# H62PEP Computing - Worksheet 5 - Graphical User Interfaces and Qt

P. Evans

## Overview

You will learn to create a Graphical User Interface (GUI) using Qt. You will create an application that will use your File I/O and Vector Maths libraries from the previous project week and this will form the basis of the final software deliverable.

## Developing Software with A Graphical User Interface (GUI)

Modern operating system have a feature called a *Window Manager* that allows applications running on the OS to have a graphical representation in the form of a window. The Window Manager manages how these windows interact with the user - through the display screen and through inputs such as keyboard button presses and mouse movement. A software application with a GUI must interact with the operating system, through the Window Manager, in two ways:

- ***Describe the appearance and behaviour of its Window(s)***. Usually using range of standard user interface components such as *buttons*, *menus*, *toolbars*, *check boxes*, etc. These components are sometimes called *Widgets*.
- ***Provide code to handle user actions***. Window Managers usually deal with user input using a message system. For example: if someone clicks a mouse in your application Window, a message will be generated that states which mouse button was clicked and the location of the mouse when it was clicked. If the mouse click also caused a user interface button to be pressed, additional messages saying that the button was pressed will also be generated. You program must react by providing code that is executed when particular messages are received. User interface components (e.g. *windows*, *buttons*, *menus*, *toolbars*, *check boxes*, …) will all generate lots of different messages according to the user's actions, but a GUI application usually only needs to idenitfy and act on a subset of these.

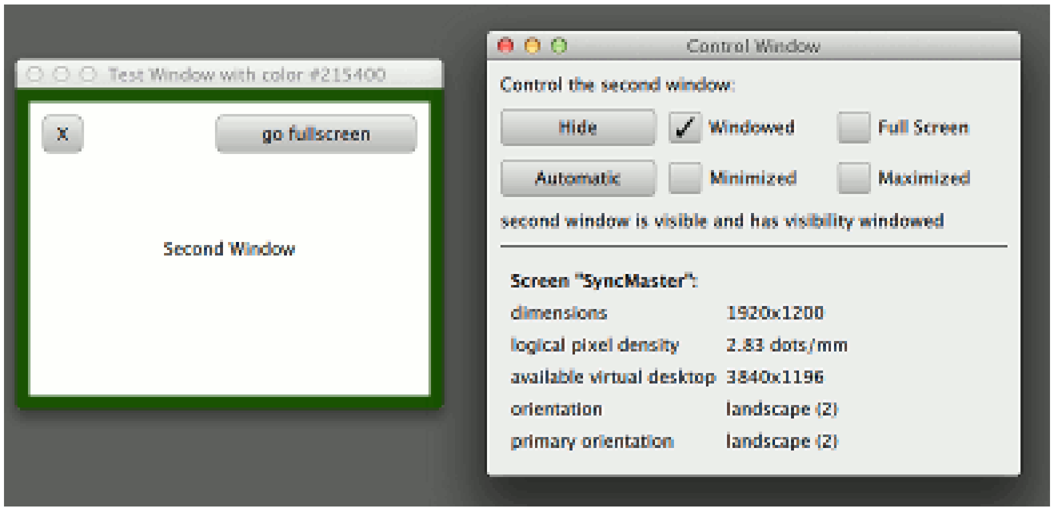
The Window Manager is typically a component of the operating system itself and GUI software applications that interact with it are developed using a set of software libraries that allow implementation of the two key functions described above. Usually these libraries are called something like *User Interface Toolkits* or *User Interface SDKs (Software Development Kits)* and can be accessed from one or more program languages (a mechanism that allows a library to be accessed by a particular programming language is called a *binding*).

There are many user interface toolkit libraries, see this list on Wikipedia to get an idea of how many. Some of these will work with only one operating system - for example Microsoft provides a number of different Windows-specific libraries. Obviously, to develop cross-platform software it is important to choose a toolkit that is capable of interacting with the Window Manager on a range of operating systems and that's what you will do as part of this project. The toolkit you'll be using is called *Qt*.

## Qt

### What is Qt?

Qt is a set of libraries that allows you to create cross-platform GUI applications with common GUI components like windows, buttons, menus, file open dialogs.



Control the second window:

[✓ Windowed] Full Screen Minimized Maximized

second window is visible and has visibility windowed

Screen "SyncMaster":

| | |
|---|---|
| dimensions | 1920x1200 |
| logical pixel density | 2.83 dots/mm |
| available virtual desktop | 3840x1196 |
| orientation | landscape (2) |
| primary orientation | landscape (2) |

Figure 1: Qt Windows

Qt is a form of *Abstraction Layer* - rather than working with *native* Microsoft Windows, Linux or Apple toolkit libraries to develop your GUI, you work with the Qt libraries. Qt then acts as the interface to the operating system on which the program runs, ensuring that your abstact, operating system independent code can successfully interact with a variety of operating systems and their Window Managers. The final executable will still be operating system specific, but Qt allows you to compile the *same code* for *different OSs*.

Qt is a set of static and dynamic libraries that your program must link against at compile-time and run-time, the default Binding for Qt is to C++. When linking a program, all libraries should usually be generated with the same compiler that is used to compile the final program as this will guarantee compatibility. For example when compiling with 64-bit MinGW the libraries you use should also have been generated with 64-bit MinGW, if you try to use libraries generated with 64-bit MS Visual C++ there is no guarantee that linking will be successful. If you try to link against 64-bit libraries when compiling a 32-bit executable, or vice-versa, it will definitely be unsuccessful. Because of this, the Qt libraries must be compiler and operating system specific and so a range of precompiled Qt libraries are available for download.

First you will need to install these Qt development libraries, if you don't have these on your computer after the last project week the follow these steps: Visit the Qt download page and download Open Source Qt for desktop applications. Note you don't need to register for an account when prompted. As part of the install process you will be prompted to choose which compilers you want Qt to support. Make sure you have selected MinGW (Qt does not support MinGW-w64 with pre-built libraries but Qt will usually install the correct version of 32-bit MinGW as one of its *Tools* components, double check you have selected this option) and Microsoft Visual Studio 64-bit if possible (choose any version(s) available that are compatible with the Visual Studio install on your computer). If you wish to change the libraries you have installed, you can do this using the Qt Maintenance tool that will also be installed.

## The Qt Framework - Compiling Qt Applications

Compiling a Qt based application is the same as compiling any other application that depends on libraries. The correct header files must be included during compilation, and the correct static libraries included during linking.

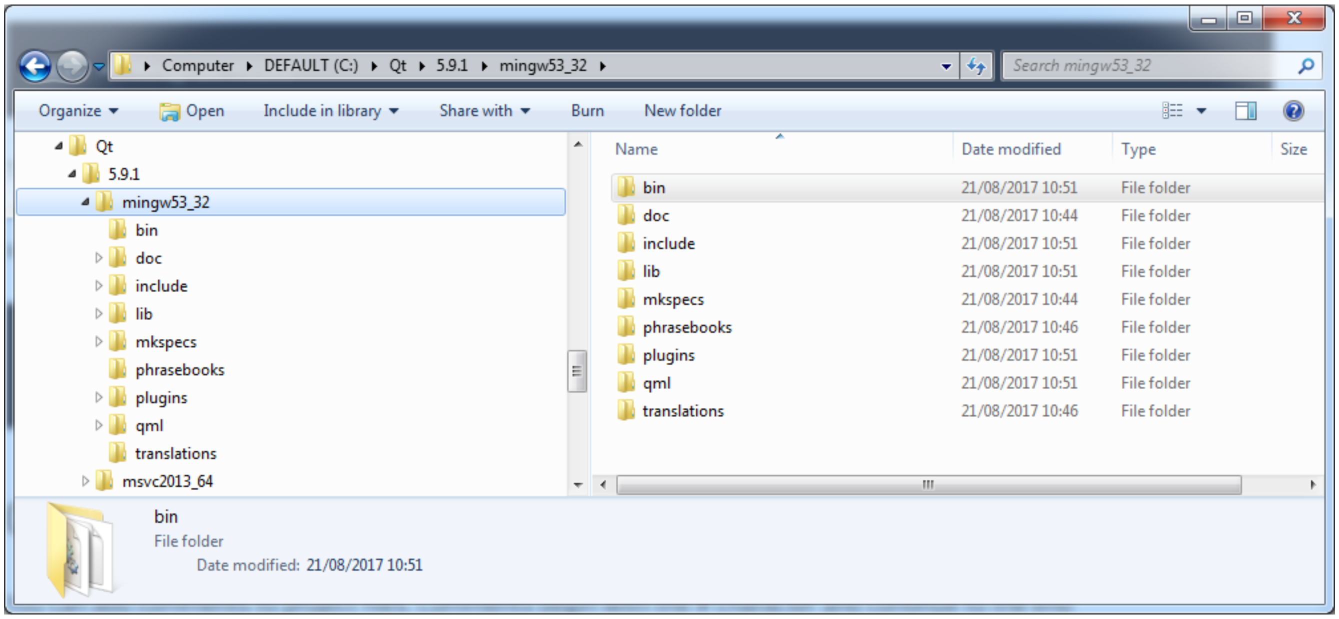
### Exercise 1 - Compiling a Hello World Application

We will create a basic Qt application and use this to understand how Qt works, and how you can compile a Qt application using cmake. First, create a new directory in your individual repository called *hello_qt* and generate the following source file within it.

```
// hello_qt.cpp -------------------------------------------------------
#include <QApplication>
```

```
#include <QWidget>

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

    QWidget window;

    window.resize(250, 150);
    window.setWindowTitle("Simple example");
    window.show();

    return app.exec();
}
// ------------------------------------------------------------------
```

Now you will need to compile this program. For each Qt version, and each compiler you selected during the install, Qt will have created a sub-directory containing both the libraries themselves and also a set of build tools that can be used to create Qt applications.



| Name | Date modified | Type | Size |
|---|---|---|---|
| bin | 21/08/2017 10:51 | File folder | |
| doc | 21/08/2017 10:44 | File folder | |
| include | 21/08/2017 10:51 | File folder | |
| lib | 21/08/2017 10:51 | File folder | |
| mkspecs | 21/08/2017 10:44 | File folder | |
| phrasebooks | 21/08/2017 10:46 | File folder | |
| plugins | 21/08/2017 10:51 | File folder | |
| qml | 21/08/2017 10:51 | File folder | |
| translations | 21/08/2017 10:46 | File folder | |

bin
File folder
Date modified: 21/08/2017 10:51

Figure 2: Qt DirStruc

Within each Qt sub-directory there are a number of further sub-directories that contain the files needed to compile a Qt program. The important ones are:

- **bin** This contains exectuable code including tools that can be used as part of the build process and also dynamic libraries (*.dll*) that will needed at run-time. The *bin* directory for your chosen Qt version and build method will need to be listed in the PATH environment variable to allow the compiled program to run.
- **lib** This contains the Qt static libraries that will be needed for linking.
- **include** This contains the Qt header files that will be needed for compilation. (e.g. the includes *QApplication* and *QWidget* from the code above are located in `include/QtWidgets/`)

To compile the application manually, you also need to ensure the correct compiler is in the system path. For MinGW this will be the version Qt installed for you and will be located in `C:\Qt\Tools\mingwXXX_32\bin`. Be careful if you have other versions of MinGW in your system path variable as it could cause problems. The correct Qt *bin* directory will also need to be in the system path to allow the application you build to execute, remember that the application will need to find the Qt dlls when it runs. Overwriting PATH in the Command Prompt so it contains only the directories you need can be the safest option if multiple versions of MinGW or Qt are installed. You could then issue the compiler and linker commands:

```
>set PATH=C:\Qt\Tools\mingw530_32\bin;C:\Qt\5.9.1\mingw53_32\bin

>g++ -c -fno-keep-inline-dllexport -pipe -O2 -std=gnu++11 -Wextra -Wall
-W -fexceptions -mthreads -DUNICODE -DQT_DEPRECATED_WARNINGS -DQT_NO_DEBUG
-DQT_WIDGETS_LIB -DQT_GUI_LIB -DQT_CORE_LIB -DQT_NEEDS_QMAIN -I. -I.
-IC:\Qt\5.9.1\mingw53_32\include -IC:\Qt\5.9.1\mingw53_32\include\QtWidgets       <- Qt include dirs
-IC:\Qt\5.9.1\mingw53_32\include\QtGui -IC:\Qt\5.9.1\mingw53_32\include\QtANGLE     <- Qt include dirs
-IC:\Qt\5.9.1\mingw53_32\include\QtCore -Irelease                                   <- Qt include dirs
-IC:\Qt\5.9.1\mingw53_32\mkspecs\win32-g++  -o hello_qt.o hello_qt.cpp

>g++ -Wl,-s -Wl,-subsystem,windows -mthreads -o hello_qt.exe hello_qt.o
-lmingw32 -LC:\Qt\5.9.1\mingw53_32\lib C:\Qt\5.9.1\mingw53_32\lib\libqtmain.a       <- link with Qt li
-LC:\utils\my_sql\my_sql\lib -LC:\utils\postgresql\pgsql\lib -lshell32              <- link with Qt li
C:\Qt\5.9.1\mingw53_32\lib\libQt5Widgets.a C:\Qt\5.9.1\mingw53_32\lib\libQt5Gui.a
 C:\Qt\5.9.1\mingw53_32\lib\libQt5Core.a

>hello_qt.exe
```

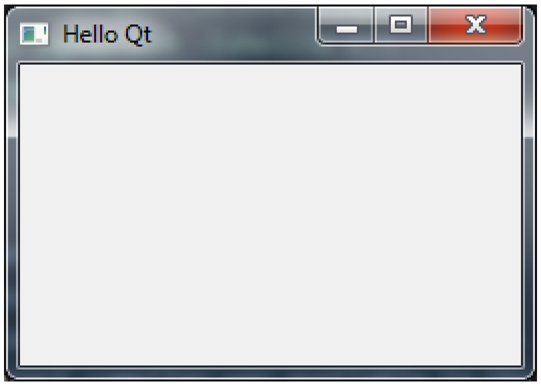


Figure 3: hello_qt

These commands are the same basic compile and link commands that you've seen previously but they also ensure that the correct Qt headers (using `-I`) and libraries (explicitly and using `-L -l`) are included at the relevant stage. If you are not using exactly the same version of Qt and MinGW, you'd need to adapt these commands appropriately - try it if you like.

Constructing these commands by hand is time-consuming and for large projects pretty much impossible (even for the commands shown above, I cheated and had Qt generate them). The solution is to use a build system generator again, but one that is aware of how to compile and link against Qt. One option is to use Qt's internal build system generator *qmake* but for better cross-platform support, and to make work in later project weeks (when we also need to link against the VTK libraries) easier, we will use *cmake* again. *cmake* will determine where the appropriate *bin*, *lib* and *include* directories are, and which libraries are needed, and use these to generate input for our chosen build system and compiler.

### Exercise 2 - Qt and CMake

Now we'll create a CMake project to compile the basic *hello_qt* application. Firstly create a new directory in your repository *hello_qt_cm* and put a copy of *hello_qt.cpp* in it. Now create a *CMakeLists.txt* file that contains the following:

```
# hello_qt_cm/CMakeLists.txt -----------------------------------------
# This is the minimum cmake version needed for Qt
cmake_minimum_required( VERSION 2.8.11 )

# Name of the project
    project( hello_qt )
```

```
# The CMake build process might generate some new files in the current
# directory. This makes sure they can be found.
set( CMAKE_INCLUDE_CURRENT_DIR ON )

# This allows CMake to run one of Qt's build tools called moc
# if it is needed. moc.exe can be found in Qt's bin directory.
# We'll look at what moc does later.
set( CMAKE_AUTOMOC ON )

# Find the Qt widgets package. This locates the relevant include and
# lib directories, and the necessary static libraries for linking.
find_package( Qt5Widgets )

# Same as previously: define the executable and it's sources.
    add_executable( hello_qt hello_qt.cpp )

# Tell CMake that the executable depends on the Qt::Widget libraries.
target_link_libraries( hello_qt Qt5::Widgets )
# ----------------------------------------------------------------------
```

To compile, you need to make sure that the correct cmake, compiler (for mingw32-make), and Qt *bin* directories (this helps CMake successfully find Qt) are in the system path. Its also a good idea to also set CMake's CMAKE_PREFIX_PATH variable with the path to the correct Qt directory using the `-D` option when cmake is called, again to provide a clue to CMake for where the correct Qt libraries are installed. The easiest way to do all of this is to create a batch file that sets everything up and calls CMake each time you need to use it. You can create a small batch script to automate this process.

```
@echo OFF
REM cmake_gen_mingw.bat ----------------------------------------------
REM Make sure this file is in the system path

REM Overwrite the path with minimal path to the correct compiler and Qt tools
REM This avoids problems if multiple (32 and 64bit) versions of MinGW have been installed
set PATH=C:\Program Files\CMake\bin;C:\Qt\Tools\mingw530_32\bin;C:\Qt\5.9.1\mingw53_32\bin

REM Invoke cmake, telling it where the correct version of Qt is located
cmake -DCMAKE_PREFIX_PATH=C:\Qt\5.9.1\mingw53_32 -G"MinGW Makefiles" %1
REM ------------------------------------------------------------------
```

Then run CMake using the batch file to generate MinGW makefiles, compile and link using mingw32-make and test.

```
>mkdir build
>cd build
>cmake_gen_mingw.bat ..
>mingw32-make
>hello_qt.exe
```

## To repeat with Visual Studio

The process is the same for Visual Studio but you need to adapt the batch script to point to an appropriate Qt base directory and setup the path for Visual Studio's compiler. The following are batch scripts to setup 32-bit targets using Visual Studio 2012 and 64-bit targets using Visual Studio 2017 Community Edition. Use these as a starting point to create your own script for the specific version of Visual Studio and Qt that you have on your computer.

**Batch Script Template for Visual Studio 2012, 32bit**

```
@echo OFF
```

```
REM cmake_gen_msvc11.bat ----------------------------------------------
REM Make sure this file is in the system path

REM Overwrite the path with path to the correct compiler and Qt tools
REM Remember the old path so can change back if needed.
set OLD_PATH=%PATH%
set PATH=C:\Program Files\CMake\bin;C:\Qt\5.5\msvc2012\bin

REM Invoke cmake, telling it where the correct version of Qt is located
cmake -DCMAKE_PREFIX_PATH=C:\Qt\5.5\msvc2012 -G"Visual Studio 11 2012" %1

REM Restore old path (otherwise vcvarsall will fail) but make sure Qt bin dir has been added to it
set PATH=%OLD_PATH%;C:\Qt\5.5\msvc2012\bin

REM Setup Compiler Path
"C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat"
REM -----------------------------------------------------------------
```

**Batch Script Template for Visual Studio 2017, 64bit**

```
@echo OFF
REM cmake_gen_msvc15.bat ----------------------------------------------
REM Make sure this file is in the system path

REM Overwrite the path with path to the correct compiler and Qt tools
REM Remember the old path so can change back if needed.
set OLD_PATH=%PATH%
set PATH=C:\Program Files\CMake\bin;C:\Qt\5.9.1\msvc2017_64\bin

REM New vcvarsall.bat will cd to C:\users\exxxx\source unless tell it to user current directory
set VSCMD_START_DIR=%CD%

REM Invoke cmake, telling it where the correct version of Qt is located
cmake -DCMAKE_PREFIX_PATH=C:\Qt\5.9.1\msvc2017_64\bin -G"Visual Studio 15 2017 Win64" %1

REM Restore old path (otherwise vcvarsall will fail) but make sure Qt bin dir has been added to it
set PATH=%OLD_PATH%;C:\Qt\5.9.1\msvc2017_64\bin

REM Setup Compiler Path
"C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\
                                                    Auxiliary\Build\vcvarsall.bat" amd64
REM -----------------------------------------------------------------
```

## Qt Designer for GUI Design

For more advanced applications, creating an attractive and intuitive GUI using code at run-time would be difficult. For this reason Qt provides a user interface design tool that you can use. This tool stores your user interface design as an XML file which is then interpreted by your executable when it runs. These examples demonstrate how you can use this mechanism to develop an GUI application.

The initial worked examples will start by analysing an existing project and then look at adding additional features to learn how to use some of Qt's key features. Finally you will be asked to generate your own custom GUI which can be the starting point for your final software.

**From this point onwards you are free to use whichever compiler you like although it is recommended that you use Visual Studio since this this is what we'll use to initially compile VTK in the next project week.**

**Exercise 3 - Structure of a Typical Qt Program**

Create a new folder *qt_designer* and obtain the source files from https://www.dropbox.com/sh/rytp4o2jkqkmmij/AADdw6dBgKz
CULXAbZiPS_a?dl=0.

```
// qt_designer/main.cpp ------------------------------------------------
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
// ---------------------------------------------------------------------

// qt_designer/mainwindow.cpp ------------------------------------------
#include "mainwindow.h"
#include "ui_mainwindow.h"

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

MainWindow::~MainWindow()
{
    delete ui;
}
// ---------------------------------------------------------------------

// qt_designer/mainwindow.h --------------------------------------------
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget * parent = 0);
    ~MainWindow();

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

```
#endif // MAINWINDOW_H

<!-- mainwindow.ui ------------------------------------------------------->
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow" >
  <property name="geometry" >
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle" >
   <string>MainWindow</string>
  </property>
  <widget class="QMenuBar" name="menuBar" />
  <widget class="QToolBar" name="mainToolBar" />
  <widget class="QWidget" name="centralWidget" />
  <widget class="QStatusBar" name="statusBar" />
 </widget>
 <layoutDefault spacing="6" margin="11" />
 <pixmapfunction></pixmapfunction>
 <resources/>
 <connections/>
</ui>
<!--------------------------------------------------------------------->

# hello_qt_cm/CMakeLists.txt -----------------------------------------
# This is the minimum cmake version needed for Qt
cmake_minimum_required( VERSION 2.8.11 )

# Name of the project
project( qt_ui )

# The CMake build process might generate some new files in the current
# directory. This makes sure they can be found.
set( CMAKE_INCLUDE_CURRENT_DIR ON )

# This allows CMake to run one of Qt's build tools called moc
# if it is needed. moc.exe can be found in Qt's bin directory.
# We'll look at what moc does later.
set( CMAKE_AUTOMOC ON )
set( CMAKE_AUTOUIC ON )

# Find the Qt widgets package. This locates the relevant include and
# lib directories, and the necessary static libraries for linking.
find_package( Qt5Widgets )

# Just for show: This is what find_package has found:
message( STATUS "Qt5Widgets Include Dirs: ${Qt5Widgets_INCLUDE_DIRS}" )
message( STATUS "Qt5Widgets Libraries: ${Qt5Widgets_LIBRARIES}" )

# Same as previously: define the executable and it's sources.
add_executable( hello_qt2 WIN32
                main.cpp mainwindow.cpp
                mainwindow.h
                mainwindow.ui )

# Tell CMake that the executable depends on the Qt::Widget libraries.
target_link_libraries( hello_qt Qt5::Widgets )
# --------------------------------------------------------------------
```

The files you have are:

- **main.cpp** - Contains `main()` - the program's *entry point*. It initialises the Qt application and passes control over to the Qt framework.
- **mainwindow.cpp** and **mainwindow.h**. These implement a C++ class which is linked to the MainWindow user interface widget. Each customised user interface widget in Qt (typically Windows, Dialogs) that you use in your program will have its own corresponding C++ class (and therefore .cpp and .h files) which defines how that widget behaves.
- **mainwindow.ui** This defines the widgets that have been used to construct the user interface and describes how these widgets are arranged.
- **CMakeLists.txt** Basically he same as before but with a couple of additions. Firstly the line `set( CMAKE_AUTOUIC ON )` has been added, this calls the *Qt user interface compiler* which processes any .ui files during the build process. Secondly, all source files (including ui files) are now added as sources for the executable target.

You should now be able to build and run the program:

```
>mkdir build
>cd build
>cmake_gen_mingw.bat ..
>mingw32-make
>hello_qt2
```

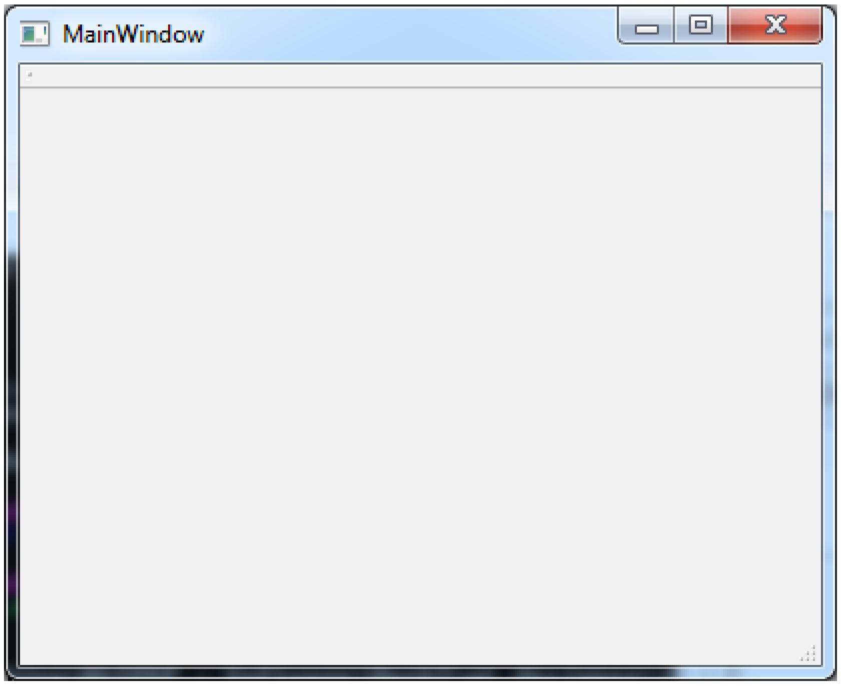


Figure 4: Hello Qt2

The user interface doesnt do anything useful but the basic elements are there: a main window, an empty menubar and toolbar at the top, an empty statusbar at the bottom, and something in between. Now you will gradually build on this to create a small "Database" application that holds a stock list for a shop. You could do this by manually editing the *.ui* file but Qt provides a graphical editor called *Qt Designer* to help. Double click on *mainwindow.ui* in the project directory to open it in *Qt Designer*.

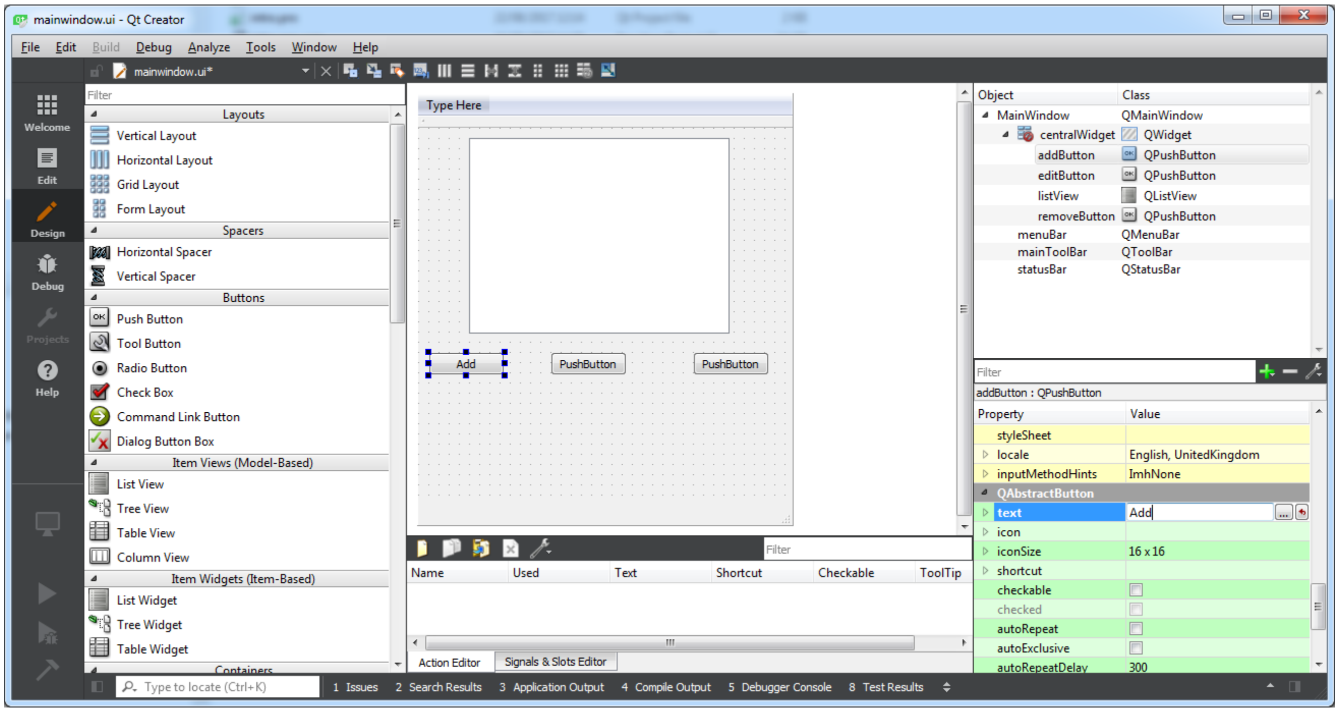
***Note on the User Interface Compiler*** This actually generates C++ code that will create your GUI for you, similar to the code in `main()` of *hello_qt.cpp*. If you look in the *worksheet5/qt_designer/build/hello_qt2_autogen/include* directory you will see a header file called *ui_mainwindow.h*, this file contains the code that was generated by *uic.exe* and is included into your original cpp file.

### Exercise 4 - Setting the basic layout

Now you're going to add the following GUI widgets:

- A ListView to display a list of products
- Buttons to add and remove products from the list

So, to start with you need at add a *List View* (not a *List Widget*) to the user interface and three buttons below it. The buttons will be used to add, edit and remove entries to the product list so change the names of the three buttons (object viewer, top right) to something sensible and change the *text* property of the buttons *QAbstractButton* base class to set the button's visible text (Property viewer, bottom right).



| Object | Class |
|---|---|
| MainWindow | QMainWindow |
| centralWidget | QWidget |
| addButton | QPushButton |
| editButton | QPushButton |
| listView | QListView |
| removeButton | QPushButton |
| menuBar | QMenuBar |
| mainToolBar | QToolBar |
| statusBar | QStatusBar |

addButton : QPushButton

| Property | Value |
|---|---|
| styleSheet | |
| locale | English, UnitedKingdom |
| inputMethodHints | ImhNone |
| QAbstractButton | |
| text | Add |
| icon | |
| iconSize | 16 x 16 |
| shortcut | |
| checkable | ☐ |
| checked | ☐ |
| autoRepeat | ☐ |
| autoExclusive | ☐ |
| autoRepeatDelay | 300 |

Figure 5: Add Buttons

Now you also need to add a *Horizontal Spacer* to the right of the three buttons. This is to take up any additional space that the buttons don't require, otherwise the buttons would expand to fill the window which would look odd.

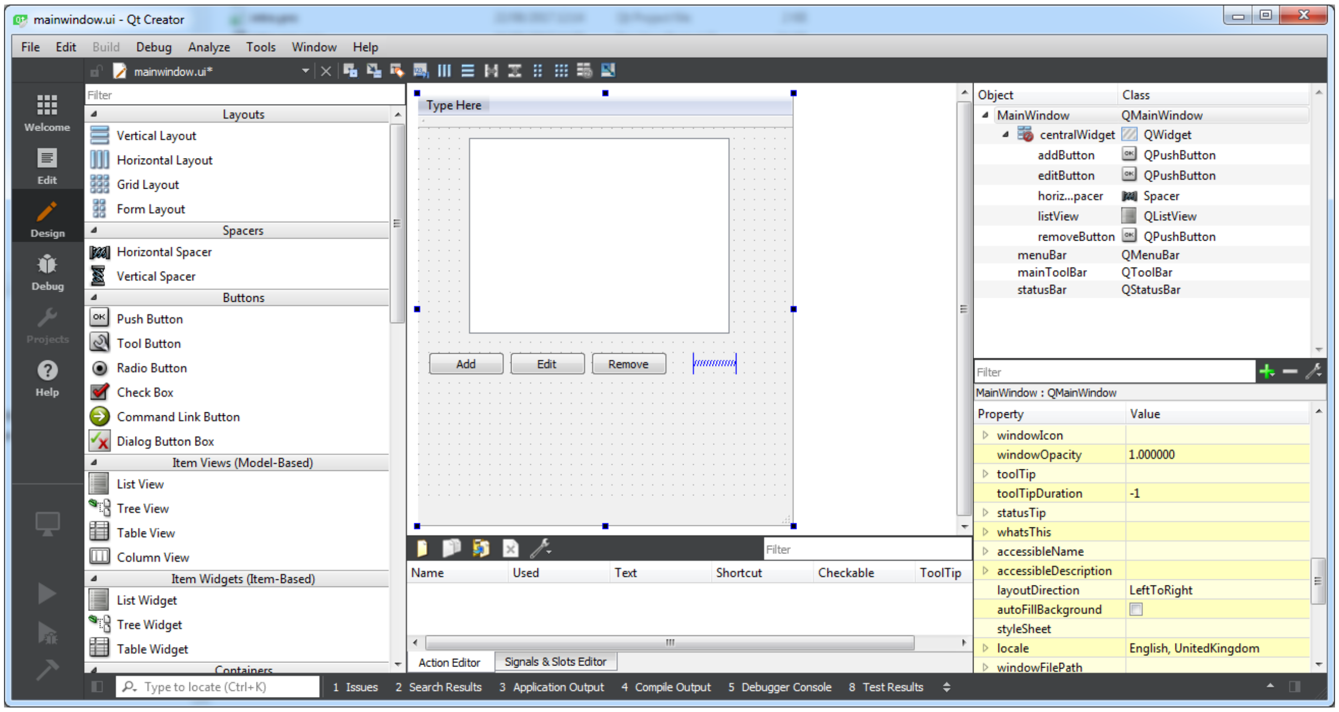
Now assign the buttons and spacer to a *Horizontal Layout*, select all of these widgets and then *Ctrl H* or select *Horizontal Layout* in the top toolbar. Finally select the *centralWidget* widget and assign a vertical layout with *Ctrl L*.

Save mainwindow.ui and rebuild from the command prompt.

```
>mingw32-make
>hello_qt2
```

## Signals and Slots

When a user interacts with a GUI, the operating system generates messages. Many common message types exists such as ButtonClicked messages, MouseButtonDown, MouseButtonReleased, etc and these are then passed to the

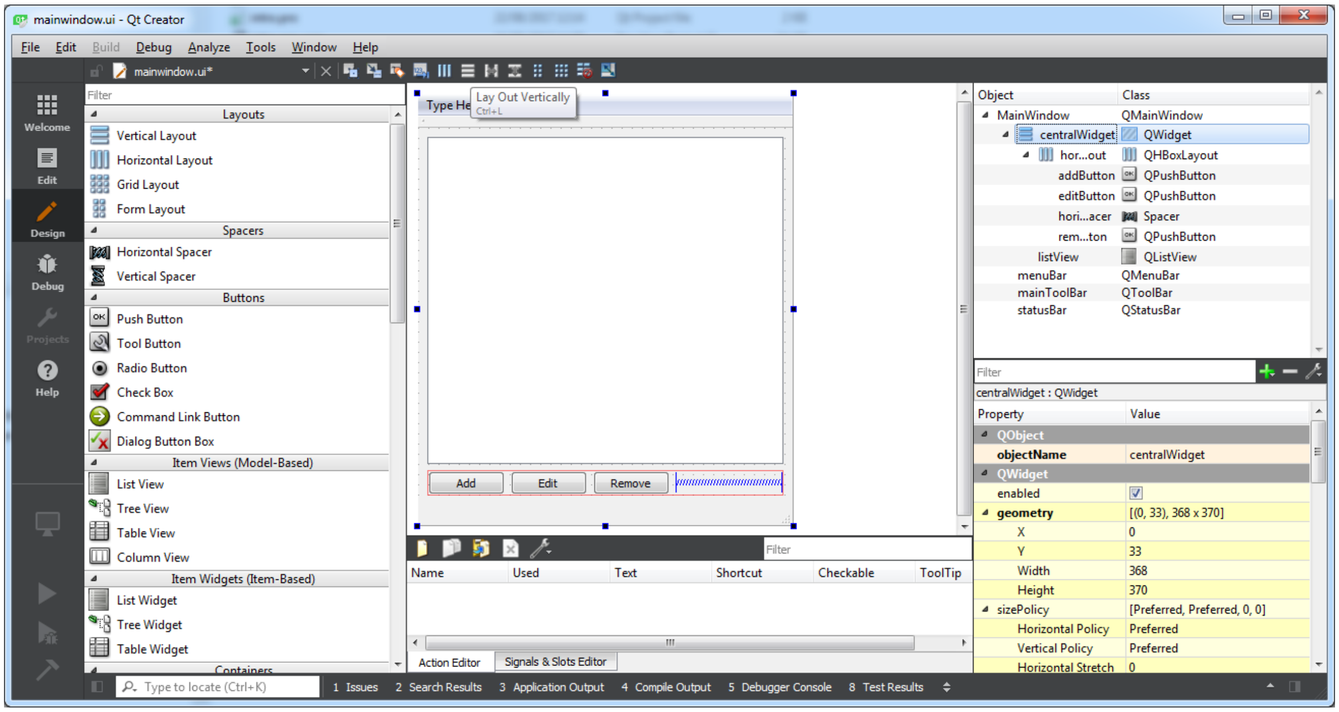


| Object | Class |
|---|---|
| MainWindow | QMainWindow |
| centralWidget | QWidget |
| addButton | QPushButton |
| editButton | QPushButton |
| horiz...pacer | Spacer |
| listView | QListView |
| removeButton | QPushButton |
| menuBar | QMenuBar |
| mainToolBar | QToolBar |
| statusBar | QStatusBar |

MainWindow : QMainWindow

| Property | Value |
|---|---|
| windowIcon | |
| windowOpacity | 1.000000 |
| toolTip | |
| toolTipDuration | -1 |
| statusTip | |
| whatsThis | |
| accessibleName | |
| accessibleDescription | |
| layoutDirection | LeftToRight |
| autoFillBackground | ☐ |
| styleSheet | |
| locale | English, UnitedKingdom |
| windowFilePath | |

Figure 6: Add Buttons



| Object | Class |
|---|---|
| MainWindow | QMainWindow |
| centralWidget | QWidget |
| hor...out | QHBoxLayout |
| addButton | QPushButton |
| editButton | QPushButton |
| hori...acer | Spacer |
| rem...ton | QPushButton |
| listView | QListView |
| menuBar | QMenuBar |
| mainToolBar | QToolBar |
| statusBar | QStatusBar |

centralWidget : QWidget

| Property | Value |
|---|---|
| QObject | |
| objectName | centralWidget |
| QWidget | |
| enabled | ✓ |
| geometry | [(0, 33), 368 x 370] |
| X | 0 |
| Y | 33 |
| Width | 368 |
| Height | 370 |
| sizePolicy | [Preferred, Preferred, 0, 0] |
| Horizontal Policy | Preferred |
| Vertical Policy | Preferred |
| Horizontal Stretch | 0 |

Figure 7: Add Buttons

relevant applications to be interpreted or ignored.

For Qt applications, Qt passes these messages onto your code in the form of *Signals*. By default when most of these signals are *emitted* nothing happens - they are essentially ignored. If you would like something to happen, you need to detect the signal and have it trigger the execution of a handling function - Qt calls these handling functions *slots*. So to get the buttons in the application to do something, you need to:

- Add a *slot* (a special type of function) to the Qt derived class that will recieve the *signal*. This function contains the code that you would like to run. Sometimes you might use a slot that already exists in a Qt Widget class.
- Define a link between the *signal* that will be emitted by the widget class when the button is clicked and the *slot* that you have just created. This link is usually defined in the constructor of the class containing the receiving slot. Sometimes you might also need to create your own signals.

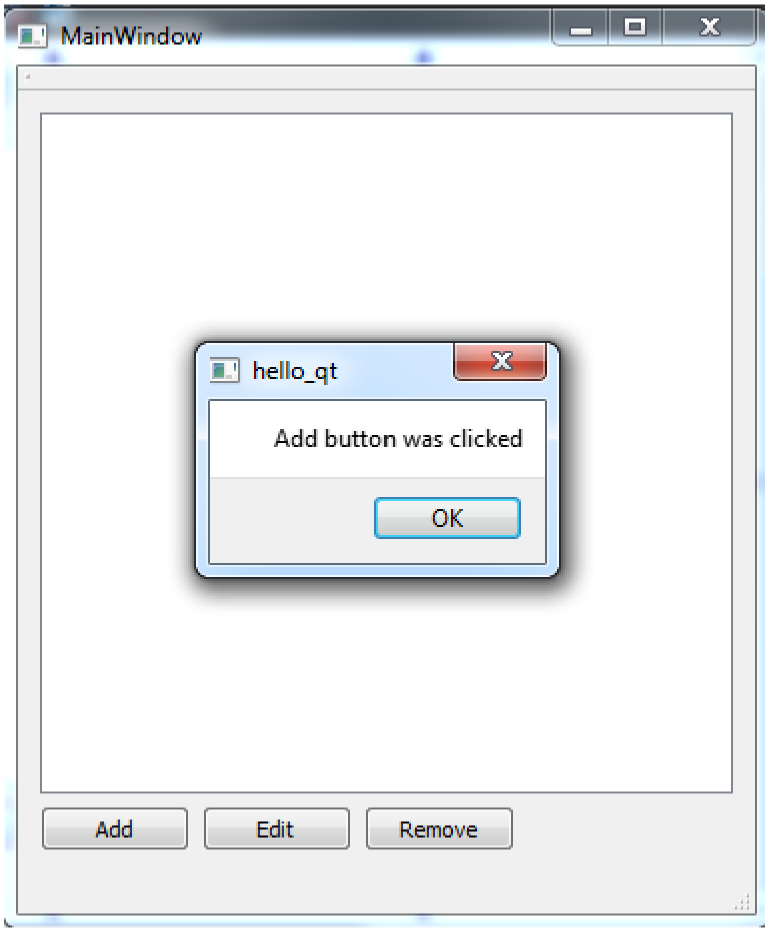
The basics are covered in the examples below, but you can refer to the Qt documentation for more information on signals and slots.

## Exercise 5 - Implementing Slots for Handling Button Clicks

First create three slots within your MainWindow class. You'll need to modify *mainwindow.h* and *mainwindow.cpp*. The slots will display a QMessageBox when the buttons are clicked.

```
// Example of slot definition in mainwindow.h---------------------------
public slots:
        void handleAddButton();
// ------------------------------------------------------------------

// Example of connecting signals and slots in mainwindow.cpp------------
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),ui(new Ui::MainWindow) {
    ui->setupUi(this);

    // Connect the released() signal of the addButton object to the handleAddButton slot in this objec
    connect( ui->addButton, &QPushButton::released, this, &MainWindow::handleAddButton );
}

// Example of slot implementation in mainwindow.cpp---------------------
// (You'll also need to include <QMessageBox>)
void MainWindow::handleAddButton() {
  QMessageBox msgBox;
  msgBox.setText("Add button was clicked");
  msgBox.exec();
}
// ------------------------------------------------------------------
```

Use this example to add slots for each button, rebuild and test. Make sure you include the necessary Qt headers.



***Note on the Meta Object Compiler*** You'll notice that some of the code below isn't strictly C++, code such as Qt's `public slots:` definition. This code is converted to pure C++ by the Qt Meta Object Compiler (*moc.exe*) before the code is compiled. That's why the `set( CMAKE_AUTOMOC ON )` settings is required in CMakeLists.txt. All classes that contain this *meta* code must contain a `Q_OBJECT` tag at the beginning of their declaration to ensure the *moc* is invoked. If you look in *worksheet5/qt_designer/build/hello_qt2_autogen/* you see a file *moc_compilation.cpp* and a sub directory with a seemingly random name, e.g. *EWIEGA46WW* - this is where the additional code generated by *moc.exe* is placed. If you are interested, see here for more information.

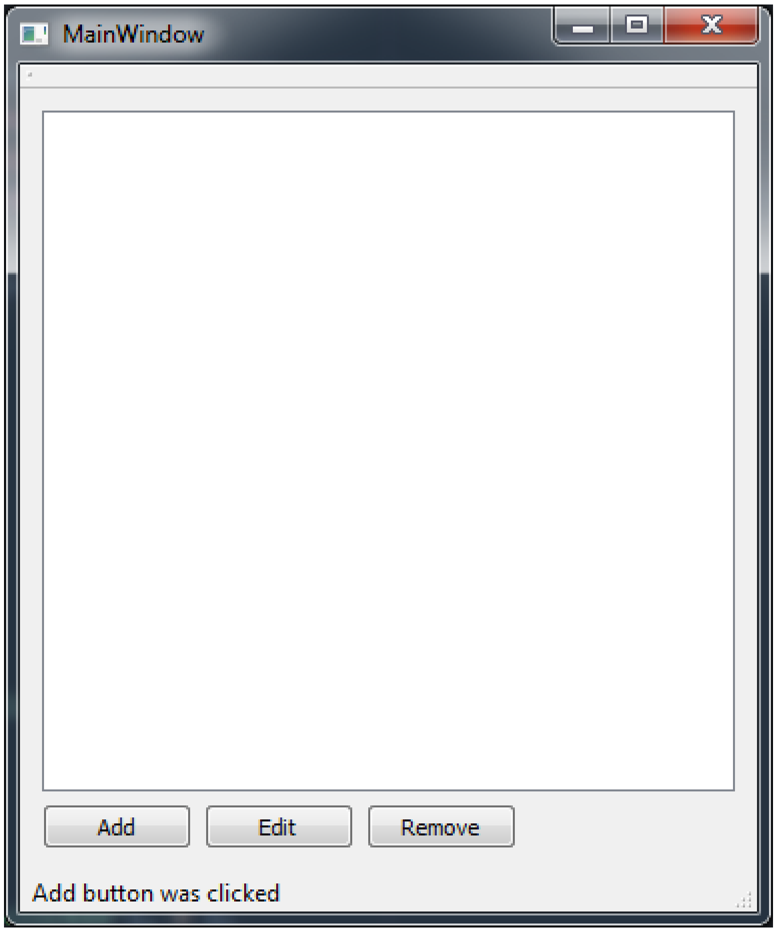
### Exercise 6 - Implementing a Signal to Update the StatusBar

The previous example created a message box each time a button is clicked, but you could also use the application's status bar to deliver this notification. The `QStatusBar` object can be updated through the signal and slot mechanism but to do this, the `MainWindow` object needs to emit a suitable signal and this signal needs to be coupled with the QStatusBar's slot. To determine the format of the signal that QStatusBar is expecting, and the slot to which you should connect it, you can refer to the Qt documentation. An appropriate signal must then be created in the MainWindow object.

```
// Example of signal definition in mainwindow.h-------------------------
signals:
        void statusUpdateMessage( const QString & message, int timeout );
// ---------------------------------------------------------------------

// Example of connecting to StatusBar signal mainwindow.cpp-------------
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent),ui(new Ui::MainWindow) {
    ui->setupUi(this);

    // Your code here to connect button signals ...
```

```
        // Connect the statusUpdateMessage() signal to the showMessage() slot of the status bar
        connect( this, &MainWindow::statusUpdateMessage, ui->statusBar, &QStatusBar::showMessage );
    }

    // Example of slot implementation in mainwindow.cpp---------------------
    void MainWindow::handleAddButton() {
        // This causes MainWindow to emit the signal that will then be
        // received by the statusbar's slot
        emit statusUpdateMessage( QString("Add button was clicked"), 0 );
    }
    // -------------------------------------------------------------------
```

Modify your program so a message appears on the statusbar when each button is clicked, rebuild and test.

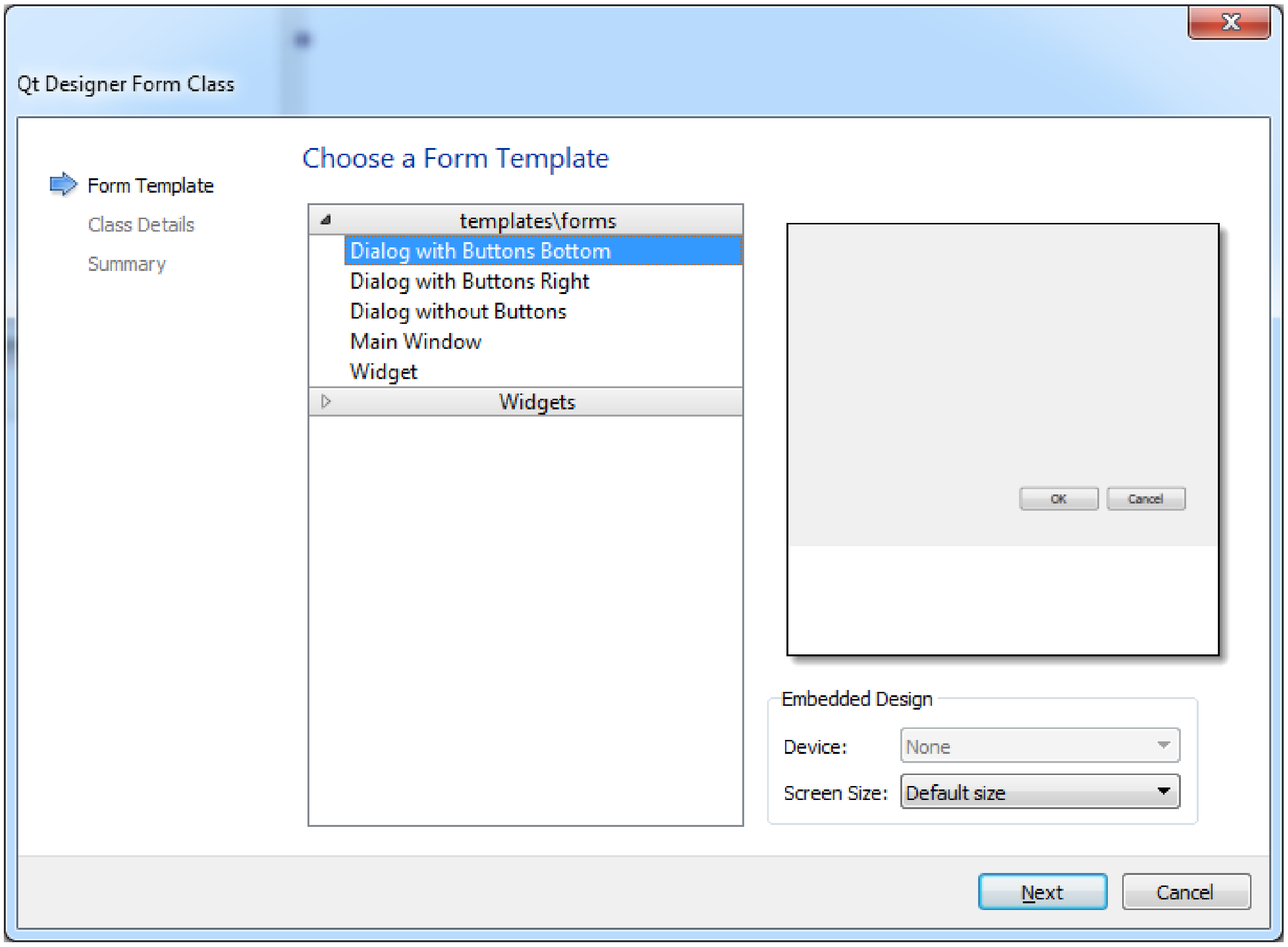


Add button was clicked

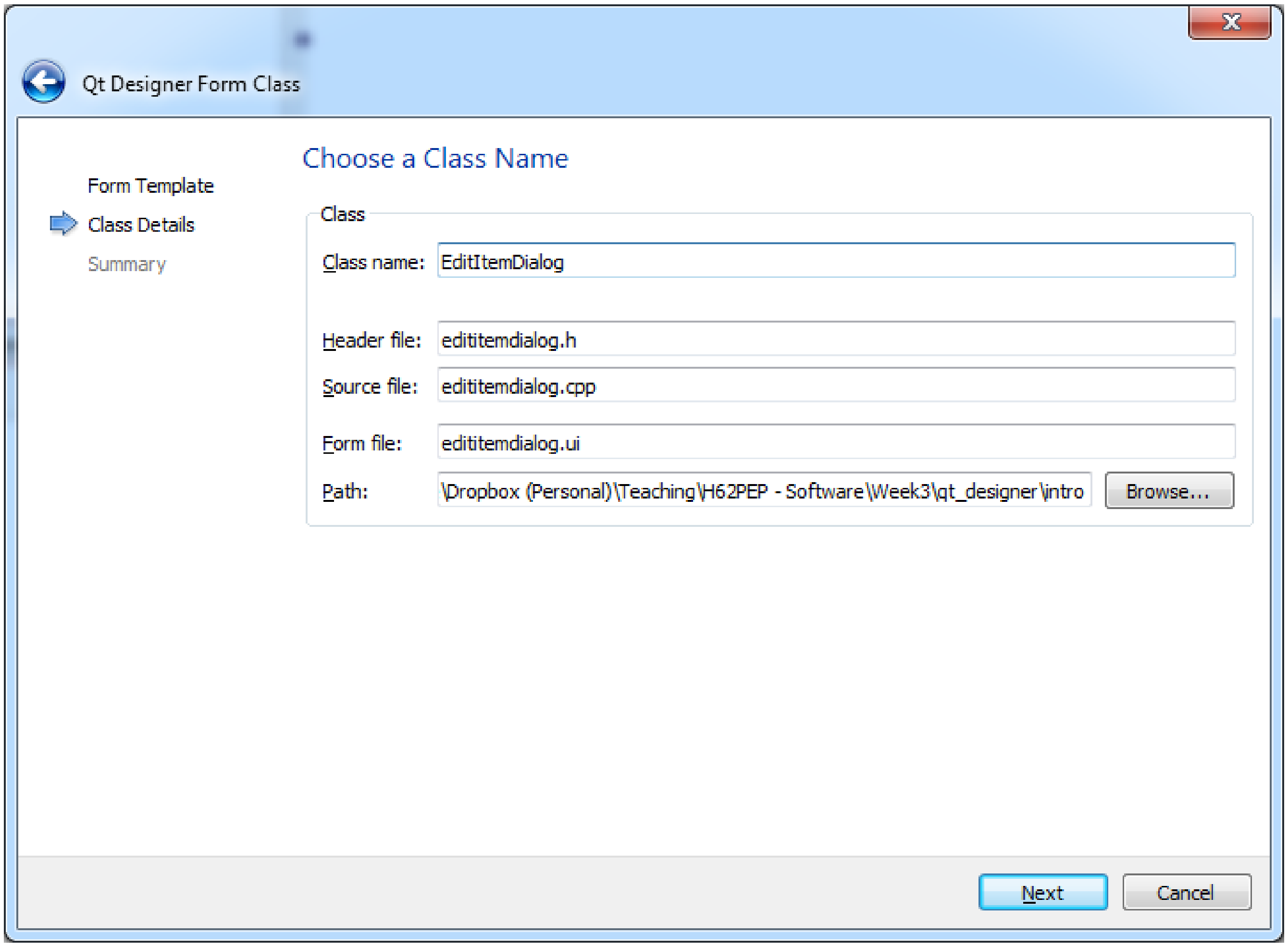
**Exercise 7 - Creating Custom Dialogs (or Widgets)**

When the *edit* button is clicked a dialog should appear to allow the user to edit the stock item that is selected in the list. You now need to create this dialog which will consist of two components:

- A new ***Qt class*** and associated *.cpp* and *.h* source files containing the source code that describes how the dialog behaves. This class will be derived from an appropriate existing Qt base class, in this case `QDialog`.
- A ***.ui file*** that describes the appearance of the new dialog. Qt Designer can be used to design the dialog and to produce templates for the source files. Go to `File->New File or Project` and select `Qt Designer Form Class`.



Choose one of the Dialog options, also notice that you can use this same method to create other custom user interface components - you'll need to use this again later to create a Main Window class and UI for your project. Give the new dialog an appropriate name and ensure the files that are created are in the same directory as your existing Qt code.



Class name: EditItemDialog

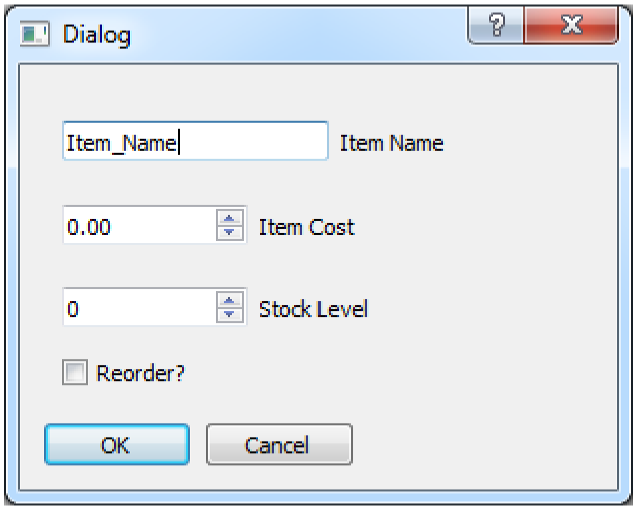
Header file: edititemdialog.h

Source file: edititemdialog.cpp

Form file: edititemdialog.ui

Path: \Dropbox (Personal)\Teaching\H62PEP - Software\Week3\qt_designer\intro

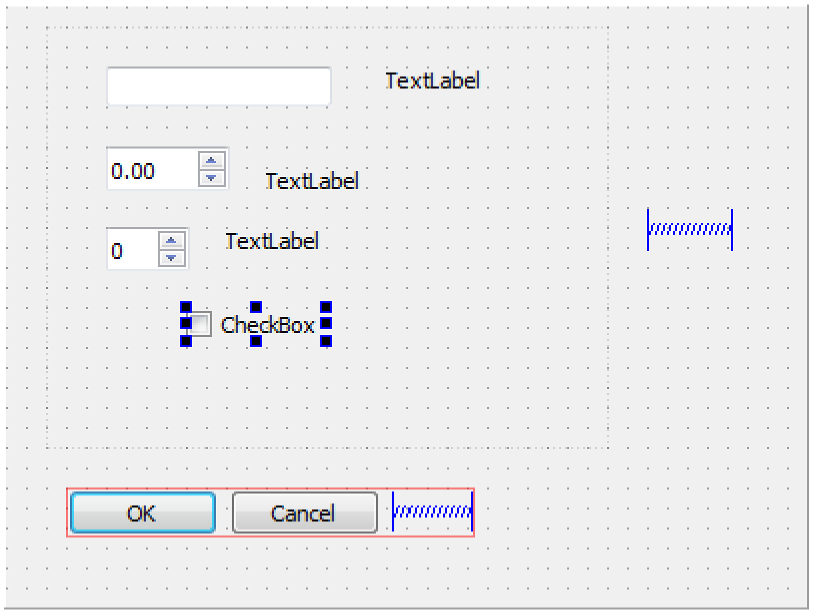
Now you need to design the dialog layout, you're free to do this however you want but an example is included below. The goal is to create a dialog that looks something like this:



The first step is to drag the necessary components onto the blank dialog form. The components used for this example are are:

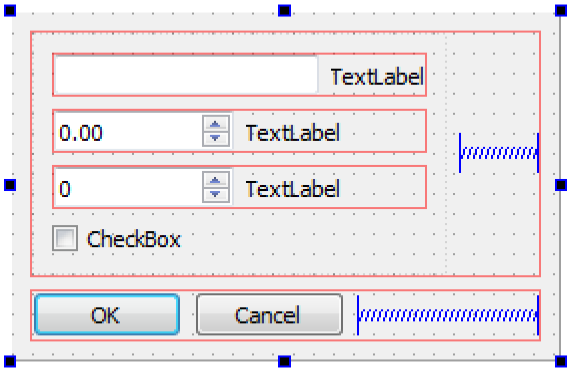
- The Ok/Cancel *ButtonBox* (already provided), plus a *Horizontal Spacer*.
- A *Frame* to contain the input widgets, the following are placed **inside** the *Frame*
  - A *Line Input* and a *Text Label* to enter an items name.
  - A *Double Spin Box* and a *Text Label* for the item's price.

    - A *Spin Box* and a *Text Label* for the stock quantity
    - A *Check Box* to signify if the item should be re-ordered.
- A second *Horizontal Spacer* to space the input widget Frame.



These represent some common user interface components and you're free to add more if you want play with other components if you want - for example numerical values could be represented by a *Slider* or *Dial*. The components must then be organised into a layout, the following sequence of steps are used for the example shown.

- The Ok/Cancel *ButtonBox* and *Horizontal Spacer* are added to a *Horizontal layout*. (Select both, then click Layout Horizontally in the Toolbar)
- The *Line Input* and *Text Label* are added to a *Horizontal layout*.
- The *Double Spin Box* and *Text Label* are added to a *Horizontal layout*.
- The *Spin Box* and *Text Label* are added to a *Horizontal layout*.
- A *Vertical Layout* is applied to the *Frame*. (Click within the Frame to select it, then click Layout Vertically in the Toolbar)
- The *Frame* and second *Horizontal Spacer* are added to a *Horizontal layout*.
- A *Vertical Layout* is applied to the *Dialog* itself.
- The *Dialog* is resized appropriately.

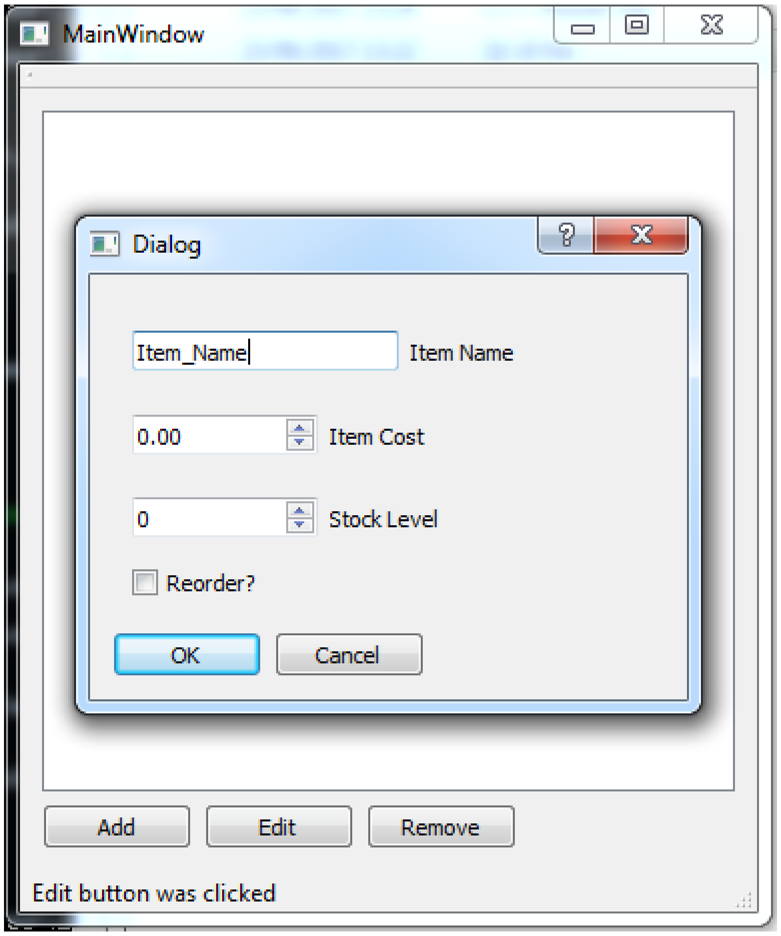


Now replace the label text with some sensible values and give the Line Input, Spin Boxes and Check Box appropriate names to make the code easier to read, and save the files you've just created. Now you'll need to add some code to *edititemdialog.cpp* and also create an additional header file that defines a `StockItem` class so you can store the data that the dialog produces.

```
// stockitem.h ------------------------------------------------------------
```

```cpp
#ifndef STOCK_ITEM_H
#define STOCK_ITEM_H

#include <QString>

class StockItem {
public:
    StockItem() :
                name( "Item_Name" ), unitCost( 0. ), stockLevel( 0 ), reorder( false ) {}
    StockItem( const char * n, double u, int s, bool b ) :
                name( n ), unitCost( u ), stockLevel( s ), reorder( b ) {}

    void setName( const QString & name );
    void setUnitCost( double unitCost );
    void setStockLevel( int stockLevel );
    void setReorder( bool reorder );

    QString getName() const;
    double getUnitCost() const;
    int getStockLevel() const;
    bool getReorder() const;

private:
    QString   name;
    double    unitCost;
    int       stockLevel;
    bool      reorder;
};

#endif
// -----------------------------------------------------------------------

// stockitem.cpp ---------------------------------------------------------
#include "stockitem.h"

void StockItem::setName( const QString & name ) { this->name = name; }
void StockItem::setUnitCost( double unitCost ) { this->unitCost = unitCost; }
void StockItem::setStockLevel( int stockLevel ) { this->stockLevel = stockLevel; }
void StockItem::setReorder( bool reorder ) { this->reorder = reorder; }

QString StockItem::getName() const { return this->name; }
double StockItem::getUnitCost() const { return this->unitCost; }
int StockItem::getStockLevel() const { return this->stockLevel; }
bool StockItem::getReorder() const { return this->reorder; }
// -----------------------------------------------------------------------

// edititemdialog.cpp ---------------------------------------------------
#include "edititemdialog.h"
#include "ui_edititemdialog.h"

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

    // need to link the ok button to the Dialog's "accept" slot
    connect( ui->buttonBox, &QDialogButtonBox::accepted, this, &QDialog::accept );
    // need to link the cancel button to the Dialog's "reject" slot
    connect( ui->buttonBox, &QDialogButtonBox::rejected, this, &QDialog::reject );
}

bool EditItemDialog::runDialog( StockItem & item ) {
    // populate the dialog's input widgets with item's parameters
    ui->name->setText( item.getName() );
    ui->unitCost->setValue( item.getUnitCost() );
    ui->stockLevel->setValue( item.getStockLevel() );
    ui->reorder->setChecked( item.getReorder() );

    // get Qt to run the dialog
    if( this->exec() == QDialog::Accepted ) {
      // if user clicked ok, update the item
      item.setName( ui->name->text() );
      item.setUnitCost( ui->unitCost->value() );
      item.setStockLevel( ui->stockLevel->value() );
      item.setReorder( ui->reorder->isChecked() );
      return true;
    }
    return false;
}

EditItemDialog::~EditItemDialog() {
    delete ui;
}
// -----------------------------------------------------------------

// edititemdialog.h ------------------------------------------------
#ifndef EDITITEMDIALOG_H
#define EDITITEMDIALOG_H

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

namespace Ui {
  class EditItemDialog;
}

class EditItemDialog : public QDialog {
    Q_OBJECT

public:
    explicit EditItemDialog( QWidget * parent = 0 );
    ~EditItemDialog();

    bool runDialog( StockItem & item );

private:
    Ui::EditItemDialog * ui;
    StockItem * result;
};

#endif // EDITITEMDIALOG_H
// -----------------------------------------------------------------

// mainwindow.cpp --------------------------------------------------
// need to modify handleEditButton slot so that it runs the dialog
void MainWindow::handleEditButton() {
    EditItemDialog dialog( this );
    StockItem item;

    emit statusUpdateMessage( QString("Edit button was clicked"), 0 );

    dialog.runDialog( item );
}
// --------------------------------------------------------------------

# hello_qt_cm/CMakeLists.txt -------------------------------------------
# This is the minimum cmake version needed for Qt
cmake_minimum_required( VERSION 2.8.11 )

# Name of the project
project( qt_ui )

# The CMake build process might generate some new files in the current
# directory. This makes sure they can be found.
set( CMAKE_INCLUDE_CURRENT_DIR ON )

# This allows CMake to run one of Qt's build tools called moc
# if it is needed. moc.exe can be found in Qt's bin directory.
# We'll look at what moc does later.
set( CMAKE_AUTOMOC ON )
set( CMAKE_AUTOUIC ON )

# Find the Qt widgets package. This locates the relevant include and
# lib directories, and the necessary static libraries for linking.
find_package( Qt5Widgets )

# Same as previously: define the executable and it's sources.
add_executable( hello_qt2 WIN32
                main.cpp
                mainwindow.cpp mainwindow.h mainwindow.ui
                edititemdialog.cpp edititemdialog.h edititemdialog.ui
                stockitem.cpp stockitem.h )

# Tell CMake that the executable depends on the Qt::Widget libraries.
target_link_libraries( hello_qt2 Qt5::Widgets )
# ---------------------------------------------------------------------
```

Once you have made these changes recompile your project and run.



### Modal and Modeless Dialogs

If you try to click on a button in the main window when the dialog is open, you'll notice that you can't. This is because the dialog is a *modal* dialog. Modal dialogs effectively hijack control of the application and demand that the user closes the dialog before control is returned to the parent window. This is useful in our case as otherwise the edit dialog could be open and editing an item, which is then removed from the list and deleted using the main window buttons.

In some cases, this behaviour is not helpful - for example if the user needs to be able to select text in the main window and then use a dialog to perform an operation on this text. In this case user interaction with main window and dialog is needed concurrently. *Modeless* dialogs allow this to happen.

Because we used QDialog->exec() to show the dialog, the dialog type defaulted to a modal dialog. Qt allows both types of dialog and other functions such as QDialog->show() will by default create a modeless dialog. There are a few different methods for creating modal and modeless dialogs in Qt, and for processing the dialog's data once *Ok* or *Cancel* is clicked. You can read more here.

### Exercise 8 - Creating a ListModel for the QListView

The previous exercise has shown how you can use simple user interface features to display and recieve data to and from the user. In GUI applications, you will often see some more advanced widgets including *Combo Boxes* (select from drop down list), *List Views* (select from displayed list), *Tree Views* (filesystem / directory structures in Windows Explorer etc) and others.

This type of widget is used to display datasets rather than single values and in most GUI toolkits, including Qt, they are implemented in a very similar way. Two objects are usually required:

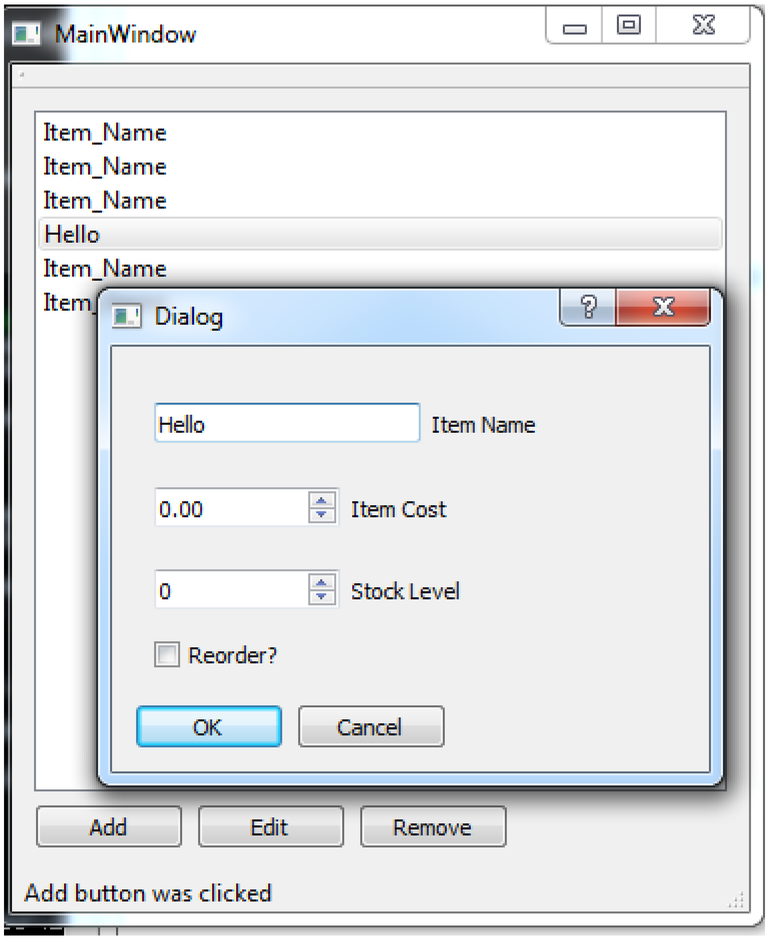
- The ***viewer*** object: `QListView` in our case.
- The ***model*** object: a sub-class of `QAbstractListModel` in our case

The *viewer* is responsible for displaying the contents of the dataset to the user and capturing the user's input to determine their selection. The *model* contains the dataset itself and is linked to the *viewer*. The *viewer* will retrieve information from the *model* so it knows what to display in the list/tree/combobox. You'll notice that the model widget is an *Abstract* class - that's because Qt doesn't know what datatype is going to be stored in the model, how it should be stored, or how it should be displayed. Its your responsibility to sub-class `QAbstractListModel` so that it can store lists of data of the type required and provide the necessary information to a `QListView` to be displayed in the GUI. You need the list model to store a list of `StockItems`, so creating an appropriate sub-class of `QAbstractListModel` is the first task. This means you need two more source files for the new class: *stockitemlist-model.cpp* and *stockitemlistmodel.h*. Additionally, some changes will need to be made to *mainwindow.cpp*. Example source code is given below, but don't forget to add the additional source files to CMakeLists.txt.

```
// stockitemlistmodel.cpp ---------------------------------------------
#include "stockitemlistmodel.h"

// function allows listview to determine the number of items in list
int StockItemListModel::rowCount( const QModelIndex & parent ) const {
    return stockItems.size();
}

// function allows the listView to request data at index
// QVariant is just Qt's way of defining a variable that could be any
// basic type (int, float, double, string, ... )
QVariant StockItemListModel::data( const QModelIndex & index, int role ) const {
    // Check that the index is valid
    if( !index.isValid() ) return QVariant();
    if( index.row() >= stockItems.size() || index.row() < 0 ) return QVariant();

    // Return item name as display variable
    if( role == Qt::DisplayRole ) {
        return QVariant( stockItems[index.row()].getName() );
    } else {
        return QVariant();
    }
}


// To allow modifying stored list
void StockItemListModel::addItem( const StockItem & s ) {
    // This emits a signal to warn the listView that extra rows will be added
    emit beginInsertRows( QModelIndex(), stockItems.size()-1, stockItems.size()-1 );
    // Add the extra item to the list
    stockItems.push_back( s );
    // Emits a signal to say rows have been added.
    emit endInsertRows();
}

void StockItemListModel::insertItem( const StockItem &s, const QModelIndex & index ) {
    //?? (same as add item)
}

void StockItemListModel::setItem( const StockItem &s, const QModelIndex & index ) {
    if( index.isValid() && index.row() >= 0 && index.row() < stockItems.size() ) {
        // Just replace the item in the vector
        stockItems[index.row()] = s;
        // Need to emit a signal that tells listView the data has changed
        emit dataChanged( index, index ); // index is the first and last row to be changed
    }
}

void StockItemListModel::removeItem( const QModelIndex & index ) {
    //?? (~ same as add item)
}

// to allow accessing stored list item properties
StockItem StockItemListModel::getItem( const QModelIndex & index ) const {
    if( index.isValid() && index.row() >= 0 && index.row() < stockItems.size() )
        return stockItems[index.row()];

    return StockItem();
}
// -------------------------------------------------------------------

// stockitemlistmodel.h ---------------------------------------------
#ifndef STOCKITEMLISTMODEL_H
#define STOCKITEMLISTMODEL_H

#include <QAbstractListModel>
#include "stockitem.h"

class StockItemListModel : public QAbstractListModel {
Q_OBJECT

public:
    // Constructor
    explicit StockItemListModel( QObject* parent = 0 ) : QAbstractListModel( parent ) {}

    // Virtual functions that must be defined in a valid ListModel
    int rowCount( const QModelIndex &parent = QModelIndex() ) const;
    QVariant data( const QModelIndex &index, int role ) const;

    // To allow modifying stored list
    void addItem( const StockItem &s );
    void insertItem( const StockItem &s, const QModelIndex &index );
    void setItem( const StockItem &s, const QModelIndex &index );
    void removeItem( const QModelIndex &index );
    void clearList();

    // to allow accessing stored list item properties
    StockItem getItem( const QModelIndex &index ) const;

private:
    std::vector<StockItem> stockItems;
};

#endif
// -------------------------------------------------------------------

// mainwindow.cpp ---------------------------------------------------
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "stockitem.h"
```

```
#include "edititemdialog.h"

#include <QMessageBox>

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

    // Link the ListModel to the ListView
    ui->listView->setModel( &stockList );
    // Tell this list view to only accept single selections
    ui->listView->setSelectionBehavior( QAbstractItemView::SelectItems );

    // Connect the released() signal of the addButton object to the handleAddButton slot in this objec
    connect( ui->addButton, &QPushButton::released, this, &MainWindow::handleAddButton );
      connect( ui->editButton, &QPushButton::released, this, &MainWindow::handleEditButton );
      connect( ui->removeButton, &QPushButton::released, this, &MainWindow::handleRemoveButton );

    connect( this, &MainWindow::statusUpdateMessage, ui->statusBar, &QStatusBar::showMessage );
}

MainWindow::~MainWindow() {
    delete ui;
}

void MainWindow::handleAddButton() {
    StockItem item;

    stockList.addItem( item );

    emit statusUpdateMessage( QString("Add button was clicked"), 0 );
}

void MainWindow::handleEditButton() {
    EditItemDialog dialog( this );
    QModelIndexList selectedList;

    selectedList = ui->listView->selectionModel()->selectedIndexes();

    if( selectedList.length() == 1 ) {
      // selectedList is a list of all selected items in the listView. Since we set its
      //  behaviour to single selection, were only interested in the first selecteded item.
      StockItem item = stockList.getItem( selectedList[0] );

      if( dialog.runDialog( item ) ) {
        // user clicked ok, need to update item in list...
        stockList.setItem( item, selectedList[0] );
      }
    } else {
      emit statusUpdateMessage( QString("No item selected to edit!"), 0 );
    }
}

void MainWindow::handleRemoveButton() {
    emit statusUpdateMessage( QString("Remove button was clicked"), 0 );
}
// ------------------------------------------------------------------

// mainwindow.h ------------------------------------------------------
// You'll need to define a new StockItemListModel as a private member
// variable in the MainWindow class:
private:
    StockItemListModel stockList;
// -------------------------------------------------------------------
```

Now rebuild and run.



Now see if you can implement the insertItem (new item is added after selected item) and removeItem function (selected item is removed).

***Final Note*** In addition for the `QListView`, `QTreeView`, ... objects there are also `QListWidget`, `QTreeWidgets`, ... objects that you can use. These are special cases where the *model* has been written for you. They are easier to use since the *viewer* and the *model* (including pre-written addItem, removeItem functions etc) are combined into a single *widget* that is ready to use. The disadvantage is that they dont offer the same flexibility as you can only create/store lists of basic `QStrings` in the object. You should be able to work out how to use these from the Qt documentation now you are familar with the concept.

## Exercise 9 - Add a *Save* Feature

Add a feature that allows the user to save the current list to a text file, the exact file format is up to you. Hopefully you can now work out how to do this yourselves using the online Qt documentation, here are some hints for where to look:

- Add a toolbar button, see here.

- Link the toolbar button *Action* to a slot in your main window, see here
- Spawn a `QFileDialog` to get the name of a file to save to, see here.
- Can you add/change the icon displayed on the toolbar?
- Can you also add a *File* dropdown to the MenuBar and add the *Save* action to this? See here.