European Organisation for Astronomical Research in the Southern Hemisphere

Programme: E-ELT

Project/WP: E-ELT Telescope Control

Control GUI Developer Guidelines

Document Number: ESO-288608

Document Version: 1

Document Type: Manual (MAN)

Released On: 2018-04-30

Document Classification: ESO Internal [Confidential for Non-ESO Staff]

Owner:	Schilling, Marcus
[Validated by PA/QA:]	Kurlandczyk, Hervé
Validated by WPM:	Kornweibel, Nick
Approved by PM:	Kornweibel, Nick
	Name

Authors

Name	Affiliation
M. Schilling	ESO

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
All	First version

Contents

1. Introduction 5
 1.1 Scope 5
 1.2 Definitions and Conventions 5
  1.2.1 Abbreviations and Acronyms 5
2. Related Documents 6
 2.1 Applicable Documents 6
 2.2 Reference Documents 6
3. All Control GUIs 7
 3.1 Toolkit and Language 7
 3.2 Libraries 7
 3.3 Tooling 7
 3.4 Concepts of User-Friendliness 8
  3.4.1 Time to Point 8
  3.4.2 Cognitive Load 8
  3.4.3 Mental Map 9
  3.4.4 Summary 10
 3.5 Implementation 11
  3.5.1 Custom Widgets 11
  3.5.2 Widget Behaviour 11
  3.5.3 Keyboard 11
  3.5.4 Responsive 11
  3.5.5 Progress/Cancel 12
  3.5.6 Confirm 12
  3.5.7 Colours 12
  3.5.8 Help 12
 3.6 Implementation II (advanced) 13
  3.6.1 Docking 13
  3.6.2 Coordinated Views 14
4. Engineering GUIs 14
 4.1 Tooling 14
 4.2 Implementation 14
5. Instrument Control GUIs 15
 5.1 Implementation 15
6. Operations GUIs 15
 6.1 Implementation 15

7. How-To 16
7.1 App with two widgets and event handler (C++) 16
7.1.1 Starting the code editor 16
7.1.2 Creating a GUI application 16
7.1.3 Adding widgets 17
7.1.4 Adding an event handler 17
7.1.5 Starting the application 18
7.2 App with dynamically loaded UI (Python) 19
7.2.1 Creating the UI 19
7.2.2 Creating the application 19
7.2.3 Adding an event handler 20
7.2.4 Starting the application 20

1. Introduction

This document is the guidelines for developers writing applications with graphical user interfaces for the control systems of the ELT.

It presents rules and recommendations for the behaviour and characteristics of ELT Control GUIs, as well as examples of good GUIs, and explains the underlying human-computer-interaction concepts.

It distinguishes between different kinds of Control GUIs, distinct by their scope and audience, and refers to this as the “layer” of the GUI. In concrete terms, an Engineering GUI is low layer, an Instrument Control GUI is middle layer, an Operations GUI is high layer.

Chapter 3 applies to any ELT Control GUI.
Chapter 4 applies to Engineering GUIs.
Chapter 5 applies to Instrument Control GUIs.
Chapter 6 applies to Operations GUIs.

Chapter 0 provides How-To recipes and code examples.

1.1 Scope

This document presents guidelines for developers writing applications with graphical user interfaces for the control systems of the ELT.

1.2 Definitions and Conventions

1.2.1 Abbreviations and Acronyms

The following abbreviations and acronyms are used in this document:

GUI	Graphical User Interface
MVC	Model-View-Controller implementation pattern

2. Related Documents

2.1 Applicable Documents

The following documents, of the exact version shown, form part of this document to the extent specified herein. In the event of conflict between the documents referenced herein and the content of this document, the content of this document shall be considered as superseding.

2.2 Reference Documents

The following documents, of the exact version shown herein, are listed as background references only. They are not to be construed as a binding complement to the present document.

RD1 *Qt 5 Documentation*

http://doc.qt.io/

3. All Control GUIs

In the current version, this document focuses on desktop-based Control GUIs.

3.1 Toolkit and Language

Control GUIs are developed on the Qt 5 toolkit.

Control GUIs are written in Python 3, or C++.

Python 3, using the PySide2 binding module, is the preferred choice.

C++ is required for a GUI component if any of the following applies:

- It has extraordinary performance needs.
- It is a custom widget. Note that a python binding should then be implemented, too.

3.2 Libraries

Third-party components to be integrated into a Control GUI must qualify as Qt plugins, i.e. must be importable as plugins by Qt.

For automated tests, use the testing framework of the ELT Dev Env.

Usage of Qt Quick Controls 2 (since Qt 5.7) or Qt Labs Controls (in Qt 5.6) is currently not allowed. This may change in a later version of this document. Qt Quick Controls v1 must not be used.

3.3 Tooling

For designing the static parts of your application's GUI, we recommend use of the graphical GUI builder Qt Designer. It comes integrated into the development editor Qt Creator. See the How-To in this document (chapter 0) for an introduction to this tool.

By using the Qt Designer, you produce declarative GUI descriptions. The Qt Designer stores them as XML files with “.ui” file extension. The “.ui” file can either be converted to code, or an application can load it at run-time to dynamically create the GUI from it. Again, see the How-To section (chapter 0) for an example. Good documentation on this topic is available at http://doc.qt.io/qt-5/designer-using-a-ui-file.html.

3.4 Concepts of User-Friendliness

This section outlines a few concepts from Human Computer Interface (HCI) research.

HCI is about efficiency of interaction:

- What format should the machine's output have, so a human can easily consume it?
- What format should the machine's input have, so a human can easily produce it?

One way to put it is that HCI strives to optimise the protocol between two data processing systems, and draws its results from investigating the capabilities and weaknesses of the human "hardware".

3.4.1 Time to Point

Reduce the "Time to Point"

- Decrease distances that mouse has to travel
- Decrease amount of mouse-clicks necessary
- Offer keyboard shortcuts
- Offer mouse gestures

To succeed with "Time to Point", you need a good idea of the most common **patterns of use** of your end-users. This is complicated by the fact that different users (Engineers, Developers, Integrators, Operators, Astronomers) have very different needs. What makes sense to you, may well be cumbersome for them. If you can, involve your end-users during development.

3.4.2 Cognitive Load

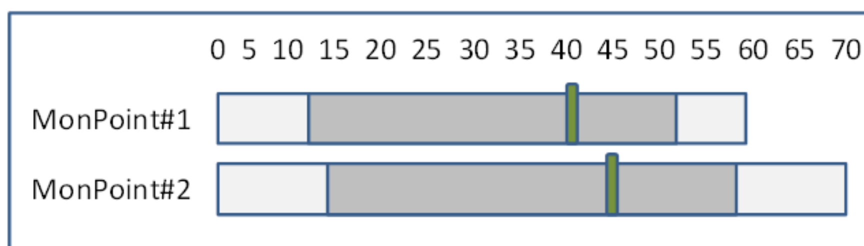
Reduce the "Cognitive Load" by offering information in a way humans can perceive it best.

Example:
The two figures below show the same information, the graphical view is much preferable.

Not great:

Value	Minimum	Current	Maximum	Upper Limit
MonPoint#1	12.2	41.7	52.0	60
MonPoint#2	14.7	45.3	58.3	70

Same info, but better:



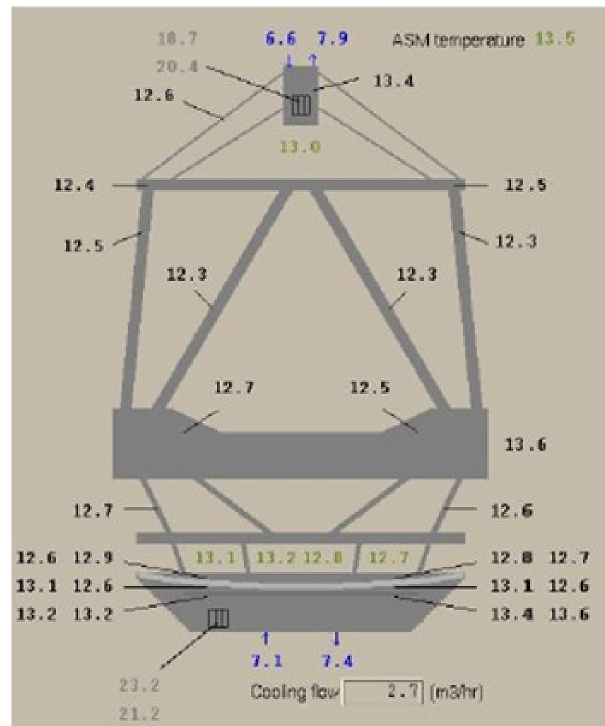
3.4.3 Mental Map

Help your user acquire a "Mental Map" of the system. Visualise the context of an item (like a displayed value or an offered command) and its relation to other items. This does not need to be graphical, but a graphical view is often a good choice.

Not great:

MonitorPoint	Temp. (C)	MonitorPoint	Temp.(C)
MonPnt#1	12.4	*MonPnt#16*	14.2
MonPnt#2	13.2	*MonPnt#17*	13.2
MonPnt#3	13.7	*MonPnt#18*	12.4
MonPnt#4	12.7	*MonPnt#19*	12.2
MonPnt#5	12.0	*MonPnt#20*	13.2
MonPnt#6	7.4	*MonPnt#21*	6.6
MonPnt#7	13.2	*MonPnt#22*	12.4
MonPnt#8	12.4	*MonPnt#23*	7.4
MonPnt#9	12.2	*MonPnt#24*	7.9
MonPnt#10	12.4	*MonPnt#25*	7.1
MonPnt#11	13.2	*MonPnt#26*	23.2
MonPnt#12	12.4	*MonPnt#27*	21.2
MonPnt#13	12.2	*MonPnt#28*	13.2
MonPnt#14	12.5	*MonPnt#29*	12.6
MonPnt#15	12.6	*MonPnt#30*	12.7

Much better:



The above figure is called a "Synoptic View".

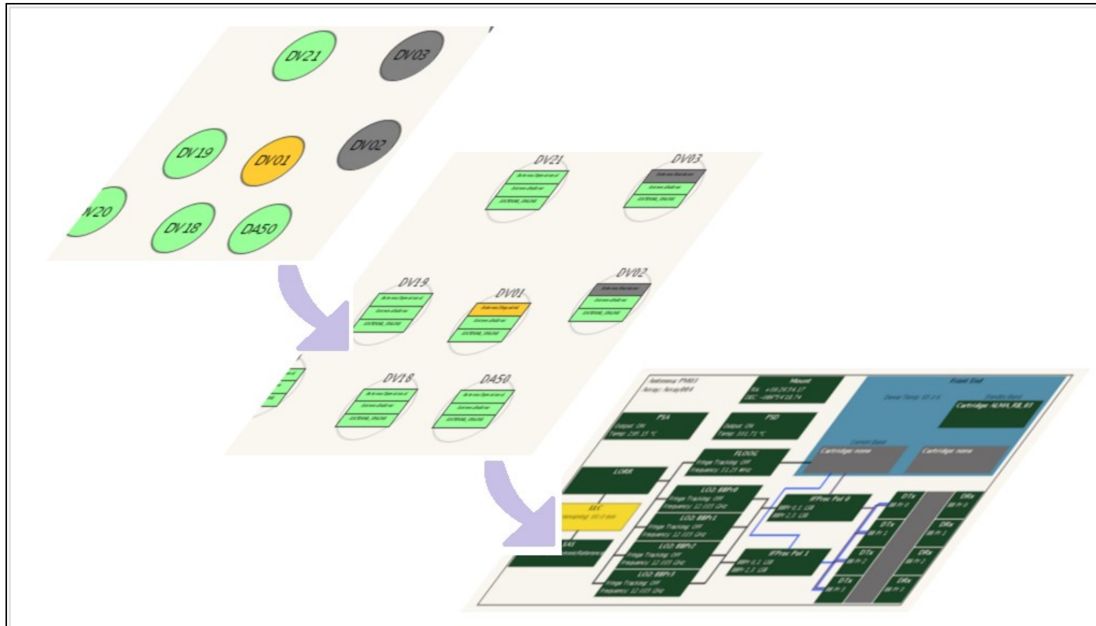
Advanced synoptic views support features like:

- Interactivity
  Examples for interaction methods: left-click to select, left-double-click to zoom and centre, mouse-wheel for zooming in and out, middle-click to pan.

- Semantic Zooming
  On zooming, displayed objects not only change in geometrical size, but also in amount of information. On higher zoom levels, the view shows more detailed information.

  Example:
  This GUI supports 3 levels of semantic zooming (all shown simultaneously in the following figure, the purple arrows represent a zoom-in action). On the bird's eye level, items are represented as colour-coded circles. When the user zooms closer,

the circles turn into squares showing 3 pieces of colour-coded information. Zooming even more, the squares turn into block diagrams showing very detailed information.



3.4.4 Summary

The more relevant a system, the more important to prudently design its UI.

If you think …

- my users are power users, they can master steep a learning curve
- I have no time to make it user-friendly, I'm busy making it stable

we would reply …

users will make more mistakes and fail to react quickly to problems

- when they are cognitively overwhelmed by an over-crowded UI
- when they are impeded by tedious UI navigation in a difficult-to-use UI

Take-aways:

- Know your end-users and their patterns of use
- Present information visually and give context
- Provide keyboard shortcuts for frequent commands
- Use screen space in smart ways

3.5 Implementation

As an application programmer, you have to make sure your application has the following behaviour and characteristics. The ELT GUI Infrastructure provides features to help with this.

3.5.1 Custom Widgets

Your application should only use widgets from the common widget library provided by the ELT GUI Infrastructure.

You may develop and use a custom widget, if none of the existing widgets comes near your needs. In this case, you are required to suggest your widget for addition to the common widget library. If accepted, it can be reused by other ELT developers.

3.5.2 Widget Behaviour

Your application must show the same behaviour for a given widget as all other ELT Control GUIs.

A bad example are two applications using two competing kinds of list boxes with slightly different behaviour: one list box triggers an action instantly on selection, while the other triggers only on selection and pressing Enter. This leads to usage errors.

3.5.3 Keyboard

Your application should support keyboard (mouse-less) operations.

Provide keyboard shortcuts (aka. hotkeys) for frequently used commands. Some technologies, like plots, do hardly allow mouse-less operation. But if your application provides keyboard shortcuts, it must follow the conventions that are already established in other ELT Control GUIs, i.e. use the same shortcut for a given functionality.

Example
See http://doc.qt.io/qt-5/qshortcut.html
Note: From Qt 5.7, you can assign a shortcut to a button directly via its widget properties.

3.5.4 Responsive

Your application must not freeze.

GUI freezes are caused by implementation mistakes, often by hogging the GUI event pump with long-running operations like a network call. This makes the GUI completely unresponsive, including no more updates to its display.

Your application should not take longer than 0.3 seconds to process a single user input. Longer processing needs to run decoupled, by using multi-threading or another asynchronous mechanism. This needs to be kept in mind from the beginning of the implementation.

3.5.5 Progress/Cancel

Your application should show a progress-indicator (e.g. a progress bar) for long-running operations, and should also offer to cancel such operations.

Example
http://doc.qt.io/qt-5/qprogressbar.html

Example 2 (including Cancel)
http://doc.qt.io/qt-5/qprogressdialog.html

3.5.6 Confirm

Your application should get user confirmation, e.g. via pop-up, before doing an irreversible or expensive (or expensive to revert) operation. Otherwise, one wrong click can cost your users a lot of time.

Your application must get user confirmation for an action that concerns safety, such as the propagation of a laser beam.

Example
```
from PySide2.QtWidgets import QMessageBox

if (QMessageBox.No ==
    QMessageBox.question (self, "Confirm", "Shutdown")):
    return
```

Example C++
```
#include <QMessageBox>

if (QMessageBox::No ==
    QMessageBox::question (this, "Confirm", "Shutdown") {
   return;
}
```

3.5.7 Colours

Your application must use the foreseen colour schemes provided by the ESO framework. Provided colour schemes (high-contrast and colour-blind, day/night switch) cater for visually impaired users and improve ergonomics.

3.5.8 Help

Your application must use the ELT GUI Infrastructure's mechanism for creation and display of documentation.
Every control must be covered by the documentation, unless the control's functionality is obvious.

In some existing GUIs, users are unsure about (or have forgotten) the effects of some parts of their complex user interface, e.g. what certain buttons will do to the system. We do not require that you document all controls - those that are obvious may be omitted.

3.6 Implementation II (advanced)

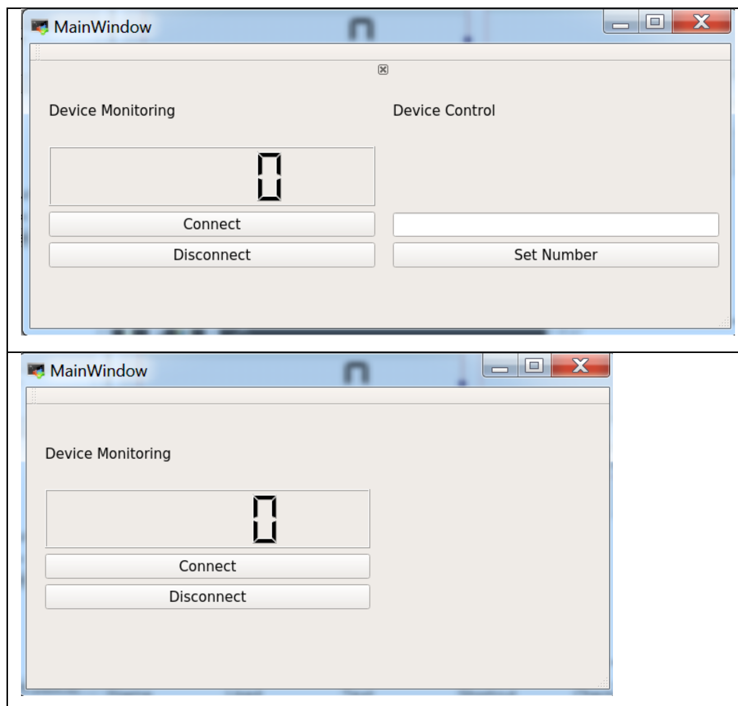
3.6.1 Docking

Whereas some GUIs serve their purpose perfectly fine with an immutable fixed arrangement of widgets, others benefit greatly from letting the user rearrange widgets at run-time.

So-called docking gives end-users some control over which information they want to see, and where. It lets users rearrange, add, or remove parts of the GUI, in order to save screen space or reduce the visual noise.

Docking is usually desirable when a GUI displays large amounts of information: the more information the less likely that all of it is interesting to all users at all times.

Example
In this example, implemented using a QDockWidget ("Dock Widget" in Qt Designer), the user can remove the “Device Control” widget by clicking on the tiny “X” icon.



3.6.2 Coordinated Views

Coordination of Views means that performing a user action in one panel triggers an associated action in another panel.

An example being an IDE that shows two sub-windows side by side: a file system browser and an editor window. When the user clicks on an entry in the file system browser, the editor accordingly shows that file.

This user-friendly feature can greatly reduce the amount of clicks, and simplify the user's navigation in the data space presented by your GUI.

Depending on the scenario, applications can let the user switch the coordination on and off, or have it always on.

In advanced setups like operator consoles, coordination of views not only happens between the internal views of one application, but also between separate GUI applications. The ELT GUI Infrastructure is currently not ready to support this, but this is expected to change in a future version.

4. Engineering GUIs

With Engineering GUIs, we denote Control GUIs for Engineers or Developers.

For hardware, they allow a hardware engineer to access the full set of properties and all commands of a device.

For software, they enable a software developer to access logs and other debug data and often to backdoor functions of a software module.

4.1 Tooling

At this time, the ELT GUI Infrastructure does not provide dedicated tooling for Engineering GUIs. So they are developed with the same tooling as every other Control GUI. This is expected to change in the future.

4.2 Implementation

Engineering GUIs are written by power users for power users.

While we always discourage to throw Engineering GUIs at users from higher layers, experience shows that it is likely to happen - at least temporarily while higher-layer GUIs have not been fully developed.

Please keep this creep-up effect in mind while developing an Engineering GUI.

- Use Docking (section 3.6)
- Design your application so dangerous interactions can be disabled.

5. Instrument Control GUIs

With Instrument Control GUIs, we denote Control GUIs for instrument developers and integrators.

They allow a developer or integrator to understand the status of an instrument, and verify the interworking between the instrument's devices.

5.1 Implementation

Instrument Control GUIs are recommended to employ the advanced usability concepts described in section 3.4.

6. Operations GUIs

With Operations GUIs, we denote Control GUIs for Operators and Astronomers on Duty.

They allow an operator to understand the high-level status of hardware and software, and to dig into problems of hard-/software in an ad-hoc and quick manner.

They allow an astronomer to understand the status, efficiency, and potentially quality of a running observation.

We strongly discourage to throw Engineering GUIs at Operators and Astronomers:

- Often not user-friendly enough, and do not represent the production procedures.
- Typically let the user do changes dangerous to the system stability.
- Sometimes not good citizens, causing high network load and system stress.

6.1 Implementation

Operational GUIs are strongly recommended to employ all advanced usability concepts described in section 3.4.

7. How-To

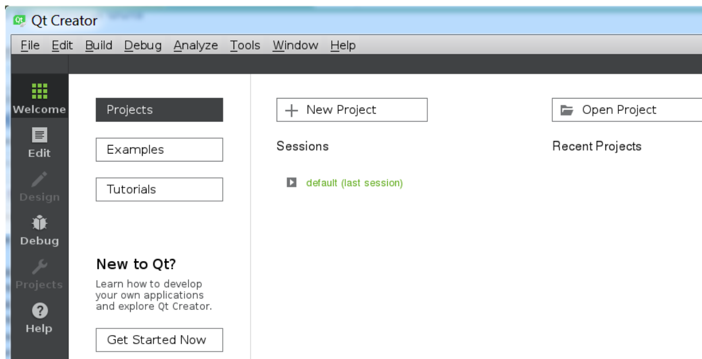
7.1 App with two widgets and event handler (C++)

7.1.1 Starting the code editor

On a host with the ELT Development Environment installed, on a terminal, type:

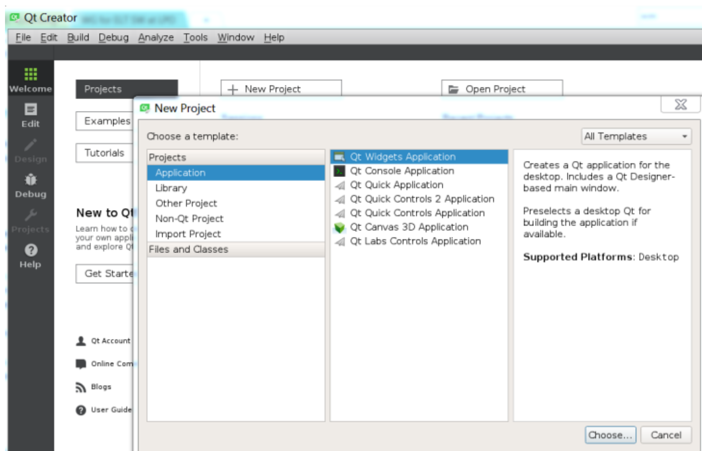
```
> qtcreator &
```

This will bring up Qt Creator, the Qt code development tool:



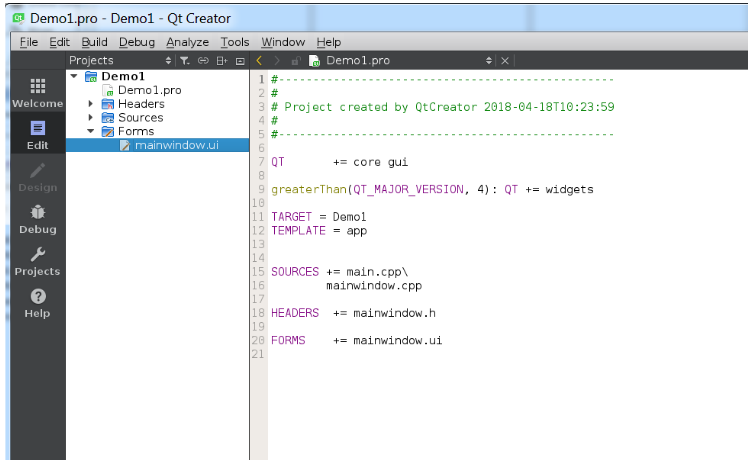
7.1.2 Creating a GUI application

Create a new project of type `Qt Widgets Application`, with Name “Demo1”.

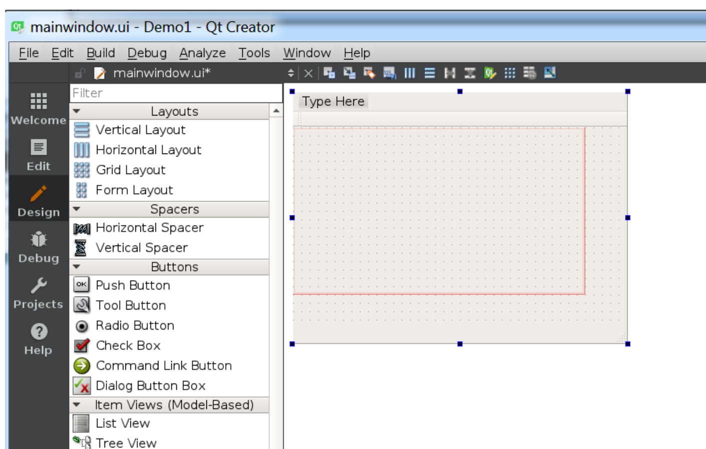


7.1.3 Adding widgets

After the project is created, open the `mainwindow.ui` file.



This will open the Qt Designer, and show an empty grey form.



From the component collection on the left, drag a `Grid Layout`, a `Push Button`, and a `LCD Number` onto the grey empty form.

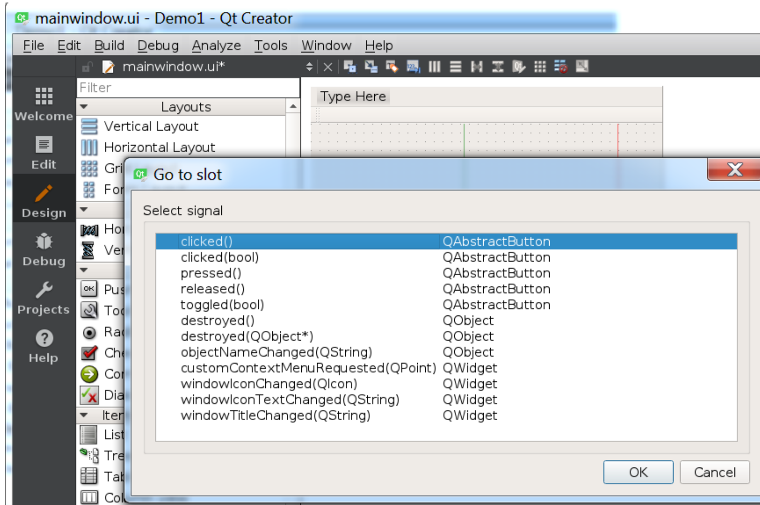
You set the caption of the button in the widget properties on the lower right, look for `QAbstractButton/text` (i.e. "`text`" in the "`QAbstractButton`" section).

Note: In the widget properties, do not modify those settings that have an effect on the look and feel of the widgets: `palette`, `styleSheet`, etc.

7.1.4 Adding an event handler

On the `LCD Number` widget, set the `QObject/objectName` property to "`numberA`".

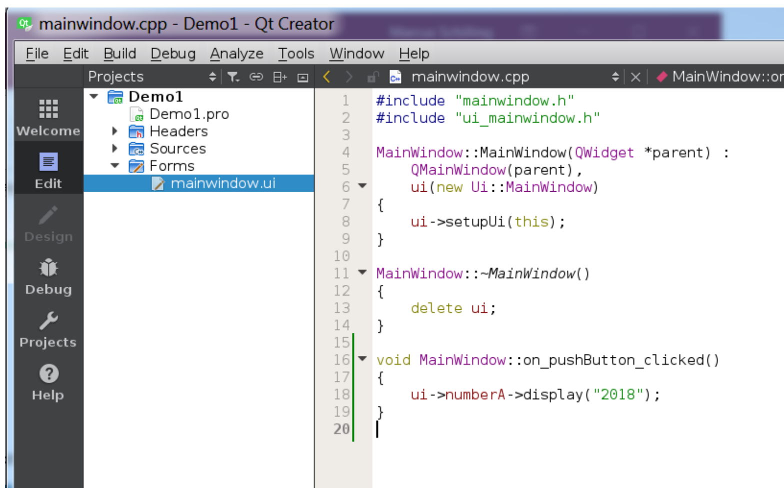
On the `Push Button` widget, → right-click → "Go to slot" will bring up the following information:



Selecting the first entry takes you back to the code editor.

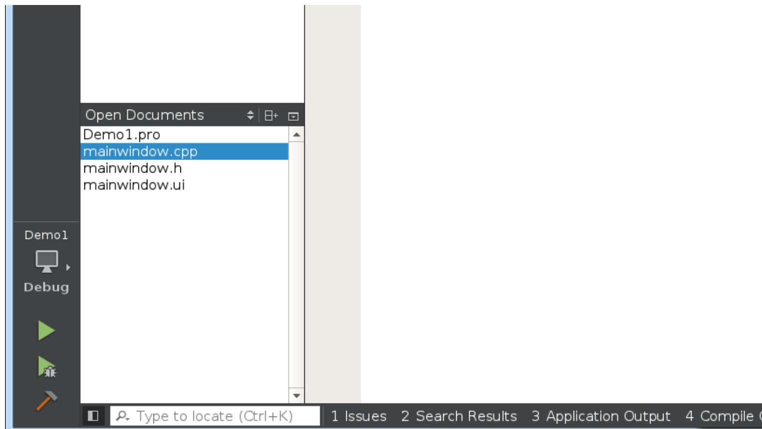
At the cursor prompt, type:

```
ui->numberA->display("2018");
```



7.1.5 Starting the application

Pressing the green "Play" button at the bottom left will run your application.



Congratulations!

7.2 App with dynamically loaded UI (Python)

7.2.1 Creating the UI

We start from the demo project described in the previous How-To. It already provides us with a ".ui" file that we will use in the following.

7.2.2 Creating the application

In Qt Creator, on the "Demo1" top-level node in the project tree, right-click → "Add New…" → Python → Python File. For Name, enter "main.py".

The new file "main.py" opens in the editor window. Type:

```
import sys

from PySide2.QtUiTools import QuiLoader
from PySide2.QtWidgets import Qapplication
from PySide2.QtCore import Qfile
from PySide2 import QtCore

if __name__ == "__main__":
    app = QApplication(sys.argv)
    file = QFile("mainwindow.ui")
    file.open(QFile.ReadOnly)
    loader = QUiLoader()
    window = loader.load(file)
    window.show()

    # ... more code to come ...

    sys.exit(app.exec_())
```

7.2.3 Adding an event handler

Your python application is now already able to construct the widget tree from your ".ui" file and render it on screen. The C++ event handler that you created in the previous How-To is not part of your python application, and needs to be re-done. Here's how:

In "main.py" after the placeholder comment, type:

```
    # ... more code to come ...

    numberA = window.findChild (QtCore.QObject,"numberA")
    pushButton = window.findChild (QtCore.QObject,"pushButton")

    def set_number():
        numberA.display("2018")

    pushButton.clicked.connect (set_number)
```

7.2.4 Starting the application

On a terminal, in directory "`Demo1`", type:

```
> python main.py
```

All done!

.oOo.