Elettra
Sincrotrone
Trieste

Elettra – Sincrotrone Trieste

QTango

A multi threaded framework to develop Tango applications

Giacomo Strangolino

IT programmer at Elettra – Sincrotrone Trieste

Assistant professor 2010-2014, University of Trieste, Faculty of engineering, principles of computer science

mailto: giacomo.strangolino@elettra.eu

Part 0

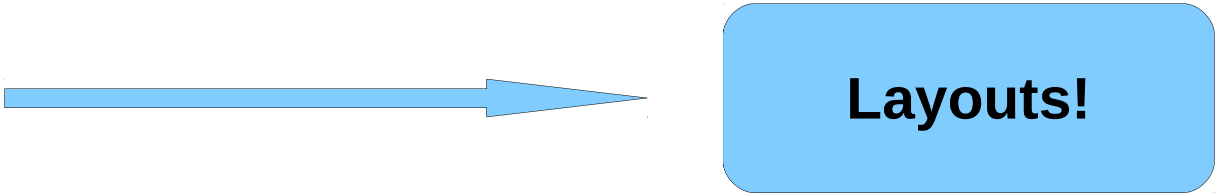
Prerequisites:

The Qt technology

http://www.qt.io/

Prerequisites: Qt

- Qt development libraries installation, Qt *designer* and *qtcreator* IDE, http://doc.qt.io/qt-5/topics-app-development.html;
- http://doc.qt.io/qt-5/gettingstarted.html
- *QPainter* API;
- *QObjects*, Properties, and Events;
- *Signals* and *slots*
- *Qt designer*



Layouts!

Signals and slots

http://doc.qt.io/qt-5/signalsandslots.html

It's probably the part that differs most from the features provided by other frameworks.

In GUI programming, we want objects to communicate with one another. For example, if a user clicks a Close button, we probably want the window's *close()* function to be called.

• Unlike other toolkits, in Qt, a *signal* is emitted when a particular event occurs. A *slot* is a function that is called in response to a particular *signal;*

• Qt's widgets have many predefined signals. We can *subclass widgets to add our own signals and slots*;

• the signature of a signal must match that of the receiving slot (type safety).

• All classes that inherit from *QObject* or one of its subclasses (e.g., *QWidget*) can contain signals and slots.

Slots can be used for receiving signals, but they are also normal member functions. Objects are unaware of possible connections between each other.

QPainter

http://doc.qt.io/qt-5/topics-graphics.html

Graphics in Qt 5 is primarily done either through the *QPainter* API, or through Qt’s declarative UI language, *Qt Quick*.

QPainter provides API for drawing vector graphics, text and images onto different surfaces, or *QPaintDevice* instances, such as *QImage*, *QOpenGLPaintDevice*, *QWidget*, and *QPrinter*. The actual drawing happens in the *QPaintDevice*'s *QpaintEngine*.

- drawImage
- drawText
- drawLines
- drawEllipse
- drawRect
- …

http://doc.qt.io/qt-5/qpainter.html

QObject

http://doc.qt.io/qt-5/topics-core.html

The QObject class forms the foundation of Qt's object model and is the parent class of many Qt classes.

- run-time introspection, manipulation, and invocation of properties and methods in the object;

- serves as the basis for Qt's event system, which is a low-level way of communicating between QObject-based objects;

- signals and slots communication mechanism;

- the QTimer class provides a high-level interface for timers.

Signals and slots

http://doc.qt.io/qt-5/signalsandslots.html

It's probably the part that differs most from the features provided by other frameworks.

In GUI programming, we want objects to communicate with one another. For example, if a user clicks a Close button, we probably want the window's *close()* function to be called.

- Unlike other toolkits, in Qt, a *signal* is emitted when a particular event occurs. A *slot* is a function that is called in response to a particular *signal;*

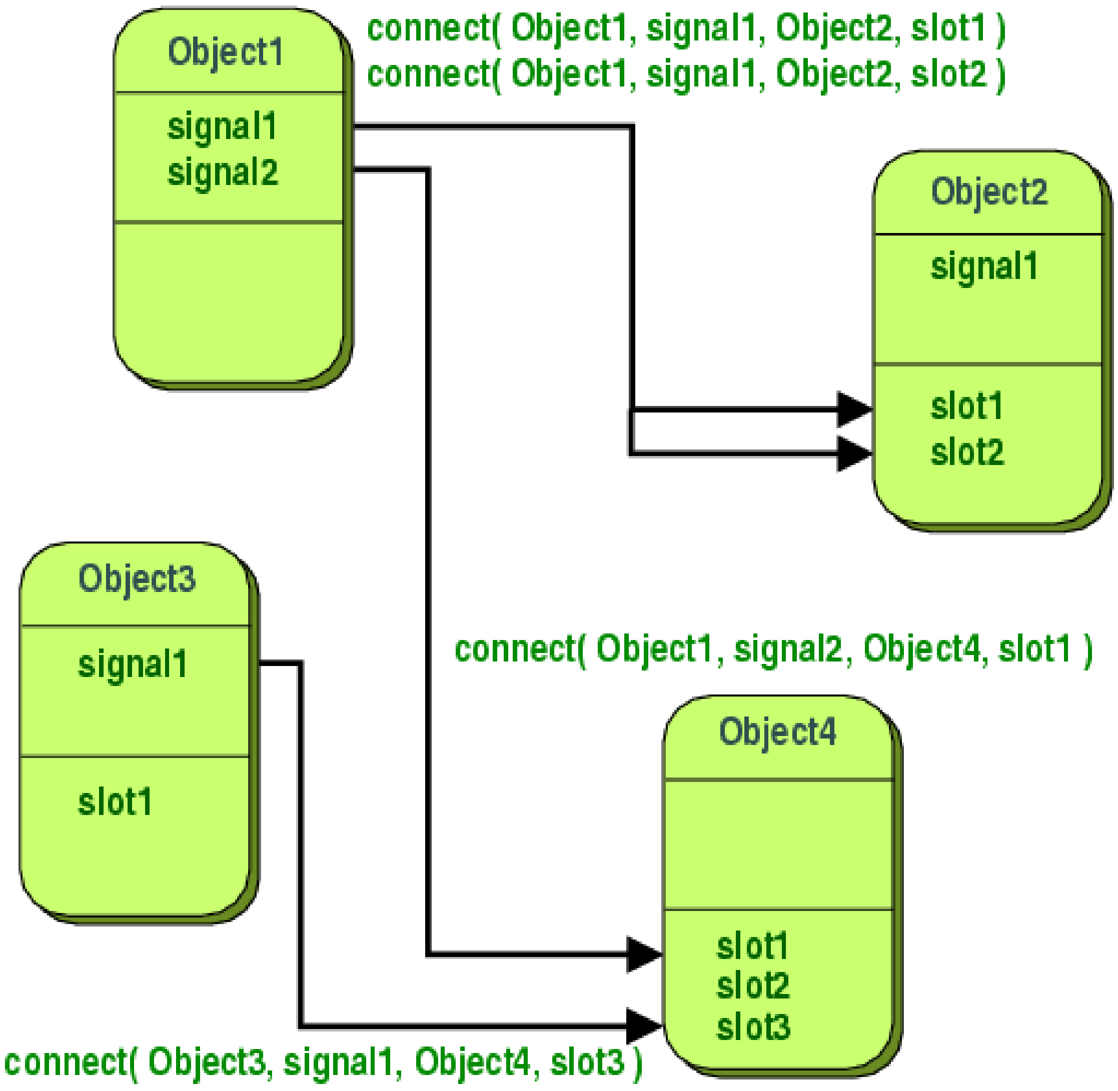
- Qt's widgets have many predefined signals. We can *subclass widgets to add our own signals and slots*;

- the signature of a signal must match that of the receiving slot (type safety).

- All classes that inherit from *QObject* or one of its subclasses (e.g., *QWidget*) can contain signals and slots.

Slots can be used for receiving signals, but they are also normal member functions. Objects are unaware of possible connections between each other.

Signals and slots (II)


Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Signals and slots (III)

- If several slots are connected to one signal, the slots will be executed one after the other, in the order they have been connected, when the signal is emitted.
- Signals do not have to be implemented (they are implemented by the *moc*, meta object compiler);
- signals do not have return types (return void).
- Slots are normal C++ functions and can be called normally;
- Slots can be *virtual.*
- All classes that contain signals or slots must mention Q_OBJECT at the top of their declaration.
- All classes that contain signals or slots must derive from QObject.
- QObject::sender() returns a pointer to the object that sent the signal.
- Compared to callbacks, signals/slots are slightly slower.

Elettra
Sincrotrone
Trieste

Signals and slots (IV)

Normal connections

- When a signal is emitted, the slots connected to it are usually executed immediately (independent of any GUI event loop);

- execution of the code following the emit statement will occur once all slots have returned.

Queued connections

→ **connect QObjects**
living in different threads!

- The code following the emit keyword will continue immediately, and the slots will be executed later

- The slot is invoked when control returns to the event loop of the receiver's thread

- The slot is executed in the receiver's thread.

- The parameters must be of types that are known to Qt's meta-object system (there's an event behind the scenes!)

Call *qRegisterMetaType()* to register the data type before connection

Properties

http://doc.qt.io/qt-5/properties.html#qt-s-property-system

A property behaves like a class data member, but it has additional features accessible through the Meta-Object System.

- Useful to export members to the Qt designer tool.

- The property type can be any type supported by *QVariant*, or it can be user-defined.

Event System

http://doc.qt.io/qt-5/eventsandfilters.html

Events are objects (not *QObjects*), derived from the abstract *QEvent* class.

- Represent things that have happened within an application;
- represent a result of outside activity that the application needs to know about.
- Events can be received and handled by any instance of a QObject subclass;
- especially relevant to widgets.
- Most events types have special classes, notably QResizeEvent, QPaintEvent, QMouseEvent;

To create events of a custom type, you need to define an event number, which must be greater than QEvent::User, and you may need to subclass QEvent in order to pass specific information about your custom event.

Event System (handlers, II)

The normal way for an event to be delivered is by calling a virtual function.

- In that function you must perform the necessary actions to react to the event;

- you may need to call the base class's implementation, for example to obtain the default behavior for any cases you do not want to handle (e.g. you handle left click only).

- Intercept events that are delivered to another object: *event filters* (*http://doc.qt.io/qt-5/qobject.html#eventFilter*).

Event System (sending, III)

You can send events by constructing suitable event objects and sending them;

- *QCoreApplication::sendEvent()* → processes the event immediately

- *QCoreApplication::postEvent()* → posts the event on a queue for later dispatch. The next time Qt's main event loop runs, it dispatches all posted events, with some optimization

postEvent **si very important for thread safe communication between QObjects**

Qt: see also...

- *QThread* http://doc.qt.io/qt-5/qthread.html
- *QWidget http://doc.qt.io/qt-5/qwidget.html*
- *QObject http://doc.qt.io/qt-5/qobject.html*
- *http://doc.qt.io/qt-5/groups.html* (classes grouped by functionality)

Part I

QtangoCore architecture overview

Overview (I)

- **Fast and easy development of graphical widgets integrated with the Tango control system;**
- **Integrated *Tango Exception* management and logging;**
- **Multi threaded environment for the creation of efficient and fully responsive graphical user interfaces:**
  - ✗ *Fulfils **Human Computer Interaction** Principles for GUI design;*
    - ✗ *Threads are grouped by device to optimize their number*

Overview (II)

- Reconnection to the device at startup;
- *asynchronous* execution of *targets* (write attributes, commands) (i.e. In the *DeviceThread*);
- get **attribute properties** at configuration time and get them asynchronously;
- get **device** and **class** properties through the *PropertyReader* utility class (blocking or asynchronous);
- monitor quantities and create custom widgets with *QTWatcher* and write or create writers with *QTWriter* utility classes;

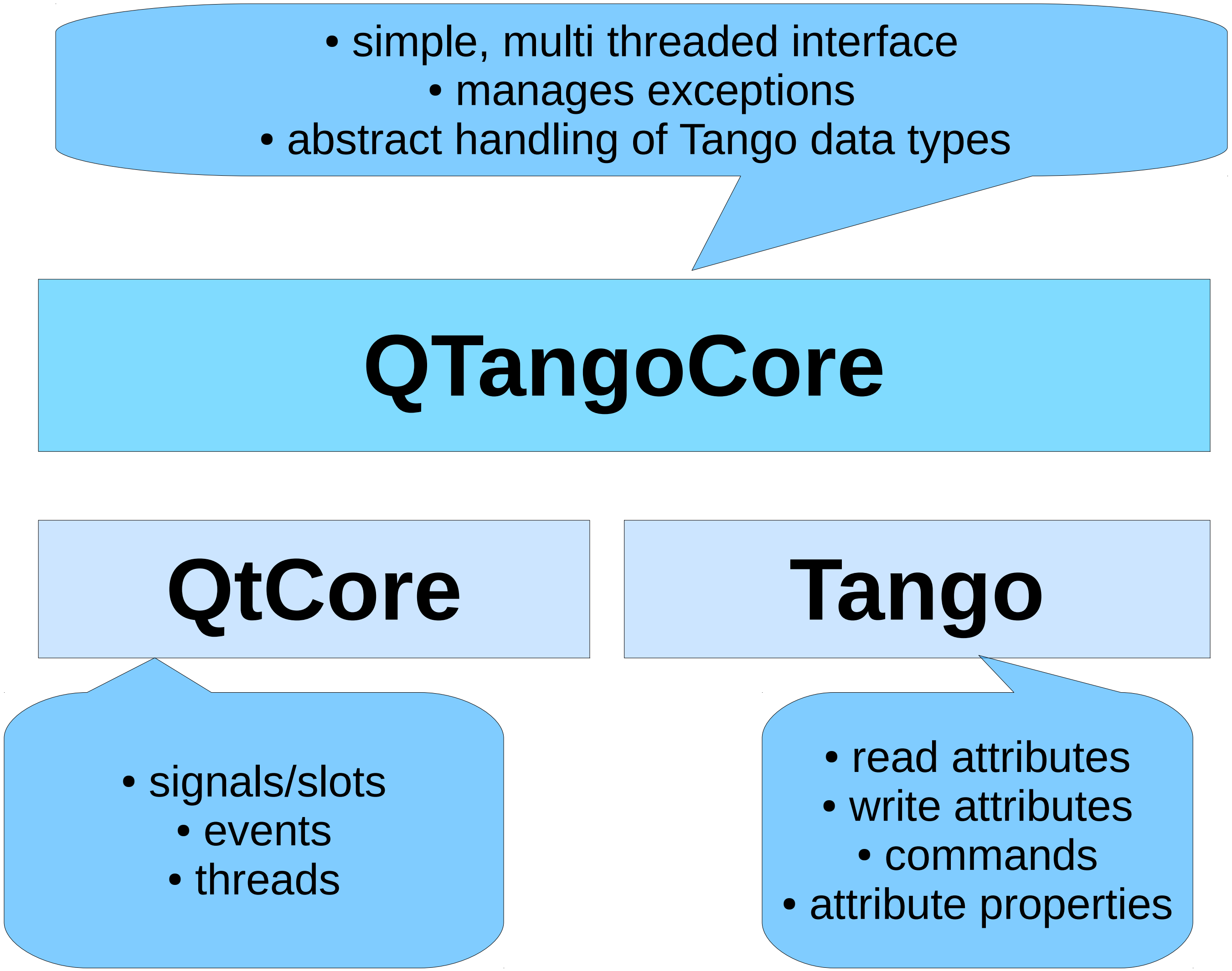
Overview (III)

Connection setup: try with events, fallback on polling (*AUTO_REFRESH*), unless otherwise specified:

- *ActionFactory::actionFactory() → setDefaultRefreshMode*
- *export DEFAULT_REFRESH_MODE=POLLED_REFRESH*
- The refresh mode can also be specified per widget (designer)

Polling is stopped when widget is not visible

Overview (III)



- simple, multi threaded interface
- manages exceptions
- abstract handling of Tango data types

QTangoCore

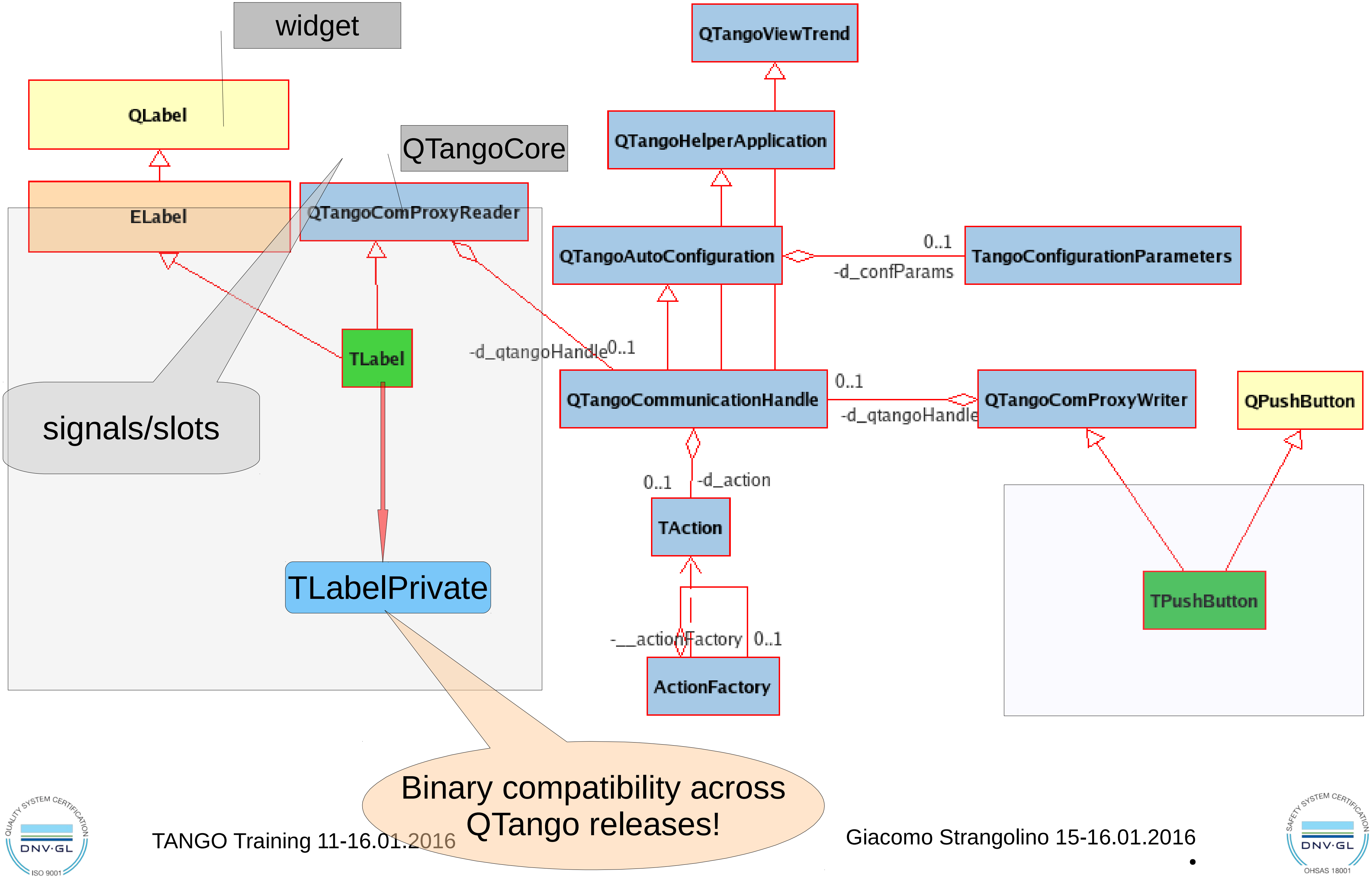
QtCore

Tango

- signals/slots
- events
- threads

- read attributes
- write attributes
- commands
- attribute properties

Class diagram with two client widgets

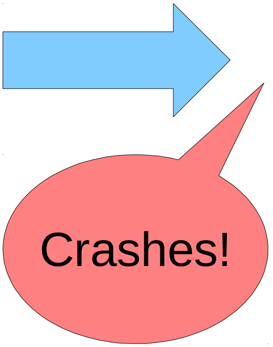


Binary compatibility across QTango releases!

Library implementation and binary compatibility

- When designing libraries, it is desirable that applications that dynamically link to them continue to run without recompiling even after the library is upgraded/replaced with another version

```
class Widget {
 ...
  private:
   Rect m_geometry;
};

class Label : public Widget {
 public:
  ...
  String text() const { return m_text; }

 private:
  String m_text;
};
```



Crashes!

```
class Widget {
 ...
 private:
  Rect m_geometry;
  String m_stylesheet; // NEW in WidgetLib 1.1
};

class Label : public Widget {
 public:

  String text() const { return m_text; }

 private:
  String m_text;
 };
```

Library implementation and binary compatibility (II)

- By adding a new data member, we ended up changing the size of Widget and Label objects;
- when the C++ compiler generates code, it uses offsets to access data within an object.

Library implementation and binary compatibility (III)

The solution

- keep the size of all public classes of a library constant by only storing a single pointer.
- It points to a private/internal data structure that contains all the data.
- The size of this internal structure can shrink or grow without having any side-effect;
- the pointer is accessed only in the library code and from the application's point of view the size of the object never changes - it's always the size of the pointer.
- Hides implementation details.
- The header file contains only the API reference;

Binary Compatibility Issues With C++

You can:

- add new non-virtual functions including signals and slots and constructors.
- add a new enum to a class.
- append new enumerators to an existing enum. (it's recommended to add a Max.... enumerator with an explicit large value: compiler may choose a larger underlying type for the enum)
- remove private non-virtual functions if they are not called by any inline functions (and have never been).
- remove private static members if they are not called by any inline functions (and have never been).
- add new static data members.

Binary Compatibility Issues With C++

You can:

- change the default arguments of a method. It requires recompilation to use the actual new default argument values.
- add new classes.
- add or remove friend declarations to classes.
- rename reserved member types
- extend reserved bit fields, provided this doesn't cause the bit field to cross the boundary of its underlying type (8 bits for char & bool, 16 bits for short, 32 bits for int, etc.)
- add the Q_OBJECT macro to a class <u>if the class already inherits from QObject</u>
- add a Q_PROPERTY, Q_ENUMS or Q_FLAGS macro as that only modifies the meta-object generated by moc and not the class itself

Binary Compatibility Issues With C++

You cannot:

- unexport or remove an exported class.
- change the class hierachy in any way (add, remove, or reorder base classes).
- Remove functions, inline them;
- Change function signature (types, *const, volatile* qualifiers, change *access rights, return* type.
- add a **virtual** function *to a class that doesn't have any virtual functions* or virtual bases.
- add new virtual functions to non-leaf classes as this will break subclasses (a class designed to be subclassed by applications is always a non-leaf class).
- change the order of virtual functions in the class declaration.
- Remove a virtual function

Binary Compatibility Issues With C++

You cannot:

- add new data members to an existing class.
- change the order of non-static data members in a class.
- change the type of the member, except for signedness
- remove existing non-static data members from an existing class.

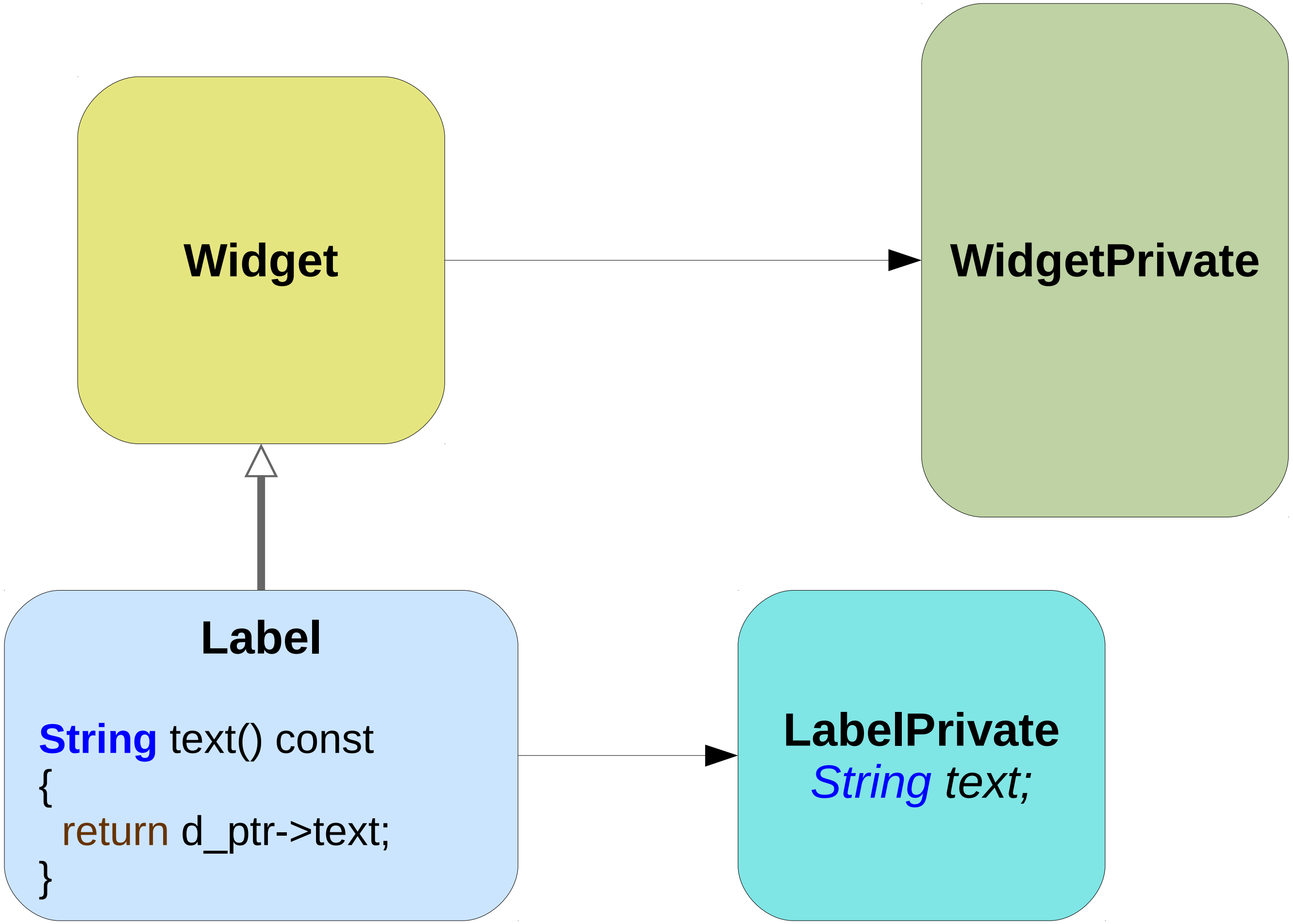
Binary Compatibility Issues With C++

To make a class to extend in the future

- add d-pointer, as discussed above;
- add non-inline virtual destructor even if the body is empty.
- reimplement *event* in QObject-derived classes, even if the body for the function is just calling the base class' implementation.
- make all constructors non-inline.
- write non-inline implementations of the copy constructor and assignment operator unless the class cannot be copied by value (e.g. classes inherited from QObject can't be)

Library implementation and binary compatibility (III)

D pointer

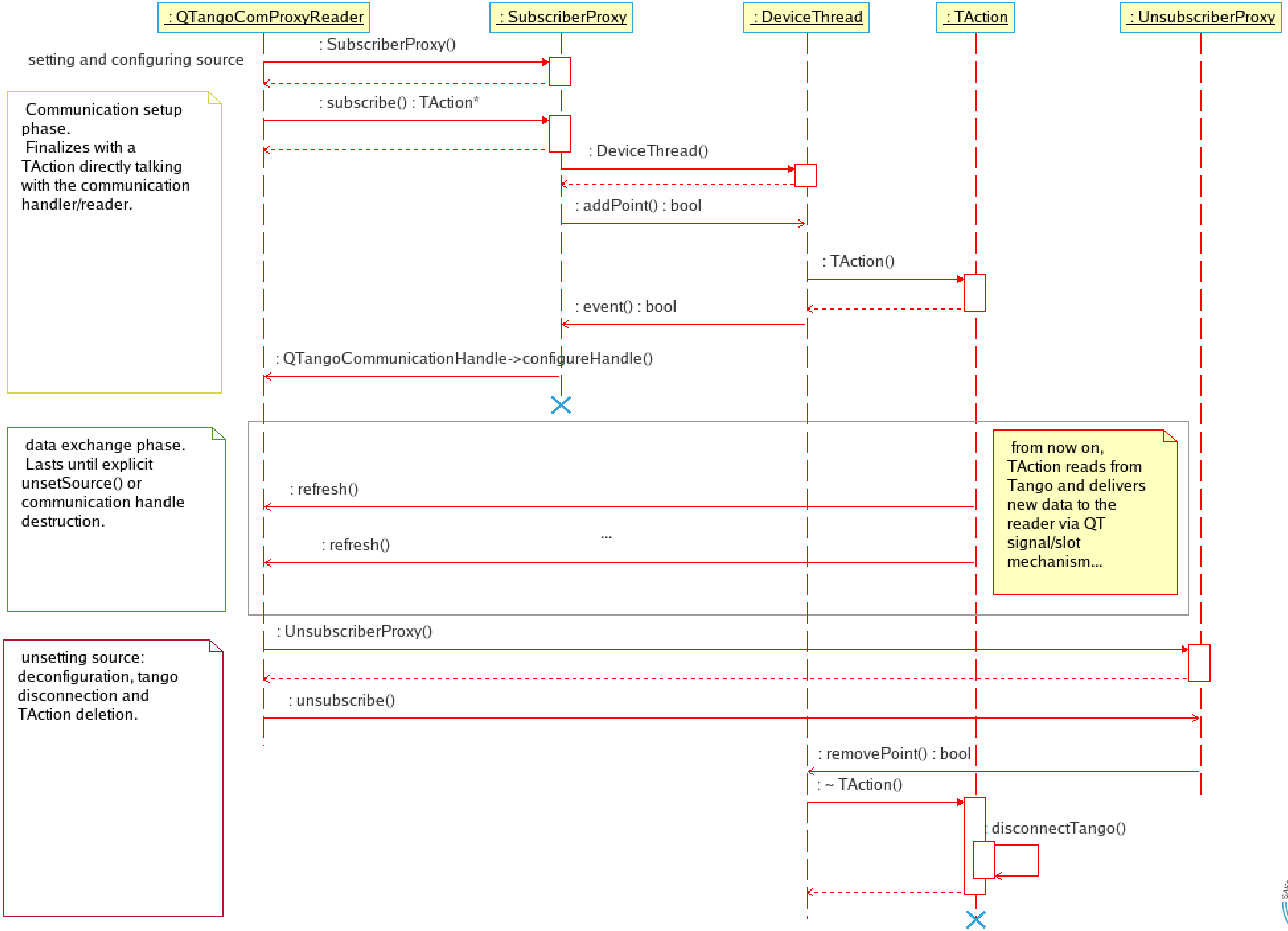


```
String text() const
{
  return d_ptr->text;
}
```

String text;

QTangoCore objects lifetime sequence diagram

Elettra
Sincrotrone
Trieste



Communication setup phase.
Finalizes with a TAction directly talking with the communication handler/reader.

data exchange phase.
Lasts until explicit unsetSource() or communication handle destruction.

from now on, TAction reads from Tango and delivers new data to the reader via QT signal/slot mechanism...

unsetting source: deconfiguration, tango disconnection and TAction deletion.

QTangoCore implementation

- One thread per device;
- *TActions* shared among readers with the same source;
- *TActions* live outside the main application thread;

Elettra
Sincrotrone
Trieste

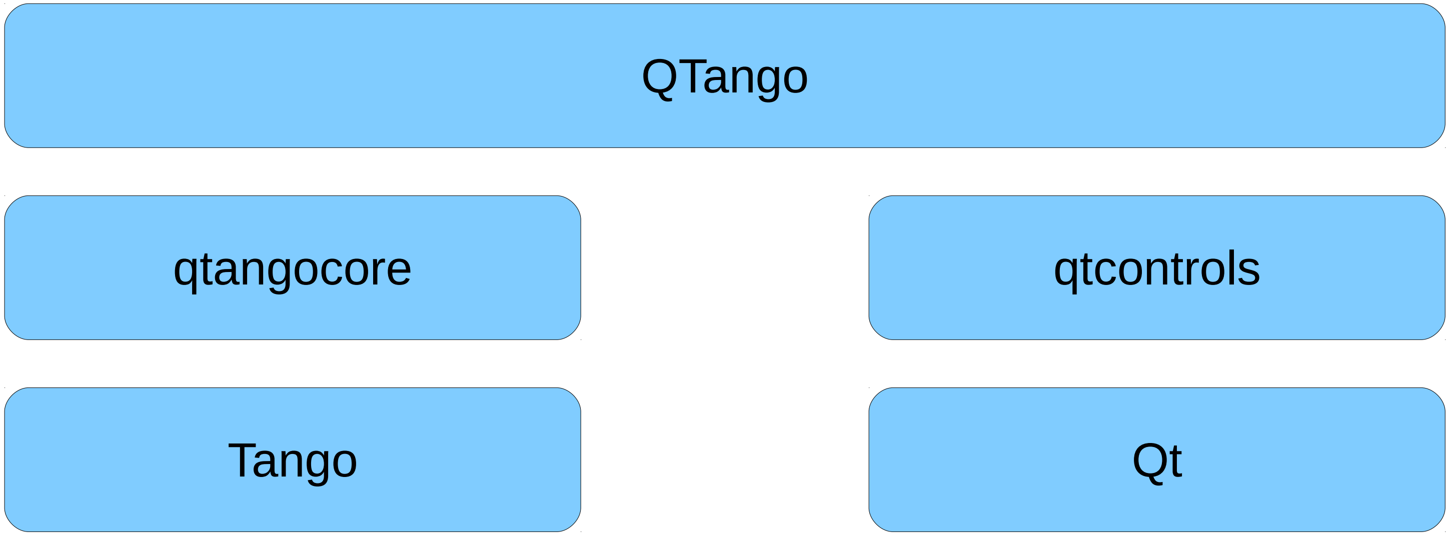
Part II

QTango

a set of Qt widgets integrated

with QTangoCore

QTango infrastructure



Overview

Right click on a widget:

- view trend of scalar attribute values (plot);
- show tango point information (connection status, time stamp, data type, refresh mode, polling period, and so on…);
- helper application (defined as an attribute or device property or in a widget property);
- copy source into clipboard.

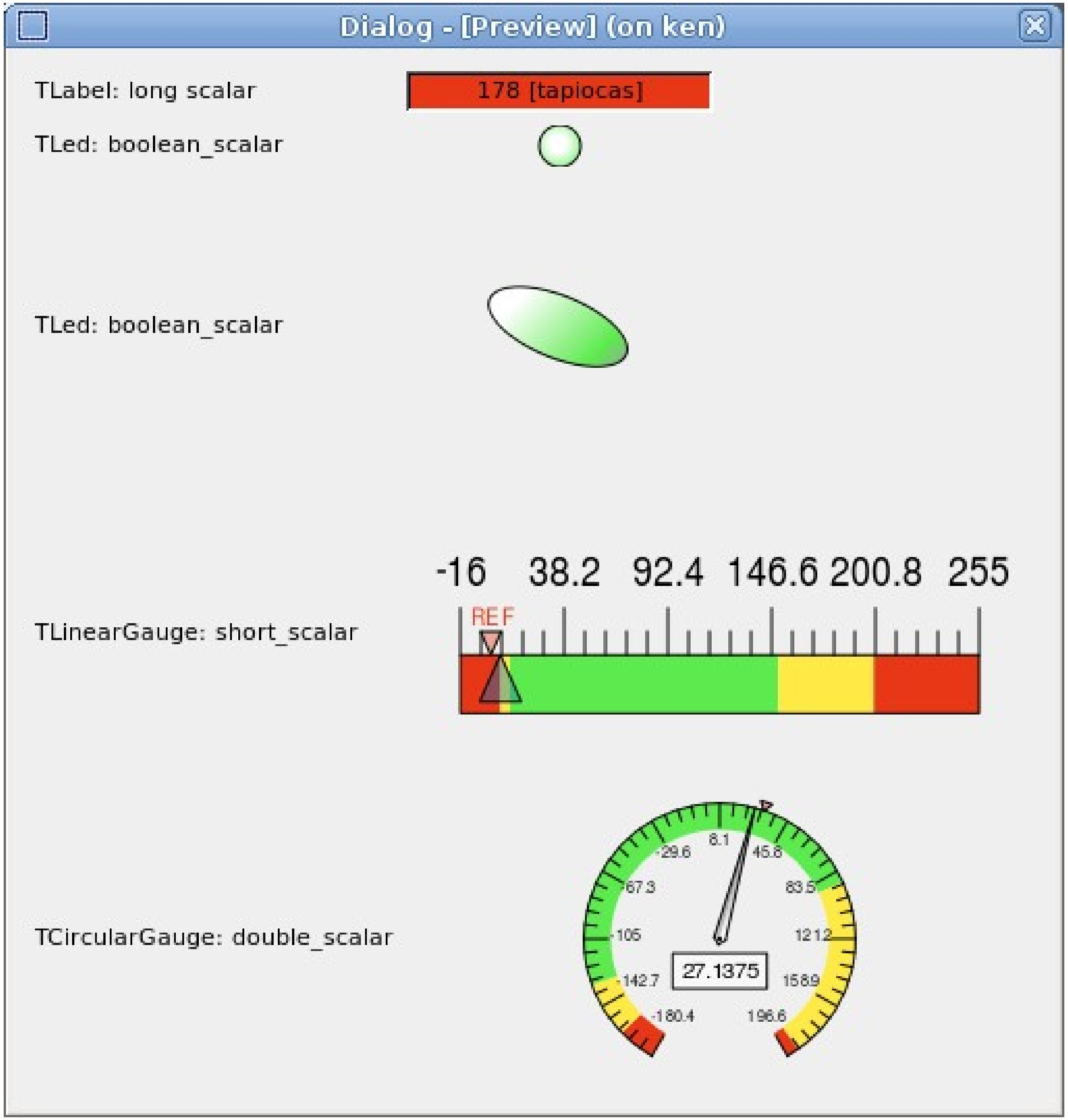
- Stop reading while hidden;

Optimization

- Widget refresh is triggered by an external clock:
  - all widget refreshed at once

- global refresh trigger can be disabled:
  - ✗ globally;
  - ✗ *per* reader
  - ✗ *little cpu overhead if many widgets refreshing independently*

Readers



Dialog - [Preview] (on ken)

TLabel: long scalar — 178 [tapiocas]

TLed: boolean_scalar

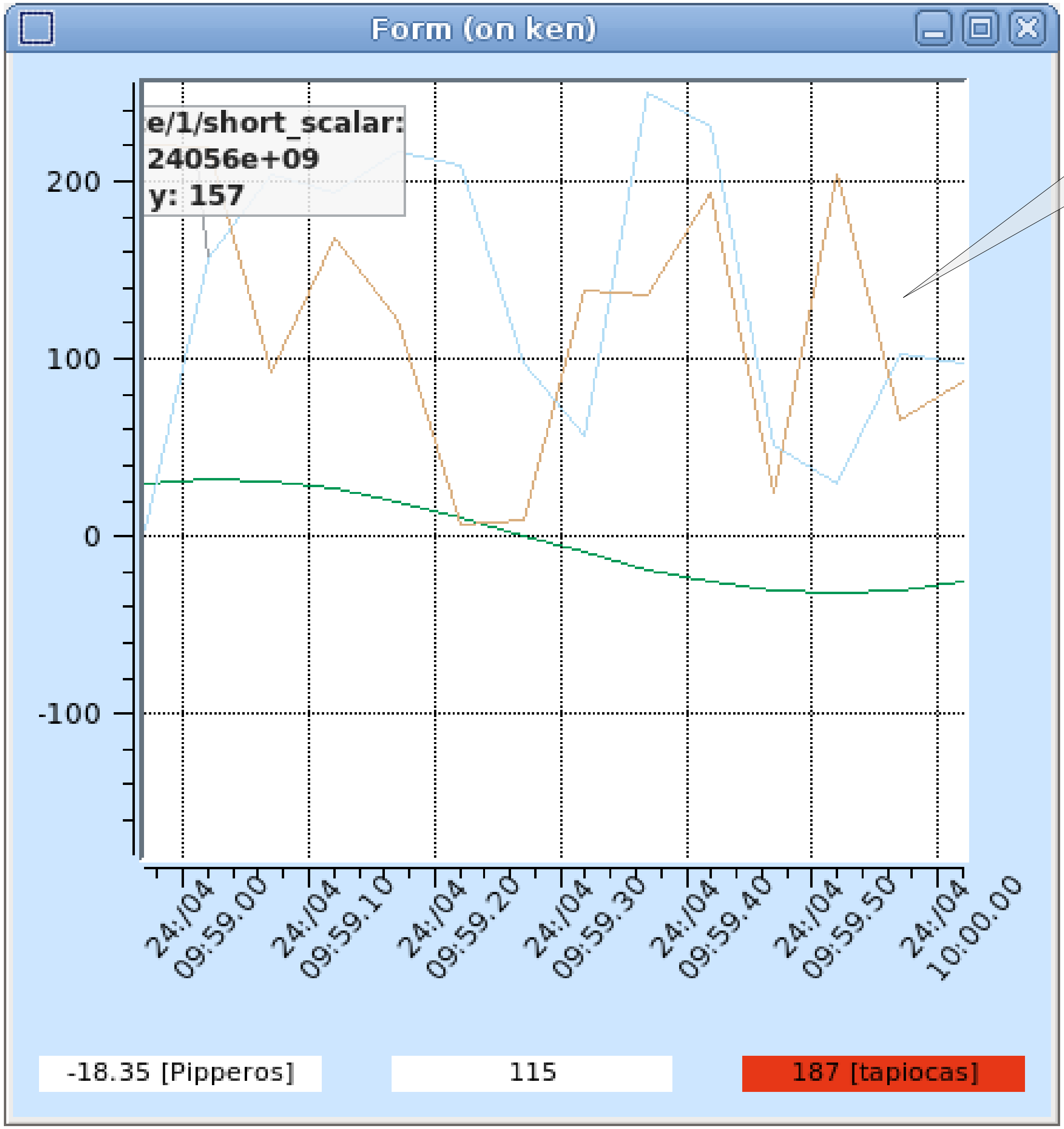
TLed: boolean_scalar

TLinearGauge: short_scalar

TCircularGauge: double_scalar — 27.1375

Readers (II)

TPLotLight



-18.35 [Pipperos]

115

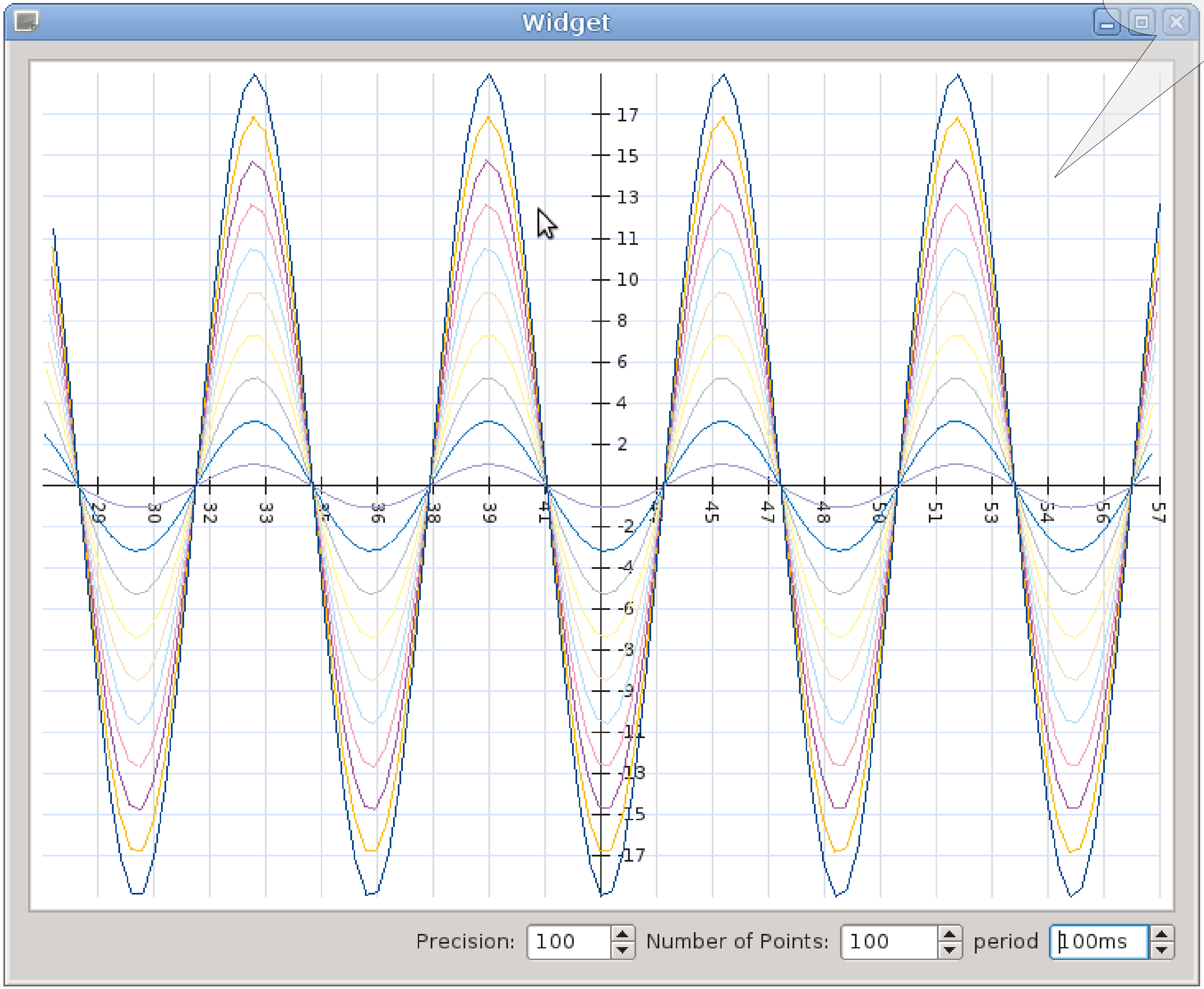
187 [tapiocas]

See also: *TRealtimePlot*, tailored for real time Tango commands;

Elettra
Sincrotrone
Trieste

Readers (III)

TGraphicsPlot



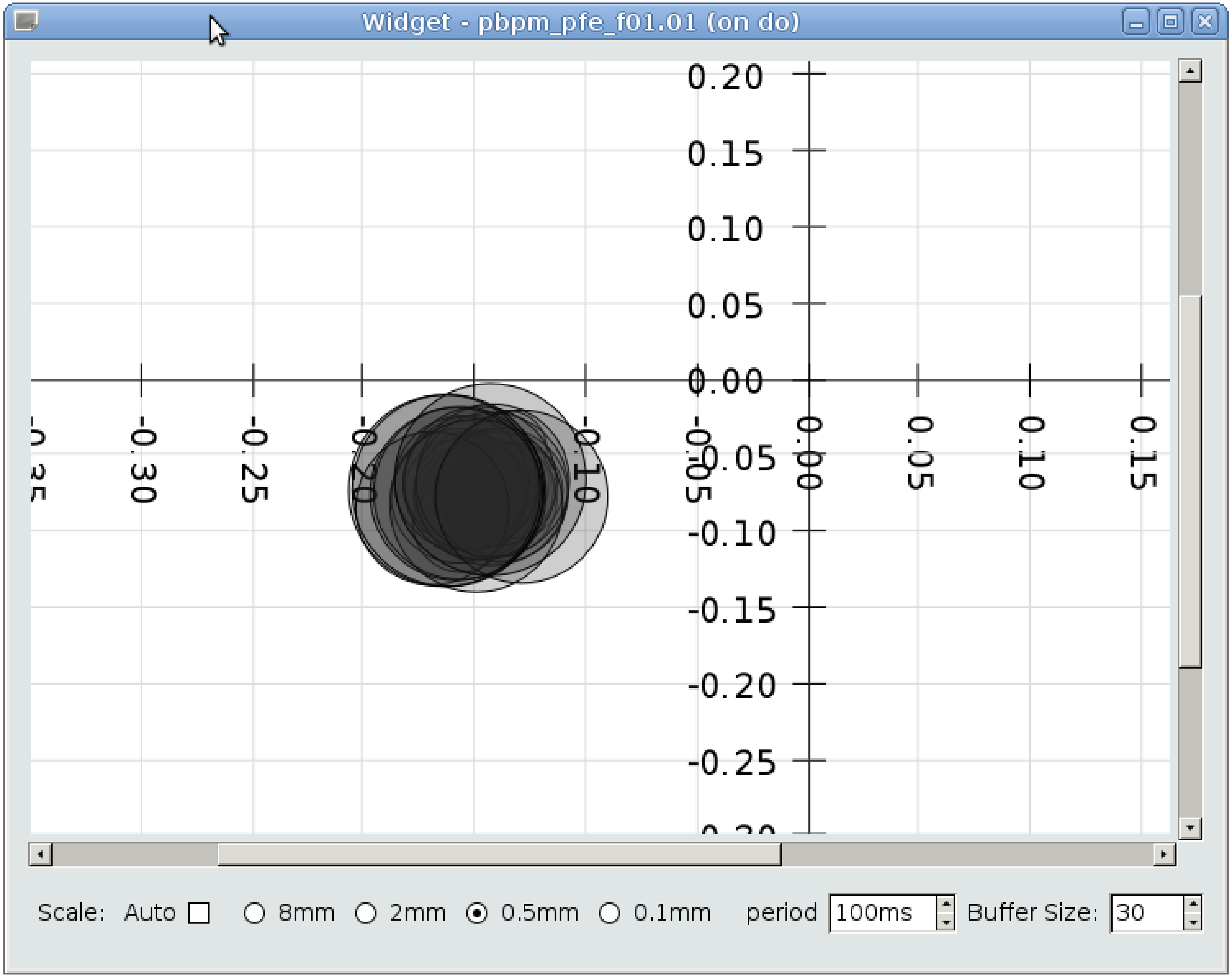
Precision: 100 Number of Points: 100 period 100ms

Uses Qt *QGraphicsView/QGraphicsScene* technology

Readers (IV): QGraphicsPlot

- Uses **QGraphicsView/QGraphicsScene** to draw curves on a plot canvas;
- *curves* are **QGraphicsItem**s, the view can be scaled, zoomed;
- there are two default axes items, but *external scales* can be configured and attached to the plot;
- first raw performance tests show off performances comparable (or even slightly better) to *QwtPlot*'s.

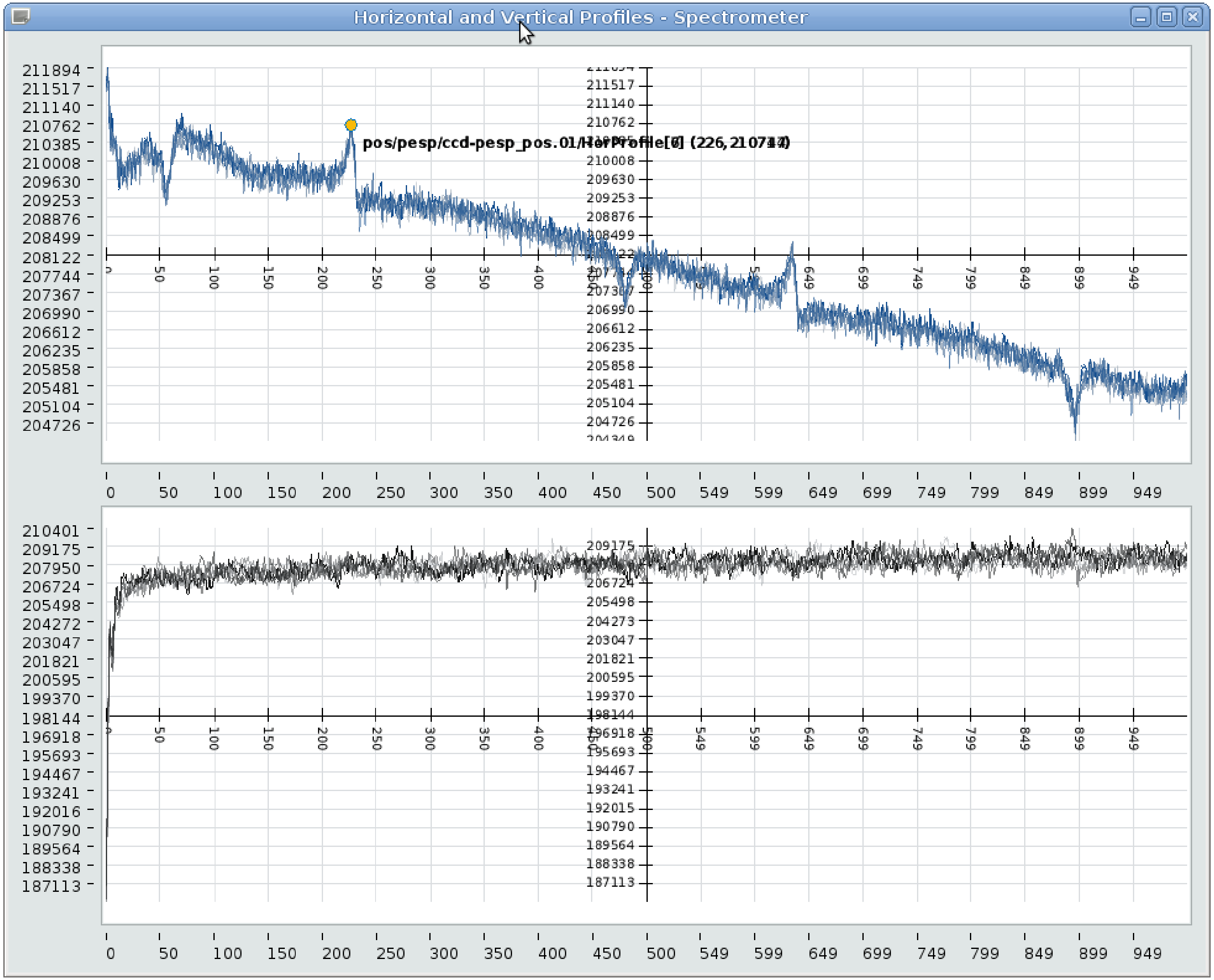
Readers (IV): QGraphicsPlot (II)



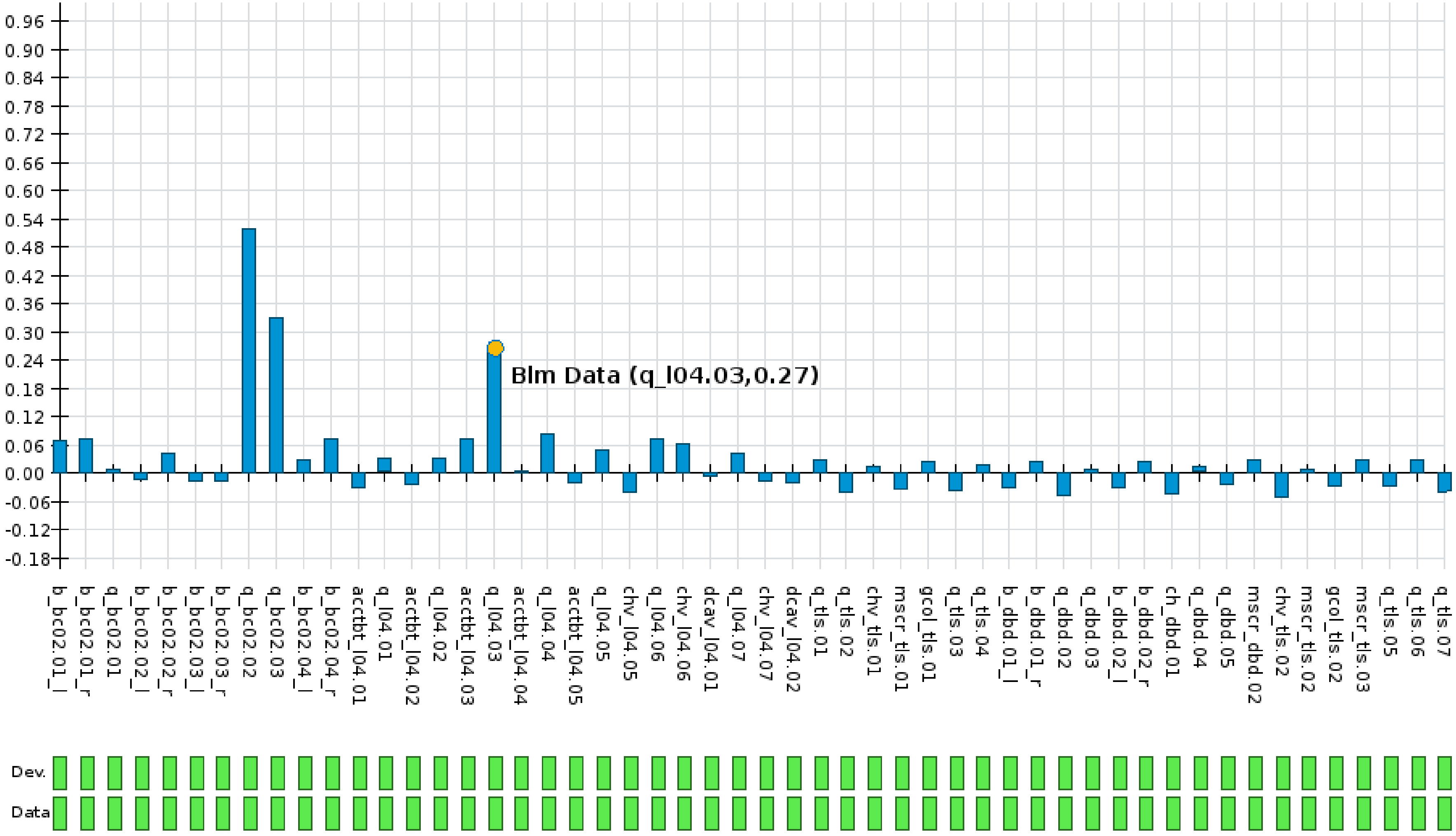
Scale: Auto ☐ ○ 8mm ○ 2mm ● 0.5mm ○ 0.1mm period 100ms Buffer Size: 30

Elettra
Sincrotrone
Trieste

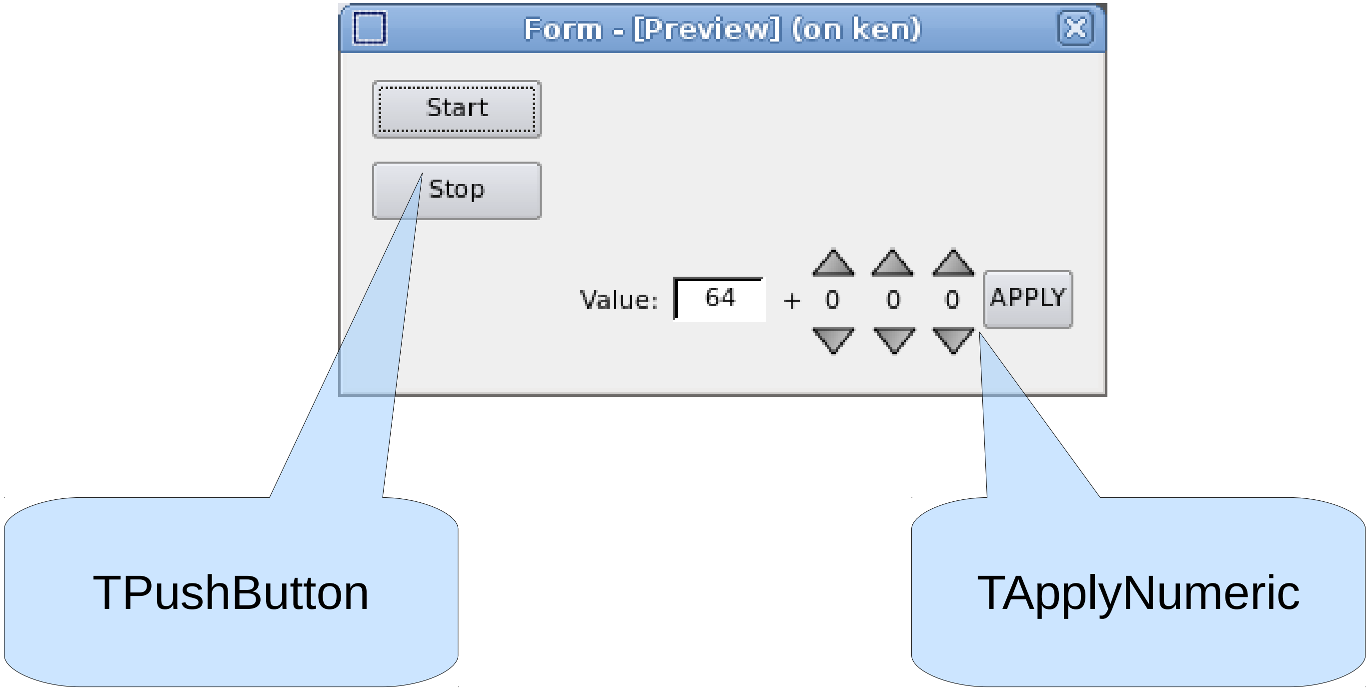
Readers (IV): QGraphicsPlot (III)



Readers (IV): QGraphicsPlot (IV)



Writers



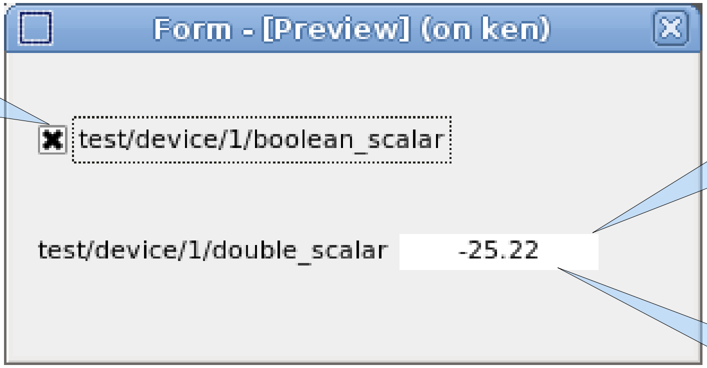
Value: 64 + 0 0 0

TPushButton

TApplyNumeric

Readers *and* Writers

TCheckBox



Form - [Preview] (on ken)

☒ test/device/1/boolean_scalar

test/device/1/double_scalar -25.22

TreaderWriter

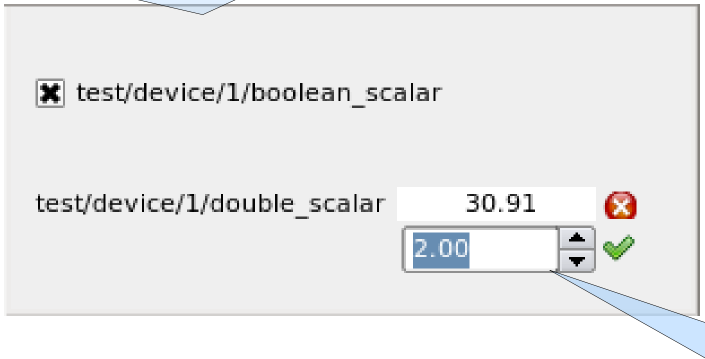
- reads a value...

TreaderWriter

- move the mouse over...

TreaderWriter

- ideal for synoptics
- occupies the space of a label with a hidden writer



☒ test/device/1/boolean_scalar

test/device/1/double_scalar 30.91

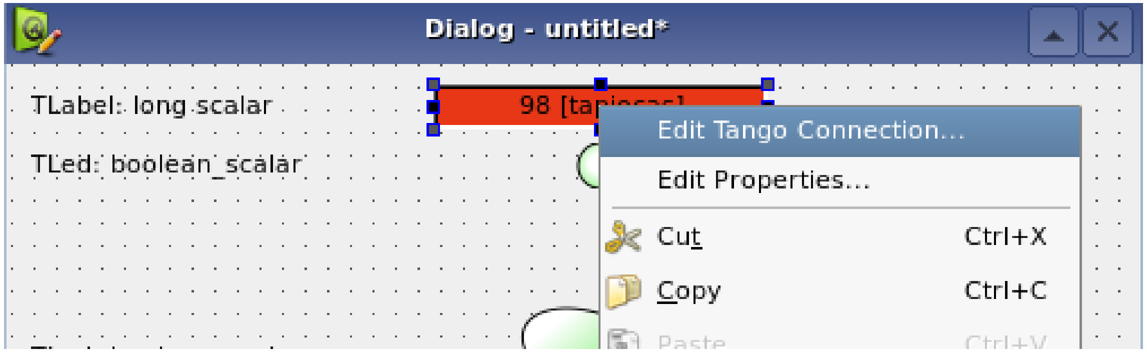
2.00

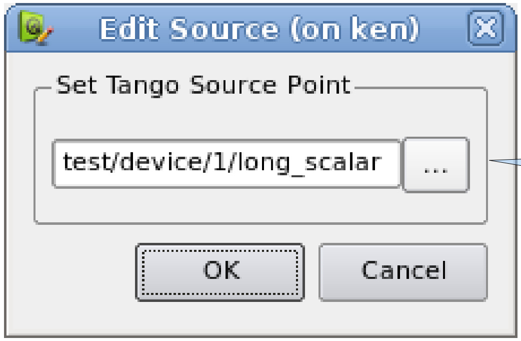
TreaderWriter

- a writer appears

Qt Designer integration

Easy configuration of tango ***source*** (for readers) and ***target*** (for writers)





test/device/1/long_scalar

Edit Source dialog

- test/device/instance/attribute_name
- test/device/instance->command_name(argin)

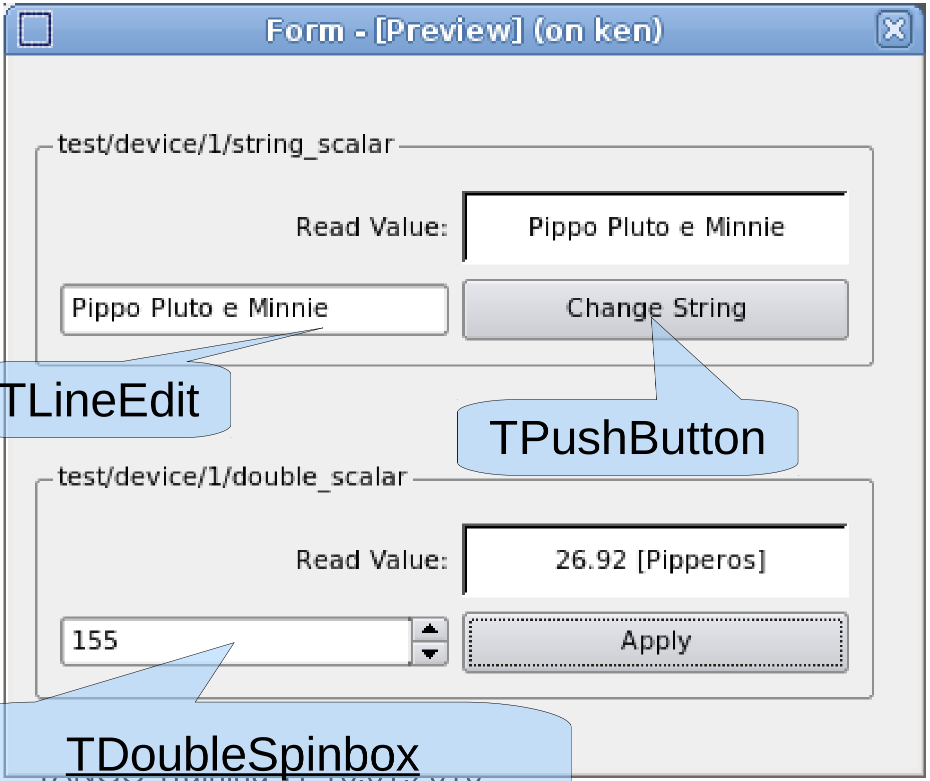
Drag and drop from Jive!

SimpleDataProxy for writers

SimpleDataProxy elements *display* data that can be used as *input arguments* for commands or attributes on *writers*

Edit Targets Dialog

- test/device/1/double_scalar(&simpleDataProxyObjectName)



Form - [Preview] (on ken)

test/device/1/string_scalar

Read Value: Pippo Pluto e Minnie

Pippo Pluto e Minnie

test/device/1/double_scalar

Read Value: 26.92 [Pipperos]

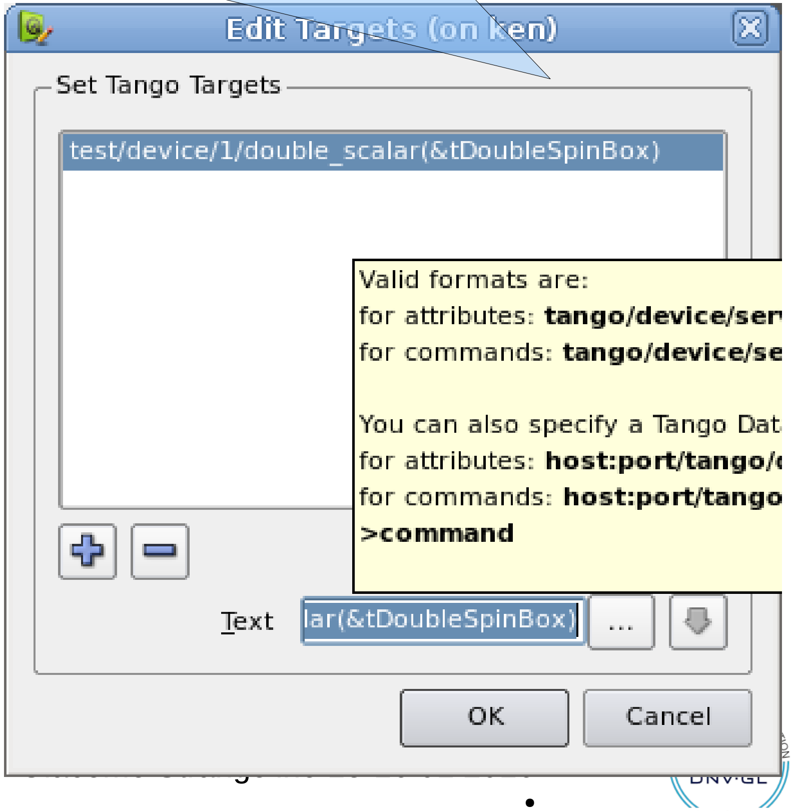
155

TLineEdit

TPushButton

TDoubleSpinbox

- *with name “tDoubleSpinBox”*



Edit Targets (on ken)

Set Tango Targets

test/device/1/double_scalar(&tDoubleSpinBox)

Valid formats are:
for attributes: **tango/device/ser**
for commands: **tango/device/se**

You can also specify a Tango Dat
for attributes: **host:port/tango/**
for commands: **host:port/tango**
>command

Text lar(&tDoubleSpinBox)

Reading and writing *Spectrum* attributes

TSpectrumButton: writes into a *spectrum* attribute fetching data from *SimpleDataProxy* widgets (*TLineEdit*, *TNumeric*, user defined...);

TwidgetGroup: groups a set of readers and refreshes them with the values extracted from a *spectrum* attribute;

→ Full *Qt designer* integration and configuration!

Elettra
Sincrotrone
Trieste

Part III

Programming with

QtangoCore

Create an object (*QWidget* or *QObject*) reading from and writing to a *Tango* device server

Includes

.pro project file:

```
include(/usr/local/qtango/include/qtango6/qtango.pri)
```

QTangoCore stuff in .h files

```
#include <com_proxy_reader.h>
#include <com_proxy_writer.h> /* for writers */
#include <tvariant.h>
```

Connection

Readers

setSource

test/device/1/double_scalar

Attributes

test/device/1->DevDouble

Commands

Writers

setTargets

unsetSource, clearTargets, setPeriod, setRefreshMode

Reader

- Readers <u>must</u> inherit from *QTangoComProxyReader*
- readers <u>must</u> implement the *pure virtual* method *refresh*()
- the *refresh*() method has a *<u>TVariant</u>* as argument. It contains the data read from the *Tango* layer.
- *connect() reader's qTangoCommunicationHandle newData() signal to the refresh() slot*
- (Optional) inherit from *QTangoWidgetCommon* in order to obtain a common behavior among QTango widgets (copy source, view trend, helper application). No methods shall be reimplemented.

auto configuration

- The Tango attribute must be configured into the database with its *minimum and maximum values* (also warning and alarm thresholds, if desired);
- Must connect the reader's *handle signal attributeAutoConfigured(const TangoConfigurationParameters *)* to your configuration *slot;*
- If *Tango events* are available, you may receive *attribute configuration events* via the connected *slot*

To disable attribute configuration change events:

export QTANGO_ATTR_CONF_CHANGE_DISABLED=1

Connection: configuration

QObject!

```
connect(myReader->qtangoComHandle(),

SIGNAL(attributeAutoConfigured(const
TangoConfigurationParameters *)),

        this,

SLOT(readerConfigured(const
TangoConfigurationParameters *)));
```

QObject!

Connection: configuration (II)

TangoConfigurationParameters

- *double maxValue() const { return mxValue; }*
- *double minValue() const { return mValue; }*
- *double maxWarning() const { return mxWarning; }*
- *double maxError() const { return mxError; } […]*
- *bool maxIsSet() const { return d_maxIsSet; }*
- *bool minIsSet() const { return d_minIsSet; }*
- *bool MErrIsSet() const { return d_MErrIsSet; }*
- *bool mWarnIsSet() const { return d_mWarnIsSet; } […]*
- *QString description() const { return d_desc; }*
- *QString label() const { return d_label; }*
- *QString stdUnit() const { return d_stdUnit; }*
- *QString displayUnit() const { return d_displayUnit; }*
- *QString format() const { return d_format; }*
- *TVariant currentValue()*

First available read value

Connection: new data signal

```
 connect(myReader->qtangoComHandle(),
SIGNAL(newData(const TVariant&)),
   this,
SLOT(refresh(const TVariant&) ));
```

QObject!

QTangoCore variant data type

Inside refresh(), extract the data

refresh() (in a reader)

- Using TVariant, test the attribute quality;
- see if *canConvert()* to the required type;
- if yes, convert it into the desired type
- do whatever you like with the extracted data

TVariant

Can convert to a certain data type?

- *bool canConvertToState() const;*
- *bool canConvertToString() const;*
- *bool canConvertToInt() const;*
- *bool canConvertToUInt() const;*
- *bool canConvertToDouble() const;*
- *bool canConvertToBool() const;*
- *bool canConvertToStringVector() const;*
- *bool canConvertToIntVector() const;*
- *bool canConvertToDoubleVector() const;*
- *bool canConvertToBoolVector() const;*

TVariant (II)

Yes, can convert

DevState	*toState() const;*
QString	*toString(bool = true) const;*
int	*toInt(bool = true) const;*
unsigned int	*toUInt(bool = true) const;*
double	*toDouble(bool = true) const;*
bool	*toBool(bool = true) const;*
QVector<QString>	*toStringVector(bool = true) const;*
QVector<int>	*toIntVector(bool = true) const;*
QVector<unsigned int>	*toUIntVector(bool = true) const;*
QVector<double>	*toDoubleVector(bool = true) const;*
QVector<bool>	*toBoolVector(bool = true) const;*

...

TVariant (III)

Get Tango data structures

- *AttributeInfo getAttributeInfo () const*
- *CommandInfo getCommandInfo () const*
- *CmdArgType type() const*
- *AttrQuality quality() const*

Works for commands and Attributes!

And…

- *QString message() const*
- *const struct timeval *timeReadRef() const*
- *struct timeval timeRead() const*
- *QString tangoPoint() const*

QTangoWidgetCommon

Provides a common set of functionalities for QTango widgets

- View trend
- Helper application
- Copy source
- Connection state

```
#include <qtango_wcommon.h>
#include <com_proxy_reader.h>
#include <QLabel>

class ReadLabel : public QLabel,
        public QTangoComProxyReader,
        public QTangoWidgetCommon
{
}
```

No method shall be reimplemented from QTangoWidgetCommon

Example: reader implementation

The reader will be able to:

- *read an attribute*;
- *auto configure* itself to be aware of warning and alarm thresholds;
- associate a *helper application* to the connected *source*.

Exercise 1

Write a reader that displays a scalar value onto a *QLabel*. The graphical interface provides also a *TLineEdit* input text to let the user input a source for the readings, which can be an attribute (e.g. test/device/1/double_scalar) or a command (test/device/1→DevDouble). A *QPushButton* with text “connect” activates the readings, a *QPushButton* with text “disconnect” stops the readings. A *led* next to the reading label is red when a read error occurs. A text area displays also a message associated to the readings.

If the reading is not available (*quality* is ATTR_INVALID), the label must display “####”.

Hints: use *qtcreator* to setup a new project, the Qt *designer* to create the user interface, a *QPlainTextEdit* for the connection message text area. For the led, derive from QLabel to make a rectangular led with a *setOk(bool ok) slot* that colors the label green when ok is true, red otherwise. The reader has the text centered and font is bold.

Compulsory: *Use layouts* when composing the interface.

Exercise 1

Testing the panel

Now test your application and notice if the led and the text message behave as expected. Try with attributes of different types and commands.

- Try to start the application when the TangoTest device is not running.
- Take the TangoTest device server down while the application is reading from it.
- Try to type a wrong source.

Exercise 1b

- After testing exercise 1, modify the reading label and let it implement QTangoWidgetCommon*.* Then see what changes…
- Modify the reading label so that if the value is inside the *warning* range the text is **yellow**, and if it is inside the *alarm* range it turns **red**. Put the measurement unit beside the value within square brackets. Supposing that the data type is a number, format it according to the format stored into the Tango database.
- Use *TApplication* instead of *QApplication* in main.cpp
- Design the new Label so that future changes to its functionalities does not affect binary compatibility of the library where it may be included.

Writer

inherits *QTangoComProxyWriter*

- auto configuration available (see considerations done for the reader)
- write is performed inside *QTangoComProxyWriter*'s *execute()* implementation

Simple Data Proxy

- provides **input data** for your **writers**;
- any QWidget displaying something can be used to implement a simple data proxy:
  - QLabel
  - Q[Double]SpinBox
  - QTextEdit/QTextBrowser
  - QComboBox
  - QLineEdit
  - …

just implement the virtual slot getData, returning a string representation of the data displayed by the widget

Simple Data Proxy (II)

- inherit from *SimpleDataProxy*;
- implement the pure *virtual QString getData()* method
- example: *QTango TLineEdit*

Exercise 2

Write a QTango component which is a horizontal *QSlider* and writes an attribute. The slider configures its minimum and maximum values from the Tango database and its position must be initialised according to the current *set point* value of the attribute itself.

- The actual writing is performed one second after the user stops moving the slider, in order to avoid continuous writings during the movement between the initial and final positions. In alternative, slider tracking property could be used (see *tracking* property in *QAbstractSlider*).
- Use the label written in exercise 1b to read the value of the same attribute.
- Let the attribute name be specified from the command line.

Exercise 2

- Let the slider emit a signal *valueChanged(double)* to notify that the value has changed in device attribute “coordinates”.
- Beside the slider put a double spin box displaying the value that is going to be written.
- Design the component so that it can be changed in the future without breaking the binary compatibility.

Exercise 2

NOTES

- It would be more appropriate to implement a writer based on the spin box, whose value is more precise.
- If writing fails, the slider (and the spin box) should be brought back to their previous values rather than display a value that was not actually written. Care must be taken when invoking setValue to prevent unwanted writings after a failure. When calling *execute()* inside the writer, check the returned *Qlist<TVariant>* and test each *TVariant* with the *bool executionFailed()* method.
- Writers can be executed asynchronously (in another thread) if *executeAsync(QVariant)* is called instead of *execute(QVariant)*. In this case, the return value cannot be obtained.

Exercise 2b

Modify exercise 2 so that the attribute is written not on slider move but after pressing a TPushButton

QTWatcher and QTWriter classes

QTWatcher

- Reads Tango variables using QTango;
- QObject or base types can be *attached()*;
- on new data, a SLOT can be invoked on the QObject;
- the *data type* is guessed from the QObject SLOT input parameter
- *auto configuration* possible if QObject has suitable slots (e.g. *QProgressBar setMinimum()*)
- On read error, *slots aren't invoked and variables aren't updated*!

QTWatcher with QObjects

```
QProgressBar *pbar = new QProgressBar(this);
QTWatcher *pbarWatcher = new QTWatcher(this);

pbarWatcher->attach(pbar, SLOT(setValue(int)));

// configure maximum and minimum values when available
pbarWatcher->setAutoConfSlot(QTWatcher::Min, SLOT(setMinimum(int)));
pbarWatcher->setAutoConfSlot(QTWatcher::Max, SLOT(setMaximum(int)));

pbarWatcher->setSource("$1/short_scalar_ro");
```

QTWatcher with simple data types

```
short int var;
QTWatcher *intWatcher = new QTWatcher(this);

pbarWatcher->attach(&var);

pbarWatcher->setSource("$1/short_scalar_ro");
```

- var is always up to date;
- tango reads are performed in another thread;
- it is safe to access var in any moment inside your thread.

QTWatcher: signals

- attributeAutoConfigured(const TangoConfigurationParameters *);
- connectionFailed();
- connectionOk(bool);
- connectionErrorMessage(const QString &);
- readOk(bool);
- newData(int), newData(double), … , newData(const QString&).

QTWatcher: filter the value

- Modify the value read before invoking your slot or using your variable (***TValueFilter*** class)

```
class PlotLevelFilter : public TValueFilter
{
public:
    PlotLevelFilter(short int *imgDepth) :
       TValueFilter(),
       imageDepth(imgDepth)
       {};

    void filter(const TVariant& variant, int &intValue,
       bool read, State updateState)
       {
              if (*imageDepth == 16)
                     intValue = round(intValue/16);
       }

       short int* imageDepth;
};
```

QTWatcher: filter the value (II)

- install the implementation of ***TValueFilter***

```
QSlider *color_Slider = new QSlider(this);
QTWatcher *plotLevelWatcher = new QTWatcher(this);
plotLevelWatcher->attach(color_Slider, SLOT(setValue(int)));

PlotLevelFilter *plotLevelFilter = new
PlotLevelFilter(&imageDepth);
plotLevelWatcher->installRefreshFilter(plotLevelFilter);

 plotLevelWatcher->setSource(“a/b/c/PlotLevel");
```

QTWriter

- Write an attribute or give a command from any QObject or Qwidget;
- a *signal* of the QObject is connected to a compatible *execute()* method implemented in QTWriter;
- a *set point slot* can be provided to initialize the object with the current value at auto configuration time;
- data type automatically detected from the *signal* specified!

QTWriter

```
QLineEdit *lineEdit = new QlineEdit(this);
QTWriter *lineEditWriter = new QTWriter(this);

lineEditWriter->attach(lineEdit,
          SIGNAL(textChanged(const QString&)),
          SLOT(setText(const Qstring&)));

lineEditWriter->setTargets(“test/device/1/string_scalar”);
```

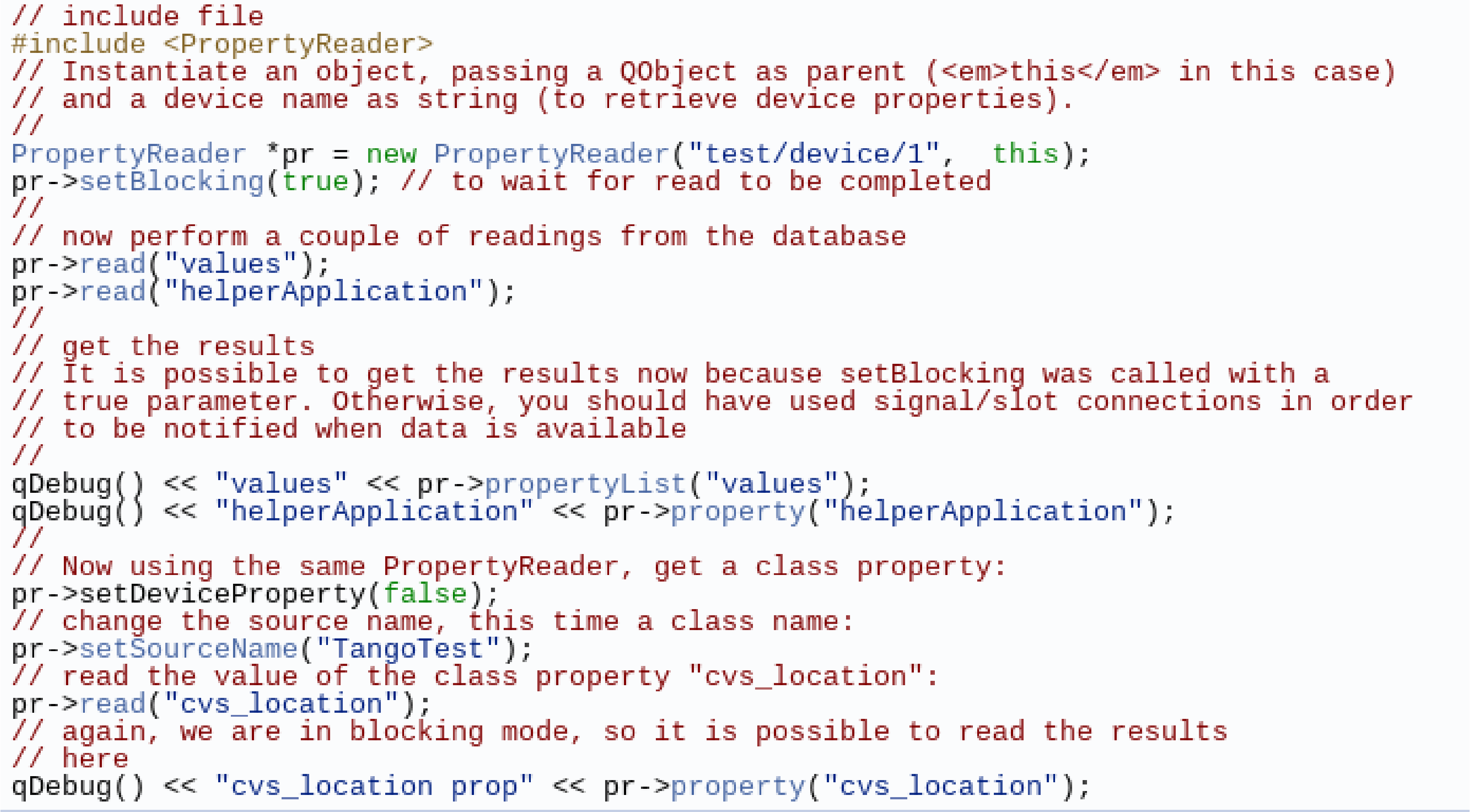
Exercise 3

Write a Qt application where the *TangoTest* attribute *double_scalar* is read by a *QProgressBar* and written by a *QDial*. The value of the attribute is displayed also by a Q*LineEdit*, while a *QLcdNumber* displays only the initial value, read on startup.
A *QCheckBox* reads and writes *boolean_scalar.*
A class member of type short is used to watch the *short_scalar* attribute.

Device and class property readers

PropertyReader

QTangoCore class: reads class and device properties.



```
// include file
#include <PropertyReader>
// Instantiate an object, passing a QObject as parent (<em>this</em> in this case)
// and a device name as string (to retrieve device properties).
//
PropertyReader *pr = new PropertyReader("test/device/1",  this);
pr->setBlocking(true); // to wait for read to be completed
//
// now perform a couple of readings from the database
pr->read("values");
pr->read("helperApplication");
//
// get the results
// It is possible to get the results now because setBlocking was called with a
// true parameter. Otherwise, you should have used signal/slot connections in order
// to be notified when data is available
//
qDebug() << "values" << pr->propertyList("values");
qDebug() << "helperApplication" << pr->property("helperApplication");
//
// Now using the same PropertyReader, get a class property:
pr->setDeviceProperty(false);
// change the source name, this time a class name:
pr->setSourceName("TangoTest");
// read the value of the class property "cvs_location":
pr->read("cvs_location");
// again, we are in blocking mode, so it is possible to read the results
// here
qDebug() << "cvs_location prop" << pr->property("cvs_location");
```

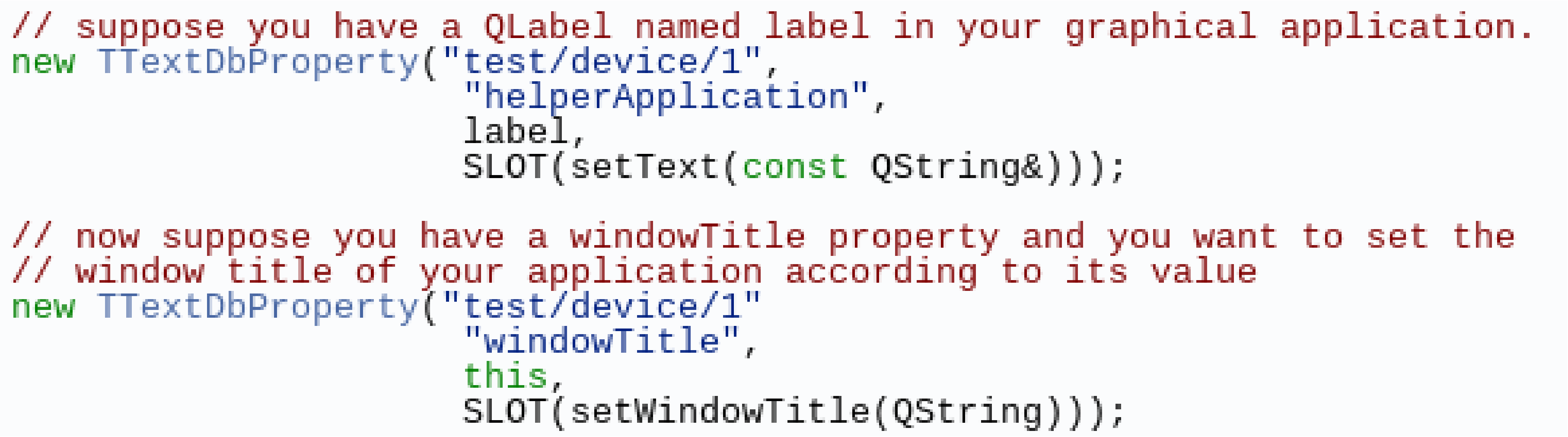
PropertyReader (II)

Errors:

- Connect the error signal to a receiver slot.
- No other way to obtain diagnostic error messages a posteriori.
- Test whether errors occurred with the errorsOccurred method.

TTextDbProperty

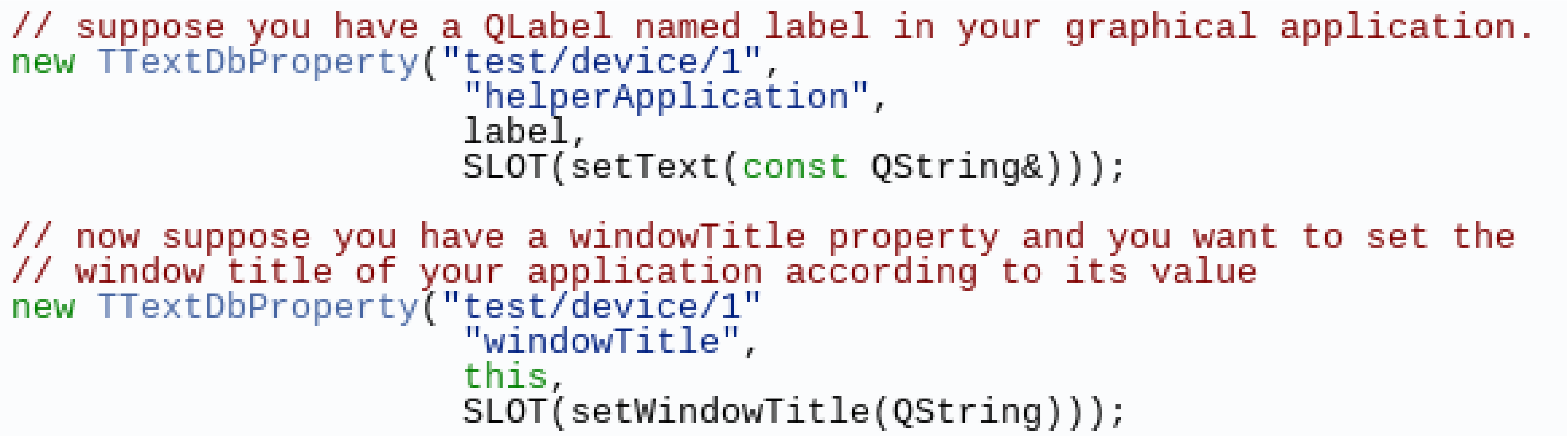
Reads a single value device or class property and invokes a *QObject*'s slot having a single const *QString* reference as argument.



```
// suppose you have a QLabel named label in your graphical application.
new TTextDbProperty("test/device/1",
                    "helperApplication",
                    label,
                    SLOT(setText(const QString&)));

// now suppose you have a windowTitle property and you want to set the
// window title of your application according to its value
new TTextDbProperty("test/device/1"
                    "windowTitle",
                    this,
                    SLOT(setWindowTitle(QString)));
```

TPropertyLabel

A *QLabel* displaying a device or class property value.



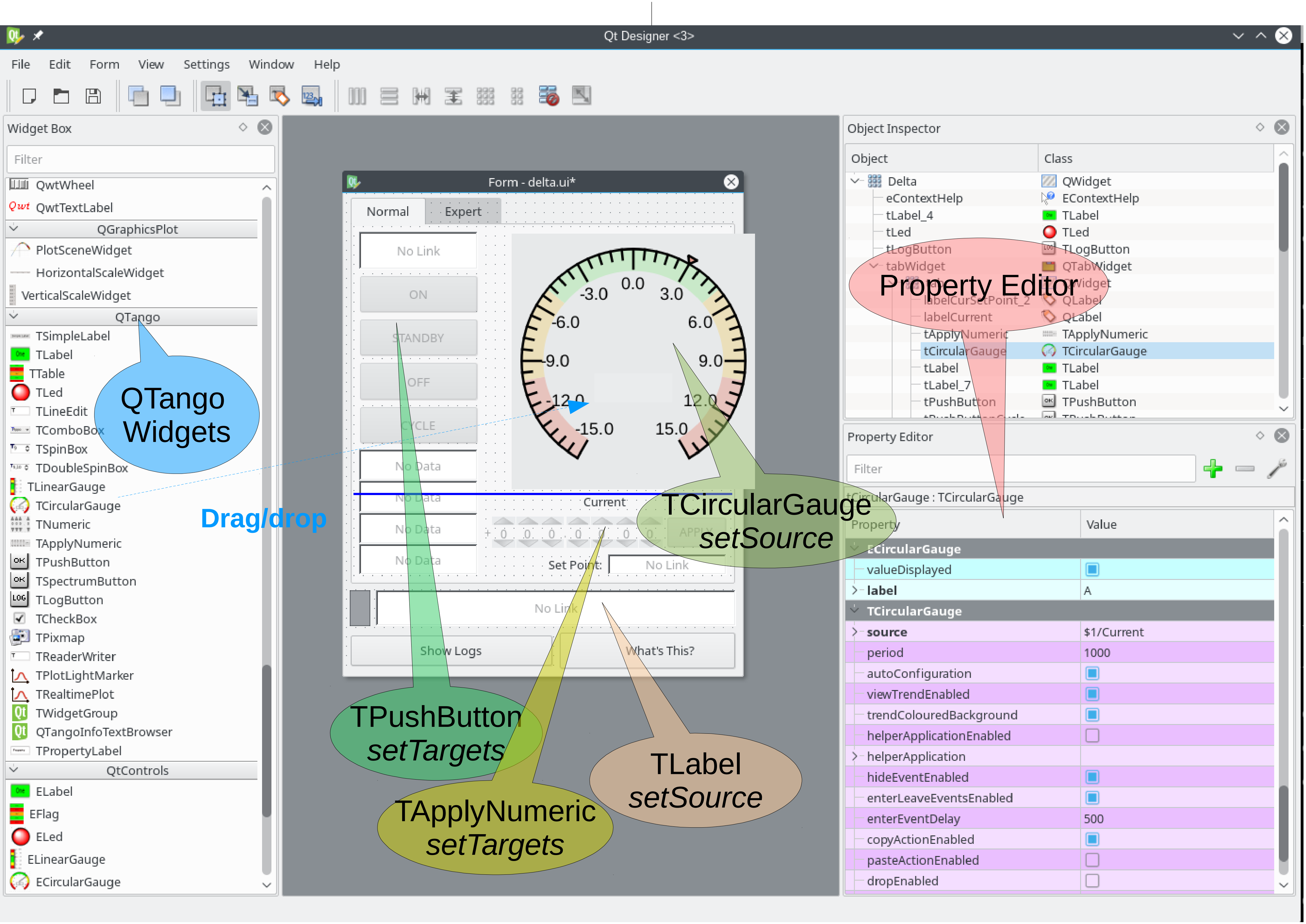
```
// suppose you have a QLabel named label in your graphical application.
new TTextDbProperty("test/device/1",
                    "helperApplication",
                    label,
                    SLOT(setText(const QString&)));

// now suppose you have a windowTitle property and you want to set the
// window title of your application according to its value
new TTextDbProperty("test/device/1"
                    "windowTitle",
                    this,
                    SLOT(setWindowTitle(QString)));
```

The Qt designer

QTango plugins

QTango, *qtcontrols*, *QGraphicsPlot* and *TGraphicsPlot* plugins are available in the Qt designer.

- Drag widgets from the *Widget Box* and drop them into the project widget
- Edit Qt, qtcontrols and QTango properties from the *Property Editor*;
- Right click on a widget to set the *QTango* source or *targets*

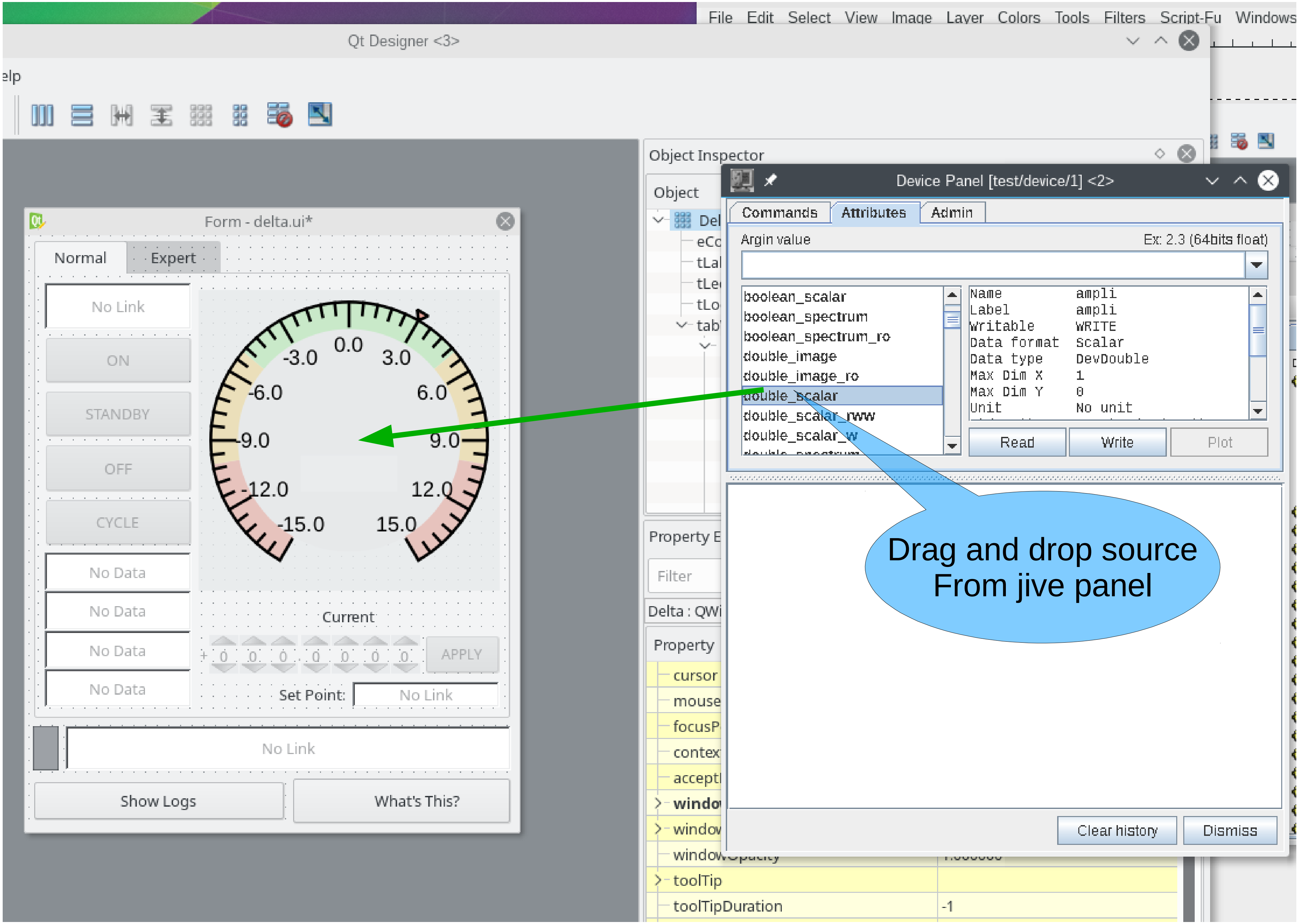


Object	Class
Delta	QWidget
eContextHelp	EContextHelp
tLabel_4	TLabel
tLed	TLed
tLogButton	TLogButton
tabWidget	QTabWidget
	QWidget
labelCurSetPoint_2	QLabel
labelCurrent	QLabel
tApplyNumeric	TApplyNumeric
tCircularGauge	TCircularGauge
tLabel	TLabel
tLabel_7	TLabel
tPushButton	TPushButton

tCircularGauge : TCircularGauge

Property	Value
ECircularGauge	
valueDisplayed	☑
label	A
TCircularGauge	
source	$1/Current
period	1000
autoConfiguration	☑
viewTrendEnabled	☑
trendColouredBackground	☑
helperApplicationEnabled	☐
helperApplication	
hideEventEnabled	☑
enterLeaveEventsEnabled	☑
enterEventDelay	500
copyActionEnabled	☑
pasteActionEnabled	☐
dropEnabled	☐

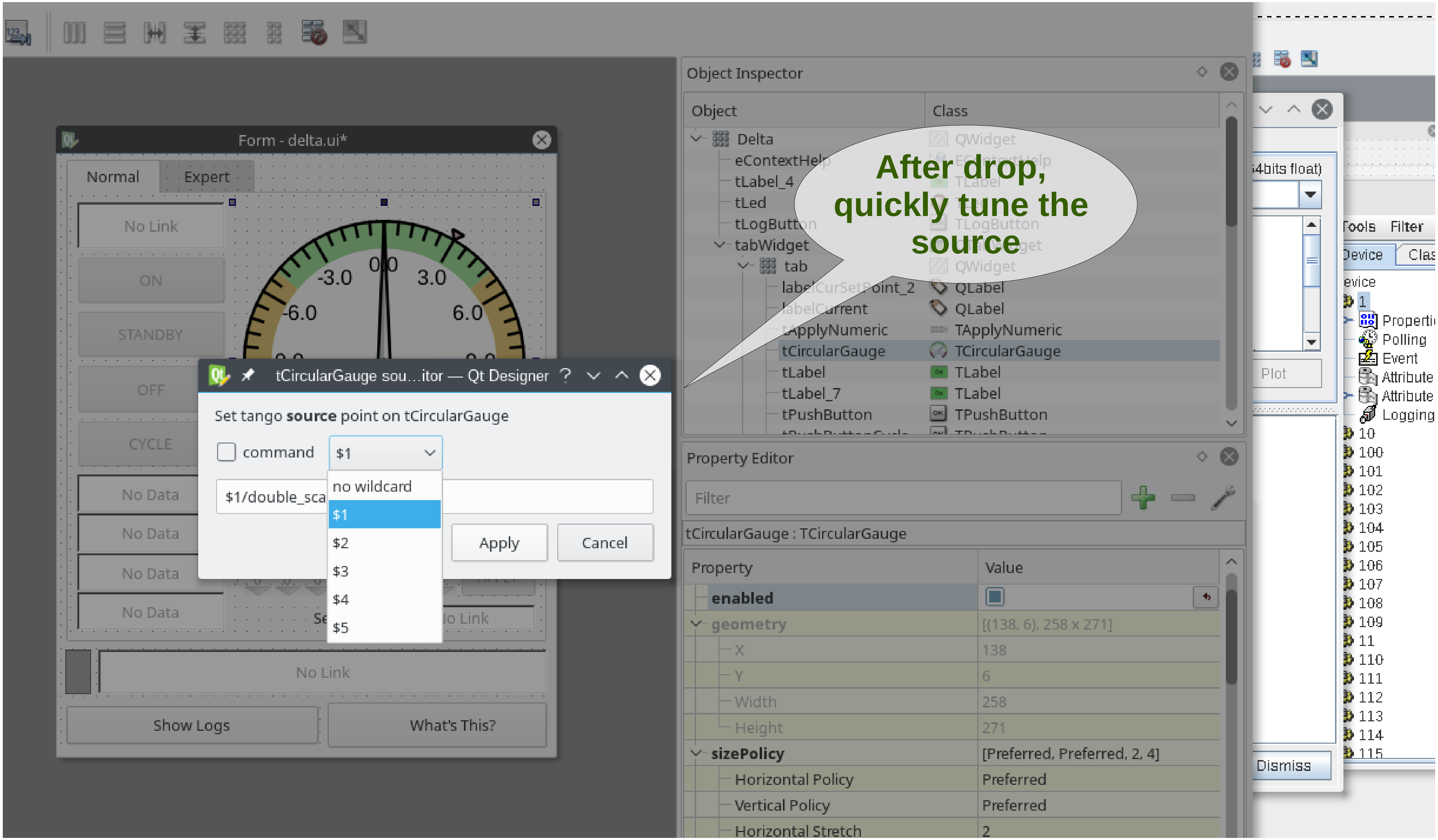
Drag 'n drop from Jive



Drag and drop source
From jive panel

```
Name          ampli
Label         ampli
Writable      WRITE
Data format   Scalar
Data type     DevDouble
Max Dim X     1
Max Dim Y     0
Unit          No unit
```

Drag 'n drop from Jive (II)



After drop, quickly tune the source

tCircularGauge sou…itor — Qt Designer

Set tango **source** point on tCircularGauge

☐ command $1

$1/double_sca

Object Inspector

Object	Class
Delta	QWidget
eContextHelp	EContextHelp
tLabel_4	TLabel
tLed	
tLogButton	TLogButton
tabWidget	QTabWidget
tab	QWidget
labelCurSetPoint_2	QLabel
labelCurrent	QLabel
tApplyNumeric	TApplyNumeric
tCircularGauge	TCircularGauge
tLabel	TLabel
tLabel_7	TLabel
tPushButton	TPushButton

Property Editor

tCircularGauge : TCircularGauge

Property	Value
enabled	☑
geometry	[(138, 6), 258 x 271]
X	138
Y	6
Width	258
Height	271
sizePolicy	[Preferred, Preferred, 2, 4]
Horizontal Policy	Preferred
Vertical Policy	Preferred
Horizontal Stretch	2

QTango plugin components

🟢 readers

🔴 writers

🟣 Readers *and* writers

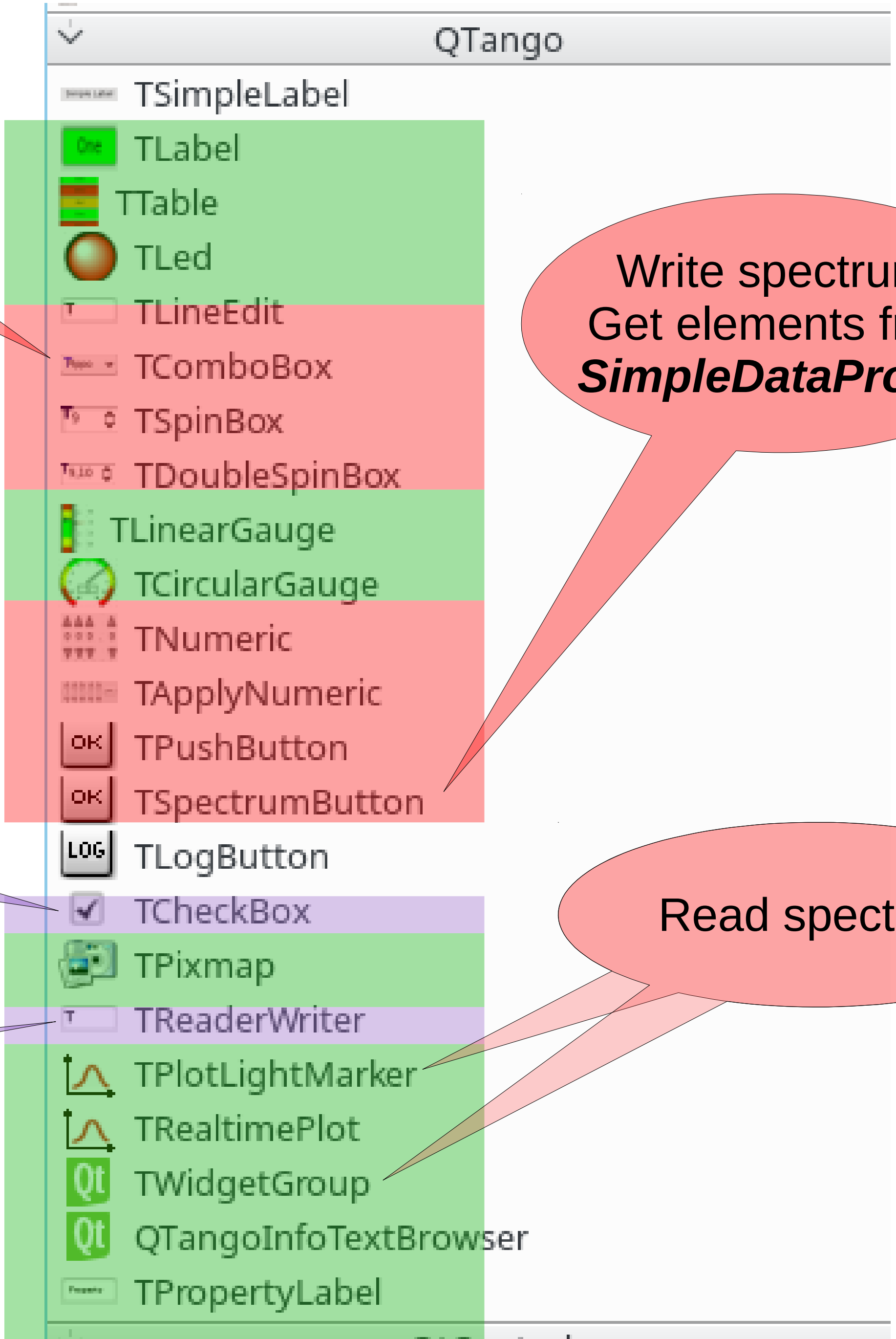
TComboBox:
Initialize with *values*
Attribute property!

Write spectrum.
Get elements from
SimpleDataProxys

NOTE: for historical reasons,
You must edit <u>both</u> source and
Targets

Read spectrum

If no targets set,
targets = source



QTango

TSimpleLabel

TLabel

TTable

TLed

TLineEdit

TComboBox

TSpinBox

TDoubleSpinBox

TLinearGauge

TCircularGauge

TNumeric

TApplyNumeric

TPushButton

TSpectrumButton

TLogButton

TCheckBox

TPixmap

TReaderWriter

TPlotLightMarker

TRealtimePlot

TWidgetGroup

QTangoInfoTextBrowser

TPropertyLabel

QTango plots

TPlotLightMarker (Qwt) with a scalar attribute/command as source:

- Plots attribute value over time
- X axis is configured as a *time scale*

***source**:* semicolon separated list of sources

TPlotLightMarker (Qwt) with a spectrum attribute/command as source:

- Plots spectrum
- X axis is [0, 1, … , spectrum.size() - 1];
- Y axis is [spectrum[0], spectrum[1], … , [spectrum[spectrum.size() - 1]

TRealTimePlot (Qwt) (spectrum, command)

- Tailored for *GetSomething(N, M)* commands used for real time quantities.
- Only for commands that return a vector.
- Tested at Fermi with several curves refreshed at 10Hz.

QTango plots (II)

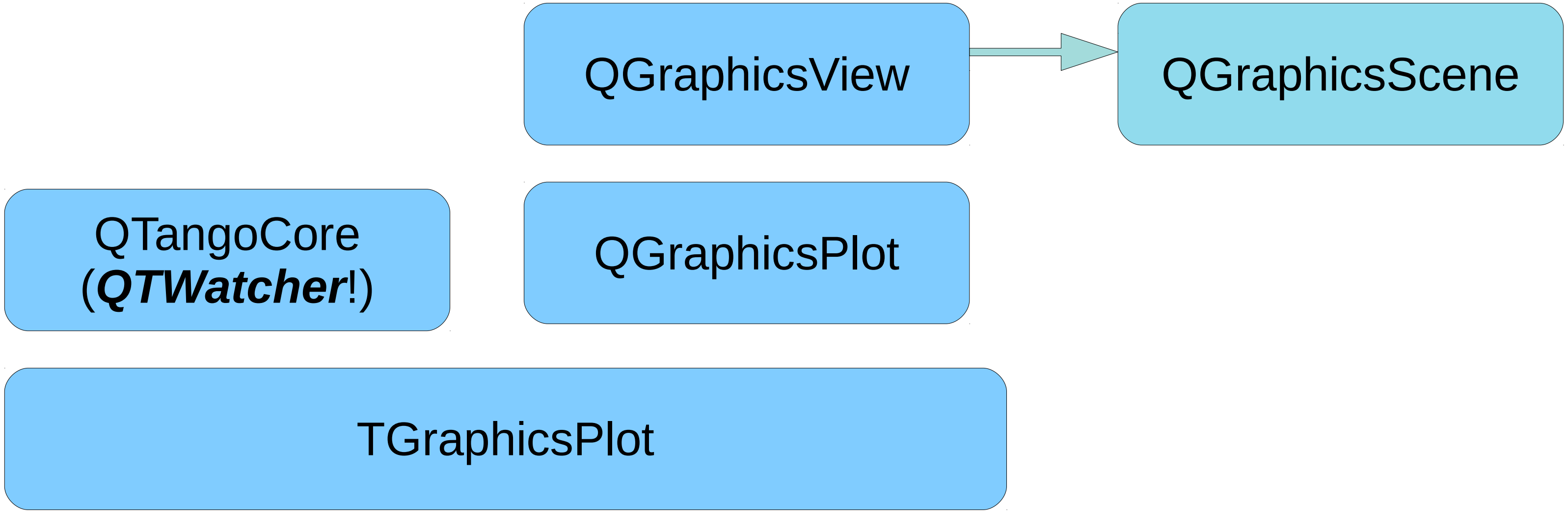
TGraphicsPlot

***sourcesList**:*
QStringList of sources

QGraphicsView / QGraphicsScene technology

- a surface for managing and interacting with a large number of custom-made 2D graphical items
- a view widget for visualizing the items
- support for zooming and rotation

http://doc.qt.io/qt-4.8/graphicsview.html



Exercise 4

Qt designer

Using the Qt designer, make an application made up of QTango widgets only.
Configure the *test/device/1* server so that the attribute *short_scalar* has a property named *values* which is a list of strings (try with at least 4 elements).
Configure the *test/device/1* with a device property named *helperApplication* and value *xclock*.
A TabWidget with two tabs shows in the first page:

- A TLabel will read the *short_scalar*, an associated TComboBox allows to select one of the available *values*, and a *TPushButton* will write the attribute.
- A TLabel displays the state of the device.
- A TPushButton “SwitchStates” executes that command.
- A TReaderWriter is connected to *double_scalar.*
- A TReaderWriter reads *string_scalar.*
- A TPlotLightMarker reads *double_scalar* and *long_scalar*.
- A TGraphicsPlot reads *double_spectrum_ro and long_spectrum_ro.*

Exercise 4 (II)

Qt designer

In the second page:

- a TWidgetGroup reads the *double_spectrum*
- a set of 4 *TDoubleSpinBox* write the first 4 elements of the vector when an Apply *TSpectrumButton* is clicked.
- A plot of your choice connects to the same attribute.
- A TTable with 10 rows and 2 columns displays the *boolean_spectrum* attribute.

➡ Set *double_spectrum* range in Tango database and verify that the double spinboxes are correctly configured.

➡ If you configure *short_scalar values* property, remember to limit the range of the attribute accordingly. For example, if you put 6 string values, limit the attribute between 0 and 5 (Simply apply a value from the combo box list).

➡ *IndexMode* property on *TComboBox* and the *configureEnumFromValuesProperty* on *TLabel* must be enabled.

Part IV

Writing *QTango* - ready Tango servers

- Correctly shape the *Tango* server paying special attention to **command** and **attribute** modeling;
  - commands only when they suit the device model;
  - no commands with strings as *argin* and/or *argout*;
  - put logic on the server rather than in the panel, as much as possible

Documentation

- *QTango* documentation is installed inside the *share* folder under the root installation of qtango (see qtango.pri project file)
- QTango documentation is in the *html* format.

Logging

- QTangoCore provides console coloured messages:

* ***error*** *message*

* ***warning*** *message*

* ***ok*** *message*

Enable them exporting ***QTANGO_PRINT=1*** *in the terminal*

Elettra
Sincrotrone
Trieste

The End

- **Thanks for your attention**

mailto: giacomo.strangolino@elettra.trieste.it