BLISS

Beamline Instrument Software Support

Graphical User Interfaces for beamlines :

the Bliss Framework project

Author : Matías Guijarro

Date : 10/7/2004

Contents

Figures

1. Introducing the Bliss Framework project

1.1 History and goals of the project

Graphical tools dedicated to experiments on beamlines have been used since the beginning at ESRF. The Beamline Instrument Software Support group has always been actively involved in providing software programs to fulfill the need of high-level user-friendly graphical interfaces for applications. Control system-oriented tools like SpecGUI, specialized tools like ProDC or imaging tools like Onze or MCAGui are examples of useful applications developed over the years. Since the beginning heterogenous graphical applications and command-line tools have cohabited and grown up together to bring the best support to the users.

In 2001, the idea of giving more flexibility when building GUI applications emerged. When writing applications, object-oriented programming put emphasis on *reusability*. The same key principle could be adapted to graphical user interfaces. Creating a graphical user interface for a beamline should be viewed like "cementing" software "bricks" together. A lot of questions arised : what are the common services a beamline GUI application need ? What are the base GUI components ? What could be a standard *look & feel* for beamline graphical user interfaces ? How can connection with the control systems be made transparent ?

Nowadays, the last evolutions in the control software on beamlines and in the programming languages and graphical toolkits permit to consider a more global approach when thinking about beamline GUI building.

The Bliss Framework is currently one of the strategic projects of the BLISS group at the ESRF. This project responds to the always growing demands for experiment automation together with the latest trends in beamline control. Its aim is providing users with an improved ergonomics in a graphical environment focused only on experiments rather than on the underlying control systems.

The main goals of the project are :

- **to help Bliss people to create GUI on beamlines**
- **to permit reusability of beamline GUI components**
- **to abstract GUI programming from control systems (Spec, Taco, Tango, Epics, etc.)**
- **to achieve maximum reliability and ease-of-use, without conceeding to functionalites**

1.2 Technical choices

The Bliss Framework project follows the last technical choices adopted by the Bliss group :

- programming language : **Python**
- GUI toolkit : **Trolltech Qt** (interfaced via PyQt by Phil Thompson)

1.3 The different aspects of the Bliss Framework project

The Bliss Framework project covers different aspects from hardware and control system abstraction to end user graphical interfaces.

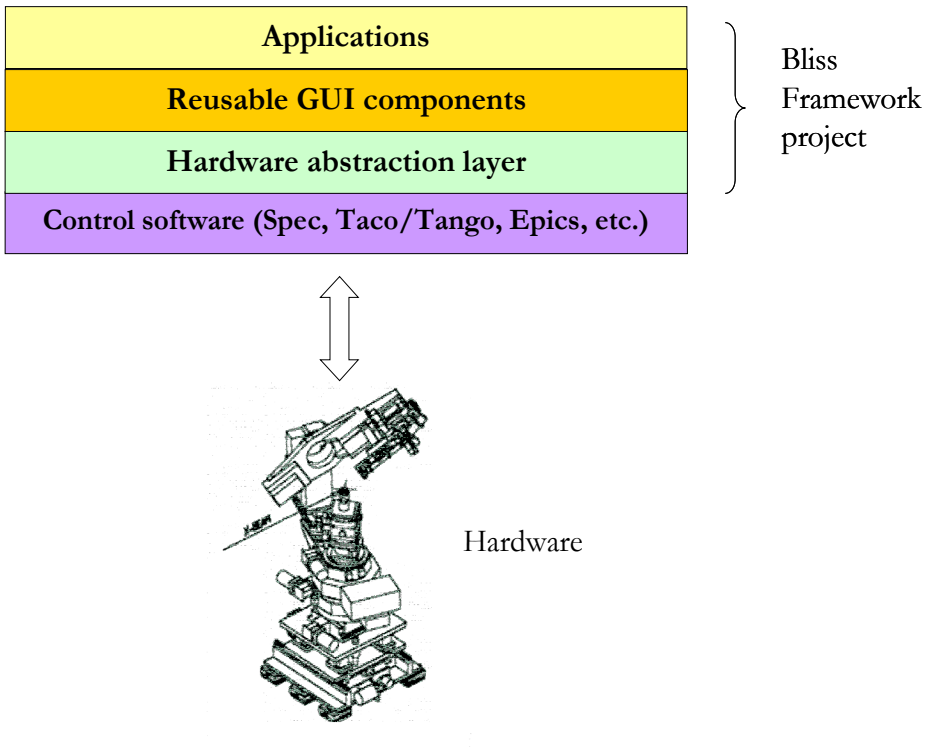


Fig. 1 : the different aspects of the Bliss Framework project

There are two main layers in the Bliss Framework project to consider :

- one is the **hardware and control system abstraction layer**, called "**Hardware Repository**"
- the other is t**he set of reusable GUI components**, called "**bricks**"

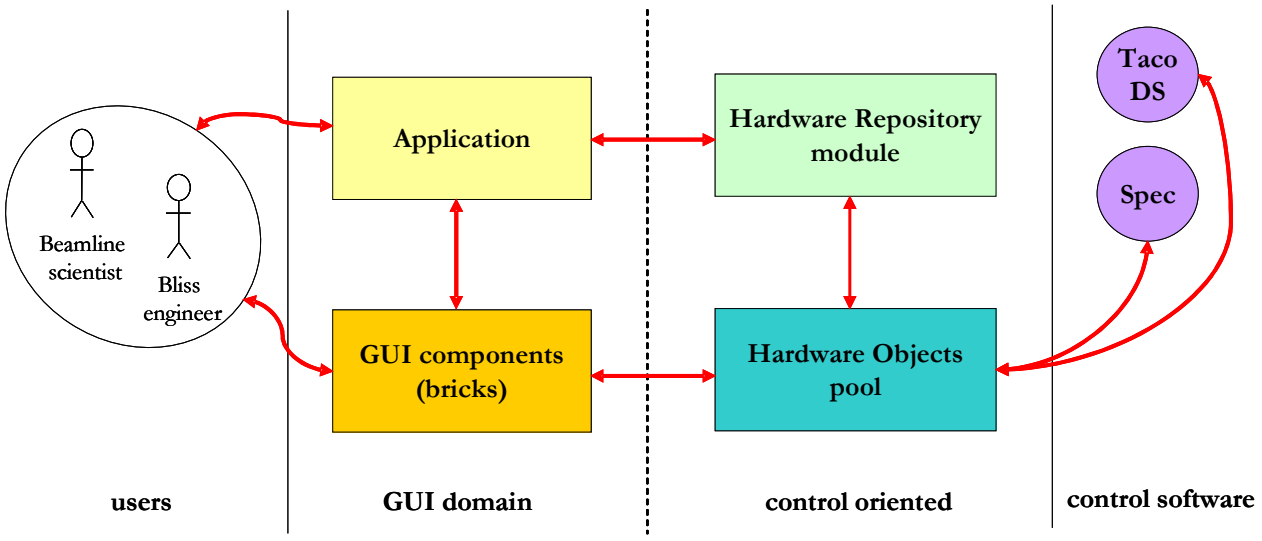


Fig. 2 : interactions between the Bliss Framework project entities

1.3.1 Hardware abstraction

The Hardware abstraction, or the ability to consider a hardware device as an "object", without worrying about how it is connected to the control software when building a GUI, is achieved through the Hardware Repository.

The Hardware Repository holds the description of the devices and equipments of a beamline in a database. The database is **a set of XML files** ; each file represent at least one Hardware Object. The Hardware Repository Python module manages a pool of Hardware Objects.

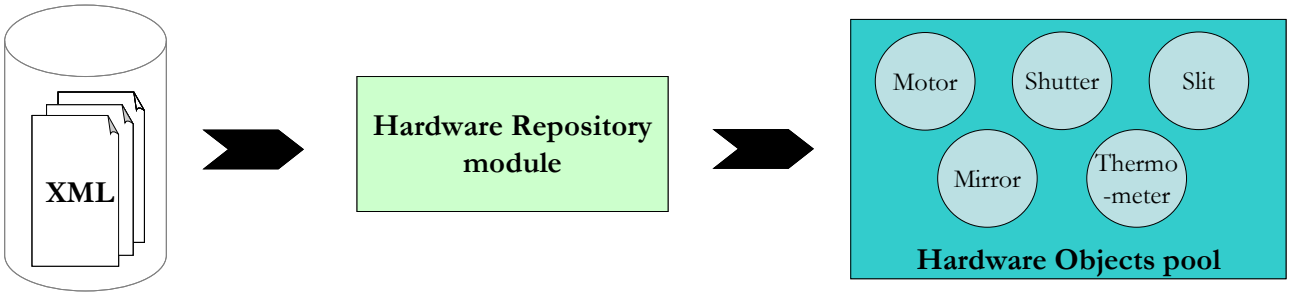


Fig. 3 : Hardware abstraction through the Hardware Repository

1.3.2 Using bricks as building blocks

One of the main goal behind the Bliss Framework project is to **permit reusability between GUI components** when building a beamline graphical user interface, therefore the Bliss Framework project rely on GUI components called "**bricks**" (**meta-widgets**).

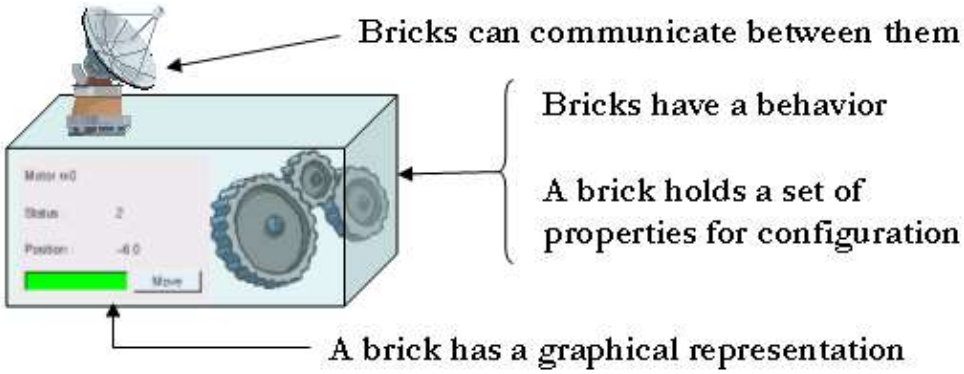


Fig. 4 : the Bliss Framework project rely on "bricks" as building blocks

These bricks can be :

- **application-oriented**, for example, a Menu brick for adding menus in GUI applications
- **control-oriented**, for example, a Motor brick for moving a motor managed by a control system
- or **experiment-oriented**, for example, a Sample Centering brick for protein crystallography beamlines.

1.4 Putting it all together

The Bliss Framework project is based both on Python libraries (modules and packages) and on a set of tools built upon.

A great attention has been granted to the software design of each entity involved in the project, to ensure reusability, consistency and a certain level of independance. In particular, the libraries can be used stand-alone, e.g when building non-graphical applications that would take advantage of the hardware abstraction facilities provided by the Hardware Repository.

The recommended way to use the Hardware Objects of the Hardware Repository is to connect them to the GUI bricks, thus creating a link between hardware (control software) and the graphical components.

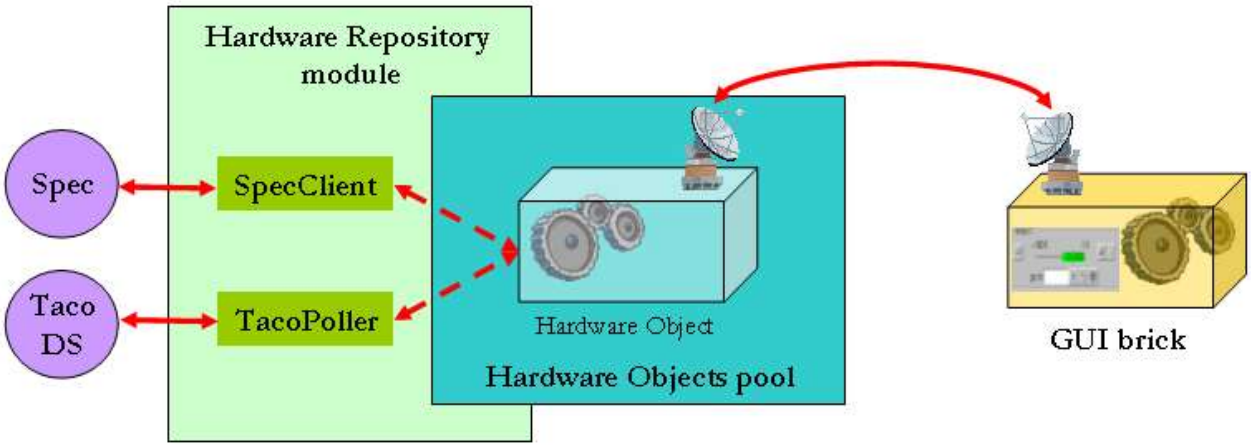


Fig. 5 : Hardware objects communicate both with the control software and with the GUI bricks

The Bliss Framework project includes a **GUI builder tool**, for rapid interface generation. The **standard application**, called "**Exes**", is another part of the Bliss Framework project : it is a sample, full-featured, ready-to-use application made with the Framework components. It includes the GUI builder and allows switching between design mode and execution mode, meaning **it is possible to modify a GUI while it is running**.

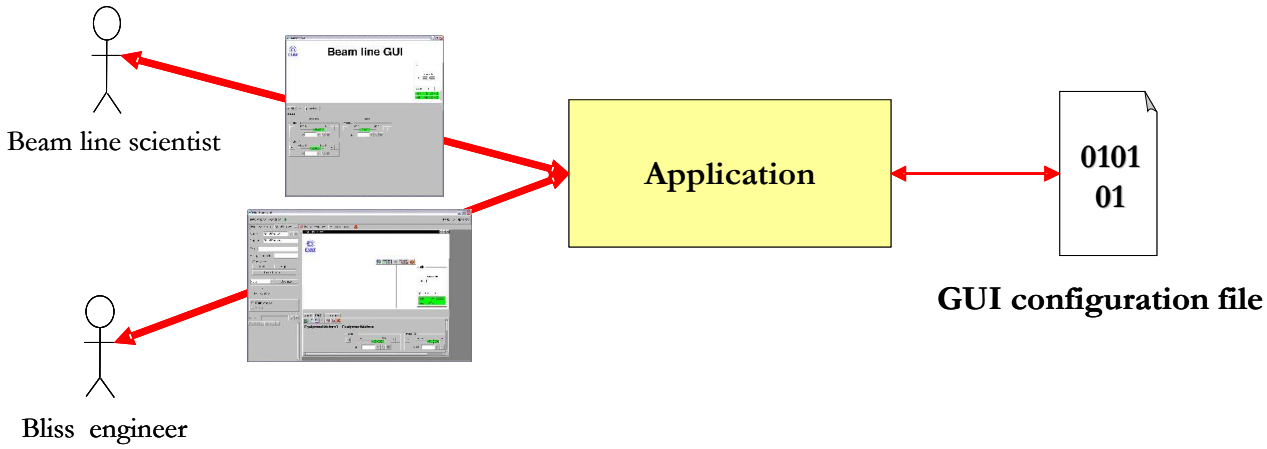


Fig. 6 : the standard application, Exes, has two "modes" for two kinds of users

2. The Hardware Repository

When building a beamline GUI, the control part of the application has to deal with a wide variety of hardware devices and control systems.

The Hardware Repository and its Hardware Objects are the control layer of the Bliss Framework project.

2.1 Describing hardware devices and control software

XML files are a very convenient way for describing hardware configuration on beamlines and organising the information required for software programs in general. It is an open format, easy to edit and human-readable, enforcing a hierarchical representation of data structures.

The Hardware Repository database can be viewed like a set of XML files *describing* "Hardware Objects".

2.1.1 Binding between XML files and Python objects

Every XML file in the Hardware Repository represents one Hardware Object. The binding between the information stored in the files and Python objects is accomplished by a module of the Hardware Repository Python package.

Almost any XML file can be converted to a "Hardware Object" ; only a few reserved XML tags and rules have been defined, for creating Hardware Objects with a predefined behaviour or meaning very easily.

We distinguished between 3 kinds of predefined Hardware Objects :

- **Device** : Device objects represent real devices in the beamline, e.g a motor, a shutter, etc. These objects should report a status, and generally require a connection to a control system
- **Equipment** : Equipment objects refer to real equipments like slits, mirrors, monochromators, i.e hardware elements containing other devices. These objects communicate with the lower level devices they are composed of, and also report a status. They offer a particular interface for accessing the contained devices.
- **Procedure** : Procedure objects are special to the Hardware Repository since they do not correspond to real hardware elements. They represent actions that can be executed by a control system. They can be bound to an Equipment to make it realize the procedure, or used stand-alone.

Hardware Objects are not limited to these 3 types. In fact, as many types of Hardware Objects as necessary can be created, as long as they obey to the OO model defined for Hardware Objects.

From the Python side, a Hardware Object is just a class written in a module. **These modules can be placed in a directory and act as plugins for the Hardware Repository**. The data for creating a Hardware Object instance **comes from the corresponding XML file**.

Let us take an example. The following XML file describes a vanilla Hardware Object :

```
exampleset.xml :


<exampleSet>
  <description>I am a set of examples</description>
  <name>Basic examples</name>
  <example>
    <name>Hello, world</name>
  </example>
  <example>
    <name>Example 2</name>
  </example>
</exampleSet>

Python :


Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
>>> from HardwareRepository import HardwareRepository
>>> hwr=HardwareRepository.HardwareRepository('boom:hwr')
>>> hwr.connect()
>>> exampleSetObject = hwr.getHardwareObject('/exampleset')
>>> print exampleSetObject.description
I am a set of examples
>>> print exampleSetObject['example']
[<HardwareRepository.HardwareObjectNode instance at 0x41e5d84c>,
<HardwareRepository.HardwareObjectNode instance at 0x41e5d74c>]
>>> print exampleSetObject['example'][0].name
Hello, world
>>>
```

Fig. 7 : Example XML file and how to access the elements using the Python object

When using a Hardware Object, you can access its properties by calling them a**s if they were standard variables of the instance**. **Assigning to a property is supported**, and **rewriting XML files is possible** by calling the `commitChanges()` method. When dealing with a Hardware Object containing nested nodes, accessing a child from its parent can be made by using its name **as a dictionary key**. **Integer indices are also supported** to specify a child in particular.

In the previous example, the root node name of the XML file is "exampleSet". Any root name can be used : a HardwareObject instance will be created. Special root names are : "device", "equipment" and "procedure". They correspond to the 3 types of Hardware Objects we defined. In this case, a Device, Equipment or Procedure Hardware Object is created.

Often one may want to create a Hardware Object acting like a device for example, but with some **additional behaviour**. The "**class**" keyword attribute permit to specify which class has to be instanciated for interpreting the information stored in the XML file. By default, the class is **the base HardwareObject class.**

The following XML file corresponds to a device Hardware Object representing a motor controlled by Spec :

phi.xml :

```
<device class="SpecMotor">
  <username>phi motor</username>
  <specname>phi</specname>
  <specversion>lid231:eh1</specversion>
</device>
```

Fig. 8 : XML file describing a motor controlled by Spec

The difference between a standard Device object and this SpecMotor object is that the Spec motor will try to connect to its corresponding Spec version. The code for connecting to the control system is written in the SpecMotor Python class. The information required for connecting is stored in the XML file.

When the Hardware Repository returns a Hardware Object, **it is initialized and should be ready to use**. In this case, connection to Spec should have been done, and the device is able to report its status.

It is possible to **add references to other Hardware Objects** in a Hardware Object using the "**hwrid**" (or "**href**") keyword attribute. This is mainly useful when writing equipments descriptions for example, because it is very likely that the different devices of the equipment are already present in the Hardware Repository.

It is also possible **to define roles** for every Hardware Object node element in a Hardware Object by using the "**role**" keyword attribute. Afterwards in Python retrieving the corresponding object is achieved with the `getObjectByRole()` method.

The example below shows a primary slit equipment XML file and some Python code to access the motors defined in the object :

pslit.xml :

```
<equipment>
  <username>Primary Slit</username>
  <motors>
    <blades>
      <device role="vo" hwrid="/eh1/psvo"/>
      <device role="vg" hwrid="/eh1/psvg"/>
      <device role="ho" hwrid="/eh1/psho"/>
      <device role="hg" hwrid="/eh1/pshg"/>
    </blades>
    <slits>
      <device role="up" hwrid="/demo/psu"/>
      <device role="down" hwrid="/demo/psd"/>
      <device role="left" hwrid="/demo/psf"/>
      <device role="right" hwrid="/demo/psb"/>
    </slits>
  </motors>
</equipment>
```

```
Python :

Python 2.3.3 (#1, May 13 2004, 14:45:22)
[GCC 2.95.3 20010315 (SuSE)] on linux2
>>> from HardwareRepository import HardwareRepository
>>> hwr=HardwareRepository.HardwareRepository('boom:hwr')
>>> hwr.connect()
>>> pslitObject = hwr.getHardwareObject('/pslit')
>>> pslitObject.username
'Primary Slit'
>>> pslitObject.getObjectByRole('up')
<HardwareRepository.Device instance at 0x41e648ac>
>>> pslitObject.getObjectByRole('up').username
'psu'
>>>
```

Fig. 9 : XML description and sample Python code for the Primary Slit Equipment

2.1.2 Hardware Objects OO model

Behind the scene, various classes have been designed to be part of the Hardware Objects Object-Oriented model. When reading and parsing the XML files, the corresponding objects are instanciated. This model can be viewed as a lightweight and limited DOM-like (Document Object Model) implementation, tailored to the needs of the Hardware Repository.

The core classes are :

- **HardwareObjectNode** : base class for every Hardware Object and every node in Hardware Objects' XML description
- **PropertySet** : a dictionary-derived class, to represent the set of properties of a HardwareObjectNode
- **HardwareObject** : class representing the root Hardware Object node of a XML file. It derives from QObject, thus objects of this class can emit signals for communicating with graphical bricks
- **Device** : Device Hardware Object
- **CommandDevice** : a CommandDevice object acts like a Device, but its behaviour is defined through the commands it can execute on a control system. It does not correspond necessarily to a real hardware device (pseudo-device)
- **DeviceContainer** : class dedicated to represent Hardware Objects that contain Device objects. It provides helper methods to count devices, to get devices by name, etc.
- **Equipment** : Equipment Hardware Object
- **Procedure** : Procedure Hardware Object
- **CommandContainer** : this mixin class can be added to an existing class in order to bring the ability to execute commands and read variables from a control system.

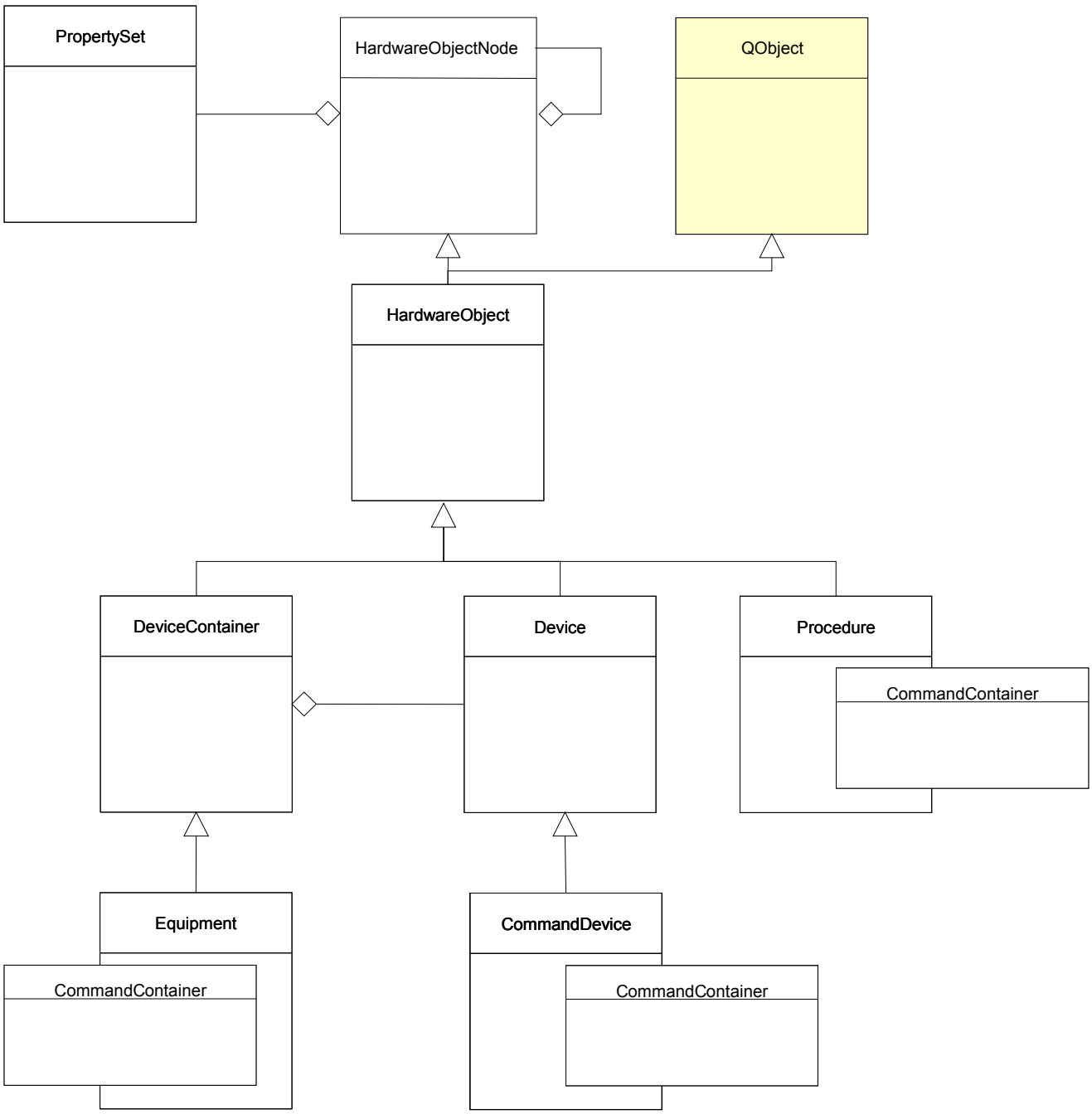


Fig. 10 : class diagram of core Hardware Repository entities

2.2 Client / Server Hardware Repository

There is **one** Hardware Repository database per beamline. This means that more than one graphical user interface may access the Hardware Repository files at the same time, both for reading (loading a Hardware Object) and writing (updating a Hardware Object properties).

We naturally oriented the Hardware Repository to a client / server model. The Hardware Repository Server delivers the contents of the XML files to its clients, and rewrites the XML files on change.

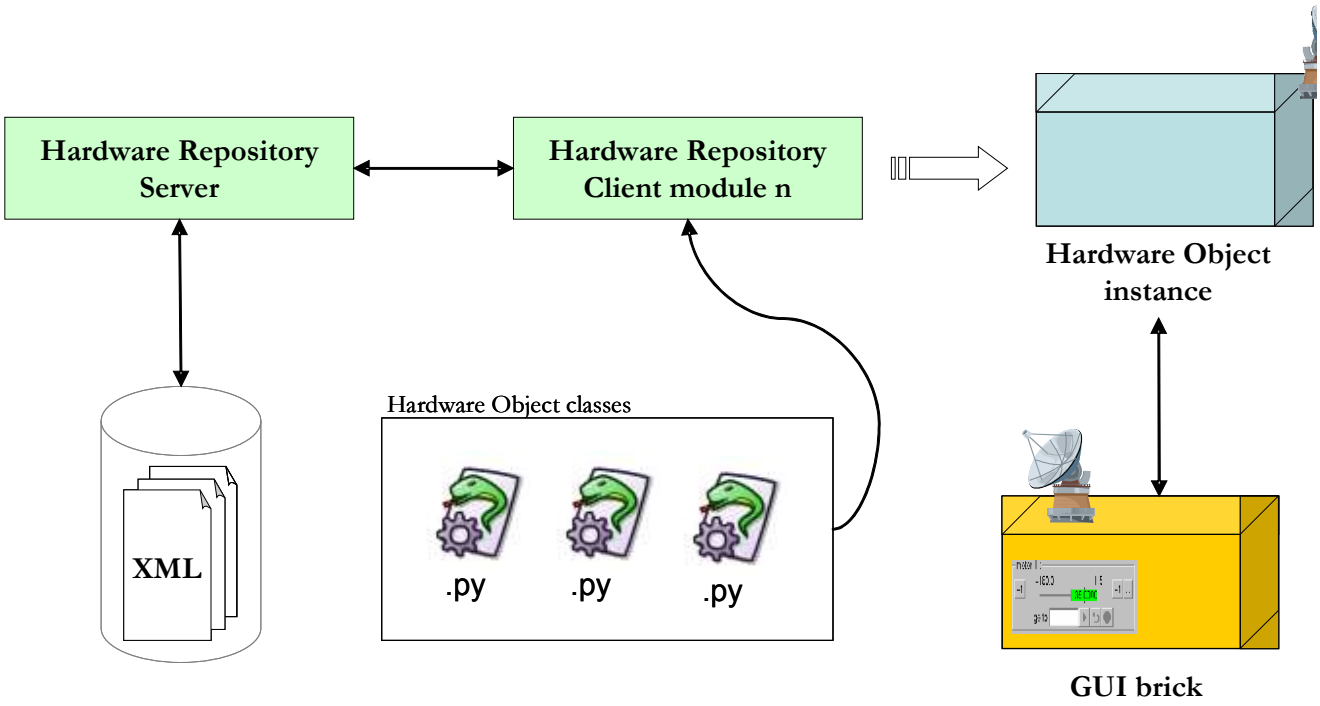


Fig. 11 : overview of client /server Hardware Repository interactions

3. GUI building with bricks

The Bliss Framework project relies on bricks as building blocks for creating graphical user interfaces. The bricks, once written and tested, can then be put on any beamline GUI. From the support group point of view, this ensures reusability, reliability and facilitates deployment. From the users point of view, this guarantees a common *look & feel* on any beamline and some standard behaviour. Moreover they benefit of software developments made on other beamlines more quickly.

3.1 Bricks in the detail

Bricks can be considered as **meta-widgets**. This means that they play at a higher level. As widgets they are dedicated to a particular task, but they are not single graphical components with a limited functionality : on the contrary they can have a lot of features.

From the Python side, bricks are classes deriving from the BlissWidget base class. Each brick class definition should be put in one Python module file. When loading a brick, the name of the module is supposed to be the name of the brick class. The BlissWidget itself derives from the QWidget class : it is a standard Qt graphical component. Basically it just adds some methods to be overwritten by subclasses.

Bricks have a graphical user interface : a brick contains standard Qt widgets inside. Bricks also have a **"property bag"**, i.e a set of **persistent properties** (key-value pairs). This allow bricks to keep their configuration inside a GUI between executions. As any QObject-derived class, **bricks can define signals and slots** to be connected between them or to Hardware Objects for example (see 2. "The Hardware Repository").

3.1.1 Bricks have a GUI

Every brick is a graphical component. The final appearance of a brick may depend on its configuration (see 3.1.2 "Bricks have properties"). Standard widgets are arranged inside a brick at initialization time, either manually (in the code) or by reading the contents of a .ui file created with the Qt Designer tool.

Fig. 12 : the standard Motor brick GUI

3.1.2 Bricks have properties

Every brick is attached to a "property bag". A property bag is an instance of the PropertyBag Python class defined in the Framework. PropertyBag objects bring persistence to the bricks : they can save their state between executions. Thus, when loading an existing beamline GUI, each brick can remember its previous configuration and show up accordingly.

A brick can define as many properties as it wants. A property is represented by three elements :

- a **name** (key)
- a **type**
- a **value**.

The available types for a property are :

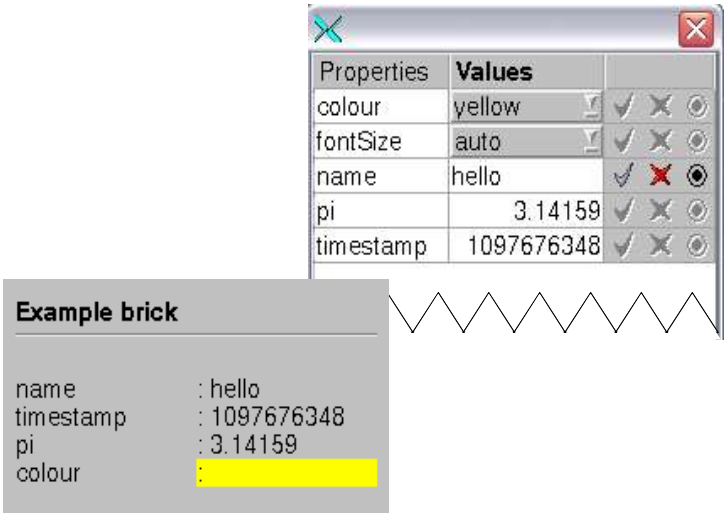
- **string**, for storing arbitrary strings
- **integer**, for storing integers
- **float**, for storing float numbers
- **combo**, for storing a list of possible string values
- **file**, for storing paths to files

It is allowed to not specify a type for the data, and to store any serializable Python value in a property. These properties are called "hidden properties", because they do not show in the Property Editor.

The Property Editor is a graphical component for editing the properties of property bag objects. By default the `editProperties()` method of a brick makes it pop up.

The following example shows the code for creating a brick with properties, and the corresponding Property Editor window :

```
import time
from qt import *
from BlissFramework.BaseComponents import BlissWidget


class Test(BlissWidget):
  def __init__(self, name, parent):
    BlissWidget.__init__(self, parent, name)


    self.addProperty('name', 'string', self.name())
    self.addProperty('timestamp', 'integer', time.time())
    self.addProperty('pi', 'float', '3.14159')
    self.addProperty('colour', 'combo', ('blue', 'yellow', 'red'),
'blue')


    gridPanel = QGrid(2, self)
    QLabel('name', gridPanel)
    self.lblName = QLabel(gridPanel)
    QLabel('timestamp', gridPanel)
    self.lblTimestamp = QLabel(gridPanel)
    QLabel('pi', gridPanel)
    self.lblPi = QLabel(gridPanel)
    QLabel('colour', gridPanel)
    self.lblColour = QLabel(':'+6*' ', gridPanel)


    QVBoxLayout(self)
    self.layout().addWidget(QLabel('<b>Example brick</b><br><hr>',
self))
    self.layout().addWidget(gridPanel)
    self.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)


  def propertyChanged(self, property, oldValue, newValue):
    self.lblName.setText(': %s' % self['name'])
    self.lblTimestamp.setText(': %s' % self['timestamp'])
    self.lblPi.setText(': %s' % self['pi'])
    self.lblColour.setPaletteBackgroundColor(QColor(self['colour']))
```



3.1.3 Bricks can communicate

As any QObject-derived class, a brick can define signals and slots. Bricks can emit signals, and other QObject-derived object signals can be connected to bricks slots.

This establishes a nice way of communicating between bricks, but not only : bricks can also be connected to Hardware Objects signals, and thus receiving update events from control systems and hardware devices.

Usually bricks define a **"mnemonic"** string property dedicated to receive the name of a Hardware Object in the Hardware Repository. Since then, the brick knows it should get a Hardware Object, and establish the required connections to the Hardware Object signals.

Every brick is specialized : it recognizes only a few Hardware Objects. For example, the EquipmentMotors brick permits to control the motors of an Equipment Hardware Object. Do you remember the Primary Slit hardware object ? (see 2.1.1 "Binding between XML files and Python objects")

Here is how the EquipmentMotors brick represents this object :

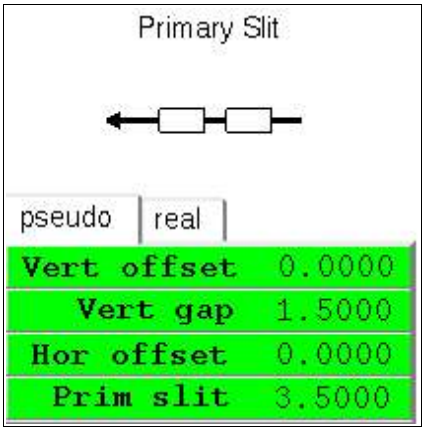


Fig. 14 : the EquipmentMotors brick is connected to a Hardware Object

The brick has established the necessary connections with the Equipment Hardware Object in order to be informed of changes in any of the motors of the primary slit.

3.2 Overview of the beamline GUI editor

In order to create beamline graphical user interfaces, the beamline GUI editor provides the "cement" for gluing bricks together.

3.2.1 An integrated environment for beamline GUI design

The GUI editor's main goal is to bring a lightweight, easy-to-use yet full-featured environment for building beamline graphical user interfaces.

The question of developing a "new" kind of IDE arised at the same time the first bricks were written. At first, we thought about using the Qt Designer as our tool for GUI building. Finally we found that a specialized tool, dedicated to generate beamline interfaces, is the best compromise between time-to-the-user and flexibility.

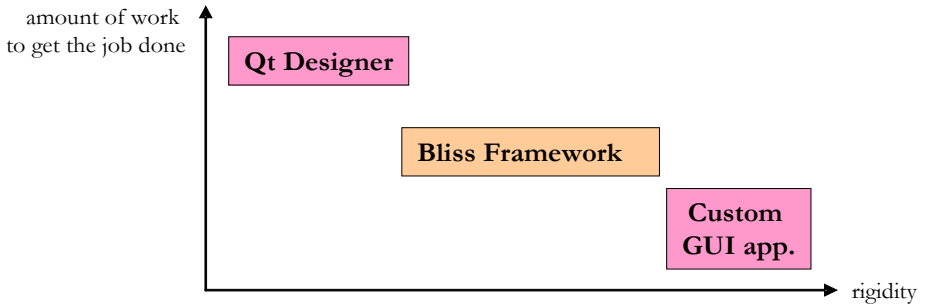


Fig. 15 : the GUI design tool is the best compromise between time-to-the-user and flexibility

The GUI editor relies on the Framework facilities. It allows the user to add windows (containers) to a beamline GUI. A beamline GUI has one window at least. Each window have one or more grids. Bricks can be arranged inside these grids. The grids are flexible : rows and columns can be added. On the left, all the properties that can be changed (name of windows, captions, bricks properties, etc.) are shown.

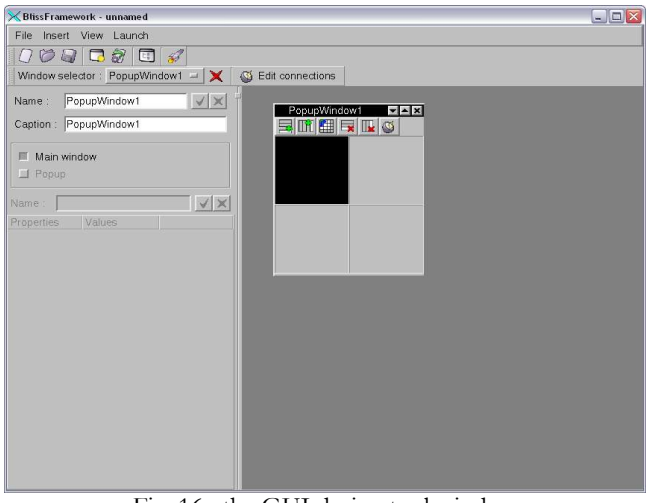


Fig. 16 : the GUI design tool window

3.2.2 Adding bricks to a beamline GUI

When adding a brick to a beamline GUI with the editor, first of all a window must have been created. Windows are containers for bricks. Each window has at least one grid, where bricks can be put inside.

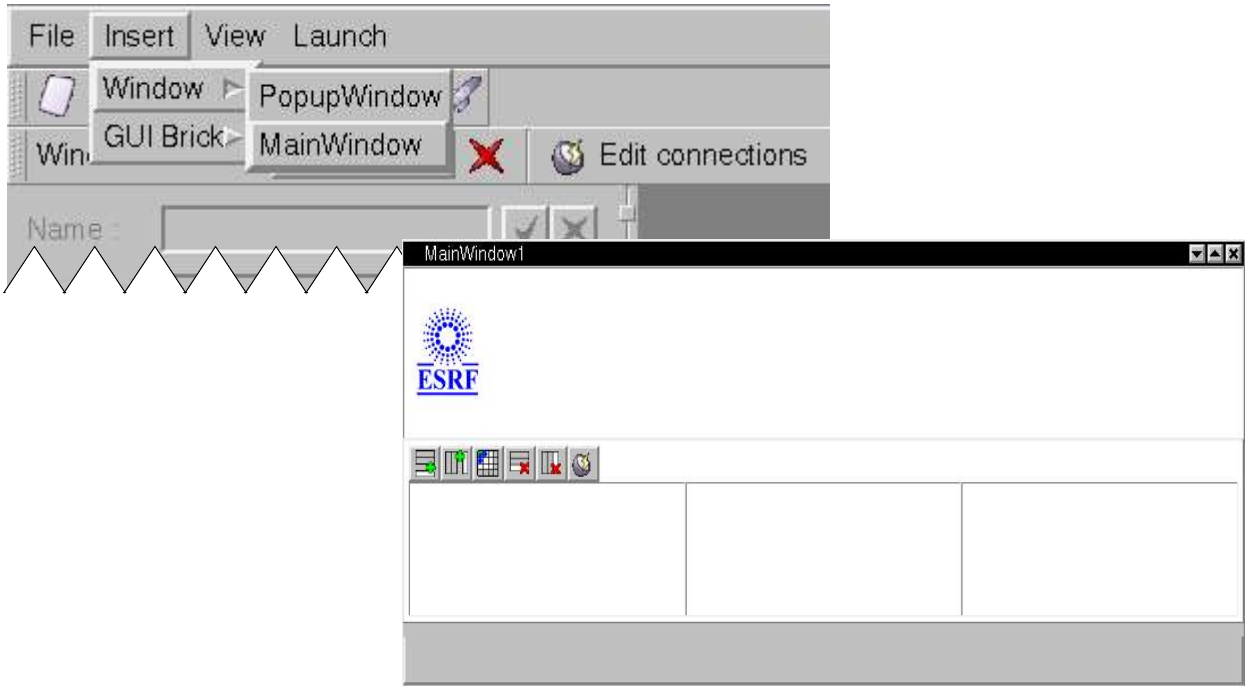


Fig. 17 : adding a main window to a beamline GUI with the editor

Then, adding a brick is possible :

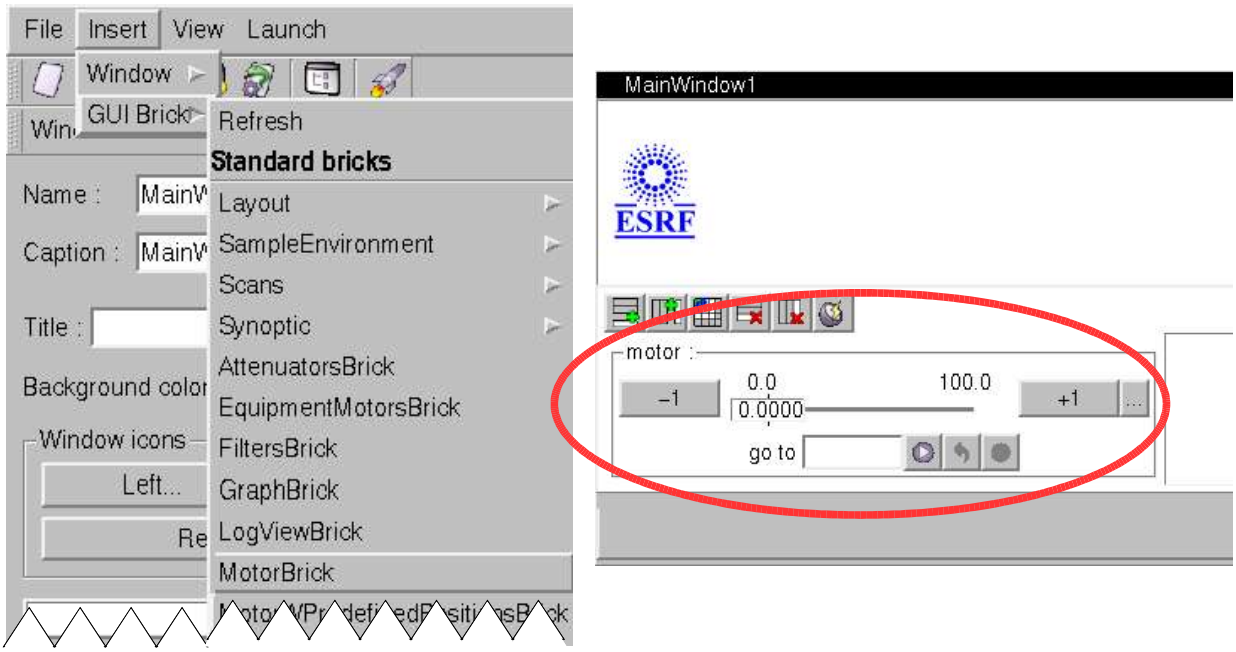


Fig. 18 : adding a motor brick in a window with the editor

From one grid cell to another, in the same window or another, bricks can be moved by drag'n'drop. User can click on a brick to access its properties ; the Property Editor appears at the bottom of the left panel inside the GUI editor window. Right-clicking on a brick gives the basic operations like removing the brick, reloading the brick, insert a row or a column on the grid, etc.

3.2.3 Connecting bricks together

The Connection Editor, integrated with the beamline GUI design tool, can connect bricks together through the signals and slots they define.

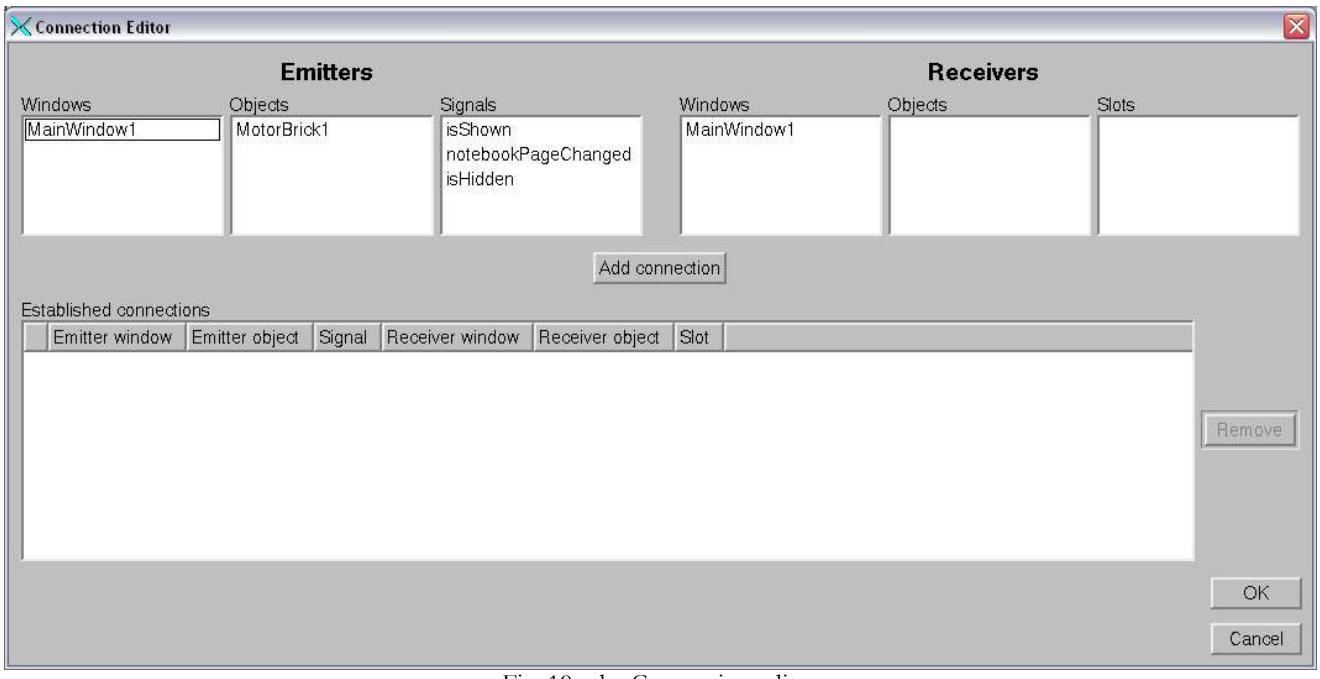


Fig. 19 : the Connection editor