User Interface (UI) designs

Widgets are the primary elements for creating user interfaces. They can display data and status information, receive user input, and provide a container for other widgets that should be grouped together. A widget that is not embedded in a parent widget is called a window.
There are various ways that widgets are used to create UI. Here we describe the one used in Qt, implemented as a class called Qwidget.
Usually, windows have a frame and a title bar, although it is also possible to create windows without such decoration using suitable window flags. In Qt, QMainWindow and the various subclasses of QDialog are the most common window types.
In applications, windows provide the screen space upon which the user interface is built. Windows separate applications visually from each other and usually provide a window decoration that allows you to resize and position the applications according to your preferences. Windows are typically integrated into the desktop environment and to some degree managed by the window management system that the desktop environment provides. For instance, selected windows of an application are represented in the task bar.
Qt Designer UI files represent the widget tree of the form in XML format. {XML stands for eXtensible Markup Language. It was designed to store and transport data. It is designed to be both human- and machine-readable.} The forms can be processed:

- At compile time, which means that forms are converted to C++ code that can be compiled.
- At runtime, which means that forms are processed by the QUiLoader class that dynamically constructs the widget tree while parsing the XML file.

Compile time form processing

You create user interface components with *Qt Designer* and use Qt's integrated build tools, qmake and uic, to generate code for them when the application is built. The generated code contains the form's user interface object. It is a C++ struct that contains:

- Pointers to the form's widgets, layouts, layout items, button groups, and actions.
- A member function called `setupUi()` to build the widget tree on the parent widget.
- A member function called `retranslateUi()` that handles the translation of the string properties of the form. For more information, see Reacting to Language Changes.

Main Windows and dialogs

The Application Main Window provides the framework for building the application's main user interface and are created by subclassing QMainWindow. QMainWindow has its own layout to which you can add a menu bar, tool bars, dockable widgets and a status bar. The center area can be occupied by any kind of QWidget.

Dialog Windows are used as secondary windows that present you with options and choices. Dialogs are created by subclassing QDialog and using widgets and layouts to implement the user interface. In addition, Qt provides a number of ready-made standard dialogs that can be used for standard tasks like file or font selection.

Both main windows and dialogs can be created with Qt Designer, Qt's visual design tool. Using Qt Designer is a lot faster than hand-coding, and makes it easy to test different design ideas. Creating designs visually and reading the code generated by uic is a great way to learn Qt!

Widgets

Here, widgets are dealt with in greater details.

Widgets are the basic building blocks for graphical user interface (GUI) applications built with Qt. Each GUI component (e.g. buttons, labels, text editors) is a widget that is placed somewhere within a user interface window, or is displayed as an independent window. Each type of widget is provided by a subclass of QWidget, which is itself a subclass of Qobject.

Many of the GUI examples provided with Qt follow the pattern of having a `main.cpp` file, which contains the standard code to initialize the application, plus any number of other source/header files that contain the application logic and custom GUI components.

```
#include <QtWidgets>

// Include header files for application components.
// ...

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    // Set up and show widgets.
    // ...

    return app.exec();
}
```

First, a *QApplication* object is constructed, which can be configured with arguments passed in from the command line. After the widgets have been created and shown, *QApplication::exec()* is called to start Qt's event loop. Control passes to Qt until this function returns. Finally, `main()` returns the value returned by *QApplication::exec()*.

Example 1 – creating a window

New project → Application(Qt) → Qt Widgets Application

modify *main.cpp* as follows:

```
#include <QtWidgets>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.resize(320, 240);
    window.show();
    window.setWindowTitle(
        QApplication::translate("toplevel", "Top-level widget"));
    return app.exec();
}
```

Example 2 – add a child widget

```
#include <QtWidgets>
```

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.resize(320, 240);
    window.setWindowTitle
          (QApplication::translate("childwidget", "Child widget"));
    window.show();

    QPushButton *button = new QPushButton(
        QApplication::translate("childwidget", "Press me"), &window);
    button->move(100, 100);
    button->show();
    return app.exec();
}
```

The button is now a child of the window and will be deleted when the window is destroyed. Note that hiding or closing the window does not automatically destroy it. It will be destroyed when the example exits.

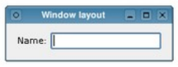
Example 3 – use of layout

Usually, child widgets are arranged inside a window using layout objects rather than by specifying positions and sizes explicitly. Here, we construct a label and line edit widget that we would like to arrange side-by-side.

```
#include <QtWidgets>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    QLabel *label = new QLabel(QApplication::translate("windowlayout", "Name:"));
    QLineEdit *lineEdit = new QLineEdit();

    QHBoxLayout *layout = new QHBoxLayout();
    layout->addWidget(label);
    layout->addWidget(lineEdit);
    window.setLayout(layout);
    window.setWindowTitle(
        QApplication::translate("windowlayout", "Window layout"));
    window.show();
    return app.exec();
}
```



The `layout` object we construct manages the positions and sizes of widgets supplied to it with the *addWidget()* function. The layout itself is supplied to the window itself in the call to *setLayout()*.

Layouts are only visible through the effects they have on the widgets (and other layouts) they are responsible for managing.

In the example above, the ownership of each widget is not immediately clear. Since we construct the widgets and the layout without parent objects, we would expect to see an empty window and two separate windows containing a label and a line edit. However, when we tell the layout to manage the label and line edit and set the layout on the window, both the widgets and the layout itself are "reparented" to become children of the window.

Signals and Slots

In GUI programming, when we change one widget, we often want another widget to be notified. More generally, we want objects of any kind to be able to communicate with one another. For example, if a user clicks a **Close** button, we probably want the window's close() function to be called.

Other toolkits achieve this kind of communication using **callbacks**. A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback when appropriate. While successful frameworks using this method do exist, callbacks can be unintuitive and may suffer from problems in ensuring the type-correctness of callback arguments.

In Qt, there is an alternative to the callback technique: Use signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but we can always subclass widgets to add our own signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to subclass widgets and add your own slots so that you can handle the signals that you are interested in.

The signals and slots mechanism is type safe: The signature of a signal must match the signature of the receiving slot. (In fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments.) Since the signatures are compatible, the compiler can help us detect type mismatches when using the function pointer-based syntax. The string-based SIGNAL and SLOT syntax will detect type mismatches at runtime. Signals and slots are loosely coupled: A class which emits a signal neither knows nor cares which slots receive the signal. Qt's signals and slots mechanism ensures that if you connect a signal to a slot, the slot will be called with the signal's parameters at the right time. Signals and slots can take any number of arguments of any type. They are completely type safe.