European Organisation for Astronomical Research in the Southern Hemisphere

Programme: ELT

Project/WP: Central Control

ELT Control UI Toolkit (CUT) User Manual

Document Number: ESO-518855

Document Version: 2

Document Type: Manual (MAN)

Released on: 2025-01-22

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

Owner:	Hoffstadt Urrutia, Arturo
Validate1 by PM:	Kownweibel, Nick
Validate2 by SE:	González Herrera, Juan Carlos
Validated by PE:	Biancat Marchet, Fabio
Approved by PGM:	Tamai, Roberto
	Name

Release

This document corresponds to `cut`[1] v3.4.1-rc1.

Authors

Name	Affiliation
Arturo Hoffstadt Urrutia	ESO

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
TODO	TODO

[1] https://gitlab.eso.org/ecos/cut

Table of Contents

1 Introduction

The Control UI Toolkit (CUT) is a suite of libraries, tools and examples used to develop Graphical User Interfaces (GUI) for the ELT. CUT is built using Qt a graphical rendering library, and Taurus as a powerful Model-View-Controller (MVC) layer that allows you to bind datapoints in the control system servers and services to widgets in your graphical application.

The toolkit is being developed by ESO to help developers implement their GUIs.

This Manual does not go into the details of design and the different considerations taken during the design. More information about it can be found in ELT Control GUI Toolkit Design[1].

Warning: Users should avoid using APIs and facilities that are not documented in this manual. Such undocumented APIs and facilities are intended for internal use and should not be relied on by an end user.

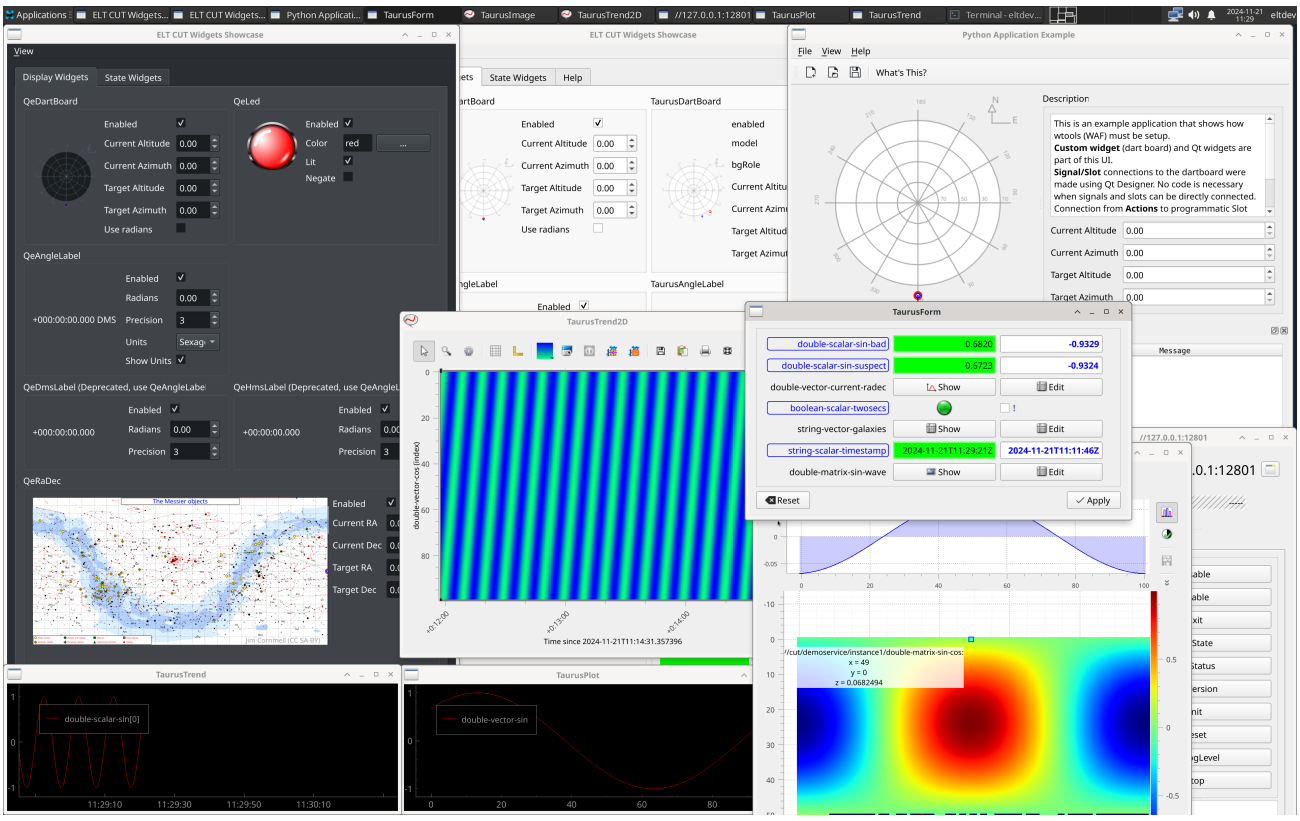


Fig. 1: Control UI Toolkit demonstration

[1] https://pdm.eso.org/kronodoc/HQ/ESO-377114/1

1.1 Scope

This document is the developers manual for the Control UI Toolkit. The intended audience are ELT users, consortia developers or software quality assurance engineers.

1.2 Definitions

Widget
Display or Control element of a GUI. Widgets present data or information to the user of the GUI, and implement interaction capturing input from the user, making it available for the application to process. Widgets are intended to be a small reusable unit of GUIs.

Composite widget
a widget that is implemented by composing several other smaller widgets. Implementing a composite widget commonly achieves a complex tasks by cooperation between its parts.

Sequential Programming
a programming paradigm in which the flow of control is predetermined by a sequence of instructions.

Asynchronous Programming
a programming paradigm in which the program reacts to events. Different kinds of inputs can be translated into events, such as key strokes, mouse movements, network connection status, battery level, etc.

Note: Asynchronous programming is not restricted by multi-threading, but some long running responses to events certainly benefit from it.

Event Loop/Engine/Pump
Asynchronous Programming needs to implement an additional section of the program that gathers input, creates events, and trigger the responses to those events. One possible implementation of this, and the one that Qt uses, is an Event Loop (See QEventLoop[2]).

Python module
A python file, may contain several classes, functions and scripts.

Python package
A directory, and a `__init__.py` file. This packages contains all other python modules in inside of the directory.

WAF module
A directory with a wscript, that is not a top-level project script.

WAF project
The first wscript of a project. It also must document the external dependencies of the project.

[2] https://doc.qt.io/qt-5/qeventloop.html

1.3 Conventions

Note: Through this document, the author will refer to widgets by name using the following notation: *TextOnTheWidget* kindOfWidget

For example: *Accept* button or *Open File* dialog

Note: When referencing code, like methods or class names, this notation will be used `closeEvent(QEvent *event)`.

Note: Executable programs will be depicted like this **`cutDemoService`**.

Note: When navigating through a menu, context menu, or a hierarchy of options, the document will use the following notation to describe the consecutive options the reader has to follow:

Start → *Programs*

Note: Special keystroke combination that the reader has to input will be given in the following format:

`Control-x Alt-f`

1.4 Overview

The Control UI Toolkit foresees the following software artefacts.

Note: Since the Control UI Toolkit is being developed in an iterative way, the current release may not yet include all the artefacts depicted above. It is also possible that some artefacs are covered only partially, they will be refined to their full functionality in future versions.

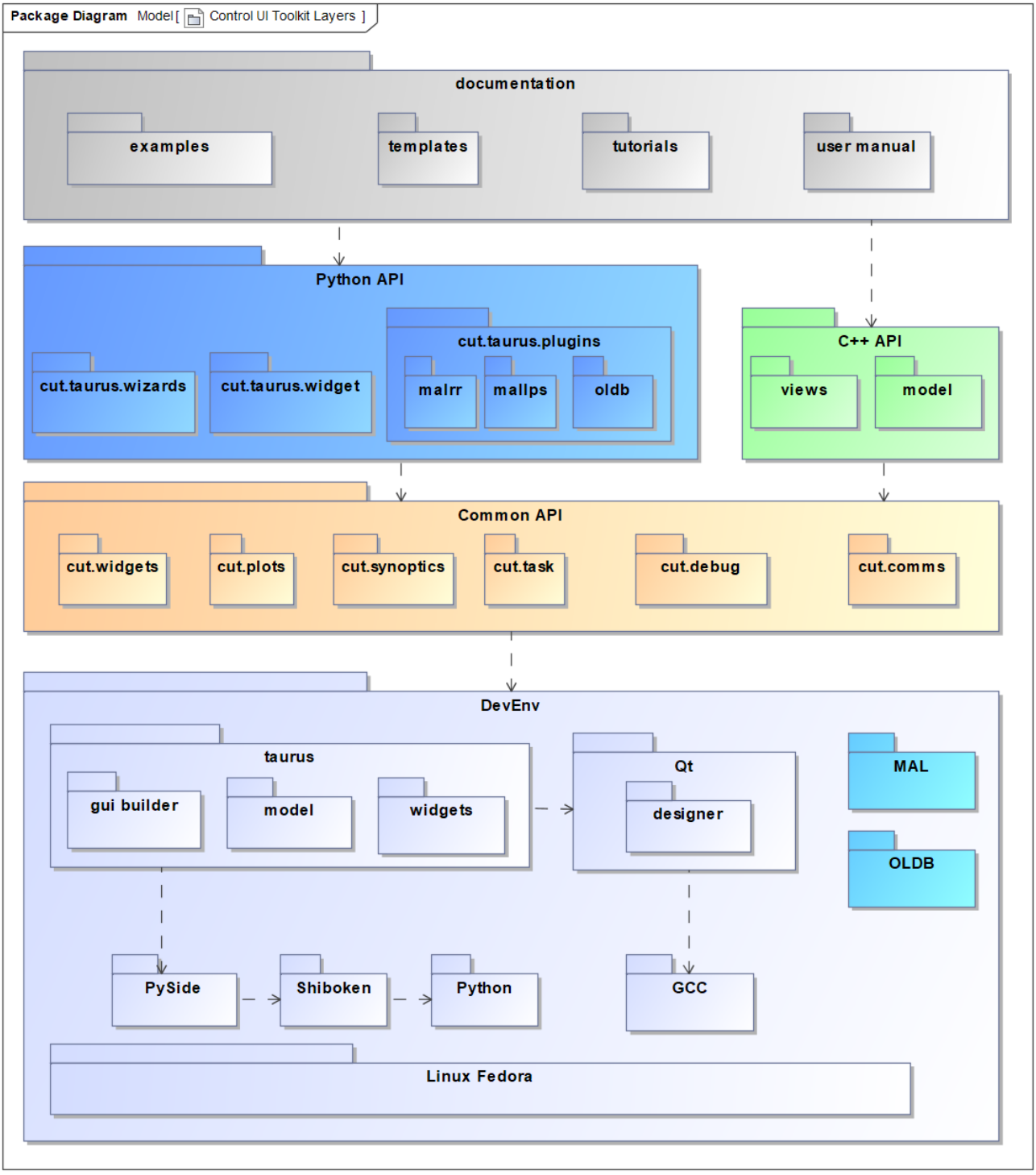


Fig. 2: Control UI Toolkit deliverables in a Layers view

1.5 Source Code

To access the CUT sources in Gitlab follow this link:

- https://gitlab.eso.org/ecos/cut/-/releases

1.6 Previous Release

https://gitlab.eso.org/ecos/cut/-/releases/v1.2.11

2 Release Notes

2.1 Summary

This release includes the following artefacts:

- Control UI Toolkit 3.4.1-rc1

And depends on the following:

- ELT Development Environment (ELT DevEnv)
- Middleware Abstraction Layer (CII MAL)
- Common Integration Infrastructure Services (CII SRV)
- Pointing Kernel (PTK)
- Qt
- PySide
- Shiboken

All dependencies are satisfied when using DevEnv. For the purpose of dependency versions, CUT adapts to the versions delivered with DevEnv.

What is the purpose of this release?

This release adds support for Qt6 and updates all documentation and examples to Qt6 version. At the same time, it introduces changes to wscripts and code to be version agnostic whenever possible, to avoid most of the tedious version changes search/replace tasks for future versions.

What is the scope of this version?

Added support for Qt6.

The most important features of this release are:

Widgets

- Angle Label
- Dartboard
- Led
- State Widget
- Detailed State Widget
- RaDec Widget

- OLDB Heartbeat
- Notification Banner
- Command Combo Box
- Python Bindings
- Qt Designer Plugins

Data Access

- OLDB read/write access, either using subscription or polling. All OLDB datatypes supported.
- MAL Requests declarative support. Returns values: basic datatypes and void. Arguments: basic datatypes and void.

Long Running Operations

- Task module

Documentation

- User Manual (this document)
- Introduction to CUT
- Application Example, Template and Tutorial
- Widget Library Example, Template and Tutorial
- API Documentation

Limitations and Constraints

This version provides the basic functionality to implement a GUI using the provided libraries. However, not all features are yet available or optimised.

- Subscription to OLDB datapoints is done at the Taurus Attribute level, and not yet the Device level. The user needs to specify the complete URI to the datapoint. In the future, the user will be able to create a TaurusForm widget that list and subscribe automatically to every datapoints in a tree branch of the OLDB.
- Matrix datapoint cannot write back to the OLDB. This is a bug, and will be fixed for next version.

Warning: Disclaimer

ESO does not warrant that the provided functions of the Control UI Toolkit will meet all requirements or that the operation of the components and libraries will be flawless. ESO does not ensure that solutions included in this version of the Control UI Toolkit are not subject to changes in future releases. The future upgrade to newer versions of Core Integration Infrastructure (CII) and adaptation to CCS development standards may introduce significant modifications to the actual

interfaces and services. While every precaution has been taken in the development of the Control UI Toolkit software and in the preparation of the documentation, ESO assumes no responsibility for errors or omissions, or for damage resulting from the use of the software or of the information contained in the documentation.

Note: The Control UI Toolkit is distributed outside ESO for the development of applications related to the ELT Project and ruled by the "General Conditions of ESO Contracts". Any other use is not permitted without prior authorization from ESO. The rights of third party products, whose software is included for convenience in the development environment, are ruled by their copyright notice included in their software.

2.2 Dependencies

Development Environment ELT DevEnv

The ELT Linux Development Environment (DevEnv) comprises a collection of hardware, software procedures and tools for the developing, testing and debugging of software components for the ELT. It has to support large-scale and long-term maintenance of software.

Further information regarding the ELT Development Environment can be found here:

- https://www.eso.org/~eltmgr/RELEASE_NOTES
- https://gitlab.eso.org/ecos/eltsw-docs/-/wikis/ELT-Control-Documentation-Links

Core Integration Infrastructure CII

The Control UI Toolkit makes use of the CII MAL (Middleware Abstraction Layer) for examples services, and a currently under development Taurus plugin. The two main mechanisms used are request/reply and publish/subscribe.

- MAL API[3]
- MAL Mappings[4]
- MAL ICD Generation[5]

In addition, this version also supports:

- CII OLDB (Online Database)

[3] https://pdm.eso.org/kronodoc/HQ/ESO-348600/1
[4] https://pdm.eso.org/kronodoc/HQ/ESO-348601/1
[5] https://pdm.eso.org/kronodoc/HQ/ESO-348602/1

Qt

Qt is a multiplatform graphical rendering library, used in an impressive array of open source projects. It provides a complete C++ framework, of which the Control UI Toolkit uses its graphical rendering libraries, and widgets.

https://www.qt.io/product/framework

PySide

Python bindings for the Qt libraries. It is provided by Qt. It uses Shiboken as bindings generator.

https://wiki.qt.io/Qt_for_Python

Shiboken

Python bindings generator from Qt. It facilitates the creation of Qt-based bindings through their highly reusable typesystem framework.

https://doc.qt.io/qtforpython/shiboken2/index.html

PTK

Pointing Kernel is the ELT library that provides positional astronomy routines.

https://gitlab.eso.org/ecos/ptk

msgsend

The msgsend[6] is a python application that provides communication with any MAL ICD defined interface. It is used by the Control UI Toolkit in the currently under development Taurus plugin to access MAL applications.

https://gitlab.eso.org/ccs/msgsend

[6] https://gitlab.eso.org/ccs/msgsend

3 Installation

Tip: When using a DevEnv-based machine, the Control UI Toolkit `elt-cut` and `elt-cut-devel` RPM packages are provided in the respective `platform` repository. Developers and users should manually install it, as it is no longer part of the DevEnv set of RPM packages.

3.1 Using RPM Packages

In case it is not installed, the user may install it executing as root:

```
$ dnf -y install elt-cut-devel
```

Beta Versions are no longer available for DevEnv 5+.

3.2 Manual Compilation

If the user wants to compile and test a development version of the Control UI Toolkit all dependencies are satisfied by the DevEnv. Use the following procedure to produce a compiled version of CUT.

1. Start the Display Manager executing as root (not required but recommended)

   ```
   $ systemctl set-default graphical.target
   ```

2. Append the following lines to your `$HOME/modulefiles/private.lua`

   ```
   local homedir = os.getenv("HOME")
   local introot = pathJoin(homedir .. "/volume/INTROOT")
   local dataroot = pathJoin(homedir .. "/volume/DATAROOT")
   setenv("PREFIX", introot)
   setenv("INTROOT", introot)
   setenv("DATAROOT", dataroot)
   setenv("CII_LOGS", "/var/log/elt")
   prepend_path("MODULEPATH", pathJoin(introot, "/etc/modulefiles"))
   prepend_path("PKG_CONFIG_PATH", pathJoin(introot, "/lib64/pkgconfig"))
   prepend_path("CFGPATH", dataroot .. "/config/persistent_repo")
   prepend_path("WDEPPATH", pathJoin(introot, "/share/wdep"))
   prepend_path("QT_PLUGIN_PATH", "/usr/lib64/qt6/plugins/")
   prepend_path("QT_PLUGIN_PATH", pathJoin(introot, "/lib64"))
   load("introot")
   try_load("ciisrv")
   try_load("ecsif")
   try_load("mal")
   ```

(continues on next page)

(continued from previous page)

```
-- This FIXES the problem when expanding on bash with TAB
execute{cmd="shopt -s direxpand", modeA={"load"}}
```

3. Logout of the session, and log back in. If you are currently in a graphical terminal, you need to logout of the graphical session as well.

4. Execute the following lines in your personal account:

```
$ getTemplate -d introot $INTROOT
$ git clone git@gitlab.eso.org:ecos/cut.git
$ cd cut
$ git lfs install
$ git lfs fetch
$ git lfs checkout
$ waf configure build install
$ mkdir -p $HOME/.config/taurus
$ cp taurus.ini $HOME/.config/taurus
```

As a result, CUT will be installed at the `$INTROOT` location.

3.3 Vagrant Virtual Machine

ELT DevEnv is delivered also as a Vagrant box https://app.vagrantup.com/eltdev.

For GUI development it is recommended to use a local VM to get the lowest latency.

Tip: In case you need a local VM, we suggest to use **`vagrant`**. CUT has a vagrant module[7] that is able to provide a local VM.

This Vagrantfile and provisioning script allows to update to the latest DevEnv version or to adopt the Beta release. Please refer to the Vagrantfile to instructions on how to use it.

[7] https://gitlab.eso.org/ecos/cut/-/tree/master/util/vagrant

4 Tutorials

4.1 Beginners Tutorial

This tutorial will teach you several basic elements from Qt, Taurus, and GUI development. Its purpose is to show simple code, and simple UI behaviour, not to produce final applications.

Note: *Disclaimer:* Several of the examples exposed in this document are modifications to examples found in the Taurus Developers Guide[8].

Starting the Demo Service

CUT makes use of CII OLDB services, so please have them configured and running.

```
$ cii-services start all
CII Services Tool (20240307)

About to start systemd-journald logrotate cii-oldb-default-redis cii-oldb-calc-
↪scheduler jaeger-all cii-oldb-calc-daemon rsyslog
```

The **cutDemoService** is provided as part of the installation. This **cutDemoService** publishes a set of datapoints both to OLDB and PS. The **cutDemoService** will be used through this document, acting as a server. To start it:

```
$> cutDemoService
INFO - Demo Service
Initializing datapoints...
  Datapoints initialized...
Initializing Request-Reply server URI: zpb.rr://127.0.0.1:12801
  Request-Reply URI: zpb.rr://127.0.0.1:12801/Commands?if=::demoserviceif::Commands
  Request-Reply URI: zpb.rr://127.0.0.1:12801/StdCmds?if=::stdif::StdCmds
  Request-Reply URI: zpb.rr://127.0.0.1:12801/Introspection?
↪if=::introspectionif::Introspection
INFO - ZpbServer: Server bound to 'tcp://127.0.0.1:12801
INFO - ZpbServer: Creating server reply listener socket and binding to inproc://
↪zpbServer.0
INFO - ZpbServer: Creating server reply notifier socket and connecting to inproc://
↪zpbServer.0
INFO - ZpbServer: Registering service 'StdCmds' with hash 1187070201.
INFO - ZpbServer: Registering service 'Commands' with hash 1843958668.
INFO - ZpbServer: Registering service 'Introspection' with hash 2208472690.
```

(continues on next page)

[8] https://taurus-scada.org/devel/examples.html

(continued from previous page)

```
  Request-Reply server initialzed
Initializing OLDB...
  cii.oldb:///cut/demoservice/instance1/state   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/status   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/boolean-scalar-fixed   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/boolean-scalar-twosecs   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/int-scalar-add   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/int-vector-sin   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/int-matrix-sin-cos   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/uint64-scalar-sin   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-add   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-sin   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-cos   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-tan   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-sin-bad   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-sin-suspect   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-rand   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-current-radec   MD  DP  VAL  ␣
↪OK
  cii.oldb:///cut/demoservice/instance1/double-vector-target-radec   MD  DP  VAL  ␣
↪OK
  cii.oldb:///cut/demoservice/instance1/double-vector-current-altaz   MD  DP  VAL  ␣
↪OK
  cii.oldb:///cut/demoservice/instance1/double-vector-target-altaz   MD  DP  VAL  ␣
↪OK
  cii.oldb:///cut/demoservice/instance1/double-vector-sin   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-cos   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-tan   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-sin   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-cos   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-sin-wave   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-sin-cos   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/strings/string-scalar-fixed   MD  DP  VAL  ␣
↪OK
  cii.oldb:///cut/demoservice/instance1/strings/string-scalar-timestamp   MD  DP  ␣
↪VAL   OK
  cii.oldb:///cut/demoservice/instance1/strings/string-vector-galaxies   MD  DP  ␣
↪VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/bool   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/string   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/float   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/double   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/int8   MD  DP  VAL   OK
```

(continues on next page)

(continued from previous page)

```
  cii.oldb:///cut/demoservice/instance1/static/int16   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/int32   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/int64   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/uint8   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/uint16   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/uint32   MD  DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/static/uint64   MD  DP  VAL   OK
  OLDB Initialized
Starting Datapoint updates...
  Datapoint updates started
Press CONTROL-C to exit
```

Lets check functionality

Keep this program running as its serves the datapoints needed for this tutorial. Please open a new terminal or a new tab so that we can continue with the workshop. To exit press `Enter` as indicated.

To quickly check if CUT is working, please execute these two lines separately:

```
$ taurus form 'eval:rand()'
```

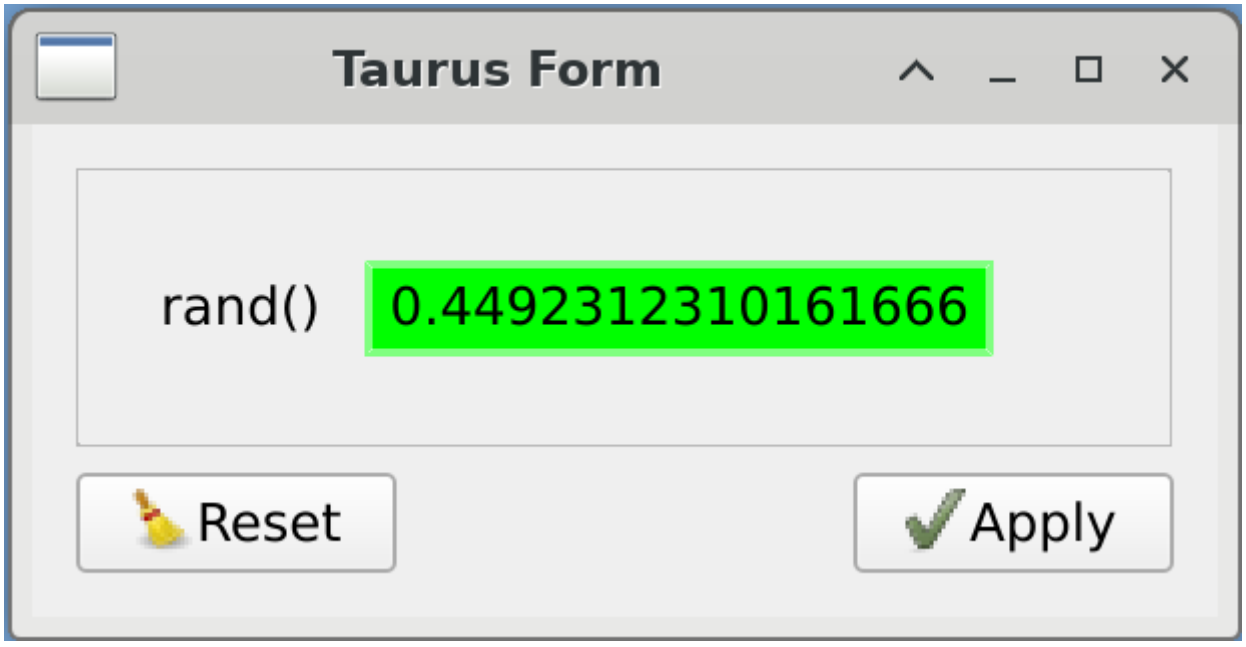


Fig. 1: Taurus form view for eval:rand()

```
$ taurus form 'cii.oldb:/cut/demoservice/instance1/double-scalar-add'
```

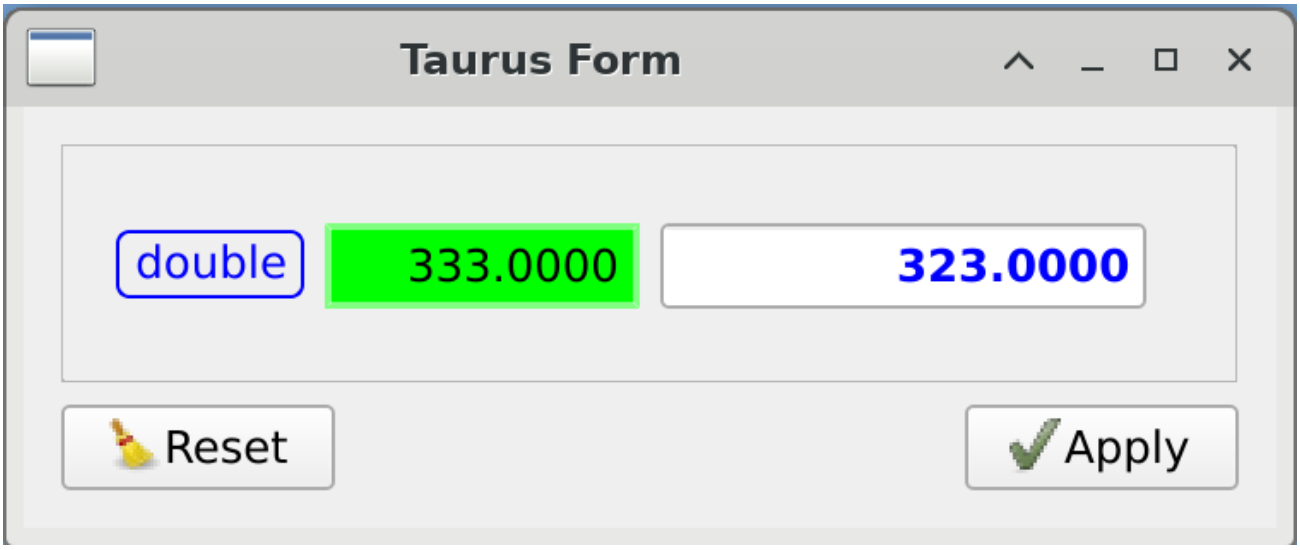


Fig. 2: Taurus form for cii.oldb:/cut/demoservice/instance1/double-scalar-add

Basic Concepts from Qt

Qt is an application development framework. It includes libraries to process sound, text, databases, and the most important: to create UIs. It is written in C++, and has two set of bindings to Python, being PySide[9] the one developed by Qt. PyQt[10] is also very popular, but due to licensing issues, we cannot use it.

Important: When searching for documentation and examples, please use the keyword `PySide6`. Most of the code examples you see in the Internet include examples for both, but always refer to PySide6 examples.

Qt is an asynchronous engine. A developer creates a `main.py` file that serves as entry point for the Qt Application we develop. This application generates the first and most important UI element, and then enters Qt's event engine. From then on, all execution happens in one thread, asynchronously.

```
#!/usr/bin/env python3
import sys
from taurus.external.qt.QtWidgets import QApplication
from taurus.external.qt.QtWidgets import QLabel

if __name__ == "__main__":
    app = QApplication(sys.argv)
    widget = QLabel()
```

(continues on next page)

[9] https://wiki.qt.io/Qt_for_Python
[10] https://riverbankcomputing.com/software/pyqt

(continued from previous page)

```
9     widget.setText("Hello World!")
10    widget.show()
11    sys.exit(app.exec_())
```

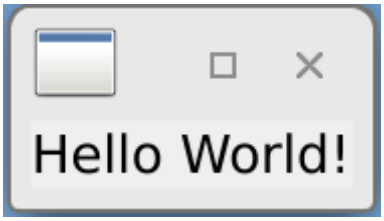


Fig. 3: Qt Hello World! example output

Every Qt application needs a QApplication[11] object. There can be only one per process. At this point, you can add widgets, like a QLabel[12], modify properties of the widget (`setText("Hello World!")`), and command it to be shown.

Qt Signals and Slots with one widget

At this point, something interesting happens. A new windows appear, but it is empty. It is not until the execution of `app.exec_()` that the window start to be drawn. At this point the application has entered Qt event engine. It will not exit the event engine until an event that indicates an exit condition, or a signal is invoked to exit. When that happens, the `app.exec_()` method returns, and we exit the Python program using `sys.exit()`.

A Developer must understand that events will be the drivers of the application behaviors from then on. Mouse clicks and key presses will be registered by Qt, and for each interaction, an event will be generated and entered into a queue. Then, Qt's event engine will take the latest one, and process it.

Following the example above, if a user of this application closes the window, the application will exit. This happens because the window was commanded to be closed by the window manager, which creates an event, the event is fed to the QLabel[13], which by default reacts to QCloseEvent[14] by closing the window. Since there is no more objects capable of handling and creating events in the application, the Qt engine event also automatically exits.

Note: Qt Applications can also be console applications. In this case, the application can be commanded to be exited by using `qApp.quit()`. This can also be done in GUI applications. See QCoreApplication.quit()[15]

---

[11] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QApplication.html
[12] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QLabel.html
[13] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QLabel.html
[14] https://doc.qt.io/qt-6/qcloseevent.html
[15] https://doc.qt.io/qtforpython-6/PySide6/QtCore/QCoreApplication.html#PySide2.QtCore.PySide2.QtCore.QCoreApplication.quit

If a mouse click was done over a widget, Qt will inform of this event to the Window contained by the widget, and pass the coordinates. The Window will check the coordinate, determine which widget is located at those coordinates, and inform the widget of the event. Then the widget will do what is requested of it when a mouse click is detected.

A developer of GUIs should not go much into the details of implementing new events, but instead should use their product: Signals[16]. When the mouse click is processed by the widget, this physical representation of the mouse click generates Signals, which are useful representations of what is happening of the GUIs. A button that gets a mouse click will generate a `clicked` Signal.

Then, the developer does not program against events, but against Signals[17]. As developers, we program applications reacts to Signals[18]. Signals are already provided by Qt, and we react to them by connecting a Signal[19] to a Slot[20].

When a Signal[21] is connected to a Slot[22], Qt event engine will automatically invoke the Slot[23] method.

```
#!/usr/bin/env python3
import sys
from taurus.external.qt.QtWidgets import QApplication
from taurus.external.qt.QtWidgets import QPushButton

if __name__ == "__main__":
    app = QApplication(sys.argv)
    widget = QPushButton()
    widget.setCheckable(True)
    widget.clicked.connect(app.quit)
    widget.setText("If you press this button\n the application will quit")
    widget.show()
    sys.exit(app.exec_())
```

This case is rather simple: QPushButton[24].clicked() signal is actually inherited from QAbstractButton[25] class. Signals[26] have a method connect() which accepts as argument the Slot which will be executed.

Now, when the user of the application click the button, `app.quit()` will be execute, and therefore the application ends.

[16] https://doc.qt.io/qt-6/signalsandslots.html#signals
[17] https://doc.qt.io/qt-6/signalsandslots.html#signals
[18] https://doc.qt.io/qt-6/signalsandslots.html#signals
[19] https://doc.qt.io/qt-6/signalsandslots.html#signals
[20] https://doc.qt.io/qt-6/signalsandslots.html#slots
[21] https://doc.qt.io/qt-6/signalsandslots.html#signals
[22] https://doc.qt.io/qt-6/signalsandslots.html#slots
[23] https://doc.qt.io/qt-6/signalsandslots.html#slots
[24] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QPushButton.html
[25] https://doc.qt.io/qt-6/qabstractbutton.html
[26] https://doc.qt.io/qt-6/signalsandslots.html#signals

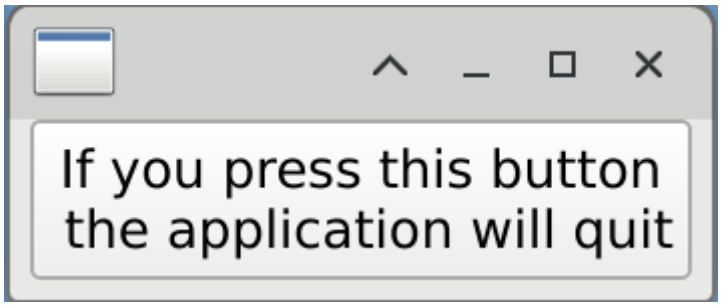


Fig. 4: Signal Slot basic connection

Qt Layouts

At this point we can create one widget, and show it. But this is seldom the case. To create an application with more than one widget we use Layouts.

```python
#!/usr/bin/env python3
import sys
from taurus.external.qt.QtWidgets import QApplication
from taurus.external.qt.QtWidgets import QWidget
from taurus.external.qt.QtWidgets import QPushButton
from taurus.external.qt.QtWidgets import QLineEdit
from taurus.external.qt.QtWidgets import QHBoxLayout

if __name__ == "__main__":
    app = QApplication(sys.argv)
    panel = QWidget()
    layout = QHBoxLayout()
    widget1 = QPushButton(panel)
    widget1.setText("Push me!")
    widget2 = QLineEdit(panel)
    widget2.setText("Enter text here")
    layout.addWidget(widget1)
    layout.addWidget(widget2)
    panel.setLayout(layout)
    panel.show()
    sys.exit(app.exec_())
```

In this example, we can see three widgets: a QWidget[27] used as container, a QPushButton[28], and a QLineEdit[29]. A new kind of object is created, the QHBoxLayout[30]. The QHBoxLayout[31] puts every

[27] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QWidget.html
[28] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QPushButton.html
[29] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QLineEdit.html
[30] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QHBoxLayout.html
[31] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QHBoxLayout.html

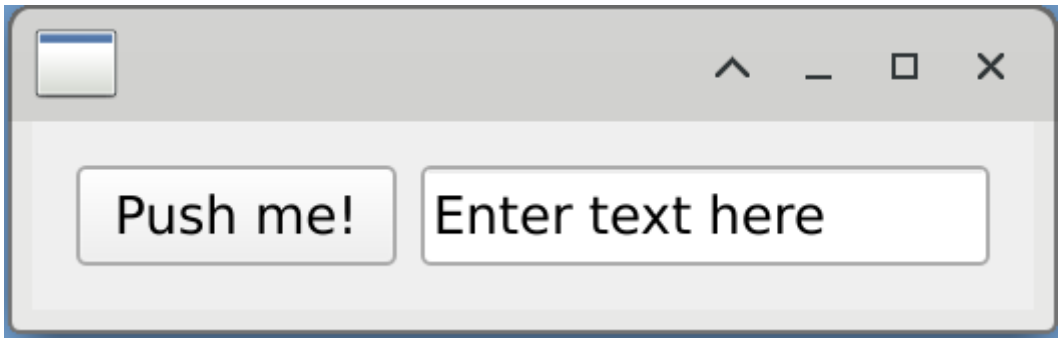


Fig. 5: Two widgets in one application

widget added to it in an horizontal row. Most of the times the widgets in the layout will be equally sized (at least horizontally).

Then, this layout is set to panel, and instead of using `show()` in each widget, we do it in the topmost widget.

Layouts are used to ensure elasticity of the application, so that it reforms itself when the size of the window is changed, and to avoid leaving empty spaces. Specifying location of widget using coordinates is a really bad practice, which can hide elements of the UI when the window is smaller than its design, and leaves empty spaces when it is bigger.

Qt Signals and Slots with two widgets

Continuing with the example above, we have connected the two widgets from the example above. From the QLineEdit[32], we use Signal `textChanged(str)`, and we connect it to the slot `setText(str)` from QPushButton[33].

```
#!/usr/bin/env python3
import sys
from taurus.external.qt.QtWidgets import QApplication
from taurus.external.qt.QtWidgets import QWidget
from taurus.external.qt.QtWidgets import QPushButton
from taurus.external.qt.QtWidgets import QLineEdit
from taurus.external.qt.QtWidgets import QHBoxLayout

if __name__ == "__main__":
    app = QApplication(sys.argv)
    panel = QWidget()
    layout = QHBoxLayout()
    widget1 = QLineEdit(panel)
```

(continues on next page)

[32] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QLineEdit.html
[33] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QPushButton.html

(continued from previous page)

```
    widget1.setText("Enter text here, and see")
    widget2 = QPushButton(panel)
    widget2.setText("Wait for it...")
    widget1.textChanged.connect(widget2.setText)
    layout.addWidget(widget1)
    layout.addWidget(widget2)
    panel.setLayout(layout)
    panel.show()
    sys.exit(app.exec_())
```

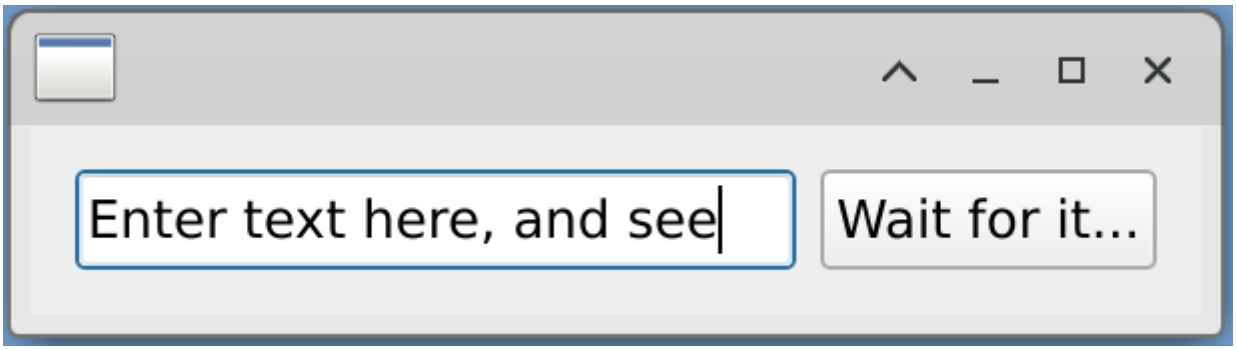


Fig. 6: Widget to widget Signal Slot connection

Signal textChanged is fired every time the text changes, and has one string argument. On the other hand, setText slot also has one string argument.

Important: Signal and Slot signature must match.

Taurus Introduction

Taurus is a UI framework oriented to control systems. It is designed using the Model View Controller pattern in mind, and offers a model with a plugin-based design capable of accessing multiple middlewares.

Taurus is developed under Python, and uses Qt as its UI toolkit library. CUT uses Taurus as it offers an extendible toolkit to develop UIs, which is easy to use.

Model View Controller Pattern

Taurus is an MVC[34] pattern based UI Framework. The MVC[35] pattern aims for re-usability of components, and the modularization of development.

- *Re-usability*: it is achieved by the separation of the Model. The Model should be able to plug into most controller and views, by sharing a common interface.
- *Modularization*: Since all views are able to present the model, development can be modularized. A new view can be developed, without losing functionality of the other view. The Model can be expanded, but since the interface is preset, the view will just show a new widget.

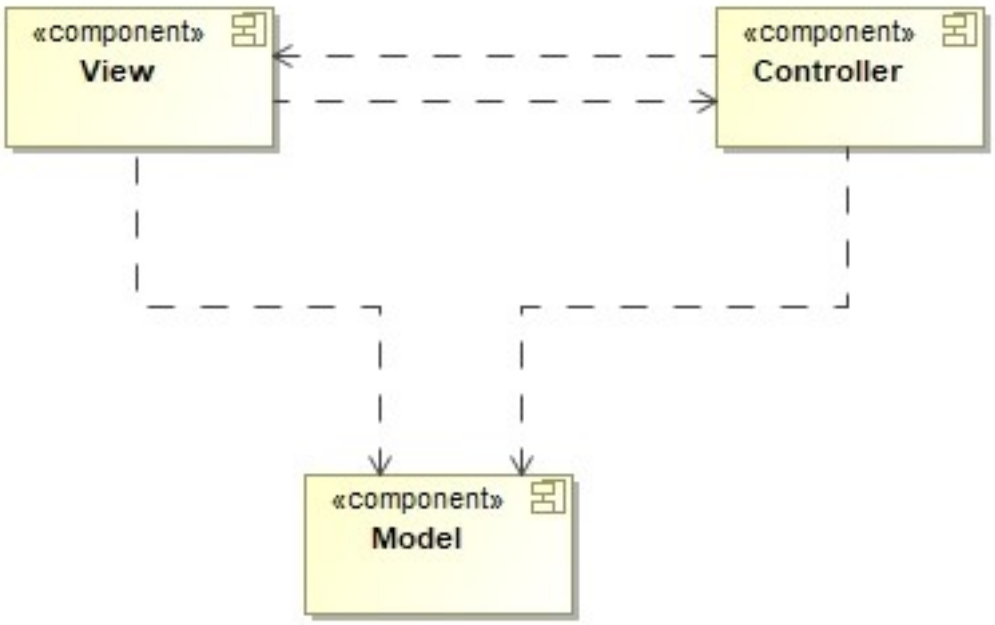


Fig. 7: MVC Component Diagram

The **Model** is a section of data from the domain of the application. Responsibilities of this class are:

- Contain the data
- Knows how to read from its source
- Knows how to write it back to its source
- Translates any metadata into usable Roles

As an example, a model can be, for a motor: the encoder value, brake state, power state, incoming voltage, speed, acceleration. Another example, from a database, the results the query `SELECT *`

[34] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[35] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

`FROM table_students;`. Each column in the result will be a Role, and each row will represent a new item, each with its Roles according to columns.

The **View** presents to the user the data from the model. Among its responsibilities:

- Takes a subset of entries in the model, and presents it.
- Determines the layout of the presentation (list, table, tree, heterogeneous, etc)
- Each piece of data can use a different Widget, these are called Delegates.

Examples of views: List, ComboBox, Table, Tree, Columns

The **Controller** takes the inputs from the user, and makes the necessary changes to the model. Responsibilities these classes have:

- Keeps references to Model and View.
- Process input from the user, converting it to domain compatible notations.
- Can also alter the user input.
- Can manipulate the model, so it changes what is presented.

URIs

Every part of a Taurus model (Attributes, Device, Authority), are identified by a URI. The URI has several parts:

```
cii.oldb:/cut/demoservice/instance1/sin
```

- `cii.oldb` **scheme**
- `:/` **authority**
- `/cut/demoservice/instance1` **device**
- `sin` **attribute**

The **scheme** normally indicates the transport and message protocol, but in taurus is used also to identify which Taurus model plugin should handle this model.

The **authority** here is almost empty. CII OLDB only allows one OLDB in the environment. But here, a different protocol could reference a particular server.

The **device** represents a physical device that is able to measure a physical phenomenon, or is able to actuate on a device. In terms of software, is a container for attributes and methods.

The **attribute** is the representation of a measurement.

Taurus Model

The Taurus Model is not a complex one, and also it is not based on QAbstractItemModel[36] Qt class.

It is located in the `taurus.core` module, as a set of 5 partially virtual classes. 3 of these classes, form a tree-like structure:

```
Authority
├---Device1
|    ├---- Attribute1
|    └---- Attribute2
└---Device2
     ├---- Device3
     |      ├---- Attribute3
     |      └---- Attribute4
     └---- Device4
            ├---- Attribute5
            └---- Attribute6
```

- TaurusModel[37], the base class for all model elements.
- TaurusDevice[38], which represents branches in a tree like structure.
- TaurusAttribute[39], which represents leaves in this tree like structure.
- TaurusAuthority[40], it represents a database of information about the control system.
- TaurusFactory[41], is in charge of preparing an instance of one of the 4 classes above. It will also use NameValidator to figure is a particular URI is well constructed or not, and which kind of class should it return.

TaurusModel[46] base class and TaurusDevice[47] class implements a Composition[48] pattern: Any device can have multiple children, but attributes cannot.

Important: The use of the Composition pattern has two purposes: Have a tree like structure of Authority, Devices and Attributes, while at the same moment, being able to access them all in the

[36] https://doc.qt.io/qt-6/qabstractitemmodel.html
[37] https://taurus-scada.org/devel/api/taurus/core/_TaurusModel.html#taurus.core.TaurusModel
[38] https://taurus-scada.org/devel/api/taurus/core/_TaurusDevice.html#taurus.core.TaurusDevice
[39] https://taurus-scada.org/devel/api/taurus/core/_TaurusAttribute.html#taurus.core.TaurusAttribute
[40] https://taurus-scada.org/devel/api/taurus/core/_TaurusAuthority.html#taurus.core.TaurusAuthority
[41] https://taurus-scada.org/devel/api/taurus/core/_TaurusFactory.html#taurus.core.TaurusFactory
[42] https://taurus-scada.org/devel/api/taurus/core/_TaurusAttribute.html#taurus.core.TaurusAttribute
[43] https://taurus-scada.org/devel/api/taurus/core/_TaurusDevice.html#taurus.core.TaurusDevice
[44] https://taurus-scada.org/devel/api/taurus/core/_TaurusAuthority.html#taurus.core.TaurusAuthority
[45] https://taurus-scada.org/devel/api/taurus/core/_TaurusModel.html#taurus.core.TaurusModel
[46] https://taurus-scada.org/devel/api/taurus/core/_TaurusModel.html#taurus.core.TaurusModel
[47] https://taurus-scada.org/devel/api/taurus/core/_TaurusDevice.html#taurus.core.TaurusDevice
[48] https://en.wikipedia.org/wiki/Composite_pattern

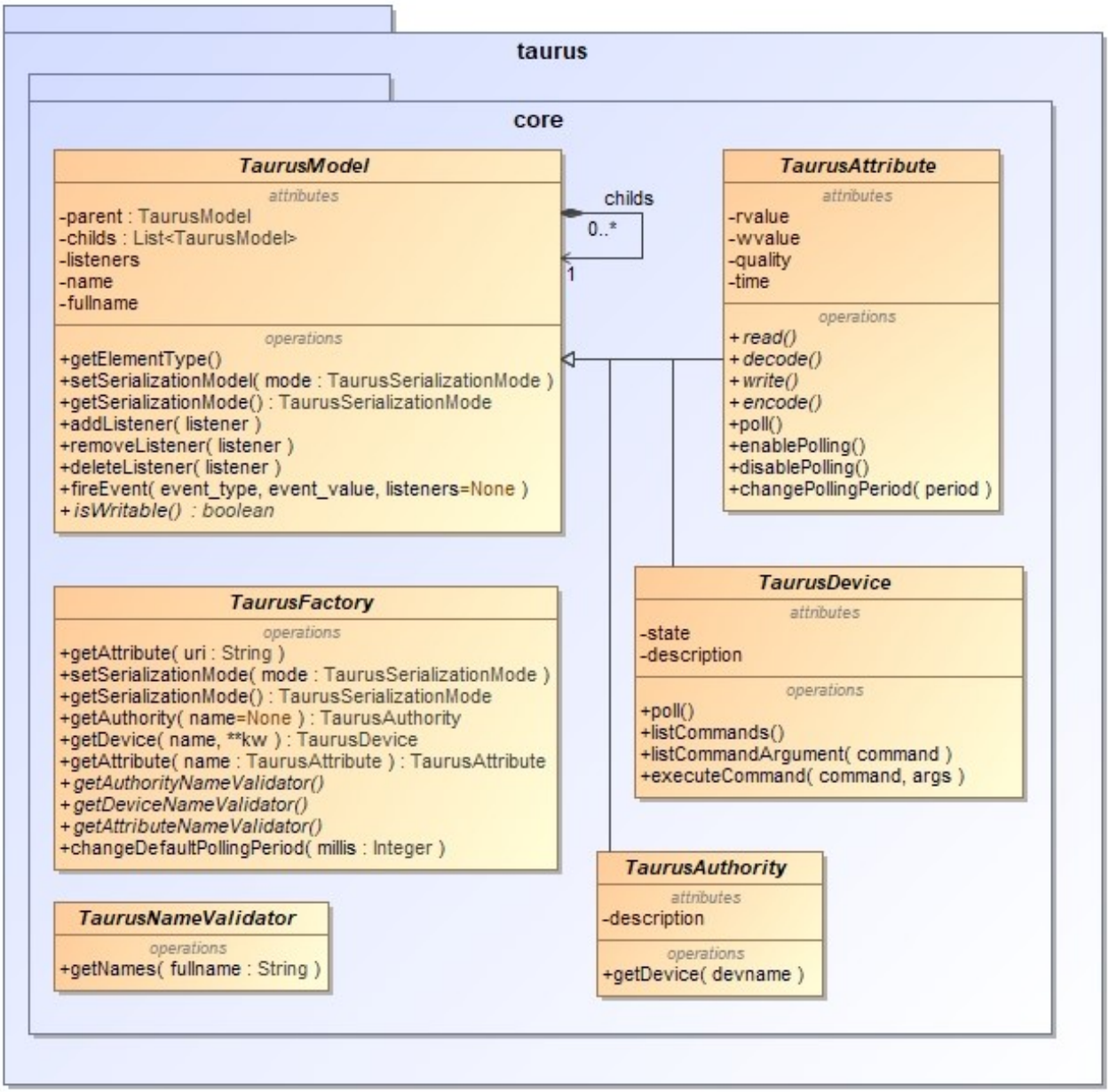


Fig. 8: Class Diagram of the `taurus.core` module. TaurusAttribute[42], TaurusDevice[Page 26, 43] and TaurusAuthority[Page 26, 44] all inherit from the same parent, the TaurusModel[Page 26, 45] class.

same way.

The TaurusFactory[49] in the model provides an Abstract Factory pattern[50], that will provide most of the logic to create the needed entities, but specifics are left to a particular scheme plugin.

By itself, `taurus.core` classes do not allow access to any control system. They are partially virtual, so they must be fully realized before use. Taurus includes models for tango, epics, h5file, tangoarchiving, pandas and eval. CUT provides an CII OLDB plugin, and in the future will support MAL Reply Request, MAL Subscriptions, CII Engineering Archive and CII Configuration.

Taurus Model Plugins

A series of data models allows Taurus access to different backends. In particular, through this document, we will be presenting example that make use of two of them:

- evaluation[51] is provided by Taurus. This model executes Python code, and translate the return value as best as possible to TaurusAttribute[52]. It is a read-only model.
- cii.oldb[53] part of the Control UI Toolkit, this module allows read/write access to CII OLDB service, as TaurusAttribute[54].
- mal.rr[55] a python module that is part of the Control UI Toolkit, allows access to commands provided by MAL Request Reply services.

Taurus Model Plugin: OLDB

To access the OLDB plugin, use Taurus Factories. The factory resolves the URI to the corresponding model plugin.

```
1 import time
2 from taurus import Attribute
3
4 sine = Attribute('cii.oldb:/cut/demoservice/instance1/double-scalar-sin')
5 sine.forceListening()
6 while(True):
7     time.sleep(1)
8     print(sine.rvalue)
```

1. Using the URI, the TaurusFactory[56] will find out the scheme for the attribute.

[49] https://taurus-scada.org/devel/api/taurus/core/_TaurusFactory.html#taurus.core.TaurusFactory
[50] https://en.wikipedia.org/wiki/Abstract_factory_pattern
[51] https://taurus-scada.org/devel/api/taurus/core/evaluation.html
[52] https://taurus-scada.org/devel/api/taurus/core/_TaurusAttribute.html#taurus.core.TaurusAttribute
[53] https://gitlab.eso.org/ecos/cut/-/tree/master/taurus/plugins/ciioldb
[54] https://taurus-scada.org/devel/api/taurus/core/_TaurusAttribute.html#taurus.core.TaurusAttribute
[55] https://gitlab.eso.org/ecos/cut/-/tree/master/taurus/plugins/mal
[56] https://taurus-scada.org/devel/api/taurus/core/_TaurusFactory.html#taurus.core.TaurusFactory

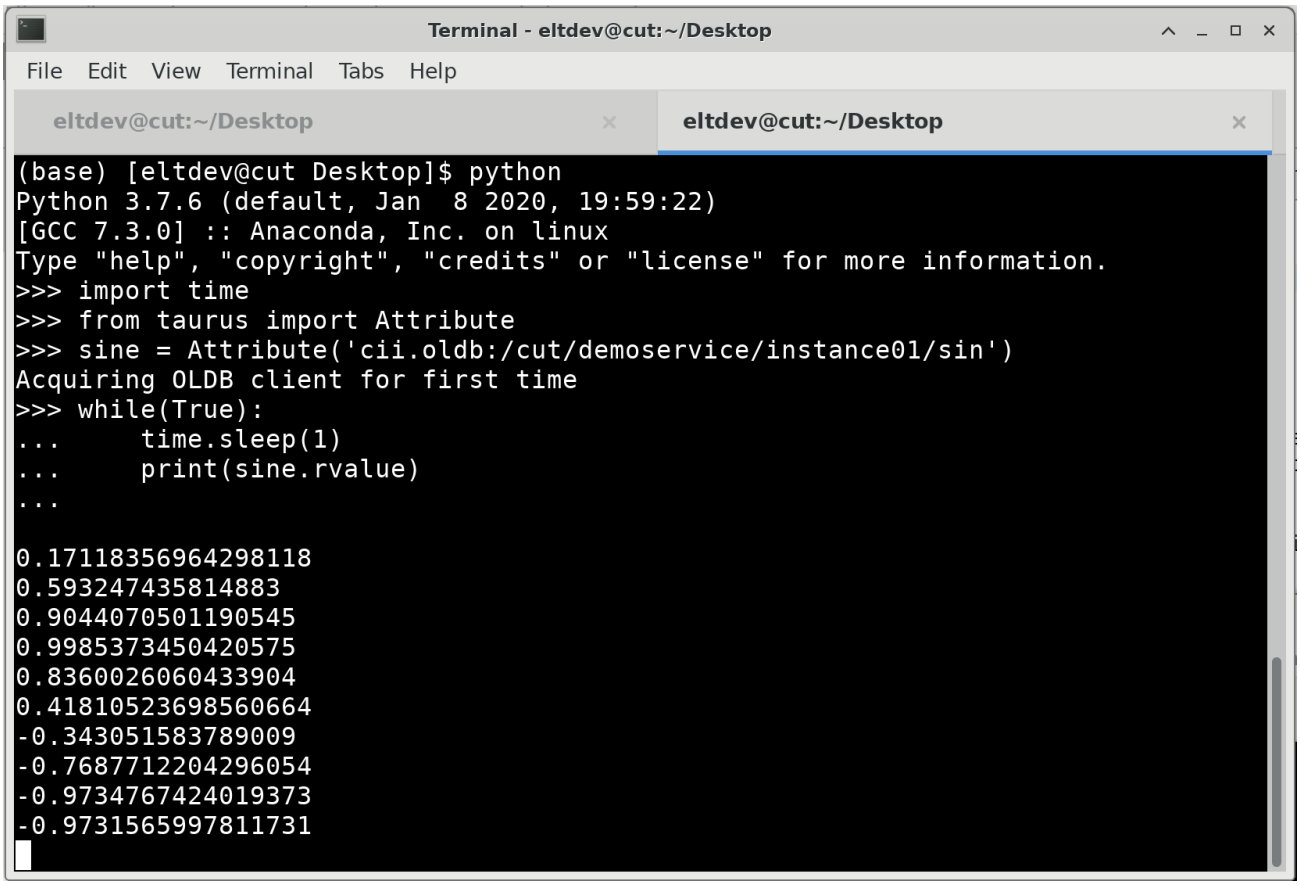


Fig. 9: Terminal output of the script above. Values may differ in your execution.

2. Using the matching plugin, it will request instances of the CiiFactory[57] and CiiValidator[58].
3. Will check the validity of the URI using the CiiValidator[59]
4. And the create the attribute using the CiiFactory[60].

Every instance of a particular URI, is a singleton. So if we were to again request for `taurus.Attribute('cii.oldb:/cut/demoservice/instance1/double-scalar-sin')`, we would get a reference to the previous object.

Tip: The developer can access `taurus.Attribute` manually. The models for widgets are expressed as strings of URIs, and is the responsibility of the widget to get an instance to the proper `taurus.Attribute` class, through the use of `taurus.Authority`.

This can be helpful while development more complex commands or methods.

You can obtain several pieces of information from the metadata stored in the OLDB. This examples shows you what you can get:

```
from taurus import Attribute

sine_value = Attribute('cii.oldb:/cut/demoservice/instance1/double-scalar-sin')
sine_value._connect()
print('sine value: %f'% (sine_value.rvalue.magnitude))
print('sine alarm ranges: ]%s, %s['% (sine_value.alarms[0], sine_value.alarms[1]))
print('sine units: %s'% (sine_value.rvalue.units))
print('sine datapoint description: %s' % (sine_value.description))
print('sine uri: %s' % (sine_value.fullname))
print('sine label: %s' % (sine_value.label))
print('Can we write into sine?: %s' % (sine_value.isWritable()))
#Tip: You can stop the cut-demo-service app for a bit.
sine_value.write(0.5)
print( sine_value.read() )
print('sine quality: %s' % (sine_value.quality.name))
```

[57] https://gitlab.eso.org/ecos/cut/-/blob/master/taurus/plugins/ciioldb/tauruscii/taurusciifactory.py
[58] https://gitlab.eso.org/ecos/cut/-/blob/master/taurus/plugins/ciioldb/tauruscii/taurusciivalidator.py
[59] https://gitlab.eso.org/ecos/cut/-/blob/master/taurus/plugins/ciioldb/tauruscii/taurusciivalidator.py
[60] https://gitlab.eso.org/ecos/cut/-/blob/master/taurus/plugins/ciioldb/tauruscii/taurusciifactory.py

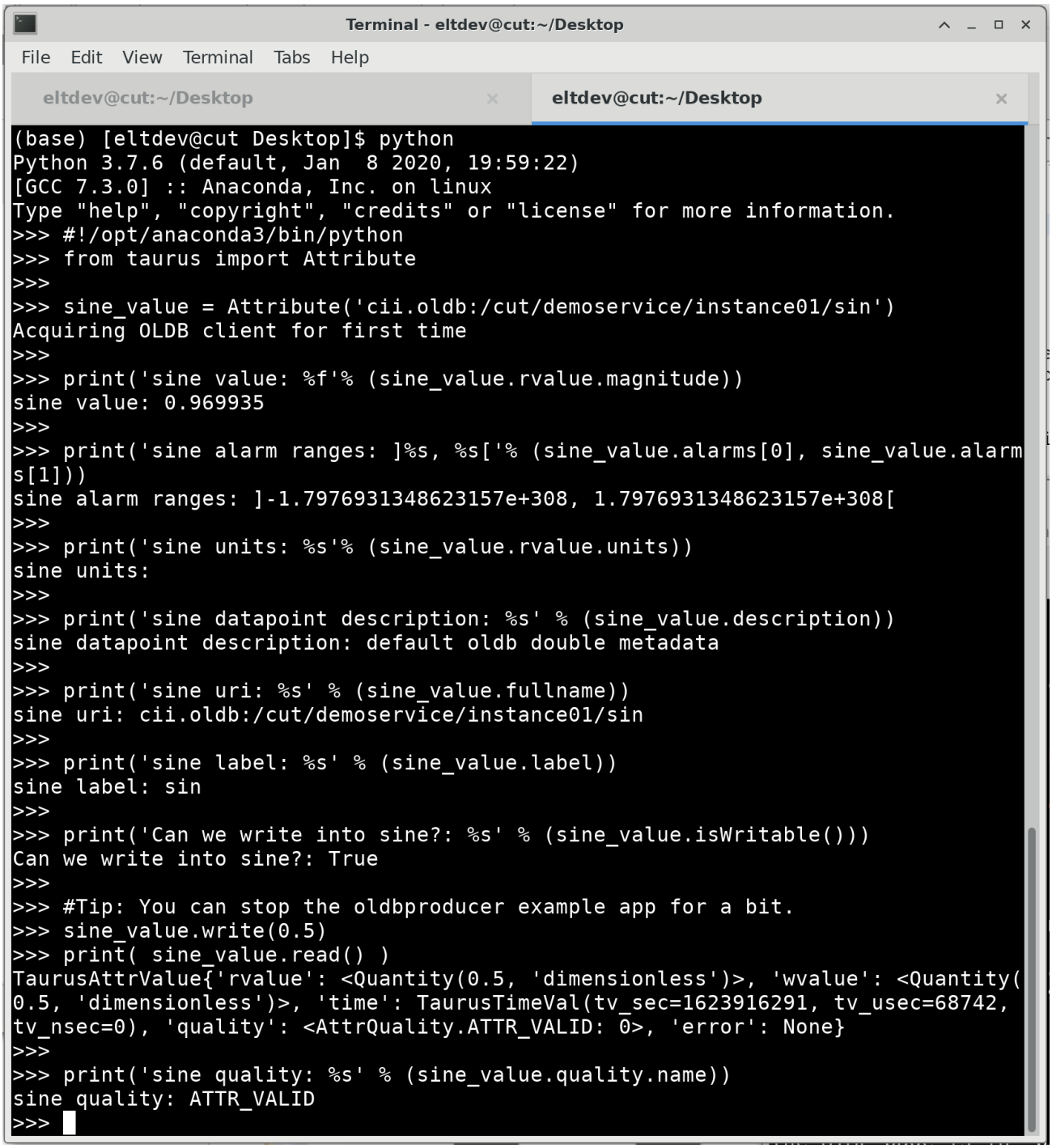


Fig. 10: Terminal output of the script above.

Taurus Widgets: TaurusLabel

Taking the examples from the first section, we now replace a few widgets by Taurus ones. These widgets automatically connect to a datapoint defined in their *model* property. Taurus does not replace Qt, but provides a library called `taurus.external.qt` that abstract the developer from the particular set of python bindings used (PySide2, PySide6, PyQt5, PyQt6). Please use the `taurus.external.qt` module as it will make your code version independent.

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication

app = TaurusApplication(sys.argv, cmd_line_parser=None,)
panel = Qt.QWidget()
layout = Qt.QHBoxLayout()
panel.setLayout(layout)

from taurus.qt.qtgui.display import TaurusLabel
w1, w2 = TaurusLabel(panel), TaurusLabel(panel)
layout.addWidget(w1)
layout.addWidget(w2)
w1.model = 'cii.oldb:/cut/demoservice/instance1/double-scalar-add'
w2.model = 'cii.oldb:/cut/demoservice/instance1/string-scalar-timestamp'

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

A couple of details of this example:

- `TaurusLabel(panel)` passes to the *__init__* the panel object. Every widget should have a parent, this is used by Qt to correctly destroy the objects when windows are closed or the application is shutdown.
- The notation `w1, w2 = TaurusLabel(panel), TaurusLabel(panel)` is permitted in Python, where return values are assigned in order.
- *TaurusApplication* class is a replacement for QApplication. It initializes Taurus logging capabilities, parses command line options, among other tasks.

Fig. 11: On this widget, the example has just started. Once resized, it looks like the figure on the right. The first widget presents a number that increases over time.

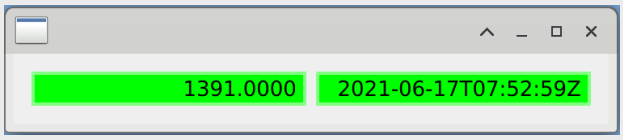


Fig. 12: The widget presents a string that contains an ISO formatted timestamp. The values are the ones stored in the OLDB, and its background color represents the quality of that particular datapoint.

Taurus Widgets: Fragments

In the next example, we explore a bit the fragments of an attribute.

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.display import TaurusLabel

app = TaurusApplication(sys.argv, cmd_line_parser=None,)
panel = Qt.QWidget()
layout = Qt.QVBoxLayout()
panel.setLayout(layout)

w1, w2, w3, w4, w5 = TaurusLabel(), TaurusLabel(), TaurusLabel(), TaurusLabel(),␣
↪TaurusLabel()
layout.addWidget(w1)
layout.addWidget(w2)
layout.addWidget(w3)
layout.addWidget(w4)
layout.addWidget(w5)
w1.model, w1.bgRole = 'cii.oldb:///cut/demoservice/instance1/int-scalar-add', ''
w2.model, w2.bgRole = 'cii.oldb:///cut/demoservice/instance1/int-scalar-add#rvalue',
↪ ''
w3.model, w3.bgRole = 'cii.oldb:///cut/demoservice/instance1/int-scalar-add#rvalue.
↪magnitude', ''
w4.model, w4.bgRole = 'cii.oldb:///cut/demoservice/instance1/int-scalar-add#rvalue.
↪units', ''
w5.model, w5.bgRole = 'cii.oldb:///cut/demoservice/instance1/int-scalar-add#label',
↪''

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

The example depicts the TaurusLabel with a bright background. The background is used automatically by the TaurusLabel to represent the quality of the Attribute. In this example, we are modifying the *bgRole* property of a widget to remove any role to be shown. This makes the widget to look more like a QLabel[61], which sometimes is desirable.

Aside from unsetting the *bgRole`s properties, the *w1* TaurusLabel uses a fragment. ``#label`* in the URI indicates that instead of the value, we are interested in displaying the label, or short name of the Attribute.

Also, the example uses a Vertical Layout, instead of an Horizontal one.

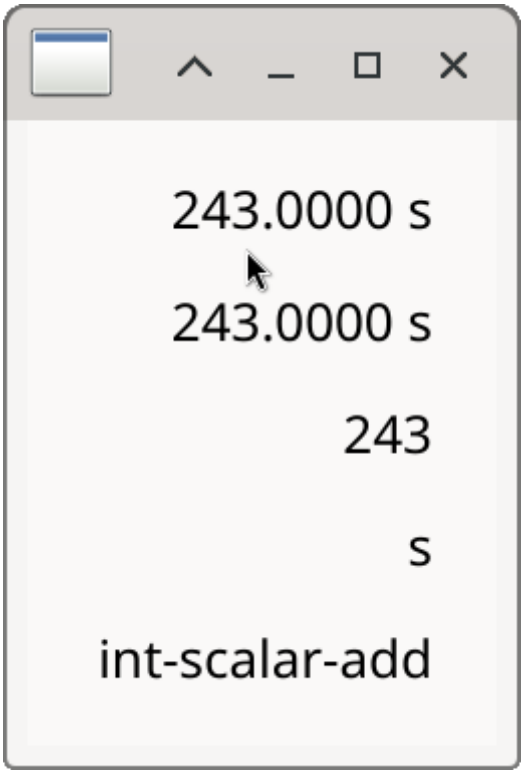


Fig. 13: TaurusLabel example

Taurus Widgets: TaurusForm

Taurus offers a way to abstract yourself from all the programming of these widgets, by using the Taurus form. This utility is a CLI command (`taurus form`), and also a class (TaurusForm[62]) allows to quickly bring up a UI with the specified _widgets_ in it.

```
$ taurus form --help
Usage: taurus form [OPTIONS] [MODELS]...
```

(continues on next page)

[61] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QLabel.html
[62] https://taurus-scada.org/devel/api/taurus/qt/qtgui/panel/_TaurusForm.html

(continued from previous page)

```
 Shows a Taurus form populated with the given model names

Options:
 --window-name TEXT  Name of the window
 --config FILENAME   configuration file for initialization
 --help              Show this message and exit.
```

The command line version of TaurusForm[63] takes a string list[64] as model, and creates a QGridLayout[65] of 5 columns by n rows, n being the number of items in the string list[66].

Label	Read Widget	Write Widget	Units	Extra

For example:

```
$ taurus form 'eval:Q("12.5m")' 'cii.oldb:/cut/demoservice/instance1/string-scalar-
↪timestamp'
```

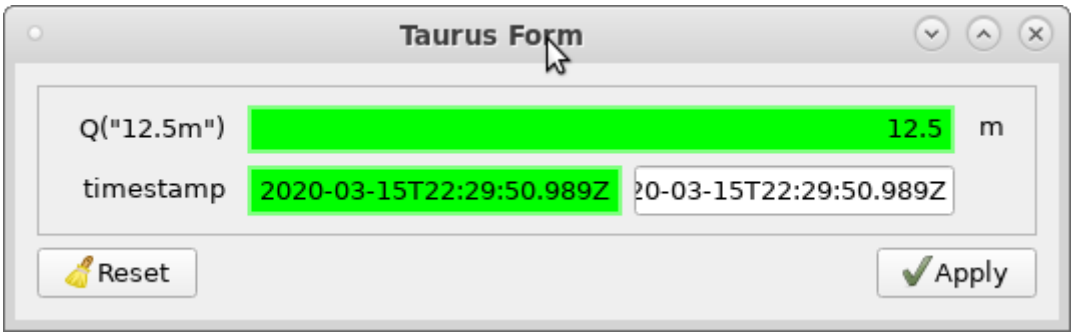


Fig. 14: Taurus Form allows to quickly access values, using a list of strings.

In this case, the first row has the `eval:Q("12.5m")` Attribute, which has a Label, Read Widget and Units entries. The second row has the `cii.oldb:/cut/demoservice/instance1/string-scalar-timestamp` which has a Label, Read Widget and Write Widget. Note that none of them has the Extra widget, as is feature intended for manual configuration.

The *Reset* button will restore the value in the Write widgets to the last know value read from the backend. The *Apply* button will send the values in the Write widget to the backend for store. When the value from the Write widget is different from the one in the backend, then the widget turns blue, and the label is highlighted in blue as well.

[63] https://taurus-scada.org/devel/api/taurus/qt/qtgui/panel/_TaurusForm.html
[64] http://www.openbookproject.net/books/bpp4awd/ch03.html
[65] https://doc.qt.io/qt-6/qgridlayout.html
[66] http://www.openbookproject.net/books/bpp4awd/ch03.html

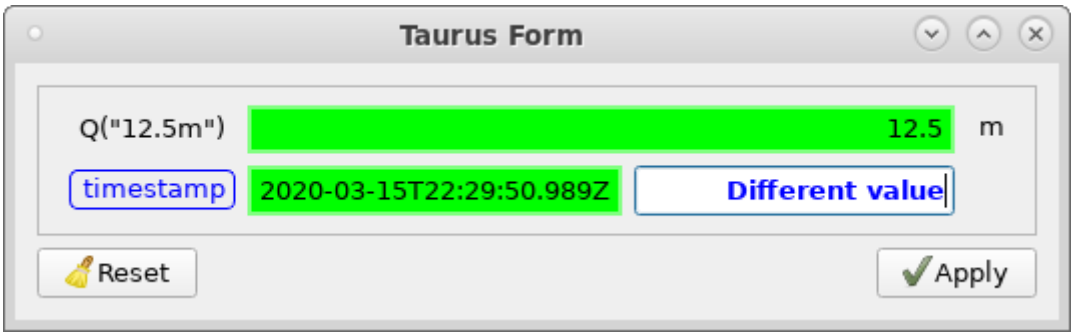


Fig. 15: Taurus Form highlighting in blue items that have been updated.

Another interesting feature of the TaurusForm[67], is that is allows to change the *Read* and *Write* widgets on runtime. Right clicking on the label allows to access this, and other kind of functions.

If you have two *Taurus Form* windows opened, or two of them in the same application, you can drag and drop model items from one to another.

Here is how to use TaurusForm[68] in code:

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.panel import TaurusForm

app = TaurusApplication(sys.argv, cmd_line_parser=None,)
panel = TaurusForm()
props = ['double-scalar-sin', 'double-scalar-cos', 'string-scalar-timestamp',
↪'double-scalar-add',
         'int-scalar-add', 'double-vector-rand']
model = ['cii.oldb:/cut/demoservice/instance1/%s' % p for p in props]
# This is the same as:
# model = [ 'cii.oldb:/cut/demoservice/instance1/double-scalar-sin',
#           'cii.oldb:/cut/demoservice/instance1/double-scalar-cos',
#           'cii.oldb:/cut/demoservice/instance1/string-scalar-timestamp',
#           'cii.oldb:/cut/demoservice/instance1/double-scalar-add',
#           'cii.oldb:/cut/demoservice/instance1/int-scalar-add',
#           'cii.oldb:/cut/demoservice/instance1/double-vector-rand']
panel.setModel(model)

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

[67] https://taurus-scada.org/devel/api/taurus/qt/qtgui/panel/_TaurusForm.html
[68] https://taurus-scada.org/devel/api/taurus/qt/qtgui/panel/_TaurusForm.html

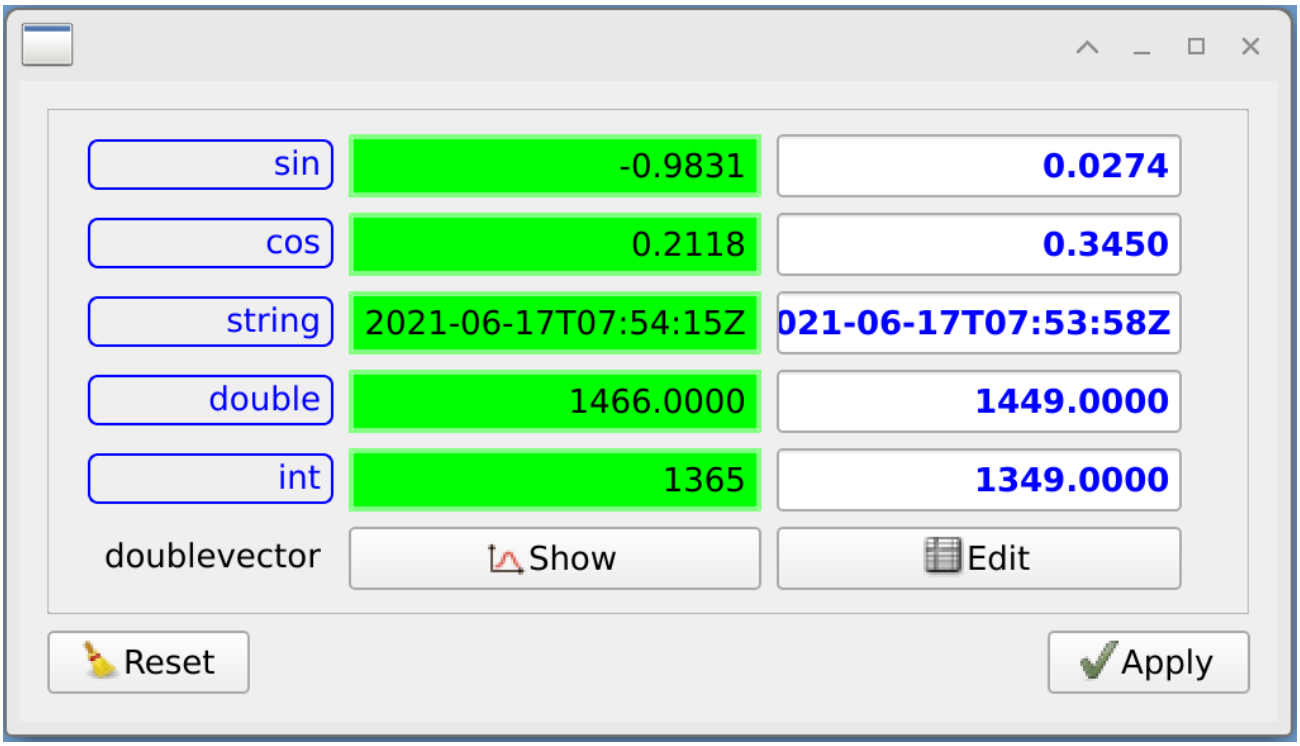


Fig. 16: TaurusForm example

Tip: Nowhere in the command line or the code, the developer indicates which kind of widget we want to use for each item in the model. This is automatically determined by the Taurus Form, according to the datatype, and this can be altered by the use of CustomMappings.

Tip: At this point, the developer may notice a pattern. The URI is used to auto-determine the kind of Model element we need. The datatype and roles are used to automatically determine the widgets we need.

If using code, the developer may want to force the use of specific widgets. This can be achieved like this:

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.panel import TaurusForm
from taurus.qt.qtgui.display import TaurusLabel
from taurus.qt.qtgui.plot import TaurusTrend
from taurus.qt.qtgui.plot import TaurusPlot
```

(continues on next page)

(continued from previous page)

```
8
9  app = TaurusApplication(sys.argv, cmd_line_parser=None,)
10 panel = TaurusForm()
11 props = ['double-scalar-sin', 'double-scalar-cos', 'double-scalar-add', 'int-scalar-
   ↪add',
12          'double-vector-rand']
13 model = ['cii.oldb:/cut/demoservice/instance1/%s' % p for p in props]
14 panel.setModel(model)
15 panel[1].readWidgetClass = 'TaurusTrend'
16 panel[4].readWidgetClass = 'TaurusPlot'
17
18 panel.show()
19 sys.exit(app.exec_())
```

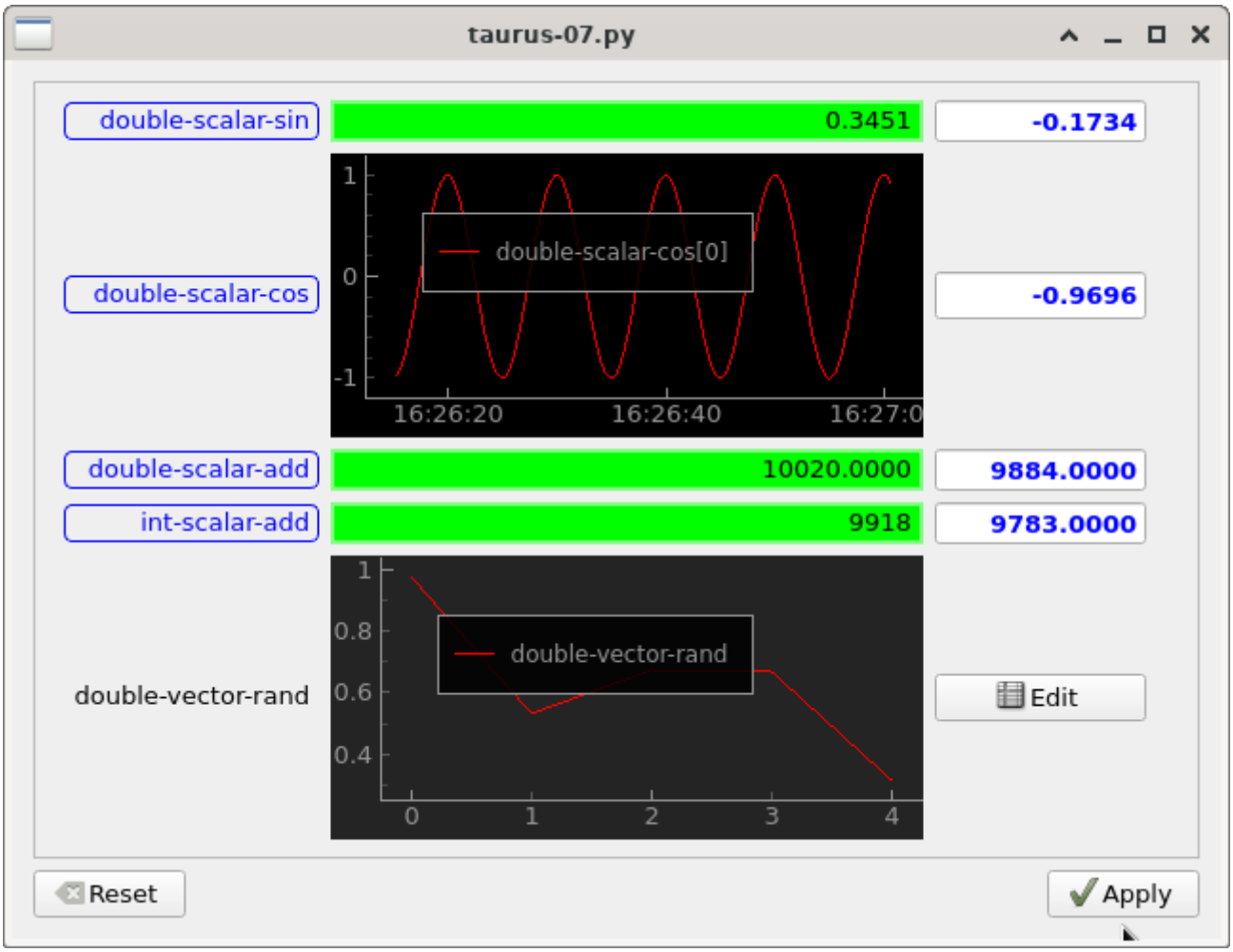


Fig. 17: TaurusForm example, forcing specific widgets

This examples is very similar to the previous one. The main difference is that we access the third and fifth elements of the *panel* object, and change its readWidgetClass. Notice that in one we used a string with the class name, and in another, the class reference.

Taurus Widgets: TaurusPlot and TaurusTrend

This examples shows how to use the TaurusTrend widget, using the model to set three URI. A developer may set a single entry, or multiple ones, as needed.

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.plot import TaurusTrend

app = TaurusApplication(sys.argv, cmd_line_parser=None,)
panel = TaurusTrend()
model = ['cii.oldb:/cut/demoservice/instance1/double-scalar-sin',
         'cii.oldb:/cut/demoservice/instance1/double-scalar-cos',
         'cii.oldb:/cut/demoservice/instance1/double-scalar-tan']
panel.setModel(model)
panel.setYRange(-1.0, 1.0)

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

The example below shows three vectors generated by the **cut-demo-service**. Each vector is a 100 points representation of `sin`, `cos`, `tan` functions.

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.plot import TaurusPlot

app = TaurusApplication(sys.argv, cmd_line_parser=None,)
panel = TaurusPlot()
model = ['cii.oldb:/cut/demoservice/instance1/double-vector-sin',
         'cii.oldb:/cut/demoservice/instance1/double-vector-cos',
         'cii.oldb:/cut/demoservice/instance1/double-vector-tan']
panel.setModel(model)
panel.setYRange(-1.0, 1.0)

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

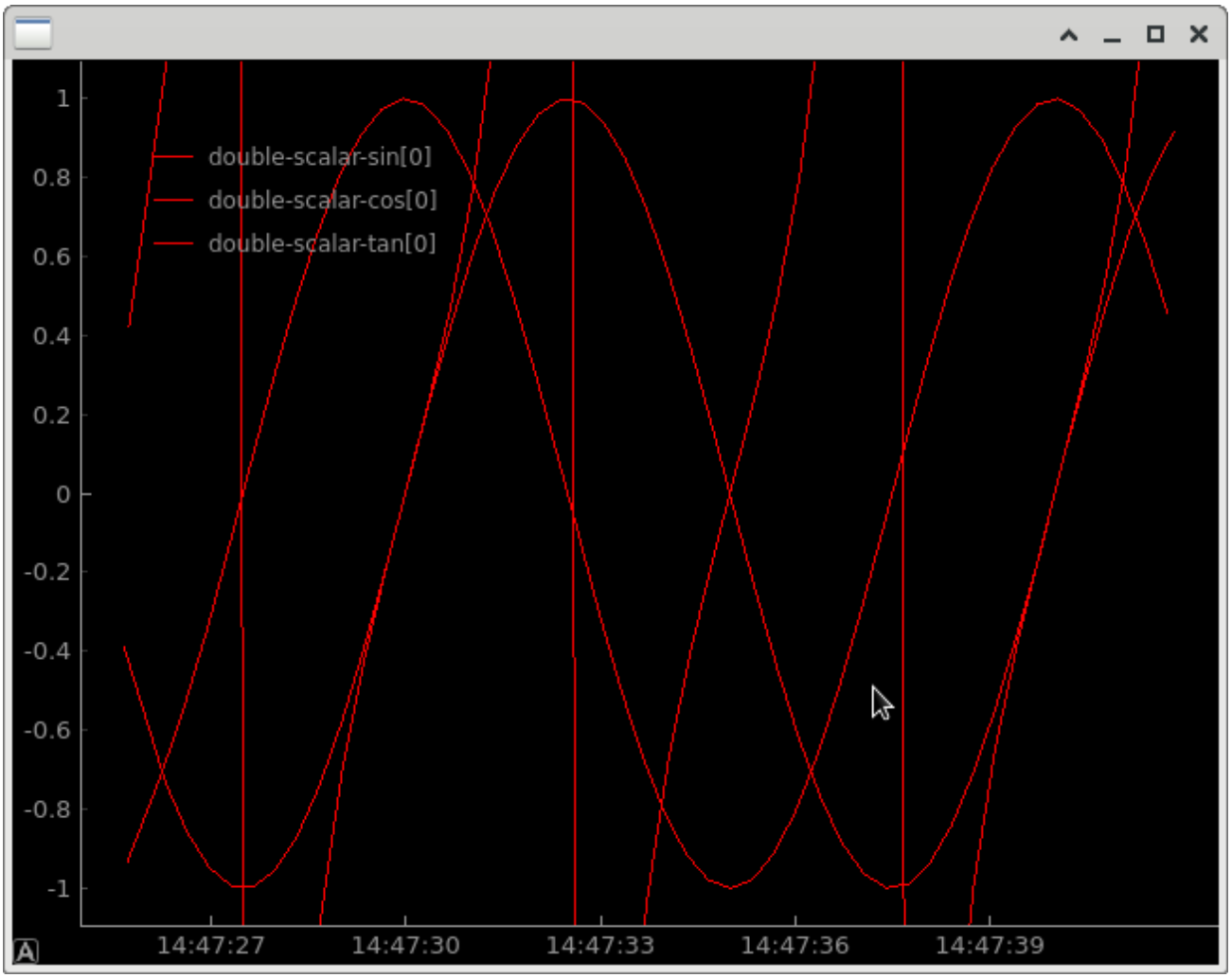


Fig. 18: TaurusTrend example

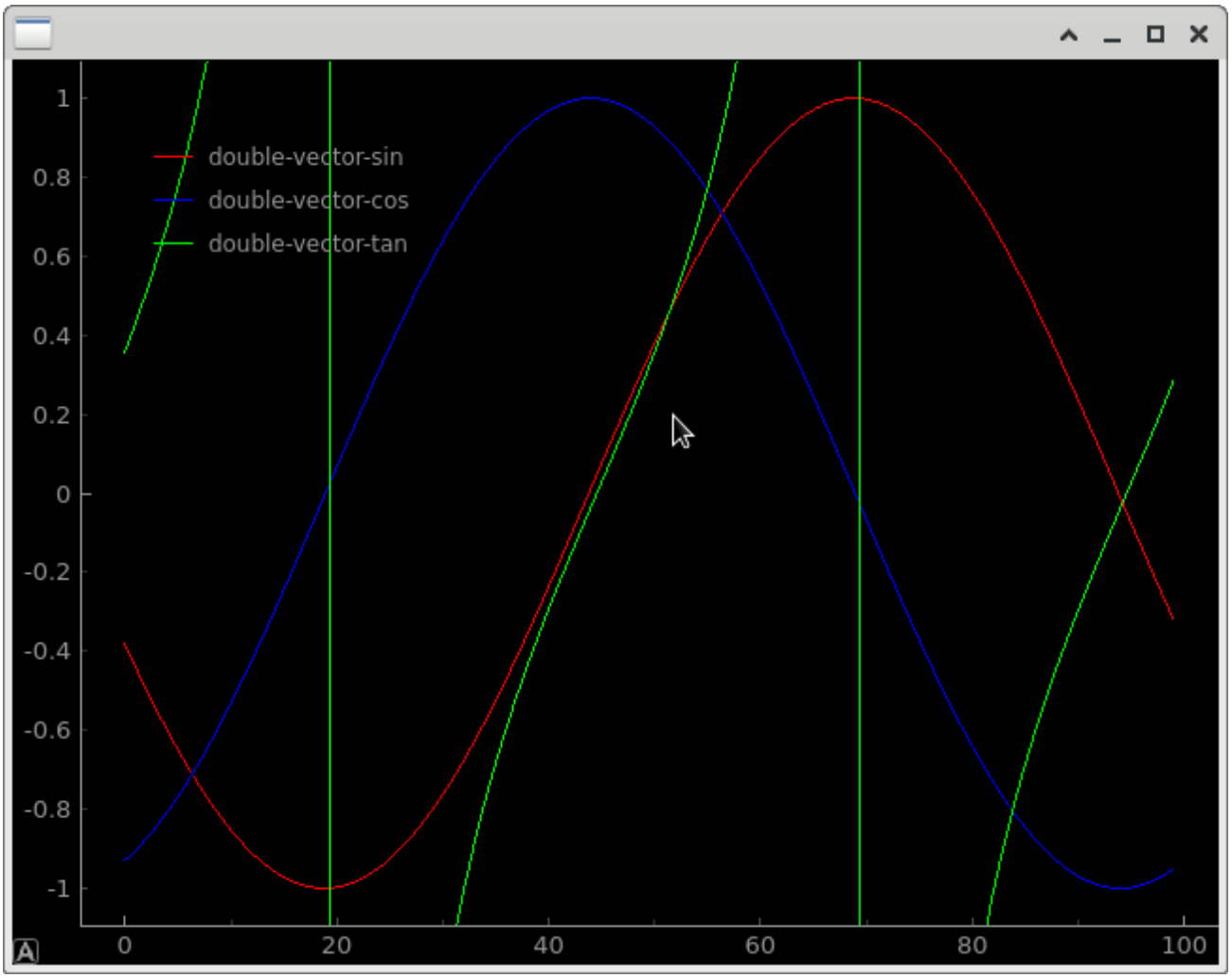


Fig. 19: TaurusPlot example

Taurus Widgets: TaurusLauncherButton

In this example, a new Widget is introduced: the TaurusLauncherButton. Even though it can store a model property, it is not used by this widget. Instead, when pressed, it will execute *show()* on the widget it has a reference to, and set the model to that widget.

Tip: It has some other cosmetic customizations. It is important to note the use of `Qt.QIcon.fromTheme` method. This is a Class method that indicates Qt to search for an icon with that name.

These names are standard (FreeDesktop Icon Naming), and its locations are also standard. Using these icon naming convention and non-direct access to icons is very important for ergonomics requirements. When the theme is changed, the icons are automatically changed as well.

```
1  import sys
2  from taurus.external.qt import Qt
3  from taurus.qt.qtgui.application import TaurusApplication
4  from taurus.qt.qtgui.button import TaurusLauncherButton
5  from taurus.qt.qtgui.display import TaurusLabel
6
7  app = TaurusApplication(sys.argv, cmd_line_parser=None,)
8  button = TaurusLauncherButton(
9      text='View timestamp',
10     widget=TaurusLabel(),
11     icon=Qt.QIcon.fromTheme('window-new')
12     )
13 button.setModel('cii.oldb:/cut/demoservice/instance1/string-scalar-timestamp')
14 button.show()
15 sys.exit(app.exec_())
```

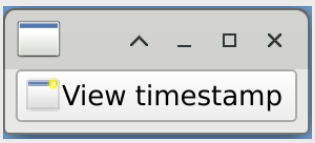


Fig. 20: On the left the example has just started

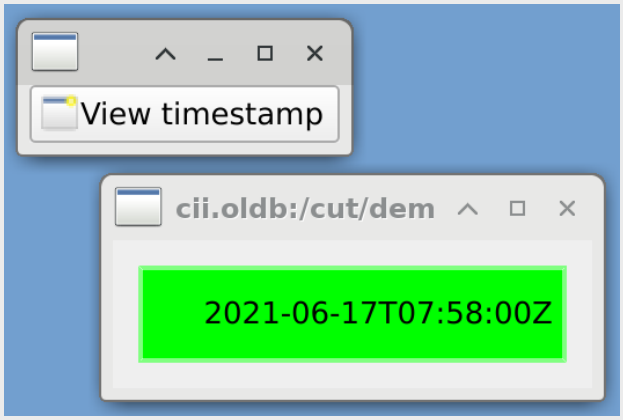


Fig. 21: On the right, the user has clicked on the button.

You can also use it to open TaurusForms. Please notice the notation on the model.

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.button import TaurusLauncherButton
from taurus.qt.qtgui.panel import TaurusForm

app = TaurusApplication(sys.argv, cmd_line_parser=None,)
button = TaurusLauncherButton(
    text='Open Demoservice',
    widget=TaurusForm(),
    icon=Qt.QIcon.fromTheme('window-new')
    )
button.setModel('cii.oldb:/cut/demoservice/instance1/double-scalar-sin,cii.oldb:/
↪cut/demoservice/instance1/double-scalar-cos,cii.oldb:/cut/demoservice/instance1/
↪int-scalar-add,cii.oldb:/cut/demoservice/instance1/double-scalar-add,cii.oldb:/
↪cut/demoservice/instance1/string-scalar-timestamp,cii.oldb:/cut/demoservice/
↪instance1/double-vector-sin')
button.show()
sys.exit(app.exec_())
```

Taurus Model Plugin: MAL Request Reply

This plugin allows to connect GUI elements to MAL services provided by server applications.

The model in this case is the URI of the MAL Server or a particular Service hosted by the server:

URI	Type	Requires Server Introspection?	Requires Interface on URI?
zpb.rr://127.0.0.1:12801	Server	Yes	No
zpb.rr://127.0.0.1:12801/StdCmds?if=::std	Service	No	Yes

The MAL Request Reply plugin uses the MalAdapter class and the Introspection interface (if available) or the ICD XML on the client host to determine which Requests (methods) are available in the server.

Similar to taurus.Attribute, taurus.Device objects can be created which allow instant access to requests on the services.

```
from taurus import Device
dev = Device("zpb.rr://127.0.0.1:12801")
dev.command_inout("Init")
dev.Init()
```

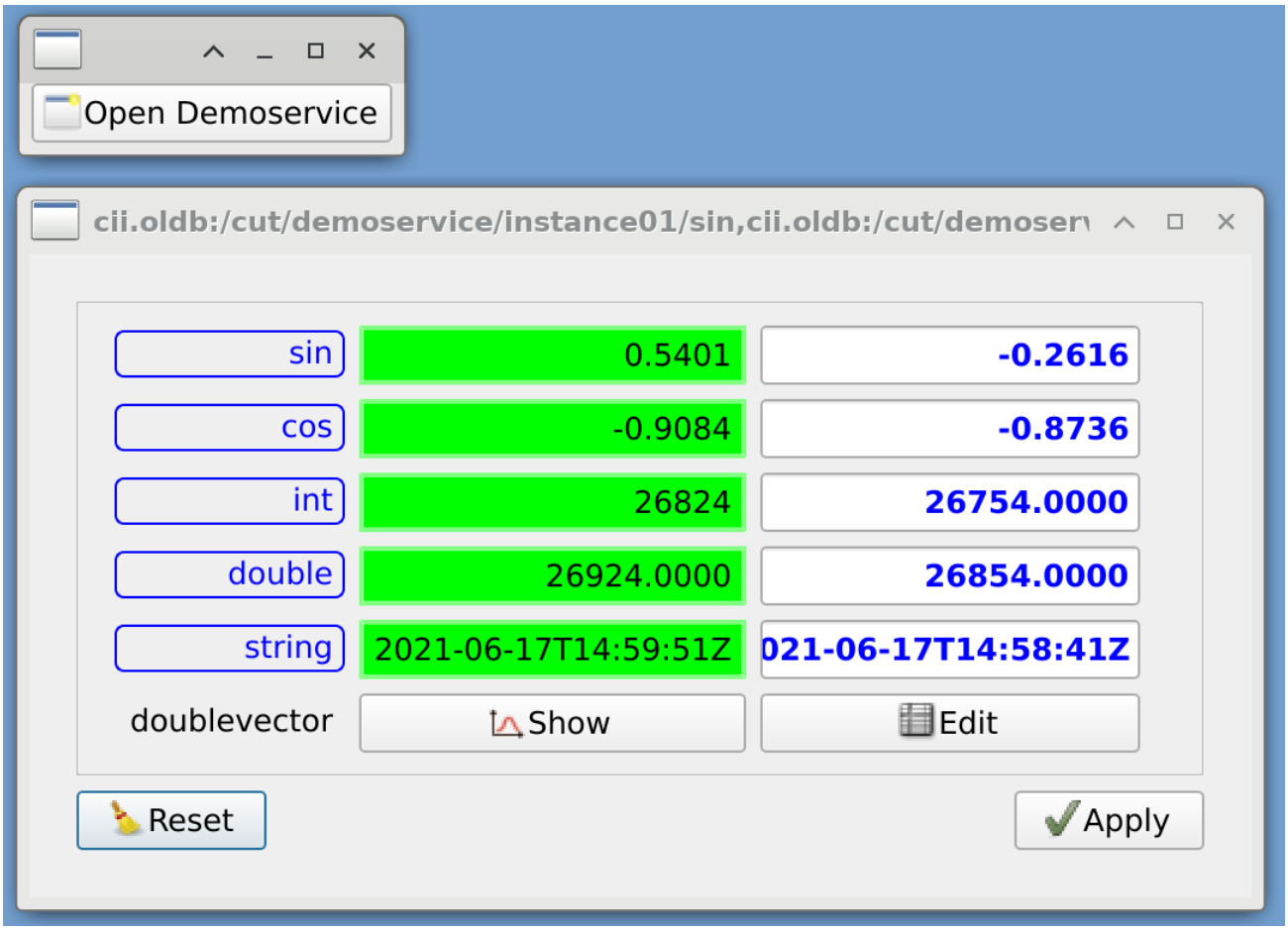


Fig. 22: TaurusLauncherButton opening a TaurusForm

The Device object has access to the requests (commands or requests) on the server. It republishes in a dynamic manner the methods in the interfaces at the Device level. It also offers a narrow interface *command_inout* that accepts the name of the request as first argument.

If a request requires an argument you can give it as argument of the methods in the Device level, or as extra (second, third, etc) arguments in the *command_inout* method.

Taurus Device

One of the three ready-to-use solutions to invoke commands on a remote MAL RR Server.

The reader may execute the CLI command *taurus device* to get access to an automatically created GUI that makes use of the *MalAdapter* capabilities to introspect the interfaces and commands available.

```
$ taurus device zpb.rr://127.0.0.1:12801
$ taurus device zpb.rr://127.0.0.1:12801/StdCmds?if=::stdif::StdCmds
```

The *Connection* state is Ready or Undefined. If there is no connection to the MAL RR Server, then a disconnection icon will be shown, and the GUI will be empty.

The *Description* is a summary of the available information from the Server. Even though Instropection interface allows to know the methods available server-side, this is a generic GUI. There is only so much that can be extracted without calling specialized methods:

- A list of the services available, via the FQN of their interfaces.
- The URI of the server or service.
- GetVersion result, if the method is available.

The *Requests* tab allows to execute remotely the methods listed in the Services ICDs. Arguments can be given. At the moment this generic GUI can only be used to pass arguments that are of common types, therefore arrays and structs are excluded at the moment.

The *Getters* tab allows access to the state of connection and the status (description) of the MAL RR Server.

Taurus Command Button

This widget executes remote commands in a MAL RR Server or Service. In requires two configuration entries which can be set through code or the *taurus designer*: model and command.

An optional property that the reader may set is the *DangerMessage*. If set this text will be presented as a confirmation message before executing the command.

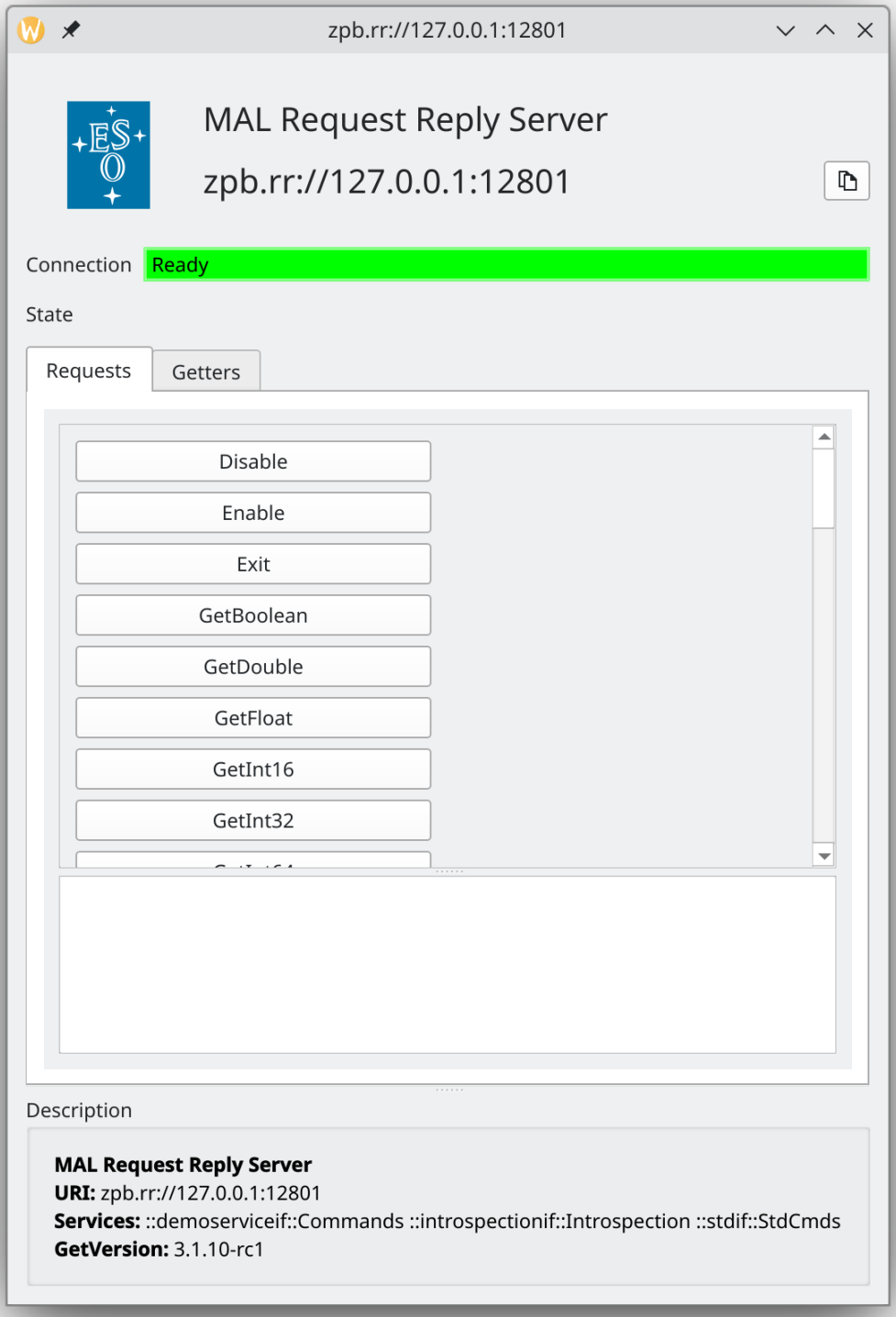


Fig. 23: Taurus Device for a MAL RR server

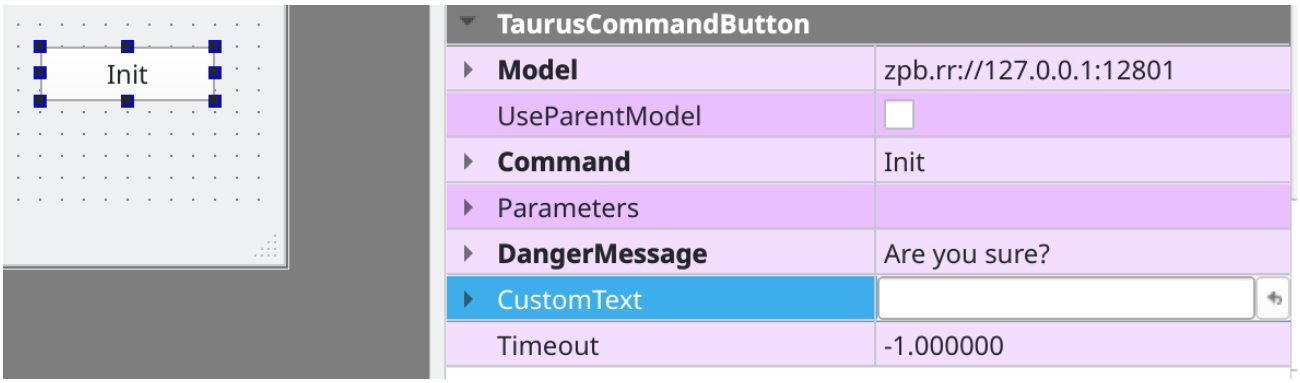


Fig. 24: Using WYSIWYG in Taurus Designer to add a Taurus Command Button for the Init command

Taurus Command Combo Box

It summarizes the available methods in a MAL RR Server or Service. It requires only one configuration: the model URI. This can be set through code or the *taurus designer*.

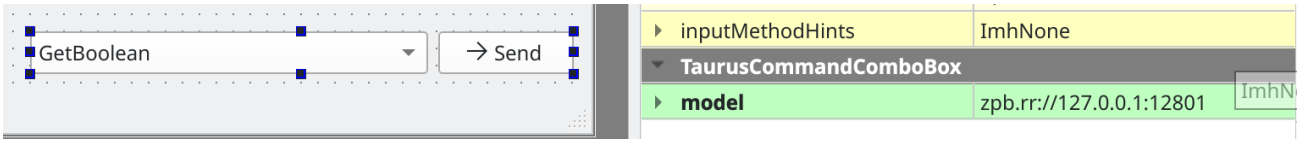


Fig. 25: Using WYSIWYG in Taurus Designer to add a Taurus Command Combobox for the cutDemoService application

Taurus Model Plugin: Evaluation

The evaluation plugin can instruct Taurus to execute basic python code instructions:

```
from taurus import Attribute

rand1 = Attribute('eval:rand()')
rand1.rvalue
angles = Attribute('eval:Q(12,"rad")')
angles.rvalue
sum = Attribute('eval:{eval:Q(12,"rad")}+{cii.oldb:/cut/demoservice/instance1/
↪double-scalar-sin}')
sum.rvalue
strsplt = Attribute('eval:{eval:"2021-10-12T11:45:00.012"}.split("T")')
strsplt.rvalue
```

A few examples using command line tool `taurus`.

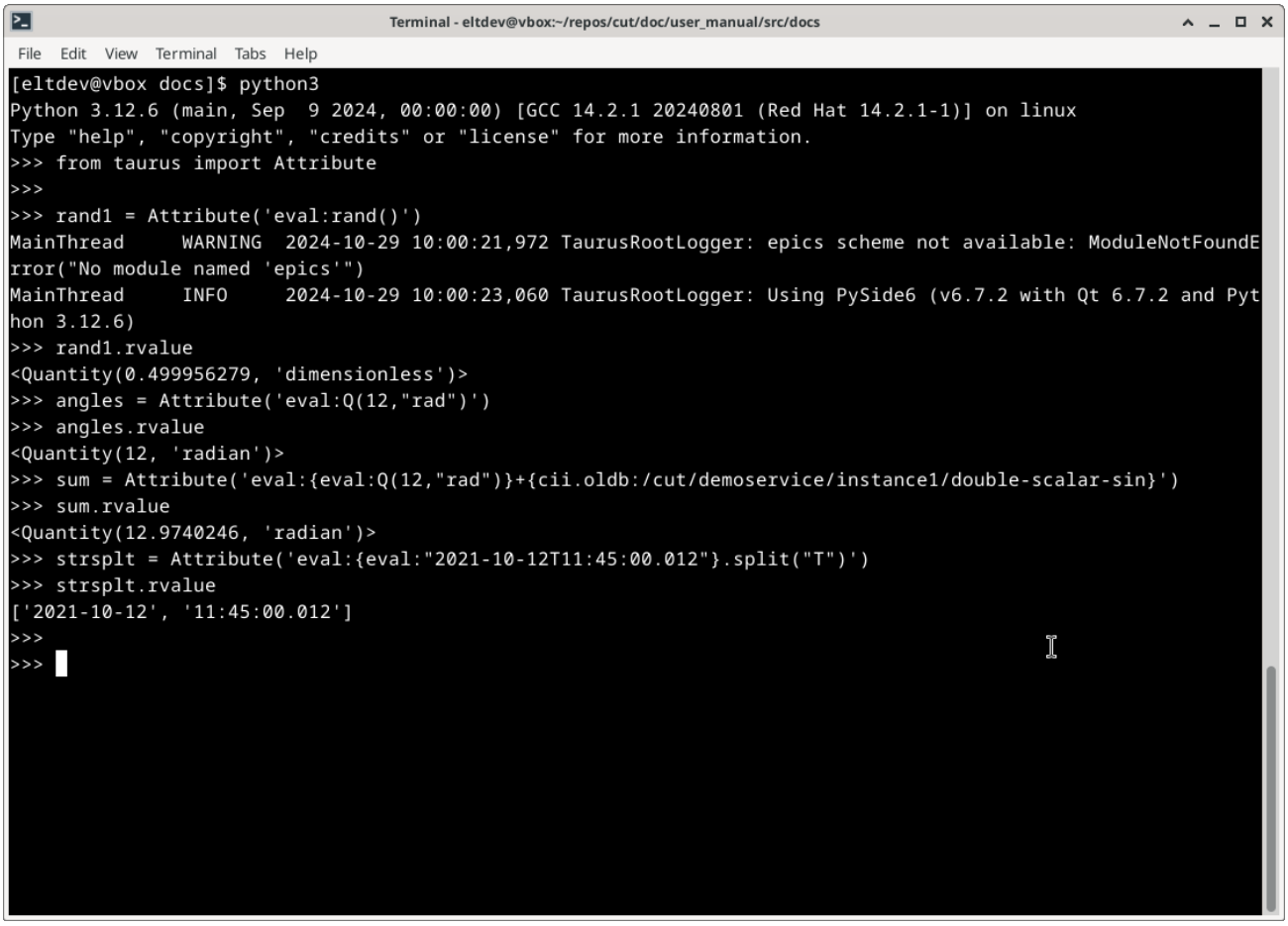


Fig. 26: Evaluation Plugin examples, through code

```
#!/bin/bash

taurus form 'eval:rand(12)' \
            'eval:2*{'cii.oldb:/cut/demoservice/instance1/int-scalar-add'}' \
            'eval:rand(16,16)' \
            'eval:@os.*/path.exists("/etc/motd")'
```

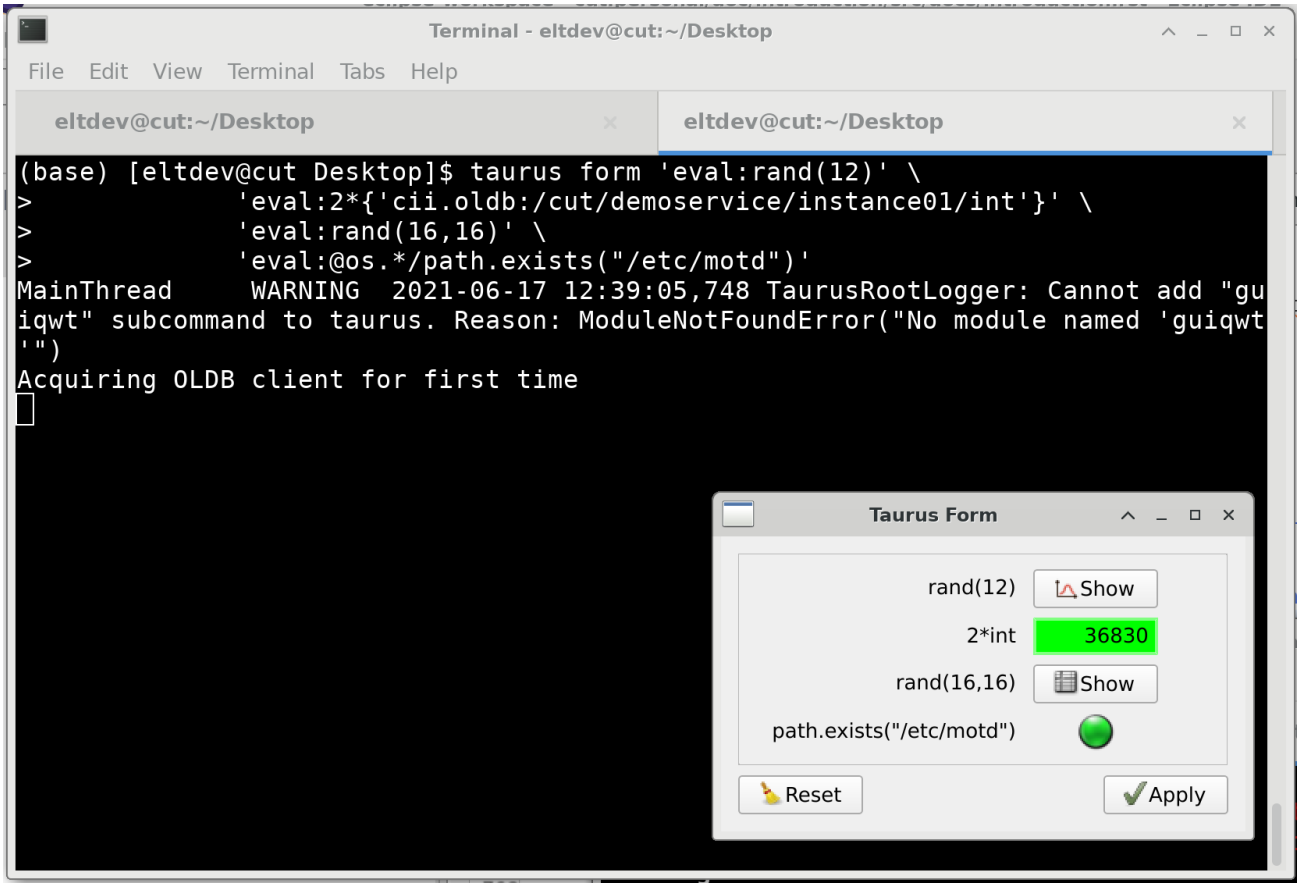


Fig. 27: Evaluation Plugin examples, through command line

Taurus Command Line Tool

Every functionality in taurus can be accessed through the `taurus` command line interface.

To see the complete syntax of taurus command line:

```
$ taurus --help
```

To quickly create a Taurus Form that immediately shows the contents of two datapoints:

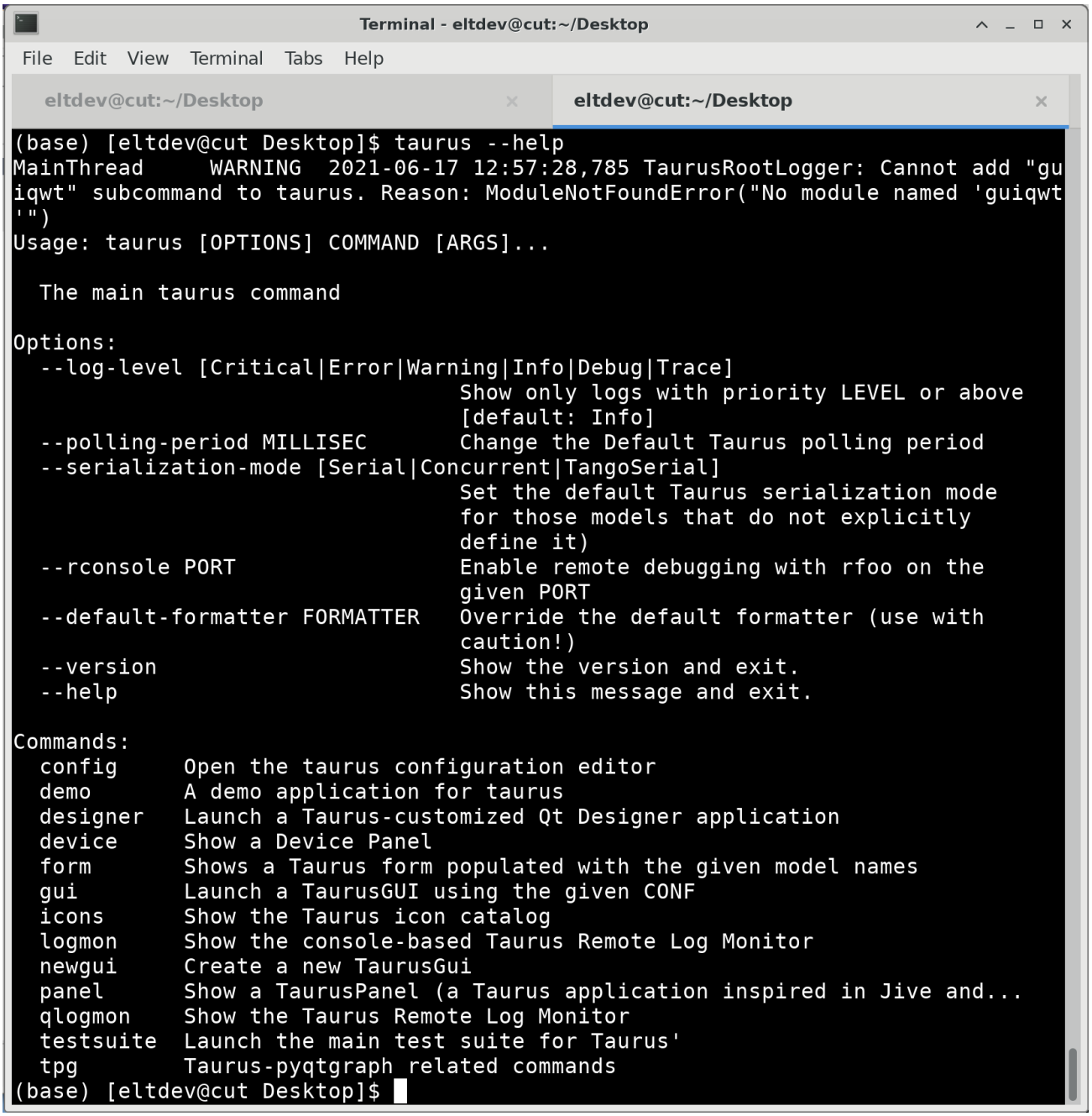


Fig. 28: Taurus command line help

```
$ taurus form 'cii.oldb:/cut/demoservice/instance1/double-scalar-sin'
'cii.oldb:/cut/demoservice/instance1/double-scalar-cos'
```

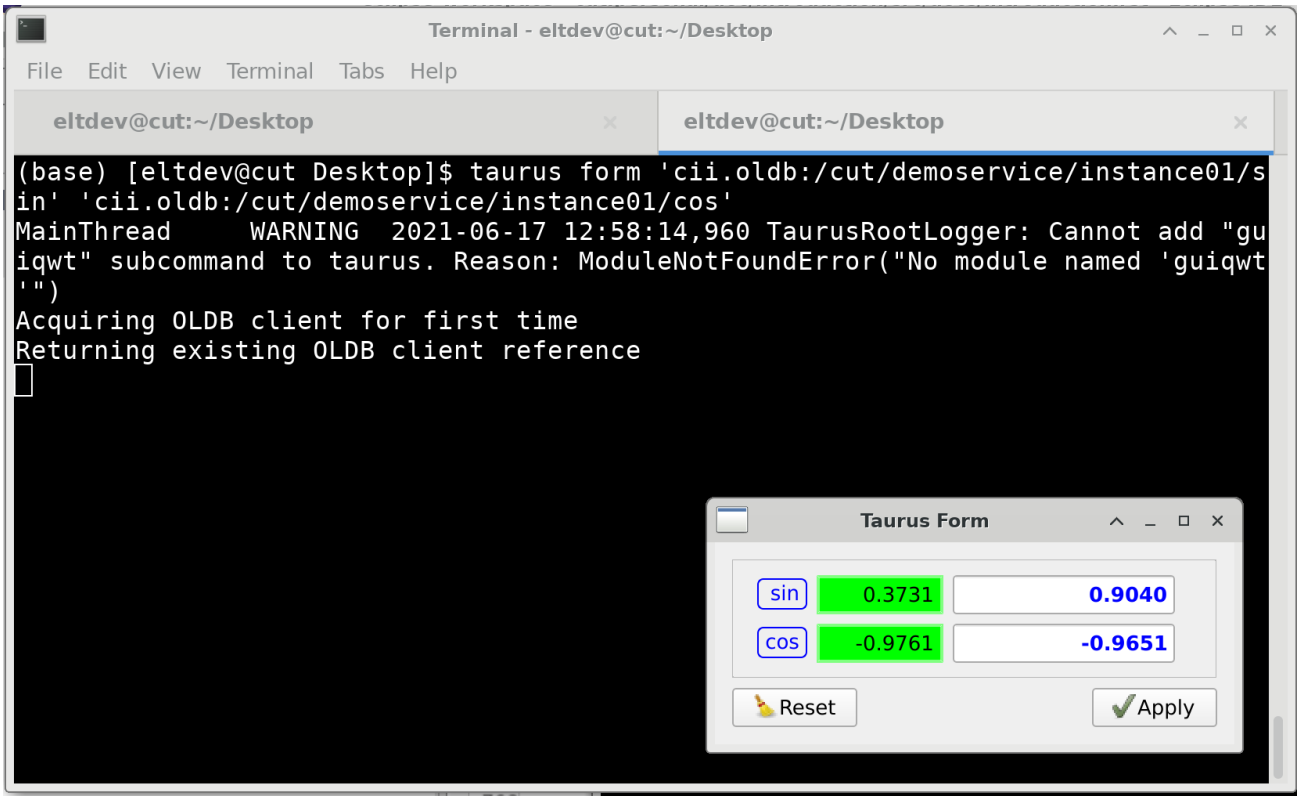


Fig. 29: Requesting two attributes to Taurus Form command line interface

To execute taurus, with trace level logs:

```
$ taurus --log-level Trace form 'cii.oldb:/cut/demoservice/instance1/double-scalar-
↪sin'
'cii.oldb:/cut/demoservice/instance1/double-scalar-cos'
```

Is possible to quickly plot and trend attributes:

```
$ taurus trend 'cii.oldb:/cut/demoservice/instance1/double-scalar-sin'
```

```
$ taurus plot 'cii.oldb:/cut/demoservice/instance1/double-vector-tan'
```

```
1 #!/bin/bash
2
3 taurus form 'cii.oldb:///cut/demoservice/instance1/boolean-scalar-fixed'\
4             'cii.oldb:///cut/demoservice/instance1/string-scalar-fixed'\
```

(continues on next page)

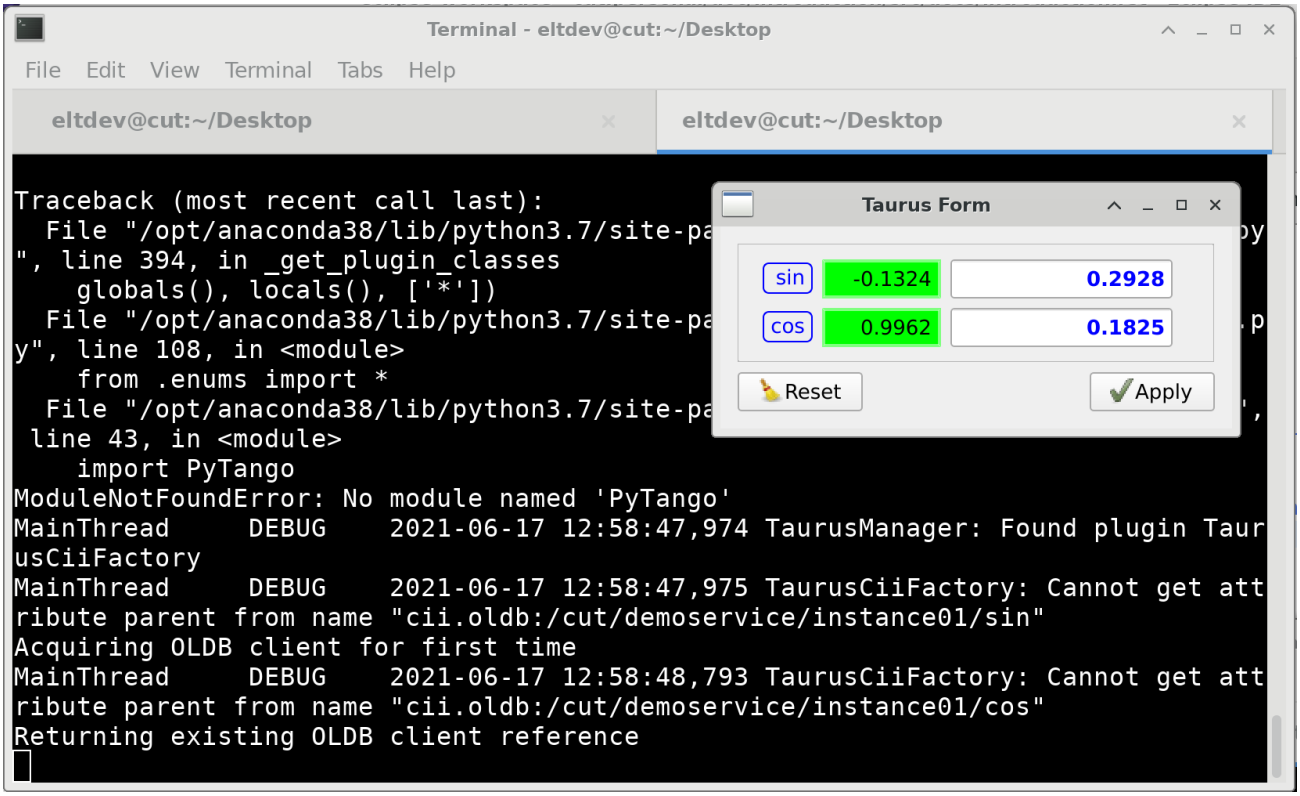
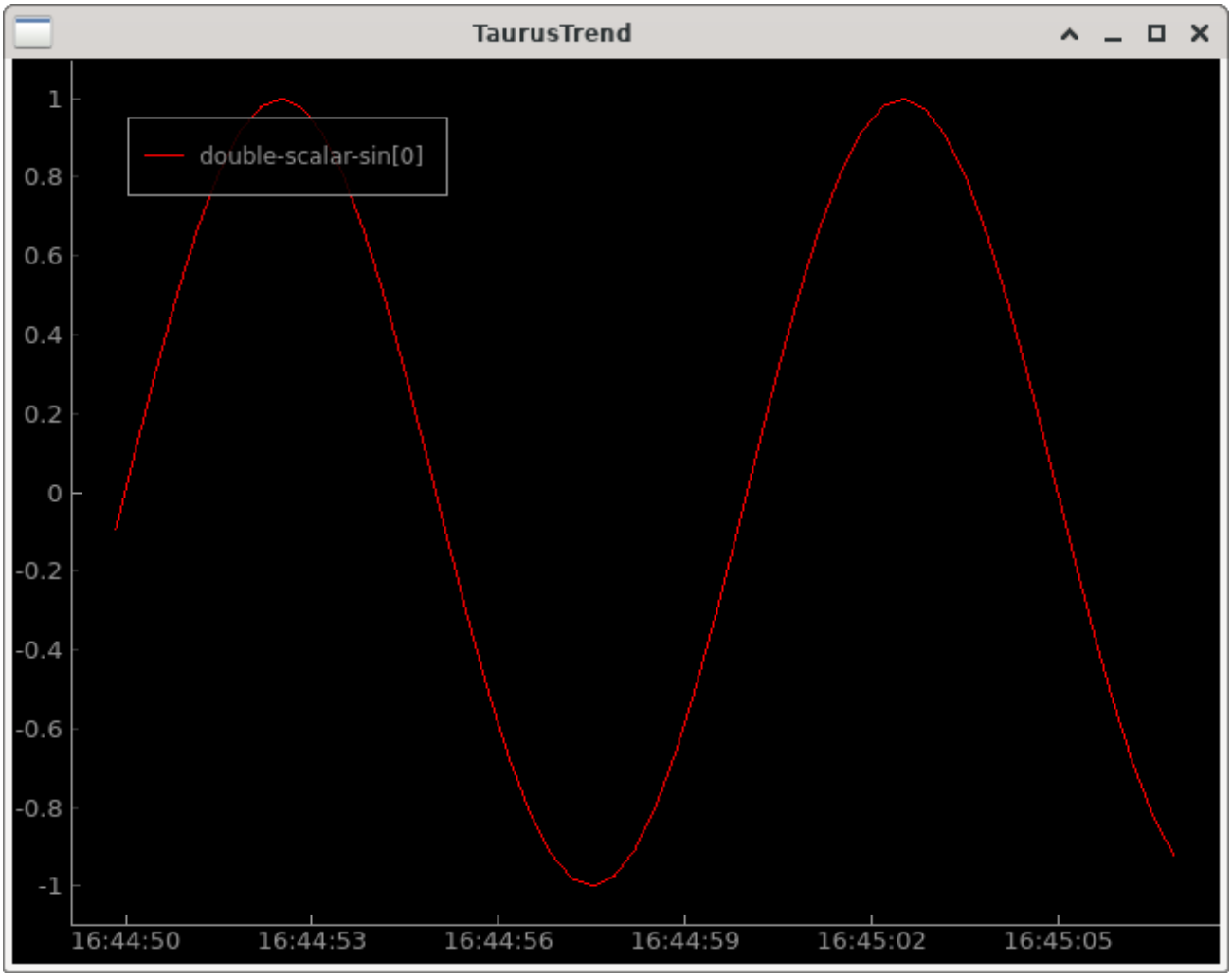


Fig. 30: Taurus command line with trace level logs enabled


TaurusTrend
double-scalar-sin[0]
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
16:44:50
16:44:53
16:44:56
16:44:59
16:45:02
16:45:05

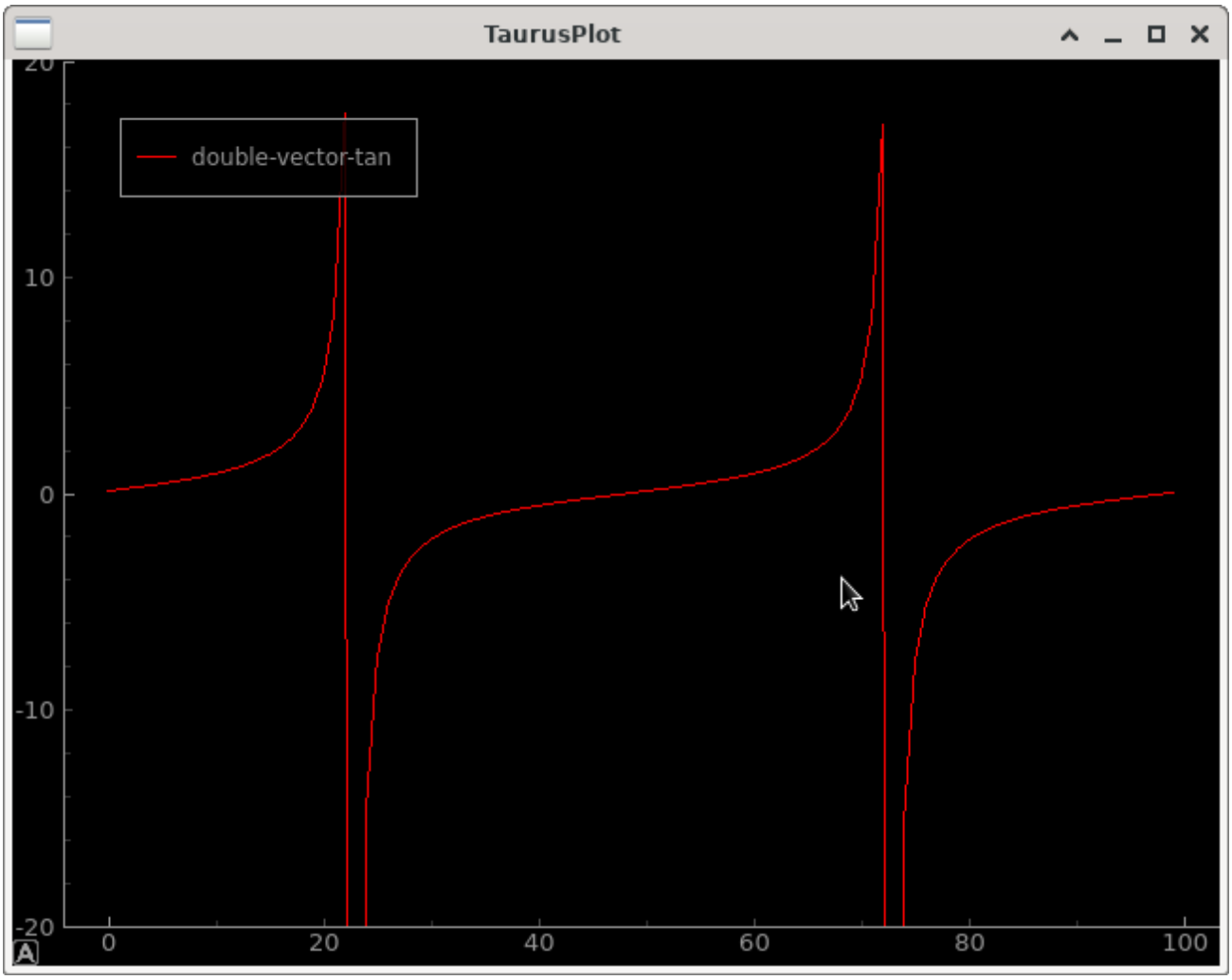


Fig. 31: Taurus command line plotting capabilities. On the left, a trend plot is created for a single point attribute. On the right, a plot of a vector.

(continued from previous page)

```
            'cii.oldb:///cut/demoservice/instance1/double-vector-rand'\
            'cii.oldb:///cut/demoservice/instance1/double-vector-sin'\
            'cii.oldb:///cut/demoservice/instance1/double-vector-cos'\
            'cii.oldb:///cut/demoservice/instance1/double-vector-tan'
```

Start the Demo of all Taurus widgets:

```
$ taurus demo
```

To start the Taurus GUI Wizard:

```
$ taurus newgui
```

Shows every icons available, and its name for QIcon::fromTheme() method:

```
$ taurus icons
```

Taurus Widgets

All Taurus widget inherit from the same base class, the TaurusBaseWidget[69]. This class inherits at some point from BaseConfigurableClass[70] and Logger[71].

- BaseConfigurableClass[72] is in charge of storing and persisting configuration of widgets. This is mostly used by the GUI builder, which uses this functionality to persist position and size of widgets, the models it should present and which roles, among other things.
- Logger[73] uses python logging framework to process every log generated by the Taurus framework.

As an example, here is the inheritance tree of the TaurusLabel widget:

Taurus Display Widgets

One main class of widgets that Taurus offers, is the display widgets. All of them are located in the `taurus.qt.qtgui.display` python module. All of them are read-only, as they are intended for the presentation of information: `taurus.qt.qtgui.display.TaurusLabel`, `taurus.qt.qtgui.display.TaurusLCD` and `taurus.qt.qtgui.display.TaurusLed`

In the same module, there are basic Qt widgets that are then used as base classes by the Taurus widgets.

[69] https://taurus-scada.org/devel/api/taurus/qt/qtgui/base/_TaurusBaseWidget.html
[70] https://taurus-scada.org/devel/api/taurus/qt/qtcore/configuration/_BaseConfigurableClass.html
[71] https://taurus-scada.org/devel/api/taurus/core/util/_Logger.html
[72] https://taurus-scada.org/devel/api/taurus/qt/qtcore/configuration/_BaseConfigurableClass.html
[73] https://taurus-scada.org/devel/api/taurus/core/util/_Logger.html

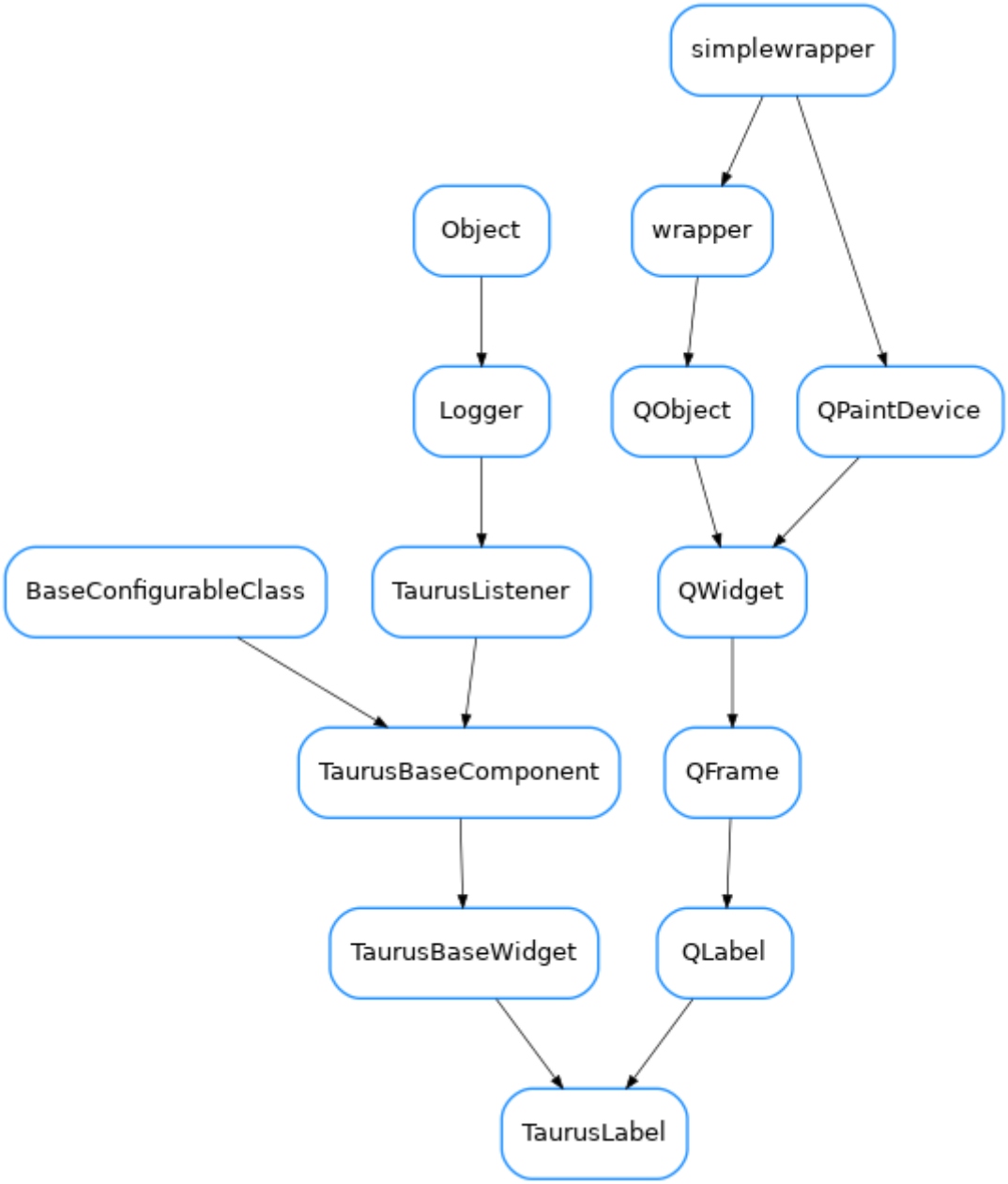


Fig. 32: Taurus Label inheritance tree, from Taurus documentation.

Fig. 33: Display widgets, from top to bottom TaurusLabel, TaurusLCD and TaurusLed

- QLed[74]
- Q7SegDigit[75]
- QLogo[76]
- QPixmapWidget[77]

Taurus widgets do not implement logic nor formatting. Taurus widgets only add three properties and one method, which are used then by its Controller.

model
: a URI stored in a *string*. The widgets are not in charge of getting its model, only to contain the string.

fgRole
: indicates which piece of data from the model, will be used as text.

bgRole
: indicates which piece of data from the model will be used as background color.

handleEvent()
: used to inform widgets of Taurus events. They are mostly related to model changes. These events are forwarded to the controller.

An interesting concept of Taurus is the fragments. Fragments are properties in the model item that we can query. Fragments are accessed by a URI, prepending `#fragname_name`. For example, we can get a short name (**label**) for the datapoint using

```
#!/usr/bin/env python3
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.display import TaurusLabel
```

(continues on next page)

[74] https://taurus-scada.org/devel/api/taurus/qt/qtgui/display/_QLed.html#taurus.qt.qtgui.display.QLed
[75] https://taurus-scada.org/devel/api/taurus/qt/qtgui/display/_Q7SegDigit.html#taurus.qt.qtgui.display.Q7SegDigit
[76] https://taurus-scada.org/devel/api/taurus/qt/qtgui/display/_QLogo.html#taurus.qt.qtgui.display.QLogo
[77] https://taurus-scada.org/devel/api/taurus/qt/qtgui/display/_QPixmapWidget.html#taurus.qt.qtgui.display.QPixmapWidget

(continued from previous page)

```
6
7
8  if __name__ == "__main__":
9      app = TaurusApplication(sys.argv, cmd_line_parser=None,)
10     panel = Qt.QWidget()
11     layout = Qt.QHBoxLayout()
12     panel.setLayout(layout)
13
14     w1, w2 = TaurusLabel(), TaurusLabel()
15     layout.addWidget(w1)
16     layout.addWidget(w2)
17
18     w1.model, w1.bgRole = 'cii.oldb:/cut/demoservice/instance1/double-scalar-sin
   ↪#label', ""
19     w2.model = 'cii.oldb:/cut/demoservice/instance1/double-scalar-sin'
20     panel.show()
21     sys.exit(app.exec_())
```

The application looks like:

Fig. 34: Taurus Labels using bgRole to change the information presented

Normal fragments included in a Taurus model are:

- label
- rvalue.quality
- rvalue.magnitude
- rvalue.units
- rvalue.timestamp
- range
- alarm
- warning

If a model needs, it can add fragments. Fragments are python properties.

Input Widgets

These are widgets that allow the user to modify a presented value. There are located in the `taurus.qt.qtgui.input` module. All of them inherit from TaurusBaseWritableWidget[78] Pressing enter on them will trigger a write operation to the model backend.

All of them present the following graphical design pattern: when the value in the widget differs from the latest read value, the widget is highlighted in blue. This indicates that there is still a change to be committed back into the control system.

- TaurusValueLineEdit[79]
- TaurusValueSpinBox[80]
- TaurusWheelEdit[81]
- TaurusValueComboBox[82]
- TaurusValueCheckBox[83]
- TaurusAttrListComboBox[84]
- GraphicalChoiceDlg[85]

As an example, the following code can show how the widgets would look like:

```
#!/usr/bin/env python3
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.input import TaurusValueLineEdit, TaurusValueSpinBox,␣
↪TaurusWheelEdit


if __name__ == "__main__":
    app = TaurusApplication(sys.argv, cmd_line_parser=None,)
    panel = Qt.QWidget()
    layout = Qt.QVBoxLayout()
    panel.setLayout(layout)

```

(continues on next page)

[78] https://taurus-scada.org/devel/api/taurus/qt/qtgui/base/_TaurusBaseWritableWidget.html
[79] https://taurus-scada.org/devel/api/taurus/qt/qtgui/input/_TaurusValueLineEdit.html#taurus.qt.qtgui.input.TaurusValueLineEdit
[80] https://taurus-scada.org/devel/api/taurus/qt/qtgui/input/_TaurusValueSpinBox.html#taurus.qt.qtgui.input.TaurusValueSpinBox
[81] https://taurus-scada.org/devel/api/taurus/qt/qtgui/input/_TaurusWheelEdit.html#taurus.qt.qtgui.input.TaurusWheelEdit
[82] https://www.taurus-scada.org/devel/api/taurus.qt.qtgui.input-TaurusValueComboBox.html
[83] https://www.taurus-scada.org/devel/api/taurus.qt.qtgui.input-TaurusValueCheckBox.html
[84] https://www.taurus-scada.org/devel/api/taurus.qt.qtgui.input-TaurusAttrListComboBox.html
[85] https://www.taurus-scada.org/_modules/taurus/qt/qtgui/input/choicedlg.html#GraphicalChoiceDlg

(continued from previous page)

```
    w1, w2, w3 = TaurusValueLineEdit(), TaurusValueSpinBox(), TaurusWheelEdit()
    layout.addWidget(w1)
    layout.addWidget(w2)
    layout.addWidget(w3)

    w1.model = 'cii.oldb:/cut/demoservice/instance1/string-scalar-timestamp'
    w2.model = 'cii.oldb:/cut/demoservice/instance1/int-scalar-add'
    w3.model = 'cii.oldb:/cut/demoservice/instance1/double-scalar-sin'
    panel.show()
    sys.exit(app.exec_())
```

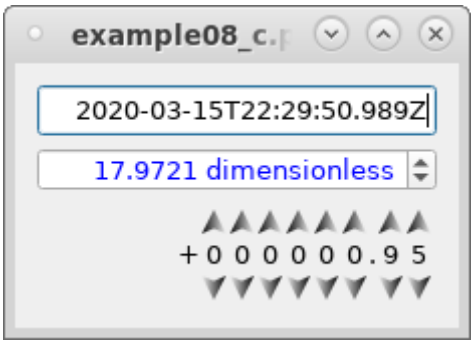


Fig. 35: Inputs widgets, from top to bottom, TaurusValueLineEdit[86], TaurusValueSpinBox[87], TaurusWheelEdit[88].

Plotting Widgets

Taurus provides two set of plotting widgets, based on different libraries. One of them is based on guiqwt[89], and the other in PyQtGraph[90].

Due to availability of libraries in DevEnv, pyqtgraph support is included mainly for investigation purposes. The final plotting solution is not yet determined, but it should adhere to our requirements and taurus widget interface.

The widgets taurus_pyqtgraph[91] offers:

- TaurusTrend[92] Plots the evolution over time of a scalar attribute.

[86] https://taurus-scada.org/devel/api/taurus/qt/qtgui/input/_TaurusValueLineEdit.html#taurus.qt.qtgui.input.TaurusValueLineEdit
[87] https://taurus-scada.org/devel/api/taurus/qt/qtgui/input/_TaurusValueSpinBox.html#taurus.qt.qtgui.input.TaurusValueSpinBox
[88] https://taurus-scada.org/devel/api/taurus/qt/qtgui/input/_TaurusWheelEdit.html#taurus.qt.qtgui.input.TaurusWheelEdit
[89] https://github.com/PierreRaybaut/guiqwt
[90] http://www.pyqtgraph.org/
[91] https://github.com/taurus-org/taurus_pyqtgraph
[92] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/trend.py

- TaurusPlot[93] Plots a curve based on an Array attribute.

In the following example, the sine scalar attribute is used as model for a TaurusTrend[94] widget:

```
#!/usr/bin/env python3
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.plot import TaurusTrend, TaurusPlot


if __name__ == "__main__":
    app = TaurusApplication(sys.argv, cmd_line_parser=None,)

    panel = TaurusTrend()
    model = ['cii.oldb:/cut/demoservice/instance1/double-scalar-sin']
    panel.setModel(model)
    panel.show()
    sys.exit(app.exec_())
```

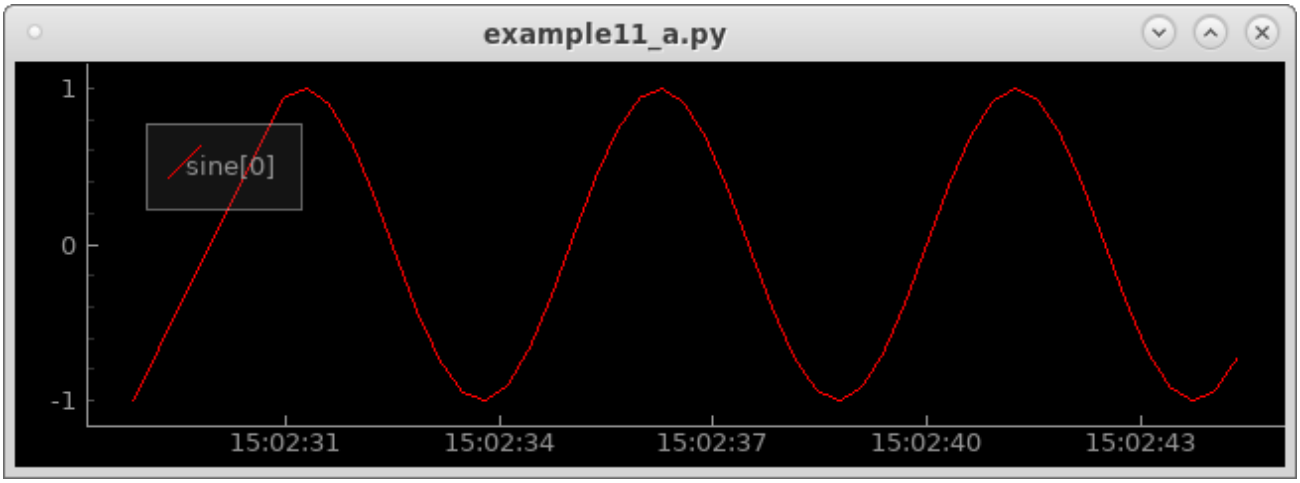


Fig. 36: TaurusTrend[Page 61, 95] widget from the program above.

This example is a bit more complex. Here we used a layout manager, to put side by side two widgets, one for plotting, the other one for trending. TaurusPlot[96] is used to plot an Array attribute, gotten from the evaluation[97] scheme, while the TaurusTrend[98] widgets trends a sine and cosine scalar attributes:

[93] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/plot.py
[94] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/trend.py
[95] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/trend.py
[96] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/plot.py
[97] https://taurus-scada.org/devel/api/taurus/core/evaluation.html
[98] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/trend.py

```python
#!/usr/bin/env python3
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.plot import TaurusTrend, TaurusPlot

if __name__ == "__main__":
    app = TaurusApplication(sys.argv, cmd_line_parser=None,)
    panel = Qt.QWidget()
    layout = Qt.QVBoxLayout()
    panel.setLayout(layout)
    plot = TaurusPlot()
    plot_model = ['cii.oldb:/cut/demoservice/instance1/double-vector-sin']
    plot.setModel(plot_model)
    trend = TaurusTrend()
    trend_model = ['cii.oldb:/cut/demoservice/instance1/double-scalar-sin',
                   'cii.oldb:/cut/demoservice/instance1/double-scalar-cos']
    trend.setModel(trend_model)
    layout.addWidget(plot)
    layout.addWidget(trend)
    panel.show()
    sys.exit(app.exec_())
```

Qt and UI Files

The final step of this introduction, is to be able to create UI file, and understand how these files are used.

In order to do so, we will use Qt Designer. You can start it using a terminal:

```
$ taurus designer
```

Once opened, the designer will ask us what kind of UI we want to create through the *New Form* Dialog. On the list of templates from the left, choose *Widget*. On the lower part of the *New Form* Dialog, click on the *Create* Button.

The Designer has three main sections:

- On the left, a *Widget Box* Docking Window.
- On the center, a grey area where the documents we are editing are located.
- On the right side, several Docking Windows, the most important are the *Object Inspector*, and the *Property Editor*.

[99] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/plot.py
[100] https://github.com/taurus-org/taurus_pyqtgraph/blob/master/taurus_pyqtgraph/trend.py

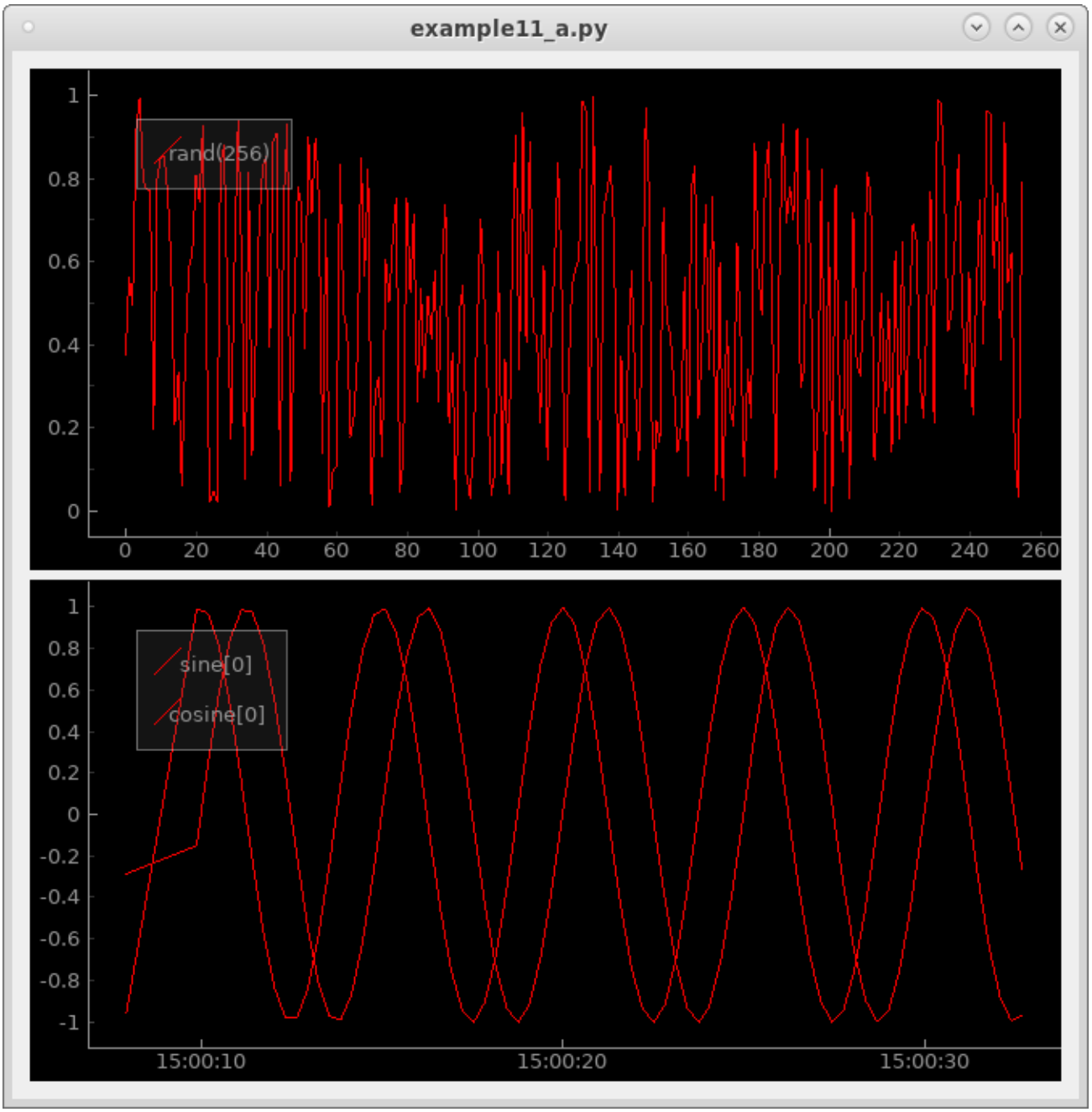


Fig. 37: TaurusPlot[Page 62, 99] and TaurusTrend[Page 62, 100] widgets from the program above.

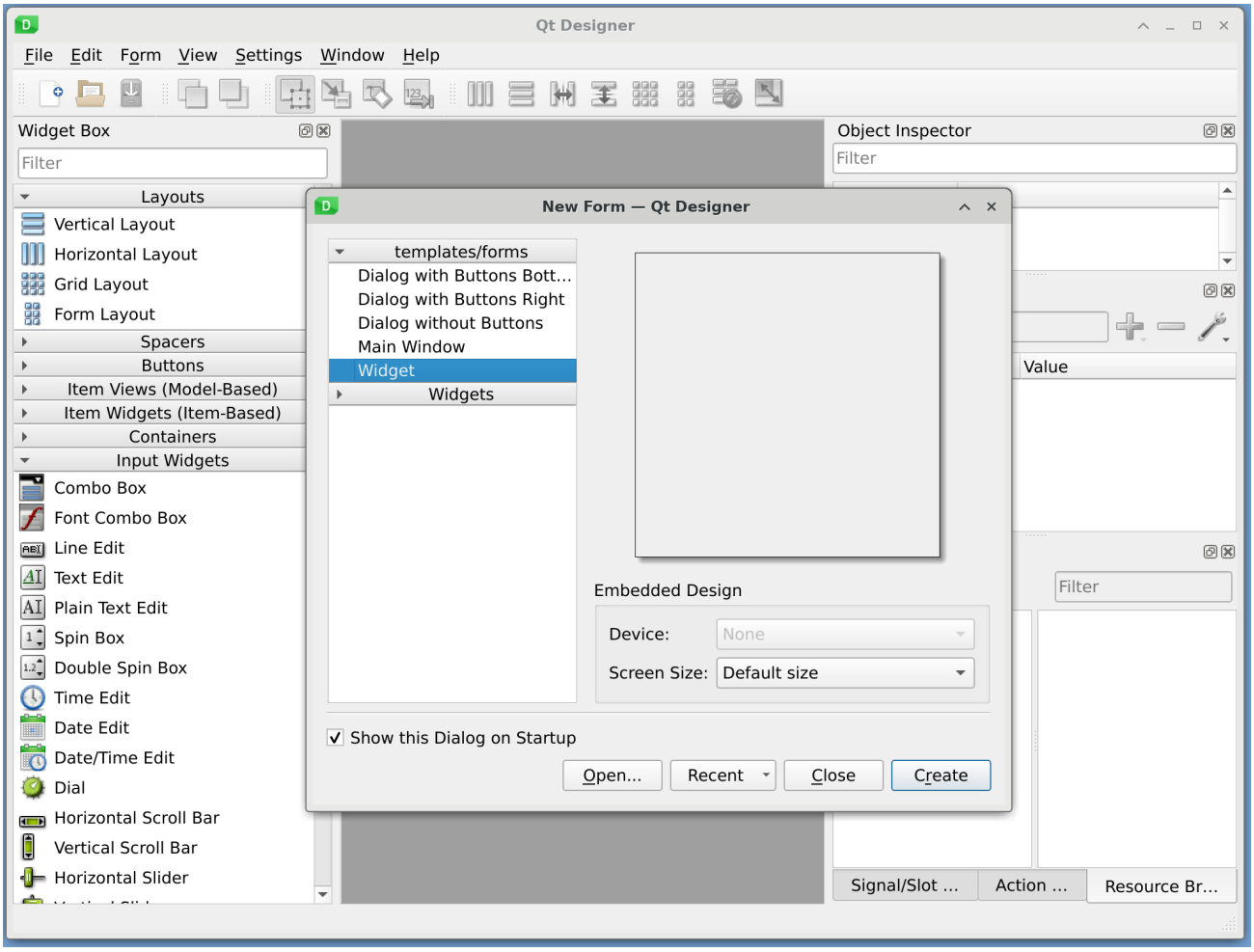


Fig. 38: Qt Designer - New Form Dialog

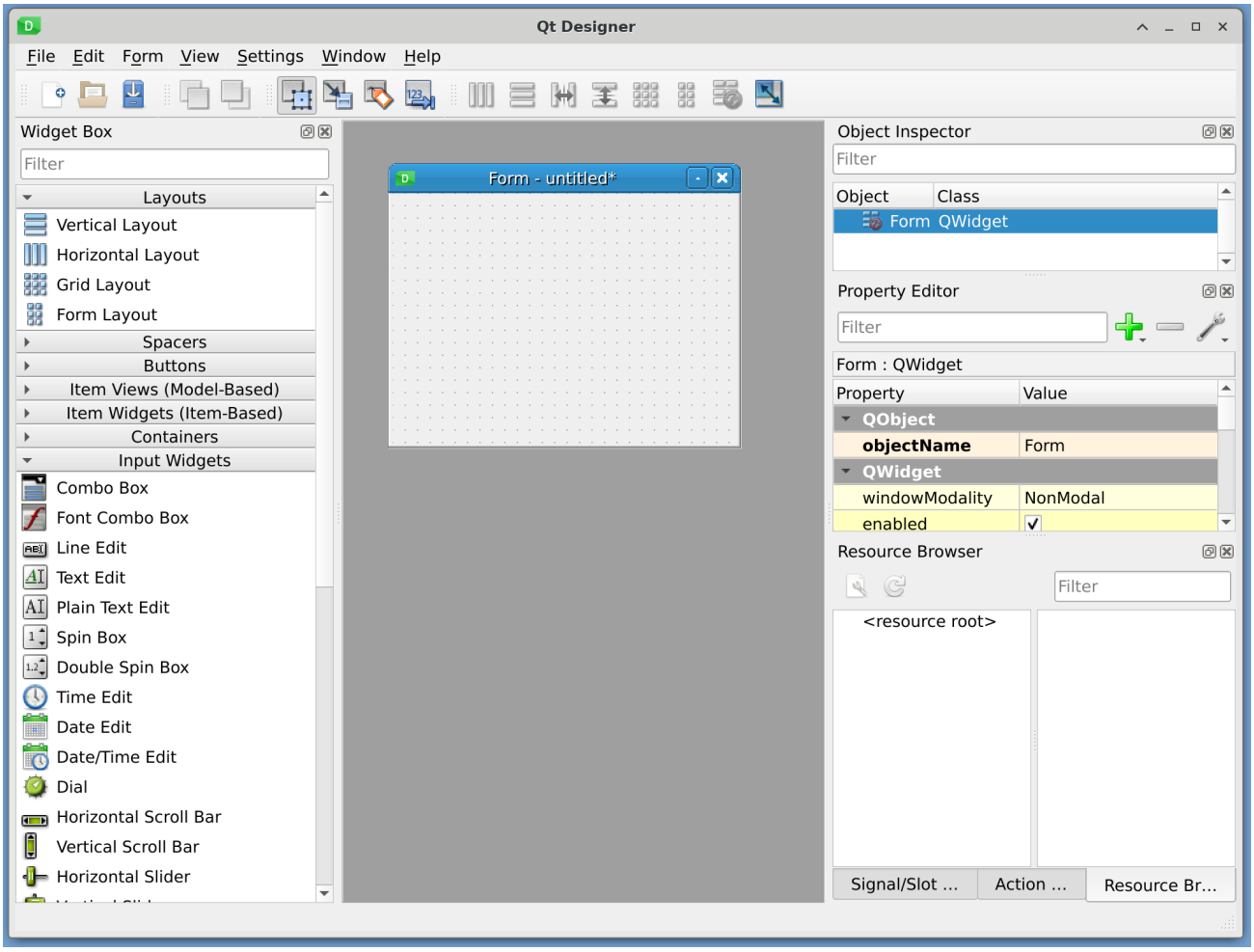


Fig. 39: Qt Designer - Main View

From the *Widget Box* Docking Window, we will drag the widgets and elements we need into the UI. Special characteristics of these widgets can be edited in the *Property Editor* Docking Window.

We will start by changing the name of the widget. In the *Property Editor* Docking Window, look for the row *Object Name*. Change it to “CustomWidget”.

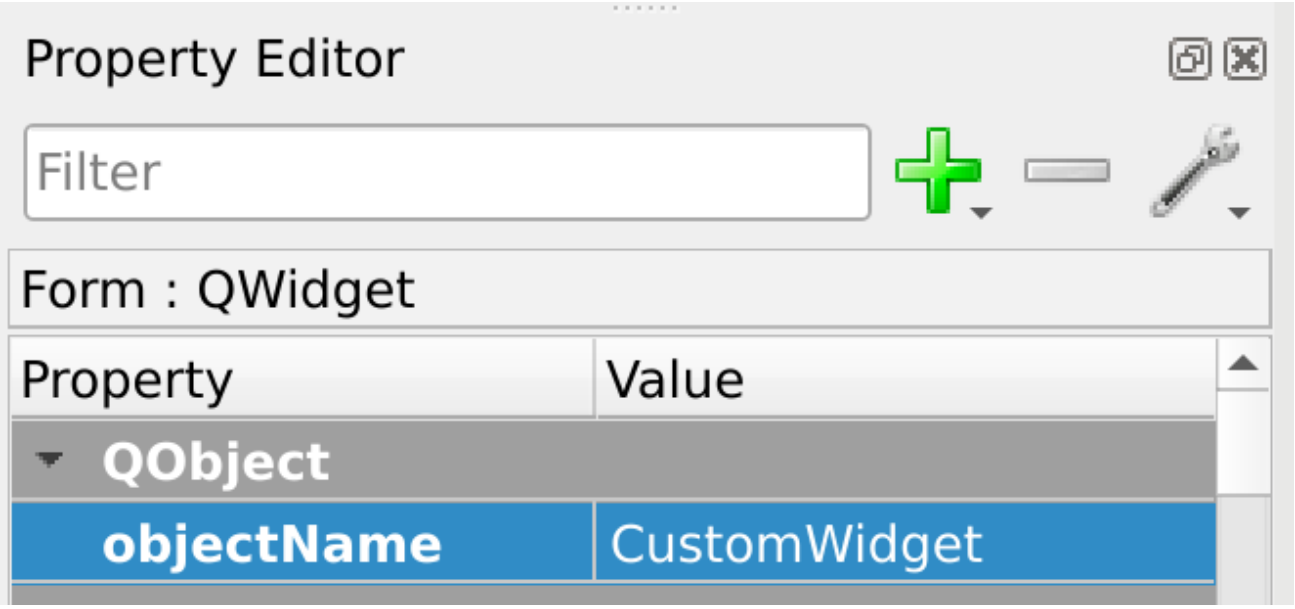


Fig. 40: Qt Designer - Property Editor

Now look in the *Widget Box* Docking Window for the *Horizontal Layout*. Drag and drop this element into the UI, in the upper section, and then repeat, dropping it in the lower section.

Next, look in the *Widget Box* Docking Window for the *Label* Widget. Drag and drop this widget inside of the upper horizontal layout. When you finish dropping it, it should be contained by the Layout. Repeat for the lower horizontal layout.

In our following step, we are looking in the *Widget Box* Docking Window for the *Line Edit* Widget. Drag but do not drop the widget yet. Move the mouse over the upper layout. See that a blue appears where it will be positioned. If the blue line is on the right side of the Label, then drop the widget.

We will repeat the same maneuver, but instead with a *Spin Box* Widget, and dropping in into the lower horizontal layout.

Now we will set up the main layout of the CustomWidget. Right click on an empty space on the widget. Navigate to *Context Menu* → *Layout* → *Lay Out Vertically* Entry and click it.

This will make our entire UI responsive to resize events. (and scaling).

Finally we will double click on the first *Label* Widget and with that, we can change the “text” property of the widget. Enter “Name:”.

Do the same for the lower *Label* Widget, but enter “Age:”.

Save this file as CustomWidget.ui

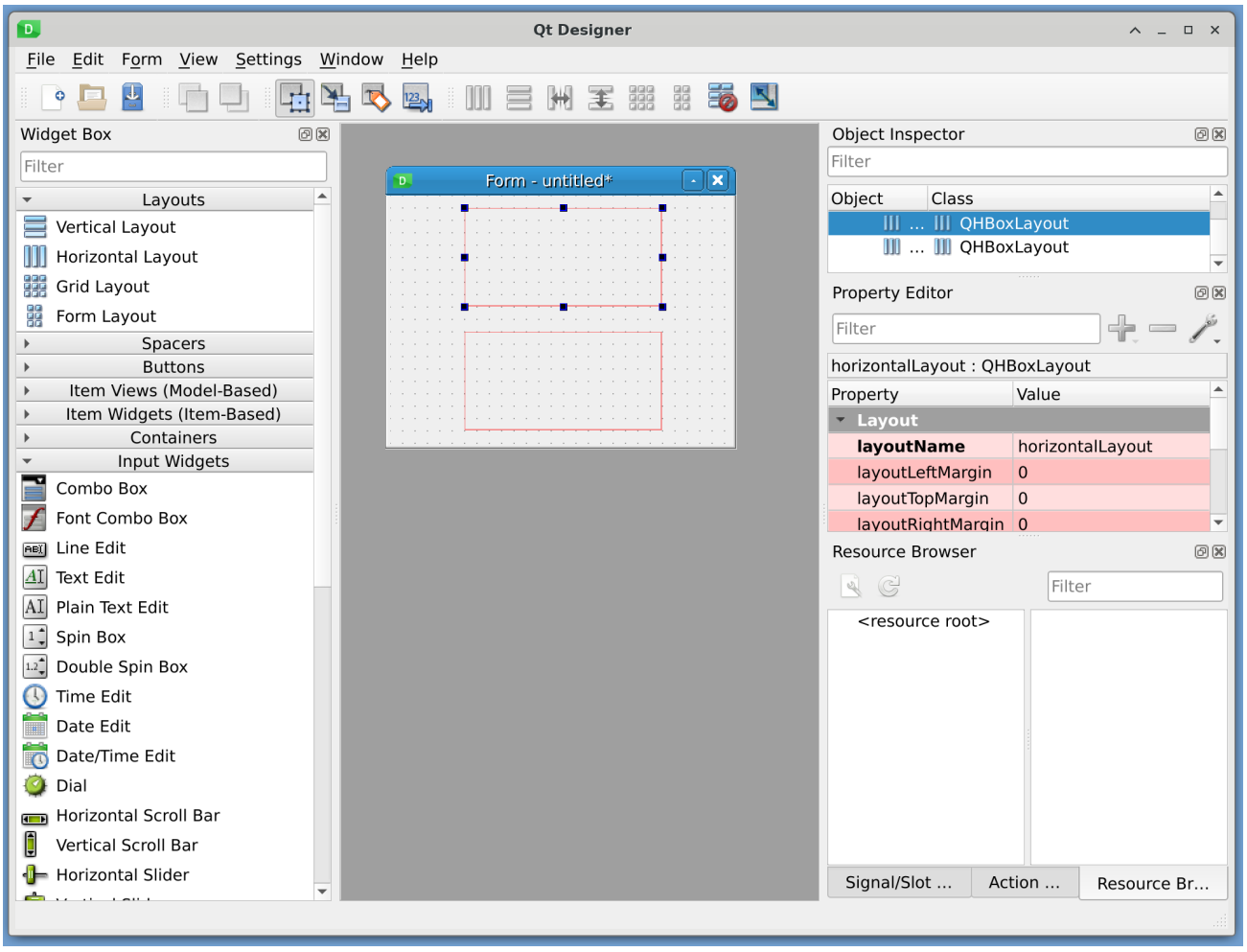


Fig. 41: Qt Designer - Two Horizontal Layouts

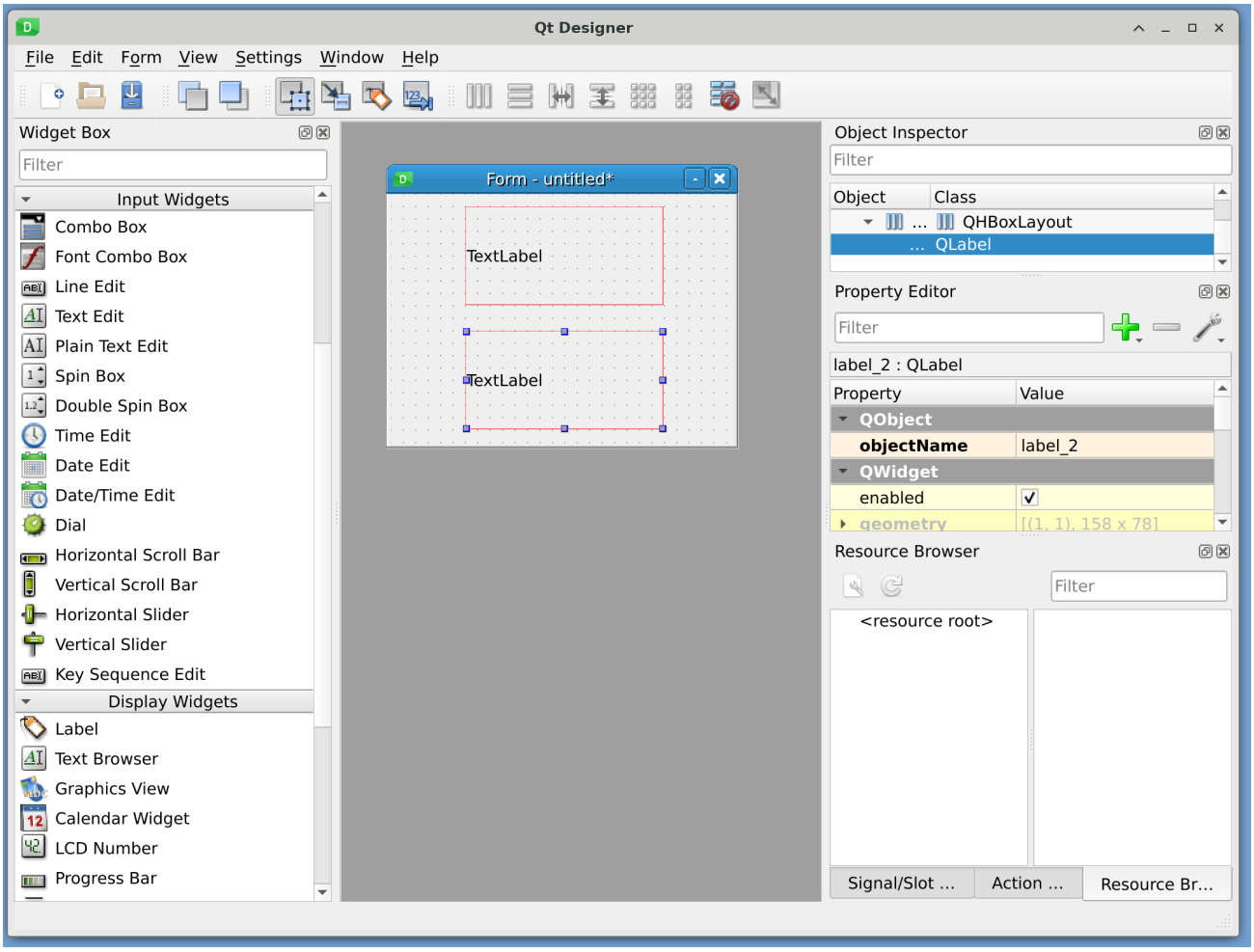


Fig. 42: Qt Designer - Two Labels

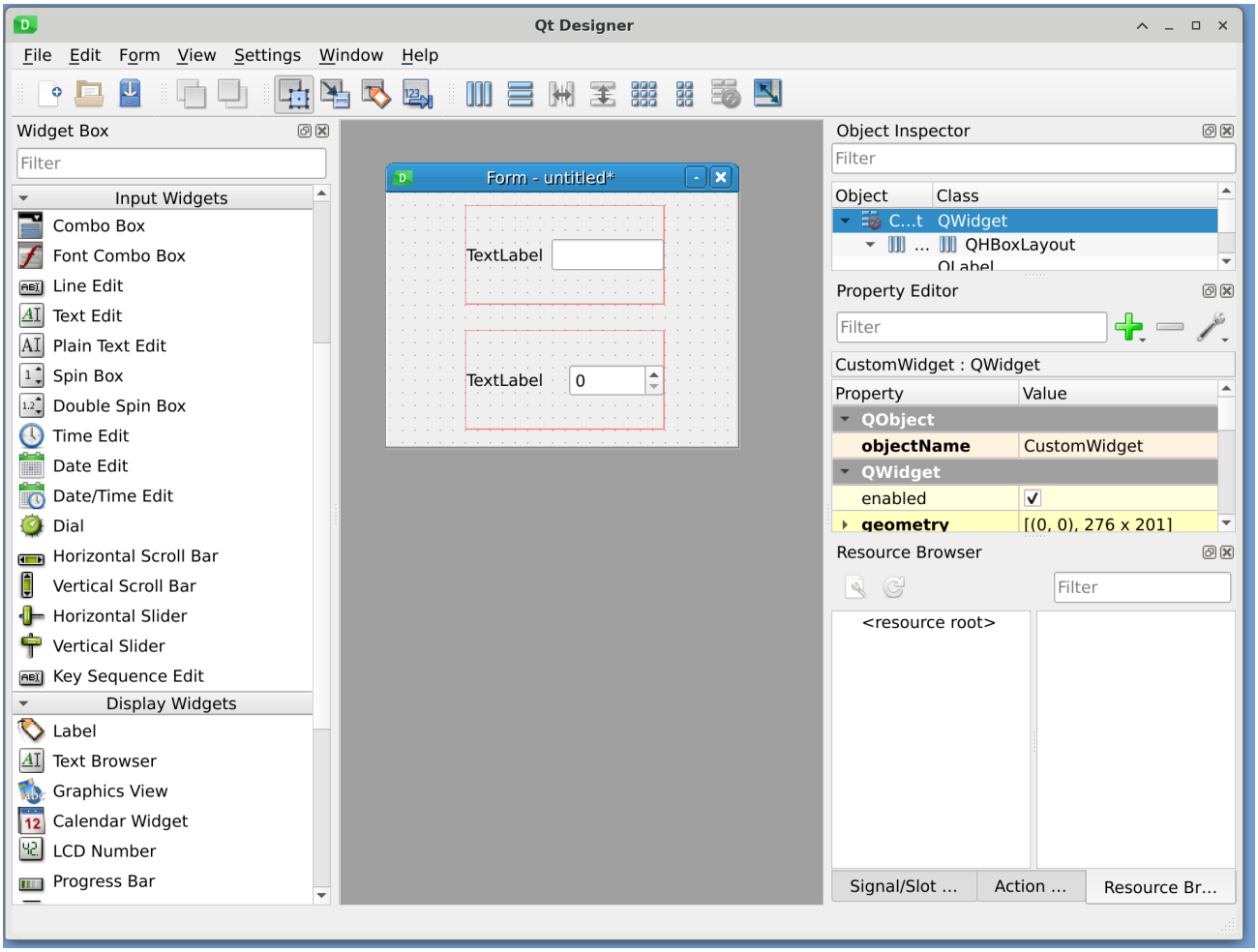


Fig. 43: Qt Designer - Two Input Widgets: Line Edit and Spin Box

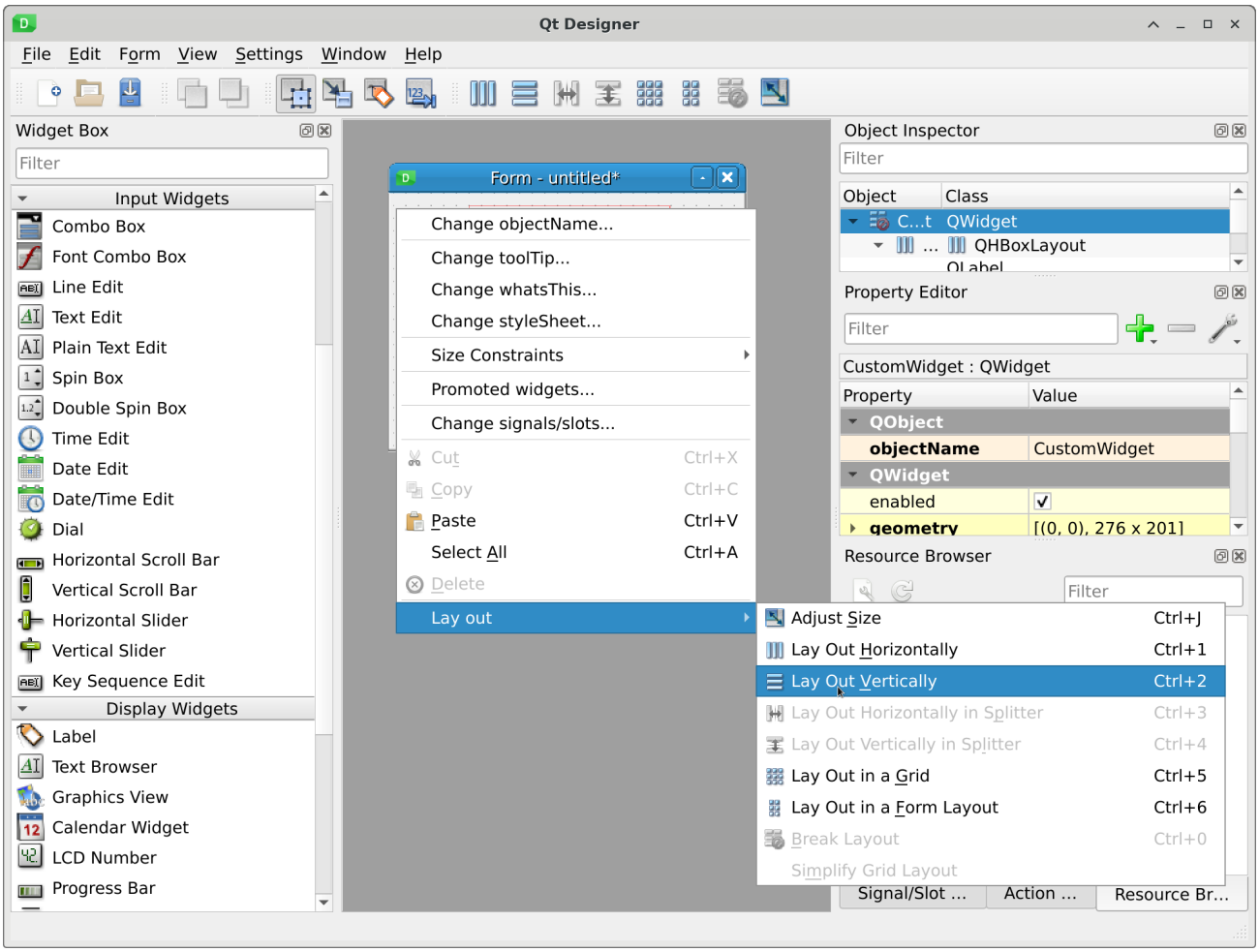


Fig. 44: Qt Designer - CustomWidget layout.

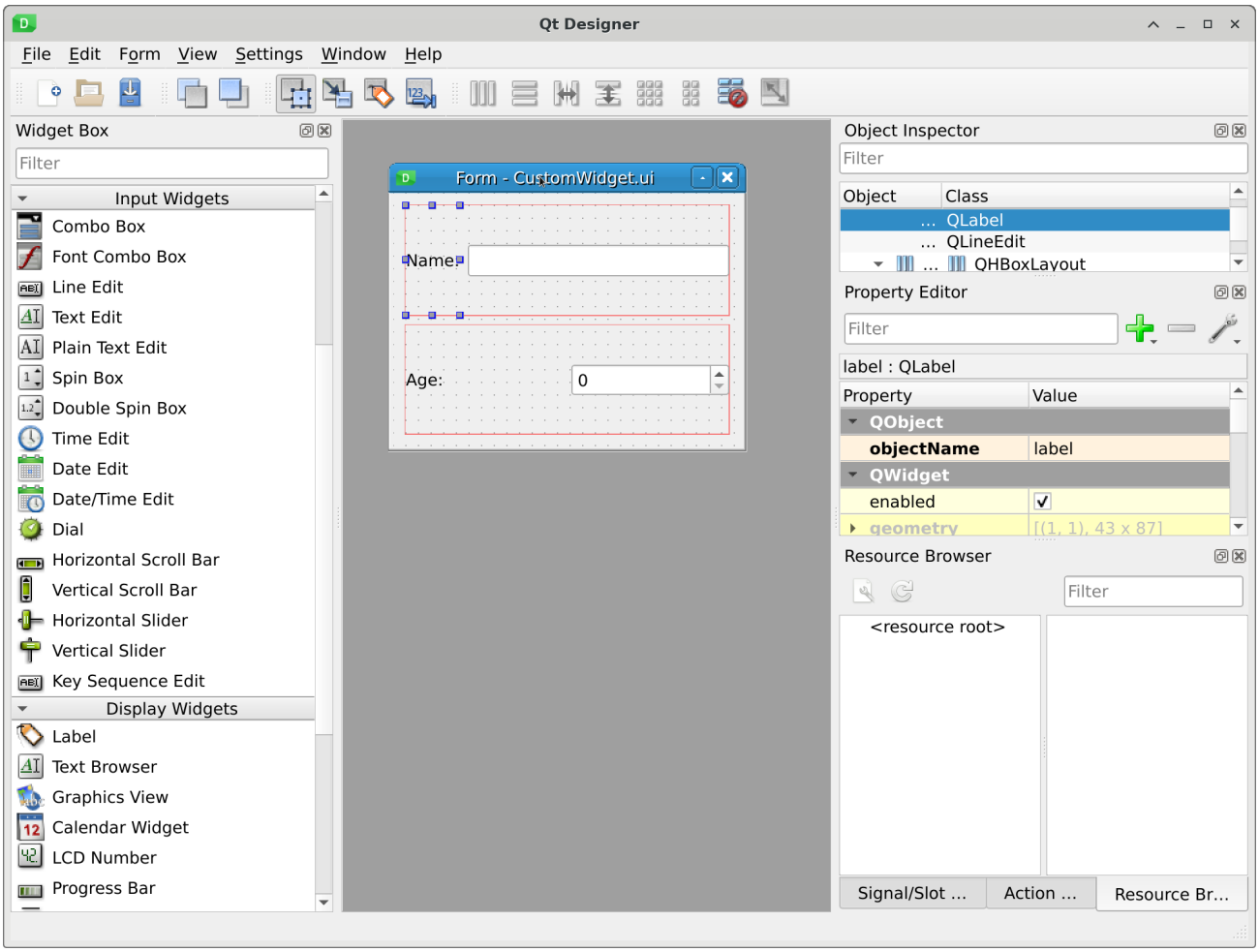


Fig. 45: Qt Designer - Text properties on Labels.

In order to use this file, we need to generate the code from it. This CustomWidget.ui file is a XML representation of the UI above. To generate python code from it, execute:

```
$ pyside6-uic -g python CustomWidget.ui > Ui_CustomWidget.py
```

And finally, we need to create the python application that will use this new python code, and render the UI for us. See the code below for it.

```
#!/usr/bin/env python3
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication
from Ui_CustomWidget import Ui_CustomWidget


class CustomWidget(Qt.QWidget):

    def __init__(self, parent=None):
        Qt.QWidget.__init__(self, parent)
        self.ui = Ui_CustomWidget()
        self.ui.setupUi(self)


if __name__ == "__main__":
    app = TaurusApplication(sys.argv)
    widget = CustomWidget()
    widget.show()
    sys.exit(app.exec_())
```

The main difference from the example we have seen so far is the inclusion of a new class. This class inherits from QWidget[101], the same one we have used to create our UI. This is very important: The type of top-level widget we use in the UI file must match the Class we inherit from.

The class only has defined the `__init__` method for it. In it, we initialize its only parent (python requires explicit initialization of parents), then create a new object from the class that is provided to us from the `pyside6-uic` execution.

Finally, we call a method from that class, `setupUi(self)`. This method will create the UI for us, as if we had programmed it ourselves. All the UI elements will be available at the `self.ui` attribute.

And with that, the widget is ready to be used. We proceed as usual, with the `show()` and `app.exec_()` methods, to show and enter the event engine.

[101] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QWidget.html

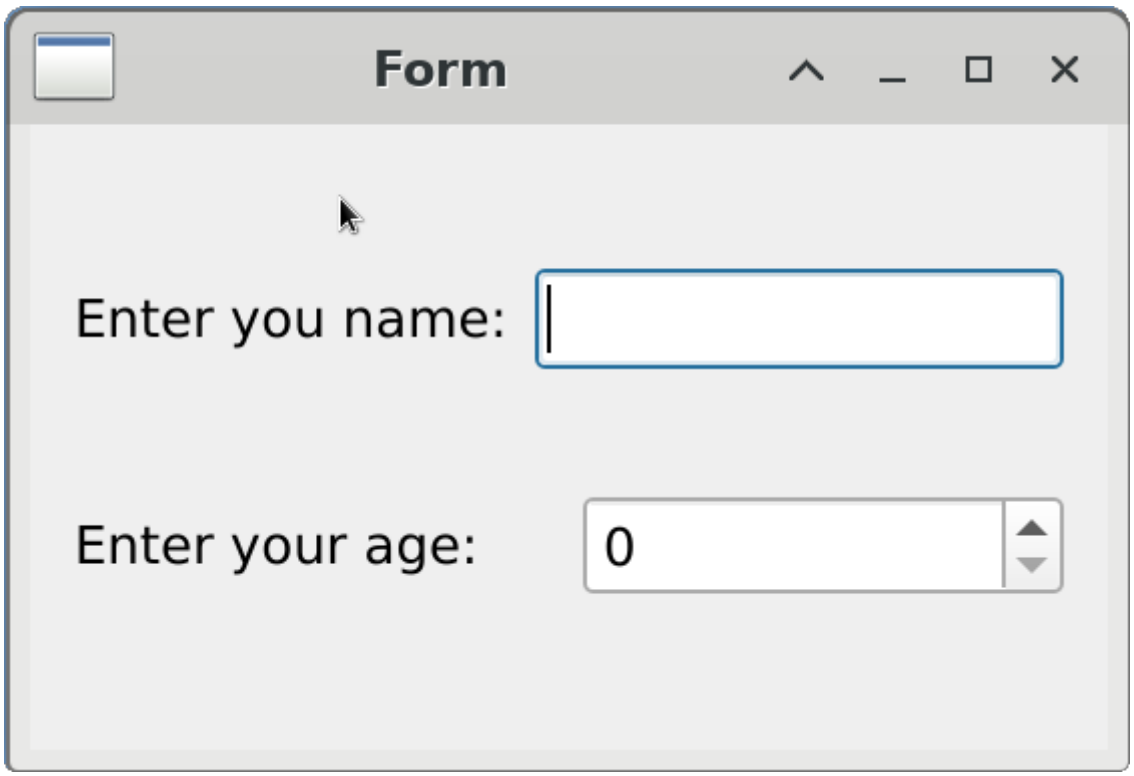


Fig. 46: Qt Designer - CustomWidget running.

4.2 Python Application Tutorial

This documents teaches the reader how to develop a basic GUI application using Python, PySide and Taurus, from scratch.

The reader will learn how to configure WAF, create an entry point program, design a UI, and create the implementation of the behavior for said UI.

Download, Compilation, and Running

This document guides the user on the creation of a new Python GUI Application, using Python, WAF modules and basic Taurus and Qt functionality. As this is a programming tutorial, you will produce file and code as instructed. You can also download a finalized version of the code from `https://gitlab.eso.org/ecos/cut/-/tree/master/examples/`, so our first step will be to download it.

```
$ git clone https://gitlab.eso.org/ecos/cut
```

Note: Cloning over HTTPS from `gitlab.eso.org` may require authentication. If you are prompted for credentials, create a Personal Access Token in GitLab (with at least the `read_repository` scope) and use it in place of your password. See the GitLab Personal Access Tokens[102] documentation for details.

```
$ mv cut/examples/python_application ./
$ rm -rf cut
```

The directory we will be using for this tutorial is the `python_application`, and it contains the code for a simple Python application. This is a waf project on its own, so it can be build in an independent manner from the rest of Control UI Toolkit. In order to do so, the reader must execute:

```
$ cd python_application
$ waf configure install
```

And to execute the application:

```
$ cutPaeGui
```

[102] https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html

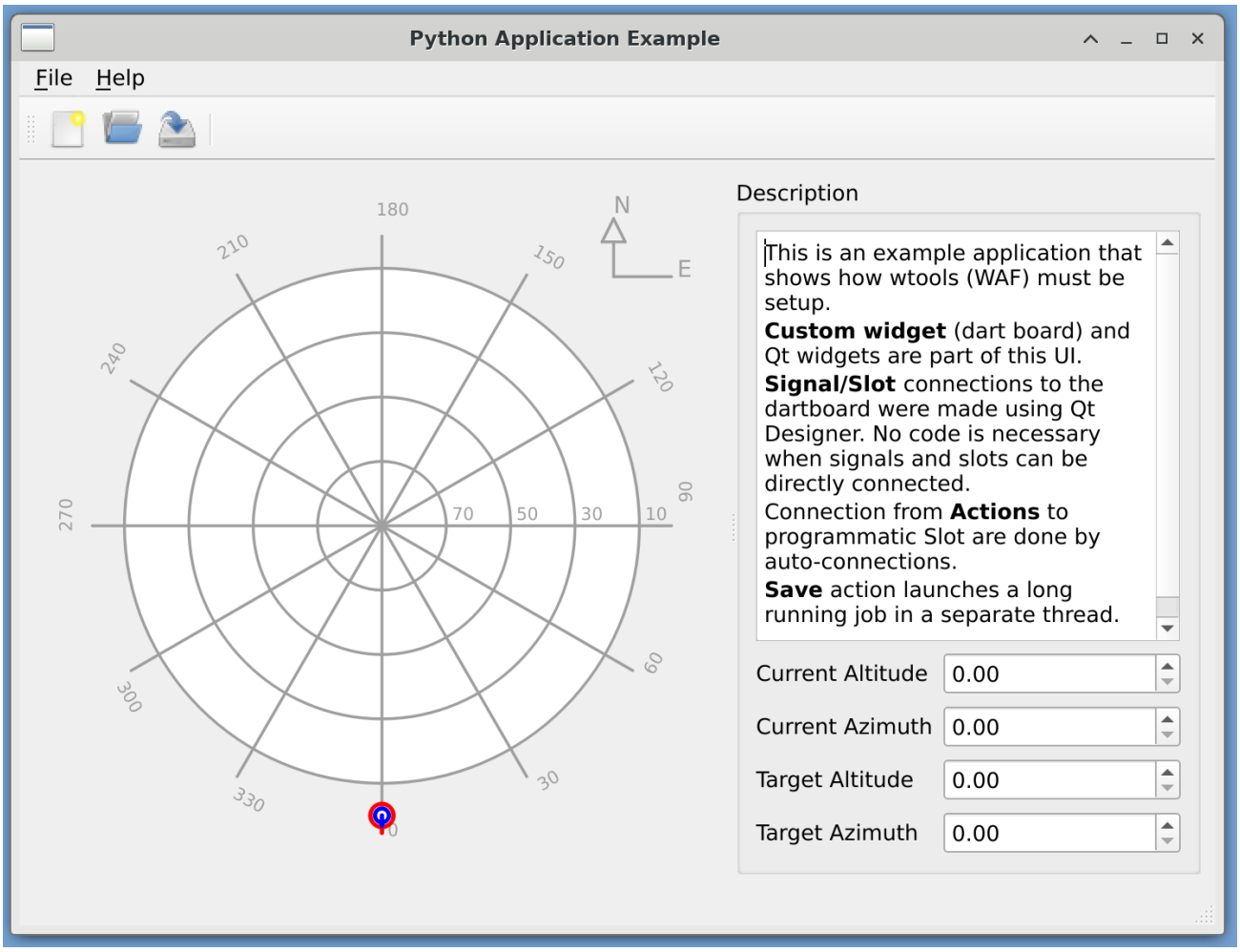


Fig. 47: Python Application Example: We will create an application like this

Project Structure

This is the structure of the project. We use as build system WAF[103], and on top wtools[104] extension to make ELT application definition easier:

```
python_application/
├── PythonApplication
│   ├── src
│   │   ├── python_application_example
│   │   │   ├── applicationwindow.py
│   │   │   ├── __init__.py
│   │   │   └── mainwindow.ui
│   │   └── paegui.py
│   └── wscript
└── wscript
```

Project wscript

The first file we should take a look is `python_application/wscript`:

```
1  #!/usr/bin/env python3
2  import os
3  from wtools.project import declare_project
4
5
6  def configure(cnf):
7      # We check which version of qt6 is requested and set the proper feature for␣
   ↪later
8      if hasattr(cnf, "want_qt6") and cnf.want_qt6 is True:
9          cnf.check_python_module("PySide6")
10     else:
11         cnf.check_python_module("PySide2", condition="ver >= num(5, 14, 1) and ver
   ↪< num(6, 0, 0)")
12     cnf.check_python_module("taurus")
13     cnf.check_python_module("elt.cut")
14     cnf.check_python_module("CutWidgets")
15
16
17 def qtcallable(cnf):
18     # We can do some checks here, for example to discriminate between Qt6 and Qt5:
19     if os.environ.get("ELT_RELEASE", "6").startswith("6"):
```

(continues on next page)

[103] https://waf.io/book/
[104] https://www.eso.org/~eltmgr/documents/latest/wtools-docs/archive/html/index.html

(continued from previous page)

```
20          return dict(version=6)
21      else:
22          return dict(version=5)
23
24
25  declare_project(
26      "PythonApplicationExample",
27      version="3.4.1-rc1",
28      requires=["cxx", "qt", "python", "pyqt"],
29      recurse="PythonApplication",
30      qt=qtcallable,
31  )
```

The syntax of this file is that of python. WAF uses several `wscript` files to describe its project and modules build instructions.

The project is named `PythonApplicationExample`, and this file is in charge of setting up dependencies. Here we check for the presence in our development environment of any required library or python module for the compilation and execution of our application. In particular, we check for:

- PySide (python bindings for Qt)
- taurus
- elt.cut
- CutWidgets

The `declare_project()` method in line 25 specifies the require keyword with values `python` `qt` and `pyqt` strings, which indicates that the included modules *could* be python programs and uses qt.

Then, the `recurse` list indicates which directory will be considered as module for this project. This WAF project has only 1 WAF module: `PythonApplication`.

The `PythonApplication` WAF module is the most important one. To create a basic UI application, we need the following files and directories.

- One python package: the directory `PythonApplication/src/python_application_example`
  - Its `__init__.py` file
  - At least one UI file `mainwindow.ui`,
  - The implementation of said UI file `applicationwindow.py`
- An entry point script `paegui.py`
- One `wscript` that handles the module.

Module wscript

`python_application/PythonApplication/wscript`:

```
1  #!/usr/bin/env python3
2  from wtools import module
3
4  # The declare_pyqtprogram instruction will automatically find
5  # the QT Resources, and the UI files, and run the necessary
6  # commands to produce the python versions of them.
7
8  module.declare_pyqtprogram(
9      target='cutPaeGui',
10     py_program="src/paegui.py",
11     py_package="src/python_application_example",
12     use=[
13         "cut.widgets.bindings",
14         "cut.task",
15     ]
16 )
```

This file uses the `declare_pyqtprogram` directive to instruct WAF that this module produces an executable GUI application.

In line 9 the reader may find the **target** argument, which tells the name of the executable. Independent of the **py_program**, when installing, the **py_program** is installed using the **target** name into the `$PREFIX/bin/` directory.

If the python application needs a package to be installed, the reader may specify it using the **py_package** argument. This will be installed as a python package under `$PREFIX/lib/python$PYTHON_VERSION/site-packages/}` directory.

The **use** argument is to provide a list of dependencies this module has. Most importantly, these dependencies will be used also during tests.

For more details and information on wtools[105] and WAF[106], please refer to their manuals.

[105] https://www.eso.org/~eltmgr/documents/latest/wtools-docs/archive/html/index.html
[106] https://waf.io/book/

Entry Point Python Script

The entry point python script `paegui.py` is the one that creates a Qt GUI Application.

```
13 import sys
14 import click
15 import taurus
16 import taurus.cli.common
17 from taurus.qt.qtgui.application import TaurusApplication
18 from paegui import ApplicationWindow
```

`taurus.qt.qtgui.application.TaurusApplication` inherits from QApplication[107], and it is in charge of application initialization, configuration of default values, logging, etc.

In this example, three `taurus` utilities are used:

- **`taurus.cli.common` provides common option parsers to both custom application and Taurus** internal tools. It is based in `click` and in this case, the `log_level` option is used. It will ensure that the format that we use to ask for the log level is the same through all applications.

```
21 @click.command("paegui")
22 @taurus.cli.common.log_level
23 def paegui(log_level):
24     ...
```

The code above is the beginning of the `paegui` python method, which is the first one invoked after the python interpreter enters the `__main__` application entry point.

Since the parser for `log_level` is already taken care of, the developer is free to use it where they want. In this case the example forwards the `log_level` value to the method that sets the root logger's level. See line 34.

```
28     app = TaurusApplication(
29         app_name = 'Python Application Example GUI',
30         app_version = '0.0.1',
31         org_name = 'ESO',
32         org_domain = 'eso.org',
33         )
34     taurus.setLogLevel(getattr(taurus, log_level))
35     window = ApplicationWindow()
36
37     window.show()
38     sys.exit(app.exec_())
39
```

(continues on next page)

[107] https://doc.qt.io/qt-5/qapplication.html

(continued from previous page)

```
40
41 if __name__ == "__main__":
42     paegui()
```

The last section of the main method is listed above. The most important part is the declaration of `app`, as a `taurus.qt.qtgui.application.TaurusApplication` in line 28. This will create a context where Qt and Taurus objects can exists.

In line 35 the user interface is instantiated by the creation of a `ApplicationWindow` object, which is then assigned to the `window` variable. The `ApplicationWindow` class is part of the examples python package, and contains the implementation of the UI.

At this point, the application is ready to start, and to do so, we order it to show something in the screen. In this case, the GUI we have just loaded, using the `show()` method.

But control is still under the script's flow; developer must surrender control of the program, to Qt's event loop. A program or a python script is a sequential flow of instructions. There can be decision points where the program flow to one section of the code to another, but in general terms, is a linear flow.

When we surrender control of the program to Qt event pump, Qt will constantly look in the system for inputs like keystrokes and mouse movements and click, and execute the appropriate methods. We have in fact changed from a sequential program, to an event-based one. Events are the deciding factor of the programs next instruction set.

To pass the control to the event pump, we invoke `app.exec_()` method. This will leave the application in an endless loop governed by events.

See *Definitions* for more details on Asynchronous programming, Events, and Qt's Event Engine.

Design of the Graphical User Interface

The GUI for this example is a simple one. We will create four QDoubleSpinBox[108], one QTextEdit[109], one `CutWidgets.QeDartBoard` widget, and several QAction[110].

The QDoubleSpinBox[111] will be used to modify the values of the QeDartBoard[112]. The QTextEdit[113] will present an explanation on the UI. QAction[114] is used to abstract the UI and the implementation from tasks.

We will start the designer, by executing this command in a terminal:

[108] https://doc.qt.io/qt-6/qdoublespinbox.html
[109] https://doc.qt.io/qt-6/qtextedit.html
[110] https://doc.qt.io/qt-6/qaction.html
[111] https://doc.qt.io/qt-6/qdoublespinbox.html
[112] https://eltjenkins.hq.eso.org/job/ECOS/job/CUT/job/CUT-Build-Master-DevEnv-5/lastSuccessfulBuild/artifact/build/docs/html/classQeDartBoard.html
[113] https://doc.qt.io/qt-6/qtextedit.html
[114] https://doc.qt.io/qt-6/qaction.html

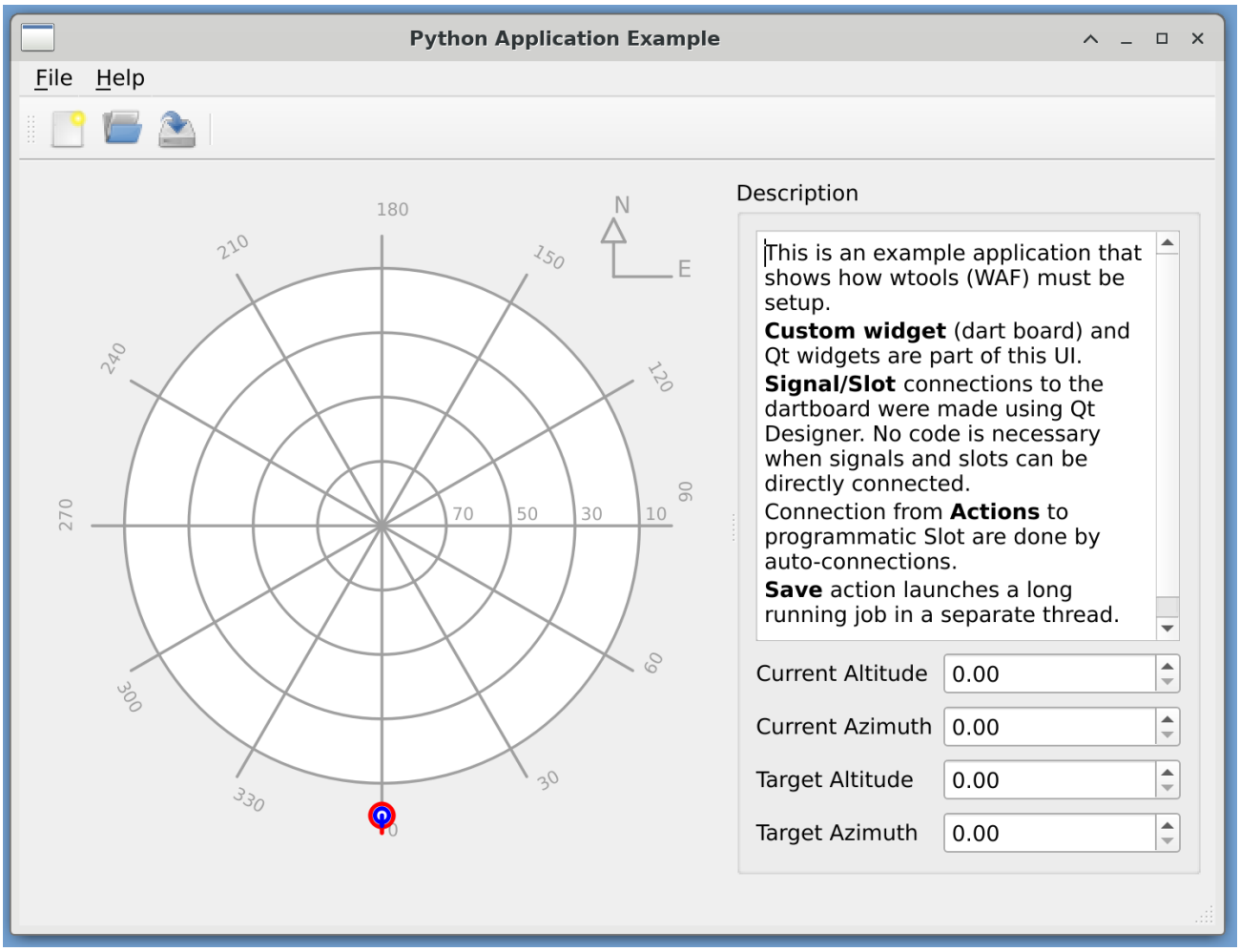


Fig. 48: Python Application Example - GUI design to achieve in this example.

```
$ taurus designer
```

The Qt designer will start, and automatically, will ask us what kind of UI this new file will be. The default selection is *Main Window*. Please click on the *Create* button in the lower right corner of this window: this will close this *New Form* dialog.

The designer will then create an empty UI that inherits from the QMainWindow[115] class, and we will have access to the whole set of designer tools.

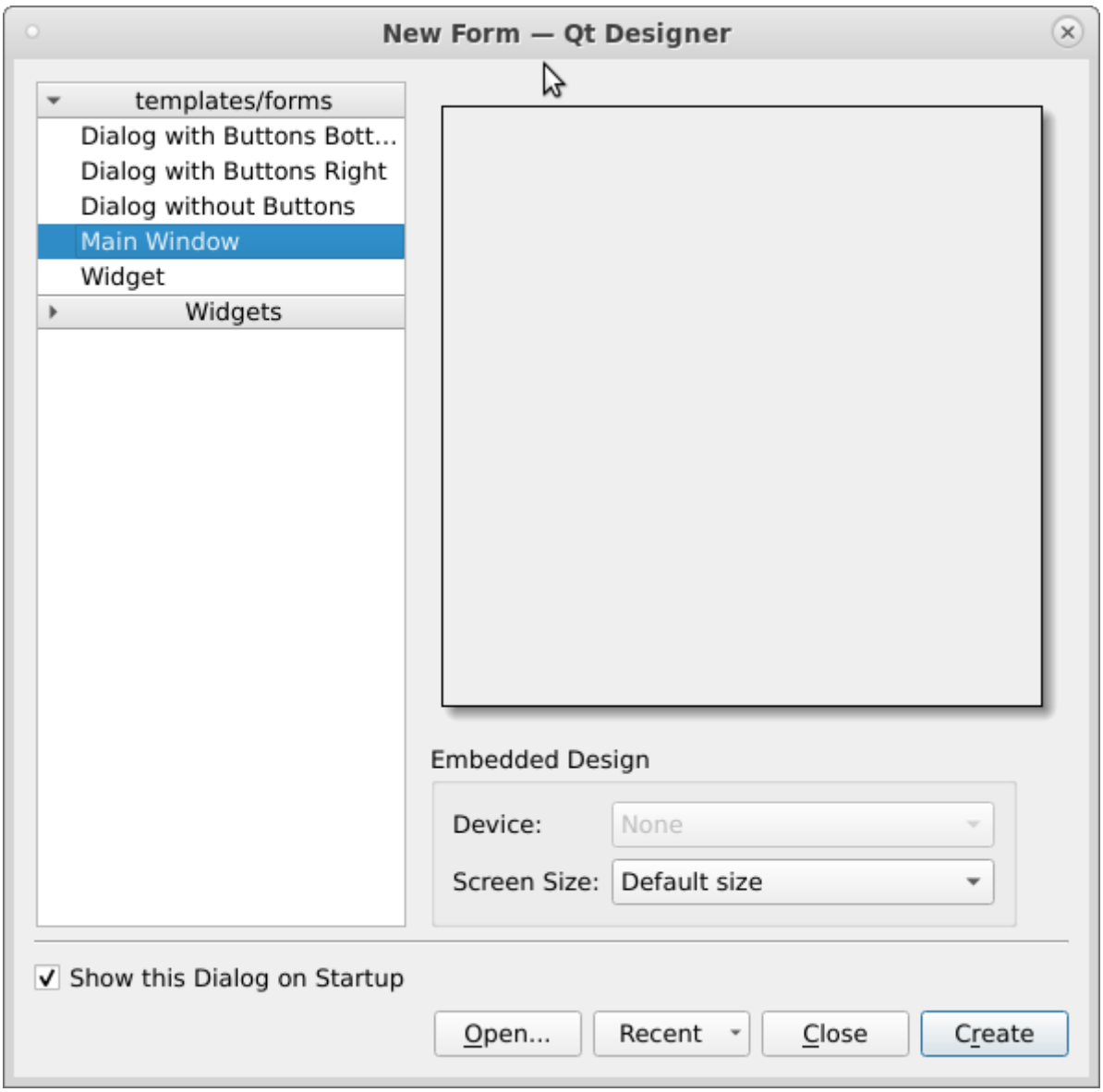


Fig. 49: Qt Designer - Selection of new Graphical User Interface template.

[115] https://doc.qt.io/qt-6/qmainwindow.html

Please save this file as `PythonApplication/src/python_application_example/mainwindow.ui`.

The most common GUI used for the presentation of an application is the QMainWindow[116]. This one has a MenuBar on top, a Toolbar, a central widget, and a StatusBar. Most desktop applications follow this setup.

1. On the left side of the Designer window, is the *Widget Box* Docking Window. We begin by dragging a *QeDartBoard* from the *Widget Box* Docking Window, to the MainWindow in the center of the Designer.
2. A new *QeDartBoard* Widget will appear. We can drag its corners to make it bigger. Make it a little bit bigger so we can see its features, but don't go overboard, leave some space free for the other widgets.
3. Drag a *Form Layout* from the *Widget Box* Docking Window to the MainWindow in the center of the Designer.
4. Drag a *Label* Widget from the *Widget Box* Docking Window, and drop it on top of the *Form Layout*. Notice that the *Form Layout* can contain other widgets, and the target's background for the new widget changes color when dragging.
5. Drag a *DoubleSpinBox* Widget from the *Widget Box* Docking Window, and drop it right of the *Label* Widget, inside of the *Form Layout*. Notice that the empty space on the right of the *Label* Widget is highlighted in red.
6. Repeat steps 4 and 5 three more times.

We have now static widgets, and a layout. We will stop at this point to explain some characteristics of positioning and size:

- You can resize the MainWindow. This will in fact change the default starting size of the application when its opened.
- If you resize the MainWindow, you will notice that the widgets do not move. If you make the MainWindow too small, some widgets will disappear until to resize it to a proper geometry.
- Resize the Form Layout. You will see that the widget it contains are properly redimensioned.

It is strongly suggested by the E-ELT Control GUI Guidelines, that User Interface scale and redimension themselves when resized. This is a basic feature of a well programmed GUI. Layout allows developers to achieve this.

Let's continue with the GUI.

1. From the *Widget Box* Docking Window, drag a *Vertical* Layout to the MainWindow. Resize it so it is bigger than the *Form Layout*.
2. From the *Widget Box* Docking Window, drag a *Text Edit* widget to the MainWindow, and drop it inside the *Vertical Layout*.
3. Drag the *Form Layout*, and place it inside the *Vertical Layout*.

[116] https://doc.qt.io/qt-6/qmainwindow.html

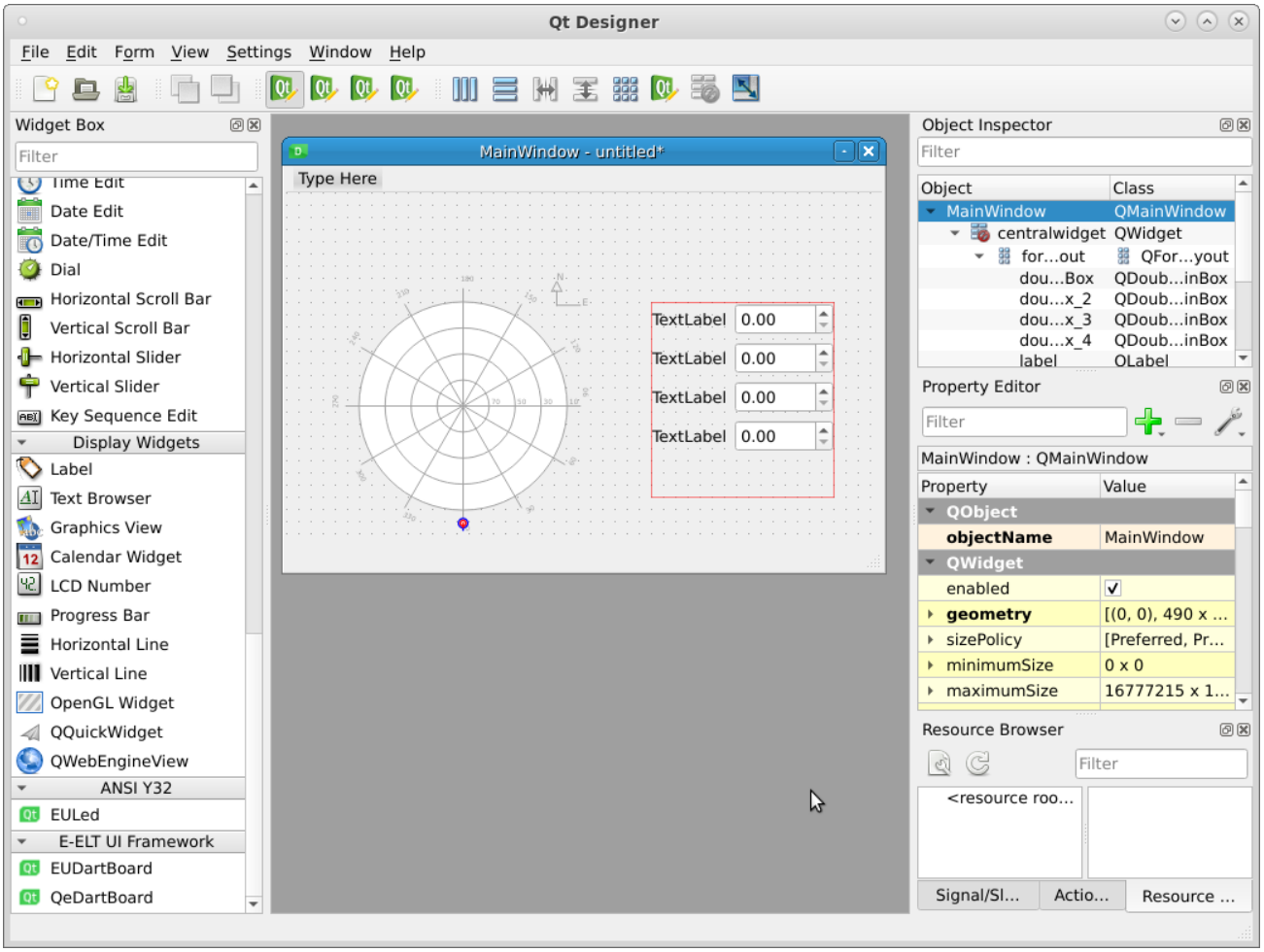


Fig. 50: Qt Designer - Progress so far.

4. Widget that can contain other widgets, have the Layout context menu. Right click on an empty space of the MainWindow, and a *context menu* will appear. Select the *Lay out* Entry. Select *Layout Horizontally* Entry.
5. On the Right hand side of the Designer, is the *Object inspector* Docking Window. This is a tree representation of the GUI. Select the *centralWidget* Entry.
6. Now, on the Right hand side of the Designer, under the *Property Editor* Docking Window, scroll down, until you see Layout. In there is the property *Layout Stretch*. Set it to “1,1”.

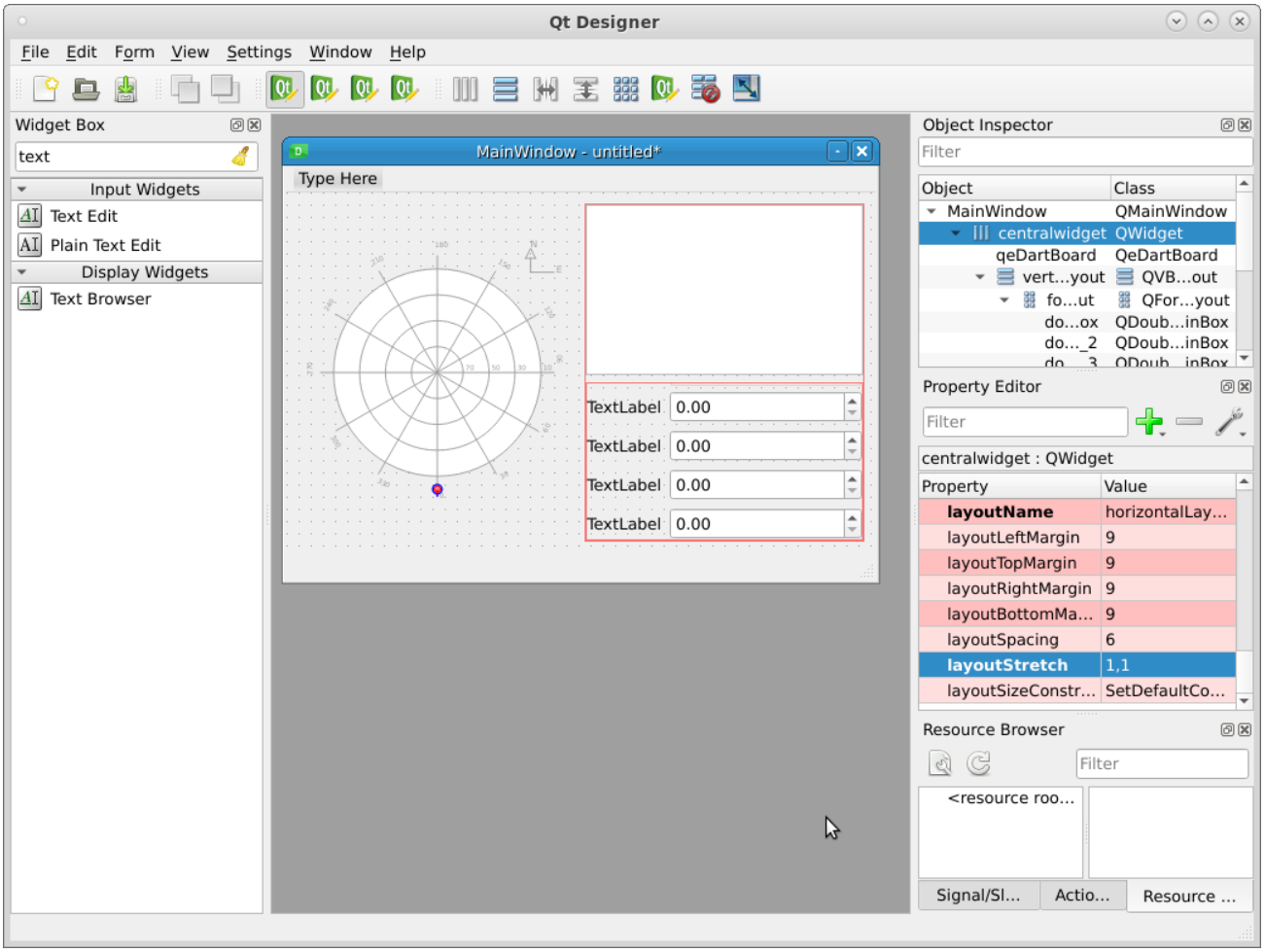


Fig. 51: Qt Designer - Progress so far.

We have now a completely elastic GUI. Go ahead and resize the MainWindow, and you will see that every component in it resizes nicely.

The `layoutStretch` property indicates in what proportion each component of the *Horizontal Layout* will stretch. In this case, there are two first order children (though the *Vertical Layout* has many children) on the MainWindow, which are the QeDartBoard[117] and the *Vertical Layout*, and each will

[117] https://eltjenkins.hq.eso.org/job/ECOS/job/CUT/job/CUT-Build-Master-DevEnv-5/lastSuccessfulBuild/artifact/build/

get 50% of the available space.

Tip: Elastic UI are achieved by use of Layouts. Do not fix the size of a widget. Instead use Layout and stretch factor to set the appropriate sizes.

At this point, only one thing is missing from the original image. Surrounding the whole *Vertical Layout*, there is a *Groupbox* with the Description title. From the *Widget Box* Docking Window in the left hand side of the Designer, drag a *Group Box*. Drop it in between the *QeDartBoard* and the *Vertical Layout*.

The *Group Box* widget will appear. Now we drag the *Text Edit* widget into the *Group Box* widget, and the *Form Layout* into the *Group Box* widget. An empty *Vertical Layout* will be left with no elements, and looking like a single red vertical line. Double click on the *Text Edit* widget and a new Dialog window will appear. Here, copy and paste this long text.

> This is an example application that shows how wtools (WAF) must be setup. Custom widget (dart board) and Qt widgets are part of this UI. Signal/Slot connections to the dart-board were made using Qt Designer. No code is necessary when signals and slots can be directly connected. Connection from Actions to programmatic Slot are done by auto-connections. Save action launches a long running job in a separate thread.

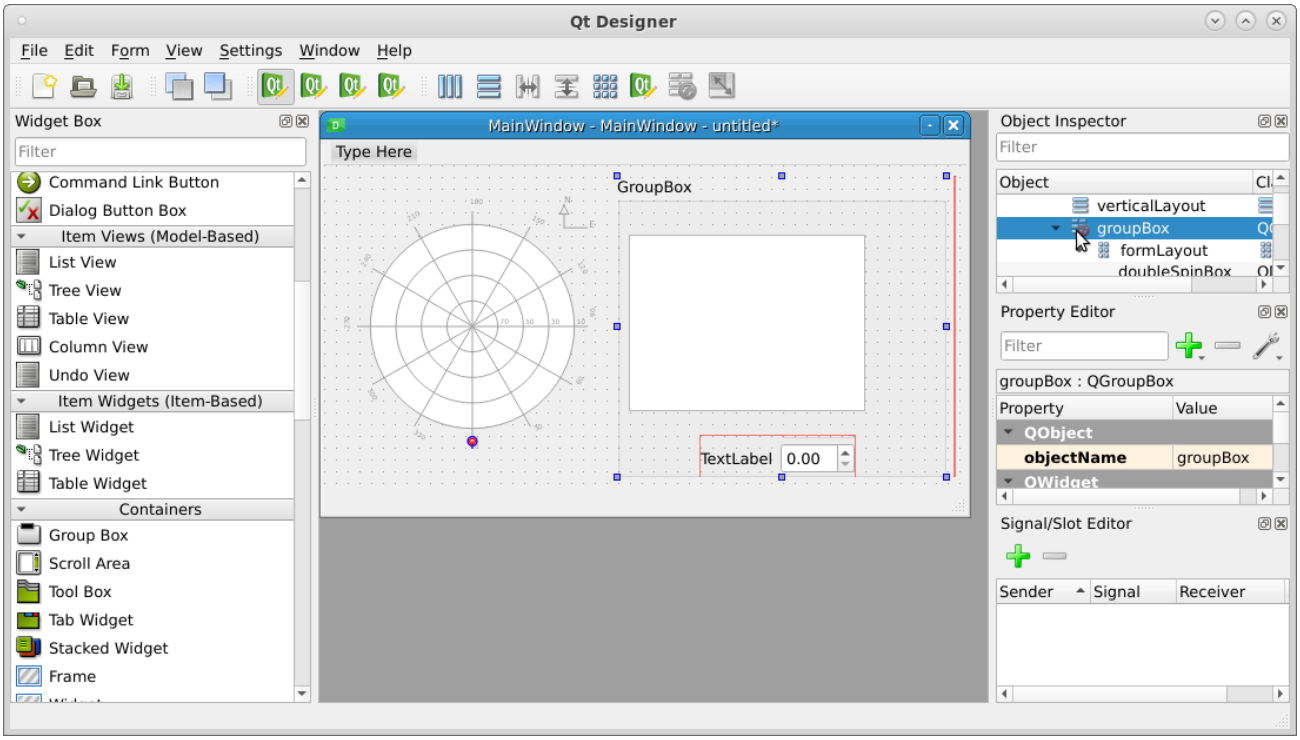


Fig. 52: Qt Designer - Progress so far.

Finally, use the *Object Inspector* Docking Window on the right hand side of the Designer. Select the

docs/html/classQeDartBoard.html

groupBox, right click on it, select *Lay out* Entry, and select *Vertical Layout* Entry. Since the *Group Box* is also a container, it can use layouts.

The UI is nearly finished. Update the *Labels* by double clicking on them, or changing its *text* properties.

Widgets Connections

But one thing is missing: connections: Qt UI files are capable of expressing in the UI files connection between elements of the UI. We will create four of them:

1. In the Designers *Toolbar*, click on *Edit Signal/Slot* button, or press `F4`. You will no longer be capable of adding widgets, but you can drag one widget and drop unto another to form a connection.
2. Drag from the first *Double Spin Box*, and drop on top of the *QeDartBoard*.

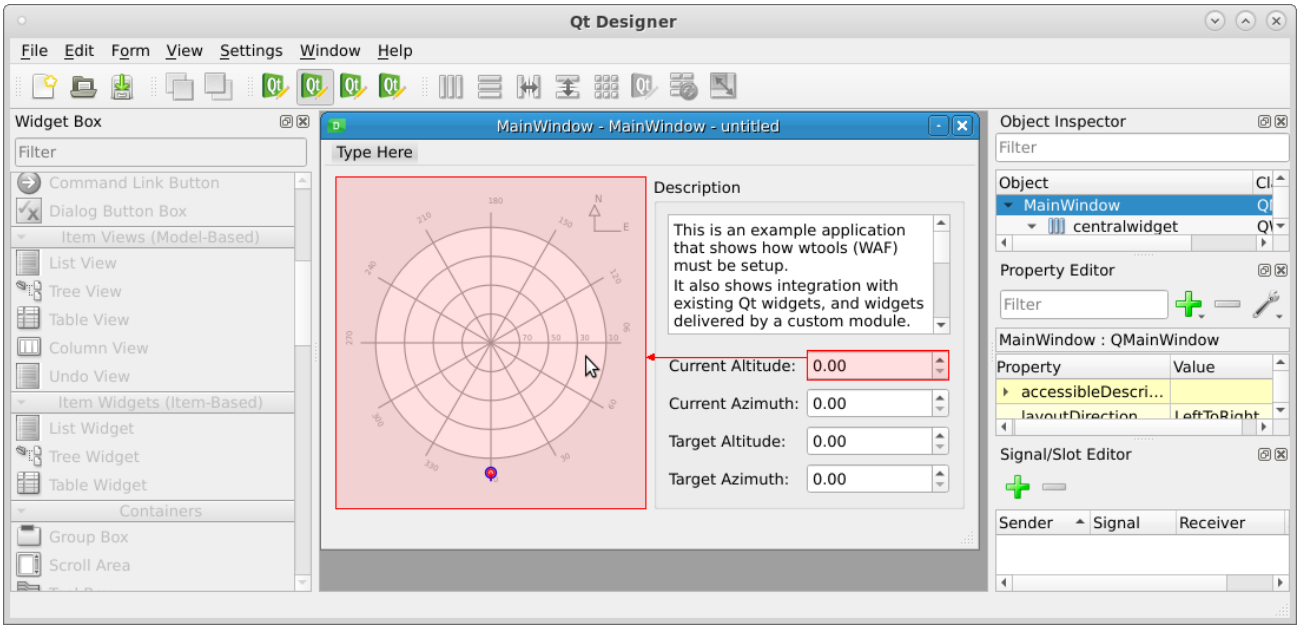


Fig. 53: Qt Designer - Drag and Drop operation using Edit Signal/Slot Designer functionality.

3. When you drop, a new *Configure Connection* dialog will open. It will automatically read from the source widget its signals, and from the destination widget its slots. Now the developer must select matching signatures. Please selected in the left side `valueChanged(double)`, and on the right side `setCurrentAlt(double)`. Press the *Ok* button.
4. Repeat the same procedure for the remaining three *Double Spin Boxes*, but using as slot: `setCurrentAz(double)`, then `setTargetAlt(double)` and for the fourth *Double Spin Box*, **setTargetAz(double)**.

The developer may test this right away inside the Designer. The Designer plugin we develop in the

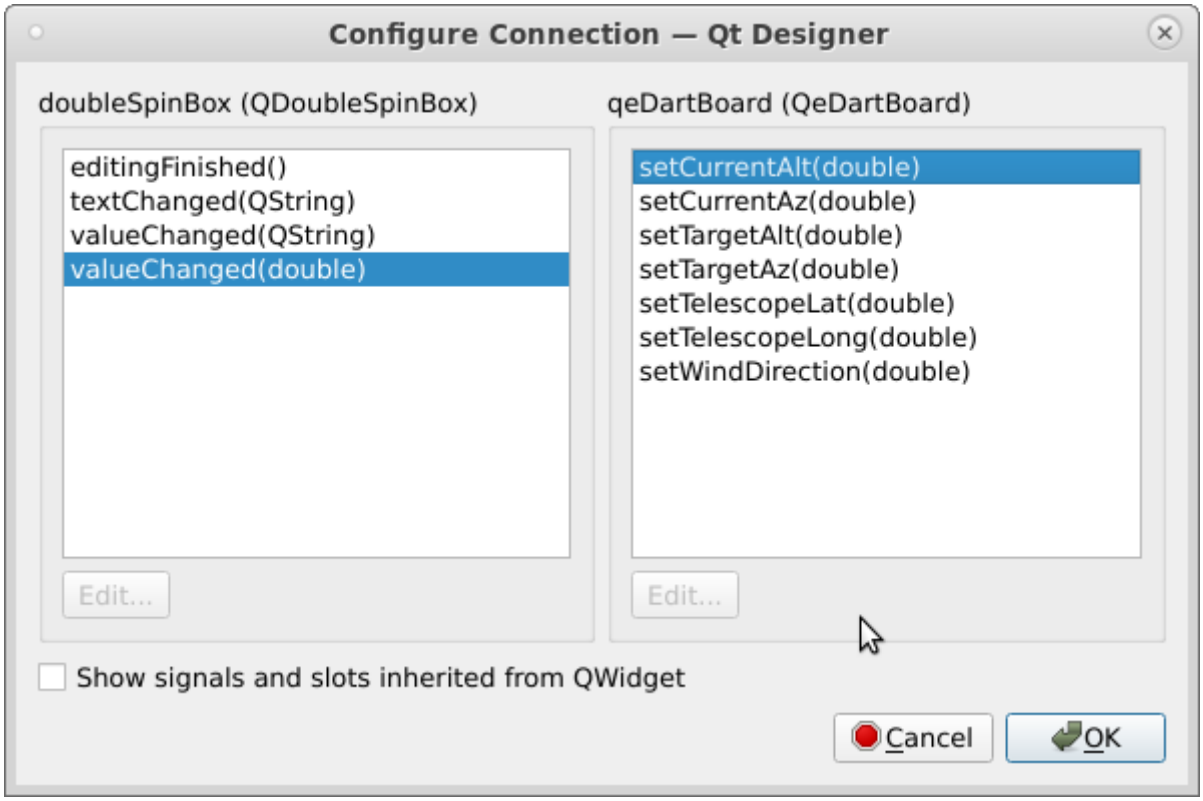


Fig. 54: Qt Designer - Configure Connection dialog window

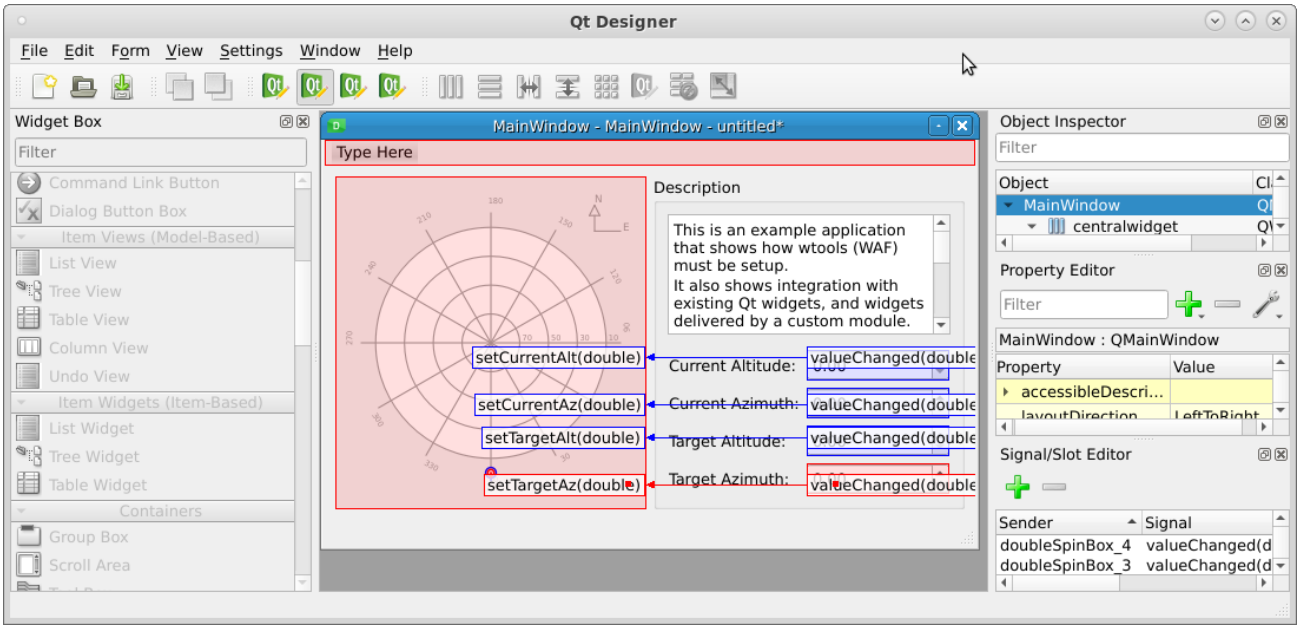


Fig. 55: Qt Designer - All four connections are made.

widget_example[118], and any Qt Plugin that is rendered in the Designer is a fully capable one.

In the Designer Toolbar press the *Edit Widgets* button, or press `F3`. Now change the values in the *Double Spin Boxes*, and play with the values in the *Double Spin Boxes* and see how the target (red) and current (blue) indicators in the *Dart Board* move.

Actions

For this the developer needs to use the *Action Editor* Tab. We recommend to drag this from its current position below the Property Editor Tab, and move it below the *MainWindow* Form. This will give us extra space. If the *Action Editor* Tab is not visible, you can make it visible using the Designer *MenuBar* → *View* → *Action Editor* Entry.

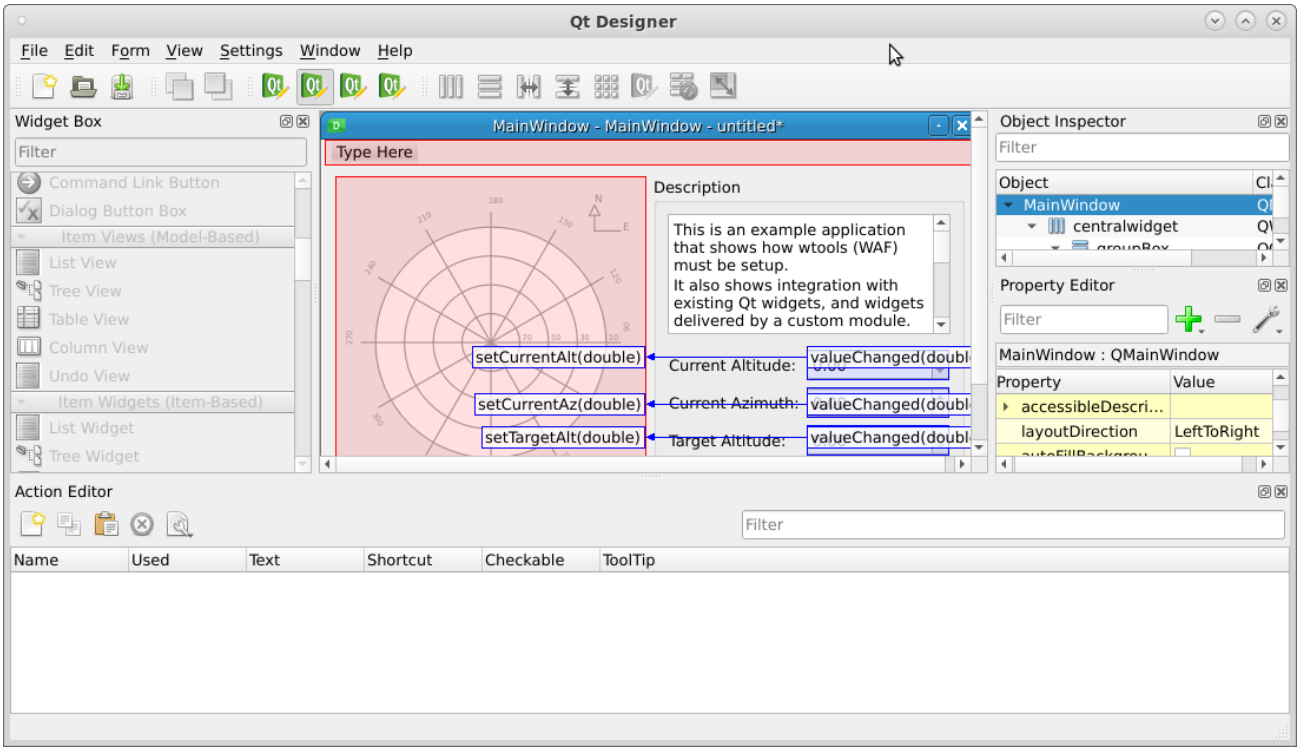


Fig. 56: Qt Designer - Action Editor placed below the Main Window Form will give the developer extra space.

Actions

Are non-visible objects that express a recurring “action” the GUI must perform. For example, when programming a text editor, its most recurring actions could be to *Open a file*, to *Save a file*, or to *Exit* from the application.

When designing a client for a server, *connect* to the server, *configure connection* and *disconnect* are recurring operations.

[118] https://www.eso.org/~ahoffsta/docs/latest/widget_development/html/

It is advisable that while designing an application, the recurring actions of the application are identified, and defined through this tab in the Qt Designer.

Now the developer may proceed to create actions:

1. On the toolbar included in the *Action Editor* Tab, press the *New* button. A *New Action* dialog will pop up. Enter the following information exactly as in the entries below (the & character included):
   - **Text**: E&xit
   - **Object Name**: actionExit
   - **ToolTip**: Exit form the application
   - **Icon Theme**: application-exit
   - Leave Checkable and Icon with its default values.
   - **Shortcut**: Click on it, and press `Control + x`.
2. Click on the *Ok* button.

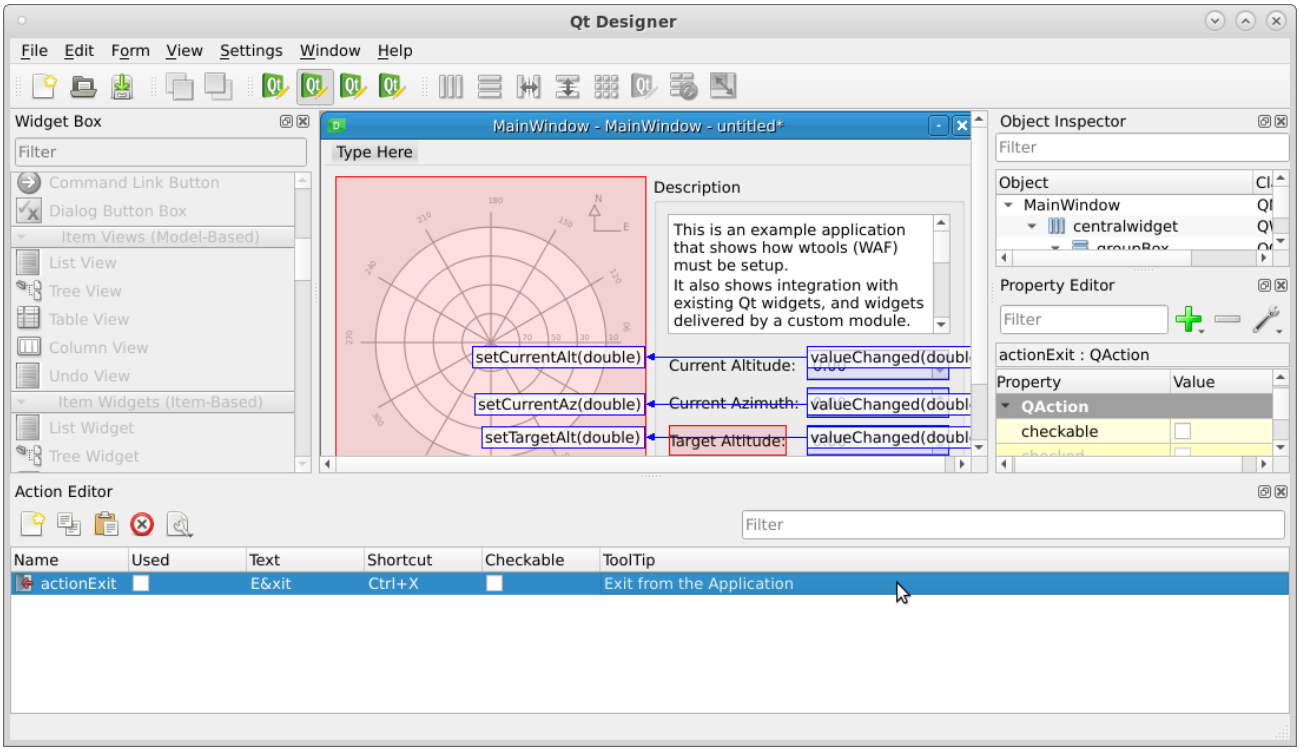


Fig. 57: Qt Designer - A new Action has been created: actionExit

In these Action, we configure the presentation and how the user access them. Its implementation is still left for the developer, as we will see in the next section, but presentation and access, is determined in the UI file, using the Designer.

Please create five more actions:

Text	Object Name	Tooltip	Icon Theme	Shortcut
&New	actionNew	Creates a new file	document-new	Control + N
&Open	actionOpen	Opens a file from the file system	document-open	Control + O
&Save	actionSave	Saves the buffer to the file system	document-open	Control + S
&Manual	actionManual	Opens the user manual	system-help	Control + H
&What's This?	actionWhatsThis	Click on anything to learn more	help	

At this moment we are ready to create the Main Menu: On the Main Window Form, where the Menu should be, there is a text that reads *Type Here*. Please click on it, type `&File` and press `Enter`. A Menu will appear, and the *Type Here* text will move right. Please click on the *Type Here* text again, type `&Help`, and press `Enter`.

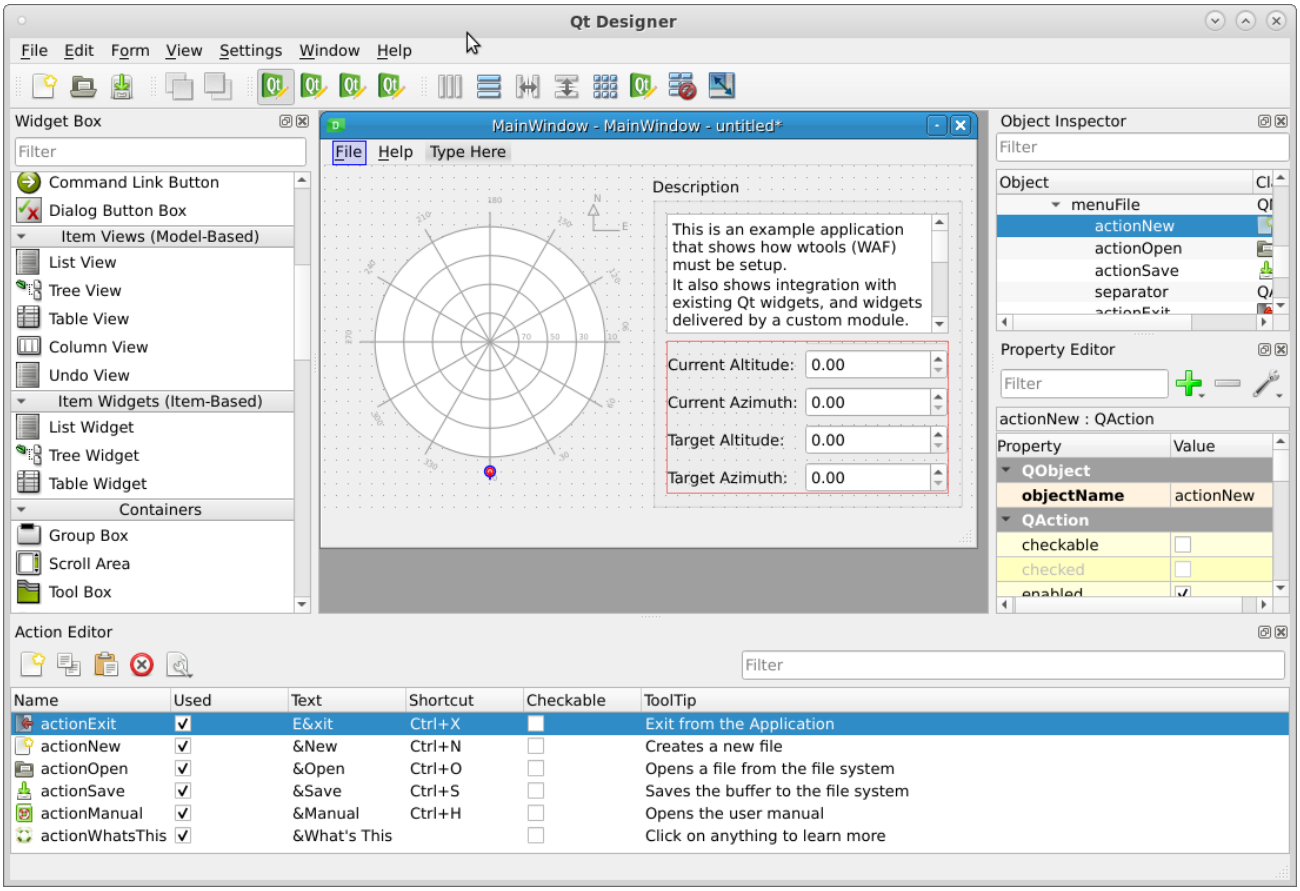


Fig. 58: Qt Designer - All six actions and two menus have been created.

Now the developer may populate the menus:

1. From the *Action Editor* Tab drag the *actionNew* Action to the *File* Menu. An empty *File* Menu will pop up. Drop the *actionNew* in this empty menu.
2. Repeat for *actionOpen* and *actionSave*.
3. At this point click on the *File* menu, and double click on the *Add Separator* entry.
4. Now drag the *actionExit* action to the *File* menu, but be sure to drop it after the separator.

Repeat the drag and drop operation for the *actionManual* and *actionWhatsThis*, but drop them instead into the *Help* Menu.

Warning: The developer must learn the difference between accelerators and shortcuts.

Shortcuts
Are keyboard combinations, usually involving two keystrokes, one of them a modifier key (`Control` or `Shift`), and are accessible from anywhere in the application while it has focus.

Accelerators
Are only valid while the user navigates a menu, for example the *File* menu. In the text of the *File* menu, the original entry was `&File`, which means that `Alt + F` will open that menu. Then instead of clicking the *New* entry (which original text is `&New`), the user can press in the keyboard the letter `n`.

Action can be placed in the Menu, or in the Toolbar.

1. On the *Object Inspector* Tab, go to the *Main Window* QMainWindow entry, right click on it, and select *Add Toolbar* entry. This will create a new toolbar for the UI.
2. From the *Action Editor* Tab, drag the *New*, *Open* and *Save* Actions.
3. On the new Toolbar, right click and select the *Append Separator* entry.
4. Now, from the *Action Editor* Tab, drag the *What's This* Action.

This reuses the definitions from the Menu, and makes them accessible from the toolbar. Actions are highly reusable, and can also be triggered, enabled and disabled by code. Disabling an Action will make them unavailable both in the menu, and the toolbar.

Important: Every piece of the UI we have defined, can be also defined through code. We **strongly recommend to use UI files instead of code**. UIs defined through code are hard to maintain and easily grow into thousands line file.

Note: All elements used in the designer, end up in an XML formatted file, the UI file. This file will be used by the `pyside6-uic` –Qt UI Compiler–, which will be automatically invoked by WAF, and generated the Python version of the same UI file.

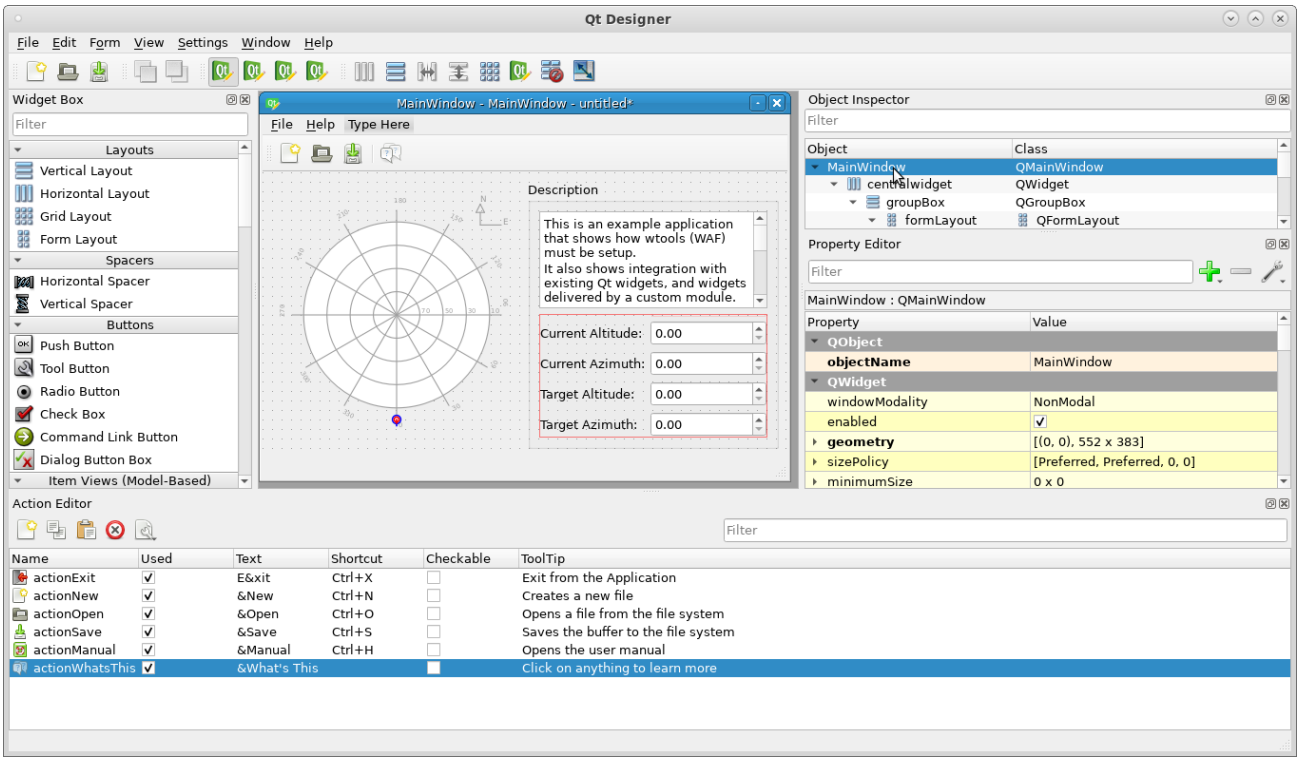


Fig. 59: Qt Designer - New toolbar and buttons in place.

Widgets, layout, properties, actions and connections, all are saved into XML, and then when WAF is executed, a python version of it is automatically generated. **This is intentionally invisible to the developer** as in no case, the developer should use this file directly.

Implementation of the User Interface

In this section, the developer will use the UI designed in the previous section, and implement basic functionality for them. Every function needed will be implemented in the `PythonApplication/src/paegui/applicationwindow.py` file.

All the entry points for missing functionality are already defined through the Actions in the UI file, we just need to connect the missing pieces. We will begin explaining the imports.

```
10 import time
11
12 from taurus.external.qt.QtCore import QObject
13 from taurus.external.qt.QtCore import Slot
14 from taurus.external.qt.QtCore import Signal
15 from taurus.external.qt.QtCore import QMetaObject
16 from taurus.external.qt.QtWidgets import QApplication
```

(continues on next page)

(continued from previous page)

```
17 from taurus.external.qt.QtWidgets import QMainWindow
18 from taurus.external.qt.QtWidgets import QWhatsThis
19
20 from taurus.core.util.log import Logger, TraceIt
21 from elt.cut.task.task import Task
22 from CutColor import QePalettesModel
23
24 from python_application_example.mainwindow import Ui_MainWindow  # WAF will␣
   ↪automatically generated this
25 from python_application_example.applicationconfiguration import␣
   ↪ApplicationConfiguration
```

Most classes in the Qt library inherit from QObject[119]. This is true also for widgets. QObject is the base class that provides Signal[120]/Slot[121] connection capabilities to the Qt Toolkit. We will use them plenty through the design and implementation of GUIs. If you are not familiar with them, we suggest you to read:

- https://doc.qt.io/qt-6/object.html
- https://doc.qt.io/qt-6/signalsandslots.html
- https://wiki.qt.io/Qt_for_Python_Signals_and_Slots

The QMainWindow[122] import is present, as this class is the implementation of a QMainWindow. At the beginning of the design section, a Main Window was selected. Therefore, the implementation class must match the top level UI element.

The `taurus.Logger` class is from the `taurus` module and it allows to enhance a class with logging capabilities.

Finally, the `Ui_MainWindow` import is the python code generated by the **`pyside6-uic`** compiler. The compiler is automatically invoked by WAF, so no extra instructions are to be given. It is important that the UI file is inside of the Python Package name `paegui`, as from this directory is where WAF will look for any UI file, and automatically compile them.

Important: To determine both the name of the **Python Module** and the **Class Name**, the name of the first element of the UI file is used:

The **Python module** name will be lowercased name of the top level element of the UI file (in our case `mainwindow.py`).

The resulting **Class Name** will be `Ui_<name of top level element>`, in this case `Ui_MainWindow`.

You can see the first element name in the Designer; *Object Inspector* Tab; the first element. You can

[119] https://doc.qt.io/qtforpython/PySide6/QtCore/QObject.html
[120] https://doc.qt.io/qtforpython/PySide6/QtCore/Signal.html
[121] https://doc.qt.io/qtforpython/PySide6/QtCore/Slot.html
[122] https://doc.qt.io/qtforpython/PySide6/QtWidgets/QMainWindow.html

also change it in the same place.

The resulting import show how the **Python Module** and the **Class Name** are used:

```
from python_application_example.mainwindow import Ui_MainWindow
```

We will now proceed to examine the Class definition:

```
28 class ApplicationWindow(QMainWindow, Logger):
29     '''
30     Implementation of the mainwindow.ui file.
31 
32     Since the UI file indicates that its root is a QMainWindow, then
33     this class should also inherit from it.
34     We should also call explicitly its parent constructors.
35 
36     The implementation for this class also includes slots for
37     actions, and management of the closeEvent.
38     '''
39 
40     ###############################################################################
41     #                 Initialization and QMainWindow Methods                      #
42     ###############################################################################
43 
44     def __init__(self, configuration: ApplicationConfiguration):
45         QMainWindow.__init__(self)
46         Logger.__init__(self, name=QApplication.instance().applicationName())
47         # Construction of UI
48         self.ui = Ui_MainWindow()
49         self.ui.setupUi(self)
50         self._init_gui()
51         self._configuration = configuration
```

Line 28 declares a new class called *ApplicationWindow*, which inherits from QMainWindow[123]. QMainWindow[124] already inherits from QObject[125], so we can use signal and slots from this class definition.

The *__init__()* manually calls both its parent classes *__init__()* method. This is very important, as Python does not make implicit calls to these.

In line 46, the `taurus.Logger` parent class is initialized, but we use as name keyworded argument, `QApplication.instance().applicationName()`. `QApplication.instance()` is a static reference to the QApplication or `taurus.qt.qtgui.application.TaurusApplication` of this process. Qt forbids

[123] https://doc.qt.io/qtforpython/PySide6/QtWidgets/QMainWindow.html
[124] https://doc.qt.io/qtforpython/PySide6/QtWidgets/QMainWindow.html
[125] https://doc.qt.io/qtforpython/PySide6/QtCore/QObject.html

running two or more QApplications in the same process, and in Python, they offer a quick manner to obtain a reference to the QApplication object. Developers may use the `QApplication.instance()` reference to access the running QApplication.

In line 48 is where we create an instance of the UI definition we have in the mainwindow.ui file. It is assigned to `self.ui`. Then, in the next line, we invoke its *setupUi()* method, passing as argument the `self` reference to the QMainWindow object being initialized. The *setupUi()* method in the *Ui_MainWindow()* class will create every element as defined in the Designer for us. It is actually Python code that was translated from the XML. When the method finishes, every widget, action and layout will be available for the developer at `self.ui`.

Important: `self.ui` is very important for UI development using Qt. We suggest the reader to familiarize themselves with this manner of accessing the UI declared in the UI file. Developers should never copy and archive to repositories the result of `pyside6-uic` compiler. Instead, they should archive the UI file, and WAF automatically will produce an updated version of the UI.

We also encourage to use an editor capable of Python introspection/code completion. This will make the navigation of `self.ui` much more easier.

The next section of the code has many entries like this one:

```
104     @Slot()
105     def on_actionNew_triggered(self):
106         '''
107         Slot that auto-connects to the actionExit.
108         actionNew is not declared in the code, but in the mainwindow.ui.
109         See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-
    ↪dialogs-with-auto-connect
110         '''
111         self.info('actionNew triggered')
```

Each of these method defined the implementation of an action. The developer may notice a pattern in the name of the method: It is a composite of an existing Action name, and a signal it has. Qt will automatically recognize this pattern, and connect that object's signal, to this slot.

Auto Connection explained

The *on_actionNew_triggered()* method has a special name. As the comment mentions, actionNew is defined in the UI, and make use of auto connection, which is a convention based manner to save lines of code.

The syntax is:

```
on_<ui_object_name>_<signal>(self, <same_arguments_as_signal>)
```

The last thing *setupUi()* method does is to execute the *QMetaObject.connectSlotsByName()*, a

static method that will:

- For every Slot declared in our implementation class (the ApplicationWindow class):
- Check if the name starts with `on_`.
- Check if the next section matches the name of an object in the UI definition.
- Check if this object in the UI definition has a Signal[126] called *triggered*.
- If all three conditions are a true, then it will automatically connect the object's Signal, against the implementation's Slot.

The only entry that is different is the `closeEvent()` method.

```
65      def closeEvent(self, event):
66          '''
67          Not to be confused with actionExit, this method is special. Instead
68          of using signals to know when an application is closed, we can
69          override the closeEvent method from the QApplication class, and
70          determine here what will happen when the window is closed.
71          In this case, we are just forwarding the event to its parent class.
72          This is useful when disconnection from server or prevention of
73          current work needs to be implemented.
74
75          :param event: Event received from Qt event pump
76          :type event: QtCore.QEvent
77
78          '''
79          self.info('Application shutting down...')
80          QMainWindow.closeEvent(self, event)
```

Let's remember that Qt is an event based library. It detect and translate many input performed by the user and internal system outcomes into Events. Example of these are mouse movement, click, keystrokes, closing minimizing, maximizing the application, among many more. When a user click on the close button of an application, this is translated into an event.

Each UI element in Qt has one main `handleEvent()` method, and many specialized Event methods: , like `closeEvent()`:

- `handleEvent()` is the main entry point for any event. The `QApplication.handleEvent()` method will process this event, classify it accordingly, and then call a specialized method.
- `closeEvent()` is the specialized method when a window or dialog is requested to close. In the case of our particular implementation, closing the application main window should exit the program. Qt does this by default, so we just call parent implementation using `QMainWindow.closeEvent()`.

[126] https://doc.qt.io/qtforpython/PySide6/QtCore/Signal.html

At this moment, we will not concern ourselves with other kinds of methods.

At this point, the implementation is ready, and will log every action that is triggered.

Long Running Jobs

Qt runs in one thread. The main thread of the process is used by the event pump to detect and conduct the necessary operations, and drawing the GUI. For example, resizing the application will create many **resizeEvent**, which will request redrawing the UI to the appropriate dimensions.

But for this to happen, the main thread needs to be free. The developer may implement short and quick methods, and connect them using signal/slot mechanism. Qt will insert in between signal/slot execution many of its own events, keeping the UI smooth. But long running jobs should not be conducted in the main thread.

For sake of simplicity, a long running job can be emulated with a *time.sleep()* call.

Warning: An example is given below. This will cause the GUI to freeze for 5 seconds, as no other event gets executed in between, including input detection, and GUI drawing instructions.

```
115     @Slot()
116     def on_actionSave_triggered(self):
117         """
118         Slot that auto-connects to the actionSave.
119         actionSave is not declared in the code, but in the mainwindow.ui.
120         See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-
    ↪dialogs-with-auto-connect
121         """
122         self.info('actionSave triggered')
123         time.sleep(5)
124         self.info('actionSave finished')
```

In this example application, we include a solution for this. In line 115, the `actionSave()` slot is queuing a long running job using the `elt.cut.task` module. This class manages a threadpool used for long running operations, pushing tasks into it. It also allows you connect a slot to receive the results of the job.

```
122     @TraceIt()
123     @Slot()
124     def on_actionSave_triggered(self):
125         """
126         Slot that auto-connects to the actionSave.
127         actionSave is not declared in the code, but in the mainwindow.ui.
128         See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-
```

(continues on next page)

(continued from previous page)

```
↪dialogs-with-auto-connect
        '''
        self.info('actionSave triggered')
        self.info('actionSave launching new Task')
        task = Task(self.save_job)
        task.signals.result.connect(self.save_job_slot)
        task.start()
        self.info('actionSave finished')
```

Both methods `save_job()` and `save_job_slot()` are defined in the code shown below. `save_job()` is a method, and it is doing the long running job (in this case, sleeping for 5 seconds). When finished, the Taurus Manager will automatically invoke the callback `save_job_slot()`, passing as argument the return value of the `save_job()` method.

```
    def save_job(self):
        self.info('save_job sleeping for 5')
        time.sleep(5)
        return 'Slept for 5 seconds'

    def save_job_slot(self, arg):
        self.info('save_job_slot reporting %s' % (arg, ) )
```

There are also signals for progress, errors and finished conditions, to create more complete behaviors.

Now the reader may run the PaeGui using this command:

```
$ cutPaeGui
```

Logs from Taurus will appear in the console and the application will start. Please exercise the different Double Spin Boxes, and see how connections works with the Dart Board. Also, if you press several times the save button (or its menu entry), you will see how each job request will get executed in its own separate thread:

```
$ cutPaeGui
MainThread     INFO     2024-10-29 13:50:31,914 TaurusRootLogger: Using PySide6 (v6.
↪7.2 with Qt 6.7.2 and Python 3.12.6)
MainThread     INFO     2024-10-29 13:50:32,085 taurus.qt.qtgui.icon.icon: Setting␣
↪breeze icon theme (from /usr/share/icons)
MainThread     INFO     2024-10-29 13:50:32,096 TaurusRootLogger: Plugin "taurus_
↪pyqtgraph" lazy-loaded as "taurus.qt.qtgui.tpg"
MainThread     INFO     2024-10-29 13:51:00,771 Python Application Example GUI:␣
↪Application shutting down...
MainThread     INFO     2024-10-29 13:51:00,785 TaurusRootLogger:
```

Taurus logs have the following syntax: Thread, Level, Timestamp, Logger Name, message.

If you need lower level logs, please start the application with the following command:

```
$ cutPaeGui --taurus-log-level Debug
```

Help is always present when using the `taurus.qt.qtgui.application.TaurusApplication` class:

```
$ cutPaeGui --help
MainThread     INFO     2024-10-29 14:17:53,358 TaurusRootLogger: Using PySide6 (v6.
↪7.2 with Qt 6.7.2 and Python 3.12.6)
MainThread     INFO     2024-10-29 14:17:53,502 taurus.qt.qtgui.icon.icon: Setting␣
↪breeze icon theme (from /usr/share/icons)
MainThread     INFO     2024-10-29 14:17:53,512 TaurusRootLogger: Plugin "taurus_
↪pyqtgraph" lazy-loaded as "taurus.qt.qtgui.tpg"
Usage: cutPaeGui [OPTIONS]

  PythonApplicationExample GUI (or paegui), is part of a tutorial intended to
  introduce how to developed a simple GUI application for the E-ELT software

Options:
  --log-level [Critical|Error|Warning|Info|Debug|Trace]
                                  Show only logs with priority LEVEL or above
                                  [default: Info]
  -m, --memory-oldb               Uses in memory OLDB, disregarding remote
                                  OLDB.
  --help                          Show this message and exit.
```

If the parser was configured as in the start of the application, the taurus default options and the application options will all be handled by the same parser.

At this point, please save your work.

Frequently Asked Question

I have connected a signal and slot, but nothing is happening. What is missing?

Make sure you are using the `@Slot()` decorator for the Slot.

The Object where the Slot is located must inherit from QObject. Make sure that at some point in the inheritance tree this happens. You can add it to the particular class definition, but please make sure first any of its ancestors do not have them.

What causes the GUI to freeze?

A slot, or method in your GUI is using too much time in the main thread. You should keep to a minimum the amount of time slots and main thread methods can use.

Qt runs all input detection and rendering instructions in the main thread. This is shared with the signal/slot mechanism. If you have a task that takes too much time to complete, you should move it to a separate thread. This tutorial includes an example how to do this.

4.3 Widget Library Tutorial

Introduction

Warning: If you have not programmed a graphical application before, do not start with this document. Creating widgets is not the starting point to learn how to write graphical application.

If you need a starting point, we recommend the *Python Application Tutorial*.

Widgets are the control and display elements of graphical application. Most of the visible elements are widgets. They are buttons, combo boxes, line edit, tables that make up the GUI. They have drawing instructions, and interaction instructions.

Widget can also contain another widgets: these are called container widgets, and examples of them are frames, and group boxes. In these widget, you can put other widgets, even another container. This allows the developer to create a structure or organization for the GUI.

In this document, we will see how to create new widgets, what different artifacts should produce, and how to integrate them into a WAF project.

Though this tutorial will start gradually, we present you the final structure of a widget library:

```
widget_library/
├── bindings
│   ├── src
│   │   ├── bindings.h
│   │   └── bindings.xml
│   └── wscript
├── plugin
│   ├── src
│   │   ├── include
│   │   │   └── cut
│   │   │       └── examples
│   │   │           └── widgets
│   │   │               └── plugin
```

(continues on next page)

(continued from previous page)

```
    │   │   │                   ├── QeExampleDmsLabelPlugin.hpp
    │   │   │                   └── QeExamplesWidgetCollection.hpp
    │   │   ├── QeExampleDmsLabelPlugin.cpp
    │   │   └── QeExamplesWidgetCollection.cpp
    │   └── wscript
    ├── showcase
    │   ├── src
    │   │   ├── include
    │   │   │   └── cut
    │   │   │       └── examples
    │   │   │           └── widgets
    │   │   │               └── showcase
    │   │   │                   ├── mainwindow.hpp
    │   │   │                   └── mainwindow.ui
    │   │   ├── main.cpp
    │   │   └── mainwindow.cpp
    │   └── wscript
    ├── showcasepy
    │   ├── src
    │   │   ├── cut
    │   │   │   └── examples
    │   │   │       └── widgets
    │   │   │           └── showcasepy
    │   │   │               ├── __init__.py
    │   │   │               ├── mainappwindow.py
    │   │   │               └── mainwindow.ui
    │   │   └── showcasepy.py
    │   └── wscript
    ├── taurus
    │   ├── src
    │   │   └── cut
    │   │       └── examples
    │   │           └── widgets
    │   │               └── taurus
    │   │                   ├── __init__.py
    │   │                   ├── taurusexampledmslabel.py
    │   │                   └── taurusexamplesimpledmslabel.py
    │   └── wscript
    ├── widgets
    │   ├── src
    │   │   ├── include
    │   │   │   ├── cut
    │   │   │   │   └── examples
```

(continues on next page)

(continued from previous page)

```
│   │   │   │           └── widgets
│   │   │   │               └── widgets
│   │   │   │                   └── QeExampleDmsLabel.hpp
│   │   │   └── QeExamplesWidgets.h
│   │   └── QeExampleDmsLabel.cpp
│   └── wscript
└── wscript
```

The project shall be called “libraryname”. It is also how the module for this library is called. The project will produce the following artefacts:

- A c++ shared library, with the widgets.
- A c++ shared library that is a plugin for the Qt Designer.
- A c shared library, that is a CPython module (widgets bound to python)
- A cpp application, that shows how to use the widgets
- A python application, that shows how to use the widgets.
- A python module that provides declarative databinding widgets using Taurus.

Tip: When developing widgets, you can forget for a while the plugin. You can use Widget Promotion[127] in the Qt Designer in the meantime. But in order to allow other developers to design Uis using the widgets, you need to provide a plugin for the Designer.

Warning: Please do not bundle the widget and the plugin together in one WAF module.

Tip: If you have already read this document, we recommend the use of CUT templates to produce a skeleton widget library:

```
(base) [eeltdev@eltdev ~]$ git clone https://gitlab.eso.org/ecos/cut.git
(base) [eeltdev@eltdev tmp]$ cookiecutter ../repos/cut/templates/widget_library

project_name [CUT Widget Library]:
project_name_slug [cut-widget-library]:
project_version [0.0.1-dev]:
widget_name [CutDartBoard]:
object_name [cutDartBoard]:
lib_name [CutWidgets]:
```

(continues on next page)

[127] https://doc.qt.io/qt-5/designer-using-custom-widgets.html

(continued from previous page)

```
pkg_name [cutwidgets]:
showcase_name [CutWidgetsShowcase]:

(base) [eeltdev@eltdev ~]$ cd cut-widget-library
(base) [eeltdev@eltdev ~]$ waf configure build install
(base) [eeltdev@eltdev ~]$ CutWidgetsShowcase
(base) [eeltdev@eltdev ~]$ CutWidgetsShowcasePy
```

Kind of Widgets

Before we start developing new widgets, the reader needs to determine what is actually needed. We strongly suggest to take a look into Qt widget library and Control UI Toolkit widget library before starting to develop a new widget. Sometimes with the correct configuration, the necessary functionality can be achieved.

If this is not the case, then the next step would be: what kind of widget is needed?

- Extending an existing one: A similar widget exists, but does not provide a way to achieve what is needed, or requires too much boilerplate code. For example, a Label that presents an angle in DMS units.
- A drawing widget: The form or graphical design of the required widget is complex. No present widgets can achieve this, or requires much instructions. For example, a DartBoard, which is a target-like representation of radial plots.
- A composite widget: The developer can achieve this through Frames or GroupBoxes, but it is required in so many places of the application, that refactoring it to a composite widget is better for code separation and maintenance.

Once the developers has in mind what is needed to be accomplished, we can continue. Start with one widget. The example in which this code is based, and also a later section of this document will expand on what is needed for multiple widgets.

Tip: For an extended discussion on how to create the different kind of widgets, please refer to: *Widget Creation*

WAF Project

We recommend to separate widgets from applications. Widgets should be self contained, and have a clear interface. You can put them in a separate project, request them to be integrated into Control UI Toolkit, or in a package in your application. For sake of simplicity, and that this tutorial can be reproduced, we will create a new project.

The starting structure of the project is the following one:

```
widget-library/
├── widgets
│   ├── src
│   │   ├── include
│   │   │   ├── cut
│   │   │   │   └── examples
│   │   │   │       └── widgets
│   │   │   │           └── widgets
│   │   │   │               └── QeExampleDmsLabel.hpp
│   │   │   └── QeExamplesWidgets.h
│   │   └── QeExampleDmsLabel.cpp
│   └── wscript
└── wscript
```

The project directory is called `widget-library`, and has one module, called `widgets`. This will be the first artifact this project produces, and it is a shared library. But first, lets see the contents of the top-level wscript: `widget-library/wscript`.

```
#!/usr/bin/env python3
"""
@file
@copyright ESO - European Southern Observatory

@defgroup ELT CCS UI Toolkit - Widget Library Example
@brief This project explains how to create new widgets for E-ELT project.
"""
import os
from wtools.project import declare_project


def configure(cnf):
    # NOTE: WAF autoimports Core Gui and Widgets. But eventually this will be␣
↪removed.
    #  developers should always explicitly declared dependencies.
    if hasattr(cnf, "want_qt6") and cnf.want_qt6 is True:
        cnf.check_cfg(package="pyside6", uselib_store="pyside6", args="--cflags --
```

(continues on next page)

(continued from previous page)

```
↪libs")
    else:
        cnf.check_cfg(package="pyside2", uselib_store="pyside2", args="--cflags --
↪libs")

    cnf.check_cfg(package="erfa", uselib_store="erfa", args="--cflags --libs")


def qtcallable(cnf):
    # We can do some checks here, for example to discriminate between Qt6 and Qt5:
    if os.environ.get("ELT_RELEASE", "6").startswith("6"):
        return dict(version=6)
    else:
        return dict(version=5)


declare_project(
    "widgetexample",
    version="3.4.1-rc1",
    requires="cxx qt python pyqt",
    recurse="widgets plugin showcase bindings taurus showcasepy",
    qt=qtcallable,
)
```

PySide6 and PyQt6

ELT software uses **PySide6** as bindings. It is important that developers are aware of this, because there are two set of bindings for Qt libraries. PyQt6, and PySide6. PyQt6 has a longer time available in the market, but its license makes it impossible for us to use. PySide6 is newer, but is developed by Qt, and uses rule based code generation for bindings.

When searching for documentation, please always include the **PySide6** keyword in your search.

The wscript for the project start with dependencies resolution. It looks for the basic libraries needed for python bindings: PySide and Shiboken. If you need extra modules from Qt, or another library, like erfa, please add them here.

Currently, an auxiliary method *qtcallable()* selects the Qt version based on the DevEnv release.

The next method is *declare_project()*. We will skip most of the declaration (for this, please refer to: WAF Tools documentation[128]) except for *requires='cxx qt python pyqt'*. The project is being set up to include support for C++, Python, Qt libraries, and Python version of the Qt libraries. The *pyqt* name is unfortunate, but it is actually PySide6 libraries being used. Please refer to the *PySide6 and PyQt6*

[128] https://ftp.eso.org/pub/elt/repos/docs/documents/latest/wtools-docs/archive/html/index.html

sidebar for more information.

Widgets WAF Module

Under `widget-library/widgets`, a new WAF module is provided. The artifact from this module will be one C++ shared library, that is used when compiling and running applications that uses the widgets from this project. The name of the library will be `lib<target>.so`.

The contents of this module `wscript` is the following.:

```
from wtools.module import declare_qtcshlib

# NOTE: WAF autoimports Core Gui and Widgets. But eventually this will be removed.
#  developers should always explicitly declared dependencies.

# NOTE: The name of the target can be different from the name of the directory.
declare_qtcshlib(
    target='cutExamplesWidgets',
    use='erfa'
)
```

The declare_qtcshlib[129] command will automatically search for any UI files, and produce the necessary UIC instruction, creating the C++ file for that UI file.

Also, when installing this library, its target will be `$INTROOT/lib64`, which is already in the `LD_LIBRARY_PATH`, and therefore your program will be able to find it. Also, this will install the includes, so that waf can find them in `$INTROOT/include`.

Warning: Any C++ class that inherits from QObject[130] (including widgets), should include this 3 lines of code in their implementation (.cpp) file:

```
#if WAF
#include "include/<path_to_header>/widgetname.moc"
#endif
```

Qt uses signal, slot and properties. Connections between them are made using instrospection utilities that are added into all classes that inherit from QObject[131]. The MetaObject Compiler (or MOC) is in charge of generating a new file that provides that added features.

WAF needs these three lines of code to determine when a class need to have its .moc version generated (See WAF Qt5 documentation[132]).

[129] https://ftp.eso.org/pub/elt/repos/docs/documents/latest/wtools-docs/archive/html/namespacewtools_1_1module.html#abdd2ed3cb4f3c30e4cc6cabf7d5df920
[130] https://doc.qt.io/qt-6/qobject.html
[131] https://doc.qt.io/qt-6/qobject.html
[132] https://waf.io/apidocs/tools/qt5.html

The files in this `cut-widget-library/widgets` WAF module are:

```
widget-library/
├── widgets
│   ├── src
│   │   ├── include
│   │   │   ├── cut
│   │   │   │   └── examples
│   │   │   │       └── widgets
│   │   │   │           └── widgets
│   │   │   │               └── QeExampleDmsLabel.hpp
│   │   │   └── QeExamplesWidgets.h
│   │   └── QeExampleDmsLabel.cpp
│   └── wscript
└── wscript
```

Two files are needed to create a widget: `widget-library/widgets/src/include/cut/examples/widgets/widgets/QeExampleDmsLabel.hpp` and `widget-library/widgets/src/QeExampleDmsLabel.cpp`. The first one defined the class of the widget, and the second one implements the methods.

Header (.hpp)

All Qt Widgets should inherit from QWidget, or any other class that also does at some point of it inheritance tree – in this case, a QLabel.

QeExampleDmsLabel.hpp listing is found below. It inherits from QLabel, which inherits from QWidget. QLabel is a widget that presents a string on the GUI. It is one of the most used widgets. In this example, we will create a new widget named CutDmsLabel, and add a new interface to it, so that it can accept radians as input values, and automatically convert that value to Degrees, Minutes, Seconds (DMS) format using erfa.

Any class that inherits from QObject should have inside its class declaration, as the first statement, the call to the Q_OBJECT macro. This prepares structures that Qt needs to provide introspection as part of the MetaObject system.

```
1 #ifndef QEEXAMPLEDMSLABEL_HPP
2 #define QEEXAMPLEDMSLABEL_HPP
3
4 #include <QObject>
5 #include <QString>
6 #include <QVariant>
7 #include <QWidget>
8 #include <QLabel>
9
```

(continues on next page)

(continued from previous page)

```
/**
 * @brief A text label that prints from a radian, it representation in Degree/Minute/Seconds notation.
 * @class QeExampleDmsLabel CutExampleWidgets.h CutExampleWidgets.h
 * @ingroup widgets_widgets
 *
 * QeExampleDmsLabel allows to represent in +-DDD:MM:SS.sss format an angle.
 * This representation is used for Declination coordinates.
 * It uses erfa for conversions.
 */
class QeExampleDmsLabel : public QLabel {
    Q_OBJECT

    /**
     * @brief Value to represent in radians
     * @accessors getRadians(), setRadians(double)
     */
    Q_PROPERTY(double radians MEMBER m_radians READ getRadians WRITE setRadians NOTIFY radiansChanged )

    /**
     * @brief Precision used in the subsecond section of the label.
     * By default, millisecond precision is used.
     * @accessors getPrecision(), setPrecision(int)
     */
    Q_PROPERTY(int precision MEMBER m_precision READ getPrecision WRITE setPrecision NOTIFY precisionChanged )

public:
    explicit QeExampleDmsLabel(QWidget *parent = Q_NULLPTR);
    ~QeExampleDmsLabel();
    double getRadians(){ return m_radians; };
    int getPrecision(){ return m_precision; };

signals:
    /**
     * @brief Indicates that the radians value has been changed.
     */
    void radiansChanged(double newValue);

    /**
     * @brief Indicates the precision configuration has been changed.
     */
```

(continues on next page)

(continued from previous page)

```
50      void precisionChanged(int newValue);
51
52  public slots:
53      void setRadians(double newValue){this->m_radians = newValue; emit␣
    ↪radiansChanged(newValue); this->update();};
54      void setPrecision(int newValue){this->m_precision = newValue; emit␣
    ↪precisionChanged(newValue); this->update();};
55      void update();
56
57  private:
58      double m_radians = 0.0;
59      int m_precision = 3;
60  };
61
62
63  #endif // QEEXAMPLEDMSLABEL_HPP
```

It is important that the developer uses Qt's features to communicate. A widget's interface is composed of **Properties**, **Signals** and **Slots**.

In this example, we have created a Property using the `Q_PROPERTY` macro:

- The type of this property is *double*.
- The name of the property is *radians*.
- The value of the property will be stored in a MEMBER attribute of this class named *m_radians*. Qt does not handle set/get methods, this piece of information is mostly for the developer and not Qt.
- the getter method, is determined after READ and is named *getRadians*. It is up to the developer to provide it.
- the setter method is determined after the WRITE entry, and is named *setRadians*.
- when the value of the *radians* property changes, Qt will automatically trigger the *radiansChanged(double)* signal, that includes the new value as argument.

The implementation of the getter is very simple, and consists in just a return statement.:

```
double getRadians(){ return m_radians; };
```

Signals on the other hand, should never be implemented. They are provided as part of the signature of a class, but Qt through the MOC compiler automatically provides an implementation. The type in the signature should match the type of the property.

```
void radiansChanged(double newValue);
```

The setter method is also simple, and accepts as argument the same type of the property. The

implementation takes the value, and assigns it to the member attribute.

```
void setRadians(double newValue){this->m_radians = newValue; this->update();};
```

But at the end, it invokes *this->update()*. This is very important, as it enqueues a graphical update of the widget. Qt will keep track of these updates, and will request only once per frame that the widget redraws itself.

The other Property *precision* follows the same logic. There is one extra method defined called *update()*, which is same one we discussed before. We are overriding the *QWidget::update()* method, and we will see soon why.

Implementation (.cpp)

The constructor implementation calls immediately its parent constructor *QLabel(parent)*. It is very important to keep this call. Qt Widgets inside a GUI form a tree structure. One QMainWindow contains a central QWidget, which in turn will contains many QLabel, QSpinBox, maybe other QWidgets. But this tree structure is kept by the *parent* argument in constructors. Every new widget shall include this constructor signature as well.

The file also includes the three lines starting with `#if WAF` that indicates WAF that it should run the MOC compiled against this file to produce the MetaObject version of it.

Below is the listing of the implementation file:

```
1  #include <erfa.h>
2  #include "include/cut/examples/widgets/widgets/QeExampleDmsLabel.hpp"
3  #if WAF
4  #include "include/cut/examples/widgets/widgets/QeExampleDmsLabel.moc"
5  #endif
6
7  QeExampleDmsLabel::QeExampleDmsLabel(QWidget *parent) :
8    QLabel(parent)
9  {
10
11 }
12
13 QeExampleDmsLabel::~QeExampleDmsLabel()
14 {
15 }
16
17 void QeExampleDmsLabel::update()
18 {
19     int values[4];
20     char sign;
21     eraA2af(this->m_precision, this->m_radians, &sign, values );
```

(continues on next page)

(continued from previous page)

```
22      //qDebug() << "Values: " << values[0] << " " << values[1] << " " << values[2] <
    ↪< " " << values[3];
23      this->setText(QString("%1%2:%3:%4.%5")
24              .arg(QChar(sign))
25              .arg(values[0], 3, 10, QChar('0'))
26              .arg(values[1], 2, 10, QChar('0'))
27              .arg(values[2], 2, 10, QChar('0'))
28              .arg(values[3], this->m_precision, 10, QChar('0')));
29      QLabel::update();
30  }
```

The *update()* method does all the work in this case. Since all drawing instruction we need are already provided by the QLabel parent, we only need to prepare the correct string for presentation. In this case we take the *this->m_radians*, and use the *eraA2af()* method from erfa to obtain the correct values of DMS.

Then, a QString is prepared with the `+DDD:MM:SS.sss` format, and we use the setText() slot to assign it to our widget. But we still need to trigger the *QLabel::update()* method, so that it can execute its rendering instructions.

In general, when inheriting from another widget, we want to conduct our operations, and after that, execute the ones from the parent classes.

Header file

There is one extra file called `cut-widget-library/widgets/src/include/CutWidgets.h`. The reader may notice that it does not follow the `.hpp` extension convention, and it is outside of the include hierarchy. These two choices are intentional. This file will provide access to all include files when using the Qt Designer Plugin of this widget.

This will be explained later in the bindings sections, but its contents is simple: It includes every widget delivered by this WAF module:

```
1  #ifndef QEEXAMPLESWIDGET_H
2  #define QEEXAMPLESWIDGET_H
3
4  #include "cut/examples/widgets/widgets/QeExampleDmsLabel.hpp"
5
6  #endif // QEEXAMPLESWIDGET_H
```

Compilation

At this point, the reader may go back to the root of the project, and execute:

```
waf configure build install
```

This will search for the dependencies expressed in the top-level wscript file, and execute the compilation instructions for our widget.

Thoughts on Widgets Properties, Signal and Slots

It is important that developers uses slots, signals and properties for the interface of the widget. The Qt Designer application included with Qt makes use of these to allow quick design of interfaces. For example:

A developer creates a widget, that plots the trend of datapoint, but it only updates the graphics if another condition is met. For this:

- A slot *criteriaMet(bool)* can be created. This will serve as a trigger to conduct a *update()* on the widget. The developer may program the criteria whereever is needed, but the widget has to be notified somehow. Slots are very useful for these cases.
- A signal *trendUpdated()* can be used to indicate any other interested part of the application (other widgets, but also other code) that the plot has been updated.
- A property with a *long int* representing the timestamp of the update can be kept.

In the example above, the property, or to be exact, the slot of a property *setRadians(double)* is used to change the value in the widget. It is a common use for widgets to have properties that allow access to the value they present. In this sense, the *precision* Property is a bit different. Though also saved, it is used as configuration of the widget.

Widgets are configured through method, slots or properties. Qt's documentation includes these characteristics in their API. Properties can be used in the Qt Designer to configure widgets, so any characteristics of your new widget that can be configured is a candidate for a property.

Plugin WAF Module

At this point, the project needs to be extended. The new `widget-library/plugin` module is added as shown in the following listing:

```
widget-library
├── plugin
│   ├── src
│   │   ├── include
│   │   │   └── cut
```

(continues on next page)

(continued from previous page)

```
│   │   │       └── examples
│   │   │           └── widgets
│   │   │               └── plugin
│   │   │                   ├── QeExampleDmsLabelPlugin.hpp
│   │   │                   └── QeExamplesWidgetCollection.hpp
│   │   ├── QeExampleDmsLabelPlugin.cpp
│   │   └── QeExamplesWidgetCollection.cpp
│   └── wscript
├── widgets
│   ├── src
│   │   ├── include
│   │   │   ├── cut
│   │   │   │   └── examples
│   │   │   │       └── widgets
│   │   │   │           └── widgets
│   │   │   │               └── QeExampleDmsLabel.hpp
│   │   │   └── QeExamplesWidgets.h
│   │   └── QeExampleDmsLabel.cpp
│   └── wscript
└── wscript
```

The `plugin/wscript` file looks a bit different than the one in `widgets` WAF module:

Listing 1: ../../../../examples/widget_library/plugin/wscript

```
from wtools.module import declare_qtcshlib

# In the "use" parameter, if using modules from the same project,
#  you should use the name of the directory. It can be different from
#  the name of the target of that directory.
declare_qtcshlib(
    target='cutExamplesWidgetsPlugin',
    use='widgets',
    qt_plugin=True
)
```

This wscript file also uses the declare_qtcshlib[133] wtools instruction. The main difference here is the additional keyword argument `qt_plugin=True`. This will indicate to the install command that the resulting library should be place into `$INTROOT/lib64/designer`. Then the `QT_PLUGIN_PATH` environment variable can be used by Qt Designer to find any widgets plugin. `QT_PLUGIN_PATH` shall be `$INTROOT/lib64`, as the designer expects a certain directory structure. One of these directories is `designer`, which is the one used to store any plugin intended for Qt Designer.

[133] https://ftp.eso.org/pub/elt/repos/docs/documents/latest/wtools-docs/archive/html/namespacewtools_1_1module.html#abdd2ed3cb4f3c30e4cc6cabf7d5df920

The second difference is that the `use` argument for declare_qtcshlib[134] command now include `widgets`. This indicates WAF that `widgets` module is a dependency, and therefore, must be compiled before this module. Also, when compiling this `plugin` module, its includes added to include paths, and its library added for linking instructions. `widgets` string must be the names of the directory structure, not the target names. If many directory levels are used in the project, then replace `/` with `..`

Also, the installation instruction does not copy the includes from this module to the `$INTROOT`. As the plugins are not intended to be used for development, these files are not meant for further development.

Header files

Two classes, one for QeExampleDmsLabel plugin, and another for a Collection, are defined in this module.

The listing of the QeExampleDmsLabelPlugin.hpp is given below:

Listing 2: examples/widget_library/plugin/src/include/cut/examples/widgets/plugin/QeExam

```
1  /**
2   * @file
3   * @ingroup widgets_plugin
4   * @copyright ESO - European Southern Observatory
5   * @author Arturo Hoffstadt Urrutia <ahoffsta@eso.org>
6   *
7   * @brief QeDmsLabel plugin for Qt Designer
8   */
9
10 #ifndef QEEXAMPLEDMSLABELPLUGIN_HPP
11 #define QEEXAMPLEDMSLABELPLUGIN_HPP
12
13 #include <QtUiPlugin/QDesignerCustomWidgetInterface>
14
15 /**
16  * @brief QeExampleDmsLabel Plugin for Qt Designer. Enables WYSIWYG design while␣
    ↪using
17  * Qt Designer.
18  * @class QeExampleDmsLabelPlugin
19  * @ingroup widgets_plugin
20  *
21  * This class implements the QDesignerCustomWidgetInterface that enables
22  * Qt Designer to create new instances of the QeDmsLabel widget, and obtain
23  * information to present the widget, in the WYSIWYG view of the form, its
24  * toolbar, and properties view.
```

(continues on next page)

[134] https://ftp.eso.org/pub/elt/repos/docs/documents/latest/wtools-docs/archive/html/namespacewtools_1_1module.html#abdd2ed3cb4f3c30e4cc6cabf7d5df920

(continued from previous page)

```
 *
 * @warning This class is not intended for developers, and is internal to CUT.
 */
class QeExampleDmsLabelPlugin : public QObject, public␣
↪QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)

public:
    QeExampleDmsLabelPlugin(QObject *parent = 0);

    bool isContainer() const;
    bool isInitialized() const;
    QIcon icon() const;
    QString domXml() const;
    QString group() const;
    QString includeFile() const;
    QString name() const;
    QString toolTip() const;
    QString whatsThis() const;
    QWidget *createWidget(QWidget *parent);
    void initialize(QDesignerFormEditorInterface *core);

private:
    bool m_initialized;
};

#endif // QEEXAMPLEDMSLABELPLUGIN_HPP
```

This file does not change much from widget to widget. Only the name of the class does which is used for Class name, and constructor.

The contents of the Collection are also not very dynamic:

Listing 3: examples/widget_library/plugin/src/include/cut/examples/widgets/plugin/QeExam

```
#ifndef QEEXAMPLEWIDGETCOLLECTION_HPP
#define QEEXAMPLEWIDGETCOLLECTION_HPP

#include <QtUiPlugin/QDesignerCustomWidgetInterface>
#include <qplugin.h>

class QeExamplesWidgetCollection : public QObject, public␣
```

(continues on next page)

(continued from previous page)

```
↪QDesignerCustomWidgetCollectionInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetCollectionInterface)
#if QT_VERSION >= 0x050000
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.
↪QDesignerCustomWidgetCollectionInterface")
#endif // QT_VERSION >= 0x050000

public:
    explicit QeExamplesWidgetCollection(QObject *parent = 0);

    virtual QList<QDesignerCustomWidgetInterface*> customWidgets() const;

private:
    QList<QDesignerCustomWidgetInterface*> m_widgets;
};

#endif // QEEXAMPLEWIDGETCOLLECTION_HPP
```

The name of this class is tied to the name of the library instead of a particular widget. This Collection will provide access to all widgets in the plugin. The matter that we only provide only changes nothing. Qt Designer provides to us an interface ready to use to extend it with more widgets.

Implementation

The plugin implementation has more details. The Collection implementation needs to add the Plugin classes to a list during construction. And a specific method returns the list, informing Qt Designer of the widgets included in this Collection.

Listing 4: examples/widget_library/plugin/src/QeExamplesWidgetCollection.cpp

```
#include "include/cut/examples/widgets/plugin/QeExamplesWidgetCollection.hpp"
#include "include/cut/examples/widgets/plugin/QeExampleDmsLabelPlugin.hpp"

#if WAF
#include "include/cut/examples/widgets/plugin/QeExamplesWidgetCollection.moc"
#endif


QeExamplesWidgetCollection::QeExamplesWidgetCollection(QObject *parent)
    : QObject(parent)
{
```

(continues on next page)

(continued from previous page)

```
12      m_widgets.append(new QeExampleDmsLabelPlugin(this));
13  }
14
15  QList<QDesignerCustomWidgetInterface*> QeExamplesWidgetCollection::customWidgets()␣
    ↪const
16  {
17      return m_widgets;
18  }
```

The Plugin implementation itself deals with necessary entries used during code generation. The UI files are themselves XML, and every time a widget is added to a UI file using the Qt Designer, it will query the different methods from this implementation.

Listing 5: examples/widget_library/plugin/src/QeExampleDmsLabelPlugin.cpp

```
1   #include "cut/examples/widgets/widgets/QeExampleDmsLabel.hpp"
2   #include "include/cut/examples/widgets/plugin/QeExampleDmsLabelPlugin.hpp"
3
4   #include <QtPlugin>
5
6   #if WAF
7   #include "include/cut/examples/widgets/plugin/QeExampleDmsLabelPlugin.moc"
8   #endif
9
10  QeExampleDmsLabelPlugin::QeExampleDmsLabelPlugin(QObject *parent)
11      : QObject(parent)
12  {
13      m_initialized = false;
14  }
15
16  void QeExampleDmsLabelPlugin::initialize(QDesignerFormEditorInterface * /* core */)
17  {
18      if (m_initialized)
19          return;
20
21      // Add extension registrations, etc. here
22
23      m_initialized = true;
24  }
25
26  bool QeExampleDmsLabelPlugin::isInitialized() const
27  {
28      return m_initialized;
29  }
```

(continues on next page)

(continued from previous page)

```
30
31  QWidget *QeExampleDmsLabelPlugin::createWidget(QWidget *parent)
32  {
33      return new QeExampleDmsLabel(parent);
34  }
35
36  QString QeExampleDmsLabelPlugin::name() const
37  {
38      return QLatin1String("QeExampleDmsLabel");
39  }
40
41  QString QeExampleDmsLabelPlugin::group() const
42  {
43      return QLatin1String("Display Widgets");
44  }
45
46  QIcon QeExampleDmsLabelPlugin::icon() const
47  {
48      return QIcon();
49  }
50
51  QString QeExampleDmsLabelPlugin::toolTip() const
52  {
53      return QLatin1String("Displays as Degrees-Minutes-Seconds, or Hour-Minutes-
    ↪Seconds, an angle in radians.");
54  }
55
56  QString QeExampleDmsLabelPlugin::whatsThis() const
57  {
58      return QLatin1String("");
59  }
60
61  bool QeExampleDmsLabelPlugin::isContainer() const
62  {
63      return false;
64  }
65
66  QString QeExampleDmsLabelPlugin::domXml() const
67  {
68      return QLatin1String("<widget class=\"QeExampleDmsLabel\" name=\
    ↪"qeExampleDmsLabel\">\n"
69                           "  <property name=\"radians\">\n"
70                           "    <double>0.0</double>\n"
```

(continues on next page)

(continued from previous page)

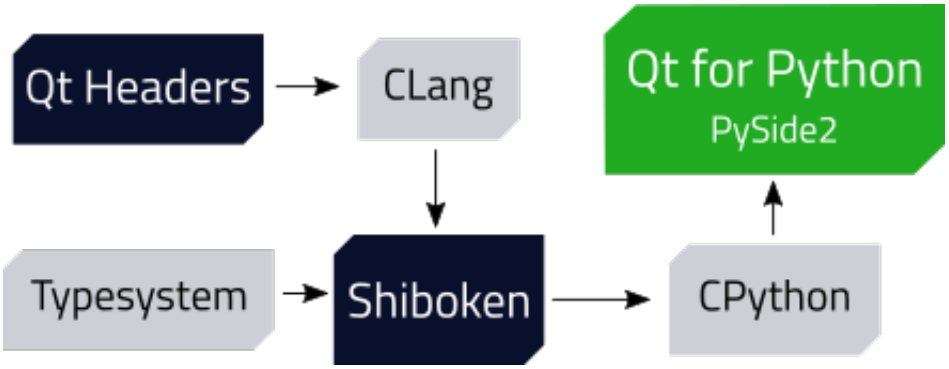
```
                                "  </property>\n"
                                "  <property name=\"precision\">\n"
                                "     <number>3</number>\n"
                                "  </property>\n"
                            "</widget>\n");
}

/*
 * We use a small hack to allow compatibility between Python
 * and C++ resolution of classes on the UI compilers.
 * C++ UI compiler uses the includeFile() to construct its
 * include statement.
 *
 * #include QeWidgetExample.h
 * QeDartBoard* qeDartBoard = new QeDartBoard();
 *
 * Python UI compiler uses the includeFile() and the name()
 * to construct the import statement, but removes the '.h'
 * suffix.
 *
 * from QeWidgetExample import QeDartBoard
 *
 * Using an intermediary file allows us to use the same
 * includeFile() string under Python and C++.
 */
QString QeExampleDmsLabelPlugin::includeFile() const
{
    return QLatin1String("QeExamplesWidgets.h");
}

////
```

- `createWidget()` should return a new widget. This is used by the designer to place a widget in the view for the WYSIWYG editor.
- `name()` returns the class name of the widget. It has to be the name of the class in C++/Python.
- `group()` returns a name for the category the widget is placed in the Qt Designer *Toolbox*.
- `icon()` returns a Pixmap that represents the widget in a concise manner. Used in the Toolbox along with the `name`, and in the *Object Inspector* Docking Widget.
- `toolTip()` contextual help shown as a tooltip on the *Toolbox*.
- `whatsThis()` contextual help shown when the What's This feature is activated. Usually is longer than the tooltip.

- `domXml()` initial XML inserted into the UI file when the widget is added to the file.

Bindings

Shiboken[135] is a Python binding generator. It uses CLang[136] to parse and analyze the structure of the C++ code (both from Qt, and our own widget), and typesystem[137] to create the generated code through instructions.



Qt uses Shiboken to create its Python bindings, and already provides typesystem files for all its libraries. This is how PySide[138] is built.

The major issue of Shiboken, is that it uses CMake3 and a huge CMakeList.txt that is very difficult to understand. There is no other build system supported. In this page, I will explain how Shiboken works, and how the CMakeLists.txt file was translated.

Input for Bindings

Input files for binding generation in Shiboken are simple. Two files are needed:

- C++ header file, that includes every class we need to bind.
- Typesetting XML file, that indicates which will be the name of the target Python module, and which classes will be bind into said module.

The header file is just a list of `#include` sentences, no extra code is needed. Every class that needs binding has to be included in this file, through the use of common `#include` macro instructions. If two classes are defined in the same file, then only one include is needed.

```
1 /*****************************************************************************
2 * ESO License
3 *****************************************************************************/
```

(continues on next page)

[135] https://doc.qt.io/qtforpython/shiboken2/shibokengenerator.html
[136] https://clang.llvm.org/
[137] https://doc.qt.io/qtforpython/shiboken2/typesystem.html
[138] https://wiki.qt.io/Qt_for_Python

(continued from previous page)

```
4
5  #ifndef BINDINGS_H
6  #define BINDINGS_H
7
8  #define QT_ANNOTATE_ACCESS_SPECIFIER(a) __attribute__((annotate(#a)))
9
10 #include "QeExamplesWidgets.h"
11
12 #endif // BINDINGS_H
```

- Line 8 indicates to the C++ preprocessor that it needs to make classifiers used by Qt visible. These classifiers are used in QObjects. (signals:, slots: in the header files).
- In Line 10, you should include every widget you need. In here, we recommend to use only file named as the resulting target name. This will ensure compatibility with Python module names, allowing both C++ and Python version of the widget to provide correct include/import statements.

The typesystem file is a bit more complex.

```
1  <?xml version="1.0"?>
2  <typesystem package="QeExamplesWidgets">
3      <load-typesystem name="typesystem_core.xml" generate="no"/>
4      <load-typesystem name="typesystem_core_common.xml" generate="no" />
5      <load-typesystem name="typesystem_widgets.xml" generate="no" />
6      <load-typesystem name="typesystem_gui_common.xml" generate="no" />
7      <primitive-type name="double"/>
8      <object-type name="QeExampleDmsLabel">
9      </object-type>
10 </typesystem>
```

- As you see here, the XML defines in line 2 a package called *QeExamplesWidgets*. This will be the name of the resulting Python package.
- Lines 3 to 6 indicate the typesystem engine to load more typesystem definitions. These ones are the base set that Qt needs for widgets.
- Then one primitive is declared in line 7, as it is used in slots and signals in the QeExampleDmsLabel class. If any other primitive is used, but it is not in the signature, then no definition for them is needed.
- In line 8 an object is declared. This object is a C++ class named QeExampleDmsLabel. This should be the same class name as in the C++ code declaration. The resulting Python library will have a class in package *QeExamplesWidgets* named QeExampleDmsLabel.

Code generation

In Qt's examples, CMake is used. I will indicate here how to do this without a build system (and eventually, how to do it in WAF).

The Code generation is one command **shiboken** with several options for includes of options, and passes two mandatory arguments, the C++ header file, and the typesystem XML file:

```
42          "--enable-pyside-extensions "\
43          "--use-isnull-as-nb_nonzero "\
44          "--avoid-protected-hack "\
45          "--language-level=c++17 "\
46          "-I${SRC_DIR}/../../widgets/src/include "\
47          "-I${SRC_DIR} "\
48          "-I${INTROOT}/include "\
49          + bld.env.GEN_QT_INCLUDES +
50          "-I/usr/include/ "\
51          "-T" + str(bld.env.GEN_PYSIDE_TS_typesystemdir) + " "\
52          "-T${SRC_DIR} "\
53          "--output-directory=${DEST_DIR} "\
54          "${SRC_DIR}/bindings.h "\
55          "${SRC_DIR}/bindings.xml",
56    source=['src/bindings.xml', 'src/bindings.h'],
57    target=[
58        'src/QeExamplesWidgets/qeexampleswidgets_module_wrapper.cpp',
59        'src/QeExamplesWidgets/qeexampledmslabel_wrapper.cpp'
60        ],
61    name='bindings-generate',
62  )
63
64  compilation = bld(
65    features='pyext cxx cxxshlib',
66    source=[
67        'src/QeExamplesWidgets/qeexampleswidgets_module_wrapper.cpp',
```

The **SHIBOKEN** variable is path to the code generator. We use `waf` to locate the executable in the configuration section. At the moment, the include statements `-I` for the code generation are still manually constructed from `pkg-config` information.

- Line 51 indicates where the header files for the source widgets are.
- Line 53 indicates where the bindings.h file is located
- Line 54 give access to QtWidgets, QtCore and QtGui shiboken, and pyside obtained through pkg-config.

The ones in `-T` entries are special. `-T` indicates where the typesettings files are located, in case we need to "include" other libraries.

- Line 56 indicates the location of all typesettings files from Qt. In this case, I do not want to specify how to bind a QWidget, QPen, QBrush classes, so I just indicate that my project uses typesettings files from Qt project.
- Line 57 indicate where the typesetting file from our project is located.

The options do not change from one project to another, only the *target* entry from the generation stage.

Tip: If you need more example, we suggest to look at typesystem files from Qt:

- All of them: https://code.qt.io/cgit/pyside/pyside-setup.git/tree/sources/pyside6/PySide6
- QtWidgets (the most useful ones, as most widgets E-ELT will develop should look like these): https://code.qt.io/cgit/pyside/pyside-setup.git/tree/sources/pyside6/PySide6/QtWidgets

---

As a result, this will generate a directory in *DEST_DIR*, that has 2 files for each bound class, and 2 more for each Python module indicated as target.

Compilation

The `compilation` of these are more or less straight C++ objects, that need to be linked together as a shared library.

A `compilation` step prepares the shared library, using as source the same list as the target from the generation step.

```
69          ],
70      target='QeExamplesWidgets',
71      includes=[
72        str(bld.env.COMP_PYSIDE_I_includedir)+'/QtCore',
73        str(bld.env.COMP_PYSIDE_I_includedir)+'/QtWidgets',
74        str(bld.env.COMP_PYSIDE_I_includedir)+'/QtGui',
75      ],
76      use=
77          bld.env.GEN_COMP_QT_USE +
78          [
79          "widgets"
80          ],
81      name='bindings',
82      # Fix for ELTDEV-853
83      pytest_path  = [ bld.path.get_bld().abspath() ],
84    )
85    compilation.cxxflags = ["-Wno-pedantic"]
86
```

(continues on next page)

(continued from previous page)

```
87
88 declare_custom(
89              #depends=['cut-widgets-widgets'],
90              provides=['bindings'],
```

This `compilation` step has the necessary features that provides it with the compiler options to properly generate a c++ shared library, that has python and qt as dependency.

An extra set of includes entries for the PySide module are added, and a use statement is introduced to depend on the **widgets** waf module from the previous chapter of this guide.

This is how the module would look like with bindings added:

```
bindings
├── src
│   ├── bindings.h
│   └── bindings.xml
└── wscript
```

Taurus Databinding Widgets

To add declarative databinding capabilities to a custom widget the developer needs to create a set of classes:

TaurusExampleDmsLabel
: It inherits from the new widget itself, and from TaurusBaseWidget. It is in charge of creating a new instance of the appropriate Controller class, and storing properties that are used by the declarative databinding layer.

  This class is used also by the designer, so the properties it declares, are present in the *Property Editor* tab.

TaurusExampleDmsLabelController
: Inherits from TaurusBaseController, which already makes the last value available using the provided `valueObj()`. This class should take the value, and apply it to the class created above.

  The TaurusBaseController.update() method invokes the self._updateConnections(), self._updateForeground(), self._updateBackground(), self._updateToolTip(). Developers needs to implement or reuse existing ones.

Tip: The `examples/widget_library/taurus` includes an example widget.

Taurus Controller

Looking into the taurus adaptation layer file in the examples `taurusexampledmslabel.py`

Listing 6: examples/widget_library/taurus/src/cut/examples/widgets/taurus/taurusexampled

```
class TaurusExampleDmsLabelController(TaurusBaseController):
    """Controller for the TaurusExampleDmsLabel widget.

    It uses the values received by the TaurusBaseController.handleEvent() method to␣
↪provide
    the appropiete data manipulation for the Widget.

    This one in particular reuses the _updateBrackground from the TaurusLabel, but␣
↪provides
    its own _updateForeground, which propagates the value magnitude to the radians␣
↪QeDmsLabel
    property.
    """

    _updateBackground = updateLabelBackground

    def __init__(self, label):
        TaurusBaseController.__init__(self, label)

    def _setStyle(self):
        TaurusBaseController._setStyle(self)
        label = self.widget()
        # if update as palette
        if self.usePalette():
            label.setFrameShape(Qt.QFrame.Box)
            label.setFrameShadow(Qt.QFrame.Raised)
            label.setLineWidth(1)

    def _updateForeground(self, widget):
        value = None
        format = None
        value = 0.0
        valueObj = self.valueObj()
        if valueObj is not None and valueObj.rvalue is not None:
            format = self.attrObj().data_format
            value = valueObj.rvalue
            if format == DataFormat._0D:
                self.widget().setRadians(value)
```

The class TaurusExampleDmsLabelController inherits from TaurusBaseController. This class will only

concern itself with the manipulation of the last value gotten from an event (Change or Periodic). For that, we need to implement two methods: `_updateBackground` and `_updateForeground`.

Since we want the background role to behave the same as the TaurusLabel, we just import that implementation, as assign to the `_updateBackground` that implementation.

The `_updateForeground` implementation is custom, but rather simple. The last value from an event can be retrieved using `self.valueObj()`. A reference to the widget this Controller class manipulates can be retrieved using `self.widget()`. This implementation just makes a simple check to see if the data format is a scalar, and sets the value using the radians property in the QeExampleDmsLabel.

Taurus Widget

Listing 7: examples/widget_library/taurus/src/cut/examples/widgets/taurus/taurusexamplec

```
    def __init__(self, parent=None, designMode=False):
        """Sets the initial values.
        It is important the the value and assignations are separate, as the␣
↪resetXYZ method for
        properties needs to get the same value to reset the property.
        """
        self._bgRole = self.DefaultBgRole
        self._fgRole = self.DefaultFgRole
        self._modelIndex = self.DefaultModelIndex
        self._modelIndexStr = ""
        self._controller = None
        name = self.__class__.__name__
        self.call__init__wo_kw(QeExampleDmsLabel, parent)
        self.call__init__(TaurusBaseWidget, name, designMode=designMode)
        self.setAlignment(self.DefaultAlignment)
```

The constructor only prepared initial values. It is important to notice that the Default values are not stored in the `__init__` method, since these are reused later in the reset methods.

Listing 8: examples/widget_library/taurus/src/cut/examples/widgets/taurus/taurusexamplec

```
    def _calculate_controller_class(self):
        ctrl_map = _CONTROLLER_MAP
        model_type = self.getModelType()
        ctrl_klass = ctrl_map.get(model_type, TaurusExampleDmsLabelController)
        return ctrl_klass
```

```
    def controller(self):
        ctrl = self._controller
        # if there is a controller object and it is not the base controller...
```

(continues on next page)

(continued from previous page)

```
        if ctrl is not None and not ctrl.__class__ ==␣
↪TaurusExampleDmsLabelController:
            return ctrl

        # if there is a controller object and it is still the same class...
        ctrl_klass = self._calculate_controller_class()
        if ctrl is not None and ctrl.__class__ == ctrl_klass:
            return ctrl

        self._controller = ctrl = ctrl_klass(self)
        return ctrl
```

```
    def controllerUpdate(self):
        ctrl = self.controller()
        if ctrl is not None:
            ctrl.update()
```

These three methods deal with the controller for the Widget. They provide a manner to retrieve the the proper controller for a given situation. In most cases, the implementation will look similar.

```
    def handleEvent(self, evt_src, evt_type, evt_value):
        ctrl = self.controller()
        if ctrl is not None:
            self.controller().handleEvent(evt_src, evt_type, evt_value)
```

The `handleEvent` method forwards the event information to the controller. This allows to move the manipulation of new values away from the widget into the controller.

```
    def setModel(self, m, **kwargs):
        # force to build another controller
        self._controller = None
        TaurusBaseWidget.setModel(self, m, **kwargs)
        if self.modelFragmentName:
            self.setFgRole(self.modelFragmentName)
```

The `setModel` implementation calls its ancestors implementation; the main difference is that it forces the re-creation of the controller, in order to react to possible need of a different kind of controller.

```
    def isReadOnly(self):
        return True
```

Returning True in `isReadOnly` indicates that this widget provides no write capability.

The rest of the implementation is declaration of properties, which is already explained in *Common to all kind of widgets*.

Tip: If this is your first time adapting a widget to a databinding layer such as Taurus, we recommend that you follow the simple solution first. You can try later to follow the more complete solution.

Multiple Widgets

Adding a second widget to the library can be done by following these instructions.

widgets
: A new hpp header file and a new widget implementation cpp. Remember to define the interface of the widget using properties, signals and slots.

plugin
: Add a new header hpp file for the plugin, and an implementation. Remember the workaround for `includeFile()` method. Another modification is needed in the Collection class: you need to add to the `m_widgets` member list the new widget plugin.

showcase
: At this point you can add the widget using drag and drop. Remember to create entries for the new properties, and connect them to the new widget.

bindings
: Add to the bindings.xml the new class. Remember to add special event handling if you have introduced overridden methods. Modify the wscript to include in src and target files the new cpp files for the bound python class.

showcasepy
: At this point you can add the widget using drag and drop. Remember to create entries for the new properties, and connect them to the new widget.

taurus
: Create a new python module, and create the TaurusController and TaurusWidget new classes.

In any case, no new WAF modules are needed, since every module already deliver artefacts that consider multiple possible widgets.

4.4 Demo Service Client

This documents teaches the reader how to develop a GUI client application to the cutDemoService server application. This tutorial uses Python, PySide and Taurus as libraries; it also utilizes the different Taurus Plugins covered in the Beginner tutorial to connect to the cutDemoService server application.

The reader will learn new dependencies needed, and how to use the *taurus designer* tool to add widgets that connect to remote servers.

Download, Compilation, and Running

This document guides the user on the creation of a new Demo Service Client, using Python, WAF modules, Qt, Taurus and CUT plugins. As this is a programming tutorial, you will produce file and code as instructed. You can also download a finalized version of the code from `https://gitlab.eso.org/ecos/cut/-/tree/master/examples/`, so our first step will be to download it.

```
$ git clone https://gitlab.eso.org/ecos/cut
```

```
$ mv cut/examples/demo_service_client ./
$ rm -rf cut
```

The directory we will be using for this tutorial is the `demo_service_client`, and it contains the code for a Python application. This is a waf project on its own, so it can be build in an independent manner from the rest of Control UI Toolkit. In order to do so, the reader must execute:

```
$ cd demo_service_client
$ waf configure install
```

And to execute the application:

```
$ cutDemoServiceClient
```

Project Structure

This is the structure of the project. We use as build system WAF[139], and on top wtools[140] extension to make ELT application definition easier:

```
demo_service_client
├── client
│   ├── src
│   │   ├── cut
│   │   │   └── examples
│   │   │       └── demo_service_client
│   │   │           ├── __init__.py
│   │   │           ├── applicationconfiguration.py
│   │   │           ├── mainwindow_ui.ui
│   │   │           └── mainwindowcontroller.py
│   │   └── cutdemoserviceclient.py
│   └── wscript
└── wscript
```

[139] https://waf.io/book/
[140] https://www.eso.org/~eltmgr/documents/latest/wtools-docs/archive/html/index.html

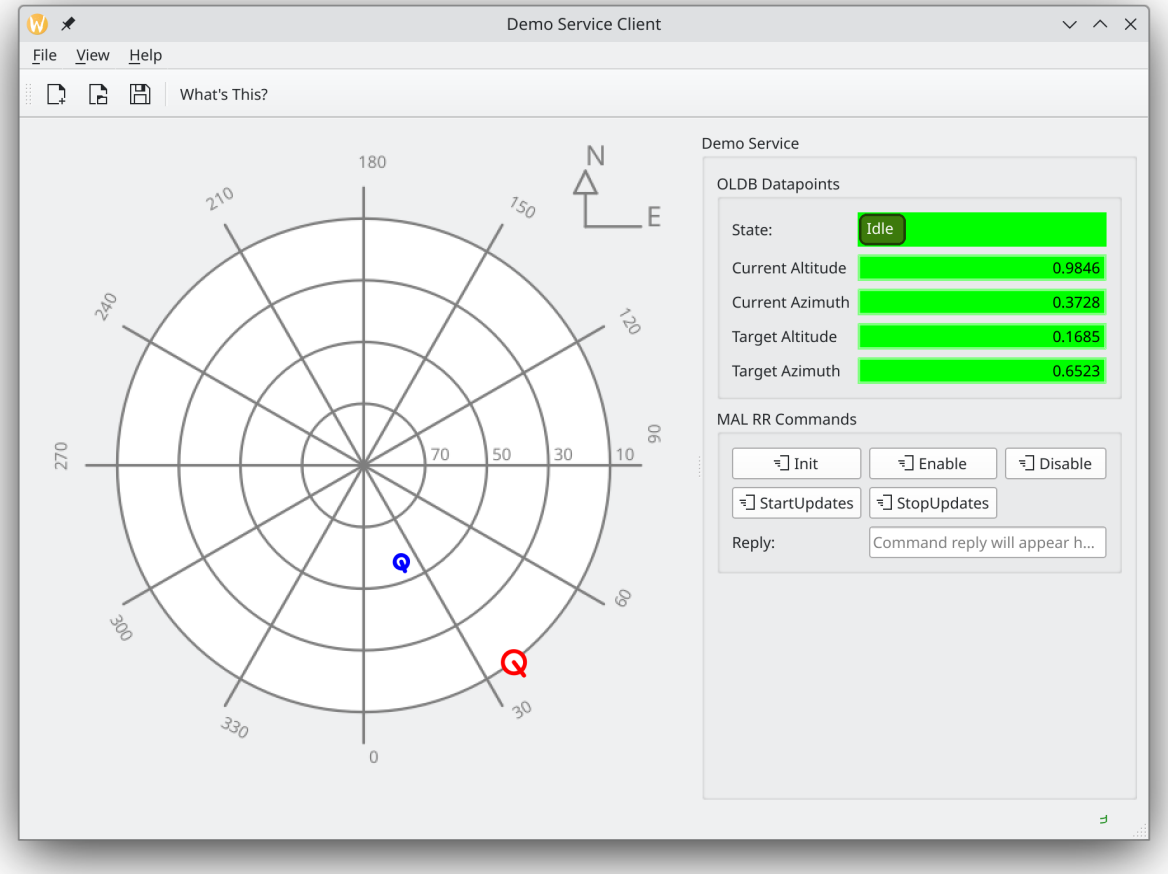


Fig. 60: Demo Service Client: The application will include these elements

Project wscript

The first file we should take a look is `demo_service_client/wscript`:

```
#!/usr/bin/env python3
import os
from wtools.project import declare_project


def configure(cnf):
    # We check which version of qt6 is requested and set the proper feature for later
    if hasattr(cnf, "want_qt6") and cnf.want_qt6 is True:
        cnf.check_python_module("PySide6")
    else:
        cnf.check_python_module("PySide2", condition="ver >= num(5, 14, 1) and ver < num(6, 0, 0)")
    cnf.check_python_module("taurus")
    cnf.check_python_module("elt.cut")
    cnf.check_python_module("CutWidgets")
    cnf.check_python_module("tauruscii")
    cnf.check_python_module("taurusmal")


def qtcallable(cnf):
    # We can do some checks here, for example to discriminate between Qt6 and Qt5:
    if os.environ.get("ELT_RELEASE", "6").startswith("6"):
        return dict(version=6)
    else:
        return dict(version=5)


declare_project(
    "DemoServiceClient",
    version="3.1.10-rc1",
    requires=["cxx", "qt", "python", "pyqt"],
    recurse="client",
    qt=qtcallable,
)
```

In contrast to what has been shown in the *Python Application Tutorial*, extra python module checks for *tauruscii* and *taurusmal* have been added. These modules provides the OLDB and MAL RR plugins for Taurus.

The rest of the project structure has not been changed.

UI Design

This UI inherits several elements from the *Python Application Tutorial* UI. The *Menu Bar* and *Toolbar* are still the same.

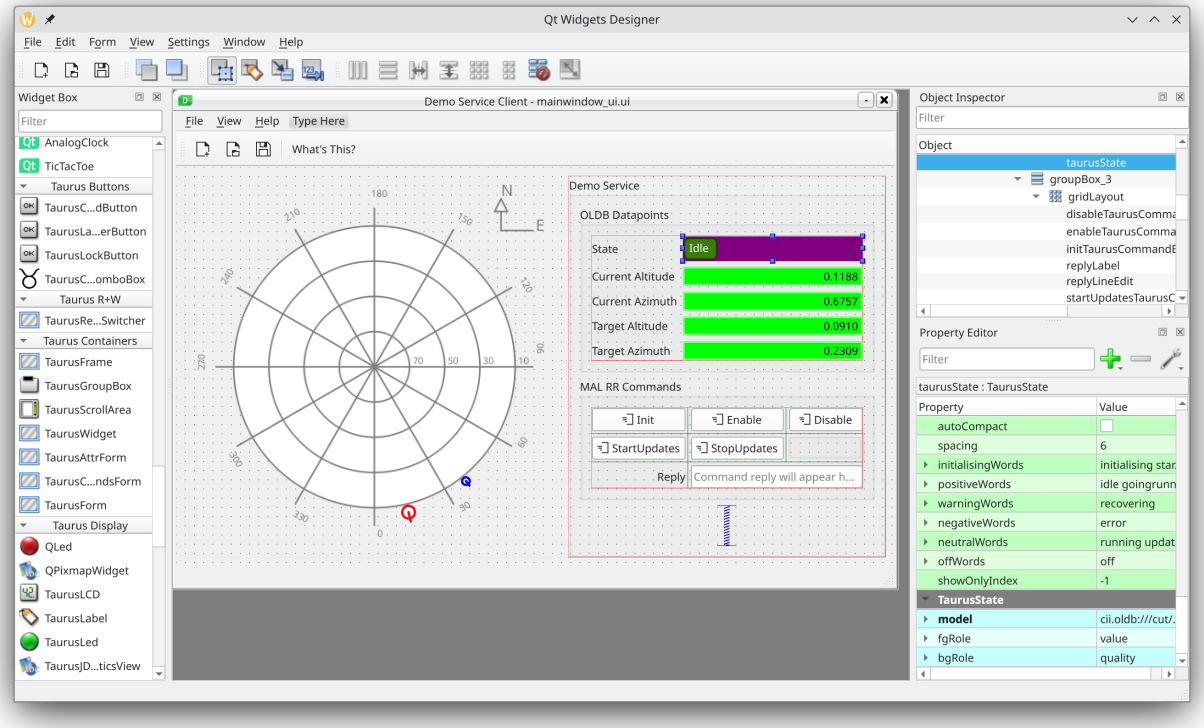


Fig. 61: Taurus Designer: UI form for Demo Service Client

The QeDartBoard has been replaced by TaurusDartBoard. The Double Spin Box entries for the angles have been replace for TaurusLabels.

Since this GUI client will connect to a server, the reader should use the widgets that provide declarative databinding. This will make the creation of the UI very fast.

Form

Start by creating a new Form. Click on the *New* button on the toolbar. This will cause the *New form – Qt Widgets Design* dialog to pop up. From the its list of options on the left, select *templates/forms* → *Main Window* entry. Then click on the *Create* button at the bottom of the dialog.

A new Form will appear at the gray area in the center of the Designer.

Start by adding three top-level menus called **&Application**, **&View**, and **&Help**. As explained in *Python Application Tutorial* the **&** creates an accelerator for accesing the menu by keyboard. You can

do this by clicking the *Type Here* entry in the new form.

Open the *Action Editor* docking window. Use the Top Menubar and select *Settings* → *Action Editor*. If the entry is already checked, then the docking window should already be present in your Designer. Please create the entries as describe in the picture below. You can use *Actions* as instructions on how to do this reference.

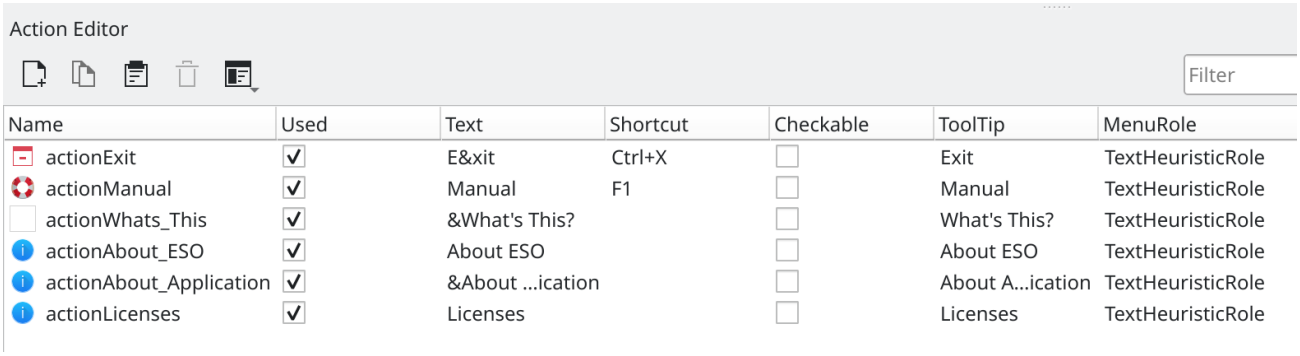


Fig. 62: Actions for the Demo Service Client

Add **Exit** action to **Application** menu, and the rest of the help actions to the **Help** menu. Small changes has been applied. The removal of **Edit** menu, changing **File** for **Application**, and the removal of several entries in **Application**. Also the **Manual** entry has a shortcut.

TaurusDartBoard

From the Designers *Widget Box* located on the left of the designers window, search or locate the *TaurusDartBoard* entry. Start the drag and drop operation by clicking and dragging a new TaurusDartBoard into an empty space of the UI Form.

Using the *Property Editor* set its model to:

```
eval:Q([{cii.oldb:///cut/demoservice/instance1/double-vector-current-altaz}[0],
↪{cii.oldb:///cut/demoservice/instance1/double-vector-current-altaz}[1], {cii.
↪oldb:///cut/demoservice/instance1/double-vector-target-altaz}[0] ,{cii.oldb:///
↪cut/demoservice/instance1/double-vector-target-altaz}[1]])
```

This combines the evaluation plugin and the OLDB plugin to generate a four element vector. You can see that the original elements of the OLDB are also vectors.

The first two elements will be used as entries for the current alt/az, while the third and the fourth will be used for the target alt/az.

OLDB Datapoints

From the Designers *Widget Box* located on the left of the designers window, search or locate the *Form Layout* entry. Start the drag and drop operation by clicking and dragging a new Form Layout into an empty space of the UI Form.

This will create a red rectangle in the UI form. A *Form Layout* always has two columns and it is a standard way design pattern for collecting or displaying information. On the left column add 5 Labels. Make sure you end the drag and drop operation on that side. Change the text on the Labels to **State**, **Current Altitude**, **Current Azimuth**, **Target Altitude** and **Target Azimuth**.

Now, drag and drop a new TaurusState widget, and four TaurusLabel widgets, each to the corresponding cell to the right of its label. Using the *Property Editor* set their model and model Index to:

Label	model	modelIndex
State	cii.oldb:///cut/demoservice/instance1/state	Do not modify
Current Altitude	cii.oldb:///cut/demoservice/instance1/double-vector-current-altaz	0
Current Azimuth	cii.oldb:///cut/demoservice/instance1/double-vector-current-altaz	1
Target Altitude	cii.oldb:///cut/demoservice/instance1/double-vector-target-altaz	0
Target Azimuth	cii.oldb:///cut/demoservice/instance1/double-vector-target-altaz	1

MAL RR Commands

From the Designers *Widget Box* located on the left of the designers window, search or locate the *Grid Layout* entry. Start the drag and drop operation by clicking and dragging a new Grid Layout into an empty space of the UI Form.

For executing commands on the server we will use the *Taurus Command Button*. Drag 5 *Taurus Command Button* to the grid layout. Edit their properties using the *Property Editor* as indicate in the table below:

Command	model	icon –> XDG Theme
Init	zpb.rr://127.0.0.1:12801	send_signal
Enable	zpb.rr://127.0.0.1:12801	send_signal
Disable	zpb.rr://127.0.0.1:12801	send_signal
StartUpdates	zpb.rr://127.0.0.1:12801	send_signal
StopUpdates	zpb.rr://127.0.0.1:12801	send_signal

Add a Label to the Grid Layout and modify its text property to **Reply**.

Add a Line Edit to the Grid Layout and make its **readOnly** property true; and set its **placeholderText** property **Command reply will appear here**.

As an exercise to the reader try to match the design at the top of this document using three GroupBoxes and moving the layouts into them. Remember to create a Splitter as well, so that the TaurusDartBoard is on the left and the GroupBoxes are on the right side.

As a final task, set the MainWindow layout to Vertical, so that all the design grows and shrinks with the MainWindow.

Save the document as `mainwindow_ui.ui`. This name convention will ensure that the generated python document has a *_ui* suffix and is not mixed with the source code that is not generated.

Implementation

This time the implementation is even more simple. OLDB datapoints are automatically handled by their widgets and the OLDB plugin. Similarly the MAL RR connection and requests happens without extra code.

We only need to indicate what to do with the replies.

Main Window Controller

The code is even smaller than *Python Application Tutorial* implementation. In the `mainwindowcontroller.py` several methods have been removed the code as in the design stage actions have been simplified; also example code for long running task has been removed.

An important piece of code to look at is the *_init_gui(self)* method. It also takes care of connecting the **commandExecuted** signal from the Taurus Command Buttons to the *replyLineEdit* widgets *setText* slot.

```
#!/usr/bin/env python3
"""
@file
@ingroup paegui
@copyright ESO - European Southern Observatory

@brief DemoServiceClient GUI implementation
"""

import time

from taurus.external.qt.QtCore import QObject
from taurus.external.qt.QtCore import Slot
from taurus.external.qt.QtCore import Signal
from taurus.external.qt.QtCore import QMetaObject
```

(continues on next page)

(continued from previous page)

```
16 from taurus.external.qt.QtWidgets import QApplication
17 from taurus.external.qt.QtWidgets import QMainWindow
18 from taurus.external.qt.QtWidgets import QWhatsThis
19
20 from taurus.core.util.log import Logger, TraceIt
21 from CutColor import QePalettesModel
22
23 from cut.examples.demo_service_client.mainwindow_ui import Ui_MainWindow  # WAF␣
   ↪will automatically generated this
24 from cut.examples.demo_service_client.applicationconfiguration import␣
   ↪ApplicationConfiguration
25
26
27 class MainWindowController(QMainWindow, Logger):
28     '''
29     Implementation of the mainwindow.ui file.
30
31     Since the UI file indicates that its root is a QMainWindow, then
32     this class should also inherit from it.
33     We should also call explicitly its parent constructors.
34
35     The implementation for this class also includes slots for
36     actions, and management of the closeEvent.
37     '''
38
39     ##########################################################################
40     #                 Initialization and QMainWindow Methods                 #
41     ##########################################################################
42
43     def __init__(self, configuration: ApplicationConfiguration):
44         QMainWindow.__init__(self)
45         Logger.__init__(self, name=QApplication.instance().applicationName())
46         # Construction of UI
47         self.ui = Ui_MainWindow()
48         self.ui.setupUi(self)
49         self._init_gui()
50         self._configuration = configuration
51
52     def _init_gui(self):
53         '''
54         In some cases, widgets are not ready for modification until
55          __init__ is done. This can be workaround by invoking another
56          method at the end of __init__.
```

(continues on next page)

(continued from previous page)

```
57
58          '''
59          self.ui.splitter.setSizes([300,300])
60          self._palettes_model = QePalettesModel()
61          self.ui.menuView.addAction(self._palettes_model.colorSchemeMenu())
62          self.ui.statusbar.addPermanentWidget(self.ui.qeHeartbeat)
63          self.ui.qeBanner.show_info("If you want to understand more of the GUI,␣
↪please click on \"What's This?\" and point to a GUI element.")
64          self.ui.initTaurusCommandButton.commandExecuted.connect(self.ui.
↪replyLineEdit.setText)
65          self.ui.enableTaurusCommandButton.commandExecuted.connect(self.ui.
↪replyLineEdit.setText)
66          self.ui.disableTaurusCommandButton.commandExecuted.connect(self.ui.
↪replyLineEdit.setText)
67          self.ui.startUpdatesTaurusCommandButton.commandExecuted.connect(self.ui.
↪replyLineEdit.setText)
68          self.ui.stopUpdatesTaurusCommandButton.commandExecuted.connect(self.ui.
↪replyLineEdit.setText)
69
70
71      def closeEvent(self, event):
72          '''
73          Not to be confused with actionExit, this method is special. Instead
74          of using signals to know when an application is closed, we can
75          override the closeEvent method from the QApplication class, and
76          determine here what will happen when the window is closed.
77          In this case, we are just forwarding the event to its parent class.
78          This is useful when disconnection from server or prevention of
79          current work needs to be implemented.
80
81          :param event: Event received from Qt event pump
82          :type event: QtCore.QEvent
83
84          '''
85          self.info('Application shutting down...')
86          QMainWindow.closeEvent(self, event)
87
88      ###########################################################################
89      #                    First Tutorial Slots Implementation                  #
90      ###########################################################################
91
92      @Slot()
93      def on_actionExit_triggered(self):
```

(continues on next page)

(continued from previous page)

```
        """
        Slot that auto-connects to the actionExit.

        actionExit is not declared in the code, but in the mainwindow.ui.

        See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-
↪dialogs-with-auto-connect

        It is up to the developer to implement its behaviour. In this case
        we call qApp.exit(), where qApp is a global reference to the
        TaurusApplication or QApplication.
        The actionExit must be manually triggered by the user, through a
        menu, toolbar button.
        """
        self.info('Application shutting down...')
        QApplication.instance().exit()

```

Every time a reply is returned from the server, this connections will put the text in the *replyLineEdit* widget.

And with that the implementation is done. As usual remember to configure compile and install the project using waf:

```
$ waf configure build install
```

Remember to start the server application using **cutDemoService** and execute **cutDemoServiceClient** to see the resulting GUI.

```
$ cutDemoService
$ cutDemoServiceClient
```

4.5 RAD Hello Tutorial

Introduction

This documents teaches the reader how to develop a basic GUI application for the RAD Hello application. This documents assumes you have read and followed both Python Application Tutorial, and the RAD Tutorial, Part 1.

In this tutorial, the reader will learn how to extend an existing application, implement MAL Reply/Request communications and trigger them from the GUI, and show OLDB data points.

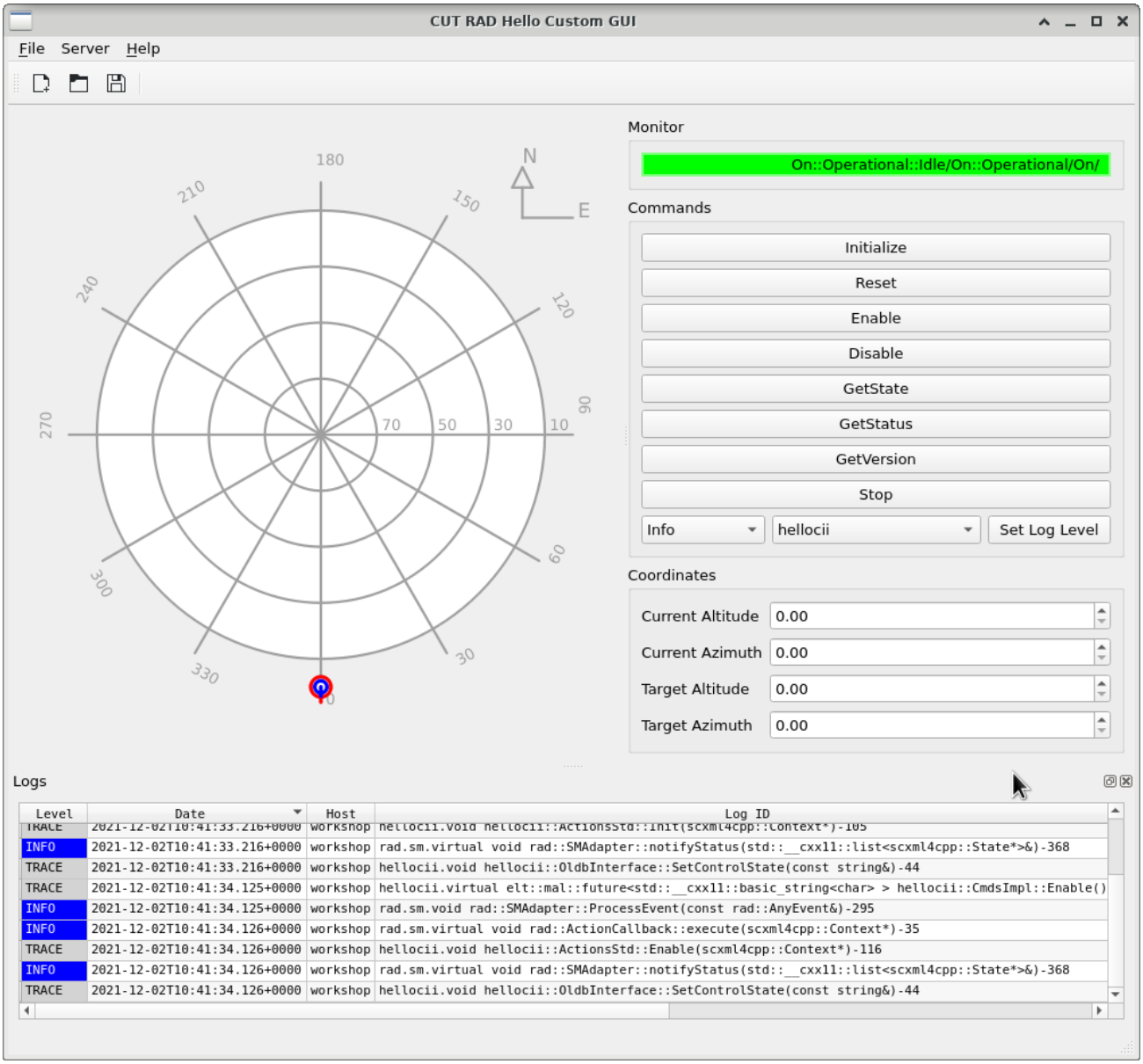


Fig. 63: Final product of this tutorial

Definitions

Python module
A python file, may contain several classes, functions and scripts.

Python package
A directory, and a __init__.py file. This packages contains all other python modules in inside of the directory.

WAF module
A directory with a wscript, that is not a top-level project script.

WAF project
The first wscript of a project. It also must document the external dependencies of the project.

Widget
Display or Control element of a GUI. Widgets present data or information to the user of the GUI, and implement interaction capturing input from the user, making it available for the application to process. Widgets are intended to be a small unit of a GUI, that can be reused.

Composite widget
a widget that is implemented by composing several other smaller widgets. Implementing a composite widget commonly achieves a complex tasks by cooperation between its parts.

Sequential Programming
a programming paradigm in which the flow of control is predetermined by a sequence of instructions.

Asynchronous Programming
a programming paradigm in which the program reacts to events. Different kinds of inputs can be translated into events, such as key strokes, mouse movements, network connection status, battery level, etc.

Note: Asynchronous programming is not restricted by multi-threading, but some long running responses to events certainly benefit from it.

Event Loop/Engine/Pump
Asynchronous Programming needs to implement an additional section of the program that gathers input, creates events, and trigger the responses to those events. One possible implementation of this, and the one that Qt uses, is an Event Loop (See QEventLoop[141]).

[141] https://doc.qt.io/qtforpython-5/PySide2/QtCore/QEventLoop.html

Conventions

Note: Through this document, the author will refer to widgets by name using the following notation: *TextOnTheWidget* kindOfWidget

For example: *Accept* button or *Open File* dialog

Note: When referencing code, like methods or class names, this notation will be used `closeEvent(QEvent *event)`.

Note: Executable programs will be depicted like this **`cut-demo-service`**.

Download, Compilation, and Running

This tutorial builds an application for the RAD Tutorial Part 1, based on the final product of the CUT Python Application Tutorial. In order to avoid much confusion, we suggest to download a final example of this tutorial from CUT repository.

From GIT repository	From Webpage
`git clone https://gitlab.eso.org/ecos/cut`	`wget https://www.eso.org/~ahoffsta/code/v1.2.2/cut.tar.gz tar xfz cut.tar.gz`

```
mv cut/examples/rad_hello ./
rm -rf cut
```

After reading this, the user may want to replicate the experience by their own, as excercise. In that case, we recommend to create a new template for the Python Application, and apply the same modification this tutorial introduces.

```
git clone https://gitlab.eso.org/ecs/cut
cookiecutter cut/templates/python_application

organization [European Southern Observatory]:
organization_domain [eso.org]:
copyright [ESO - European Southern Observatory]:
author []: ESO Developer
project_name [Template Python Application]: CUT RAD Hello GUI
project_name_slug [cut-rad-hello-gui]:
```

(continues on next page)

(continued from previous page)

```
project_version [0.0.1-dev]:
app_name [TemplatePyApp]: cut_rad_hello
app_exe_name [cut_rad_hello]:
waf_module [cut_rad_hello]:
pkg_name [cut_rad_hello]:
```

The directory we will be using for this tutorial is the `hello_rad`, and it contains the code for a GUI application that is compatible with the RAD Tutorial Part 1. This is a WAF project on its own, so it can be build in an independent manner from the rest of Control UI Toolkit. In order to do so the reader must execute:

```
cd hello_rad
waf configure build install
```

And to execute the application:

```
cut_rad_hello
```

In order to use the cut_rad_hello, you must either have started the RAD hellocii example, or select from the Top Menu Bar the entry *Server* → *Start Server Process*. This menu entry will open the **`hellocii`** server application for you.

Project Structure

The final structure of this WAF project is drawn below.:

```
rad_hello
├── cut_rad_hello
│   ├── src
│   │   ├── cut_rad_hello
│   │   │   ├── applicationwindow.py
│   │   │   ├── __init__.py
│   │   │   ├── mainwindow.ui
│   │   │   └── stdcmdsfacade.py
│   │   └── cut_rad_hello.py
│   └── wscript
└── wscript
```

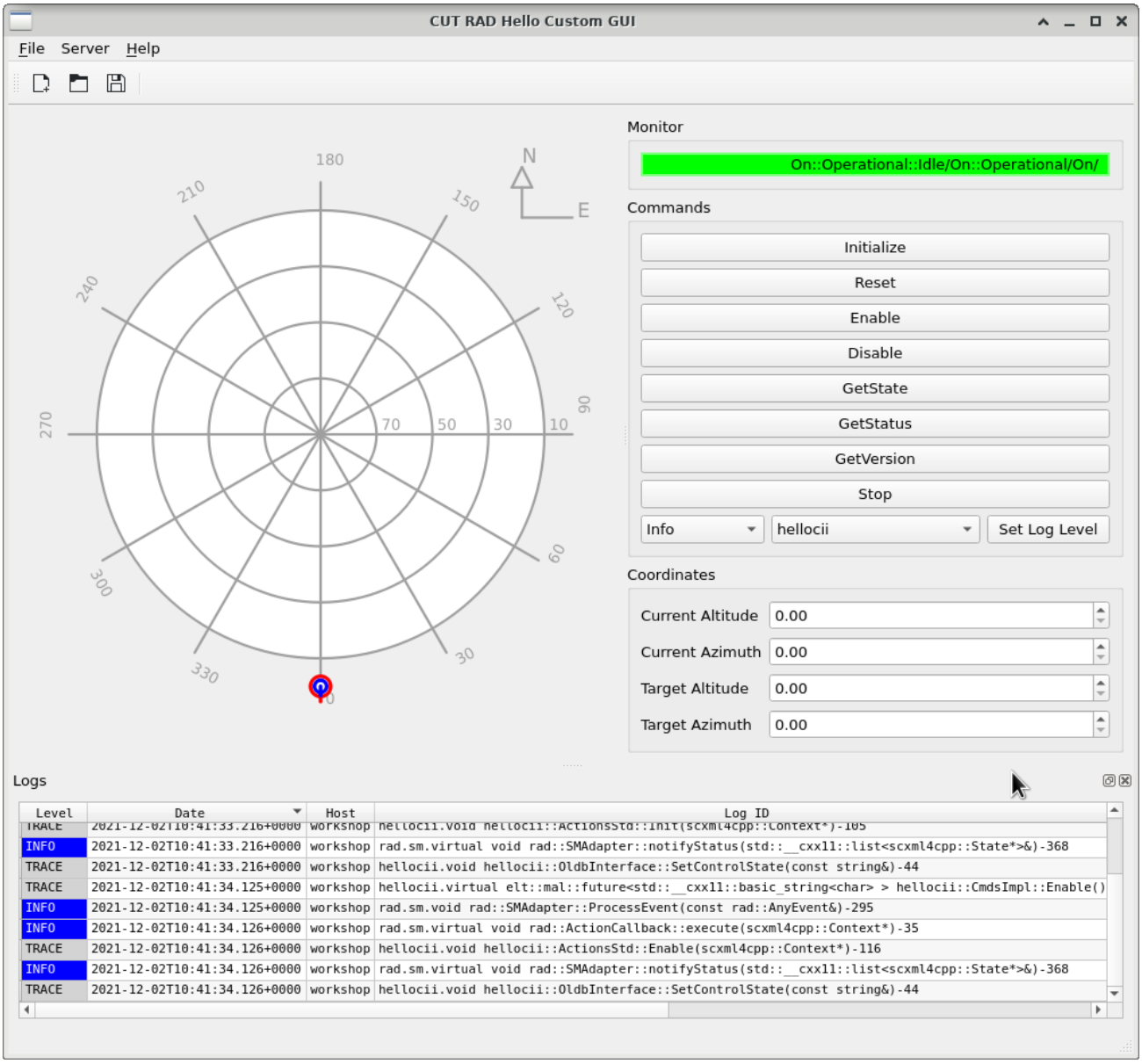


Fig. 64: CUT RAD Hello GUI: We will update an application to interface the hellocii example code from RAD.

Project Definition (wscript)

The first file we should take a look is `hello_rad/wscript`:

```
from wtools.project import declare_project

def configure(cnf):
    cnf.check_wdep(wdep_name='hello.hellociiif-python', uselib_store='hello.
↪hellociiif-python', mandatory=True)
    # We check which version of qt6 is requested and set the proper feature for␣
↪later
    if hasattr(cnf, "want_qt6") and cnf.want_qt6 is True:
        cnf.check_python_module('PySide6', condition="ver >= num(6, 0, 0) and ver
↪<= num(7, 0, 0)")
    else:
        cnf.check_python_module('PySide2', condition="ver >= num(5, 14, 1) and ver
↪< num(6, 0, 0)")
    cnf.check_python_module('taurus')
    cnf.check_python_module('tauruscii')
    cnf.check_python_module('elt.cut')
    cnf.check_python_module('elt.pymal')
    cnf.check_python_module('CutWidgets')

declare_project('cut-rad-hello-gui',
                version='3.4.1-rc1',
                requires=['cxx',
                          'python',
                          'pyqt'],
                recurse='cut_rad_hello',
)
```

The project is now named `cut-rad-hello-gui`, and its recurse list has change also to reflect the new project structure. The main difference is the new `check_wdep` method invocation in the `configure` step. This is used to import the dependency of the ICD defined in the RAD tutorial.

The ICD is a very important construct, as it defined the communication protocol between the server (hellocii application) and the client (cut_rad_hello GUI application). We should only need to depend on that library.

Aside from that change, the directory structure include one new file: `stdcmdsfacade.py`. This class is in charge of implementing communication using the ICD library we just check.

Module Definition (cut_rad_hello/wscript)

Though a short difference, the module definition has an important one:

```
from wtools import module

# The declare_pyqtprogram instruction will automatically find
# the QT Resources, and the UI files, and run the necessary
# commands to produce the python versions of them.

module.declare_pyqtprogram(target='cut_rad_hello',
                           use='hello.hellociiif-python')
```

This line indicates that this module uses the ICD definition from the `hellociiif` module. Without it, waf indicates failure during the build stage.

Entry Point Python Script (cut_rad_hello/src/cut_rad_hello.py)

The entry point python script `cut_rad_hello.py` is the one that creates a Qt GUI Application. It remain mostly the same as in the previous tutorial, except that new command line options for the parser has been added. This example add options for host and port of the server application, and to set the logging level of the example GUI.

```
 parser.add_option( '-p', '--port', dest='port', help='Changes the network port␣
↪used to connect to the RAD Hello Application',
                    default=12081)
 parser.add_option("-H", '--host', dest='host', default='localhost', help='Changes␣
↪the network host used to connect to the RAD Hello Application')
 parser.add_option("-v", '--verbose',dest="verbose", action="store_true",␣
↪default=False,
                       help='Changes the LogLevel of the application to Debug.␣
↪Output more information on application procedures.')
 parser.add_option("-V", '--extra-verbose',dest="extraverbose", action="store_true",
↪ default=False,
                    help='Changes the LogLevel of the application to Trace. Outputs␣
↪everything.')
```

Only the extra logging options are used in this file to set up the proper logging level.

```
app = TaurusApplication(
    app_name = 'CUT RAD Hello GUI',
    app_version = '0.0.1-dev',
    org_name = 'European Southern Observatory',
    org_domain = 'eso.org',
```

(continues on next page)

(continued from previous page)

```
    cmd_line_parser=parser
    )
if(app.get_command_line_options().verbose):
    logger = Logger(app.applicationName())
    logger.setLogLevel(Logger.Debug)
if(app.get_command_line_options().extraverbose):
    logger = Logger(app.applicationName())
    logger.setLogLevel(Logger.Trace)
```

Since the TaurusApplication is already created at this point, we can get the Logger objects using the taurus class Logger[142].

Please remember that at this point the flow of control is still sequential. When the `app.exec_()` method is invoked, we surrender control of the application to Qt's QEventLoop, and we switch to an Asynchronous paradigm. Everything from now on is governed by events.

Graphical Design Updates (cut_rad_hello/src/cut_rad_hello/mainwindow.ui)

The updates for this examples are several:

- A new menu called Server
- Deleted QTextEdit[143] widget
- A new QGroupBox[144] that holds a QLabel[145] and a TaurusLabel[146]
- A second QGroupBox[147] that holds 8 new QPushButton[148] for commands in the StdCmds Interface and a nested Horizontal Layout with 2 QComboBox[149] and a QPushButton[150]
- A third QGroupBox[151] for the Coordinates

You can see the final design on the image below.

We will start the Qt Designer, by executing this command in a terminal:

```
designer cut_rad_hello/src/cut_rad_hello/mainwindow.ui
```

We will start with the changes following a top to bottom visual order.

[142] https://taurus-scada.org/devel/api/taurus/core/util/_Logger.html
[143] https://doc.qt.io/qt-5/qtextedit.html
[144] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QGroupBox.html
[145] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[146] https://taurus-scada.org/devel/api/taurus.qt.qtgui.display-TaurusLabel.html#taurus.qt.qtgui.display.TaurusLabel
[147] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QGroupBox.html
[148] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html
[149] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QComboBox.html
[150] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html
[151] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QGroupBox.html

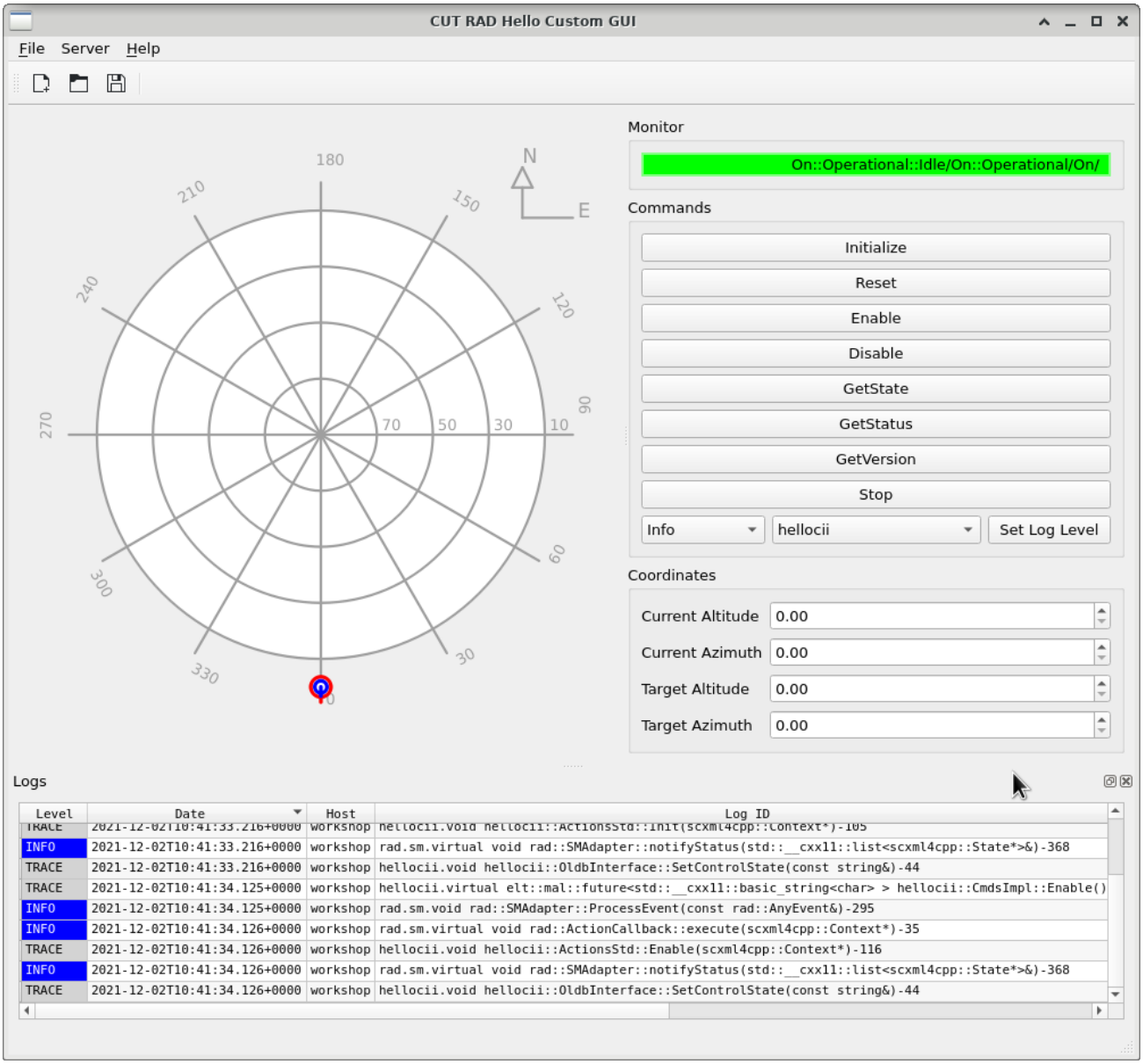


Fig. 65: CUT RAD Hello GUI - Updates on the GUI.

New Menu: Server

This menu can be created by clicking on the menu entry called *Type Here*. Type `&Server` and press `Enter`. Once done, drag and drop it to in between *File* and *Help* menu entries.

Now that the top-level menu is created, we can create its entries: Please create them using the *Action Editor* Dock Widget in the designer.

Text	Object Name	Tooltip	Icon Theme	Short-cut
&Connect to Server	actionServerConnect	Connect to Server	network-receive	
&Disconnect from Server	actionServerDiscon-nect	Disconnect from Server	network-offline	
&Start Server Process	actionServerStart	Start Server	network-server	
&Exit Server	actionServerExit	Executes Exit on Server	application-exit	

1. Drag and drop the *Connect to Server* and *Disconnect from Server* Actions to the menu.
2. Create a new separator in the menu.
3. Drag and drop the *Start Server* and *Exit Server* Actions to the menu.

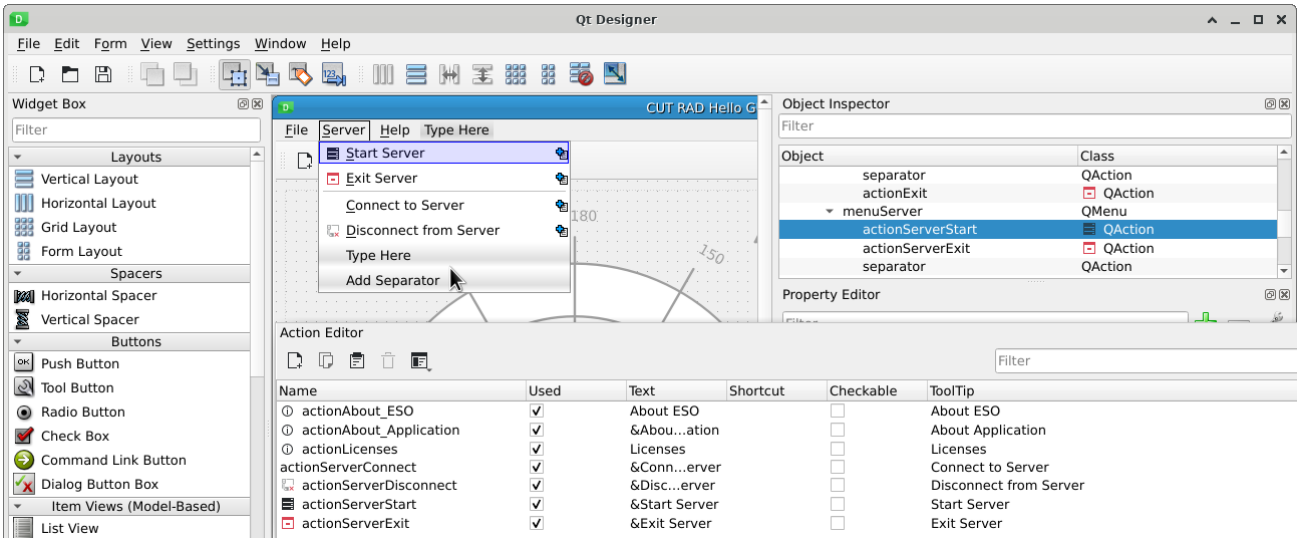


Fig. 66: New Actions and Menus.

Deleted QTextEdit[Page 150, 152] widget

The QTextEdit[153] that holds the long text can be deleted just like that. Left click on the widget, and press the `Delete` key.

A new QGroupBox

We start by dragging a new QGroupBox[154] from the *Widget Toolbox*. This widget is located in the *Containers* section. Drop it in the Vertical Layout that holds the *Coordinates*, above of the Layout that is already holding the coordinates widgets. Select it, and change its **title** property to **Monitor**.

From the *Widget Toolbox* drag two Label to the QGroupBox[155] that was just created. Assign a Form Layout to it.

Change the **text** property on the first QLabel[156] to **State:**. For the second one, we will use a technique called **Widget Promotion**.

Widget Promotion is used to change the widget from one recognized by the Designer, to another that though is available, it is not recognized by it (does not have a designer plugin).

We will use it to declare that a QLabel[157] is instead a TaurusLabel[158]. This is very useful while developing, because it can be used to add widget that are part of this application, but we do not want to create a full library that includes widgets, designer plugin and bindings for it. We can also use it while developing widgets, to make sure the widget works properly even before we have created the plugin library for it.

To **promote** the second QLabel[159], we right click on it, and select *Promote to...* context menu entry. A new dialog window named *Promoted Widgets* will appear. Here we can add new widgets classes. These are custom widgets that the designer is not currently aware of.

To create a new entry, we need to know from which Qt class it inherits, its class name, and its header file. Write the following entries:

- Base class name: QLabel
- Promoted class name: TaurusLabel
- Header file: taurus.qt.qtgui.display

Then click on the *Add* Button, right click the new entry, and click on the *Promote* button in the bottom of the *Promoted Widgets* dialog.

[152] https://doc.qt.io/qt-5/qtextedit.html
[153] https://doc.qt.io/qt-5/qtextedit.html
[154] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QGroupBox.html
[155] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QGroupBox.html
[156] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[157] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[158] https://taurus-scada.org/devel/api/taurus.qt.qtgui.display-TaurusLabel.html#taurus.qt.qtgui.display.TaurusLabel
[159] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html

Now this label is a TaurusLabel. As we already know, the most important property of a TaurusLabel[160] is the **model**. We can use the *Property Editor* to create properties the Qt Designer is not aware of. Use the *Plus* button on the *Property Editor* and select *String...*. For Property Name use **model**. Then repeat for a second property but *Bool...* and name it **autoTrim**.

Browse to the end of the *Property Editor*, and set the properties to:

- model: cii.oldb:/elt/hellocii/mon/state
- autoTrim: True

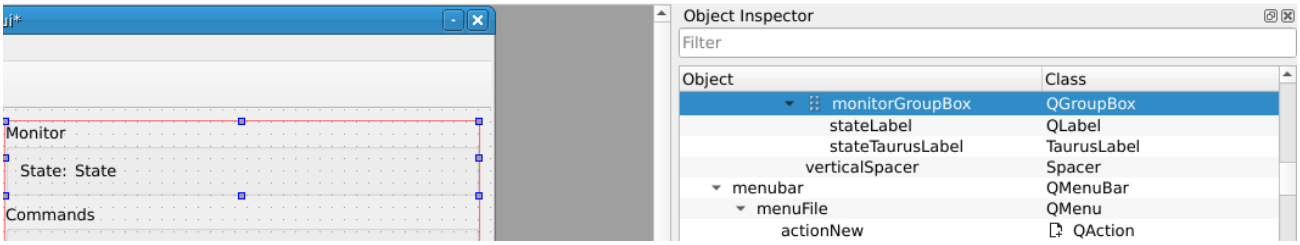


Fig. 67: First Group Box Added.

A second QGroupBox

Drag a new QGrouBox from the *Widget Toolbox*. Drop it in the Vertical Layout, below the one from before. Change its **title** property to **Commands**.

Set its layout to Vertical.

Drag and drop 8 QPushButton[161] to it. Change their **text** and **objectName** properties as indicated:

text property	objectName property
Initialize	initPushButton
Reset	resetPushButton
Enable	enablePushButton
Disable	disablePushButton
GetState	getStatePushButton
GetStatus	getStatusPushButton
GetVersion	getVersionPushButton
Stop	stopPushButton

Drag a new Horizontal Layout from the *Widget Toolbox* to the QGroupBox we are working on, below all the other buttons. The reader will notice that this layout appear as a thin red line. Dropping new widget in it requires very fine precision with the mouse, or to use the *Object Inspector*. Search for the

[160] https://taurus-scada.org/devel/api/taurus.qt.qtgui.display-TaurusLabel.html#taurus.qt.qtgui.display.TaurusLabel
[161] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html

QGroupBox in question and drag and drop a QComboBox[162] from the *Widget Toolbox* to the *Object Inspector* entry.

Double click on the QComboBox[163] and a new *Edit Combobox* dialog will appear. You can create the entries for this Combo Box here. Please create the following entries:

- Trace
- Debug
- Info
- Warn
- Error
- Fatal

Then change the Combo Box property **currentIndex** to **2**. This will make *Info* the default entry.

Repeat the creation of a new Combo Box. This one will identify the Loggers.

- hellocii
- rad
- rad.sm
- scxml4cpp
- malZpbClientAsyncImpl
- malZpbServer

Leave the **currentIndex** to its default, which should be **hellocii**.

Add a new QPushButton[164] with **objectName** to the same Horizontal Layout, set its as **setLogLevelPushButton** and change its **text** property to **Set Log Level**.

A third QGroupBox

Drag a new QGroupBox[165] from the *Widget Toolbox*. Drop it in the Vertical Layout, below the one from before. Change its **title** property to **Coordinates**.

Drag the Form Layout from the previous tutorial inside of this new Layout. This can be a bit tricky, because you can only drag and drop a Layout by dragging its line or by the empty space between the widgets it coordinates.

With this, we are ready with all UI modifications.

[162] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QComboBox.html
[163] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QComboBox.html
[164] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html
[165] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QGroupBox.html

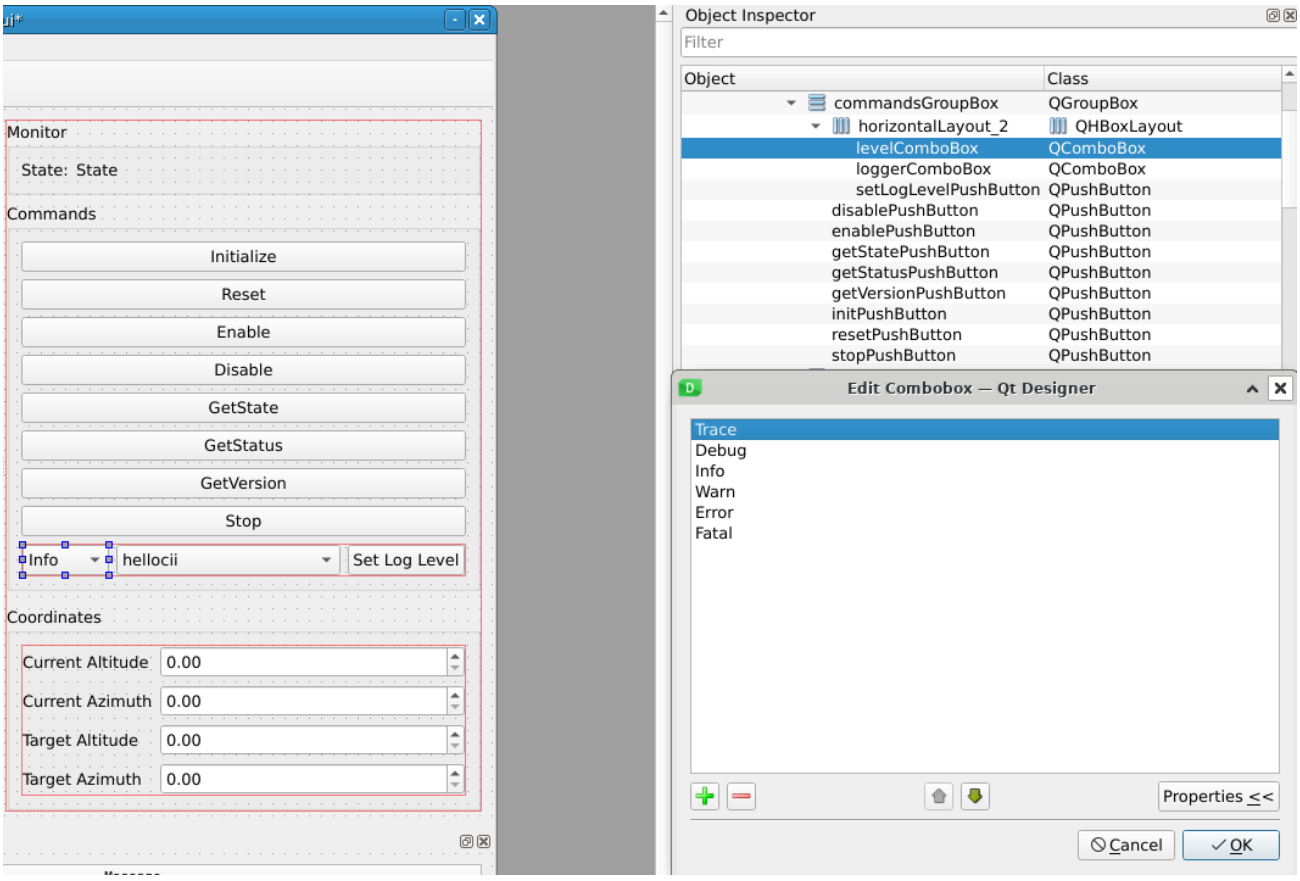


Fig. 68: Second Group Box Added.

StdCmds Interface (cut_rad_hello/src/cut_rad_hello/stdcmdsfacade.py)

`stdcmdsfacade.py` is a new file, and will encapsulate all communications to the `hellocii` application. This class will interface against a GUI, so it is imperative that we use Signals and Slots, and that we connect those to our UI.

Imports

The most important section are the `ModHellociiif` imports. There are the library produced by the ICD compilation for python. MAL sets the name as: Mod<Package_name>.<Package_name>. Please notice that MAL uses uses the <PackageName> but uppercases the first letter. This python module contains at this level the structures and exceptions definitions, and at the Mod<PackageName>.<PackageName>.<interfaceName> level the <StdCmdsAsync> class, which we use in this file.

```
from ModHellociiif.HellociiifStdCmds import StdCmdsAsync
from ModHellociiif.Hellociiif import LogInfo
```

Class Definition and Initializator

The class `StdCmdsFacade` inherits from MalAdapter[166]. MalAdapter[167] is in charge of the connection logic and exposes the connection status through a Property[168], which in turn has a Slot to change the connection state, and a Signal[169] that indicates if the connection status has changed.

In short, it exposes a MAL Interface in a Qt compatible manner.

```
class StdCmdsFacade(MalAdapter):

    def __init__(self, uri: str):
        """
        Initializator for a new `StdCmdsFacade`. It inherits from MalAdapter,
        which provides all the connection logic ready to be used.

        We mainly interact with MalAdapter through `get_interface()`
        `get_mal()` and `get_factory()`. We do not keep references to them.

        This class exposes Slots that use the elt.cut.task.Task class to
        execute long-running operations in a separate thread. Each Slot
        is a method in the hellociiif interface.
```

(continues on next page)

[166] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classcomms_1_1maladapter_1_1MalAdapter.html
[167] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classcomms_1_1maladapter_1_1MalAdapter.html
[168] https://doc-snapshots.qt.io/qtforpython/PySide2/QtCore/Property.html
[169] https://doc.qt.io/qtforpython-5/PySide2/QtCore/Signal.html

(continued from previous page)

```
        :param uri: URI for the hellocii MAL reply request endpoint
        '''
        MalAdapter.__init__(self, uri, StdCmdsAsync)
```

We call explicitly its parent initializator and pass the URI of the **hellocii** endpoint, and the class of the MAL generated interface. This is enough information for the MalAdapter[170] to create the connection to the endpoint.

Interface Methods with no Arguments

The implementation of these methods is simple, lets see for example the one corresponding to the `Init()`.

```
@TraceIt()
@Slot()
def init(self, result_slot=None, error_slot=None):
    '''
    This method executes the Init command on the server.
    It uses the Asynchronous interface.
    It will wait or the future object to has its result, and then
    it will call the slot function with the result as argument.

    Parameters
    ----------
    result_slot: QtCore.QSlot
        Slot method that will be triggered when the result is available.
    error_slot: QtCore.QSlot
        Slot method that will be triggered when an error is detected.

    Return
    -------
    None
    '''
    if(not self.isConnected()):
        self.error('Application is not connected to TrkLsv. \
                    Please connect first.')
        return
    task = Task(self.get_interface().Init)
    if result_slot is not None:
        task.signals.result.connect(result_slot)
```

(continues on next page)

[170] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classcomms_1_1maladapter_1_1MalAdapter.html

(continued from previous page)

```
    if error_slot is not None:
        task.signals.error.connect(error_slot)
    task.signals.finished.connect(self.thread_complete)
    task.start()
```

The decorators (`@TraceIt` and samp:*@Slot*) we use indicate that this has to be traced, and that it is a Qt Slot. Notice that this Slot[171] has not arguments.

This `init()` method check first if the class `isConnected()` before proceeding.

Then it creates a Task[172] that calls the `Init()` from the interface its parent got for us.

At this point, the reader may wonder where is the connection handled? The logic is provided by MalAdapter[173], and it is triggered at the start of the application in `applicationwindow.py`. But we will see more details on that later.

In this example, we are setting three of the Task[174] Signals. We are using the `result_slot`, the `error_slot` and the `finished_slot`.

At the end of the method, we `start()` the task. Everything else is handled in an asynchronous manner by the slots we specified.

Interface Method with Argument

This method is the only one that uses arguments in the `Hellociiif` Interface. We want to shield the rest of the code from Structures that are specific to the Interface module, so this Slot accepts strings as arguments, aside from the Slots for the feedback on the operation.

```
@Slot(str, str)
def set_log_level(self, log_level: str, logger: str, result_slot=None, error_
↪slot=None):
    '''
    This method executes the SetLogLevel command on the server.
    It uses the Asynchronous interface.
    It will wait or the future object to has its result, and then
    it will call the slot function with the result as argument.

    Parameters
    ----------
    log_level: str
        Log Level
```

(continues on next page)

[171] https://doc.qt.io/qtforpython-5/PySide2/QtCore/Slot.html
[172] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtask_1_1task_1_1Task.html
[173] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classcomms_1_1maladapter_1_1MalAdapter.html
[174] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtask_1_1task_1_1Task.html

(continued from previous page)

```
    logger: str
        Logger to be affected by this command
    result_slot: QtCore.QSlot
        Slot method that will be triggered when the result is available.
    error_slot: QtCore.QSlot
        Slot method that will be triggered when an error is detected.

    Return
    -------
    None

    '''
    if(not self.isConnected()):
        self.error('Application is not connected to TrkLsv. \
                    Please connect first.')
        return
    loginfo = self.get_mal().createDataEntity(LogInfo)
    loginfo.setLevel(log_level)
    loginfo.setLogger(logger)
    task = Task(self.get_interface().SetLogLevel, loginfo)
    if result_slot is not None:
        task.signals.result.connect(result_slot)
    if error_slot is not None:
        task.signals.error.connect(error_slot)
    task.signals.finished.connect(self.thread_complete)
    task.start()
```

`SetLogLevel()` uses a `LogInfo` Structure. We need to use the MAL to create a data entity of the specific type. Once we have it, we can fill its members with the required strings.

We pass this structure as the second argument in the initializator of the Task[175] class.

UI Implementation (cut_rad_hello/src/cut_rad_hello/applicationwindow.py)

Imports

Lines 3 and 10 change. We have added QProcess[176] to handle the start of the hellocii application, and QEventLoop[177] for special circumstance on closing of the application.

On line 10 we add the `StdCmdsFacade` class we have just programmed.

[175] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtask_1_1task_1_1Task.html
[176] https://doc.qt.io/archives/qtforpython-5.14/PySide2/QtCore/QProcess.html
[177] https://doc.qt.io/qtforpython-5/PySide2/QtCore/QEventLoop.html

```
import time

from PySide2.QtCore import QObject, QMetaObject, Slot, Signal, SIGNAL, SLOT, QTimer,
↪ QProcess, QEventLoop
from PySide2.QtWidgets import QMainWindow
from PySide2.QtGui import QWhatsThis

from elt.cut.task import Task
from taurus.core.util.log import Logger, TraceIt
from taurus import Manager
from cut_rad_hello.stdcmdsfacade import StdCmdsFacade
from cut_rad_hello.mainwindow import Ui_MainWindow # WAF will automatically␣
↪generated this
```

Initialization

We create a new object called `self.server_facade`. This is a an object from the `StdCmdsFacade` class. In order to properly construct it, we need to pass it the URI of the **`hellocii`** endpoint.

We have also added a new `self.server_process` object from the QProcess[178] class. This will keep track of a process for us while offering Signal/Slots in order to do so.

```
def __init__(self):
    QMainWindow.__init__(self)
    Logger.__init__(self,name=qApp.applicationName())
    # Construction of UI
    self.ui = Ui_MainWindow()
    self.ui.setupUi(self)
    MY_SERVER_URL = qApp.get_command_line_options().host
    MY_SERVER_PORT = qApp.get_command_line_options().port
    uri = 'zpb.rr://' + str(MY_SERVER_URL) + ':' +\
          str(MY_SERVER_PORT) + '/StdCmds'
    self.server_facade = StdCmdsFacade(uri)
    self.server_process = QProcess(self)
    self._init_gui()
```

In the `_init_gui` method we have added a bit of customization to our GUI, and recommended, we put it in this method. We start the application by setting some default values. First of all, the `self.ui.centralwidget` (line 8) is set to be disabled. This will disable interaction of most of the GUI. Particularly useful when the facade is not connected to the server, which is the starting case.

We also disable the `actionServerExit` QAction[179], which automatically disables its entry in the Menu.

[178] https://doc.qt.io/archives/qtforpython-5.14/PySide2/QtCore/QProcess.html
[179] https://doc.qt.io/qt-5/qaction.html

This goes in the same line of thought as the previous change. You cannot execute an `Exit` command on a server you are not yet connected to.

On line 10, we connect the Signal `self.server_facede.connectionChanged(bool)` to Slot `self.on_server_connection_changed(bool)`. This will handle the enabling and disabling of widgets and menu entries when the connection status is detected to have change.

On line 12 and 13, we connect started and finished Signals to slots we implement that handled again, enabling and disabled menu entries to avoid presenting inconsistent options to the user.

```
1  def _init_gui(self):
2      '''
3      In some cases, widgets are not ready for modification until
4       __init__ is done. This can be workaround by invoking another
5       method at the end of __init__.
6
7      '''
8      self.ui.centralwidget.setEnabled(False)
9      self.ui.actionServerExit.setEnabled(False)
10     QObject.connect( self.server_facade, SIGNAL('connectionChanged(bool)'),
11                      self, SLOT('on_server_connection_changed(bool)'))
12     self.server_process.started.connect(self.on_server_process_started)
13     self.server_process.finished.connect(self.on_server_process_finished)
14     QTimer.singleShot(250, lambda: self.on_actionServerConnect_triggered())
15     pass
```

At the end of this, we use QTimer[180] to automatically call a slot when a timer has ended. This will start after our GUI has been created and shown to the user. We avoid starting the connection while initializing the GUI to present it promptly to the user.

closeEvent()

We override the usual behavior defined by Qt library by redefining this method by adding more instructions before invoking the parent class `closeEvent`. This will ensure that the normal behaviour is not lost (otherwise, the window will not close).

We start a new QEventLoop[181], and set a timeout of 1 second. At that point, the Event Loop will quit. When closing windows we reach a very particular condition, where we cannot keep a QTimer[182] tied to this class, because it can be destroyed while closing the window. In fact, as usual, while destroying or closing windows, one should avoid the creation of object tied to the lifecycle of the very same object being destroyed.

We need the timer in order to give the `on_actionServerDisconnect_triggered` method to perform its duties, but still having a timeout for the operation. This gives us that.

[180] https://doc.qt.io/qtforpython-5/PySide2/QtCore/QTimer.html
[181] https://doc.qt.io/qtforpython-5/PySide2/QtCore/QEventLoop.html
[182] https://doc.qt.io/qtforpython-5/PySide2/QtCore/QTimer.html

You can see that we start another Event Loop by the name of the `loop.exec_()` method.

```
def closeEvent(self, event):
    '''
    Not to be confused with actionExit, this method is special. Instead
    of using signals to know when an application is closed, we can
    override the closeEvent method from the QApplication class, and
    determine here what will happen when the window is closed.
    In this case, we are just forwarding the event to its parent class.
    This is useful when disconnection from server or prevention of
    current work needs to be implemented.

    :param event: Event received from Qt event pump
    :type event: QtCore.QEvent

    '''
    self.info('Application shutting down...')
    loop = QEventLoop()
    self.server_facade.connectionChanged.connect(loop.quit)
    self.on_actionServerDisconnect_triggered()
    QTimer.singleShot(1*1000, lambda: loop.quit())
    self.info('Waiting for Hellocii client to shutdown or timeout of 1 second...')
    loop.exec_()
    QMainWindow.closeEvent(self, event)
```

Server (dis)connection Slots

The first and second Slot in this code snippet reach to the Action defined in the UI. The first starts with the `server_facade.setConnection(True)` method call a request to connect to the **`hellocii`** server. The second one does the opposite. Both also take care of enable/disable certain widgets/action of the UI so that erroneous operations are avoided.

```
@TraceIt()
@Slot()
def on_actionServerConnect_triggered(self):
    '''
    Slot that auto-connects to the QAction actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
↪with-auto-connect
    '''
    self.server_facade.setConnection(True)
    self.ui.actionServerConnect.setEnabled(False)
    self.ui.actionServerDisconnect.setEnabled(True)
```

(continues on next page)

(continued from previous page)

```
    self.ui.actionServerExit.setEnabled(True)

@Slot()
def on_actionServerDisconnect_triggered(self):
    '''
    Slot that auto-connects to the QAction actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
↪with-auto-connect
    '''
    self.server_facade.setConnection(False)
    self.ui.actionServerConnect.setEnabled(True)
    self.ui.actionServerDisconnect.setEnabled(False)
    self.ui.actionServerExit.setEnabled(False)


@TraceIt()
@Slot(bool)
def on_server_connection_changed(self, state):
    '''
    Slot that auto-connects to the QAction actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
↪with-auto-connect
    '''
    self.ui.centralwidget.setEnabled(state)
    self.ui.actionServerConnect.setEnabled(not state)
    self.ui.actionServerDisconnect.setEnabled(state)
    self.ui.actionServerExit.setEnabled(state)
```

The third method has been connected during the initialization to the `connectionChanged` Signal. This means that every time the connection state is True (connected) or False (disconnected), this Slot is called with that boolean. It then proceeds to set different state for Action/Widgets so that erroneous operations are avoided.

Slots for methods in the interface

There are 8 Slots that react to buttons being pressed, and one for an Action being triggered. They are quite simple, and just invoke the appropriate method in the `server_facade` object.

```
1 @TraceIt()
2 @Slot()
3 def on_initPushButton_clicked(self):
```

(continues on next page)

(continued from previous page)

```
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
↪with-auto-connect
    '''
    self.server_facade.init()

@TraceIt()
@Slot()
def on_resetPushButton_clicked(self):
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
↪with-auto-connect
    '''
    self.server_facade.reset()

@TraceIt()
@Slot()
def on_enablePushButton_clicked(self):
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
↪with-auto-connect
    '''
    self.server_facade.enable()

@TraceIt()
@Slot()
def on_disablePushButton_clicked(self):
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
↪with-auto-connect
    '''
    self.server_facade.disable()
    pass

@TraceIt()
```

(continues on next page)

(continued from previous page)

```
@Slot()
def on_getStatePushButton_clicked(self):
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-with-auto-connect
    '''
    self.server_facade.get_state(result_slot=lambda x: self.info("Return: {}".format(x)))
    pass

@TraceIt()
@Slot()
def on_getStatusPushButton_clicked(self):
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-with-auto-connect
    '''
    self.server_facade.get_status(result_slot=lambda x: self.info("Return: {}".format(x)))
    pass

@TraceIt()
@Slot()
def on_getVersionPushButton_clicked(self):
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
    actionServerConnect is not declared in the code, but in the mainwindow.ui.
    See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-with-auto-connect
    '''
    self.server_facade.get_version(result_slot=lambda x: self.info("Return: {}".format(x)))
    pass

@TraceIt()
@Slot()
def on_stopPushButton_clicked(self):
    '''
    Slot that auto-connects to the QPushButton actionServerConnect.
```

(continues on next page)

(continued from previous page)

```
80      actionServerConnect is not declared in the code, but in the mainwindow.ui.
81      See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
    ↪with-auto-connect
82      '''
83      self.server_facade.stop()
84      pass
85
86  @TraceIt()
87  @Slot()
88  def on_actionServerExit_triggered(self):
89      '''
90      Slot that auto-connects to the QAction actionServerConnect.
91      actionServerConnect is not declared in the code, but in the mainwindow.ui.
92      NOTE: This is different from others slots. It uses a QAction instead of a␣
    ↪QPushButton
93      See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
    ↪with-auto-connect
94      '''
95      self.server_facade.exit()
96      pass
97
98  @TraceIt()
99  @Slot()
100 def on_setLogLevelPushButton_clicked(self):
101     '''
102     Slot that auto-connects to the QPushButton actionServerConnect.
103     actionServerConnect is not declared in the code, but in the mainwindow.ui.
104     See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
    ↪with-auto-connect
105     '''
106     self.server_facade.set_log_level(
107         self.ui.levelComboBox.currentText(),
108         self.ui.loggerComboBox.currentText(),
109         result_slot=lambda x: self.info("Return: {}".format(x)))
110     pass
```

The `on_actionServerExit_triggered` is auto-connected to a QAction[183] instead of a QPushButton[184].

`on_setLogLevelPushButton_clicked` is special, as it needs to collect arguments for the method. It is important to establish a line between UI and domain/middleware. Here is an example where it is drawn. This file tried to concern itself only with UI elements. We pass the level and logger as strings

[183] https://doc.qt.io/qt-5/qaction.html
[184] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html

to the `self.server_facade` method.

Starting Hellocii process

This are auxiliary methods that will make our like more easy. The first one is an auto-connected slot, that react to a QAction[185] (also located in the Top-Level Menubar Entry *Server* → *Start Server Process*). This will start a process using the QProcess[186] object we created during initialization.

Also during initialization, we have connected two signals from the QProcess[187] object, to the last two Slots. When the process is detected to be started, and when it finishes, the slots will enable Actions so that erroneous operations are avoided.

```
@TraceIt()
@Slot()
def on_actionServerStart_triggered(self):

    if (self.server_process.state() == QProcess.NotRunning):
        self.warning("Hellocii will be started up in a xfce4-terminal" )
        self.server_process.start("xfce4-terminal",["-x","hellocii","-c", "hellocii/config.yaml", "-l", "DEBUG"])
    else:
        self.warning("Hellocii already running" )

@TraceIt()
@Slot()
def on_server_process_started(self):
    self.ui.actionServerStart.setEnabled(False)
    self.ui.actionServerExit.setEnabled(True)

@TraceIt()
def on_server_process_finished(self, code):
    self.ui.actionServerStart.setEnabled(True)
    pass
```

[185] https://doc.qt.io/qt-5/qaction.html
[186] https://doc.qt.io/archives/qtforpython-5.14/PySide2/QtCore/QProcess.html
[187] https://doc.qt.io/archives/qtforpython-5.14/PySide2/QtCore/QProcess.html

RAD Tutorial Second Part

You can see the final product of this section of the workshop in the CUT repository. Please download it from:

From GIT repository	From Webpage
`git clone https://gitlab.eso.org/ecs/cut`	`wget https://www.eso.org/~ahoffsta/code/v1.2.2/cut.tar.gz tar xfz cut.tar.gz`

```
mv cut/examples/rad_hello_custom ./
rm -rf cut
```

You can run it by building it and installing:

```
cd cut
waf configure build install
cut_rad_hello_custom
```

As the previous example, you can start the **`hellocii`** before, or by using the Top Menu Entry *Server* → *Start Server*, and then *Server* → *Connect to Server* .

The second RAD tutorial adds the `Preset` command that executes an Activity:

- The `Preset` command itself accepts a `TelPosition` structure as argument (which has two members: ra and dec).
- The Activity publishes four OLDB data points: a pair of numbers that represent the Actual coordinates of a Telescope and a pair that represents the Target coordinates.

To allow the user access to the Preset Command and the OLDB Data points, we will introduce the following modifications:

- Four new QLabel[188] and two TaurusDmsLabel[189] and TaurusHmsLabel[190] to present the OLDB data points.
- For the Preset command, we will add two QDoubleSpinBox[191] to enter the coordinates, two QLabel[192] to indicate which coordinate (ra/dec), and a QPushButton[193] to trigger the command.

[188] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[189] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusDmsLabel_1_1TaurusDmsLabel.html
[190] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusHmsLabel_1_1TaurusHmsLabel.html
[191] https://doc.qt.io/qt-5/qdoublespinbox.html
[192] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[193] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html

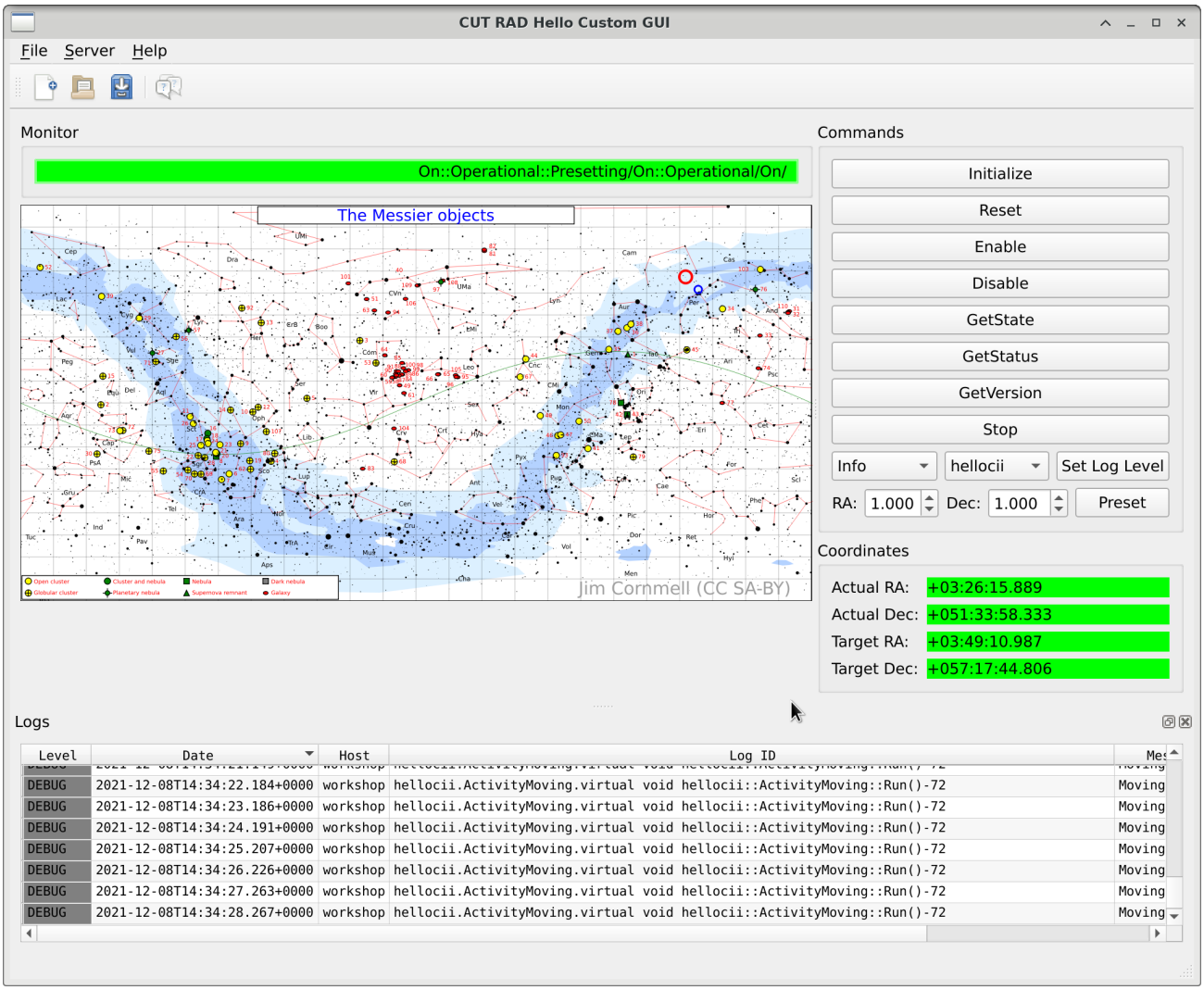


Fig. 69: CUT RAD Hello Custom GUI - Final product of this section of the tutorial.

OLDB Data Points (cut_rad_hello_custom/src/cut_rad_hello_custom/mainwindow.ui)

This section is purely declarative (no code) and here is where we can see Qt and Taurus in action through declarative databinding.

We will delete the existing Form Layout that contains the *Coordinates*. We will not use those widgets. Instead we will use TaurusDmsLabel and TaurusHmsLabel to preset radians as HMS and DMS angle notation.

Set a Form Layout for the *Coordinates* Group Box. Next, please drag 8 Labels into the Group Box that was just created. You should end up with a 2x4 layout of Labels.

Name the **first** column of Labels:

- Actual RA:
- Actual Dec:
- Target RA:
- Target Dec:

We will use **Widget Promotion** to transform the second column of widgets. We will start with the QLabel[194] widget right of *Actual RA* Label. Right click on the QLabeL[195], and from the context menu select the *Promote to...* entry. You will see a Dialog like this one:

Enter these values:

- Base class name: QLabel
- Promoted class name: TaurusHmsLabel
- Header file: elt.cut.widgets.taurus

Click on the *Add* Button. Select the new entry for TaurusHmsLabel[196], and click on the button *Promote* Button.

Once a new widget is defined, it is save for future use within the `mainwindow.ui` file. So **promote** now the Label right to the *Target RA:* Label. You can right click on the Label, and click on the *Promote to >* menu entry. Please note that this is a different menu entry from *Promote to...*. There you can select the *TaurusHmsLabel* entry.

We will repeat the process above for TaurusDmsLabel[197] on the two other Labels. The entries for the new widgets in the *Promoted Widgets* Dialog are:

- Base class name: QLabel
- Promoted class name: TaurusDmsLabel
- Header file: elt.cut.widgets.taurus

[194] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[195] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[196] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusHmsLabel_1_1TaurusHmsLabel.html
[197] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusDmsLabel_1_1TaurusDmsLabel.html

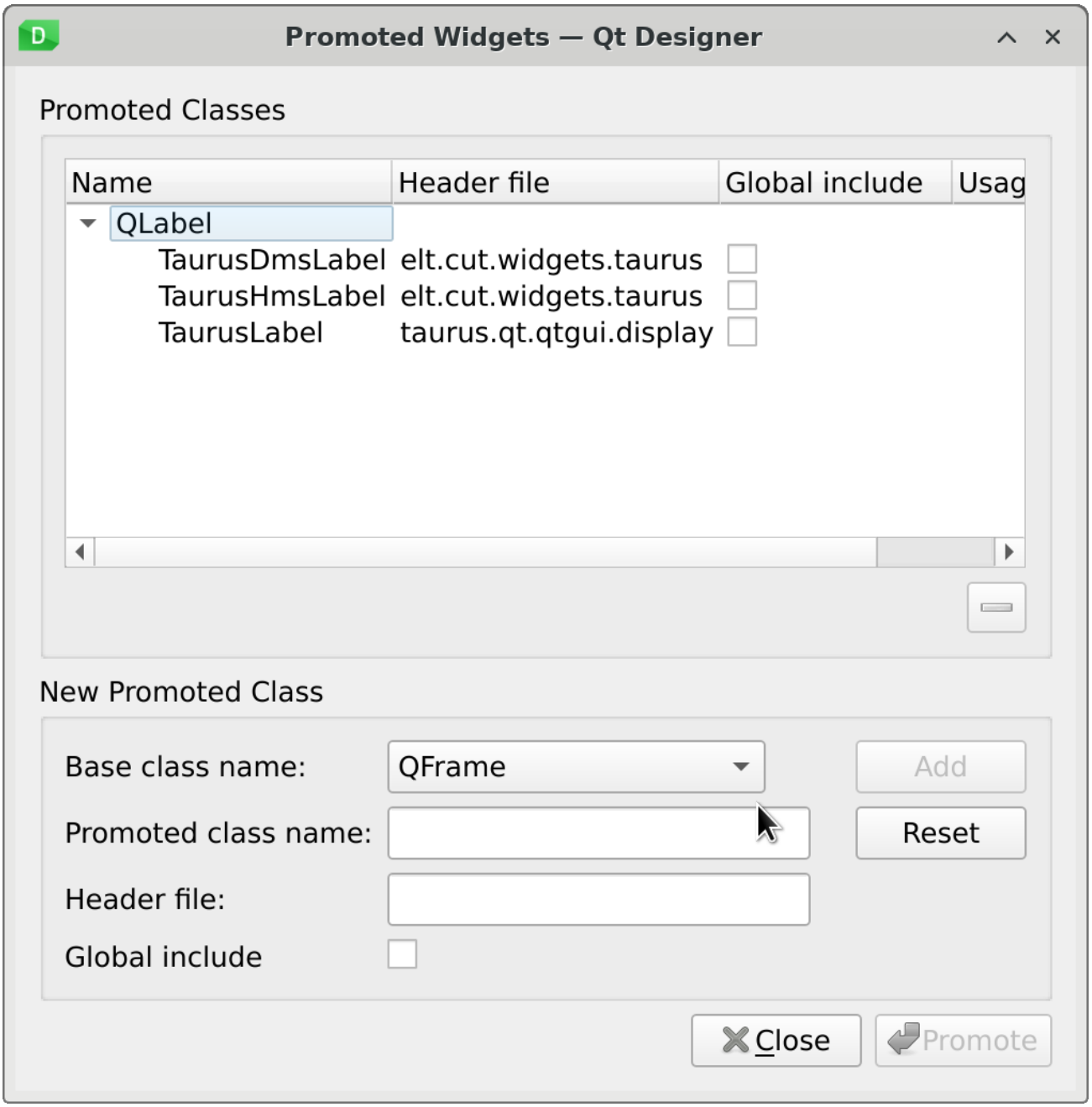


Fig. 70: *Promoted Widgets* Dialog

Now, lets add dynamic properties using the *Property Editor* Dock Widget. Click on the first TaurusHmsLabel widget, and using the *Property Editor* Dock Widget, click on the *Plus* Button. Select string, and name it **model**.

The values for the **model** properties are in order:

- cii.oldb:///elt/hellocii/mon/actual/ra
- cii.oldb:///elt/hellocii/mon/actual/dec
- cii.oldb:///elt/hellocii/mon/target/ra
- cii.oldb:///elt/hellocii/mon/target/dec

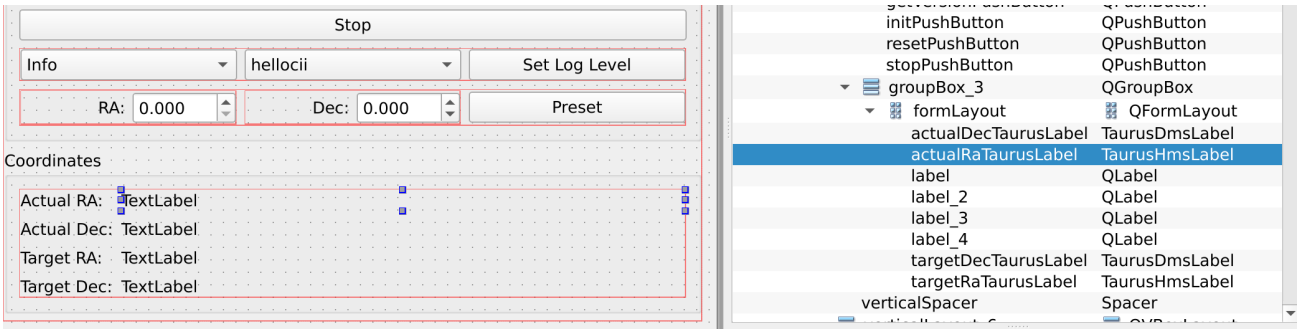


Fig. 71: Coordinates Group Box

OLDB Data Points in QeDartBoard

At this point, we can execute the application and we will see that the application will present the OLDB values. But we can do better: we can propagate these values to the QeDartBoard[198].

We will use Qt Designer to connect the TaurusHmsLabel and TaurusDmsLabel Signals to the QeDartBoard Slots.

Click on the *Edit Signal/Slot* Toolbar Button, or press `F4`. This will change the behavior of the Qt Designer. At this point, we cannot add nor remove widgets, we can only lay connections between widgets. We start a Drag operation in the first TaurusHmsLabel[199] and drop it on the QeDartBoard[200].

This will open a new *Configure Connection* Dialog, were we can select a pair of Signal (from the TaurusHmsLabel[201]), and a Slot (from QeDartBoard[202]).

Click on the *Edit...* Button on the left side of the *Configure Connection* Dialog. A new Dialog will open, called *Signals/Slots for TaurusHmsLabel*, and it contains the the list of Signals and Slots that the TaurusHmsLabel[203] widgets offers. We have added manually the widget, and because of this, Qt

[198] http://www.eso.org/~ahoffsta/docs/latest/doxygen/html/classQeDartBoard.html
[199] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusHmsLabel_1_1TaurusHmsLabel.html
[200] http://www.eso.org/~ahoffsta/docs/latest/doxygen/html/classQeDartBoard.html
[201] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusHmsLabel_1_1TaurusHmsLabel.html
[202] http://www.eso.org/~ahoffsta/docs/latest/doxygen/html/classQeDartBoard.html
[203] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusHmsLabel_1_1TaurusHmsLabel.html

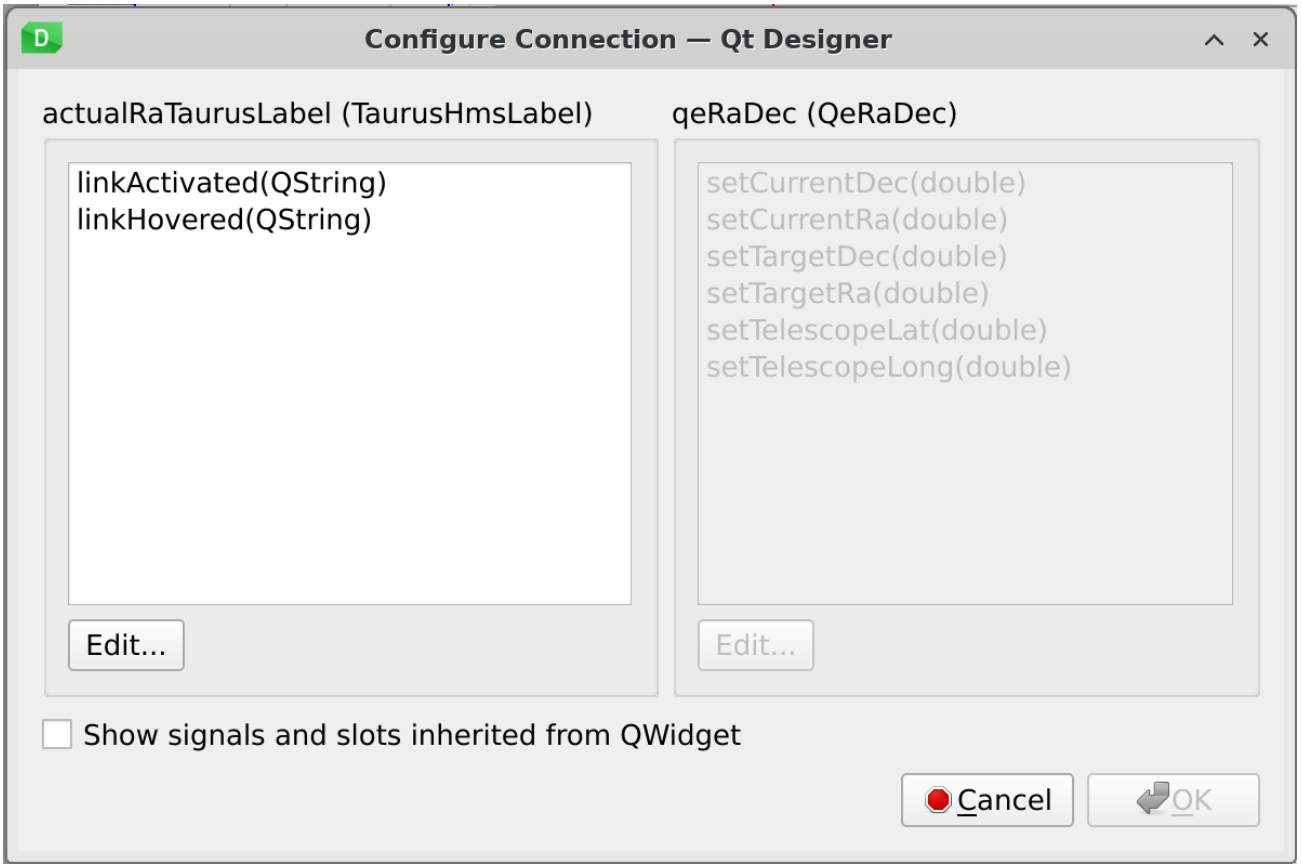


Fig. 72: *Configure Connection* Dialog: Here we click on the *Edit...* button on the left.

Designer does not know its extras signals. We will add only one signal, and for that click on the bottom *Plus* Button. A new entry is created. Type **radiansChanged(double)**. Click on the *OK* button, which will close the Dialog.

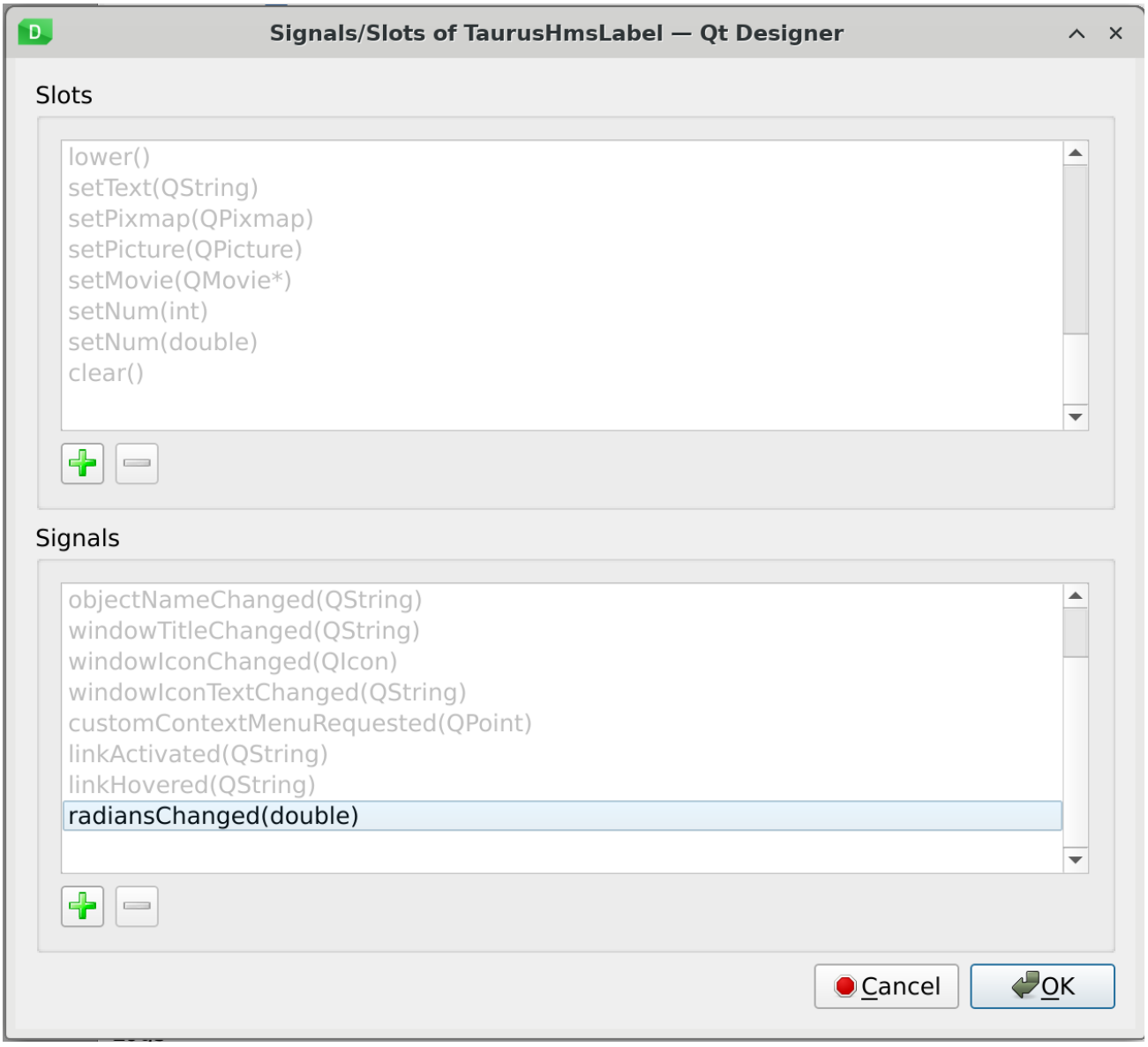


Fig. 73: *Configure Connection* Dialog: Here we add the signal provided by TaurusHmsLabel

At this point you can select the **radiansChanged(double)** Signal, and once it is selected, the Dialog will show us the entries that match the signature. Select the Slot **setCurrentAlt(double)**.

We repeat the same procedure for the next three widgets.

- Current RA: setCurrentAlt(double) – Already done
- Current Dec: setCurrentAz(double)

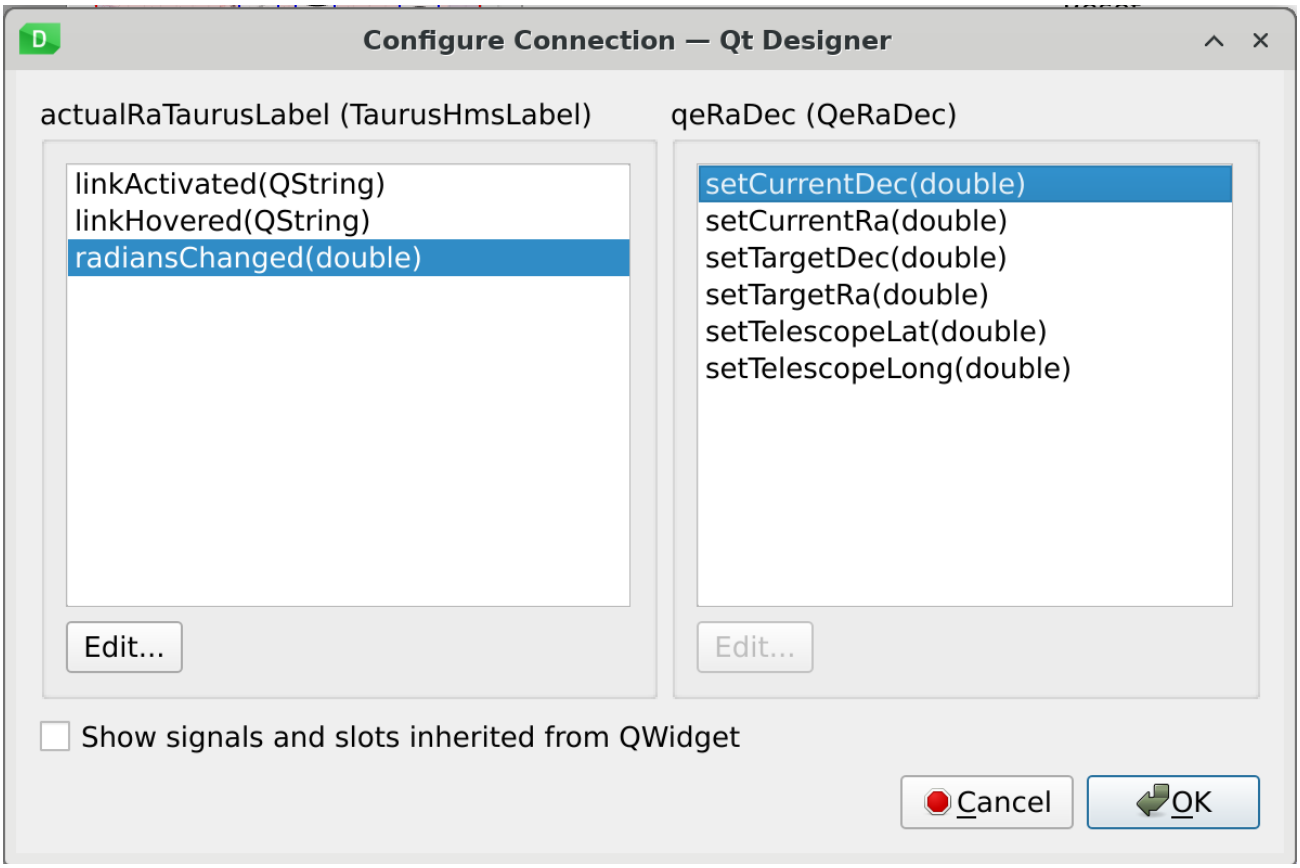


Fig. 74: *Configure Connection* Dialog: We now select **radiansChanged(double)** as Signal and **setCurrentAlt(double)** as Slot.

- Target RA: setTargetAlt(double)
- Target Dec: setTargetAz(double)

Now, every time a new value is displayed in the TaurusHmsLabel[204] or TaurusDmsLabel[205] widgets, the same coordinate is updated in the QeDartBoard[206].

Press `F3` or click on the *Edit Widgets* Toolbar Button to return to the normal view.

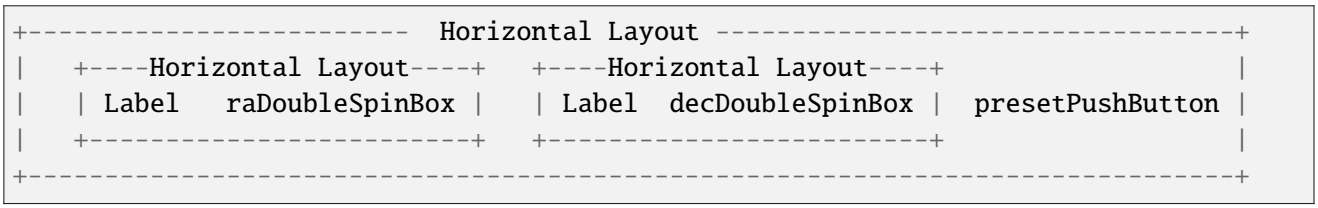
Preset Command

The Preset Command works very similar to the `SetLogLevel` Command. We will need to add the necessary widgets to the UI, a method that handles the click in the button, and a method that manager triggers the call in the server application.

Modifications to (cut_rad_hello_custom/src/cut_rad_hello_custom/mainwindow.ui)

Drag a new Horizontal Layout from the *Widget Toolbox* to the :*Commands* Group Box below all the other buttons. The reader will notice that this layout appear as a thin red line. Dropping new widget in it requires very fine precision with the mouse, or to use the *Object Inspector*.

We will do a bit more complex layout this time:



Is a double nested Horizontal Layout. This will allow us to match the layout from the `SetLogLevel` command, and have the widgets needed to collect the arguments for the Preset Command. Try and replicate this on your own. Remember to use the *Object Inspector* Dock Widget to place the widgets and layouts. This will make you life much more easier.

Set the stretch factor of the first Horizontal Layout to **1,1,1**.

Right the **alignment > Horizontal** property of the Two Labels to Right. This will create a bit of spacing, while putting the Label closer to the Double Spin Box it represents.

Remember to set the **objectNames** of the QDoubleSpinBox[207] widgets and the QPushButton[208] widget. We will use these in the implementation.

[204] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusHmsLabel_1_1TaurusHmsLabel.html
[205] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classtaurus_1_1TaurusDmsLabel_1_1TaurusDmsLabel.html
[206] http://www.eso.org/~ahoffsta/docs/latest/doxygen/html/classQeDartBoard.html
[207] https://doc.qt.io/qt-5/qdoublespinbox.html
[208] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html

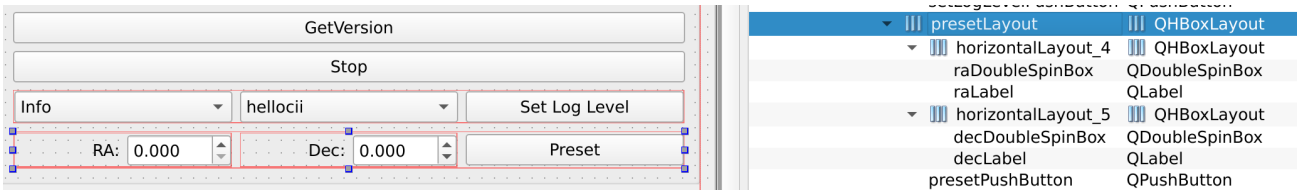


Fig. 75: Preset Layout and its Object Inspector view

Modifications to UI Implementation (cut_rad_hello_custom/src/cut_rad_hello_custom/applicationwindo

The `applicationwindow.py` file needs to include one more method. This is a Slot that through **auto-connection** takes cares of calling the `preset()` method in the `server_facade` object.

We can see that in the name of the Slot defined in line 3, we refer to the `presetPushButton` object, which hold the actual widget we have defined in the `mainwindow.ui` designer UI file. It reacts to the `clicked` Signal.

Just as the ICD method `SetLogLevel`, `Preset` also uses two members to fill a structure. This file `applicationwindow.py` concerns itself only with presentation and interaction, so we only pass the values captured form the user by the UI to the `server_facade.preset()` method.

To get the values from the `raDoubleSpinBox` and `decDoubleSpinBox` widgets, we use the provided method `value()`.

```
1  @TraceIt()
2  @Slot()
3  def on_presetPushButton_clicked(self):
4      '''
5      Slot that auto-connects to the QPushButton presetPushButton.
6      presetPushButton is not declared in the code, but in the mainwindow.ui.
7      See: https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-
   ↪with-auto-connect
8      '''
9      self.server_facade.preset(
10         self.ui.raDoubleSpinBox.value(),
11         self.ui.decDoubleSpinBox.value(),
12         result_slot=lambda x: self.info("Return: {}".format(x)))
13     pass
```

Modifications to Server Class (cut_rad_hello_custom/src/cut_rad_hello_custom/stdcmdsfacade.py)

We need to add the method listed below to the `stdcmdsfacade.py` file. This creates in line 29 a new Structure using the `mal` object from its MalAdapter[209] parent class. Then, it takes the arguments `ra` and `dec` and sets them to the `TelPosition` Structure.

After this, the call procedes just as usual.

```
1  @Slot(float, float)
2  def preset(self, ra: float, dec: float, result_slot=None, error_slot=None):
3      """
4      This method executes the Preset command on the server.
5      It uses the Asynchronous interface.
6      It will wait or the future object to has its result, and then
7      it will call the slot function with the result as argument.
8
9      Parameters
10     ----------
11     ra: float
12         Right Ascension value in radians.
13     dec: float
14         Declination value in radians.
15     result_slot: QtCore.QSlot
16         Slot method that will be triggered when the result is available.
17     error_slot: QtCore.QSlot
18         Slot method that will be triggered when an error is detected.
19
20     Return
21     -------
22     None
23
24     """
25     if(not self.isConnected()):
26         self.error('Application is not connected to Hellocii application. \
27                     Please connect first.')
28         return
29     coordinates = self.get_mal().createDataEntity(TelPosition)
30     coordinates.setRa(ra)
31     coordinates.setDec(dec)
32     task = Task(self.get_interface().Preset, coordinates)
33     if result_slot is not None:
34         task.signals.result.connect(result_slot)
35     if error_slot is not None:
36         task.signals.error.connect(error_slot)
```

(continues on next page)

[209] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classcomms_1_1maladapter_1_1MalAdapter.html

(continued from previous page)

```
37     task.signals.finished.connect(self.thread_complete)
38     task.start()
```

At this point, the application is ready to be used.

Replace QeDartBoard by QeRaDec (cut_rad_hello_custom/src/cut_rad_hello_custom/mainwindow.ui)

We recommend the reader to replace the QeDartBoard[210] with a QeRaDec[211] as an extra exercise. The change is rather simple. Just click on the QeDartBoard[212] and press `Delete`. Drag a new QeRaDec[213] to the same position.

We need to remake the connection we establish before, but using slots at widgets that are more appropriate to the coordinates system:

- Actual RA: setCurrentRa(double)
- Actual Dec: setCurrentDec(double)
- Target RA: setTargetRa(double)
- Target Dec: setTargetDec(double)

The QeRaDec[214] is a rather simple implementation. It uses a vector-based image (whose author it Jim Cornmell), and a cylindrical projection over the image. The ratio is of the image is 2:1, and uses radians as input, just like the QeDartBoard[215].

[210] http://www.eso.org/~ahoffsta/docs/latest/doxygen/html/classQeDartBoard.html
[211] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classQeRaDec.html
[212] http://www.eso.org/~ahoffsta/docs/latest/doxygen/html/classQeDartBoard.html
[213] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classQeRaDec.html
[214] https://www.eso.org/~ahoffsta/docs/v1.2.2/doxygen/html/classQeRaDec.html
[215] http://www.eso.org/~ahoffsta/docs/latest/doxygen/html/classQeDartBoard.html

5 Examples

CUT code repository[216] includes code that describes the following examples.

5.1 Python Application

Created from the Python Application Template, it shows a very basic GUI Application and how its project is to be structured.

A thorough explanation on the details of this example can be found in the Python Application Tutorial, see *Python Application Tutorial*.

The code can be found at https://gitlab.eso.org/ecos/cut/-/tree/master/examples/python_application

5.2 Widget Library

Created from the Widget Library Template, this example introduces the structure needed to produce a widget library. Special emphasis in WAF project wscripts file is given.

The code can be found at https://gitlab.eso.org/ecos/cut/-/tree/master/examples/widget_library

5.3 Analog Clock

Based on Qt's Analog Clock, it shows how a new widget is created. It also uses WAF and the widget library template.

Please take a look at the examples/analogclock/widgets directory (where the code of the widget is located), and *Creating a New Widget* for more details on the implementation.

The code can be found at https://gitlab.eso.org/ecos/cut/-/tree/master/examples/analogclock

5.4 Complex Application

This application has many widgets, tabs, and dialogs. Most of them are placeholders, but this project structure exemplifies how you can also structure a complex GUI application.

[216] https://gitlab.eso.org/ecos/cut

5.5 Color Scheme

Example application that shows the current color scheme, and shows how the different schemes looks with the different widgets provided by Qt, Taurus and CUT.

5.6 Form Item Factory

Example Taurus Form plugin. It provides a new plugin for Taurus Form that dynamically extends the capabilities of all Taurus-based applications.

Includes specific widgets that are used by TaurusForm (`taurus form`) when connecting to a MAL Request/Reply server that has the StdCmds interface or the DemoService interface.

It exemplifies how a developer may provide reusable widgets for special devices or server applications.

https://gitlab.eso.org/ecos/cut/-/tree/master/examples/form_item_factory

5.7 Pre Focal Station

Example application that demonstrates how to embed PFS widget with Taurus bindings in GUI application and assign a model to it.

The code can be found at https://gitlab.eso.org/ecos/cut/-/tree/master/examples/pfswidget

5.8 cutDemoService

This tool is not a graphical application, but provides datapoints needed to demonstrate and test the capabilities of CUT, its widgets, and its taurus plugins.

The code of this application can be found at https://gitlab.eso.org/ecos/cut/-/tree/master/examples/demo_service

It has an `icd` module to interact with it, and a `service` module that uses the `icd` to create a server application that publishes many datapoints.

```
$ cutDemoService --help
Demo Service
Options for cutDemoService:
  -i [ --instance ] arg (=1)  Instance in OLDB used in path of the URI. Use it
                              to avoid conflicts with other users.
  -f [ --frequency ] arg (=3) Frequency in Hz at which datapoints are updated
                              and published.
  -p [ --port ] arg (=12801)  Base value for port: Will be used to RR, and
                              subsequent numbers for PS.
  -o [ --oldb ] arg (=1)      Initializes and publishes example datapoints to
                              OLDB.
```

(continues on next page)

(continued from previous page)

```
  --ps arg (=0)              Creates MAL publisher using the example
                             datapoints.
  --rr arg (=0)              Creates MAL request-reply server using ZeroMQ and
                             Protocol Buffers.
  -l [ --list ]              List the datapoints available and exit.
  -h [ --help ]              show help for cutDemoService.
```

The user may customize its behaviour using the presented options.

Once started `cutDemoService` will present the list of datapoints.

```
$ cutDemoService
Demo Service
Initializing datapoints...
  Datapoints initialized...
Initializing OLDB...
  cii.oldb:///cut/demoservice/instance1/boolean-scalar-fixed  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/boolean-scalar-twosecs  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/int-scalar-add  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/int-vector-sin  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/int-matrix-sin-cos  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/uint64-scalar-sin  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-add  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-sin  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-cos  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-tan  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-sin-bad  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-scalar-sin-suspect  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-rand  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-current-radec  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-target-radec  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-current-altaz  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-target-altaz  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-sin  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-cos  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-vector-tan  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-sin  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-cos  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-sin-wave  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/double-matrix-sin-cos  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/string-scalar-fixed  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/string-scalar-timestamp  MD DP  VAL   OK
  cii.oldb:///cut/demoservice/instance1/string-vector-galaxies  MD DP  VAL   OK
  OLDB Initialized
```

(continues on next page)

(continued from previous page)

```
Starting Datapoint updates...
  Datapoint updates started
Press ENTER to exit
```

The `cutDemoService` will publish the following datapoints into the OLDB:

cii.oldb:///cut/demoservice/instance1/boolean-scalar-fixed
: Fixed boolean

cii.oldb:///cut/demoservice/instance1/int-scalar-add
: Integer datapoint, add 1 every update

cii.oldb:///cut/demoservice/instance1/double-scalar-add
: Double datapoint adds 1 every update

cii.oldb:///cut/demoservice/instance1/double-scalar-sin
: Double datapoint, returns the value of sin(t) with a period of 10 seconds.

cii.oldb:///cut/demoservice/instance1/double-scalar-cos
: Double datapoint, returns the value of cos(t) with a period of 10 seconds.

cii.oldb:///cut/demoservice/instance1/double-scalar-tan
: Double datapoint, returns the value of tan(t) with a period of 10 seconds.

cii.oldb:///cut/demoservice/instance1/double-vector-rand
: Vector of doubles, size 5, randomly generated between 0 and 1.

cii.oldb:///cut/demoservice/instance1/double-vector-sin
: Vector of 100 double datapoints, describes sin(t+dx) with period 2Pi covered of 10 seconds

cii.oldb:///cut/demoservice/instance1/double-vector-cos
: Vector of 100 double datapoints, describes sin(t+dx) with period 2Pi covered of 10 seconds

cii.oldb:///cut/demoservice/instance1/double-vector-tan
: Vector of 100 double datapoints, describes sin(t+dx) with period 2Pi covered of 10 seconds

cii.oldb:///cut/demoservice/instance1/double-matrix-sin
: Matrix of 16x16 double datapoints, describes sin(t+dx) with a period of 10 second fitted over x, copied over y.

cii.oldb:///cut/demoservice/instance1/double-matrix-cos
: Matrix of 16x16 double datapoints, describes cos(t+dx) with a period of 10 second fitted over x, copied over y.

cii.oldb:///cut/demoservice/instance1/double-matrix-sin-wave
: Matrix of 64x64 double datapoints, describes sin(t+dx+dy) with a period of 10 seconds.

cii.oldb:///cut/demoservice/instance1/double-matrix-sin-cos
: Matrix of 101x101 double datapoints, describes sin(t+dx)*cos(t+dy) with a period of 10 seconds.

cii.oldb:///cut/demoservice/instance1/string-scalar-fixed
: String datapoint, fixed.

cii.oldb:///cut/demoservice/instance1/string-scalar-timestamp
String datapoint, published the ISO timestamp on the server.

cii.oldb:///cut/demoservice/instance1/string-vector-galaxies
String vector with random galaxy names.

5.9 RAD Hello GUI

Created from the Python Application Template, the example adapts the template to connect and interact with a RAD application using its interface.

This example requires the user to follow first the RAD Tutorial 1[217].

The code can be found at https://gitlab.eso.org/ecos/cut/-/tree/master/examples/rad_hello

5.10 RAD Hello Custom GUI

This example customizes the previous one to interact with the RAD application that results from completing the RAD Tutorial 2[218].

This examples requires the user to follow the RAD Tutorial 2[219].

The code can be found at https://gitlab.eso.org/ecos/cut/-/tree/master/examples/rad_hello_custom

[217] https://www.eso.org/~eltmgr/ICS/documents/RAD/sphinx_doc/html/docs/rad.html#tutorial-1-creating-an-application-with-rad-cii
[218] https://www.eso.org/~eltmgr/ICS/documents/RAD/sphinx_doc/html/docs/rad.html#tutorial-2-customizing-an-application-with-rad-c
[219] https://www.eso.org/~eltmgr/ICS/documents/RAD/sphinx_doc/html/docs/rad.html#tutorial-2-customizing-an-application-with-rad-c

6 Widget Creation

When a developer finds that the existing widgets are no longer useful, they may find the need to create new widgets. This is a common task while developing GUIs.

Tip: The ELT Control GUI Toolkit Design[220] includes more information on how to use existing widgets.

A new widget usually falls into the following three categories.

New Widget
When a new mode of display or interaction is needed, and there is not currently available widget that can meet the requirements, then a completely new widget is needed. This kind of widgets are for example, the DartBoard and the Analog Clock.

Specialize Existing Widget
If an existing widget matches closely the representation and interaction capabilities of an existing widget, it is recommended to specialize the similar widget.

Composite Widget
When developing complex UIs, it is a common task to create a widget than contains a set of widgets. This commonly represent real world concepts such as a Motor, or can be used to present several monitor points.

We include also a section that discusses topics common to all kind of widgets.

6.1 Common to all kind of widgets

Widget Class Declaration

It is required that the widgets uses Signal, Slots and Properties as interface. GUIs should consider it their design the Qt event oriented paradigm. This paradigm is best suited when using Qt intended interfaces. This means:

Properties
Properties are members of a widget class definition that express a configuration. Later, when using the Qt Designer, properties are automatically displayed by the designer in the *Property Editor* tab.

Properties should use basic datatypes, such as int, unsigned int, float, bool, str.

A Property has normally a related signal that acts as a change notification, and a related slot that allows to set the property.

Signals
Signals are send by a widget when a property changes, or when an event of significance has

[220] https://pdm.eso.org/kronodoc/HQ/ESO-377114/1

happened, and the developer of the widget wants other GUI elements to react to this significant event.

Signals can be typeless, or can carry arguments. For example, when a QPushButton[221] is pressed[222], a typeless signals is send. When a QCheckBox[223] has its stateChanged[224], the signals carries the new state value.

Note: Signals require no implementation, only declaration.

Slots

Slots are normally a setter for a property, or allow to perform an action on the widget.

It is useful to create Slots for Properties: then the developer may connect one signal that provides the value, and the slot that sets the value to the widget.

Tip: Declaring signals and slots is cheap and imposes no penalty in performance. Using connections in the same thread is a direct function call.

Warning: Connecting a signal to a slot in different threads is slow. Qt will by default create an event with the arguments of the signal, and pass it to the appropriate Event Pump.

The definition of Properties in Python should be done as in this example:

The definition of Properties in Python should be done as in this example:

```
#include <QWidget>

class AWidget : public QWidget
{
    Q_OBJECT

    /**
     * @brief Altitude coordinate of the current position in radians.
     * @accessors getCurrentAlt(), setCurrentAlt(double)
     */
```

(continues on next page)

[221] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QPushButton.html
[222] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QAbstractButton.html#PySide2.QtWidgets.PySide2.QtWidgets.QAbstractButton.pressed
[223] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QCheckBox.html
[224] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QCheckBox.html#PySide2.QtWidgets.PySide2.QtWidgets.QCheckBox.stateChanged

(continued from previous page)

```
    Q_PROPERTY(double currentAlt MEMBER m_currentAlt READ getCurrentAlt WRITE␣
↪setCurrentAlt NOTIFY currentAltChanged )

public :
    AWidget(QWidget* parent = Q_NULLPTR): QWidget(parent){
    }
    double getCurrentAlt(){
        return this->m_currentAlt;
    };

public slots:
    double setCurrentAlt(double newValue){
        if( this->m_currentAlt != newValue ){
            this->m_currentAlt = newValue;
            emit currentAltChanged(this->m_currentAlt);
        }
    };

signals:
    void currentAltChanged(double newValue);

private:
    double m_currentAlt = 0.0;
}
```

The comments over the Q_PROPERTY macro are doxygen compatible. Doxygen can be configured to process and understand Qt Properties, Signals and Slots.

Note that the signal has no implementation.

```
from PySide2.QtCore import QWidget, Signal, Property, Slot

class AWidget(QWidget):

    def __init__(self):
        self._current_alt = 0.0

    def get_current_alt(self) -> double:
        """Gets the value of the current altitude
        :returns: double with value of the current altitude in radians.
        :rtype: double
        """
        return self._current_alt

```

(continues on next page)

(continued from previous page)

```
15      @Slot(double)
16      def set_current_alt(self, new_value: double):
17          """Sets the current altitude to new_value.
18          :param[in]: new_value double with value of the current altitude in radians.
19          """
20          if( self._current_alt != new_value ):
21              self._current_alt = new_value
22              self.current_alt_changed.emit(self._current_alt)
23
24      ## Signal that indicates that the current altitude has changed.
25      current_alt_changed = Signal(double)
26
27      ## Property that holds the current altitude to be rendered.
28      #  It uses radians as units
29      current_alt = Property(double, get_current_alt, set_current_alt, notify=current_
↪alt_changed)
```

Error: Do not use QStringList as a bug in Qt 5.15 (https://jira.eso.org/browse/ETCS-1036, https://bugreports.qt.io/browse/PYSIDE-1942) does not make it usable under python. As a workaround until we port to Qt 6, please use QString with a separator.

WAF project/module for the Widget

After the declaration of the Widget, this files needs to part of WAF project. The preparation of a WAF project that can host Widget is discussed extensible in *Widget Library Tutorial*. Developers may also use the provided example code, or the template to create a new library for their own widgets.

6.2 Creating a New Widget

The creation of a completely new widget is the most complex kind of widgets a developer may create.

Warning: We recommend the reader to be certain that existing widgets cannot satisfy display and interaction requirements.

Tip: An updated version of Qt's Analog Clock Example is provided in the CUT examples. This example explains how to create a simple widget that draws its own presentation. The updates include: the use of real/float datatypes for coordinates; access to colors using palette; and a WAF project. This can be found at `examples/analogclock`.

In order to develop a new widget, it is important that the developer understands how palette, fonts, coordinates and size/layout works in Qt.

The new widget will most likely inherit from QWidget[225]. If not, it should inherit from a class that at some point in its ancestors inherits from it.

We encourage that completely New Widgets are developed in C++ and bound to Python.

update() and paintEvent() methods

The QWidget[226] class provides the `update()` method. Invoking this method indicates to Qt that the widget has changes that affect its presentation, and should queue a `paintEvent()` when it estimates necessary. Developer should never invoke `paintEvent()` directly.

When properties that influence presentation are set with new values, the `update()` method should be invoked.

When Qt requests a widget to update its presentation, it uses a QPaintEvent. The method that receives the QPaintEvent, and implements the drawing instructions of the widget is the `paintEvent()`. Qt uses properties to store the configuration values of the widget, and then `paintEvent()` method is used to prepare the pixmap for the presentation. During the course of the `paintEvent()` implementation properties can used to draw the pixmap according to the final user's requests.

event(QEvent*) and specialized event methods

The `event(QEvent*)` method is declared in the QObject[227] class but the QWidget[228] has a reimplementation of it. This reimplemented method processes many interaction events common to widgets and provides developers with more handy methods to override and customize specific behaviour.

Developers are encourage to override these specialized methods. In case a developer overrides a common behaviour, you should still call the method from the QWidget[229] implementation.

Note: Commonly overriden methods are `moveEvent` to get coordinates of the widget, and the `mousePressEvent()/mouseReleaseEvent()` to implement interaction.

A list of the events handled by QWidget[230] and a basic description is included:

actionEvent
: widget's action have changed.

[225] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[226] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[227] https://doc.qt.io/qt-5/qobject.html
[228] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[229] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[230] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html

changeEvent
widget state has changed. Examples: font, enabled, palette.

closeEvent
widget received a close request from the window system. Default behaviour is to accept widget and close the window.

contextMenuEvent
commonly triggered by right clicking on a widget, or using the context menu key.

dragEnterEvent, dragLeaveEvent, dragMoveEvent
While a drag operation is in progress, these methods are called when entering the area of the widget, when leaving the area of the widget, or while moving above the widget.

dropEvent
the drag operation finished via drop.

enterEvent, leaveEvent, mouseMoveEvent
event send when the mouse cursor enters/leaves or moves on the area of the widget.

focusInEvent, focusOutEvent
widget has gained/lost keyboard focus.

hideEvent, showEvent
sent when a widget is set to hidden/shown.

keyPressEvent, keyReleaseEvent
event that indicates a key has been pressed or released.

mouseDoubleClickEvent, mousePressEvent, mouseReleaseEvent
events triggered when double clicking, when a mouse button is pressed, or released.

moveEvent
the widget has moved to a different position.

resizeEvent
the widget's size has changed.

tabletEvent
events specific to digitizer tablets, such as WACOM products.

wheelEvent
mouse or touchpad/trackpad wheel events.

See also:

Please refer to QWidget[231] documentation for complete documentation on events.

[231] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html

Colors and Palette

All QWidgets have a reference to a QPalette[232] object. The `QWidget::palette()` is propagated from the QApplication and modified according to context. The developer is expected to use it to get the proper colors using ColorRole[233].

Qt prepares and makes available the `palette` member in all `QWidget`. This palette is defined by the system, and the Control GUI Guidelines indicates the ELT uses the system defined values to set its color scheme.

Warning: Do not deviate from the QWidget[234]'s `palette()` values if possible.

When creating QPen and QBrush, please use `QPalette::color()` and the appropriate `ColorRole` to obtain colors from the color scheme. A complete explanation with examples can be found at *QPalette*.

Fonts

Fonts are the main manner a GUI communicates information to a User. It is essential to be congruent with the font, since any deviation from the standard one will result in the user having to guess why the font changed.

Warning: Please use QWidget[235]'s `font()` to get the current font to render text.

When rendering a widget, introducing text results most of the time having to calculate the height and width of the text. To do this, please use QFontMetrics[236] class to calculate its dimensions. While QFont[237] includes information on the point size of the Font and its family, QFontMetrics[238] is used to do calculation with a QFont.

```
#include <QFontMetrics>
#include <QFont>

// ...

void AWidget::methodInAWidget(){
    QFontMetrics fm(this->font());
```

(continues on next page)

[232] https://doc.qt.io/qtforpython-5/PySide2/QtGui/QPalette.html
[233] https://doc.qt.io/qtforpython-5/PySide2/QtGui/QPalette.html#PySide2.QtGui.PySide2.QtGui.QPalette.ColorRole
[234] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[235] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[236] https://doc.qt.io/qt-5/qfontmetrics.html
[237] https://doc.qt.io/qt-5/qfont.html
[238] https://doc.qt.io/qt-5/qfontmetrics.html

(continued from previous page)

```
    int fh = fm.height();
    int fw = fm.horizontalAdvance("Text to Render");
}
```

```
from PySide2.QtGui import QFontMetrics
from PySide2.QtGui import QFont
from PySide2.QtWidgets import QWidget

// ...

class AWidget(QWidget):

    # ...

    def method_in_a_widget(self):
        fm = QFontMetrics(self.font());
        fh = fm.height();
        fw = fm.horizontalAdvance("Text to Render");
```

The height of a font never varies, but the width of a text does due to glyphs width, advance and kerning. Fonts include a kerning table, that indicates how much space should be between two pairs of letters (in case any modification is needed).

QFontMetrics[239] `horizontalAdvance()` allows to quickly calculate the width of a string as rendered with the appropriate QFont[240].

Coordinates

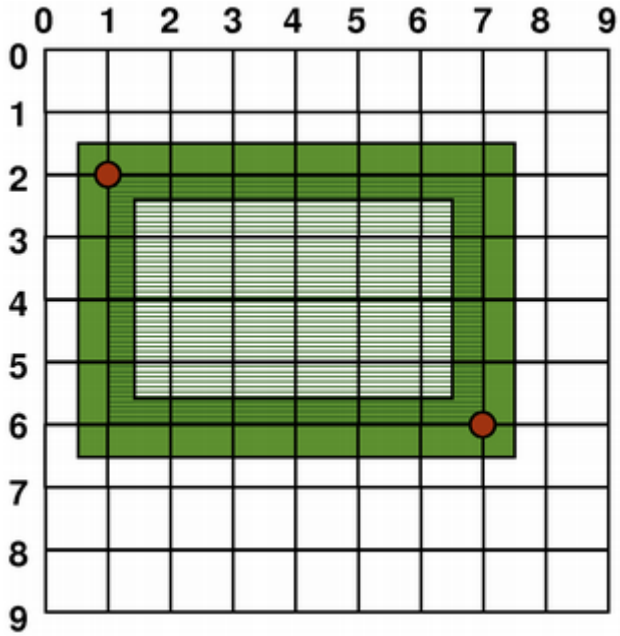
When drawing with QPainter or QPixmap, the origin of the coordinate system is located at the top-left of the widget area, increasing downwards and eastward.

More details on the coordinate system can be found here CoordinateSystem[241]

Warning: Use real number alternatives. QRectF[242] instead of QRect. QPointF[243] instead of QPoint. This is required by Qt to be compatible with HiDPI scaling.

Tip: *[Optional] Translate/Scale the coordinate system*

[239] https://doc.qt.io/qt-5/qfontmetrics.html
[240] https://doc.qt.io/qt-5/qfont.html
[241] https://doc.qt.io/qt-5/coordsys.html
[242] https://doc.qt.io/qtforpython-5/PySide2/QtCore/QRectF.html
[243] https://doc.qt.io/qtforpython-5/PySide2/QtCore/QPointF.html



As a first operation when drawing, the developer may introduce a translation, rotation and scaling instruction to effectively transform the coordinate system.

A common transformation is for radial plotting:

```
int side = qMin(widgetWidth, widgetHeight);
painter.translate(widgetWidth / 2.0, widgetHeight / 2.0);
painter.scale(side / 100.0, side / 100.0);
```

This transformation puts the origin at the center of the widget, and scales the coordinates to [-100.0, 100.0]

save() and restore()

The QPainter[244] object is a statefull rasterizer. It **will remember** translations, rotation and pen/brush configurations. The QPainter[245] methods `save()` `restore()` operate on a stack of states that include pen/brush and transformations.

For every `save()` invocation a `restore()` call must complete the pair. This is really helpful when iterating over coordinate position to render different elements.

[244] https://doc.qt.io/qt-5/qpainter.html
[245] https://doc.qt.io/qt-5/qpainter.html

Widget Size and Ratio

Some widgets impose a size restrictions. For example the QDial has a circular representation that requires a 1:1 ratio between height and width. The QKeySequenceEdit widget is as tall as one line of input using the current font.

The developer of a new widget needs to consider if the widget's size is to be constrained.

Even when the size is constrained, the surface available to draw may be bigger than what the widget needs. Expanding Policies and Default Alignment values could need to be changed in the constructor of the widget.

Also, when drawing, alignment may influence the point where the widget is drawn.

Most widgets include a minimum size. This is commonly the size where the contents would be unreadable and/or the interaction functionality is diminished.

Caching Pixmaps

While preparing the pixmap, QPainter object is used. It is possible to create new QPainter objects and draw temporary contents to these objects.

Since properties of the widget are used to draw these cached pixmaps, then when they change, the cache is invalidated.

If cache is to be implemented, we recommend the introduction of a member invalidatePixmapCache of type bool. Properties setters should invalidate the Pixmap cache, as well as certain QWidget properties. These QWidget properties do not provide signals, but events can be used:

```
void AWidget::changeEvent(QEvent *event)
{
    if( event->type() == QEvent::FontChange ||
        event->type() == QEvent::PaletteChange ||
        event->type() == QEvent::StyleChange ||
        event->type() == QEvent::EnabledChange
    ){
        this->m_invalidatePixmapCache = true;
        update();
    }
    QWidget::changeEvent(event);
}

void AWidget::resizeEvent(QResizeEvent *event)
{
    this->m_invalidatePixmapCache = true;
    QWidget::resizeEvent(event);    }
```

To render into a QPixmap[246] please follow this example:

```
void AWidget::paintEvent(QPaintEvent *event)
{
    Q_UNUSED(event);
    if( this->m_invalidatePixmapCache ){
        this->p_background = std::make_unique<QPixmap>(this->size());
        this->p_background->fill(Qt::transparent);
        painter->begin(p_background.get());
        // ... rendering instructions
        painter->end();
    }

    // ... normal rendering instructions
}
```

Plugin, Bindings and Example Application

All of this is covered in the *Widget Library Tutorial*.

6.3 Specialize an Existing Widget

Specialization of an existing widget is the easiest way to create new widgets. It is achieved by declaring a class that inherits from an existing functional widget that is not a QWidget[247].

In this case, the presentation of the widget satisfies the requirements, but some manipulation of an input property is needed.

We will explore an existing widget, the QeDmsLabel[248] to explain more on this topic.

Declaration

This is the class declaration of the QeDmsLabel, a widget that is part of CUT. This widget inherits from QLabel. We only need to present text, but we want to make the transformations needed to present the value part of the widget.

```
1 class QeDmsLabel : public QLabel {
2     Q_OBJECT
3
4     Q_PROPERTY(double radians MEMBER m_radians READ getRadians WRITE setRadians␣
```

(continues on next page)

[246] https://doc.qt.io/qt-5/qpixmap.html
[247] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[248] https://eltjenkins.hq.eso.org/job/ECOS/job/CUT/job/CUT-Build-Master-DevEnv-5/lastSuccessfulBuild/artifact/build/docs/html/classQeDmsLabel.html

(continued from previous page)

```
    ↪NOTIFY radiansChanged )
5       Q_PROPERTY(int precision MEMBER m_precision READ getPrecision WRITE␣
    ↪setPrecision NOTIFY precisionChanged )
6
7   public:
8       explicit QeDmsLabel(QWidget *parent = Q_NULLPTR);
9       ~QeDmsLabel();
10      double getRadians(){ return m_radians; };
11      int getPrecision(){ return m_precision; };
12
13  signals:
14      void radiansChanged(double newValue);
15      void precisionChanged(int newValue);
16
17  public slots:
18      void setRadians(double newValue){this->m_radians = newValue; emit␣
    ↪radiansChanged(newValue); this->update();};
19      void setPrecision(int newValue){this->m_precision = newValue; emit␣
    ↪precisionChanged(newValue); this->update();};
20      void update();
21
22  private:
23      double m_radians = 0.0;
24      int m_precision = 3;
25  };
```

- In line 1, the widget is declared to inherit from QLabel[249]. This in turn inherits from QWidget[250]. As always, any widget must inherit at some point from QWidget[251].
- Lines 4 and 5 add two properties: `radians` and `precision`.
- getter method of the properties are usually public.
- Notice that in lines 14 and 15, both properties have signals as declared after the `NOTIFY` section of the `Q_PROPERTY` macro.
- setter methods are declared under the `public slots` qualifier (line 17). It is a common practice to do so, as it allows the widget to be connected to another signal.

The two new properties are:

Radians
: This `radians` property will act as the new input property, instead of `text`.

Precision

[249] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[250] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html
[251] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html

`precision` is a configuration property, and modifies the way the widget behaves.

Note: Though in this document `radians` and `precision` are called input and configuration properties respectively, there is no distinction between them except how we use them in implementation.

Implementation

The implementation of the methods is rather simple. Please keep in mind that signals need no implementation:

```
1  QeDmsLabel::QeDmsLabel(QWidget *parent) :
2    QLabel(parent)
3  {
4
5  }
6
7  QeDmsLabel::~QeDmsLabel()
8  {
9  }
10
11 void QeDmsLabel::setRadians(double newValue){
12     if( newValue == this->m_radians ){
13         return;
14     }
15     this->m_radians = newValue;
16     emit radiansChanged(newValue);
17     this->update();
18 }
19
20 void QeDmsLabel::setPrecision(int newValue){
21     if( newValue == this->m_precision ){
22         return;
23     }
24     this->m_precision = newValue;
25     emit precisionChanged(newValue);
26     this->update();
27 }
28
29 void QeDmsLabel::update()
30 {
31     int values[4];
32     char sign;
33     eraA2af(this->m_precision, this->m_radians, &sign, values );
```

(continues on next page)

(continued from previous page)

```
34      this->setText(QString("%1%2:%3:%4.%5")
35                    .arg(QChar(sign))
36                    .arg(values[0], 3, 10, QChar('0'))
37                    .arg(values[1], 2, 10, QChar('0'))
38                    .arg(values[2], 2, 10, QChar('0'))
39                    .arg(values[3], this->m_precision, 10, QChar('0')));
40      QLabel::update();
41  }
```

- The constructor at line 1 calls its ancestors constructor. It is important that it accepts the `parent` argument in the widget constructor, as many properties are passed from parent to child during UI setup.
- The setter methods in line 11 and 20 check first if the value is the same, in which case it returns immediately. The widget only acts when there is change.
- Lines 16 and 25 uses the `emit` keyword, that is used to emit a signal. In this case, the corresponding signal for the property. The argument is the new value for the property.
- Lines 17 and 26 shows how the setter, at the end, call the `update()` method. Since a property essential for presentation has changed, we queue indicate to Qt that a refresh or update of the presentation is needed.
- Lines 29 to 39 shows the implementation of the update method. Essential is call an external library to perform a radians to Degrees Minutes Seconds representation, with a defined precision. It sets the `text` property to that string.
- Notice how in line 40, the ancestors `update()` method is invoked. At this point, we have called no drawing instruction, no custom QPainter[252] methods. Since the only alteration this implementation does to a QLabel[253] is to prepare a custom string of text, we only need to invoke the `QLabel::update()` method and it will render it for us.

With that, the implementation is done.

6.4 Create a Composite Widget from Existing Ones

Warning: We recommend to use Widget Composition in python. Declarative databinding is available only in python.

The previous two sections (*Creating a New Widget*, *Specialize an Existing Widget*) covered single widget creation. It is a common requirement for widget to be composed of several other widgets.

[252] https://doc.qt.io/qtforpython-5/PySide2/QtGui/QPainter.html
[253] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html

Widget composition can be achieved by creating a new widget that inherits from QWidget, and populating it with several member widgets that have as a parent the declared widget.

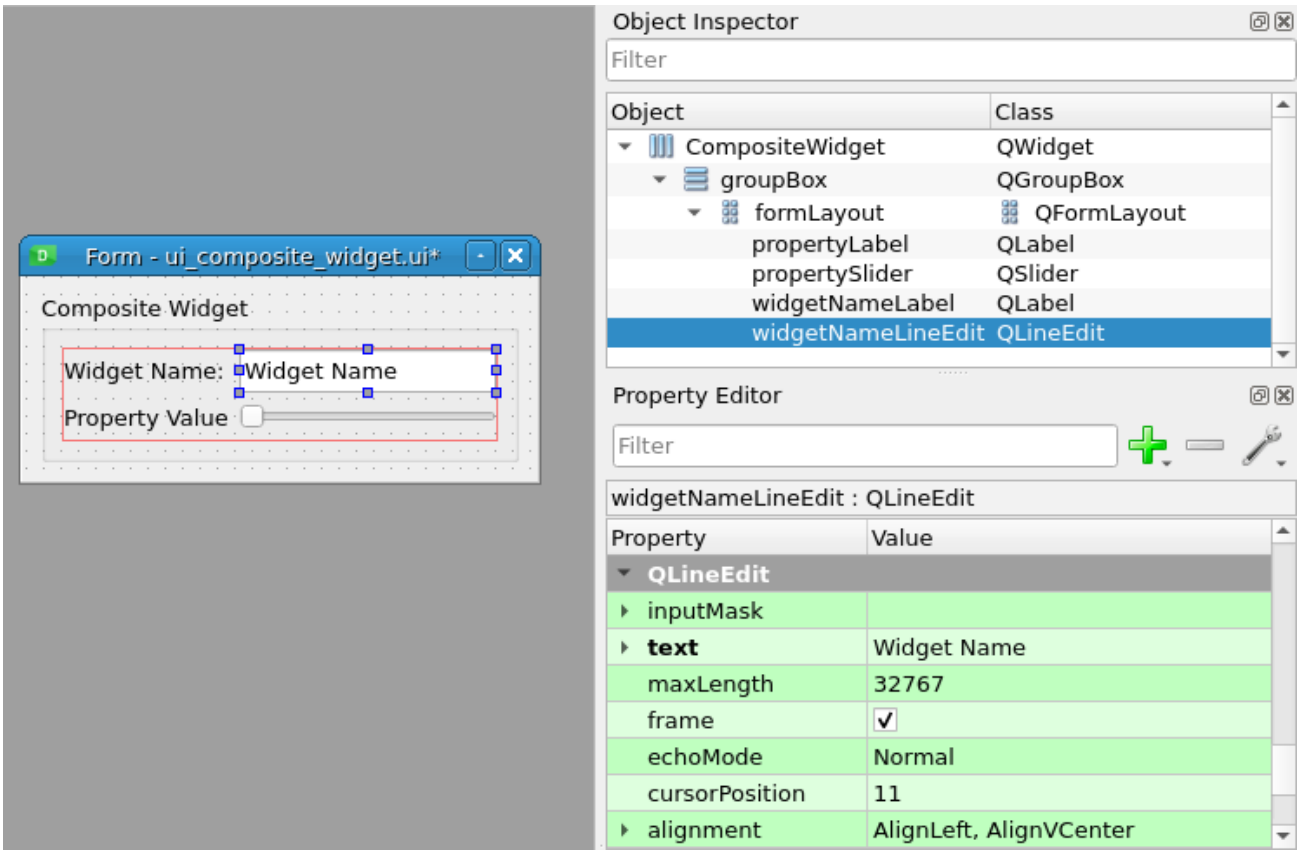
This was discussed briefly in the *Qt and UI Files* section. Here we will expand on it a bit more.

Tip: The CompositeWidget presented here is part of the `examples/composite_widget` project.

UI Declaration

The UI will consist of one QWidget, that has a QGroupBox and a Form Layout that contains:

- Two QLabel[254]: one for “Widget Name”, and another for “Property Value”.
- One QLineEdit[255] with an initial value of “Widget Name”.
- One QHorizontalSlider[256] with an initial value of 0.



It is important that the name of the top-level QWidget is “CompositeWidget”. Please change it on the

[254] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLabel.html
[255] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLineEdit.html
[256] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QHorizontalSlider.html

Object Inspector tab. See the figure for reference.

Widget Class

Please refer to `examples/complex_application/app/src/cut/examples/complex_application/widgets/composite_widget.py`.

```python
#!/usr/bin/env python3

import sys
from taurus.external.qt import Qt
from cut.examples.complex_application.widgets.ui_composite_widget import Ui_CompositeWidget


class CompositeWidget(Qt.QWidget):
    """Composite Widget.

    It uses declarative UI defined in ui_composite_widget.ui.
    Provides two properties, which in turn excersize the the widgets defined in the UI.

    It shows how to expose a cleaner interface using Properties, Signals and Slots.
    """

    def __init__(self, parent=None):
        Qt.QWidget.__init__(self, parent)
        self._numerical_value = 0
        self._widget_name = "Widget Name"
        self.ui = Ui_CompositeWidget()
        self.ui.setupUi(self)

        # Connects the value from the slider to the widget's numerical_value property.
        self.ui.propertySlider.valueChanged.connect(self.set_numerical_value)
        self.numerical_value_changed.connect(self.ui.propertySlider.setValue)
        # Connects the text from the line edit to the widget's widget_name property.
        self.ui.widgetNameLineEdit.textChanged.connect(self.set_widget_name)
        self.widget_name_changed.connect(self.ui.widgetNameLineEdit.setText)

    def get_numerical_value(self) -> int:
        """Gets the numerical_value property
        :returns: The value of the property
        :rtype: double
```

(continues on next page)

(continued from previous page)

```
        """
        return self._numerical_value

    def set_numerical_value(self, new_value: int):
        """Sets the numerical value to new_value
        :param[in]: new_value int with value to set in the numerical_value property
        """
        if(self._numerical_value != new_value):
            print("set_numerical_value: {}".format(new_value))
            self._numerical_value = new_value
            self.numerical_value_changed.emit(self._numerical_value)

    ## Signal that indicates that the numerical_value has changed.
    numerical_value_changed = Qt.Signal(int)
    ## Property that holds the numerical_value.
    numerical_value = Qt.pyqtProperty(int, get_numerical_value, set_numerical_value,
↪ notify=numerical_value_changed)

    def get_widget_name(self) -> str:
            """Gets the widget_name property
            :returns: The widget_name property value
            :rtype: str
            """
            return self._widget_name

    def set_widget_name(self, new_value: str):
        """Sets the widget_name to new_value
        :param[in]: new_value str with value to set in the widget_name property
        """
        if(self._widget_name != new_value):
            print("set_widget_name: {}".format(new_value))
            self._widget_name = new_value
            self.widget_name_changed.emit(self._widget_name)

    ## Signal that indicates that the widget_name has changed.
    widget_name_changed = Qt.Signal(str)
    ## Property that holds the widget_name.
    widget_name = Qt.pyqtProperty(int, get_widget_name, set_widget_name,␣
↪notify=widget_name_changed)


# This method is never used by the application, but as a developer they are handy␣
↪during early
```

(continues on next page)

(continued from previous page)

```
# development. You can call this python module using:
#     python3 -m cut.examples.complex_application.widgets.composite_widget
if __name__ == "__main__":
    # Since the TaurusApplication is not used in the CompositeWidget class, it is␣
↪imported here.
    from taurus.qt.qtgui.application import TaurusApplication

    app = TaurusApplication(sys.argv)
    widget = CompositeWidget()
    widget.show()
    sys.exit(app.exec_())
```

This class includes two properties:

- `widget_name` that is connected to the QLineEdit[257]
- `numerical_value` that is connected to the QHorizontalSlider[258]

It is good practice to declare properties at the CompositeWidget level, so that there is only one interface with the rest of the application.

The properties are connected bi-directionally. Qt can detect bidirectional connections and will not enter an endless loop (you can go ahead and try commenting the `if` checks). Regardless, the `if` check is not there to avoid endless loops, but to avoid emitting the signal when developers sets manually its value twice to the same value. Signal are often meant to signal change in the property. If this is not the behaviour you are looking for, you can remove the `if` check.

[257] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QLineEdit.html
[258] https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QHorizontalSlider.html

7 GUI Development

The *Python Application Tutorial* is hardly a complete application. It is intended to introduce the most basic WAF structure needed to produce a working product. For very simple applications this could suffice, but most likely it will fall short.

This chapter will introduce a most complex application and go into details of implementation.

7.1 CLI Options and Arguments

We recommend the use of `click` to parse arguments for the application. This document does not intended to teach you how to use `click`. There are several resources available for that.

Tip: To learn more about `click`, please refer to Click Quickstart[259]

Taurus includes several predefined command line options, so that every application matches in style. There are basic options we recommend you to use are located in `taurus.cli.common`

Some examples:

- `--log-level` to set the log level of the application
- `--config` to set the configuration file
- `--polling-period` to set the polling period

When capturing options from the click parser, if you have for one option, a predetermined ammount of posibilities, prefer the use of enumerations instead of strings.

```
import click
import enum

class RepositoryOptions(str, enum.Enum):
    RTR = "rtr"
    PTS = "pts"
    OLDB = "oldb"

@click.command("appentry")
@taurus.cli.common.log_level
@click.argument("selected_repo", type=click.Choice(RepositoryOptions))
def appentry(selected_repo: RepositoryOptions, log_level):
    print("Selected Repository {}".format(selected_repo))
    pass
```

(continues on next page)

[259] https://click.palletsprojects.com/en/8.1.x/quickstart/#basic-concepts-creating-a-command

(continued from previous page)

```
if __name__ == "__main__":
    appentry()
```

– Adapted from EnumChoice[260]

The `click` module can work as a single command, or multiple commands. Single command example would be `ls`. It executes one task, and for that it collects options and arguments. A `click`-based application can also have multiple commands, like `git`. Git has an entrypoint command `git` and several commands like `git commit` or `git add`, sharing options, or having command-specific options and arguments.

1. Decide if you application is to be a single or multiple command and list them.
2. Add help for each command in the docstring
3. Enumerate their options and argument
4. Set their datatypes
5. Set default values
6. Accept each option and argument as argument to the method that handles the command
7. Save all of these into a temporary object
8. Parse configuration files.
9. Compare the values from configuration files against the ones in the temporary object. Keep them as you defined which one has priority.
10. Create a final ParsedConfiguration object, and pass this object as reference to your MainWindowImpl class.

7.2 Data Manipulation and Taurus

Taurus Model and Widgets offers a quick way to declare databinding through the UI definition. This comes with some restrictions, as code cannot be introduced in the UI definition.

`taurus.qt.qtgui.base.taurusbase.TaurusBaseComponent.insertEventFilter()` provides a way to manipulate values reported by the polling or subscription mechanisms supported by any Taurus Model Plugin.

The example below shows how to use EventFilters.

```
#!/usr/bin/env python3

import sys
```

(continues on next page)

[260] https://github.com/pallets/click/issues/605#issuecomment-1016750937

(continued from previous page)

```
from taurus import Attribute
from taurus.qt.qtgui.application import TaurusApplication
from taurus.qt.qtgui.display import TaurusLabel
from taurus.core import TaurusEventType
from taurus.core.taurusbasetypes import TaurusAttrValue
from taurus.core import DataType
from taurus.external.qt import Qt


class EventValueMap(dict):
    """A filter destined to change the original value into another one
    according to a given map. Example:

        filter = EventValueMap({1:"OPEN", 2:"CHANGING", 3:"CLOSED"})

    this will create a filter that changes the integer value of the event
    into a string. The event type is changed according to the python type in
    the map value.

    For now it only supports simple types: str, int, long, float, bool
    """

    def __call__(self, s: Attribute, t: TaurusEventType, v: TaurusAttrValue):
        # It needs to be Change (subscription) or Periodic (polling) event
        if t not in (TaurusEventType.Change, TaurusEventType.Periodic):
            return s, t, v  # Otherwise, we do not apply mapping
        if v is None:  # If there is not value, no mapping can be applied
            return s, t, v  # Otherwise, we do not apply mapping

        # get from dict: applies transformation
        newvalue = self.get(v.rvalue)
        if(newvalue):
            v.rvalue = newvalue

        v.type = DataType.from_python_type(type(v.rvalue))
        return s, t, v


if __name__ == "__main__":
    app = TaurusApplication()

    panel = Qt.QWidget()
    layout = Qt.QVBoxLayout()
```

(continues on next page)

(continued from previous page)

```
47      panel.setLayout(layout)
48
49      w1, w2, w3, w4 = TaurusLabel(), TaurusLabel(), TaurusLabel(), TaurusLabel()
50      f1 = EventValueMap({1: "OPEN", 2: "CHANGING", 3: "CLOSED"})
51      f2 = EventValueMap({False: "ABSENT", True: "PRESENT"})
52      f3 = EventValueMap({True: "TRUE", False: "FALSE"})
53      f4 = EventValueMap({1: 1001, 2: 1002, 3: 1003, 4: 1004, 5: 1005, 6: 1006})
54      w1.insertEventFilter(f1)
55      w2.insertEventFilter(f2)
56      w3.insertEventFilter(f3)
57      w4.insertEventFilter(f4)
58      layout.addWidget(w1)
59      layout.addWidget(w2)
60      layout.addWidget(w3)
61      layout.addWidget(w4)
62      w1.model, w1.bgRole = 'cii.oldb:///cut/demoservice/instance1/uint64-scalar-sin',
    ↪ ''
63      w2.model, w2.bgRole = 'cii.oldb:///cut/demoservice/instance1/boolean-scalar-
    ↪fixed', ''
64      w3.model, w3.bgRole = 'cii.oldb:///cut/demoservice/instance1/boolean-scalar-
    ↪twosecs', ''
65      w4.model, w4.bgRole = 'cii.oldb:///cut/demoservice/instance1/uint64-scalar-sin',
    ↪ ''
66
67      # First read is not mapped.
68      # Could it be that first read is not triggering an event?
69
70      panel.show()
71      sys.exit(app.exec_())
```

A class *EventValueMap* is declared. It uses a `dict` as method of translation between the source value and the returned value. There are three types of Events taurus sends: Configuration, Periodic, and Change. If we want to alter values, then we need to catch events in the Periodic (Polling) and Change (Subscription) types (see line 28).

If the type of the data changes, it is important that the value also has its datatype changed. (see line 38).

Installing the filter needs to be done using code. Developers may still define the GUI in a declarative manner (i.e.: UI files), but the creation of the Filtel object and installing the filter needs to be done as shown in lines 50 and 54.

Tip: An EventFilter is a `collections.abc.Callable`. Therefore you can also use a function or class method.

As an alternative, a developer may implement their own widget that uses Taurus capabilities. For this, please follow custom-widgets

7.3 Data Units and Conversion

The Taurus library uses `Quantity` (from the python `pint` library) to express a datapoint from the OLDB. Please include the original unit in the metadata of the datapoint.

Then you can convert the *rvalue* to other units using the method `pint.Quantity.to()`, and pass as argument the new unit. The Pint library has conversions for common units already available.

```
>>> import taurus
>>> taurus.Attribute("cii.oldb:///elt/telif/ccs/target_observed_altaz")
MainThread WARNING 2023-11-09 08:16:45,488 TaurusRootLogger: epics scheme not␣
↪available:
ModuleNotFoundError("No module named 'epics'")
>>> lala = taurus.Attribute("cii.oldb:///elt/telif/ccs/target_observed_altaz")
>>> lala
TaurusCiiAttribute(cii.oldb:///elt/telif/ccs/target_observed_altaz)
>>> lala.rvalue
<Quantity([0.78539816 0.78539816], 'radian')>
>>> lala.rvalue.to("degrees")
<Quantity([45. 45.], 'degree')>
```

Please take a look at Pint Tutorial[261] for more information on Pint, or at `taurus.core.units` to see how taurus makes the Quantity and UnitRegistry classes available.

If a conversion is not available, you can add it to the `UnitRegistry`.

7.4 WAF and Project Structure for Bigger Applications

- use python package structure
- one subpackage for widgets
- one subpackage for comms
- one subpackage for domain specific code if needed

[261] https://pint.readthedocs.io/en/0.10.1/tutorial.html

7.5 Connecting to MAL or OLDB

Try to reuse MalAdapter and OldbAdapter.

If you don't remember to do it async, and push to other threads, because you will have to wait for connection timeout otherwise.

7.6 Logging

Remember to always include the CII Logger to $CII_LOGS. Any other log output will be lost during operations.

7.7 Widgets and Application properties as Configurable

Taurus provides a base class that allows to register properties within classes. Then an application can save all registered properties to a file, creating a state.

7.8 Complex UI

If a UI is too comple, refactor it into smaller Widgets. If the widget can be reused, perfect. If not, then it do it just for maintenance sake.

Use always UI file first, customize through code second.

UIs should always have a starting state. Think what that state is. (enable/disable, values at 0?, etc, certain actions enabled, other disabled)?

7.9 Declarative UI

7.10 Widget Promotion

7.11 Auto Slot Connection

name clashes

7.12 Declarative Databinding

7.13 Models

The MVC pattern can be used and applied to your application in many different forms.

CUT only includes a Model for single attributes. Getting multiple values and presenting them in complex ways is not possible to make it in a generic form.

The developer of complex apps will end up creating model. This does not mean that they will look similar all the time, or that the datasource is the same. You will need to adapt.

QStardardItemModel is the easiest.

A List can also be a model. But I suggest to use a QStandardItemModel because you can later user the ModelMapper.

Remember that you can also use taurus.Attribute to get values without doing declarative databinding.

7.14 Taurus Model Plugin Development

7.15 Debugging

Taurus Log Level

Any taurus command can change its logging output level using the option:

```
[eltdev@srtc1 cut]$ taurus --help
Usage: taurus [OPTIONS] COMMAND [ARGS]...

  The main taurus command

Options:
  --log-level [Critical|Error|Warning|Info|Debug|Trace]
                                  Show only logs with priority LEVEL or above
                                  [default: Info]
```

A good first step to debug a TaurusWidget or Scheme Plugin is to enable trace level logs and see what is going on.

OLDB GUI

Sometimes is can be good to compare the values presented in Taurus with the one in the database. You can do this using the `oldb-gui`. Its will also present you the metadata of the datapoint, which is used by cii.oldb plugin to fill more details into the `taurus.core.taurusattribute.TaurusAttribute` object.

The OLDB GUI also allows you to subscribe to datapoints changes.

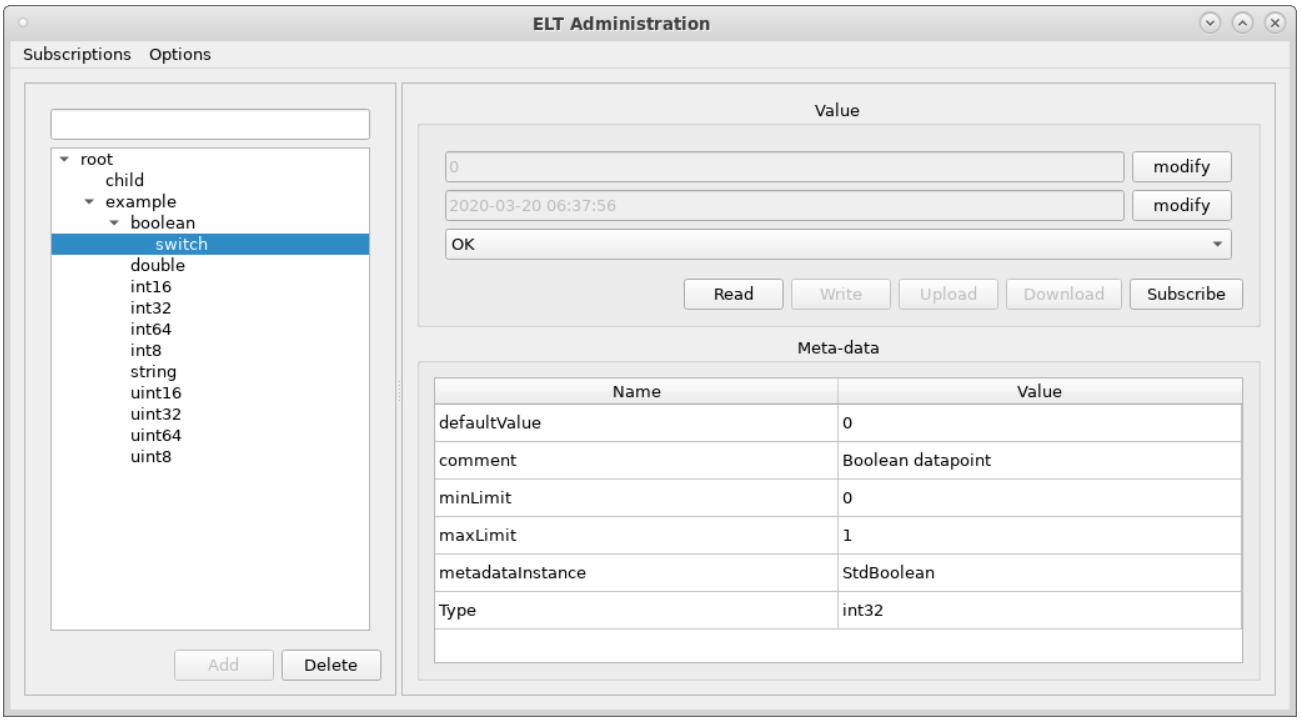


Fig. 1: OLDB GUI. In the left there is a browsable tree view of the database contents, and the user can select a datapoint. On the right hand side are the details of the selected datapoint.

GDB

Since python is programmed in C, and we are using several bindings, some errors can come from the libraries developed in C++. To obtain more information, a developer may still use gdb:

```
$ which <script_name>
$ gdb
...
(gdb)$ file python
(gdb)$ run <return_of_which_command_above>
```

Python faulthandler

In case needed, you can get a more complete error or exception output using this:

```
python3 -q -X faulthandle [script_name]
```

You can use it in the interactive script console, or while executing a python file.

Python Debugger

Since Python 3.7, the *breakpoint()* method is part of the language. You can add them to your code at any point, and the python debugger will start immediately.

You can run any script with the python debugger.:

```
$ python3 -m pdb <path_to_python_script.py>
...
(Pdb)$ run
(Pdb)$ continue
```

Profiler

We recommend to use Plop[262]. Plop[263] comes in two modules, one needed for collection of data, and the other one use to present the data. Here is how to use it:

```
$ su - -c 'pip install plop'
$ python3 -m plop.collector <script.py>
$ python3 -m plop.viewer --datadir=profiles
```

The output of Plop[264] is a very nicely constructure graph view of the calls. The size of the bubble indicate the ammount of time spend of the method, and the arrows indicate the backtrace to it.

[262] https://github.com/bdarnell/plop
[263] https://github.com/bdarnell/plop
[264] https://github.com/bdarnell/plop

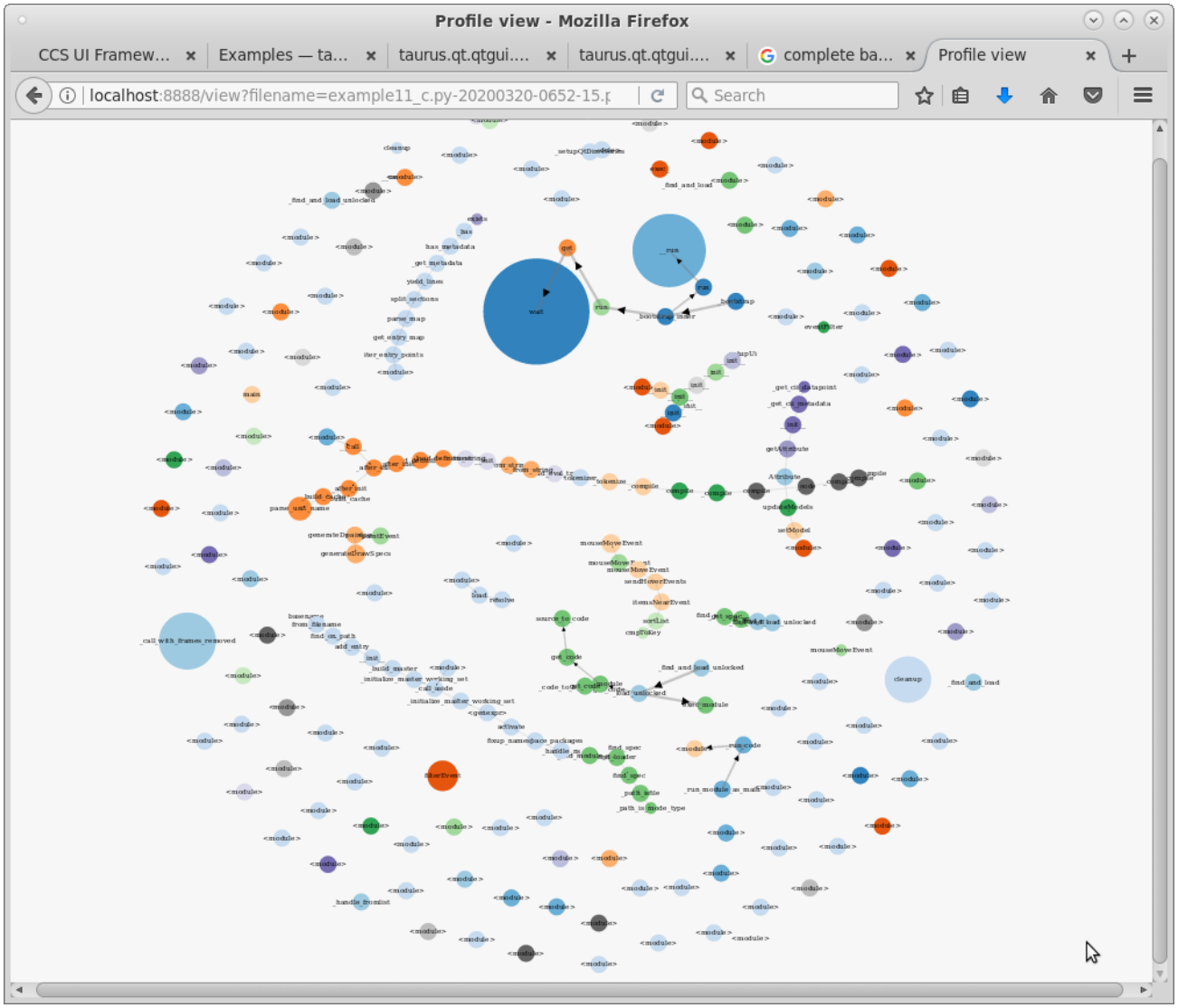


Fig. 2: Plop viewer in action. It presents the information in a dynamic and interactive graph plot.

8 GUI Testing

A collection of details for creating tests will be collected here. In a later version a guide on how to create test cases will be introduced

8.1 pytest-qt caveats

Some details when dealing with pytest-qt are important.

Timeout on tests but the exact time does not pass, do not call QApplication.quit()

I have run into this once. pytest-qt reported a timeout, but if I increased the timeout in the QtBot.waitForSignal() method, the tests did not take longer, and still, the timeout error was thrown.

After disabling part of the test suite, I realized the error came from executing QApplication.quit() in a previous test. The timeout was generated from the waitForSignal() method, but from pytest-qt communication with its instance of QApplication.

While the test itself did not call QApplication.quit(), it closed the MainWindow, and its closeEvent() method did execute QApplication.quit(). As a general recommendation, don't call it but instead call the parent (QApplication.closeEvent()) method, so that the normal workflow of qt takes care of this instead. Just handle the resources deallocation and disconnection.

9 Color Schemes

The developer should avoid setting color schemes or palettes manually. The ELT project includes in their guideles a set of color schemes that may change in the future; so in order to make every application adaptable to future color schemes, these are provided as configuration files in the system.

A particular color scheme configuration in CUT is one configuration files with the extension `.color`. The format of these files is an INI based file, that follows the same format as KDE to specify their color palette.

Nonetheless, the developer should not interact with these files. Instead, two artefacts are available to the developer to access the palettes.

9.1 QPalette

A Qt application when started will read the XDG_CURRENT_DESKTOP environment variable to determined where it should get its color palette from. From the desktop environment configuration entries will get the colors, and create a QPalette object and set it as the Application wide palette object.

QPalette is a data class in Qt. This means that it only contains data, and is not a QObject.

The QPalette separates the colors into three groups (ColorGroup)

ColorGroups

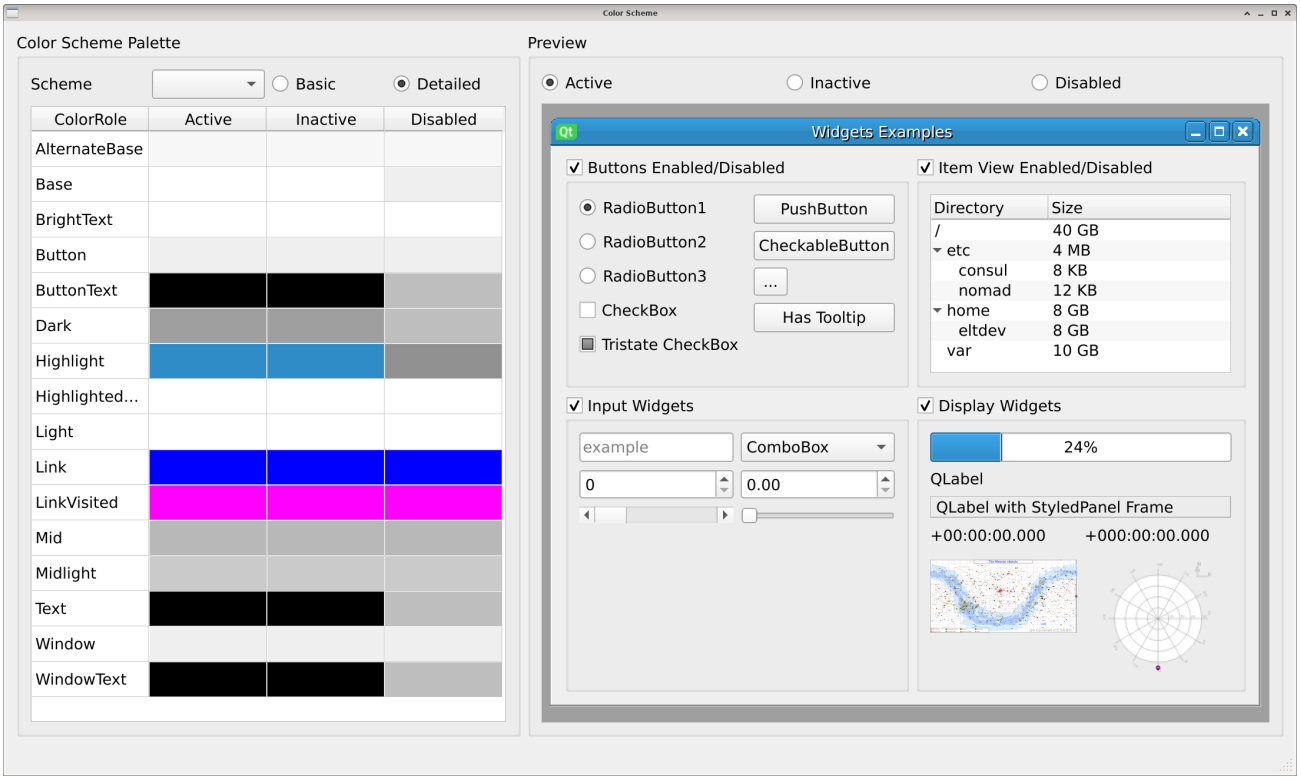
Active
: When a window is selected, this is the color palette applied to it. It is the most used palette.

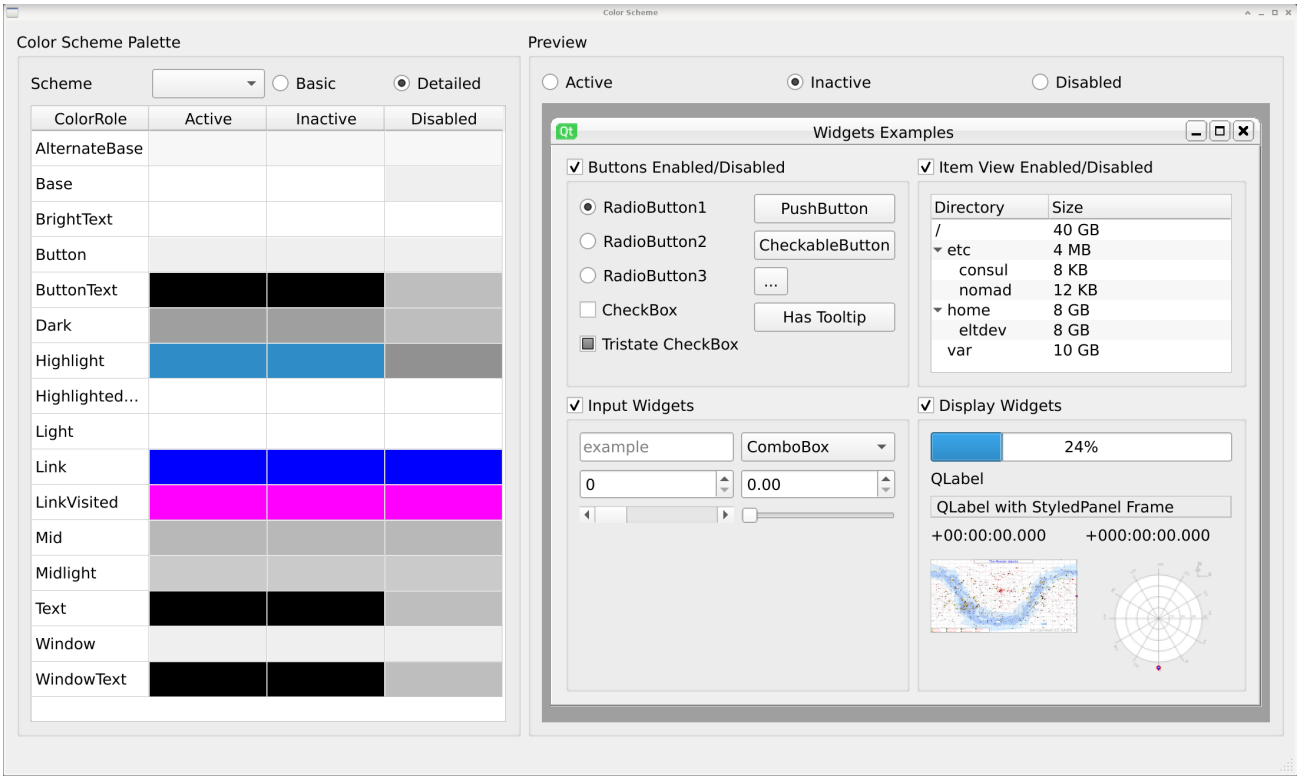
Inactive
: When the window looses focus, the application detects this, and applies this color palette. Normally is looks very similar to Active, except for the window title. Due to this particularity this ColorGroup is only visible when using QMdiArea, the only Widget that creates window titles

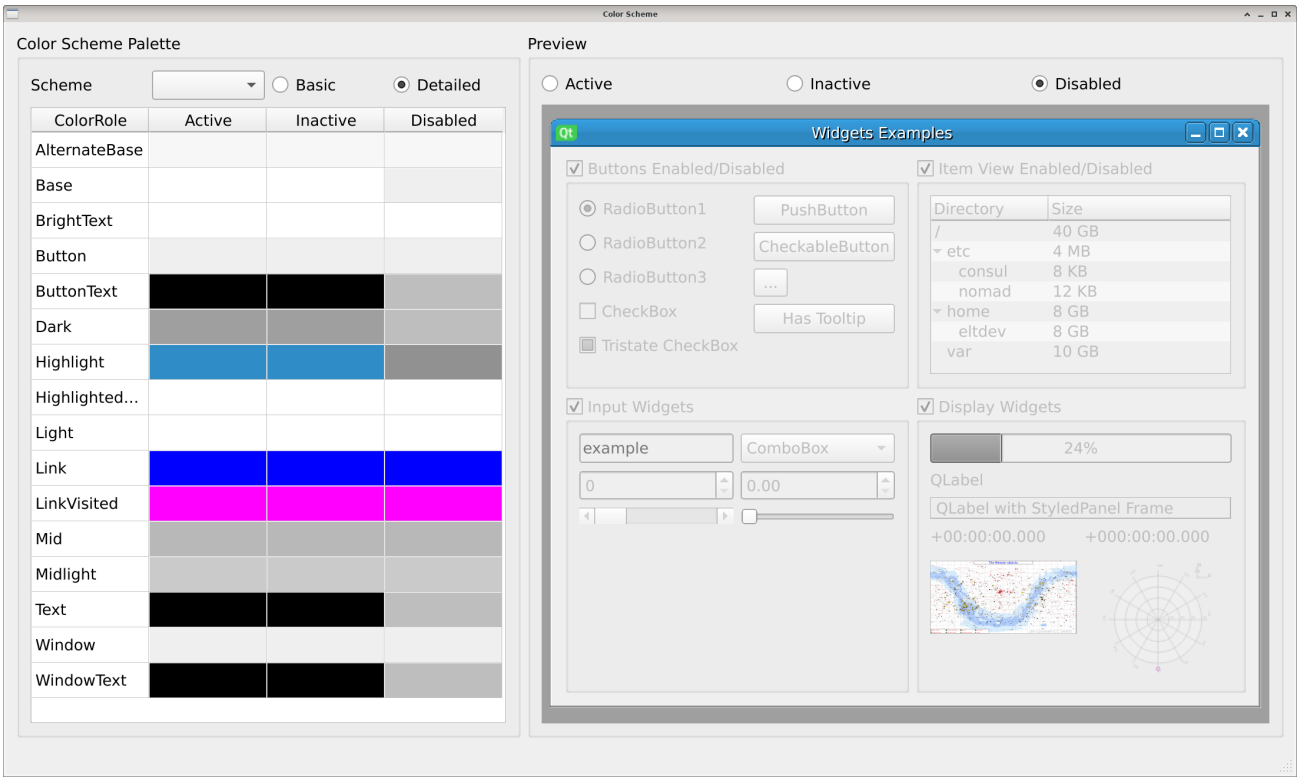
---

Tip: Window decoration under Linux is responsability of the Window Manager, a separate application provided by the Desktop Environment. Though not recommended, if you were to create a window-less application, it is recommended that accent colors are respected for the window title, otherwise a user will not know when the application has lost focus.

Disabled
: Developer can mark widgets (and their children) as disabled. This cuts interaction with widgets, and presents the widgets as gray or a lighter color. This communicates to the user that those widgets are no longer available. The colors uses for widgets come from the Disabled ColorGroup.


Color Scheme
Color Scheme Palette
Scheme
Basic
Detailed
ColorRole
Active
Inactive
Disabled
AlternateBase
Base
BrightText
Button
ButtonText
Dark
Highlight
Highlighted...
Light
Link
LinkVisited
Mid
Midlight
Text
Window
WindowText
Preview
Active
Inactive
Disabled
Widgets Examples
Buttons Enabled/Disabled
RadioButton1
RadioButton2
RadioButton3
CheckBox
Tristate CheckBox
PushButton
CheckableButton
...
Has Tooltip
Item View Enabled/Disabled
Directory
Size
/ 40 GB
etc 4 MB
consul 8 KB
nomad 12 KB
home 8 GB
eltdev 8 GB
var 10 GB
Input Widgets
example
ComboBox
0
0.00
Display Widgets
24%
QLabel
QLabel with StyledPanel Frame
+00:00:00.000
+000:00:00.000


Color Scheme
Color Scheme Palette
Scheme
Basic
Detailed
ColorRole
Active
Inactive
Disabled
AlternateBase
Base
BrightText
Button
ButtonText
Dark
Highlight
Highlighted...
Light
Link
LinkVisited
Mid
Midlight
Text
Window
WindowText
Preview
Active
Inactive
Disabled
Widgets Examples
Buttons Enabled/Disabled
RadioButton1
RadioButton2
RadioButton3
CheckBox
Tristate CheckBox
PushButton
CheckableButton
...
Has Tooltip
Item View Enabled/Disabled
Directory
Size
/ 40 GB
etc 4 MB
consul 8 KB
nomad 12 KB
home 8 GB
eltdev 8 GB
var 10 GB
Input Widgets
example
ComboBox
0
0.00
Display Widgets
24%
QLabel
QLabel with StyledPanel Frame
+00:00:00.000
+000:00:00.000

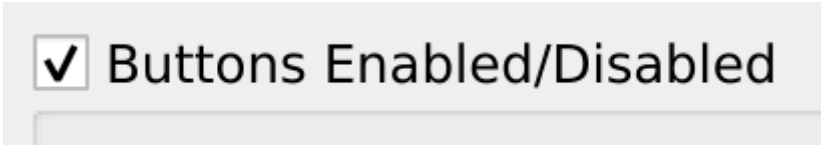


ColorRoles

The colors (ColorRole) offered by each Color Group are the following ones:

Window
: This is the color used for the background of the application. In the examples above is color similar to white. Most of the application and dialogs will use this color as background. An important thing to notice is that widgets will only draw over their borders and will not paint their background. QLabel for example, only draws text over the background of the widget.

WindowText
: This is the foreground color of the Window color. Most colors come in pairs. One represents background (Window), and another foreground (WindowText). This allows to implement the necessary contrast when creating QPalettes.



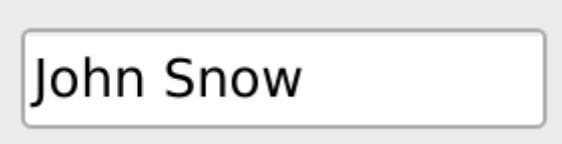
Tip: Color schemes from CUT already consider contrast.

Base

Certain widgets implies an editable area or dynamic data presentation by using a different background color. QLineEdit and QTreeWidget use Base as their background color.

Text

If Base is the background, Text is used to render text foreground. Please refer to the figure below to see how a QLineEdit uses as background color Base, and Text to render fonts.



PlaceholderText

User by QLineEdit as example text color. In the figure below a QLineEdit uses PlaceholderText color to render examples.

Button

ColorRole used to identify the color for Buttons background. Other widgets may use it as well, such as the ComboBox.

ButtonText

Foreground color of the pair Button, ButtonText. Used to render fonts over a button. In the figure below, both the Button and ButtonText roles are used.

Dark, Mid, Midlight and Light

Are roles used to create a sense of depth in the GUI. Their are used in QHorizontalSlider, QScrollBar, to “sink” part of their representation, and “raise” the draggable part of the widget. Similarly, Tables, and Tree widgets use them to render separation lines.

In the figure below: A QLineEdit, QComboBox, QSpinBox and QDoubleSpinBox all use the same color Mid to render a border line that indicates where the widgets area starts. Dark is used to render the triangle that indicates that interaction is posible. The QScrollBar (bottom left) uses Mid for its sunken section of the widget, over which the draggable components slides. The QHorizontalSlider (bottom right) uses Dark to render an inset-rail, over which the draggable component is rendered with Light.

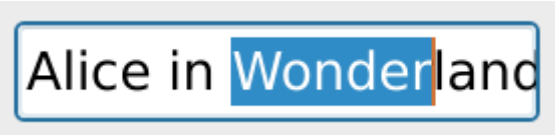
ToolTipBase

Background color for tooltips.

ToolTipText
Text color for tooltips.

Highlight
When selecting text in a QTextEdit widget, or selecting cells in a QTableWidget this color is used to draw the background. This ColoRole provide high contrast against Base and HightlightText.

HighlightText
Color of the font when selecting text or cells. Hight contrast from both Highlight and Text is provided by this ColorRole.



All colors are accesible via the *QPalette* `color()` method.

```cpp
#include <QPalette>
#include <QColor>

// ...

void AWidget::methodInAWidget(){
    // Request color by Group and Role
    QColor color = this->palette().color(QPalette::Active, QPalette::WindowText);
    // Request color of the current Group (most used method)
    QColor color2 = this->palette().color QPalette::WindowText);
}
```

```python
from PySide2.QtCore import QColor
from PySide2.QtWidgets import QPalette
from PySide2.QtWidgets import QWidget

// ...
```

(continues on next page)

(continued from previous page)

```
class AWidget(QWidget):

    # ...

    def method_in_a_widget(self):
        # Request color by Group and Role
        color = self.palette().color(QPalette.Active, QPalette.WindowText);
        # Request color of the current Group (most used method)
        color2 = self.palette().color(QPalette.WindowText);
    }
```

9.2 QeColorsModel

A model based on QStandardItemModel which list all color scheme found in the system. It looks for color schemes in the following directories and stores their QPalettes in memory for easy retrieval.

- `/usr/share/color-schemes`
- `/elt/cut/resource/cut/color-schemes`
- `$INTROOT/resource/cut/color-schemes`

9.3 QeColorsMenu

Uses QeColorsModel to create a QMenu that list the available color scheme, and when an entry is selected, applies the QPalette to the application.

10 Standalone Tools

10.1 cutWidgetsShowcase and cutWidgetsShowcasePy

Exploratory tool that shows CUT widgets and how their properties influence their presentation.

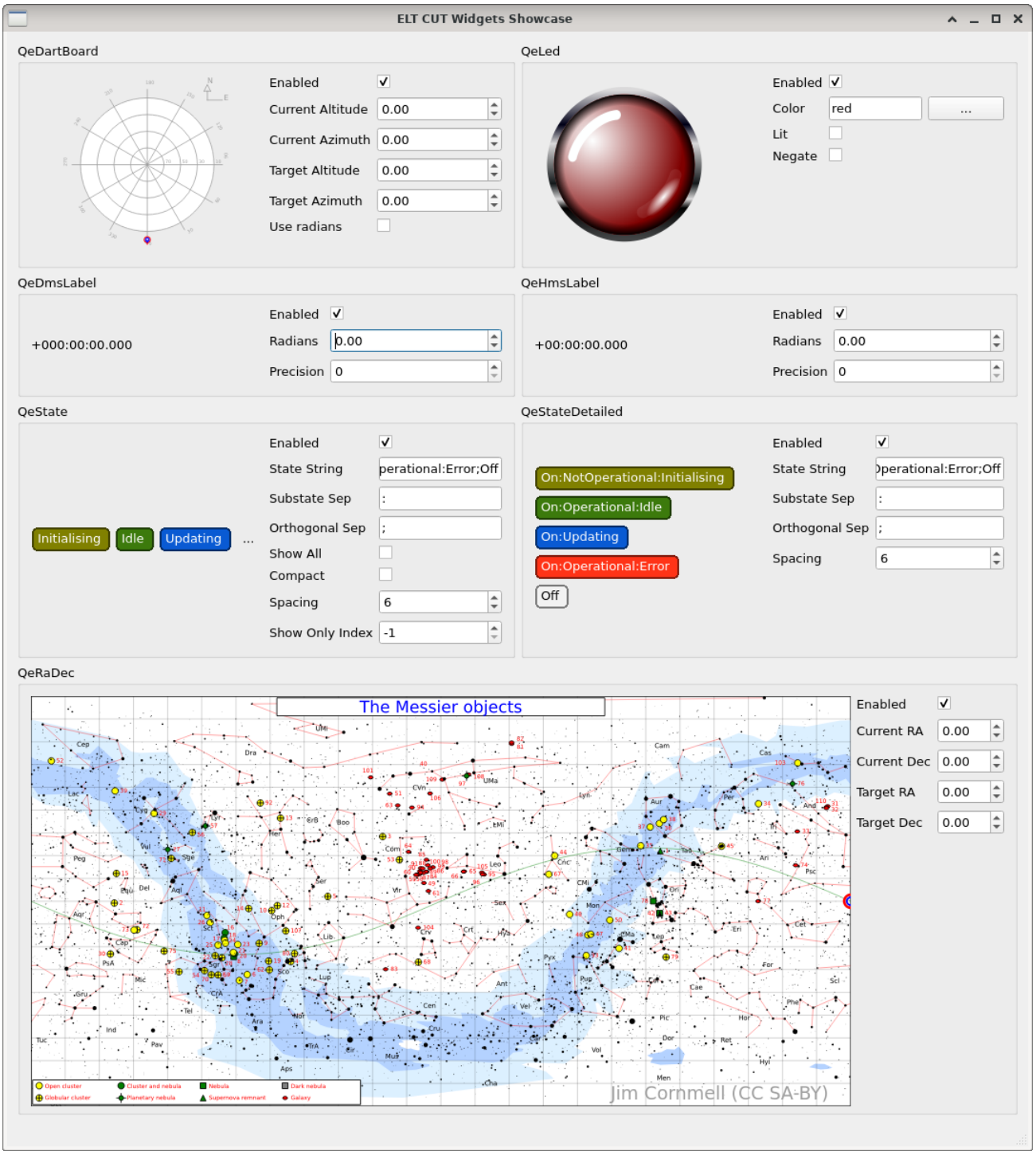
10.2 cutExampleColorScheme and cutExampleColorSchemeCpp

Allows the user to explore how the provided Color Scheme looks like.

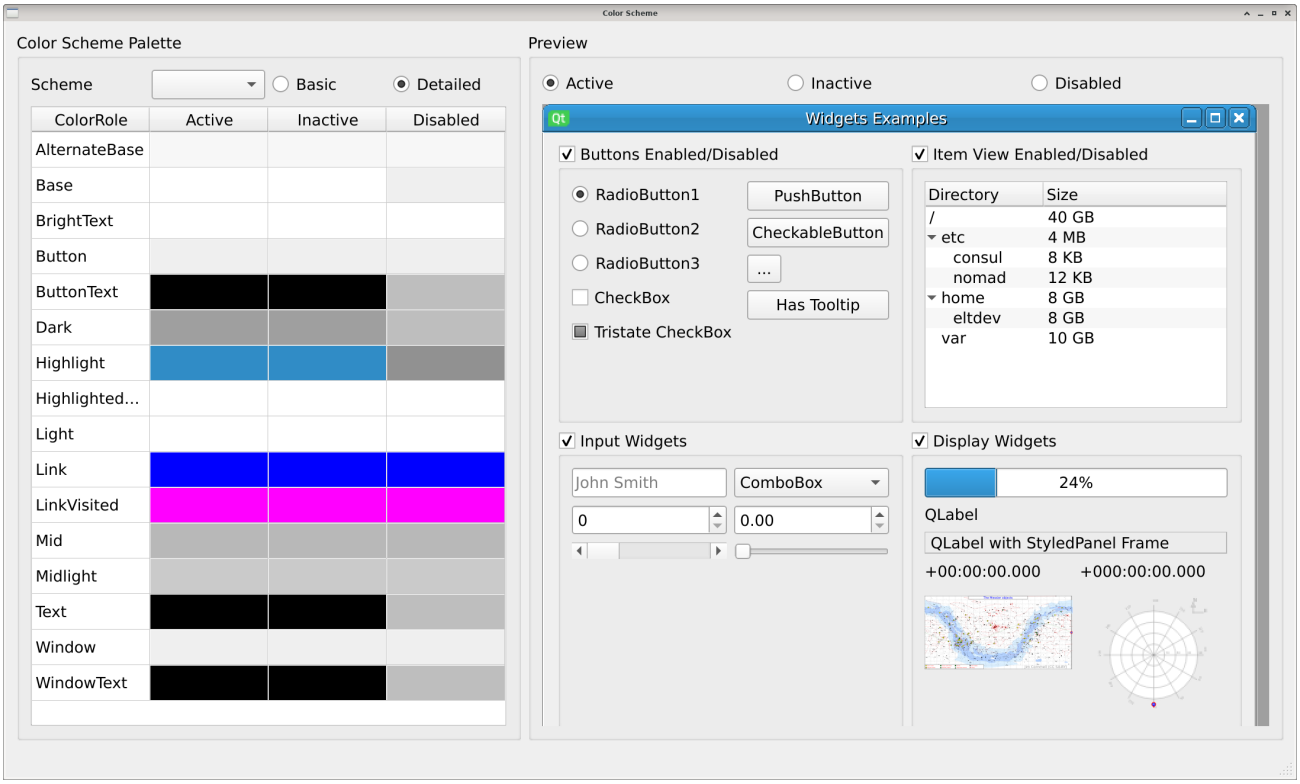
10.3 cutDemoService

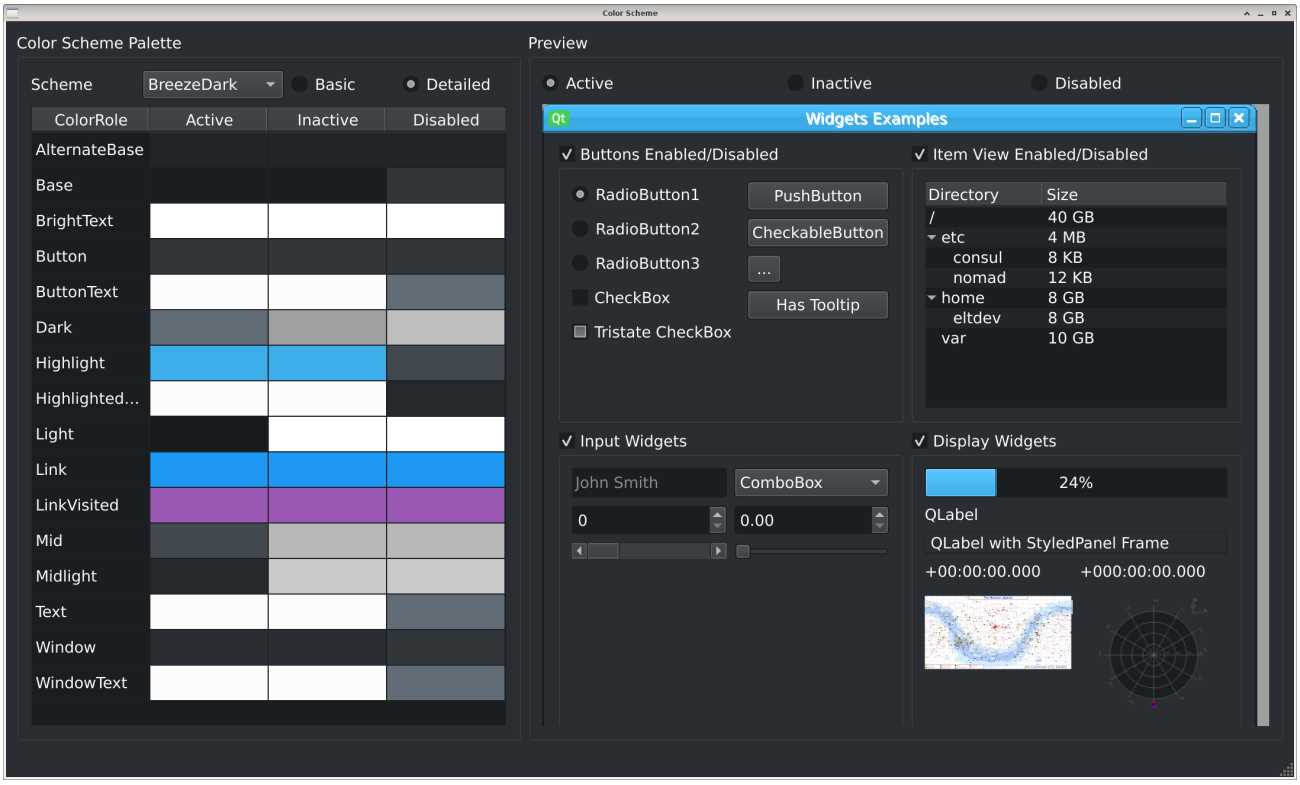
Demonstration applicaiton that create and updates several datapoints. It is used to demonstrate the declarative databinding capabilities of CUT, but it can also be used to quickly provide a set of datapoits for other testing purposes.

See *cutDemoService* for details on the cutDemoService usage.


ELT CUT Widgets Showcase
QeDartBoard
Enabled
Current Altitude 0.00
Current Azimuth 0.00
Target Altitude 0.00
Target Azimuth 0.00
Use radians
QeLed
Enabled
Color red
Lit
Negate
QeDmsLabel
+000:00:00.000
Enabled
Radians 0.00
Precision 0
QeHmsLabel
+00:00:00.000
Enabled
Radians 0.00
Precision 0
QeState
Initialising
Idle
Updating
Enabled
State String perational:Error;Off
Substate Sep :
Orthogonal Sep ;
Show All
Compact
Spacing 6
Show Only Index -1
QeStateDetailed
On:NotOperational:Initialising
On:Operational:Idle
On:Updating
On:Operational:Error
Off
Enabled
State String)perational:Error;Off
Substate Sep :
Orthogonal Sep ;
Spacing 6
QeRaDec
The Messier objects
Jim Cornmell (CC SA-BY)
Enabled
Current RA 0.00
Current Dec 0.00
Target RA 0.00
Target Dec 0.00


ELT CUT Widgets Showcase
QeDartBoard
Enabled
Current Altitude 0.00
Current Azimuth 0.00
Target Altitude 0.00
Target Azimuth 0.00
Use radians
TaurusDartBoard
Enabled
Model eval:rand(4)
Current Altitude 0.43
Current Azimuth 0.02
Target Altitude 0.97
Target Azimuth 0.36
QeDmsLabel
+000:00:00.000
Enabled
Radians 0.00
Precision 0
TaurusDmsLabel
+016:02:34.146
Enabled
Model eval:rand()
Radians 0.28
Precision 0
QeHmsLabel
+00:00:00.000
Enabled
Radians 0.00
Precision 0
TaurusHmsLabel
+01:04:10.276
Enabled
Model eval:rand()
Radians 0.28
Precision 0
QeState
Initialising Idle Updating ...
Enabled
State String Operational:Error;Off
Substate Sep :
Orthogonal Sep ;
Show All
Auto Compact
Spacing 6
Show Only Index -1
TaurusState
Initialising Idle Updating ...
Enabled
Model perational:Error;Off")
State String Operational:Error;Off
Substate Sep :
Orthogonal Sep ;
Show All
Auto Compact
Spacing 6
Show Only Index -1
QeStateDetailed
On:NotOperational:Initialising
On:Operational:Idle
On:Updating
On:Operational:Error
Off
Enabled
State String :Operational:Error;Off
Substate Sep :
Orthogonal Sep ;
Spacing 6
TaurusStateDetailed
On:NotOperational:Initialising
On:Operational:Idle
On:Updating
On:Operational:Error
Off
Enabled
Model perational:Error;Off")
State String :Operational:Error;Off
Substate Sep :
Orthogonal Sep ;
Spacing 6


Color Scheme
Color Scheme Palette
Scheme
Basic
Detailed
ColorRole
Active
Inactive
Disabled
AlternateBase
Base
BrightText
Button
ButtonText
Dark
Highlight
Highlighted...
Light
Link
LinkVisited
Mid
Midlight
Text
Window
WindowText
Preview
Active
Inactive
Disabled
Widgets Examples
Buttons Enabled/Disabled
RadioButton1
RadioButton2
RadioButton3
CheckBox
Tristate CheckBox
PushButton
CheckableButton
...
Has Tooltip
Item View Enabled/Disabled
Directory
Size
/ 40 GB
etc 4 MB
consul 8 KB
nomad 12 KB
home 8 GB
eltdev 8 GB
var 10 GB
Input Widgets
John Smith
ComboBox
0
0.00
Display Widgets
24%
QLabel
QLabel with StyledPanel Frame
+00:00:00.000
+000:00:00.000


Color Scheme
Color Scheme Palette
Scheme
BreezeDark
Basic
Detailed
ColorRole
Active
Inactive
Disabled
AlternateBase
Base
BrightText
Button
ButtonText
Dark
Highlight
Highlighted...
Light
Link
LinkVisited
Mid
Midlight
Text
Window
WindowText
Preview
Active
Inactive
Disabled
Widgets Examples
Buttons Enabled/Disabled
RadioButton1
RadioButton2
RadioButton3
CheckBox
Tristate CheckBox
PushButton
CheckableButton
...
Has Tooltip
Item View Enabled/Disabled
Directory
Size
/ 40 GB
etc 4 MB
consul 8 KB
nomad 12 KB
home 8 GB
eltdev 8 GB
var 10 GB
Input Widgets
John Smith
ComboBox
0
0.00
Display Widgets
24%
QLabel
QLabel with StyledPanel Frame
+00:00:00.000
+000:00:00.000

11 API

The API documentation is maintained in code using doxygen.

Please refer to CUT API[265] for more information.

[265] https://eltjenkins.hq.eso.org/job//ECOS/job/CUT/job/CUT-Build-Master-Platform-26-DevEnv/lastSuccessfulBuild/artifact/build/docs/html/index.html

12 Frequently Asked Question

12.1 Python Bindings

How do I use the widget?

The typesettings file (the XML in the Input section), includes the name of the Python package. Next, you need to know the name of the class, which is the same as in C++. A very simple application could be:

```
import sys
from taurus.external.qt.QtWidgets import QApplication
from QeWidgetExample import QeDartBoard

app = QApplication()
dartboard = QeDartBoard()
dartboard.show()
sys.exit(app.exec_())
```

If I connect a signal to one of the widget's slots, it does nothing.

Most likely, the `bindings.h` file does not include the sentence to make visible the classifiers (signal, slots keywords in the C++ header).

```
#define QT_ANNOTATE_ACCESS_SPECIFIER(a) __attribute__((annotate(#a)))
```

Remember Shiboken2 needs to pass this instruction to the C++ preprocessor in order to generate the code.

Source: https://blog.basyskom.com/2019/using-shiboken2-to-create-python-bindings-for-a-qt-library/

PySide::Signal namespace not found

Compiler outputs an error like this one:

```
error: 'PySide::Signal' has not been declared
```

Most likely, you have not activated in Shiboken2 executable, the option that enables support for signal and slots. Shiboken is a fully capable binding generator, and Qt's is one of the things it does. PySide support is not enabled by default. Check if your generation script includes this option for the Shiboken2 executable:

```
--enable-pyside-extensions
```

Source: https://github.com/ros-visualization/qt_gui_core/issues/142

If I connect a signal to one of the widget's slots and trigger it, the application crashes with a Segmentation Fault.

Most likely, you have not activated in Shiboken2 executable, the option that enables support for signal and slots. Shiboken2 is a fully capable binding generator, and Qt's is one of the things it does. PySide support is not enabled by default. Check if your generation script includes this option for the Shiboken2 executable:

```
--enable-pyside-extensions
```

A symbol is missing (a method), and it is from my widgets:

This happens when you generate the code for all the widgets you want to, but forget to include them in the final linking of the Python shared library that acts as a Python Package. Check in the wscript that does the compilation, if all <widget_class_name>_wrapper.cpp are listed as target for the generation task, and listed as sources in the compilation task.

Application crash on startup, indicates Qt_5_6 version is used, but it was compiled against Qt_5_14

DevEnv comes with PyQt installed (provided by anaconda) and it is used by multiple applications. But the compilation procedure will resolve against Qt5.14, if every dependency is declared.

In your project, you have to manually configure the dependencies, and then use them in widgets, and in bindings.

For example, in the project (first) wscript:

```
def configure(cnf):
    for pkg in 'Qt5Core Qt5Gui Qt5Widgets'.split():
        cnf.check_cfg(package=pkg, uselib_store=pkg)
```

And then in your wscript of the widget:

```
declare_qtcshlib( target='widgetexample-widgets',
                  use='QT5CORE QT5GUI QT5WIDGETS'
)
```

These are just snipper of code. Take a look into the example in https://gitlab.eso.org/ahoffsta/cut/-/tree/master/examples/widget_example for complete wscripts.

My application crashes, in a section that is C++, how can I get a back trace

You need to start the python interpeter with GDB:

```
gdb
file python
run <path_to_python_script>
```

When the application crashes, type in the GDB console:

```
bt
```

12.2 Widget Library

Why the separation between the widget and plugin library

The Plugin library is only used by Qt Designer. When compiling an application, only the header file and the widgets library themselves are only needed. The Qt Designer Plugins only provides presentation in the Qt Designer for the plugin, and a piece of XML to be included in the UI File. This piece of XML include the header and Class for the include statement.

Also, these library are expected to be found in different places. Qt Designer needs them to be in a `designer` directory to find them, while the widgets needs to be in `LD_LIBRARY_PATH` for the application to find them. Certainly one can point `LD_LIBRARY_PATH` to `$INTROOT/lib64/designer`, but this is not encouraged.

This way you can also debug and developed modularly the widgets. When you have the widget code ready, then you focus on the plugin aspect of it. Since plugin code is very similar to all plugin, then this is very straightforward. As a recommendation, I would suggest to add a fourth module: a **showcase** application, that allows you to see and interact with the widgets. Think of it as a "demo". This showcase application should only link to the **widgets** module, but its UI file should be constructed using Qt Designer, using the provided plugins. If the plugin is not ready, you can use Widget Promotion ref:*widget_promotion* to transform one widget into another that does not have a plugin.

The widget plugin does not load

Try using `designer` command. This will start the Qt Designer in standalone mode. In the main window *Menu > Help > Plugins*, the application will show you a dialog with information of the plugins found, and why they failed to load. If the plugin is not listed, then the `QT_PLUGIN_PATH` is not set.:

```
QT_PLUGIN_PATH=$INTROOT/lib64:$QT_PLUGIN_PATH designer
```

What method of library is used for bindings

Shiboken2. At the moment this is not supported by wtools, but support will come soon.

Why is there a header called "libraryname.h"

While creating python bindings, a module name must be given to the resulting python module (bindings.xml file has this information).

The widgets plugin return through the *name()* method the name of the class used to create the widget Object. The *includeFile()* method returns the C++ header file to put into the generated code `#include` statement.

`uic-qt5` –the tool that generates code from UI file– uses the same plugin methods to get the name of the include and import statements. If the names differs (between the Python bindings module and the C++ include), either the C++ or Python version of the resulting generated code for the UI file would fail.

Example 1: We do not have the libraryname.h file:

- C++ Class name: widgetnameone
- C++ header file: widgetnameone.hpp
- Python Class name: widgetnameone
- Python module: libraryname

Then, the resulting include and import statements would be:

```
#include <widgetnameone.hpp>
From widgetnameone.hpp import widgetnameone
```

`pyside2-uic` cuts the last two characters from the C++ header file, and uses the name of the file as the module it should include. In this case, the python import fails.

Example 2: We create the `libraryname.h` file, and include other widgets here:

- C++ Class name: widgetnameone
- C++ header file: libraryname.h
- Python Class name: widgetnameone
- Python module: libraryname

Then, the resulting include and import statements would be:

```
#include <libraryname.h>
From libraryname import widgetnameone
```

libraryname.h? why not libraryname.hpp?

Code generation in Qt PySide2, only uses names that ends in `.h` or without any extension. If the file has any other suffix, then the whole string is used. This would cause that under Python, the UIC generated code would look like:

```
from libraryname.hpp import WidgetNameOne
```

I have created a templated QObject, but it does not compile. What am I doing wrong?

> **Error:** An important limitation of QObject is that it cannot include templates, either in the class or in a method.

Developers may mitigate the need for templates by using QVariant, which is a class that acts as a union, but is aware of its datatype and can be queried about it.

Also, a series of signals or slots required, each with the needed datatype can be created.

Source: https://stackoverflow.com/questions/4397478/qt-templated-q-object-class

paintEvent() is not called in python

This can happen to any xyzEvent() method. The cpp widget works, but the python bound version does not respond to events.

When overriding a *xyzEvent(QEvent *)* method in cpp, the python bindings needs to be told about this. Adapt this example to you typesystem file:

```
<object-type name="AWidget">
<modify-function signature="resizeEvent(QResizeEvent*)">
    <modify-argument index="1" invalidate-after-use="yes">
        <rename to="event"/>
    </modify-argument>
</modify-function>
</object-type>
```

12.3 Qt in General

How to start a process and monitor if it is still running?

QProcess[266] is the class you are looking for. It allows to start a second process as a child to your QApplication.

It can start an executable. It has a signal that changes when the state of the process changes. You can use it to changes the state of a QLed or QLabel to indicate was is the current state of the monitored process. This is particularly useful when a console process is launched, and it has no GUI.

It is recommended to use QProcess.start() instead of QProcess.startDetached(). The second one does not monitor not update the signal.

If your use case needs to keep the process running after your exit the QApplication , add this to the class that holds the QProcess object:

```
def __del__(self):
    # This allows a process started by QProcess.start()
    # to be detached when the QApplication exits.
    self._qprocess_object.setProcessState(QProcess.NotRunning)
```

This will detach the second process upon exiting the QApplication.

NOTE: details of implementation at https://code.qt.io/cgit/qt/qtbase.git/tree/src/corelib/io/qprocess.cpp#n1311. Here the reader can see that only when the second process state is different from NotRunning, the kill() command is executed.

[266] https://doc.qt.io/qt-5/qprocess.html

13 Qt5 to Qt6 (DevEnv5 to DevEnv6) Migration Guide

Qt has two big families of Widgets: Quick/Qml and QtWidgets. The Control UI Guidelines indicates that for ELT GUIs developers should only use widgets that belong QtWidgets family of Widgets. They are stable, and their code receive only fixes. Quick/Qml are modern and are prone to changes. It is becoming a stable library, but its adoption pace of new features is fast.

QtWidgets module has few updates in their port to Qt6 than Quick/Qml, so the migration is relatively easy.

C++ major changes:

- QList and QVector have a new implementation, and QVector is a typedef to QList. **IMPORTANT**
- QProcess::start have changed its behaviour.
- QtOpenGL module and OpenGL classes have been restructured.
- QtSvg module have been separated into two: QtSvg and QtSvgWidgets.
- QtGui: QBitmap: Implicit construction of a QBitmap[267] from a QPixmap[268] is no longer supported
- QtGui: Affected by the QtOpenGL module restructure. [https://doc.qt.io/qt-6/portingguide.html{]}(https://doc.qt.io/qt-6/portingguide.html)

Python mayor changes:

- Enums: Python3 recommends the use of fully qualified names to identify the values of enumerations. Avoid the use of self.<enum_value>, and instead use <EnumerationName>.<enum_value>. Taurus and PySide6 has compatibility implemented with the previous notation, but this is a recommendation from python coding standards.
- **exec**: no longer a python reserved keyword. (QApplication.exec_() to QApplication.exec()). Taurus and PySide6 still includes the older notation for compatibility, but you should move to the newer notation.
- Please move from 'import PySide2' to 'import taurus.external.qt': this will make your code compatible with both current versions, and future versions of Qt. If your code cannot depend on taurus (ifw-sequencer), then replace for 'import PySide6'.

[267] https://doc.qt.io/qt-6/qbitmap.html
[268] https://doc.qt.io/qt-6/qpixmap.html

13.1 wscript (Project)

TIP: An updated version of the python_application example project is available at: https://gitlab.eso.org/ecos/cut/-/blob/master/examples/python_application, along with the DevEnv5 version in its respective maintenance branch: https://gitlab.eso.org/ecos/cut/-/blob/2.x/examples/python_application/wscript?ref_type=heads (useful for comparison)

To start your migration, you first stop is the project's wscript.

declare_project() has two mayor changes:

- **requires** keyword argument now uses a version-less qt string to add it to waf configuration
- **qt** new keyword argument that configures the version of qt to use.

DevEnv5 version

```
declare_project(
    'qt-widgets',
    version='5.0.0-pre3',
    requires='cxx qt5 boost gtest python pyqt5',
    recurse='cpp',
    boost=dict(libs='filesystem'),
)
```

If you want to keep compatibility Qt5 compatibility, please use the following solution.

DevEnv5/6 version

```
def qtcallable(cnf):
    # We can do some checks here, for example to discriminate between Qt6 and Qt5:
    if os.environ.get("ELT_RELEASE", "6").startswith("6"):
        return dict(version=6)
    else:
        return dict(version=5)


declare_project(
    'qt-widgets',
    version='5.0.0-pre3',
    requires='cxx qt boost gtest python pyqt',
    recurse='cpp',
    boost=dict(libs='filesystem'),
    qt=qtcallable
)
```

13.2 wscript (Configuration)

In case your configuration section of the project configure extra Qt libraries, please replace the old for the new versions:

DevEnv5

```
cnf.check_python_module('PySide2', condition="ver >= num(5, 14, 1)")
for pkg in 'Qt5Svg Qt5Designer shiboken2'.split():
    cnf.check_cfg(package=pkg, uselib_store=pkg)
```

DevEnv6

```
cnf.check_python_module('PySide6')
for pkg in 'Qt6Svg Qt6Designer shiboken6'.split():
    cnf.check_cfg(package=pkg, uselib_store=pkg)
```

NOTE: you then have to changes the **use** statements in the module where the library is included/linked.

DevEnv5/6

We use version-agnostic uselib_store variables.

```
qt_pkgs = ''
if hasattr(cnf, "want_qt6") and cnf.want_qt6 is True:
    qt_pkgs = 'shiboken6 pyside6'
    cnf.check_python_module('PySide6', condition="ver >= num(6, 0, 0)")
    cnf.check_cfg(package="Qt6Svg", uselib_store="QTSVG", args='--cflags --libs')
    cnf.check_cfg(package="Qt6Designer", uselib_store="QTDESIGNER", args='--cflags -
↪-libs')
else:
    qt_pkgs = 'shiboken2 pyside2'
    cnf.check_python_module('PySide2', condition="ver >= num(5, 14, 1)")
    cnf.check_cfg(package="Qt5Svg", uselib_store="QTSVG", args='--cflags --libs')
    cnf.check_cfg(package="Qt5Designer", uselib_store="QTDESIGNER", args='--cflags -
↪-libs')
for pkg in qt_pkgs.split():
    cnf.check_cfg(package=pkg, uselib_store=pkg, args='--cflags --libs')
```

NOTE: you then have to changes the **use** statements in the module where the library is included/linked.

13.3 wscripts (modules)

All auto wtools method to declare Qt applications and modules have dropped the number from the method name. This change has been introduced also in DevEnv5, to allow a period of time to implemented these changes.

These require modifications.

- declare_qt5cshlib() -> declare_qtcshlib()
- declare_qt5cprogram() -> declare_qtcprogram()
- declare_pyqt5package() -> declare_pyqtpackage()
- declare_pyqt5program() -> declare_pyqtprogram()

In a similar manner, C++ shared libraries that compile into a Qt Designer plugin required a special keywork argument. This also has to drop the number in their name:

DevEnv5 wscript

```
declare_qtcshlib(target='elt-qt-widgets-plugin',
                 use='cpp.elt-qt-widgets.widgets',
                 qt5_plugin=True)
```

DevEnv5/6 wscript

```
declare_qtcshlib(target='elt-qt-widgets-plugin',
                 use='cpp.elt-qt-widgets.widgets',
                 qt_plugin=True)
```

13.4 Python Code changes

Your .py files require that you change the import statements from using directly PySide2 to the use of taurus external qt module. This module has the same structure that the PySide2 modules, and makes the adoption of future versions of Qt easier.

import PySide2 -> import taurus.external.qt

The declaration of signals has also become stricter. PySide2 allowed to use tuples or lists to declare a signal with multiple argument type signatures. See the following example:

```
my_signal = Signal([int], [int, str])
```

PySide6 now requires this to be a tuple. Using a list can lead to runtime exceptions being thrown indicating the signature cannot be found. For the above example, the code should be adjusted as follows:

```
my_signal = Signal((int,), (int, str))
```

When a class has multiple inheritance, and one of them is a QObject, the following issue could appear, depending on the `__init__` implementation:

```
QObject.__init__(self, parent=parent, *args, **kwargs)
TypeError: Logger.__init__() got an unexpected keyword argument 'parent'
```

Code from DevEnv5 that looks like this:

```
class NewClass(QObject, Logger):

    def __init__(self, parent=None):
        QObject.__init__(self, parent=parent)
        Logger.__init__(self)
```

In DevEnv6 we recommend to remove the parent keyword while invoking the QObject's init method.

```
class NewClass(QObject, Logger):

    def __init__(self, parent=None):
        QObject.__init__(self, parent)
        Logger.__init__(self)
```

This is a workaround, as the proper solution (using `super()`) yields the same error.

QAction has changed its module between Qt5 and Qt6.

```
from taurus.external.qt.QtWidgets import QAction  # DevEnv5
from taurus.external.qt.QtGui import QAction      # DevEnv6 (this is backwards␣
↪compatible)
```

13.5 Python Bindings for Qt Widgets

The Shiboken "generator" script and wscript has been unified into only one wscript. It drops the legacy-hardcoded entries, and instead uses the pkg-config entries provided now by the official Qt RPMs.

Important: This is a mayor change that may require expert help. Please contact Arturo Hoffstadt Urrutia <ahoffsta@eso.org> if you have issues with a Qt widget bound to python (m1-gui, ddt, cii-srv).

The generator.sh script is gone. It used to be in charge of preparing a long list of arguments for the shiboken executable, generating the code from the typesystem files. Then the bindings/wscript compile the C++ products from shiboken into a C++ shared library with Qt libraries.

Now, bindings/wscript do both steps. A generation step prepares the list of arguments using now pkg-config entries it extracts during the configuration part of the wscript. Then next compilation step is very similar to its previous version, but we have eliminated several unnecessary option or replaced then with pkg-config entries.

The new script is explained with details in *Bindings*.

13.6 Others

UI files and rcc files are not affected. There are independent of Qt version, and when setting in the project the qt version to use, the proper version of the code generator from Qt will be used.