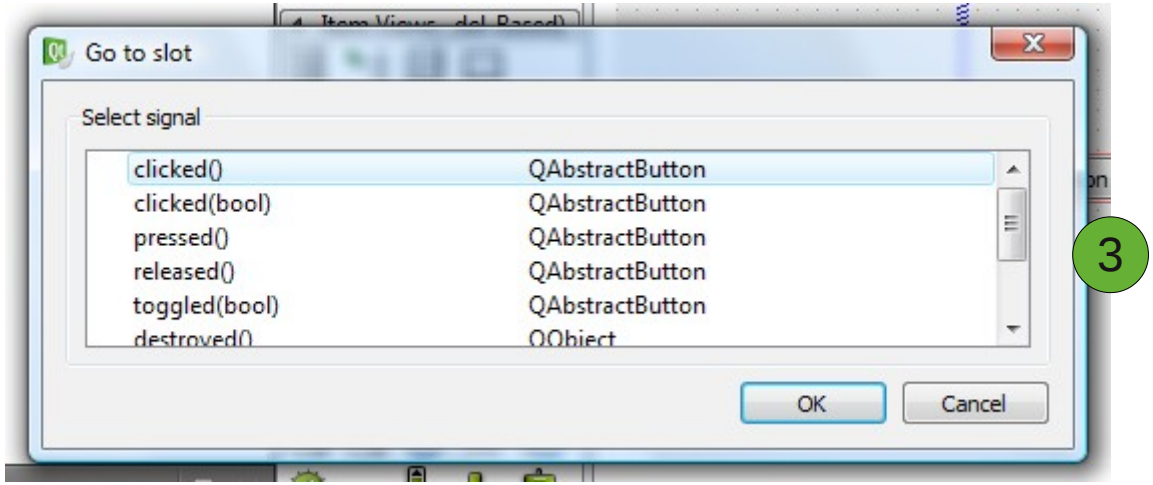


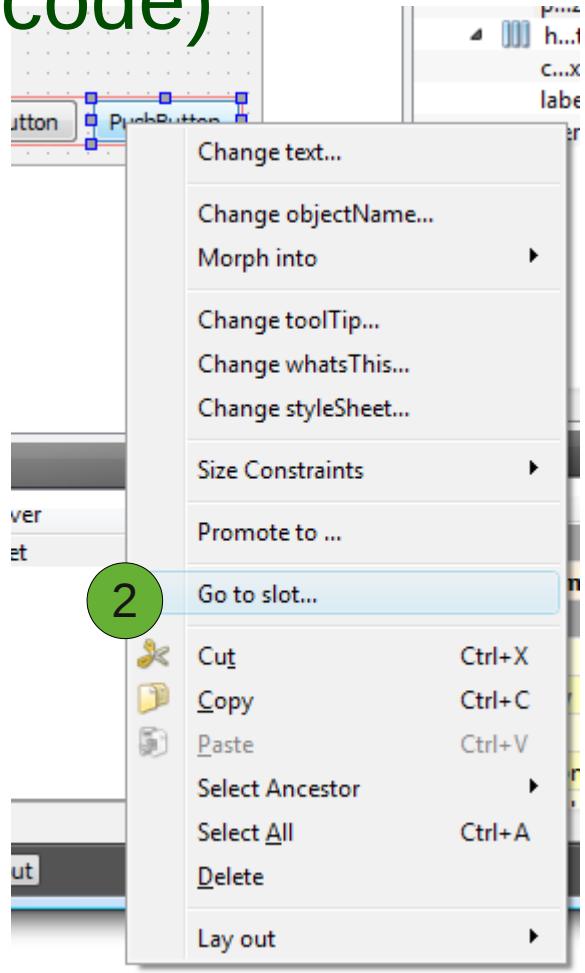
1. Switch to signals and slot editing mode, **2**. drag from one widget to another, **3**. pick the signal and slot, **4**. see the result in the connections' dock

Using Designer

Make connections (to your code)

1





1. Use the widget editing mode, **2**. right click on a widget and pick *Go to slot...*
3. pick the signal to connect to your code

Using Designer

Use from code

- Access all widgets through the `ui` class member

```
class Widget : public QWidget {
    ...
private:
    Ui::Widget *ui;
};
```

```
void Widget::memberFunction()
{
    ui->pushButton->setText(...);
}
```

Top-level Windows

- Widgets without a parent widget automatically become windows
  - `QWidget` – a plain window, usually non-modal
  - `QDialog` – a dialog, usually expecting a result such as OK, Cancel, etc
  - `QMainWindow` – an application window with menus, toolbars, statusbar, etc
- `QDialog` and `QMainWindow` inherit `QWidget`

Using QWidget as Window

- Any widget can be a window

- Widgets without a parent are automatically windows
- Widgets with a parent have to pass the `Qt::Window` flag to the `QWidget` constructor

- Use `setWindowModality` to make modal
  - `NonModal` – all windows can be used at once
  - `WindowModal` – the parent window is blocked
  - `ApplicationModal` – all other windows are blocked

Window Properties

- Set the window title using `setWindowTitle`
- The `QWidget` constructor and window flags

```
QWidget::QWidget(QWidget *parent, Qt::WindowFlags f=0)
```

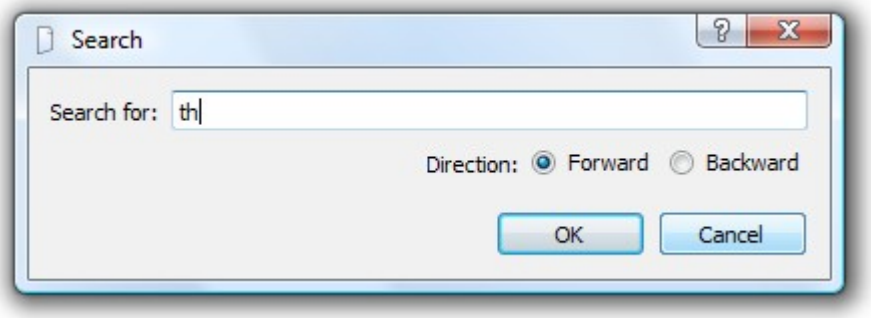
  - `Qt::Window` – creates a window
  - `Qt::CustomizeWindowHint` – clear defaults
    - `Qt::WindowMinimizeButtonHint`
    - `Qt::WindowMaximizeButtonHint`
    - `Qt::WindowCloseButtonHint`
    - etc

The word ***hint*** is important

Different platforms and window managers affect the effect of these settings

Using QDialog

- A search dialog is a typical custom dialog



- Inherited from `QDialog`
- User interface created using Designer or code
  - `QLabel` and `QRadioButton` are “outputs”
  - Buttons for accepting or rejecting

The Programming Interface

```
class SearchDialog : public QDialog
{
    Q_OBJECT
public:
    explicit SearchDialog(const QString &initialText,
                          bool isBackward, QWidget *parent = 0);

    bool isBackward() const;
    const QString &searchText() const;

private:
    Ui::SearchDialog *ui;
};
```

Initialize the dialog in the constructor

Getter functions for clean access of data

The Implementation

```
SearchDialog::SearchDialog(const QString &initialText,
                           bool isBackward, QWidget *parent) :
    QDialog(parent), ui(new Ui::SearchDialog)
{
    ui->setupUi(this);

    ui->searchText->setText(initialText);
    if(isBackward)
        ui->directionBackward->setChecked(true);
    else
        ui->directionForward->setChecked(true);
}

bool SearchDialog::isBackward() const
{
    return ui->directionBackward->isChecked();
}

const QString &SearchDialog::searchText() const
{
    return ui->searchText->text();
}
```

Initialize dialog according to settings

Getter functions

Using the Dialog

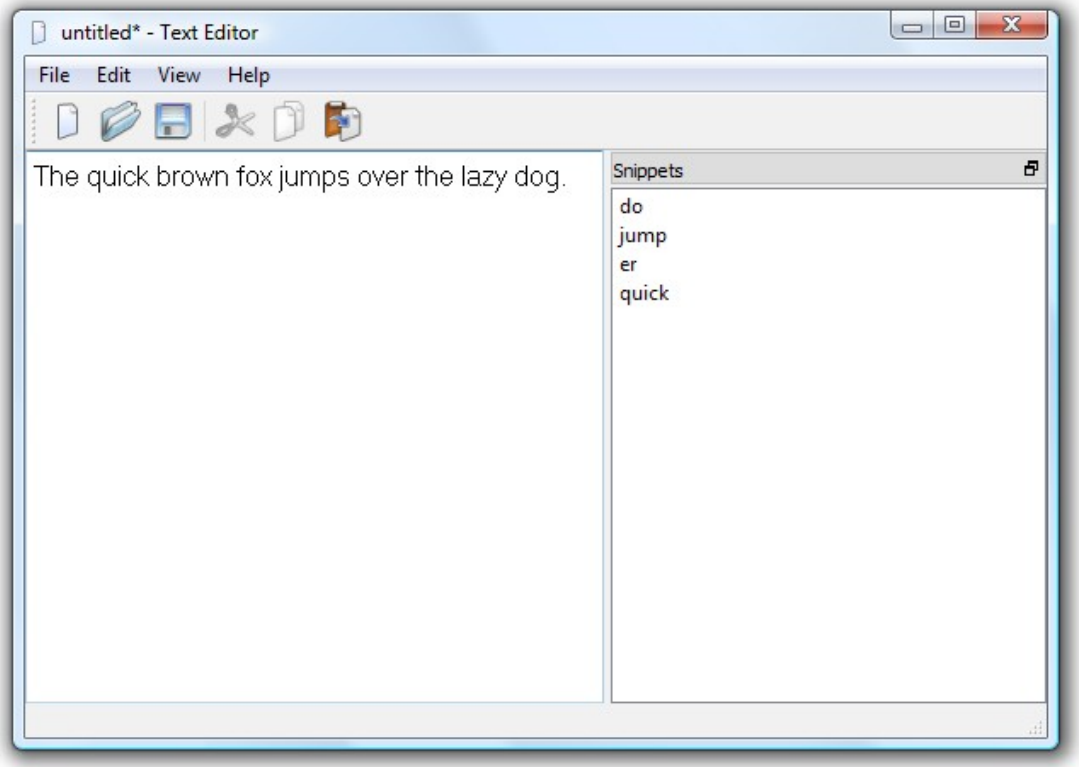
- The software interface has been defined to make it easy to use the dialog

```
void MyWindow::myFunction()
{
    SearchDialog dlg(settings.value("searchText","").toString(),
                     settings.value("searchBackward", false).toBool(), this);

    if(dlg.exec() == QDialog::Accepted)
    {
        QString text = dlg.searchText());
        bool backwards = dlg.isBackward());
        ...
    }
}
```

`QDialog::exec` shows a modal (blocking) dialog and returns the result as accepted or rejected

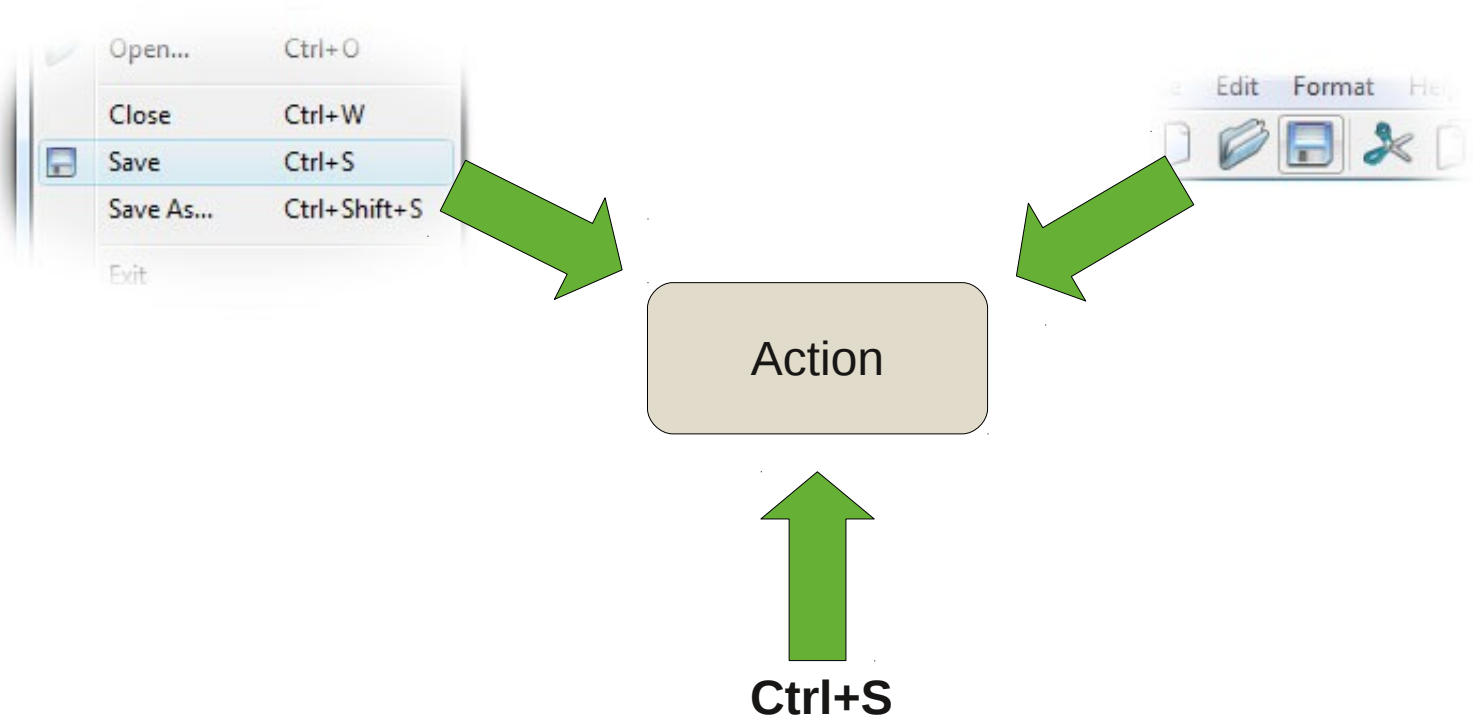
Using QMainWindow

- A `QMainWindow` is the document window of the average desktop application
  - Menus
  - Toolbar
  - Statusbar
  - Docks
  - Central widget



Introducing QAction

- Many user interface elements refer to the same user action



- A QAction object can represent all these access ways – and hold tool tips, statusbar hints, etc too

Introducing QAction

- A `QAction` encapsulates all settings needed for menus, tool bars and keyboard shortcuts
- Commonly used properties are
  - `text` – the text used everywhere
  - `icon` – icon to be used everywhere
  - `shortcut` – shortcut
  - `checkable`/`checked` – whether the action is checkable and the current check status
  - `toolTip`/`statusTip` – tips text for tool tips (hover and wait) and status bar tips (hover, no wait)

Introducing QAction

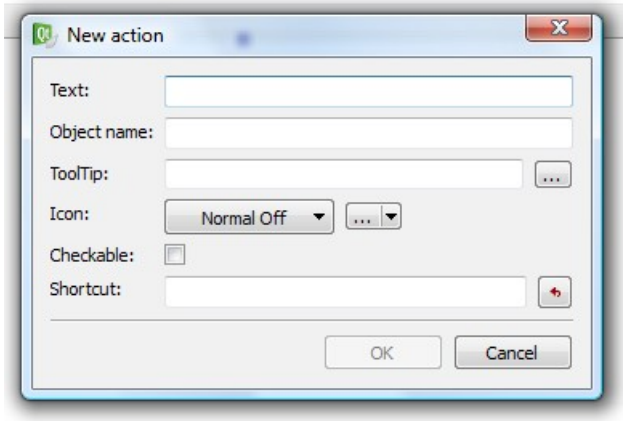
```
QAction *action = new QAction(parent);

action->setText("text");

action->setIcon(QIcon(":/icons/icon.png"));

action->setShortcut(QKeySequence("Ctrl+G"));


action->setData(myDataQVariant);
```

Creating a new action

Setting properties for text, icon and keyboard short-cut

A QVariant can be associated with each action, to carry data associated with the given operation

- Or use the editor in Designer

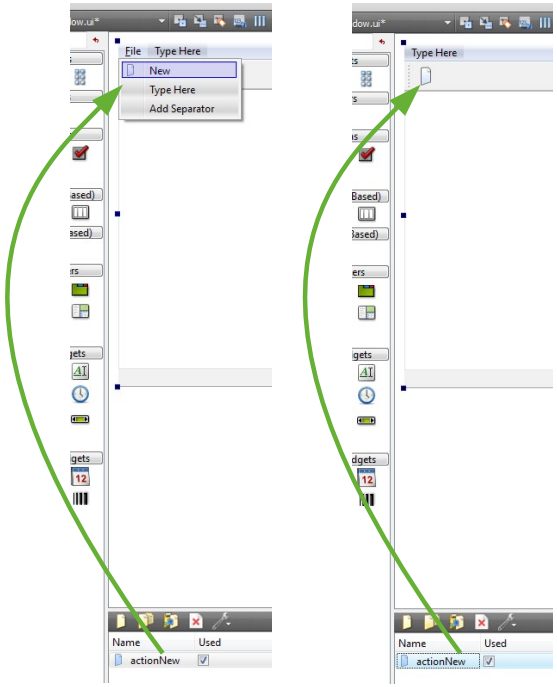


Adding actions

- Adding actions to different parts of the user interface is as easy as calling add Action

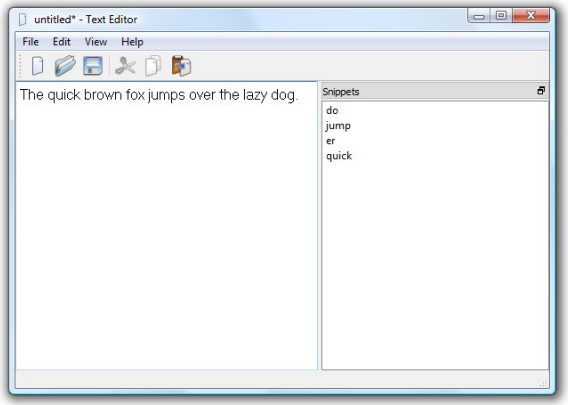
```
myMenu->addAction(action);
myToolBar->addAction(action);
```

- In Designer, simply drag and drop each action into place on a tool bar or menu

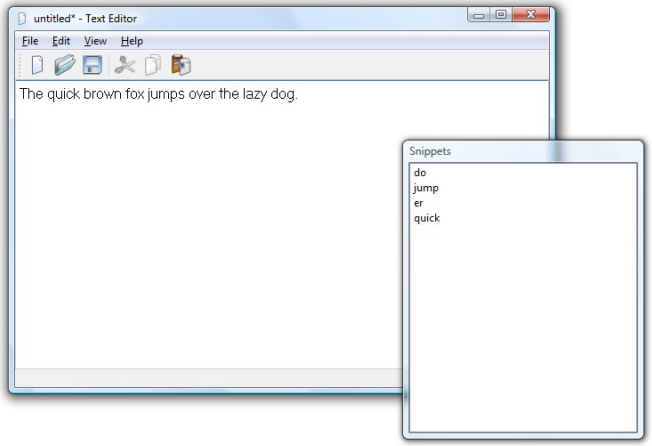


Dock widgets

- Dock widgets are detachable widgets placed around the edges of a `QMainWindow`
  - Great for multi-head setups



- Simply place your widget inside a `QDockWidget`
- `QMainWindow::addDockWidget` adds the docks to the window



Dock widgets

```
void MainWindow::createDock()
{
    QDockWidget *dock = new QDockWidget("Dock", this);

    dock->setFeatures(QDockWidget::DockWidgetMovable |
                      QDockWidget::DockWidgetFloatable);
    dock->setAllowedAreas(Qt::LeftDockWidgetArea |
                          Qt::RightDockWidgetArea);
    dock->setWidget(actualWidget);

    ...

    addDockWidget(Qt::RightDockWidgetArea, dock);
}
```

A new dock with a title

Can be moved and floated (not closed!)

The actual widget is what the user interacts with

Can be docked along the sides

Finally, add it to the window

Icon resources

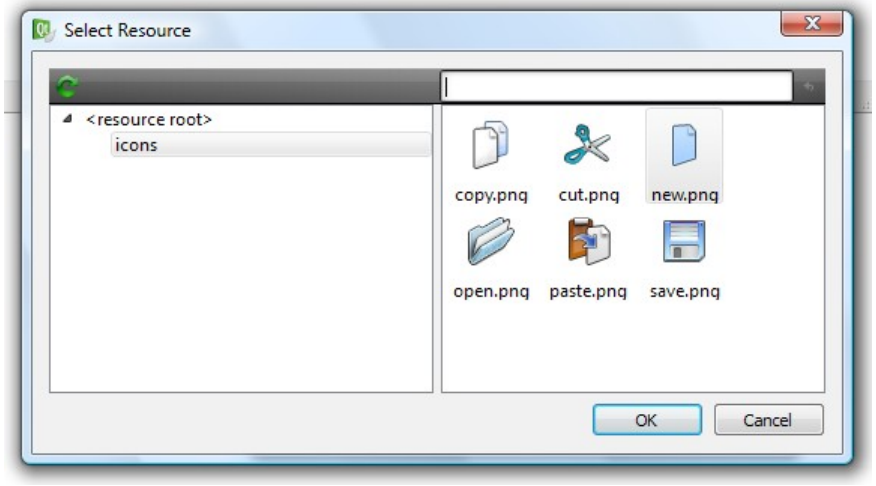
- Putting icons in a resource file lets Qt embed them into the executable
  - Avoid having to deploy multiple files
  - No need to try to determine the path for the icons for each specific install type
  - All fits neatly into the build system
  - ...
  - You can add anything into resources, not only icons

Icon resources

- You can easily manage resource files in Qt Creator
- Prefix path and filenames with `:` to use a resource

```
QPixmap pm(":/images/logo.png");
```

- Or simply pick an icon from the list in Designer



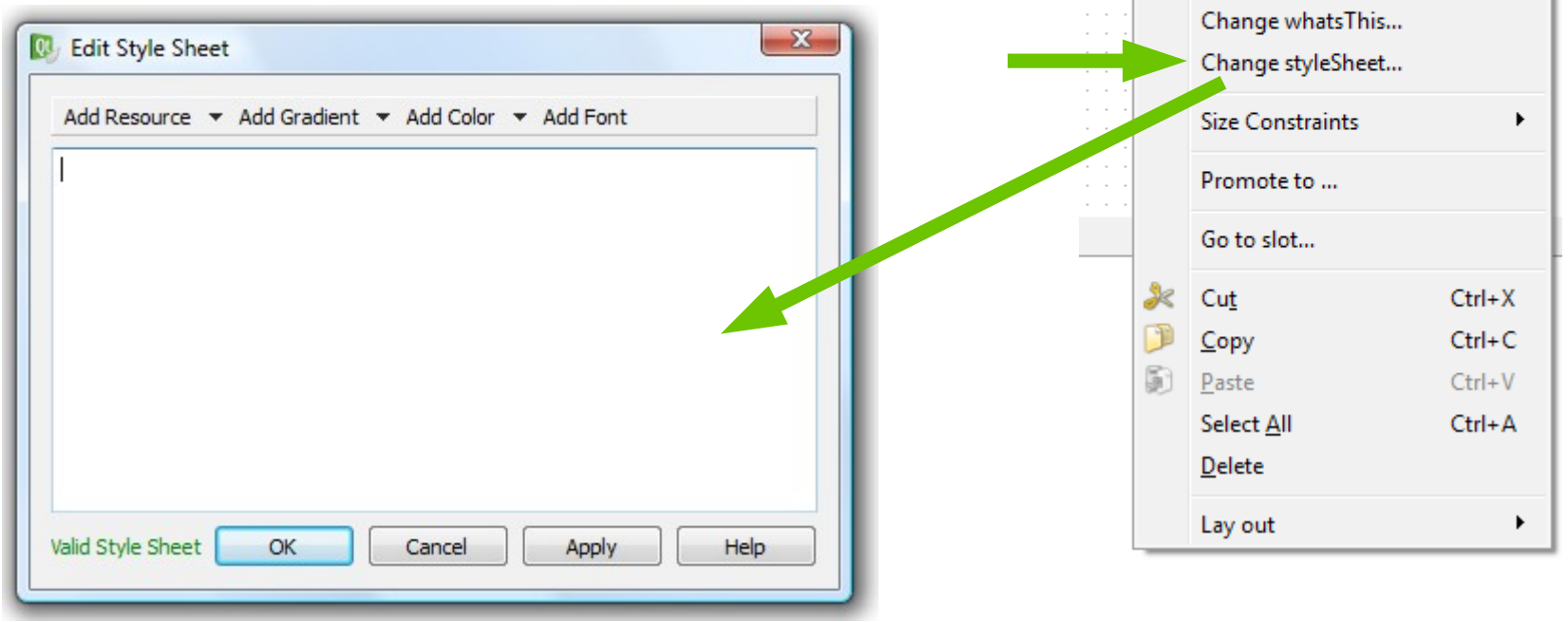
Style sheets

- For highlighting and cross platform styling, all `QWidget` classes have a `styleSheet` property
- Style sheets are inspired from CSS
- They can be used for highlighting and for various small alternations

- As well as a total overhaul of the entire user interface

Style sheets

- The easiest way to apply a style sheet to an individual widget is to use Designer



Stylesheet

- To style an entire application, use `QApplication::setStyleSheet`

Select a class

Select an object by name

Use images

Build these in Designer's editor

```
QLineEdit { background-color: yellow }
QLineEdit#nameEdit { background-color: yellow }

xtEdit, QListView {
    background-color: white;
    background-image: url(draft.png);
    background-attachment: scroll;
}

QGroupBox {
    background-color: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
                                      stop: 0 #E0E0E0, stop: 1 #FFFFFF);
    border: 2px solid gray;
    border-radius: 5px;
    margin-top: 1ex;
}
```