EEEE2076 - Software Development Group Design Project

Worksheet 6 - Graphics User Interfaces with Qt

P Evans

February 5, 2026

Contents

1 Qt **1**
1.1 Overview . . . 1
1.2 Installing Qt . . . 1

2 Creating a Qt Project using CMake **1**
2.1 Creating the base project and user interface with Qt Creator . . . 2
2.1.1 Exercise 1 . . . 2

3 Signals and Slots **5**
3.0.1 Exercise 2 - Handling a button click . . . 6
3.0.2 Exercise 3 - Status bar . . . 7

4 Model Based Treeview **9**
4.1 The ModelPart class . . . 10
4.2 The ModelPartList class . . . 10
4.3 Modifying your MainWindow class . . . 11
4.3.1 Exercise 4: Working Tree View . . . 11
4.3.2 Exercise 5: Determine user's Tree View Selection . . . 12
4.4 Toolbars and Menus . . . 14
4.4.1 Exercise 6: Adding Tool Bars and Menu Bars . . . 14

5 Resource files and Icons **16**
5.1 Icon files and resource . . . 16
5.1.1 Exercise 7: Resource files and Icons . . . 16

6 Dialogs **17**
6.1 Standard File Open/Save Dialogs . . . 17
6.1.1 Exercise 8: Open File Dialog Example . . . 17
6.2 Custom Dialogs . . . 18
6.2.1 Exercise 9: Custom Dialog Example . . . 18
6.3 Context Menus . . . 19
6.3.1 Exercise 10: Add context menu to treeview . . . 19
6.4 Signoff criteria . . . 20

1 Qt

1.1 Overview

This worksheet is longer and more difficult than the previous sheets but it is important as it builds the basic software framework that you will need for the group work.

Make your you add/commit your work to Github after each exercise.

1.2 Installing Qt

Search for Qt online, select download and then find Open Source installer. Run the installer, you'll need to register when you do this. When you install, choose a custom install, select the most recent "stable" version, this was 6.10.2 when this was written. **Do not just install all of this version**, expand the folder containing the version and only select the MSVC (Microsoft Visual Studio) 64-bit component. If you install everything, you'll get Qt for lots of different compilers - for Android, etc and the download will be well over 10GB. Under "Developer and Design Tools", make sure Qt Creator is selected as we will use this for editing the GUI.

After you have installed it, add the *c:\Qt\6.10.2\msvc2022_64\bin* folder to your system path. This will be needed to allow a Qt application to run (puts the Qt dlls on the path), but it also acts as a hint to CMake to help it find Qt. Note - this path may be different depending on the Qt version you install - check it, and I recommend browsing the folder and copy/paste the path from file explorer into the PATH editor.

2 Creating a Qt Project using CMake

Qt enabled programs have 'normal' source code files (.cpp, .h) and additional .ui files that describe the graphical user interface. The .ui files are created using Qt Creator. When the program is compiled, Qt first processes the .ui files and automatically generates extra source code for the user interface.

2.1 Creating the base project and user interface with Qt Creator

2.1.1 Exercise 1

We want to create a CMake enabled Qt project. The easiest way of doing this is use Qt's Creator program to set everything up.

- Open Qt Creator, it should be in the Start Menu / Windows search or alternatively it will be in the \tools\QtCreator\bin sub-directory of your Qt install.
- Choose File - New Project - Qt Widgets Application, choose a location for your project (a Worksheet 6 sub-folder of your Github repository)
- Give the project a name, click next and select CMake as the build system when prompted.
- It will ask you to specify a 'build kit', this is the compiler Qt will use if you build the project from within the Qt UI. If you export the project to another computer and rebuild using CMake you will have the option to change this.
- Qt will also give you the option to use Git version control. Don't do this, we'll add it to your existing Git repo manually to avoid any issues - we can control what is added, etc.

- Qt should now have created you a project folder containing:
  - *CMakeLists.txt* - The CMake project description.
  - *main.cpp* - Entry point to the program, it just creates the MainWindow class
  - *mainwindow.h* - The header that defines the MainWindow class for the program
  - *mainwindow.cpp* - Source code for the MainWindow class. Code in here will do things like add extra items to MainWindow GUI and handle button clicks
  - *mainwindow.ui* - This is an XML description of the GUI that you create in Qt designer.

The process I am going to follow now is: edit the mainwindow.ui file in Qt Creator, open the project with CMake/Visual Studio as in Worksheet 4. I prefer the Visual Studio environment for writing code but Qt Creator is needed to edit the GUI. You can do everything in Qt Designer if you prefer.

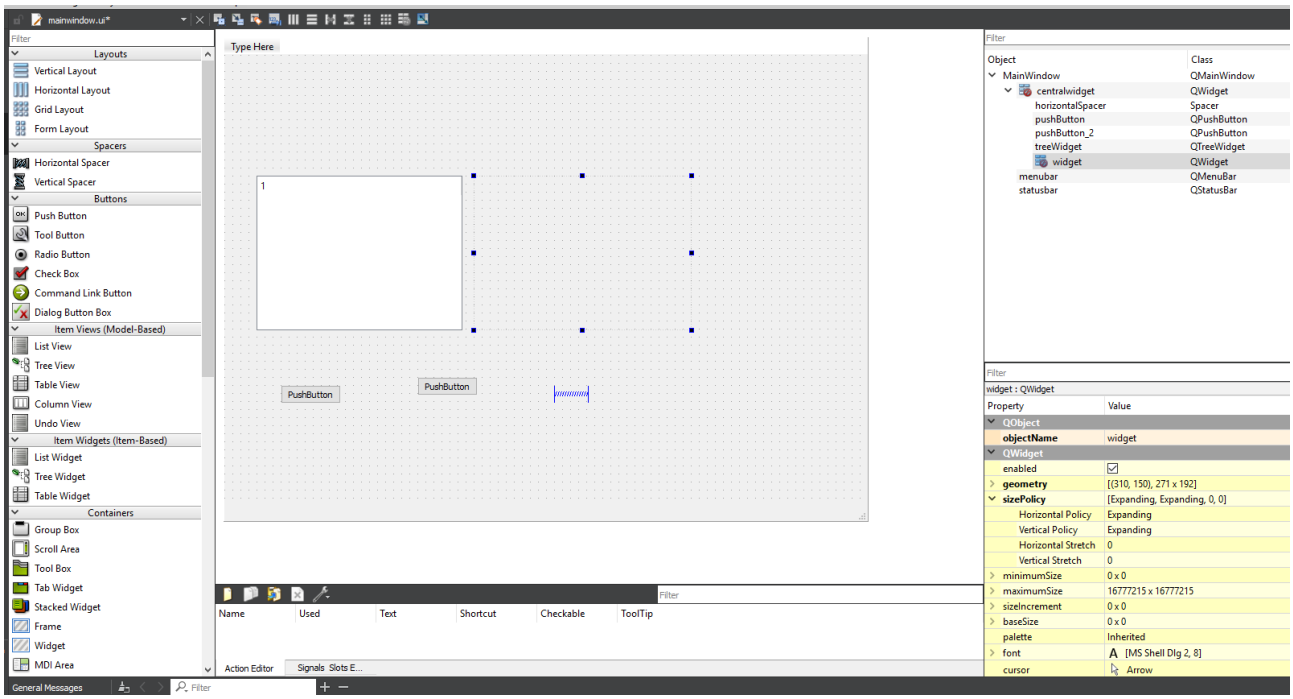

Figure 1: Setting the size policy for a widget - these controls, along with the min/max size numbers are hints to Qt that it used when working out how to divide available space between the widgets.

- Open the mainwindow.ui file from within Qt Creator. If it has already been opened it will be available to select from the dropdown menu at the top, otherwise File menu and then Open File.
- You now have the blank MainWindow UI open. You need to drag some Widgets (user interface components) onto the blank UI and then create a layout so that the Widgets stretch and scale with the window.
- Drag the following items onto the screen and arrange as show in Fig. 1
  - A model-based TreeView

  - An empty 'Container Widget' (this will be 'promoted' to a VTK display widget in the next worksheet)
  - Two push buttons
  - A horizontal spacer

- The initial attempt at a GUI will consist of two 'rows' - the top row contains the TreeView and empty Widget, then there is a row of buttons below. The process for creating this GUI is to create layouts for each of the rows first, then add these layouts to a global layout for the Window.
- To create a layout for the TreeView/Container Widget combination - select both and them and hit Control-H or choose the 'Layout Horizontally' button from the toolbar.
- We'll now do the same for the buttons, however a horizontal spacer is also needed. This is because when the two rows are added to the Window layout, Qt will expect the button row to be the same width as the Tree/Container row and will stretch the widgets to make this happen. Without the spacer the buttons will end up unreasonably wide, but with the spacer it is the spacer that takes up the slack.
- Once you have the two rows (you'll have the widgets in two sets, each surrounded by a red box), click anywhere in the MainWindow (do not select the row groups, just click elsewhere in the MainWindow widget) and hit the Layout Vertically button or Ctrl-L. This will create a vertical layout that is mapped to the MainWindow size so the widgets scale as the Window size changes. If you select the two groups and Ctrl-L, the groups will be joined but will just sit at a fixed position and size within the window (try this).
- Save mainwindow.ui
- Now follow the procedure from Worksheet 4. Open the project with CMake, configure, generate, build and run in Visual Studio.
- You will probably find that the tree view dominates and the space for the 3D model viewer is non-existent (Fig. 2). Although you have told Qt how to position the UI components, you haven't explicitly told it how they should share the available space.

  With GUI designs, generally components can either be fixed size, prefer to take up as little space as possible, or prefer to take up as much space as possible. The GUI tookit (Qt) will take these preferences (known as the size policy) into account when deciding how much space in the window to allocate to components that share a row for example, both when the window is created and again when it ie resized. Fig. 3 shows how to change these options to give different behaviour.

  Try this and other options, you can more details on this topic [here] but spending a bit of time on a trial and error approach is quite helpful in developing an understand of how different option combinations behave.

3 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 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

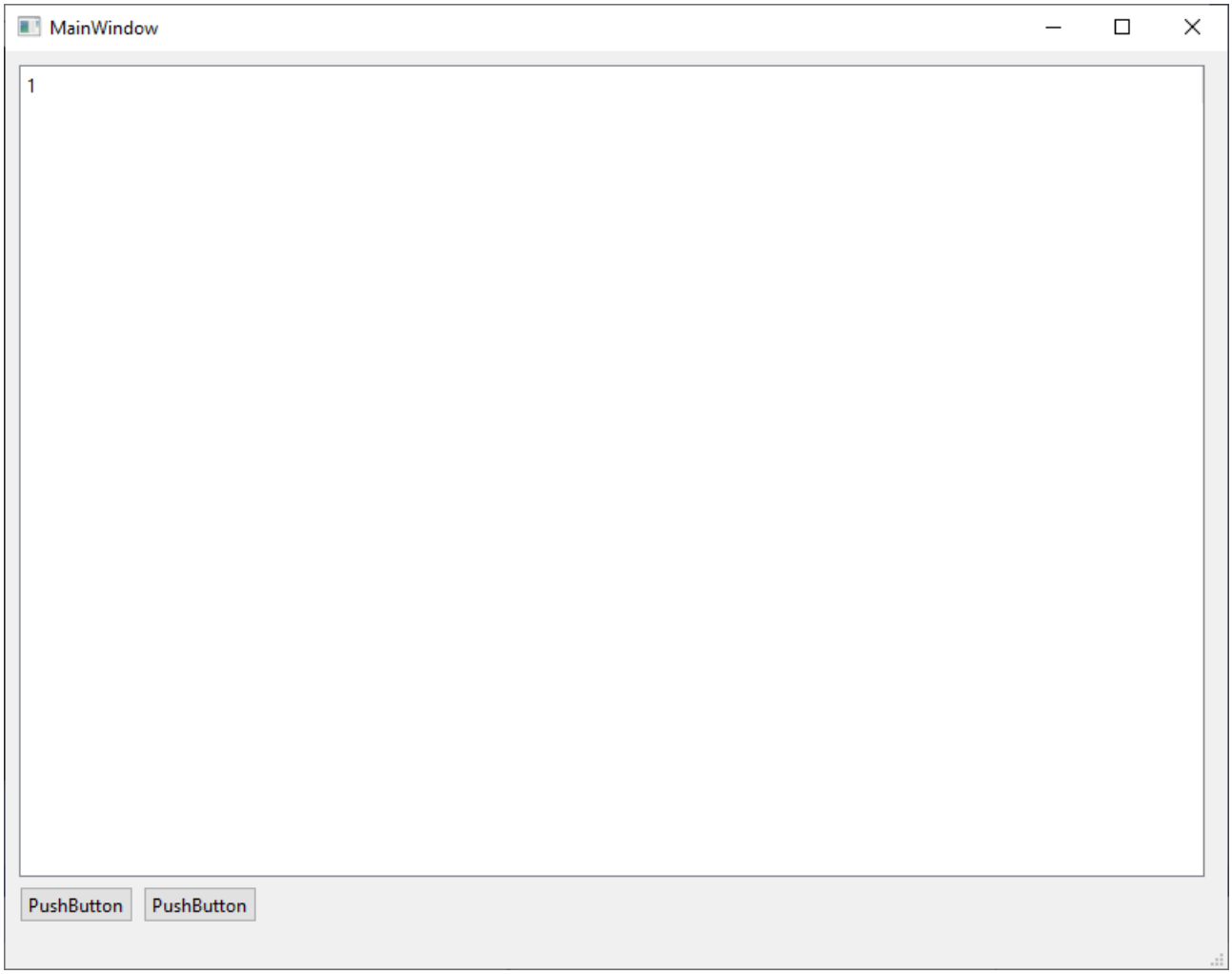

Figure 2: Setting the size policy for a widget - these controls, along with the min/max size numbers are hints to Qt that it used when working out how to divide available space between the widgets.

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 receive 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.

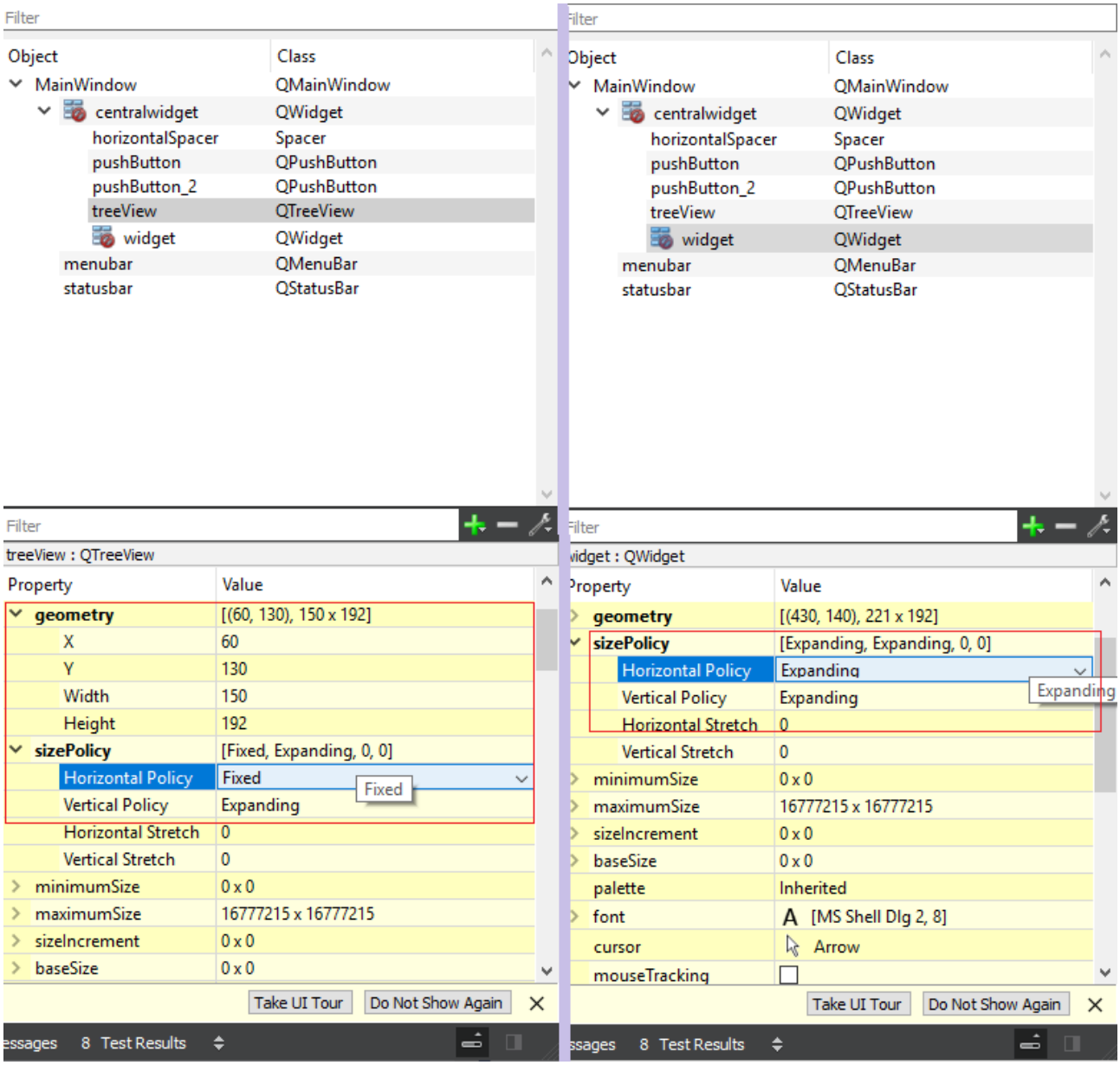

Figure 3: Setting the size policy for a widget - these controls, along with the min/max size numbers are hints to Qt that it used when working out how to divide available space between the widgets.

3.0.1 Exercise 2 - Handling a button click

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----------------------------
// Add this to the MainWindow class definition
public slots:
    void handleButton();
// ----------------------------------------------------------------------
```

```
// Example of connecting signals and slots in mainwindow.cpp-------------
// Add the following lines at the end of the MainWindow constructor

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

```
// Example of slot implementation in mainwindow.cpp----------------------
// Add the following function definition to mainWindow.cpp
// You'll need to #include <QMessageBox>

void MainWindow::handleButton() {
    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 translated to pure C++ by the Qt Meta Object Compiler (moc.exe) before the code is compiled. That's why the 'set(CMAKE_AUTOMOC_ON)' line 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 applied to the class. If you are interested, see [here] for more information.

3.0.2 Exercise 3 - Status bar

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--------------------------
// This needs adding to the MainWindow class definition
signals:
    void statusUpdateMessage( const QString & message, int timeout );
// ----------------------------------------------------------------------
```

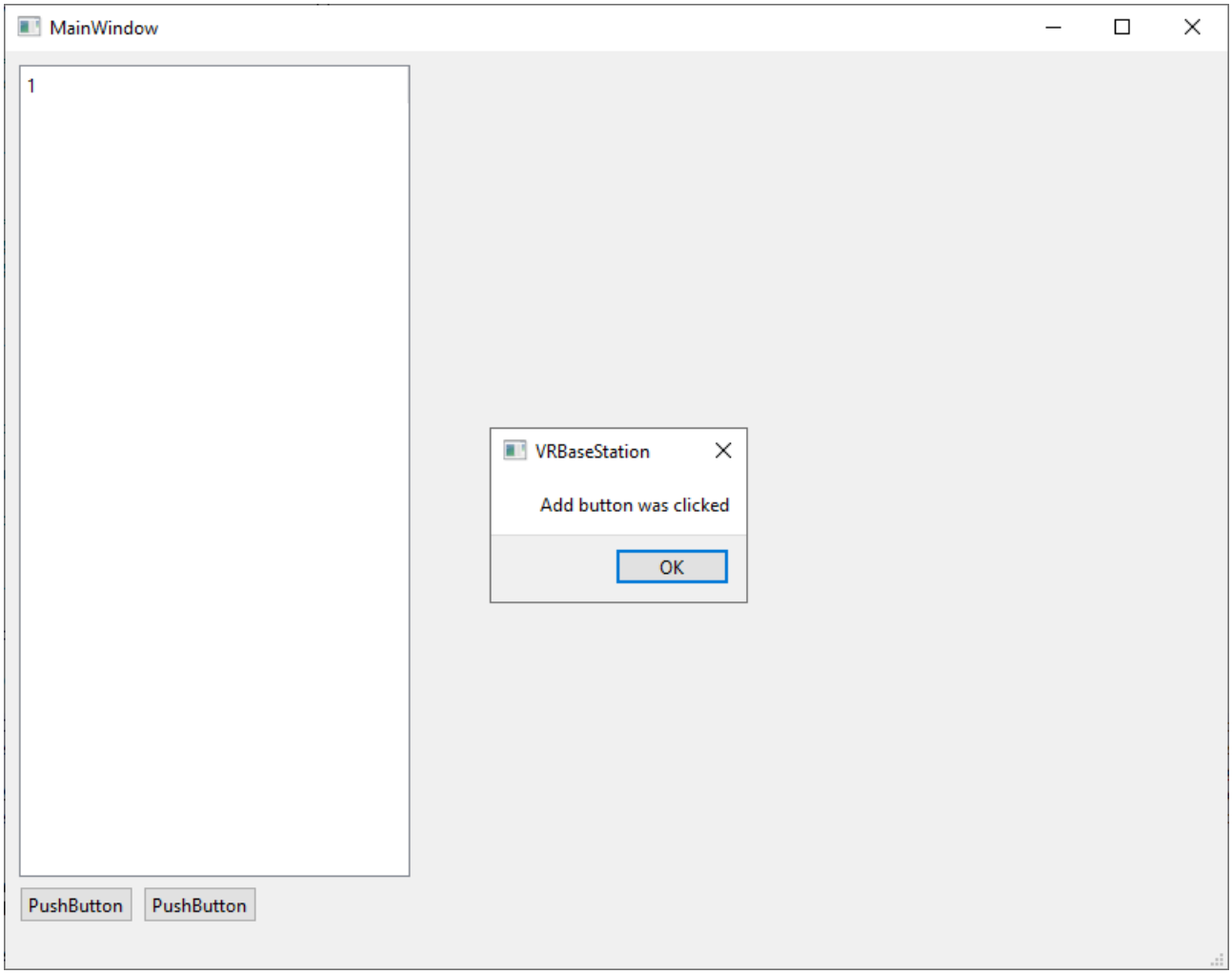

Figure 4: You should be able to get a messagebox to appear when the push button is clicked.

```
// Example of connecting to StatusBar signal mainwindow.cpp----------------
// This needs adding to end of MainWindow constructor

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

```
// Update your button handling slot so it activates statusbar

void MainWindow::handleButton() {
    // 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.

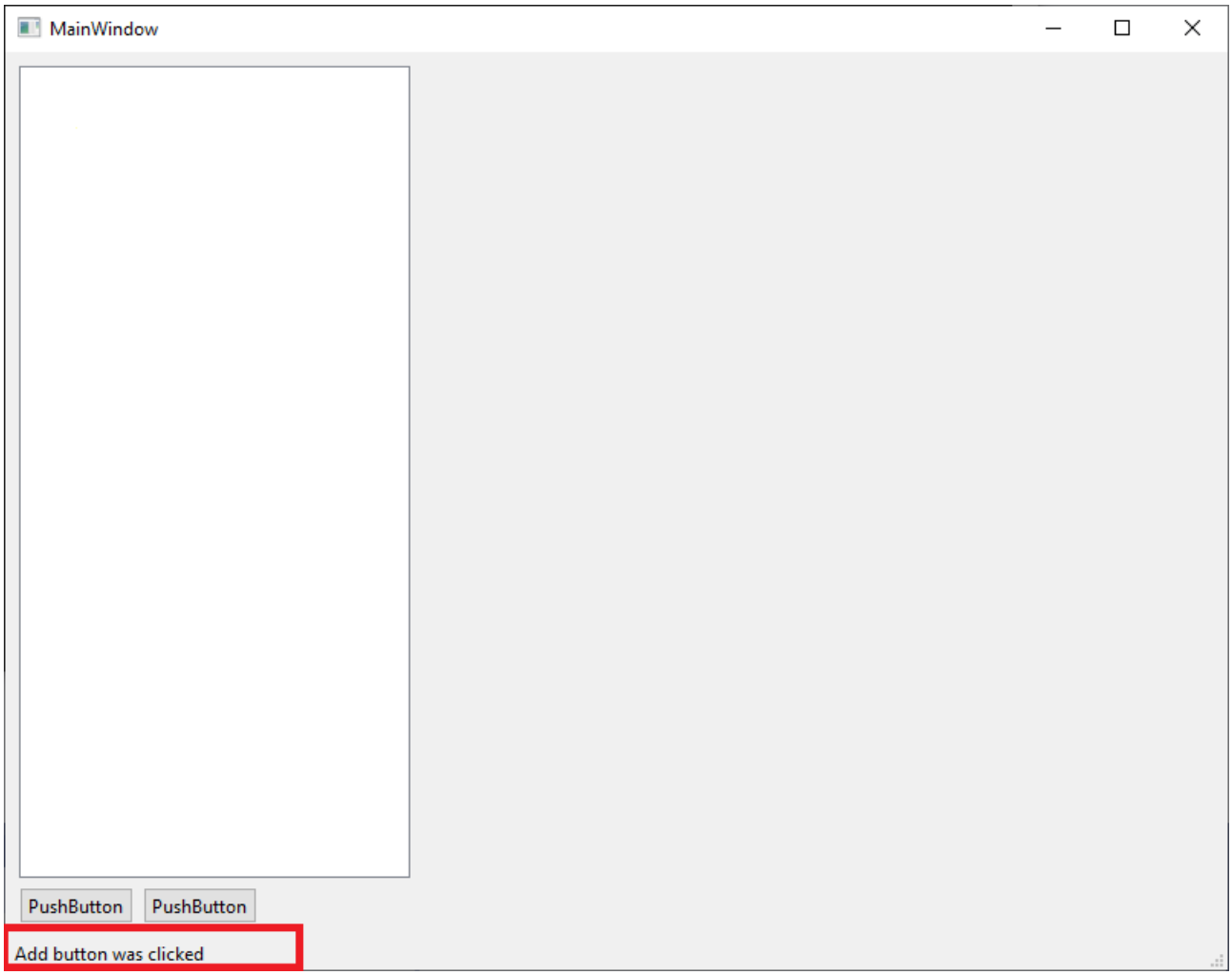

Figure 5: Status bar message

4 Model Based Treeview

The VR applications requires you to be able to be able to add multiple components or parts into the 3D rendering. A common method for representing a collection of components in a GUI is the use of treeviews. These allow the parts to be organised in a logical way.

Qt has two types of treeview: a basic TreeWidget which allows a text based representation of objects in a tree; and a generic TreeView widget. The TreeView widget is able to store a set of any C++ class in a tree structure. This is useful as it allows you to define a class which contains the data you actually want to work with (e.g. CAD models of components), and store these in the tree structure. You have added the TreeView widget to the GUI in the first step, now you need to create two classes:

- A ListModel class - this represents a list or array of the items that will go in the tree. It must follow a standard template and be a sub-class of the Qt QAbstractListModel class. Ours will be called the ModelPartList class and consist of ModelPartList.h and ModelPartList.cpp files.
- A ListItem class - this represents the items that will go in the list. It must follow a standard template and provide certain functions to Qt that allow Qt to represent the item in the tree. It will store key properties (e.g. name) of the item in an array of Qt data structures called QtVariants, as well as links to the parent and child items in the tree. Ours will be called the ModelPart class and consist of ModelPart.h and ModelPart.cpp files.

These classes are based on the example available [here] and modified source files are available for you to download from [here] (**I am not asking you to write the classes, download them from this link!). After downloading the class files, you need to modify CMakeLists to add them to the project.** The structure of the two classes is explained below.

4.1 The ModelPart class

- The class will contain custom properties for each item. Some properties of the class are stored in an array of QVariant variables - this is standardised way of storing various Qt variables. The reason for using this approach is that it allows Qt to access key information about the part when constructing the treeview - e.g. retrieving the part name so the treeview display can be updated.
- In this case, two items will be stored in the QVariant array - the part name and another variable called 'visible', this second variable is used only to illustrate how multiple columns look in the treeview. You can edit the columns later for your application.
- There are some other pointers than are used to describe the tree structure - the part's parent and children.
- You can add additional variables to the class to store other properties later.
- Many of the functions in the class are required by Qt to enable it to extract and set relevant information.
- There are some functions and variables that are commented out - these involve VTK variables. If uncommented they will cause compile errors as you haven't installed VTK yet.

Make sure you look at *ModelPart.cpp* and *ModelPart.h* and understand the code. Ask if you are not sure what the code does, if you don't understand ut will make your life difficult later.

4.2 The ModelPartList class

- The class is a container which holds the list of items. The list is held in a standard format so Qt is able to retrieve information to build the treeview.
- The class must be a subclass of the Qt abstract class QAbstractItemModel. Being a subclass of this forces the class to comply with certain rules (contain certain functions etc) which allows it to be linked to the treeview you see in the GUI,
- The class definition (taken from ModelPartList.h) is shown below.

Make sure you look at *ModelPartList.cpp* and *ModelPartList.h* and understand the code. Ask if you are not sure what the code does, if you don't understand ut will make your life difficult later.

4.3 Modifying your MainWindow class

- You will need to **add a private variable**, consisting of a pointer to a ModelPartList class to your MainWindow.h header file:
  ModelPartList* partList;

- **Include the necessary headers in MainWindow.h**, so that your MainWindow class is aware of ModelPart and ModelPartList.
- **Add the code shown below to the constructor of your MainWindow class**, in MainWindow.cpp. This code initialises the ModelPartList, links it to the treeview, and then adds some items to the tree as a demonstration.

```
/* Create / allocate the ModelList */
this->partList = new ModelPartList("PartsList");

/* Link it to the treeview in the GUI */
ui->treeView->setModel(this->partList);

/* Manually create a model tree - there a much better and more flexible ways of doing
    this,
e.g. with nested functions. This is just a quick example as a starting point. */
ModelPart *rootItem = this->partList->getRootItem();

/* Add 3 top level items */
for (int i = 0; i < 3; i++) {
   /* Create strings for both data columns */
   QString name = QString("TopLevel %1").arg(i);
   QString visible("true");

   /* Create child item */
   ModelPart *childItem = new ModelPart({ name, visible });

   /* Append to tree top-level */
   rootItem->appendChild(childItem);

   /* Add 5 sub-items */
   for (int j = 0; j < 5; j++) {
      QString name = QString("Item %1,%2").arg(i).arg(j);
      QString visible("true");

      ModelPart *childChildItem = new ModelPart({ name, visible });

      /* Append to parent */
      childItem->appendChild(childChildItem);
   }
}
```

4.3.1 Exercise 4: Working Tree View

Implement the above to get the tree view demo working. If successful, you should see something like what is shown in Fig. 6.

4.3.2 Exercise 5: Determine user's Tree View Selection

In your VR application, you might want to know which item in the tree view the user has selected (e.g. if you have a menu that then edits the properties of an item, or if you want to highlight the model of the selected item, or if you want to add a new item as the child of the selected item). You can do this as a two stage process:

1. Capture the 'clicked()' signal that is emitted when a user clicks on the Tree View. To do this:
   - Create a slot function. Copy the button handler slot function, rename to handleTreeClicked() or something. Remember there is code in both the header and the source file.
   - Then add a connect() call to the constructor. You can create a copy of the connect that current handles you button click connect, but remember that this time your are capturing the *QTreeView::clicked*

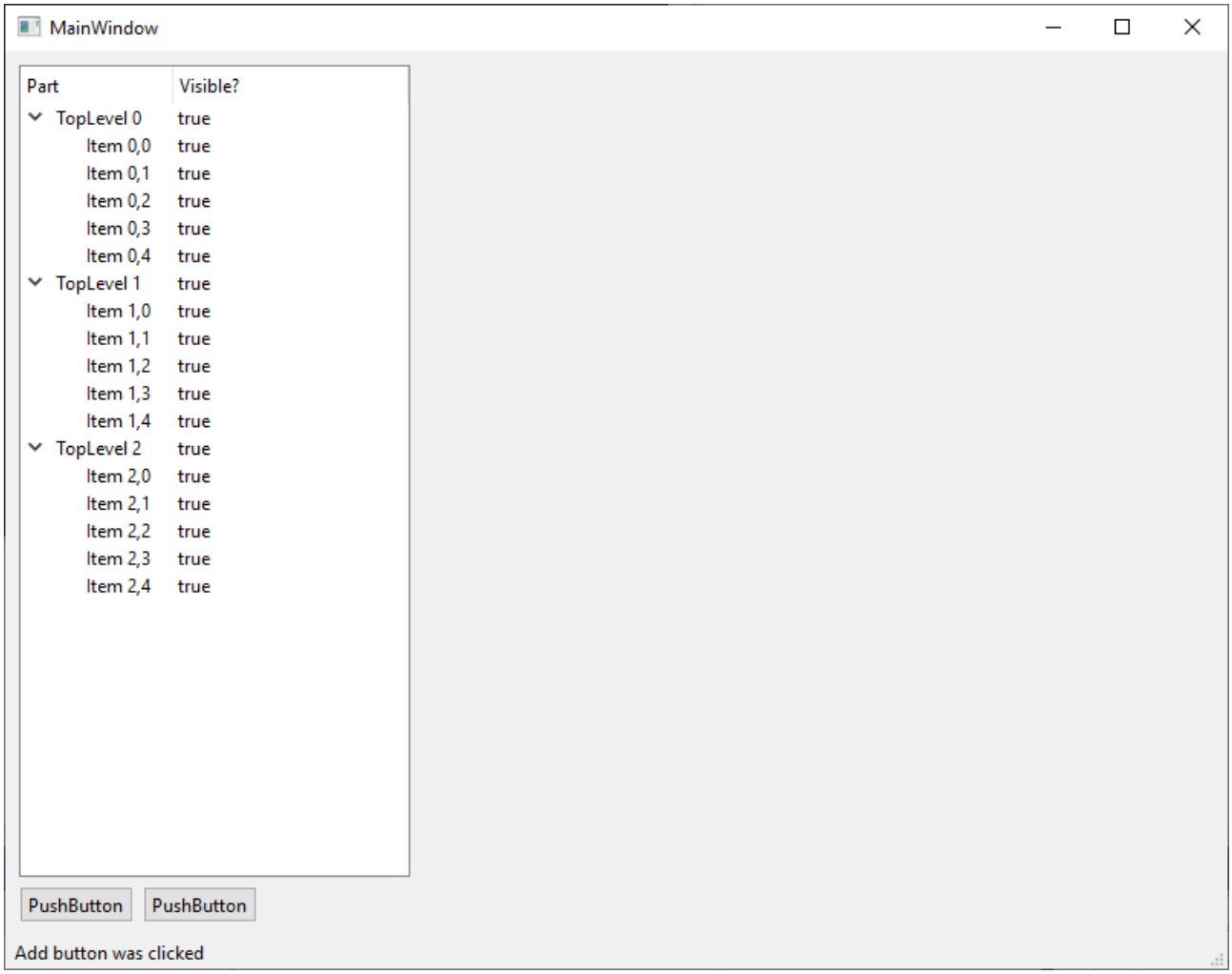

Figure 6: Working treeview with attached list model.

signal rather than *QPushButton::released*, and that the widget is no longer *ui->pushButton* - you will need to adapt the code.

2. You then need to query the tree model to get the selected item index, from this you get a pointer to the actual selected item, and from this you can extract any variable from within the selected item (e.g. its name in this case). The code below does this and can be **put in your new slot function**, try to understand what it does as you might need to reuse it later.

```
/* Get the index of the selected item */
QModelIndex index = ui->treeView->currentIndex();

/* Get a pointer to the item from the index */
ModelPart *selectedPart = static_cast<ModelPart*>(index.internalPointer());

/* In this case, we will retrieve the name string from the internal QVariant data
    array */
QString text = selectedPart->data(0).toString();

emit statusUpdateMessage(QString("The selected item is: ")+text, 0);
```

4.4 Toolbars and Menus

4.4.1 Exercise 6: Adding Tool Bars and Menu Bars

Applications usually have a toolbar (icons) along the top of the main window and a menu bar (drop down menus), both to allow operations such as opening or saving files. We will now add this to your application. You will now add these.

- Right click on the main window in the QtCreator project tree, select 'Add Tool Bar' (Fig. 7)
- A toolbar will be added but it can be quite difficult to see
- To add a menu bar, you right click on the main window in the QtCreator project tree, select 'Create Menu Bar'. The application will probably already have a menu bar and so this option might not be visible. If you have a menu bar, there will be a small 'Type Here' label in the top right of you MainWindow.

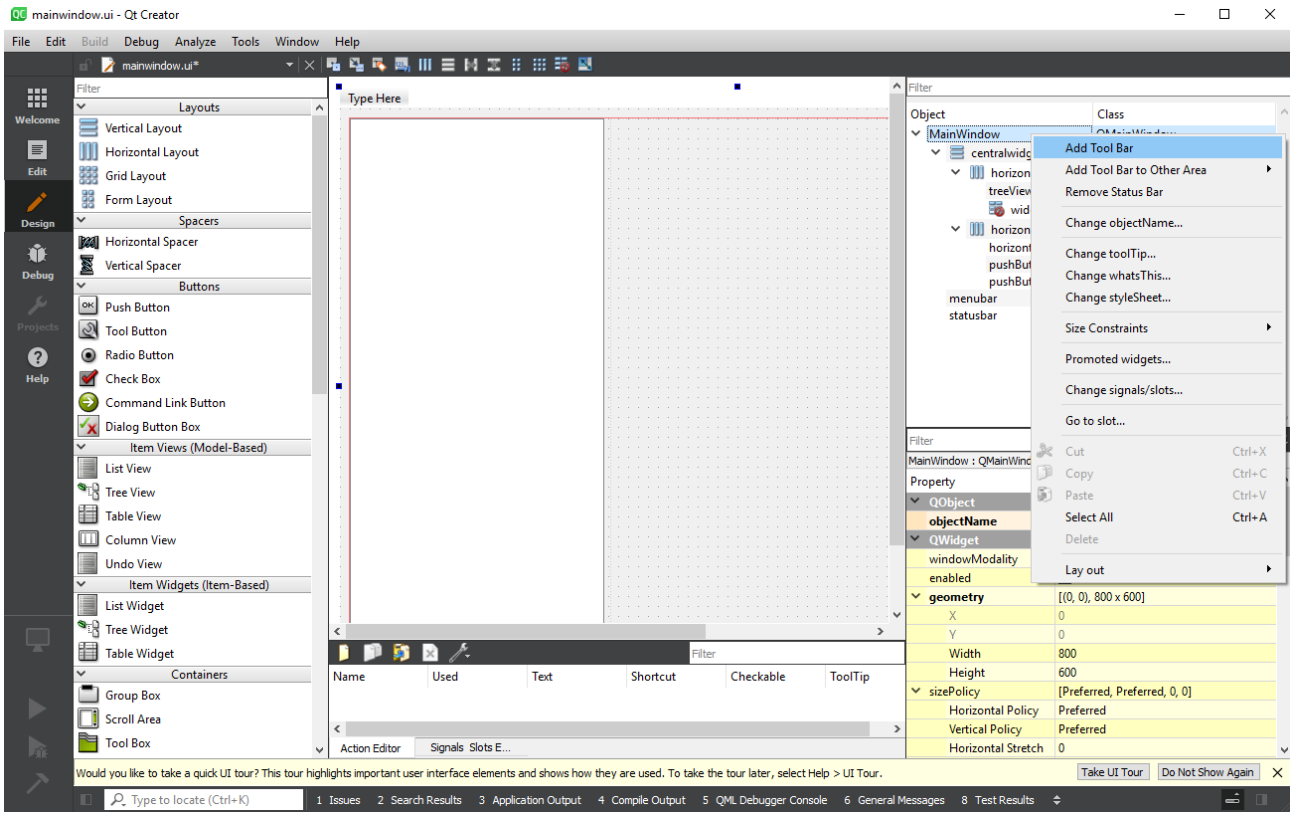

Figure 7: Adding a toolbar.

Once you have the tool and menu bars, you need to create some 'Actions'. Actions emit signals when they are triggered, and they can be configured to trigger when a use selects an item on a menu or tool bar. The Action Editor is the area below the MainWindow design area in Qt Creator. There are a series of buttons along the top, click the first one which is to add an action. You will be prompted to enter a name and configure an icon, enter "Open File" as the name but don't worry about the icon for now, we will come back to this. You should now see the action in the action list, you can drag and drop the action onto the tool bar.

You can also add the action to the menu bar, type ‘File’ into the ‘Type Here’ box at the top of the mainWindow design. You will now get the familiar ‘File’ dropdown menu and you can also drag and drop the ‘Open File’ action into the File menu.

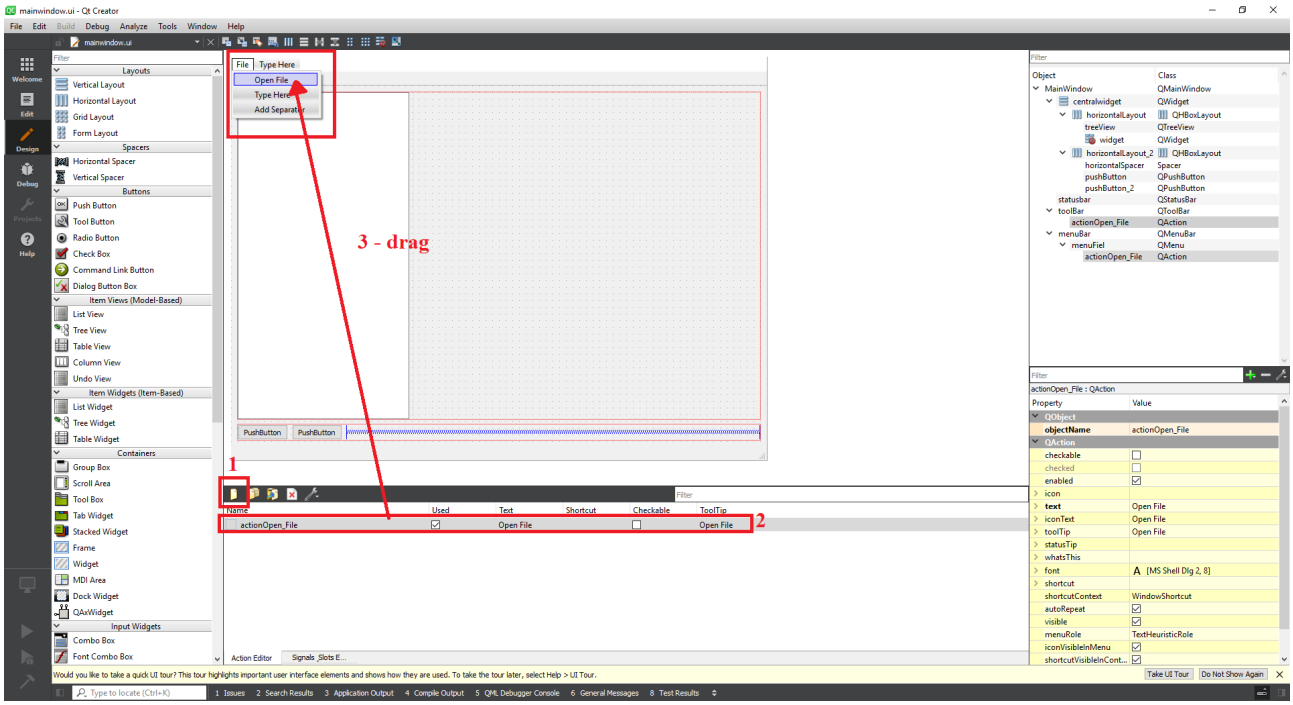

Figure 8: Adding an action

Once the action is added and linked to menus, you need to link the signal emitted by the action to a slot. Qt can create a code template for you but you must have the entire project open in Qt Creator. Make sure you have all of the project files open in QtCreator. If this isn’t the case, re-open QtCreator, go to Open File or Project, select all the .cpp, .h and CMakeLists files. In the Edit tab at the left of the screen, you should see your project. Right click on the action and select ”Go to slot”, select the triggered() slot, and then ok. Qt will create the slot function for you in mainWindow.h and mainWindow.cpp. The code added is shown below, you can add this manually if Qt fails to do it. Make sure you add a line of code in the new slot function to display a message on the statusbar when the action is triggered. Build the program and make sure everything is working.

```
// Slot function added to MainWindow.cpp
void MainWindow::on_actionOpen_File_triggered() {

    // Add this line of code so you can see if the action is working
    emit statusUpdateMessage( QString("Open File action triggered" ), 0 );
}
```

```
    // Slot function declaration added to MainWindow
public slots:
    void on_actionOpen_File_triggered();
```

Note: If the action and slot are named as they are in this example, the two should automatically connect (you don't need a connect() in the constructor). In this case, the action is called 'actionOpen_File' and the slot is called 'on_actionOpen_File_triggered() so the triggered signal of the action is connected to the slot automatically. If you don't follow the naming convention for the slot, or if the connecting fails for some other reason, you can manually connect them using a connect() function call in the class constructor.

```
connect( ui->actionOpen_File , &QAction::triggered , this , &MainWindow::
 alternative_slot_function_name );
```

5 Resource files and Icons

You should now have working menu and toolbar actions but they don't really look like they would in a typical program, they will look much better if you add icons.

5.1 Icon files and resource

You are going to add an icon to the Open File action and this will subsequently make your toolbar and menu look much more professional. First download a set of icon files from [here], you will need to create a 'Icons' sub-directory in your project folder and copy the icon pngs into this. You are free to find your own icons if you wish.

Obviously the program (the .exe) will need to access these icon (.png) files when it runs to be able to display the icon images. There are two ways that this can happen:

- The program can open the .png files directly. This can cause problems though - if the path to the png file is encoded as an absolute path (c:\...) then the icons will not work if the program is moved to another PC or even to another folder on the same PC and the path to the icons will change. Relative paths can be used but this still requires the png files to be copied to the build folder, and to any install folder if the program is installed.
- There is an alternative approach: resource files. Resource files are a way of embedding data within the .exe itself, copies of the .png files are placed at the end of the .exe. When the program runs, it reads the .pngs from within its own .exe file, rather than separate .png files - this means it can never fail to find the icons.

To get this to work with Qt, you need to create a resource file and add it to the project. A resource file is just an xml file that lists the files you would like to be embedded in the .exe.

5.1.1 Exercise 7: Resource files and Icons

- Make sure you have all of the project files open in QtCreator. Open QtCreator, go to Open File or Project, select all the .cpp, .h, .ui and CMakeLists.txt files. In the Edit tab at the left of the screen, you should see your project - it will have a tree view with the project name and all source/ui files listed under it.
- Now go to File - New File - New *Qt Resource File* in Qt Creator, select choose, and create a file called icons. If your project is open properly in QtCreator, it should give you the option of adding the new .qrc resource file to your project.
- You may need to manually add the icons.qrc file to your CMakeLists.txt - add *icons.qrc* to the *PROJECT_SOURCES* list. Qt Creator will probably remind you that you need to do this. Once added, the qrc file should show up as file in your project.

- Then select the icons.qrc file in Qt Creator and Click *Add Prefix* to create a new *Prefix* called *Icons*. A prefix is like a sub-directory in the resource file.
- Then right click on icons.qrc in Qt Creator and *Add Existing Files* and add the pngs you downloaded - the icons should now be listed under *icons.qrc* in the project tree.
- Go back to the mainWindow ui, select your Open File action, select Edit, got to the Icon selection line, choose *Resource* from the dropdown, you should see your icons for selection.

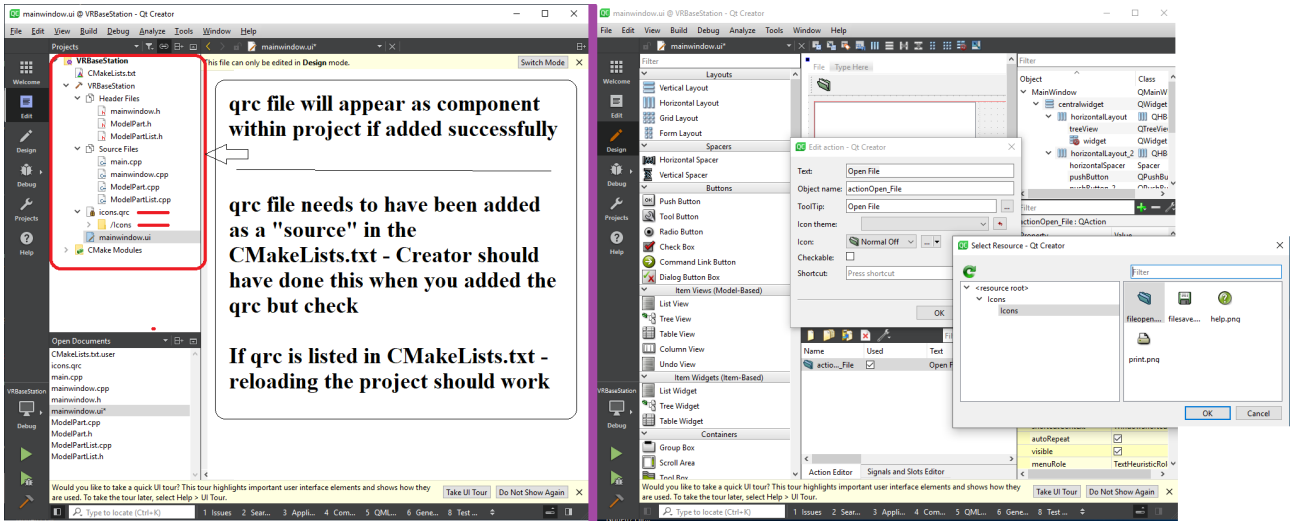

Figure 9: Adding a resource file icon to an action.

6 Dialogs

Dialogs are additional windows that are usually child windows of the main window, and temporarily created to serve a particular function - e.g. select a file to be opened. Dialogs are generally divided into two categories - modal and modeless. Modal dialogs take over control from the main window - you can't use any widgets in the main window until you have closed the dialog whereas Modeless can co-exist with the main window.

We'll look at two ways of creating dialogs in Qt, one is a quick shortcut to create the common Open or Save file dialogs and the second allows you to create fully custom dialogs.

6.1 Standard File Open/Save Dialogs

6.1.1 Exercise 8: Open File Dialog Example

These are really easy to implement and use the [QFileDialog] class. There are three things you need to do:

- Include the QFileDialog header
- Call the static QFileDialog::getOpenFileName() function.
- Use the return value for this function to determine the file that the user selected.

You really only need one line of code, other than the include. This line is something like:

```
QString fileName = QFileDialog::getOpenFileName(
                                this,
                                tr("Open File"),
                                "C:\\",
                                tr("STL Files(*.stl);;Text Files(*.txt)") );
```

Obviously you need to put the line of code somewhere so it is called when the user chooses to open a file, i.e. the relevant slot. You also need to do something with the filename it returns such as open the file - it doesn't do this for you!

Try to implement this in your program, for now you can just write the name of the file selected to the status bar to prove that it works.

6.2 Custom Dialogs

You can use Qt Creator to design dialogs in the same way as you did for the main window. The following is a basic example to illustrate the process.

6.2.1 Exercise 9: Custom Dialog Example

Create a new dialog that will ultimately allow the user to edit the properties of a model part (name, colour, etc)

- Make sure you have all of the project files open in QtCreator. Open QtCreator, go to Open File or Project, select all the .cpp, .h and CMakeLists files. In the Edit tab at the left of the screen, you should see your project.
- File - New File or Project - *Qt Designer Form Class* - Dialog with Buttons. Give it a name - e.g. *Option Dialog.*
- You need appropriate widgets to allow the user to specify *name, colour, e.g. R,G,B, a boolean "isVisible" flag* - you can choose
- Now you need to get this dialog to run when you click a button. Create a slot linked to the second pushButton (copy and edit the one that is already working for the first pushButton)
- Add the code below to this slot function.
- Try to build and run. If you get linked errors ("unresolved external symbol...") then make sure that Qt had added the dialog.cpp, .h and .ui file to your CMakeLists.

```
OptionDialog dialog(this);

if (dialog.exec() == QDialog::Accepted) {
    emit statusUpdateMessage(QString("Dialog accepted "), 0);
} else {
    emit statusUpdateMessage(QString("Dialog rejected "), 0);
}
```

The result should be something like that shown in Fig. 10 if it works correctly. You will need a way of saving any information entered in the dialog to the relevant model part - can you work out how to do this? You'll need to look at documentation for widgets such as the [QLineEdit] to work out how to get/set the display information. You'll probably need a function in the dialog class that can update the dialog from a ModelPart, or update a ModelPart from the dialog.

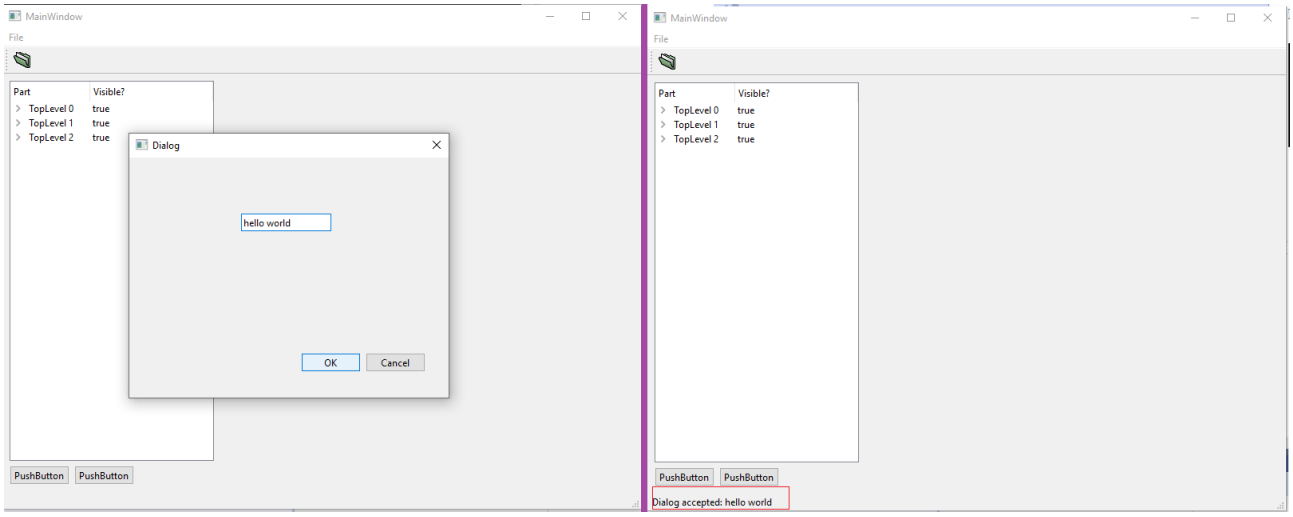

Figure 10: Working custom dialog (not all required widgets have been added to this - you will need more)

6.3 Context Menus

Context menus are menus that appear when you right click on something.

6.3.1 Exercise 10: Add context menu to treeview

Add a context menu to your TreeView - select the tree view and find the *Context Menu Policy* in the parameter list. Set this to *Actions Context Menu* (Fig. 11). This will allow you to link actions to a *right click context menu*.

Next create a new action in the action editor, this is the action that will be triggered by the context menu item. The action needs to be linked to its own slot, e.g. following the standard naming convention if I create an action called *actionItem_Options* (Fig. 12), my slot would need to be called on_actionItem_Options_triggered() for it to be automatically linked.

```
void MainWindow::on_actionItem_Options_triggered() {
}
```

Finally an item needs adding to the right click context menu and linking to this action/slot. This is done by adding a single line of code to the MainWindow constructor:

```
ui->treeView->addAction(ui->actionItem_Options);
```

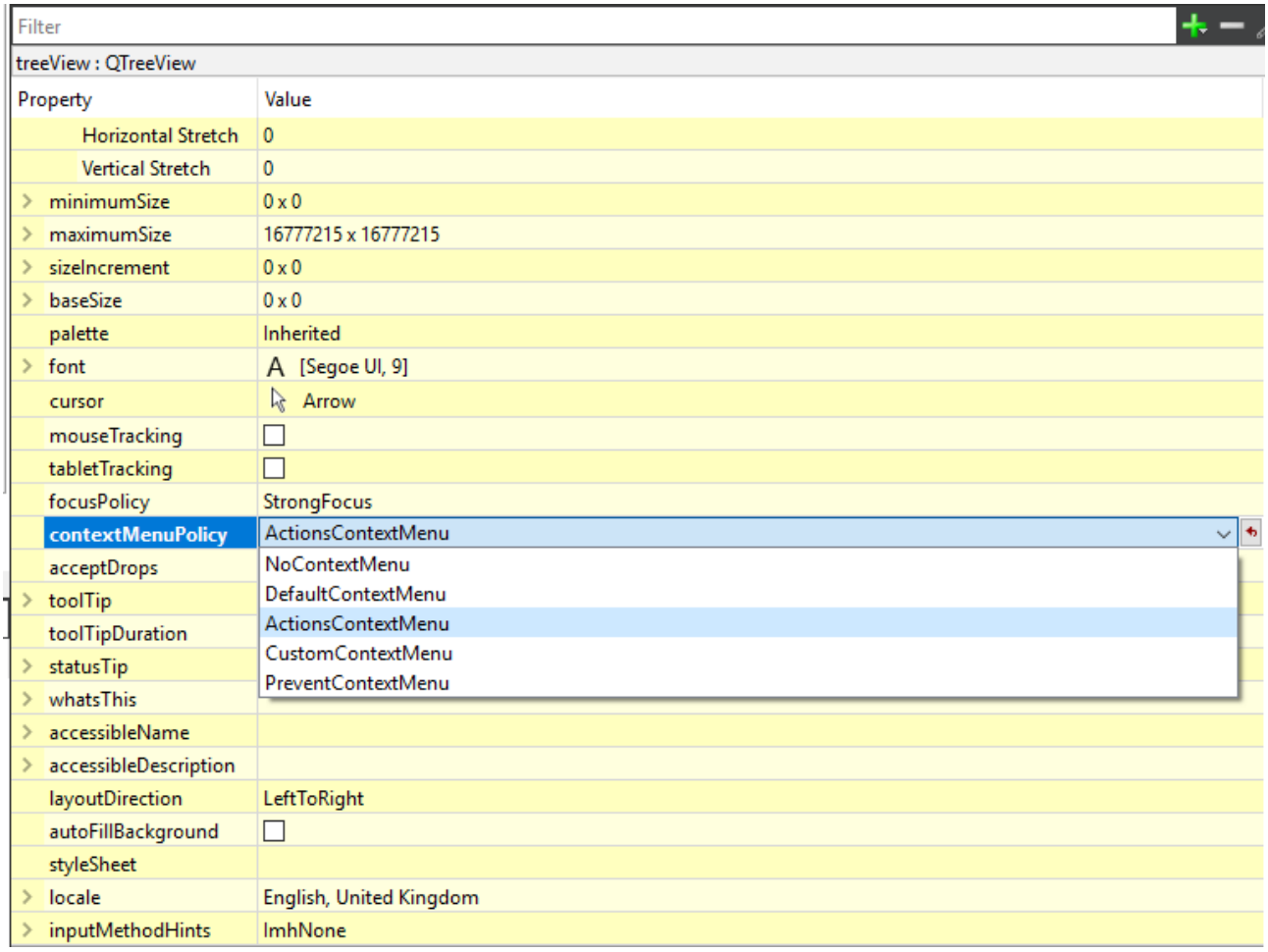

Figure 11: Context Menu

6.4 Signoff criteria

When we check your code, we will be looking for the following:

- Your tree view is working properly.
- You have a tree view with multiple levels, as shown in the figures above.
- We can right click on an item in the tree view and a dialog will appear. The dialog will allow me to edit the item properties:
  - Name
  - Colour (you can represent the colour as three RGB values)
  - isVisible property.
- We should be able to close the dialog and edit other items in the same manner. If I reopen the dialog for any item, previous modifications to that item's properties should have been saved and the saved values will be shown.

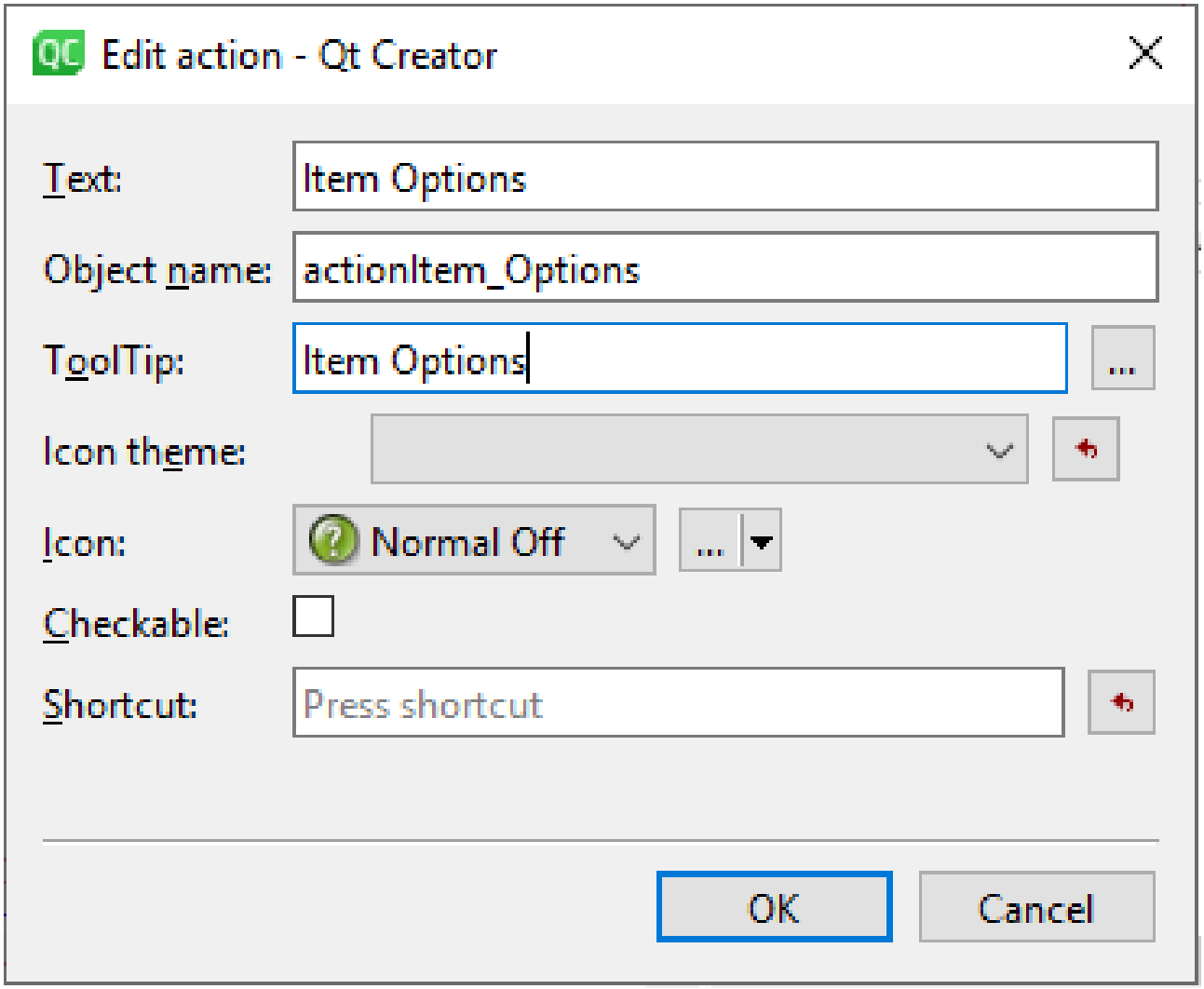

Figure 12: Creating an action that will be linked to the context menu

- An Open File dialog will be available using the toolbar and File menu. When this is used, the item in the tree that is currently selected will have its name property updated to reflect the filename selected.
- At least one button should do something when clicked
- Messages must appear in the status bar when actions occur.