INTERNAL REPORT

Developement of a framework for building tools for managing observations with a generic telescope.

Alessandro Corongiu

Report N. 37, released: 30/07/2014

Reviewer: N. D'Amico, M. Murgia

OAC Osservatorio Astronomico di Cagliari

Contents

Chapter 1

Introduction

1.1 The SEADAS project

The Sardinia Radio Telescope[1] (SRT) is a brand new observing facility located about 35 km north of Cagliari, the main city of Sardinia. It has been designed for covering a frequency range from 300 MHz up to 100 GHz, by hosting up to 13 receivers in various focal positions, and for covering a wide set of topics in radio astronomy, including radio pulsars. The telescope controlling software, called `NURAGHE`, has been developed in the Alma Common Software[2] (ACS) environment. `NURAGHE` functionalities allow the whole control of an observing session, from the antenna setup to the source tracking, from the backends setup to the data acquisition.

The peculiar requirements for a radio pulsar observation lead to the non integration of all pulsar backends in the NURAGHE framework, and triggered the necessity of a dedicated tool for managing sessions involving observations of these celestial objects. The general design of this software (Srt ExpAnded Data Acquisition System, SEADAS) has been already described in a former internal report[3] of Cagliari Observatory. The mentioned report explains the general organization of the software, with particular attention at the hierarchy of the moduli for managing the devices involved in an observation.

The complexity of a software like SEADAS suggested to organize the code by clearly separating the features that are common to any telescope from the ones specific of SRT. Such a separation has been obtained thanks to the polymorphism and inheritance concepts of the `c++` programming language and, as a matter of fact, lead to the design of `Qt` derived classes that can be used for building an application devoted to the management and control of an observing session with a radio telescope.

[1] http://www.srt.inaf.it
[2] http://www.eso.org/projects/alma/develop/acs/
[3] Internal report N.35, March 1st 2014

1.2 The programming language

The choice of the programming language is grounded on the following arguments:

1) it has to be a fully supported language, from the point of view of the maintenance and the development of the related libraries;

2) it has to easily allow the distinction between the telescopes' common and the SRT peculiar features;

3) it has to easily allow the management of events that can happen at an unpredictable time.

The `Qt4`[4] toolkit has been identified as a language that meets the three mentioned requirements. `Qt4` is well known for being a powerful `c++` based toolkit for building graphic user interfaces (GUIs). An application whose GUI is based on it is usually designed so that the core application is written in `c++`, while the GUI only is written in `Qt4`. In this specific case it has been chosen to write the entire application in `Qt4` since the main task is to handle events, and those algorithms that could in principle be written in pure `c++` play a role that does not justify such a separation.

The first requirement is met since `Qt4` is developed by a company that makes use of this toolkit for commercial applications. The second requirement is given by the fact that the `Qt4` toolkit is built on the `c++` programming language. The third requirement is met by the `Qt4` peculiar feature that allows to establish the so called `signal/slot` connection. The occurrence of an event is identified by the emission of a signal that identifies the event itself, and its connection with a slot means that a well identified function is called whenever the event happens for analyzing the situation ad taking the opportune decisions.

1.3 The framework organization

The Seadas Developing System (SDS) framework consists of several classes built upon the `Qt4` toolkit, i.e. classes that inherit from a `Qt4` class. SDS classes can be divided in four groups, accordingly to the role of the objects they provide:

1) Value classes: they provide objects for managing the value for a given parameter;

[4] http://qt.digia.com/

2) Information classes: they provide objects for storing general purposes informations;

3) Tool classes: they provide objects that play the role of useful tools for the GUI;

4) Device classes: they provide objects for managing the devices that are involved in any observation.

The following chapters of this report are devoted to the description of all SDS classes, accordingly to the mentioned classification.

1.4 To whom is this report addressed

This report is addressed to software developers that aim to build a tool for controlling a software by starting from the classes already implemented in the `SDS` framework. All `SDS` classes are here explained in their functionalities, in their structure, and detailed explanations of all properties are here provided.

Chapter 2

Value classes

2.1 Introduction

Value classes provide objects devoted for managing the value of parameter. These objects have to meet the following requirements:

1) they are designed so that it becomes very easy, inside the source code, declaring and instantiating the object and accessing to all its functions;

2) they allow the setting of the parameter via both an user's action in the GUI, and programmatically in all tasks devoted to set the value;

3) they effectively store the value and make it available, through opportune query functions, to all tasks that require the value of the parameter;

4) they allow an immediate propagation of the newly set value to other objects of the application, whenever necessary.

All mentioned requirements have been met by building a composite widget, whose graphic properties can be set by a set of dedicated slots. The main elements of the object are a label that shows in the GUI the parameter's name, and a second object, here called the *value managing object*, that allows the user to manually set the value and effectively stores it. Two sets of functions allow the programmatic setup an query of the parameter, while a set of signals allow to pass to other objects the newly set value.

The choice of the `Qt` class for the value managing object is related to the degree of freedom in choosing the value for a given parameter. The length of the observation, e.g., is a parameter whose value is subject to the only constraint of being a numeric value, while the coordinate system, e.g., is a parameter whose values have to be constrained to be few and predefined. In

cases like the first one, the best choice for the value managing object is a single line editable field, while in cases like the second one the best choice is a not editable combo box.

The mentioned parameters' classification lead to build a base class, where it has been implemented all functionalities that not depend on the constraint on the value, hence to the `Qt` class for the value managing object. All implemented functionalities have been designed so that the interaction with the value managing object is obtained by calling virtual functions.

Two classes inherit from the base class. The implementation of this classes consists in the instantiation of the value managing object, i.e. in declaring it as an object of a well precise `Qt` class, and in the consequent implementation of the virtual functions declared in the base class.

2.2 SDSValue class

2.2.1 Class overview

`SDSValue` class is the base class for all objects devoted to the managament of a parameter. It is a composite `QWidget` designed for hosting two objects: a label for displaying the parameter's name or short description, member `QLabel SDSValue::vLabel`, and the value managing object, not yet instantiated for the already mentioned reasons.

Four public slots allow to set all graphic properties. Slot `void SDSValue::setGeometry(QString psn, int px, int py, int w1, int w2, int w3)` indicates the position `psn` of the label with respect to the value managing object, integers `px` and `px` are the `SDSValue` object's upper corner's coordinates in the parent widget, while integers `w1`, `w2` and `w3` determine the dimensions and the position for each member as follows:

1) `psn = left or psn = right`: `w1` is the width for the label, `w2` is the width for the value managing object and `w3` is their common height;

2) `psn = top or psn = bottom`: `w2` is the height for the label, `w3` is the height for the value managing object and `w1` is their common width;

Slot `void SDSValue::setLabelStyle(QFrame::Shape sp,QFrame::Shadow sd)` sets the `QFrame::Shape` and `QFrame::Shadow` properties for member `vLabel`, while slot `void SDSValue::setTitle(QString ttl)` sets the text to be displayed in the label. A shortcut for setting at once all these properties, and also the parent widget, is given by slot `void SDSValue::setAll(QWidget *parent, QString psn, int px, int py, int w1, int w2, int w3, QFrame::Shape sp, QFrame::Shadow sd, QString ttl)`.

The overloaded slot `void SDSValue::setValue` allows to set the new value for the parameter. Its argument can be a `QString` character string, or an integer, a float or a double numeric value.

Four slots allow to query the current value for the parameter. Slot `virtual QString SDSValue::value()` returns the parameter value as a `QString`. It is a virtual slot since it is also the function that effectively extracts the value from the value managing object. The other three slots, `double SDSValue::dValue()`, `float SDSValue::fValue()` and `int SDSValue::iValue()` return, if meaningful in the requested format, the parameter value as a double, a float and an integer number respectively.

The overloaded signal `void SDSValue::newValue` is emitted, whenever a new value is set, for passing to other objects the newly set value. Regardless of the way the value is set and of the format, this signal is emitted for passing the new value as a `QString` and as compatible numeric formats.

Two classes inherit from `SDSValue`, namely `SDSEdit`, that provides objects for managing free value parameters, and `SDSCombo`, that provides objects for managing parameters that can assume predefined values. Their implementation is grounded on the declaration of the value managing object, an editable `QLineEdit` widget and a not editable `QComboBox` widget respectively.

2.2.2 Technical description

A `SDSValue` object is a `QWidget` object designed for hosting two child widgets, namely a label for displaying the parameter name and the object responsible for effectively managing the parameter value, the so called *value managing object*.

The label object, member `QLabel vLabel`, has already been declared and all related functionalities have already been implemented. The value managing object is instead not yet declared, since its `Qt` class depends on the kind of parameter, as explained above. All functionalities have been designed by separating the tasks that do not depend from its `Qt` class from the ones that instead depend from this aspect. The result of this approach lead to the declaration of some virtual functions that are called when the interaction with the value managing object is required, and that have been implemented in the `SDSValue` subclasses with the opportune tasks.

These virtual slots are:

1) `virtual QString SDSValue::value()`, for extracting the current value and returning it as a `QString` object;

2) `virtual void SDSValue::setNewValue(QString nv)`, for setting in the value managing object the new value;

3) `virtual void SDSValue::placeValue(int px, int py, int wdh, int hgt)`, for resizing the value managing object and placing it in the `SDSValue` widget.

The implementation for all other query and set is based on a call to `virtual QString SDSValue::value()` and `virtual void SDSValue::setNewValue(QString nv)` respectively. The two mentioned derived classes have been implemented as follows:

1) the value managing object is declared in the class header accordingly to the chosen `Qt` class for it;

2) slot `virtual QString SDSValue::value()` is accordingly implemented for extracting from the value managing object the parameter's value as a `QString`;

3) slot `virtual void SDSValue::setNewValue(QString nvl)` is accordingly implemented for setting the new value in the value managing object;

4) slot `virtual void SDSValue::placeValue(int x, int y, int w, int h)` is accordingly implemented for placing the value managing object inside the `SDSValue widget`, by calling the function that sets its width and height equal to `w` and `h` respectively, and the function that places its upper left corner at the `x` and `y` coordinates in the generic parent widget;

5) custom functionalities are implemented;

Inheritance

Inherits from: `QWidget`
Inherited by: `SDSEdit`, `SDSCombo`

Public members

`QLabel vLabel`
This object is the label devoted for displaying the parameter's name or its short description. Its properties can be set by calling slots `void SDSValue::setLabelStyle(QFrame::Shape,QFrame::Shadow)` and `void SDSValue::setTitle(QString ttl)`, or by calling the opportune `QLabel` functions.

Public slots

`double SDSValue::dValue()`
Returns the current parameter's value as a `double` numerical value, if applicable.

`float SDSValue::fValue()`
Returns the current parameter's value as a `float` numerical value, if applicable.

`int SDSValue::iValue()`
Returns the current parameter's value as an `integer` numerical value, if applicable.

`void SDSValue::setAll(QWidget *parent, QString psn, int px, int py, int w1, int w2, int w3, QFrame::Shape sp, QFrame::Shadow sd, QFont vFont, QString title)`
Shortcut slot for setting at once all geometric and style properties for the object. This function calls, in this given order:

1) `QWidget::setParent(QWidget *parent);`

2) `SDSValue::setGeometry(QString psn,int px,int py,int w1,int`

w2,int w3);

3) SDSValue::setLabelStyle(QFrame::Shape sp, QFrame::Shadow sd);

4) SDSValue::setTitle(QString title).

void SDSValue::setGeometry(QString psn, int px, int py,int w1, int w2, int w3)
This slot sets the SDSValue object's geometric properties. Argument psn indicates the position of vLabel with respect to the value managing object: top, bottom, left, right. Arguments px and py are the coordinates of the SDSValue object's top-left corner in the parent widget. Arguments' w1, w2 and w3 meaning depends on vLabel's position, namely:

1) psn = left or psn = right:
   w1 = vLabel width
   w2 = value managing object width
   w3 = vLabel and value managing object common height
   SDSValue widget width = w1 + w2
   SDSValue widget height = w3

2) psn = top or psn = bottom:
   w1 = vLabel and value managing object common width
   w2 = vLabel height
   w3 = value managing object height
   SDSValue object width = w1
   SDSValue object height = w2 + w3

Accordingly to the dimensions of the two child widgets, their position in the SDSValue widget, expressed in terms of the position of their top left corner results:

1) psn = left: vLabel at (0,0), value managing object at (w1,0)

2) psn = right: vLabel at (w2,0), value managing object at (0,0)

3) psn = top: vLabel at (0,0), value managing object at (0,w2)

4) psn = top: vLabel at (0,w3), value managing object at (0,0)

Once the dimensions and the positions for both child widgets are determined, `vLabel` is finally resized and displaced, while the resizing and displacement for the value managing object is performed by slot `virtual void SDSValue::placeValue(int, int, int, int)`, called with the opportune arguments.

`void SDSValue::setLabelStyle(QFrame::Shape sp,QFrame::Shadow sd)`
This slot sets the style properties for `vLabel` through `QFrame::Shape` and `QFrame::Shadaow` properties.

`void SDSValue::setTitle(QString title)`
This slot sets to `title` the text displayed in `vLabel`.

`void SDSValue::setValue(QString vl)`
This slot sets the parameter's value to `QString vl`, and calls protected slot `void SDSValue::valueChanged()`, responsible for emitting signals that pass the newly set values in various formats.

`void SDSValue::setValue(double dvl)`
This overloaded slot sets the parameter's value to `double dvl`.

`void SDSValue::setValue(float fvl)`
This overloaded slot sets the parameter's value to `float fvl`.

`void SDSValue::setValue(int ivl)`
This overloaded slot sets the parameter's value to `int ivl`.

Virtual public slots

`virtual QString SDSValue::value()`
This slot is implemented in SDSValue subclasses for extracting the parameter's value as a string. Its implementation depends on the value managing object's `Qt` class.

Protected slots

`void SDSValue::valueChanged()`
This slot is responsible for the emission of signals that pass the newly set value in various formats. Its called by slot `void SDSValue::setValue(QString vl)` and always emits the signal `void SDSValue::newValue(QString vl)`, while signals that pass the value in numeric formats are emitted only if the value itself can be treated as a number of that format.

Virtual protected slots

`virtual void SDSValue::setNewValue(QString newVal)`
This slot is implemented in `SDSValue` subclasses for setting in the value managing object the parameter's value. Its implementation depends on the value managing object's `Qt` class.

`virtual void SDSValue::placeValue(int px, int py, int wdh, int hgt)`
This slots is implemented in `SDSValue` subclasses for resizing the value managing object and placing it in the `SDSValue` widget. Arguments `int px` and `int py` indicate the position, in the `SDSValue` widget, of the parameter's managing object top left corner, while `int wdh` and `int hgt` are its width and height respectively. It's called by public slot `void SDSValue::setGeometry` since these properties have to be set accordingly to the `vLabel` ones.

Signals

`void SDSValue::newValue(QString)`
This signal passes the newly set value for the parameter as a string.

`void SDSValue::newValue(double)`
This signal passes the newly set value for the parameter as a `double` numeric value.

`void SDSValue::newValue(int)`
This signal passes the newly set value for the parameter as a `integer` numeric value.

`void SDSValue::newValue(float)`
This signal passes the newly set value for the parameter as a `float` numeric value.

2.3 SDSEdit class

2.3.1 Class overview

The `SDSEdit` class is the fully implemented class that provides objects for managing parameters, whose allowed values cannot be constrained to belong to a predefined set. This class has been derived from the `SDSValue` class by declaring the *value managing object* as `QLineEdit field`, by default set as editable, and by accordingly reimplementing all `SDSValue` virtual functions.

In order to take account for parameters to which it is *mandatory* to assign a value, such a functionality has been implemented. Member `bool isMandatory` stores, by means of a boolean value, whether it is mandatory to assign the parameter a value or not. Its value is set by default to `false`, which means that the parameter value can remain unset, and can be modified by calling slot `void SDSEdit::setMandatory(bool mnd)`. Slot `void SDSEdit::setMissing(bool mss)` modifies, for mandatory parameters, the background color for `QLineEdit field`, setting it red if no value is stored (a string that contains only blanks is considered an empty string) and white if a value has been given.

An `SDSEdit` object can be also used for displaying a status. In this case member `QLineEdit field` has to be set as not editable, since this use has to avoid any user editing of the stored value. Slot `void SDSEdit::setEditable(bool edt)` allows to modify this property by means of a boolean argument.

Finally, slot `void SDSEdit::clear()` has been implemented for clearing the stored parameter's value.

2.3.2 Technical description

Class `SDSEdit` is derived from class `SDSValue` by following the prescription mentioned in §2.2.2. Its implementation is based on the declaration of the value managing object as `QLineEdit field`, consists in the reimplementation of the three slots that have been declared as virtual in the `SDSValue` class, and in the established signal/slot connection between signal `void QLineEdit::editingFinished()` from member `field` and slot `void SDSValue::valueChanged`, a connection that ensures the emission of the `void SDSValue::newValue()` signals whenever the parameter value is manually entered by the user.

Custom functionalities have been implemented as follows:

1) Clearing the value: slot `void SDSEdit::clear()` is a shortcut for calling slot `void QLineEdit::clear()` for member `field`;

2) Setting the `SDSEdit` object as editable/not editable: slot `void SDSEdit::setEditable(bool)` is a shortcut for calling `void QLineEdit::setReadOnly(bool)` for member `field`;

3) Setting the mandatory property and accordingly managing the `SDSEdit` object: member `bool isMandatory` stores the property, while slot `void SDSEdit::setMandatory(bool)` sets the property, and slot `void SDSEdit::setMissing(bool)` sets the graphic properties for member `field` accordingly to the mandatory property and the newly set value;

Inheritance

Inherits from: `SDSValue`

Public members

`QLineEdit field`
This member is the value managing object. By default `field` is set as editable, and this property can be changed by calling slot `void SDSEdit::setEditable(bool edt)`.

`bool isMandatory`
This member holds the property that indicates whether a value for the managed parameter has to be always set or not. By default its value is `false`, which means that the parameter can remain unset, and this property can be changed by calling slot `void SDSEdit::setMandatory(bool mnd)`.

Public slots

`void SDSEdit::clear()`
This slot unsets the parameter, by clearing `QLineEdit field`.

`void SDSEdit::setEditable(bool edt)`
This slot sets whether the text of member `QLineEdit field` can be edited by the user, `edt = true`, or not, `edt = false`.

`void SDSEdit::setMandatory(bool mnd)`
This slot sets the mandatory property for the managed parameter, by setting the value for member `bool isMandatory`. Its argument is the value that member `bool isMandatory` has to assume.

`void SDSEdit::setMissing(bool mss)`
This slot sets the background color for member `QLineEdit field` in mandatory `SDSEdit` objects. It's called by `void SDSEdit::setNewValue(QString txt)` whenever a new value has to be set for the managed parameter. If the new value is an empty string, its called with argument `true` and the background color is set to `red`. If instead the new value is not an empty string, it's called with argument `false` and the background color is set to `white`.

`QString SDSEdit::value()`, reimplemented from
`virtual QString SDSValue::value()`
This slot returns the current parameter's value, in the `QString format`.

Protected slots

`void SDSEdit::placeValue(int px,int py,int wdt, int hgt)`, reimplemented from
`virtual void SDSValue::placeValue(int px,int py,int wdt, int hgt)`
This slot places the top left corner of member `QLineEdit field` at coordinates `(px,py)`, and sets its width and height to `wdt` and `hgt` respectively. Its implementation calls `void QWidget::move(int px, int py)` and `void QWidget::resize(int wdt, int hgt)`.

`void SDSEdit::setNewValue(QString txt)`, reimplemented from
`virtual void SDSValue::setNewValue(QString txt)`
This slot is responsible for effectively setting the new value for the managed parameter, by setting the text in `QLineEdit field` equal to the content of `QString txt`. If the managed parameter is mandatory, this slot also calls `void SDSEdit::setMissing(bool mss)` with the opportune argument. A sequence of blanks characters is considered an empty string.

Internal signal/slot connections

Signal `void QLineEdit::editingFinished()`, emitted by member `QLineEdit field`
to slot `void SDSValue::valueChanged()`
This connection ensures that slot `void SDSValue::valueChanged()` is called when the user has finished to manually set the new parameter's value by typing it in `QLineEdit field`.

2.4 SDSCombo class

2.4.1 Class overview

The `SDSCombo` class is the fully implemented class that provides objects for managing parameters, whose allowed values belong to a predefined set. This class has been derived from the `SDSValue` class by declaring the value managing object as `QComboBox field`, set by default as not editable, and by accordingly reimplementing all `SDSValue` virtual functions.

The allowed parameter's values are loaded into the `QComboBox field` by slot `void SDSCombo::loadItems(QString datasource)`, whose argument is the text file that contains all predefined values. This file has to be indicated with its absolute path.

Since the selected item for a `QComboBox` object can be set and queried through the index the item is assigned in the items list, some functionalities have been implemented for allowing to manage the parameter with respect to this peculiar feature. Slot `void SDSCombo::setIndex(int idx)` allows to set the item whose index is the value stored in `int idx`, while slot `int SDSCombo::index()` returns the index of the currently selected item, and signal `void SDSCombo::newIndex(int)`, emitted whenever the selection in `QComboBox field` changes, allows to pass to other objects the index of the newly set item.

2.4.2 Technical description

Class `SDSCombo` is derived from class `SDSValue` by following the prescription mentioned in §2.2.2. Its implementation is based on the declaration of the value managing object as `QComboBox field`, consists in the reimplementation of the three slots that have been declared as virtual in the `SDSValue` class, and in the established signal/slot connection between signal `void QComboBox::currentIndexChanged(int)` from member `field` and slot `void SDSCombo::indexChanged(int)`, a connection that ensures the emission of the `void SDSValue::newValue()` signals whenever the parameter value is manually entered by the user.

Custom functionalities have been implemented as follows:

1) Loading the list of allowed values: slot `void SDSCombo::loadItems(QString items list)` reads the values from a text file and accordingly builds the `QComboBox field` items list;

2) Value management by using the items list index: slot `int SDSCombo::index()` is a shortcut for calling `int QComboBox::currentIndex()` for member field, slot `void SDSCombo::setIndex(int)` is a shortcut for calling `void QComboBox::setCurrentIndex(int)` for member field, slot `void SDSCombo::indexChanged(int)` ensures that `SDSValue` signals are emitted and also emits the `void SDSCombo::newIndex(int)`;

Inheritance

Inherits from: `SDSValue`

Public members

`QComboBox field`
This member is the value managing object. By default it is set as not editable, and the predefined values for the managed parameter are loaded by calling slot `void SDSCombo::loadItems(QString datasource)`.

Public slots

`int SDSCombo::index()`
This slot returns the index of the current item for `QComboBox field`.

`void SDSCombo::loadItems(QString datasource)`
This slot loads into the `QComboBox field` items list the allowed values for the managed parameter. Its argument is the text file containing the values to be loaded into the items list, and has to be indicated with its full path. Items have to be organized in a single column.

`void SDSCombo::setIndex(int idx)`
This slot selects in `QComboBox field` the item, whose index in the items list is indicated by `int idx`.

`QString SDSCombo::value()`, reimplemented from
`virtual QString SDSValue::value()`
This slot returns a string containing the current item text, in the `QString` format.

Protected slots

`void SDSCombo::indexChanged(int idx)`
This slot is responsible for the emission of signals for passing the index of the newly set item and its content. It is triggered by the signal `void QComboBox::currentIndexChanged(int)`, from `QComboBox field`, thanks to a signal/slot connection. It emits the signal `void SDSCombo::newIndex(int)` and calls slot `void SDSValue::valueChanged()`.

`void SDSCombo::placeValue(int px,int py,int wdt, int hgt)`, reimplemented from
`virtual void SDSValue::placeValue(int px,int py,int wdt, int hgt)`
This slot places the top left corner of member `QComboBox field` at the coordinates `(px,py)`, and sets its width and height to `wdt` and `hgt` respectively.

`void SDSCombo::setNewValue(QString txt)`, reimplemented from

`virtual void SDSValue::setNewValue(QString txt)`
This slot selects in `QComboBox field` the item that corresponds to the string stored in `QString txt`.

Signals

`void SDSCombo::newIndex(int)`
This signal passes the index of the newly set item in `QComboBox field`. It is by default called by slot `void SDSCombo::indexChanged(int idx)`.

Internal signal/slot connections

Signal `void QComboBOx::currentIndexChanged(int)`, emitted by member `QComboBox field` to slot `void SDSCombo::indexChanged(int)`
This connection ensures that slot `void SDSCombo::indexChanged(int)` is called whenever the selected item changes in `QComboBox field`.

Chapter 3

Information classes

3.1 Introduction

Data files contain a set of scientific, technical and management informations which are not necessary to the backends for their signal processing, but are required for the data real time and/or offline processing, for the reconstruction of the observation history and for an easy organization of the observations database.

These informations are usually related to the scientific project, to the source and the receiver's parameters. In a modular application they are set in the object that manages the observing session and the antenna, and they have to be transfered somehow to the ones devoted to the backends control, so that they can be stored in the header of the data file.

In order to easily transfer these informations, classes have been designed for organizing them in well defined arguments and providing objects that allow an organized management and transferring for this kind of informations.

The identified arguments are:

1) the observing session;

2) the source to be observed;

3) the receiver and its setup;

4) the telescope's basic parameters.

Informations in these classes are stored in dedicated `QString` members without setting any predefined format. Classes that manage these informations are `SDSSession` for the observing session, `SDSSource` for the target source, `SDSReceiver` for the receiver to be used, and `SDSReceiver` for the telescope.

3.2 SDSReceiver class

3.2.1 Class overview

`SDSReceiver` class provides objects that store all informations about the receiver. The informations that can be stored are:

1) the receiver name;
2) the string that identifies the the receiver's band;
3) the lower limit for the frequency band;
4) the upper limit for the frequency band;
5) the number of active beams, which has to be set equal to one for single beam receivers;
6) the reference beam, which has to be set equal to one for single beam receivers;
7) the detected polarization, linear or circular.

3.2.2 Technical description

A `SDSReceiver` object is a `QObject` whose members are protected `QString` variables, each of them can be set and queried by calling dedicated slots. All informations can be set at once by calling slot `void SDSReceiver::setReceiver(SDSReceiver *rec)`, whose argument is a reference to an object of this class.

Inheritance

Inherits from: `QObject`

Protected members

`QString band`
This object stores the frequency band identification string. Its value is set by slot `void SDSReceiver::setBand(QString str)` and queried by

slot `QString SDSReceiver::bandID()`.

`QString fmin`
This object stores the value for the lower limit of the receiver's frequency band. Its value is set by slot `void SDSReceiver::setLowerFrequency(QString str)` and queried by slot `QString SDSReceiver::lowerFrequency()`.

`QString fmax`
This object stores the value for the upper limit of the receiver's frequency band. Its value is set by slot `void SDSReceiver::setUpperFrequency(QString str)` and queried by slot `QString SDSReceiver::upperFrequency()`.

`QString name`
This object stores the receiver's name. Its value is set by slot `void SDSReceiver::setReceiverName(QString str)` and queried by slot `QString SDSReceiver::receiverName()`.

`QString nBeams`
This object stores the number of the receiver's active beams. Its value is set by slot `void SDSReceiver::setNumberOfBeams(QString str)` and queried by slot `QString SDSReceiver::numberOfBeams()`. For single beam receivers it has to be set to `1`.

`QString polType`
This object stores the receiver's detected polarization, namely circular or linear. Its value is set by slot `void SDSReceiver::setPolarization(QString str)` and queried by slot `QString SDSReceiver::polarization()`.

`QString refBeam`
This object stores the number that identifies the beam adopted as reference for the telescope pointing. Its value is set by slot `void SDSReceiver::setReferenceBeam(QString str)` and queried by slot `QString SDSReceiver::referenceBeam()`. For single beam receivers it has to be set to `1`.

Public slots

`QString SDSReceiver::bandID()`
This slot returns the string that identifies the receiver's frequency band, as stored in member `QString band`.

`QString SDSReceiver::numberOfBeams()`
This slot returns the number of the receiver's active beams, as stored in member `QString nBeams`.

`QString SDSReceiver::lowerFrequency()`
This slot returns the lower limit of the receiver's frequency band, as stored in member `QString fmin`.

`QString SDSReceiver::polarization()`
This slot returns the receiver's detected polarization, as stored in member `QString polType`.

`QString SDSReceiver::receiverName()`
This slot returns the receiver's name, as stored in member `QString name`.

`QString SDSReceiver::referenceBeam()`
This slot returns the number that identifies the beam used as reference for the telescope pointing, as stored in member `QString refBeam`.

`void SDSReceiver::setBand(QString bnd)`
This slot sets to `bnd` the value for member `QString band`.

`void SDSReceiver::setNumberOfBeams(QString nbm)`
This slot sets to `nbm` the value for member `QString nBeams`.

`void SDSReceiver::setLowerFrequency(QString lfq)`
This slot sets to `lfq` the value for member `QString fmin`.

`void SDSReceiver::setPolarization(QString lpz)`

This slot sets to `plz` the value for member `QString polType`.

`void SDSReceiver::setReceiver(SDSReceiver *rcv)`
This slot sets all receiver's informations, which are stored in the `SDSReceiver` object pointed by `SDSReceiver *rcv`.

`void SDSReceiver::setReceiverName(QString nm)`
This slot sets to `nm` the value for member `QString name`.

`void SDSReceiver::setReferenceBeam(QString rbm)`
This slot sets to `rbm` the value for member `QString refBeam`.

`void SDSReceiver::setUpperFrequency(QString ufq)`
This slot sets to `ufq` the value for member `QString fmax`.

`QString SDSReceiver::upperFrequency()`
This slot returns the upper limit of the receiver's frequency band, as stored in member `QString fmax`.

3.3 SDSSession class

3.3.1 Class overview

`SDSSsession` class provides objects that store the essential informations about the observing session. The informations that can be stored are:

1) the project title;

2) the project code, usually but not necessarily assigned by the telescope *Time Allocation Committee*;

3) the name(s) of the observer(s).

3.3.2 Technical description

A `SDSSession` object is a `QObject` whose members are protected `QString` variables, each of them can be set and queried by calling dedicated slots. All informations can be set at once by calling slot `void SDSSession::setSession(SDSSession *session)`, whose argument is a

reference to an object of this class.

Inheritance

Inherits from: `QObject`

Protected members

`QString ProjectName`
This member stores the title of the scientific project. Its value is set by slot `void SDSSession::setProjectName(QString str)` and queried by slot `QString SDSSession::projectName()`.

`QString ProjectCode`
This member stores the project code, usually assigned by the telescope *Time Allocation Committee*. Its value is set by slot `void SDSSession::setProjectCode(QString str)` and queried by slot `QString SDSSession::projectCode()`.

`QString ObserverName`
This member stores the name(s) of the observer(s). Its value is set by slot `void SDSSession::setObserverName(QString str)` and queried by slot `QString SDSSession::observerName()`.

Public slots

`QString SDSSession::observerName()`
This slot returns the observer(s) name(s), as stored in member `QString ObserverName`.

`QString SDSSession::projectCode()`
This slot returns the project's identification code, as stored in member `QString ProjectCode`.

`QString SDSSession::projectName()`

This slot returns the project's name, as stored in member `QString ProjectName`.

`void SDSSession::setObserverName(QString on)`
This slot sets to `on` the value for member `QString ObserverName`.

`void SDSSession::setProjectCode(QString pc)`
This slot sets to `pc` the value for member `QString ProjectCode`.

`void SDSSession::setProjectName(QString pn)`
This slot sets to `pcn` the value for member `QString ProjectName`.

`void SDSSession::setSession(SDSSession *session)`
This slot sets all session's informations, which are stored in the `SDSSession` object pointed by `SDSSession *session`.

3.4 SDSSource class

3.4.1 Class overview

`SDSSource` class provides objects that store the essential informations about the source to be observed. The informations that can be stored are:

1) the source name;

2) the coordinate system through which the source coordinates are expressed;

3) the value for the source longitude in the selected coordinate system;

4) the value for the source latitude in the selected coordinate system;

3.4.2 Technical description

A `SDSSource` object is a `QObject` whose members are protected `QString` variables, each of them can be set and queried by calling dedicated slots. All informations can be set at once by calling slot `void SDSSource::setSource(SDSSource *source)`, whose argument is a reference to an object of this class.

Inheritance

Inherits from: `QObject`

Protected members

`QString CoordSys`
This member stores the coordinate system through which the source coordinates are expressed. Its value is set by slot `QString SDSSource::setCoordSys()` and queried by slot `void SDSSource::coordSys(QString cs)`.

`QString Latitude`
This member stores the value for the latitude coordinate in the selected coordinate system. Its value is set by slot `void SDSSource::setLatitude(QString lt)` and queried by slot `QString SDSSource::latitude()`.

`QString Longitude`
This member stores the value for the longitude coordinate in the selected coordinate system. Its value is set by slot `void SDSSource::setLongitude(QString lg)` and queried by slot `QString SDSSource::longitude()`.

`QString Name`
This member stores the source name. Its value is set by slot `void SDSSource::setName(QString nm)` and queried by slot `QString SDSSource::name()`.

Public slots

`QString SDSSource::coordSys()`
This slot returns the coordinate system through which the source's longitude and latitude are expressed, as stored in member `QString CoordSys`.

`QString SDSSource::latitude()`
This slot returns the source's latitude in the selected coordinate system, as stored in member `QString Latitude`.

`QString SDSSource::longitude()`
This slot returns the source's longitude in the selected coordinate system, as stored in member `QString Longitude`.

`QString SDSSource::name()`
This slot returns the source's name, as stored in member `QString Name`.

`void SDSSource::setCoordSys(QString cs)`
This slot sets to `cs` the value for member `QString CoordSys`.

`void SDSSource::setLatitude(QString lt)`
This slot sets to `lt` the value for member `QString Latitude`.

`void SDSSource::setLongitude(QString lg)`
This slot sets to `lg` the value for member `QString Longitude`.

`void SDSSource::setName(QString nm)`
This slot sets to `nm` the value for member `QString Name`.

`void SDSSource::setSource(SDSSource *source)`
This slot sets all source's informations, which are stored in the `SDSSource` object pointed by `SDSSource *source`.

3.5 SDSTelescope class

3.5.1 Class overview

`SDSTelescope` class provides objects that store the essential informations about the telescope. The informations that can be stored are:

1) the telescope name;

2) the code that identifies the telescope in data analysis softwares;

3) the short code that identifies the telescope in data analysis softwares;

4) the geographic longitude of the telescope's site;

5) the geographic latitude of the telescope's site;

6) the geographic altitude of the telescope's site;

3.5.2 Technical description

A `SDSTelescope` object is a `QObject` whose members are protected `QString` variables, each of them can be set and queried by calling dedicated slots. All informations can be set at once by calling slot `void SDSTelescope::setTelescope(SDSTelescope *telescope)`, whose argument is a reference to an object of this class.

Inheritance

Inherits from: `QObject`

Protected members

`QString Altitude`
This member stores the value for the geographic altitude of the telescope site. Its value is set by slot `QString SDSSource::setAltitude()` and queried by slot `QString SDSSource::altitude()`.

`QString Code`
This member stores the code that identifies the telescope in the data analysis softwares. Its value is set by slot `QString SDSTelescope::setCode()` and queried by slot `QString SDSTelescope::code()`.

`QString Latitude`
This member stores the value for the geographic latitude of the telescope's site. Its

value is set by slot `QString SDSTelescope::setLatitude()` and queried by slot `QString SDSTelescope::latitude()`.

`QString Longitude`
This member stores the value for the geographic longitude of the telescope's site. Its value is set by slot `QString SDSTelescope::setLongitude()` and queried by slot `QString SDSTelescope::longitude()`.

`QString Name`
This member stores the telescope name. Its value is set by slot `QString SDSTelescope::setName()` and queried by slot `QString SDSTelescope::name()`.

`QString ShortCode`
This member stores the short code that identifies the telescope in the data analysis softwares. Its value is set by slot `QString SDSTelescope::setShortCode()` and queried by slot `QString SDSTelescope::shortCode()`.

Public slots

`QString SDSTelescope::altitude()`
This slot returns the geographic altitude of the telescope site, as stored in member `QString Altitude`.

`QString SDSTelescope::code()`
This slot returns the telescope's code, as stored in member `QString Name`.

`QString SDSTelescope::latitude()`
This slot returns the geographic latitude of the telescope site, as stored in member `QString Latitude`.

`QString SDSTelescope::longitude()`
This slot returns the geographic longitude of the telescope site, as stored in member `QString Longitude`.

`QString SDSTelescope::name()`
This slot returns the telescope's name, as stored in member `QString Name`.

`QString SDSTelescope::shortCode()`
This slot returns the telescope's short code, as stored in member `QString ShortCode`.

`void SDSTelescope::setAltitude(QString lt)`
This slot sets to `at` the value for member `QString Altitude`.

`void SDSTelescope::setCode(QString cd)`
This slot sets to `cd` the value for member `QString Code`.

`void SDSTelescope::setLatitude(QString lt)`
This slot sets to `lt` the value for member `QString Latitude`.

`void SDSTelescope::setLongitude(QString lg)`
This slot sets to `lg` the value for member `QString Longitude`.

`void SDSTelescope::setName(QString nm)`
This slot sets to `nm` the value for member `QString Name`.

`void SDSTelescope::setShortCode(QString scd)`
This slot sets to `scd` the value for member `QString ShortCode`.

`void SDSTelescope::setTelescope(SDSTelescope *telescope)`
This slot sets all telescope's informations, which are stored in the `SDSTelescope` object pointed by `SDSTelescope *telescope`.

Chapter 4

Tools classes

4.1 Introduction

Tools classes provide objects that are not strictly required for running an observation, but may be helpful for the devices and session management, for the observation setup and monitoring.

A first tool is a button devoted to both enable and disable a feature, in a very general sense, and to display the feature status by changing its graphic properties. This object is provided by class `SDSStatusButton`.

A second tool is a frame that allows the user the interaction with a text by mouse button clicks. Several tools require this kind of interaction, namely the source catalogue (mouse clicks allow the selection of the source), the schedule manager (mouse clicks allow the selection of the schedule lines to be effectively done) and the observation list (mouse clicks allow the rearrangement of the selected schedule lines). For this reason a base class, `SDSTextEdit`, has been designed by implementing the base functionalities that allow text−mouse interactions, and three classes have been developed, whose core is a `SDSTextEdit` object: `SDSCatalogue` for the source catalogue, `SDSScheduleManager` for the management of the schedule and `SDSObservationList` for the management of the observation list.

Finally, the key role played by system messages lead to the design of a tool for both displaying these messages and storing them in a file. Class `SDSLog` provides objects that ensure these functionalities.

4.2 SDSStatusButton class

4.2.1 Class overview

Class `SDSStatusButton` provides a widget that acts both as a button for both enabling and disabling a feature, in a very general sense, and as a label that displays the feature status.

The feature status is indicated by the background color, `red` if deactivated and `green` if activated, and by opportune texts that illustrate the current status. The feature status is also stored by means of a boolean variable, whose values `false` and `true` respectively indicate the deactivated and activated status.

4.2.2 Technical description

An `SDSStatusButton` object is a `QLabel`, whose button properties result by setting its focus policy to `Qt::ClickFocus`, and by implementing slot `void QMouseEvent::mouseReleaseEvent(QMouseEvent *event)` so that the signal `void SDSStatusButton::activate(bool)` is emitted for passing the boolean value indicating the requested status. Member `bool isEnabled` stores the feature current status, hence the boolean value passed by signal `void SDSStatusButton::activate(bool)` is always `!isEnabled`. Two `QString` objects, `QString offTxt` and `QString onTxt`, store the texts to be displayed when the feature is respectively deactivated and activated. Slot `void SDSStatusButton::setTexts(QString it, QString ot)` allows to set their content at once. Public slot `void SDSStatusButton::setEnabled(bool bl)` changes the object's graphic properties accordingly to the feature status. It is `NOT` automatically called by slot `mouseReleaseEvent(QMouseEvent *event)`: this setting is intentional since the feature activation and deactivation may require a task that may not be successfully accomplished. The connection between the feature and the related `SDSStatusButton` object has to be implemented as follows:

1) In the object, whose feature is managed by a `SDSStatusButton` object, a slot has been implemented for triggering both the deactivation and activation tasks. This slot has to take a boolean argument, and has to be implemented so that the deactivation task is called if the argument is `false` and the activation one if the argument is `true`.

2) A signal/slot connection has to be established so that the above mentioned slot is called whenever the signal `void SDSStatusButton::setEnabled(bool bl)` is emitted.

3) The object, whose feature is managed by a `SDSStatusButton` object, has to emit a signal whenever the feature status change. This signal has to pass the boolean value `false` if the new status is deactivated, `true` if activated.

4) A second signal/slot connection has to be established between the object's above mentioned signal and the slot `void SDSStatusButton::setEnabled(bool bl)`, so that the `SDSStatusButton` object's graphic properties change accordingly to the new status.

The dimensions and position in the parent widget for a `SDSStatusButton` object can be set at once by calling slot `SDSStatusButton::setGeometry(int px, int py, int wdh, int hgt)`, whose arguments are, in the indicated order, the `x` and `y` coordinate of its top left corner in the parent widget, its width and height.

Inheritance

Inherits from: `QLabel`

Public members

`bool isEnabled`
This member stores the feature status by mean of a boolean value, namely `false` if the feature is deactivated, `true` if the feature is activated.

`QString offTxt`
This member stores the text to be displayed when the feature is deactivated. Its value can be set by calling slot `void SDSStatusButton::setTexts(QString it, QString ot)` (second argument).

`QString onTxt`
This member stores the text to be displayed when the feature is activated. Its value can be set by calling slot `void SDSStatusButton::setTexts(QString it, QString ot)` (first argument).

Protected members

`QPalette Pal`
This member stores the color palette for the background color to be displayed accordingly to the feature status.

Public slots

`void SDSStatusButton::setEnabled(bool bl)`
This slot sets the background color, the displayed text and the graphic properties for the `SDSStatusButton` accordingly to the status to be displayed. If the related feature is activated the argument has to be the boolean value `true`. In this case the displayed text is the string stored in member `QString onTxt`, the background color is set to `Qt::green` and the `QFrame::Shadow` property is set to `QFrame::Sunken`. If instead the related feature is deactivated the argument has to be the boolean value `false`. In this case the displayed text is the string stored in member `QString offTxt`, the background color is set to `Qt::red` and the `QFrame::Shadow` property is set to `QFrame::Raised`.

`void SDSStatusButton::setTexts(QString it, QString ot)`
This slot sets the strings stored in members `onTxt` (first argument) and `offTxt` (second argument).

`void SDSStatusButton::setGeometry(int px, int py, int wdh, int hgt)`
This slot places the `SDSStatusButton` widget's top left corner at coordinates `(px,py)` in the parent widget, and sets its width and height to `wdh` and `hgt` respectively.

Protected slots

`void SDSStatusButton::mouseReleaseEvent(QMouseEvent *event)`, reimplemented from `void QMouseEvent::mouseReleaseEvent(QMouseEvent *event)`
This slot is responsible for giving the button behaviour to the `SDSStatusButton` object. It determines if it has been left-clicked by checking for two conditions to be simultaneously met, namely the mouse left button has been clicked and the `SDSStatusButton` object has focus.

If so the signal `void SDSStatusButton::activate(bool)` is emitted, and the passed value is always `!isEnabled`.

Signals

`void SDSStatusButton::activate(bool)`
This signal passes the boolean value that corresponds to the requested action, namely it passes `false` if a deactivation is requested, `true` if instead an activation is requested.

4.3 SDSTextEdit class

4.3.1 Class overview

Class `SDSTextEdit` provides the core objects for building tools that allow mouse−text interaction. Mouse buttons to which an action can be assigned are the left and the mid button. The `enum SDSTextEdit::action` lists the possible actions:

0) `SDSTextEdit::NoAction`: no action is assigned to the mouse button;

1) `SDSTextEdit::CutLine`: the whole line under the mouse pointer is removed from the displayed text, the text below the cut line is shifted upwards and a signal is emitted for passing a string containing the cut text;

2) `SDSTextEdit::SelectLine`: the whole line under the mouse pointer is selected, and a signal is emitted for passing a string containing the selected text;

3) `SDSTextEdit::PasteLine`: a previously selected or cut line is inserted at the mouse pointer position; if there is some text at that position, that text is shifted downwards;

4) `SDSTextEdit::CustomAction1`: an user's custom action;

5) `SDSTextEdit::CustomAction2`: a second user's custom action;

An action is assigned to a mouse button by setting the value for members `action leftAct` and `action midAct`, that store the action assigned to the left

and middle button respectively. Their value is set by calling public slot `void SDSTextEdit::setActions(SDSTextEdit::action la, SDSTextEdit::action ma)`, whose first and second argument are the values to be assigned to `action leftAct` and `action midAct` respectively.

4.3.2 Technical description

A `SDSTextEdit` object is a read only `QPlainTextEdit` object upon which functionalities have been implemented for allowing the mouse−text interaction. Three default actions have been already implemented, and the possibility has been also given of implementing custom actions. The possible actions to be assigned to the left and middle mouse button are listed in the `enum SDSTextEdit::action` structure and are stored in members `action leftAct` and `action midAct` respectively, whose default value is `SDSTextEdit::NoAction`.

Slot `void SDSTextEdit::mouseReleaseEvent(QMouseEvent *event)`, reimplemented from `void Qwidget::mouseReleaseEvent(QMouseEvent *event)`, is the core of the mouse−text interaction. It reads the button that has been clicked, then the action assigned to the clicked button, and accordingly calls the slot that performs the assigned action. Slots `void SDSTextEdit::cutLine(QMouseEvent *event)`, `void SDSTextEdit::selectLine()` and `void SDSTextEdit::pasteLine(QMouseEvent *event)` have been fully implemented for respectively cutting, selecting the clicked line and inserting at the mouse pointer position a previously cut or selected line. Whenever a line is selected or cut, the signal `void SDSTextEdit::newSelection(QString str)` is emitted for passing to other objects a string containing the text line. Virtual slots `virtual void SDSTextEdit::customAction1()` and `virtual void SDSTextEdit::customAction2()` are called if the assigned action respectively is `SDSTextEdit::CustomAction1` `SDSTextEdit::CustomAction2`. They have to be reimplemented in a `SDSTextEdit` subclass if custom actions on text are required.

Private member `QString selection` stores the content of the last cut or selected line, and `QTextCursor *tc` is a pointer to the text cursor of the displayed document.

Inheritance

Inherits from: `QPlainTextEdit`

Properties

`enum SDSTextEdit::action {NoAction, CutLine, SelectLine, PasteLine, CustomAction1, CustomAction2}`
This property enumerates the possible actions to be assigned to a mouse button:

0) `SDSTextEdit::NoAction`: no action is assigned to the mouse button;

1) `SDSTextEdit::CutLine`: the whole line under the mouse pointer is removed from the displayed text, the text below the cut line is shifted upwards and a signal is emitted for passing a string containing the cut text;

2) `SDSTextEdit::SelectLine`: the whole line under the mouse pointer is selected, and a signal is emitted for passing a string containing the selected text;

3) `SDSTextEdit::PasteLine`: a previously selected or cut line is inserted at the mouse pointer position; if there is some text at that position, that text is shifted downwards;

4) `SDSTextEdit::CustomAction1`: an user's custom action;

5) `SDSTextEdit::CustomAction2`: a second user's custom action;

Public members

`action leftAct`
This member stores the action assigned to the mouse left button.

`action midAct`
This member stores the action assigned to the mouse middle button.

Protected members

`QString selection`
This member stores the content of the last cut or selected line.

`QTextCursor *tc`
This member is a pointer to the text cursor of the displayed text.

Public slots

`void SDSTextEdit::setActions(action la, action ma)`
This slot assigns the actions to both the mouse left (first argument) and middle (second argument) button, by setting the related values for members `action leftAct` and `action midAct`.

Protected slots

`void SDSTextEdit::cutLine(QMouseEvent *event)`
This slot removes from text the line under the mouse pointer and stores it in member `QString selection`. If the cut line is not the last one, all text below the cut line is shifted upwards. Signal `void SDSTextEdit::newSelection(QString)` is emitted for passing the content of the cut line.

`void SDSTextEdit::mouseReleaseEvent(QMouseEvent *event)`, reimplemented from `void QMouseEvent::mouseReleaseEvent(QMouseEvent *event)`
This slot determines which button has been clicked and calls the slot that performs the associated action.

`void SDSTextEdit::pasteLine(QMouseEvent *event)`
This slot inserts at the mouse pointer position a previously selected or cut line. If some text is already present at the pointer position, the whole text from that line to the end is shifted downwards. The inserted text is the string stored in `QString selection`.

`void SDSTextEdit::selectLine()`
This slot selects the line under the mouse pointer position, stores its content in `QString selection` and emits the signal `void SDSTextEdit::newSelection(QString)` for passing the content of the selected line.

Virtual protected slots

`virtual void SDSTextEdit::customAction1()`
This slot has to be reimplemented in `SDSTextEdit` subclasses for performing a custom action on the text. The `enum SDSTextEdit::action` to be assigned to the mouse button devoted for triggering this action is `SDSTextEdit::CustomAction1`.

`virtual void SDSTextEdit::customAction2()`
This slot has to be reimplemented in `SDSTextEdit` subclasses for performing a second custom action on the text. The `enum SDSTextEdit::action` to be assigned to the mouse button devoted for triggering this action is `SDSTextEdit::CustomAction2`.

Signals

`void SDSTextEdit::newSelection(QString)`
This signal passes the content of a cut or selected line. In `SDSTextEdit` subclasses it can be used for passing any string anyhow related to the clicked text.

4.4 SDSCatalogue class

4.4.1 Class overview

Class `SDSCatalogue` provides objects for managing the catalogue of sources. Its core object is `SDSTextEdit *catalogue`, set so that a source can be selected by left clicking on the related displayed line. Once a source is selected, signal `void SDSCatalogue::newSourceParameters(QString)` is emitted for passing a string that contains all informations required by the observation. In its displayed version, the catalogue is organized

by lines, each of them dedicated to a single source. Source informations can be programmatically requested when the user manually inserts the source name in the GUI, or when setup files or schedule lines are read. Slot `virtual QString SDSCatalogue::findSourceCoords(QString sourceName)` has to be implemented in a `SDSCatalogue` subclass for extracting source informations in such situations.

The informations required for the observation may depend on the coordinate system. Member `QString CoordSys` is devoted to store this parameter, and its value can be set by the overloaded slot `void SDSCatalogue::setCoordSys`, whose argument can be a `QString` whose content is the coordinate system name (`J2000` or `Galactic`), or an integer value (`1 = J2000` or `2 = Galactic`).

A second widget, `QPlainTextEdit *parametersBar`, is devoted for hosting the description for the displayed source parameters. Its content is set by slot `void SDSCatalogue::setParametersNames(QString)`.

`SDSCatalogue` widgets are designed for being hosted in a dedicated window, that can be resized so that catalogue line are entirely visible without any horizontal scrolling. What remains fixed is the height of `QPlainTextEdit *parametersBar`, that is set by slot `void SDSCatalogue::setParametersBarHeight(int)`.

The text font and color palette have to be explicitly set. Slot `void SDSCatalogue::setStyle(QFont cf, QPalette cp)` allows this setting.

4.4.2 Technical description

An `SDSCatalogue` object is a `QWidget` that hosts two child widgets: `SDSTextEdit *catalogue`, devoted for displaying the source catalogue, and `QPlainTextEdit *parametersBar`, devoted for hosting a description for the displayed source parameters.

The text displayed in member `SDSTextEdit *catalogue` has to be organized in rows, each of them containing informations related to a single source, and stored in a text file. Slot `void SDSCatalogue::loadCatalogue(QString catFile)`, whose argument is the mentioned text file including its full path, puts the content of such *catalogue file* in `SDSTextEdit *catalogue`.

The extraction of the source informations depends on the catalogue organization and on the context in which they are requested. For this reason two virtual slots have been declared. These slots have to be implemented in a `SDSCatalogue` subclass for extracting all necessary informations and returning them in a single string, but differ in the content of their arguments, accordingly to the different situations in which these informations are requested. Slot `virtual QString SDSCatalogue::findSourceCoords(QString sourceName)` is meant

for being used when the source name is read in a setup file, in a schedule line or is manually inserted by the user: its argument has to be the source name. Slot `virtual QString SDSCatalogue::getSourceParameters(QString catalogueLine)` is called whenever a source is selected by left clicking on the related line displayed in `SDSTextEdit *catalogue`. The source parameters that are displayed in `SDSTextEdit *catalogue`, may not be the ones necessary for the observation, but have been chosen so that it's easier for the user the selection of the source to be observed. This slot has hence to be implemented for extracting all necessary informations starting by the source catalogue line, as displayed in `SDSTextEdit *catalogue`.

The signal/slot connection between signal `void SDSTExtEdit::newSelection(QString)`, emitted by `SDSTextEdit *catalogue`, and protected slot `void SDSCatalogue::newSourceSelected(QString)` ensures that lines selected by a left click are processed and signal `void SDSTExtEdit::newSourceParameters(QString)` is emitted for passing to other objects a string containing the extracted informations.

`SDSCatalogue` objects are designed for being hosted in a dedicated window that can be resized for an easy reading of the catalogue lines. Slot `void SDSCatalogue::resizeEvent(QResizeEvent *event)`, reimplemented from `virtual void QWidget::resizeEvent(QResizeEvent *event)`, resizes `SDSTextEdit *catalogue` and `QPlainTextEdit *parametersBar` so that they fit in width in the hosting window, and `SDSTextEdit *catalogue` fills the space between `QPlainTextEdit *parametersBar` and the bottom of the window.

Inheritance

Inherits from: `QWidget`

Protected members

`SDSTextEdit *catalogue`
This member is the frame where the source catalogue is displayed. The text to be displayed has to be stored in a text file that is loaded by calling slot `void SDSCatalogue::loadCatalogue(QString)`.

`QString CoordSys`
This member stores the coordinate system in which source coordinate have to be extracted from the catalogue. It can be set by calling the overloaded slot `void SDSCatalogue::setCoordSys`.

`QPlainTextEdit *parametersBar`
This member is the frame where the parameters' description is displayed. Its content is set by calling slot `void SDSCatalogue::setParametersNames(QString)`.

Public slots

`void SDSCatalogue::loadCatalogue(QString ct)`
This slot reads the text file that contains the source lines to be displayed in `SDSTextEdit *catalogue`. The file has to be indicated with its absolute path.

`void SDSCatalogue::setCoordSys(QString cs)`
This slot sets the coordinate system through which the source coordinates have to be expressed. Its allowed values are `J2000` and `Galactic`.

`void SDSCatalogue::setCoordSys(int csi)`
This slot sets the coordinate system through which the source coordinates have to be expressed, by using integer arguments. Its allowed values are `1 = J2000` and `2 = Galactic`.

`void SDSCatalogue::setParametersBarHeight(int py)`
This slot sets the height for member `QPlainTextEdit *parametersBar`, vertically displaces `SDSTextEdit *catalogue` just below `QPlainTextEdit *parametersBar` and adjusts its height so that it fits in the hosting window.

`void SDSCatalogue::setParametersNames(QString txt)`
This slot displays in `QPlainTextEdit *parametersBar` the description for the source parameters displayed in `SDSTextEdit *catalogue`.

`void SDSCatalogue::setStyle(QFont cFont, QPalette cPal)`
This slot sets the text font and the color palette for the `SDSCatalogue` widget and its child

objects.

Virtual public slots

`virtual QString SDSCatalogue::findSourceCoords(QString sourceName)`
This slot has to be reimplemented in `SDSCatalogue` subclasses for extracting the source parameters by using as input the source name.

`virtual QString SDSCatalogue::getSourceParameters(QString)`
This slot has to be reimplemented in `SDSCatalogue` subclasses for extracting the source parameters by using as input the source line displayed in `SDSTextEdit *catalogue`.

Protected slots

`void SDSCatalogue::newSourceSelected(QString)`
This slot extracts from the catalogue all source informations that are necessary for the observation, and emits the signal `void SDSCatalogue::newSourceParameters(QString)` for passing a string containing the extracted informations.

`void SDSCatalogue::resizeEvent(QResizeEvent *event)`
This slot readjusts the width for both `QPlainTextEdit *parametersBar` and `SDSTextEdit *catalogue` so that they fit in width in the hosting window, and the height for the latter so that it fits in height in the space between `QPlainTextEdit *parametersBar` and the bottom of the window.

Signals

`void SDSCatalogue::newSourceParameters(QString)`
This signal passes the string that contains the value for those parameters that are required for the observation.

Internal signal/slot connections

Signal `void SDSTextEdit::newSelection(QString)`, emitted by member `SDSTextEdit *catalogue`
to slot `void SDSCatalogue::newSourceSelected(QString)`
This connection triggers the extraction of the source parameters whenever a left click is done on a source line displayed in `SDSTextEdit *catalogue`. The passed argument is the clicked line.

4.5 SDSScheduleManager class

4.5.1 Class overview

Class `SDSScheduleManager` provides the object for managing the schedule file, which is meant as a collection of observations that can be done in any order. Its core object is `SDSTextEdit *schedFrame`, for which the mouse left button has been assigned the action `SDSTextEdit::SelectLine`. A schedule line is selected by left clicking on it, and signal `void SDSTextEdit::obsLineSelected(QString)` is emitted for passing the selection, typically to an `SDSObservationList` object (see §4.6). If it's necessary to select all schedule lines at once, the button named `Select all`,member `QPushButton SelectAll`, has to be clicked.

A default directory for the schedules can be set by calling slot `void SDSScheduleManager::setScheduleDir(QString sd)`. A schedule line is loaded by clicking the button labelled `Load sched`, member `QPushButton LoadSchedule`: a system dialog window pops up for browsing the system directories and selecting the file, and the name of the selected schedule is displayed in the field at the right of the `Load sched` button. A schedule can also be loaded by typing its name in the mentioned field. If the selected schedule is located in the default directory its name only has to be given, if it's located in a subdirectory it's name has to be given with its relative path with respect to the default directory, and if it's located elsewhere its name has to be indicated with its absolute path. If no default directory is set the schedule name must be always indicated with its absolute path.

A loaded schedule can be edited by the user. The `Start edit` labelled button, member `QPushButton StartEdit`, sets `SDSTextEdit *schedFrame` as editable, while the `End edit` labelled button, member `QPushButton EndEdit`, sets it as not editable. If button `End edit` is

clicked but there are unsaved changes, a system dialog window appears for saving schedule's changes. An edited schedule can be saved with its current name by clicking on button `Save` (member `SaveSched`), or with a different name by clicking on button `Save as...`, member `SaveSchedAs`: also in this case a system dialog window appears for saving the schedule in the opportune directory and/or with the opportune name, and the new schedule name is displayed. Changes since the last saving can be cleared by clicking on button `Clear changes`, member `ClearChanges`.

Class `SDSScheduleManager` has been designed so that an object of this class can be hosted in a dedicated window. Slot `void SDSScheduleManager::setStyle(QFont smFont,QPalette smPal)` allows in this case to set the text font and the color palette.

4.5.2 Technical description

The core object of a `SDSScheduleManager` widget is `SDSTextEdit *schedFrame`: its actions are set to `SDSTextEdit::SelectLine` and `SDSTextEdit::NoAction` for the left and middle mouse button respectively. Its signal `void SDSTextEdit newSelection(QString)` is connected to slot `void SDSScheduleManager::sendObsLine(QString)`, which in turn emits the signal `void SDSScheduleManager::obsLineSelected(QString)` only if `SDSTextEdit *schedFrame` is in read only mode.

The selection of the entire schedule is performed by slot `void SDSScheduleManager::selectAll()`, connected to the signal `void QPushButton::clicked()` emitted by member `QPushButton SelectAll`: the whole displayed text is selected and signal `void SDSScheduleManager::obsLineSelected(QString)` is emitted with the mentioned selection as argument.

Member `SDSEdit SchedField`, placed without its label at the right side of `QPushButton LoadSchedule`, allows the user to manually insert the schedule name. Its signal `void SDSEdit::newValue(QString)` is connected to the slot `void SDSScheduleManager::readSchedule(QString)`, which displays the content of the schedule file in `SDSTextEdit *schedFrame`. Signal `void QPushButton::clicked()`, emitted by `QPushButton LoadSchedule`, is connected to slot `void SDSScheduleManager::loadSchedule()`. This slot opens a dialog window for browsing the system directories and selecting the schedule file: if a file is selected its name is displayed in `SDSEdit SchedField`.

Slots that perform the actions related to the schedule editing are connected to signal `void QPushButton::clicked()` emitted by the related `QPushButton` object.

Slot `void SDSScheduleManager::startEdit()`, called by clicking `QPushButton StartEdit`, sets as editable the text displayed in `SDSTextEdit *schedFrame`. Slot `void SDSScheduleManager::saveSched()`, called by clicking `QPushButton SaveSched`, builds the absolute path of the file whose name is displayed in `SDSEdit SchedField` and writes the displayed text in it, while slot `void SDSScheduleManager::saveSchedAs()`, called by clicking `QPushButton SaveSchedAs`, pops up a system dialog window and saves the displayed text in a file whose name and location are the selected ones. Slot `void SDSScheduleManager::clearChanges()`, called by clicking `QPushButton ClearChanges`, clears `SDSTextEdit *schedFrame` and loads the last saved version of the file whose name is displayed in `SDSEdit SchedField`. Slot `void SDSScheduleManager::endEdit()`, called by clicking `QPushButton EndEdit`, sets `SDSTextEdit *schedFrame` in read only mode. If there are unsaved changes, this slot calls `void SDSScheduleManager::saveSchedAs()` for saving the unsaved changes.

`SDSScheduleManager` objects are designed for being hosted in a separate window. Slot `void SDSScheduleManager::resizeEvent(QResizeEvent *event)`, reimplemented from `virtual void QWidget::resizeEvent(QResizeEvent *event)`, accordingly readjusts the positions and dimensions of all child widgets, whenever the hosting window is resized.

Inheritance

Inherits from: `QWidget`.

Public Members

`SDSEdit SchedField`
This member acts as the filed for maunally inserting the schedule name and displaying it. Once a file name is inserted, the file is automatically read and displayed in `SDSTextEdit *schedFrame`. If no default directory has been set, the schedule name has to be inserted with its absolute path. If instead a default directory has been set, the file name only has to be given if it's located in the default directory, it has to be indicated with its relative path with respect to the default directory, if it is located in a subdirectory of the default one, and with its absolute path if located elsewhere.

Protected Members

`QPushButton ClearChanges`
This button clears all schedule changes since the last saving by calling slot `void SDSScheduleManager::clearChanges()`.

`QPushButton EndEdit`
This button terminates the editing mode for `SDSTextEdit *schedFrame` by calling slot `void SDSScheduleManager::endEdit()`.

`QPushButton LoadSchedule`
This button opens a system dialog window, for selecting the schedule to be loaded, by calling slot `void SDSScheduleManager::loadSchedule()`.

`QPushButton SaveSched`
This button saves the displayed text in a file, whose name is displayed in `SDSEdit SchedField`, by calling slot `void SDSScheduleManager::saveSched()`.

`QPushButton SaveSched`
This button opens a system dialog window, for selecting the name and directory of the file into the displayed text has to be saved, by calling slot `void SDSScheduleManager::saveSchedAs()`.

`SDSTextEdit *schedFrame`
This member is the `SDSTextEdit` widget devoted to displaying the loaded schedule. Its mouse button actions are set to `SDSTextEdit::SelectLine` and `SDSTextEdit::NoAction` for the left and middle mouse button respectively.

`QString scheduleDir`
This member stores the full path of the default directory for the schedule files. It is set by slot `void SDSScheduleManager::setScheduleDir(QString sd)`.

`QPushButton SelectAll`
This button selects the entire displayed text for being transfered to the observation list by calling slot `void SDSScheduleManager::selectAll()`.

`QPushButton StartEdit`
This button enables the editing of the displayed schedule by calling slot `void SDSScheduleManager::startEdit()`.

Public slots

`void SDSScheduleManager::setScheduleDir(QString sd)`
This slot sets the default directory for the schedule files, by storing its value in member `QString scheduleDir`.

`void SDSScheduleManager::setStyle(QFont smFont,QPalette smPal)`
This slot sets the text font and the color palette for the `SDSScheduleManager` object.

Protected slots

`void SDSScheduleManager::clearChanges()`
This slot clears the text displayed in `SDSTextEdit *schedFrame` and loads the content of the file whose name is displayed in `SDSEdit SchedField`.

`void SDSScheduleManager::endEdit()`
This slot closes the editing mode. If there are unsaved changes, it calls `void SDSScheduleManager::saveSchedAs()` for allowing the user to save the schedule displayed version.

`void SDSScheduleManager::loadSchedule()`
This slot opens a system dialog window for selecting the schedule file to be loaded. If a file is selected, its name is displayed in `SDSEdit SchedField`.

`void SDSScheduleManager::readSchedule(QString schName)`
This slot reads the file whose name is displayed in `SDSEdit SchedField`, after rebuilding its absolute path accordingly to the rules for entering a schedule file name.

`void SDSScheduleManager::resizeEvent(QResizeEvent *event)`
This slot readjusts the dimensions and positions of all child widgets whenever the hosting window is resized. `QPushButton LoadSchedule` and `QPushButton SelectAll` remain fixed in their position and dimensions while `SDSEdit SchedField`, located at the right of `QPushButton LoadSchedule`, is resized in width so that it horizontally fits from `QPushButton LoadSchedule` to the window's right side. `QPushButton ClearChanges`, `QPushButton EndEdit`, `QPushButton SaveSched`, `QPushButton SaveSchedAs` and `QPushButton StartEdit` remain fixed in their dimensions and horizontal positions, while they are vertically displaced so that they remain at the bottom side of the window. `SDSTextEdit *schedFrame` is resized in width and height so that it fits the space in between all other widgets.

`void SDSScheduleManager::saveSched()`
This slot saves the displayed text in a file whose name is displayed in `SDSEdit SchedField`, after rebuilding its absolute path accordingly to the rules for entering a schedule file name.

`void SDSScheduleManager::saveSchedAs()`
This slot opens a system dialog window for selecting the directory and the name for the file in which the displayed text has to be saved. The name of the new file is displayed in `SDSEdit SchedField` accordingly to the rules for entering a schedule file name.

`void SDSScheduleManager::selectAll()`
This slot emits the signal `void SDSScheduleManager::obsLineSelected(QString)` by giving it as argument the entire text displayed in `SDSTextEdit *schedFrame`.

`void SDSScheduleManager::sendObsLine(QString ol)`
This slot emits the signal `void SDSScheduleManager::obsLineSelected(QString)` by giving it as argument the schedule line that has been left clicked.

`void SDSScheduleManager::startEdit()`
This slot sets as editable the text displayed in `SDSTextEdit *schedFrame`.

Signals

`void SDSScheduleManager::obsLineSelected(QString)`
This signal is emitted when a selection is made on the displayed text. It is emitted by slots `void SDSScheduleManager::selectAll()` and `void SDSScheduleManager::sendObsLine(QString ol)`.

Established signal/slot connections

Signal `void QPushButton::clicked()`, emitted by member `QPushButton ClearChanges`
to slot `void SDSScheduleManager::clearChanges()`
This connection allows the call to slot `void SDSScheduleManager::clearChanges()` whenever `QPushButton ClearChanges` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton EndEdit`
to slot `void SDSScheduleManager::endEdit()`
This connection allows the call to slot `void SDSScheduleManager::endEdit()` whenever `QPushButton EndEdit` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton LoadSchedule`
to slot `void SDSScheduleManager::loadSchedule()`
This connection allows the call to slot `void SDSScheduleManager::loadSchedule()` whenever `QPushButton LoadSchedule` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton SaveSched`
to slot `void SDSScheduleManager::saveSched()`
This connection allows the call to slot `void SDSScheduleManager::saveSched()` whenever `QPushButton SaveSched` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton SaveSchedAs`
to slot `void SDSScheduleManager::saveSchedAs()`
This connection allows the call to slot `void SDSScheduleManager::saveSchedAs()` whenever `QPushButton SaveSchedAs` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton SelectAll`
to slot `void SDSScheduleManager::selectAll()`
This connection allows the call to slot `void SDSScheduleManager::selectAll()` whenever `QPushButton SelectAll` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton StartEdit`
to slot `void SDSScheduleManager::startEdit()`
This connection allows the call to slot `void SDSScheduleManager::startEdit()` whenever `QPushButton StartEdit` is clicked.

Signal `void SDSEdit::newValue(QString)`, emitted by member `SDSEdit SchedField`
to slot `void SDSScheduleManager::readSchedule(QString)`
This connection allows the call to slot `void SDSScheduleManager::readSchedule(QString)` whenever a new text is entered in `SDSEdit SchedField`.

Signal `void SDSTextEdit::newSelection(QString)`, emitted by member `SDSTextEdit *schedFrame`
to slot `void SDSScheduleManager::sendObsLine(QString)`
This connection ensures that slot `void SDSScheduleManager::sendObsLine(QString)` is called whenever a schedule line is selected by left clicking on it.

4.6 SDSObservationList class

4.6.1 Class overview

Class `SDSObservationList` provides the objects devoted for managing the *observation list*, i.e. the list of the observations that have to be performed. Observation lines are loaded by selecting them in the `SDSScheduleManager` window (see §4.5), and can be rearranged at any time before and during the observing session. An observation line can be cut by clicking on it with the mouse left button, and pasted by clicking at the desidered position with a mouse middle click. The last cut line can be pasted multiple times. The button labelled `Clear` allows to clear all displayed lines.

4.6.2 Technical description

A `SDSObservationList` is a composite `QWidget`. Its core is member `SDSTextEdit *obsFrame`, devoted to displaying the observation lines, and whose mouse button actions are set to `SDSTextEdit::CutLine` and `SDSTextEdit::PasteLine` respectively.

Two slots allow to add an observation line in `SDSTextEdit *obsFrame`. Slot `void SDSObservationList::addObsLine(QString ol)` is devoted for appending a line at the end of the displayed text, hence it's meant for being called when a line is selected in the `SDSScheduleManager` window. Slot `void SDSObservationList::restoreLine(QString obsLine)` is devoted for inserting a line at the beginning of the displayed text, hence it's meant for being called if the observation is interrupted.

Slot `QString SDSObservationList::getObsLine()` extracts the first displayed line and returns it, hence it's meant to be called at the start of each planned observation for retrieving all necessary informations.

Member
`QPushButton clear`, the `Clear` labelled button, calls slot `void QPlainTextEdit::clear()` for clearing the text displayed in `SDSTextEdit *obsFrame`.

`SDSObservationList` widgets are designed for being hosted in a separate window. Slot `void SDSObservationList::setStyle(QFont olFont,QPalette olPal)` allows to set at once the widget's text font and the color palette. Slot `void SDSObservationList::resizeEvent(QResizeEvent *event)`, reimplemented from `virtual void QWidget::resizeEvent(QResizeEvent *event)`, resizes `SDSTextEdit *obsFrame` so that it always fits in the hosting window.

Inheritance

Inherits from: `QWidget`

Protected members

`QPushButton clear`
This button clears the text displayed in `SDSTextEdit *obsFrame`, by calling slot `void QPlainTextEdit::clear()`.

`SDSTextEdit *obsFrame`
This member is the `SDSTextEdit` object devoted to displaying the observations' lines. Its mouse actions are set to `SDSTextEdit::CutLine` and `SDSTextEdit::PasteLine` for the left and middle button respectively.

Public slots

`void SDSObservationList::addObsLine(QString ol)`
This slot appends an observation line at the end of the displayed text.

`QString SDSObservationList::getObsLine()`
This slot extract the first line displayed in `SDSTextEdit *obsFrame` and returns it. The extracted line is removed from `SDSTextEdit *obsFrame`.

`void SDSObservationList::restoreLine(QString obsLine)`
This slot inserts its argument string at the beginning of the text displayed in `SDSTextEdit *obsFrame`.

`void SDSObservationList::setStyle(QFont olFont,QPalette olPal)`
This slot sets at once the text font and the color palette for the `SDSObservationList` object.

Protected slots

`void SDSObservationList::resizeEvent(QResizeEvent *event)`, reimplemented from `virtual void QWidget::resizeEvent(QResizeEvent *event)`
This slot resizes `SDSTextEdit *obsFrame` so that it always fits in the hosting window.

Established signal/slots connections

Signal `void QPushButton::clicked()`, emitted by member `QPushButton clear`
to slot `void QPlainTextEdit::clear()`, for member `SDSTextEdit *obsFrame`
This connection allows the clearing of the text displayed in `SDSTextEdit *obsFrame` when `QPushButton clear` is clicked.

4.7 SDSLog class

4.7.1 Class overview

Class `SDSLog` provides objects for managing system messages, by both displaying them in a dedicated frame and saving them in a file. All messages are reformatted as shown here below, before being both displayed and saved in the log file.

```
DEV 14:34:52 This example shows how a long message is splitted in shorter
             parts. Messages splitting makes them much more easily readable,
             expecially if the substrings' maximum length is tuned so that
             they are shorter than the messages' display frame width.
```

The first line contains the three characters code that identifies the device that generated the message (see §5.2), followed by the UTC at which it has been received by the `SDSLog` object. The device code has to be set in the `SDSLog` object by calling slot `void SDSLog::setDeviceID(QString cld)`. The message is automatically split in sections whose maximum number of characters is set by slot `void SDSLog::setMaxLineLength(int mxl)`. A message is added to the log by the overloaded slot `void SDSLog::addToLog`. Its base implementation is `void SDSLog::addToLog(QString org, QString logEntry, QColor tCol)`, whose arguments are the device code, the log message and the text color to use in the `SDSLog` object. The default text color is `Qt::black` and the default device code is the one set by slot `void SDSLog::setDeviceID(QString cld)`. In the form `void SDSLog::addToLog(QString logEntry)`, the default value for both the text color and the device code are used, while in the form `void SDSLog::addToLog(QString logEntry,QColor tCol)` the default value for the device code only is used.

User annotations can be also put in the log file by typing them in the field at the bottom of the `SDSLog` widget. The string `User note` is prepended to these annotation, and they are displayed by using the default device code and using the `QT::green` color.

The signal `void SDSLog::newLogEntry(QString,QString,QColor)` passes all log messages but the user annotations, so that they can be also displayed in other `SDSLog` objects and saved in the related log file.

The log file is named as follows:

```
[device code]_YYYYMM.log
```

where `[device code]` is the three character code for the related device, as set by slot `void SDSLog::setDeviceID(QString cld)`, and `YYYY` and `MM` are the current year and month, in four and two digit format respectively. The log file is opened in `QIODevice::Append` mode by calling slot `void SDSLog::openFileInDir(QString lDir)`, whose argument is the full path to the directory the file has to be put.

Figure 4.1 displays the widget organization. At its top finds place a label for introducing the object as a whole, in the middle the frame where log messages are displayed and at the bottom an `SDSEdit` object for inserting the user's annotations.

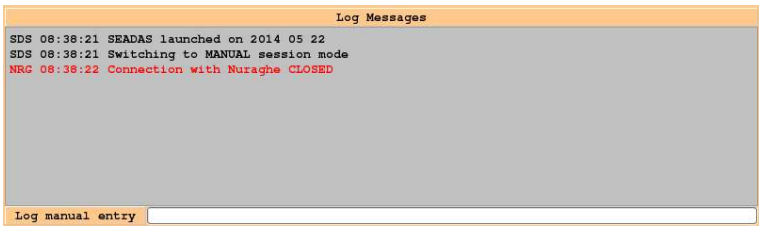

Figure 4.1: `SDSLog` widget example.

Slot `void SDSLog::setGeometry(int px, int py, int wdt, int hgt, int mlw, int txh)` sets the `SDSLog` object position in the parent widget and its dimensions, and all dimensions and positions for all child widget as follows:

a) `px` and `py` are the coordinates fir the `SDSLog` object's top left corner in the parent widget;

b) `int wdt` is the width for the `SDSLog` object;

c) `int hgt` is the height for the `SDSLog` object;

d) `int mlw` is the width for the `SDSEdit` object's label;

e) `int txh` is the height for the upper label and the `SDSEdit` object;

f) All other dimensions and positions are derived from these parameters so that all child widgets fit in the `SDSLog` object as displayed in figure 4.1.

Slot `void SDSLog::setTitles(QString title, QString mtitle)` sets the texts to be displayed in the upper and lower label respectively. `SDSLog` objects are meant for being child widgets, hence they hereditate the text font and color palette from their parent. If they are placed in a dedicated window, these properties can be set by calling slot `void SDSLog::setStyle(QFont lf, QPalette lp)`. All graphic properties can be set at once by calling slot `void SDSLog::setAllGraphics(QWidget *parent, int px, int py, int wdt, int hgt, int mlw, int txh, QPalette lPal,QFont lFont,QString title, QString mtitle)`, a shortcut for calling `void QWidget::setParent(QWidget *parent)` and all other mentioned slots for setting the graphic properties.

4.7.2 Technical description

A `SDSLog` object is a composite widget, whose child objects are `QTextEdit messageDisplay`, the object that dislplays all messages, `QLabel mainLabel`, the label at the top of the widget for hosting the object's title, and `SDSEdit manualEntry`, devoted for entering the user's annotations. The graphic organization for the `SDSLog` object is shown in figure 4.1.

Member `QString devID` stores the device three character code, member `QFile logFile` manages the file where all messages are saved, and member `int maxlen` stores the maximum length, expressed in terms of the number of character, for each line displayed in `QTextEdit messageDisplay`.

Protected slot `void SDSLog::lineToLog()` is devoted for inserting in the log history any user annotation. It is called whenever the return key is pressed while `SDSEdit manualEntry` has focus, thanks to its connection to signal `void QLineEdit::returnPressed()`, emitted by member `QLineEdit field` of `SDSEdit manualEntry`. Slot `void SDSLog::lineToLog()` checks if the string displayed in `SDSEdit manualEntry` is an empty string or not. A sequence of blank characters only is considered empty. If such string is not empty, slot `void SDSLog::addToLog(QString logEntry,QColor tCol)` is called by setting the text color to `Qt::green`.

Inheritance

Inherits from: `QWidget`

Protected members

`QString devID`
This member stores the three characters string that identifies the device. It's content is set by slot `void SDSLog::setDeviceID(QString cld)`.

`QFile logFile`
This object manages the `ASCII` file where all log messages are saved. Its name is by default:

`[device code]_YYYYMM.log`

where `[device code]` is the string stored in `QString devID`, and `YYYY` and `MM` are the year and the month, in four and two digit format respectively. The directory where log files are created and opened is set by slot `void SDSLog::openFileInDir(QString cld)`.

`QLabel mainLabel`
This member is the label at the top of the `SDSLog` widget, and is devoted to host a title for the log frame. The displayed text is set by slot `void SDSLog::setTitles(QString title, QString mtitle)` to `QString title`. Its position is fixed, and its dimensions are set by slot `void SDSLog::setGeometry(int px, int py, int wdt, int hgt, int mlw, int txh)`, while its `QFrame Shape` and `QFrame Shadow` properties have the default values of `QFrame::Panel` and `QFrame::Raised`.

`SDSEdit manualEntry`
This member provides the object for adding users' annotations to the log history. Once the annotation is entered, it is displayed with a green color and saved in the log file by pressing the return key. Only non blank strings can be added to the log; a sequence of blank characters is considered empty. Its default position is at the bottom of the `SDSLog` object, and its dimensions are set by slot `void SDSLog::setGeometry(int px, int py, int wdt, int hgt, int mlw, int txh)`, while the text displayed in its label is set by slot `void`

`SDSLog::setTitles(QString title, QString mtitle)`) to `QString mtitle`.

`int maxlen`
This member stores the maximum length, expressed in number of charactes, for each log message. Its value is set by slot `void SDSLog::setMaxLineLength(int mxl)`. If a message is longer than this value, it's split sections.

`QTextEdit messageDisplay`
This member is the frame where log messages are in real time displayed. By default it is located in the `SDSLog` object between `QLabel mainLabel` and `SDSEdit manualEntry`. Its dimensions are set by slot `void SDSLog::setGeometry(int px, int py, int wdt, int hgt, int mlw, int txh)`, while its text font and color palette are set by slot `void SDSLog::setStyle(QFont lf, QPalette lp)`.

Public slots

`void SDSLog::addToLog(QString org, QString logEntry, QColor tCol)`
This slot displays the string `QString logEntry` in `QTextEdit messageDisplay`, saves it in file `QFile logFile` and emits the signal `void SDSLog::newLogEntry(QString, QString, QColor)` for passing to other objects the newly added message. Its first argument is `QString org`, i.e. the three character code of the device that originated the message, while the third is the text color to use in `QTextEdit messageDisplay`. Their default values respectively are the device code the `SDSLog` object is member and `Qt::black`. This slot also reformats the log message accordingly to the value set in member `int maxlen` for the maximum length of lines. The device code an the message `UT` are prepended to the message and if a string splitting is necessary, all lines are indented as in this example:

```
DEV 14:34:52 This example shows how a long message is splitted in shorter
             parts. Messages splitting makes them much more easily readable,
             expecially if the substrings' maximum length is tuned so that
             they are shorter than the messages' display frame width.
```

`void SDSLog::addToLog(QString logEntry, QColor tCol)`
This overloaded slot is a shortcut for calling `void SDSLog::addToLog(QString org, QString logEntry, QColor tCol)` by using as device code the one of the device the `SDSLog` object is member.

`void SDSLog::addToLog(QString logEntry)`
This overloaded slot is a shortcut for calling `void SDSLog::addToLog(QString org, QString logEntry, QColor tCol)` by using as device code the one of the device the `SDSLog` object is member, and the text color `Qt::black`.

`void SDSLog::openFileInDir(QString lDir)`
This slot opens `QFile logFile` in the directory `lDir`, indicated with its absolute path.

`void SDSLog::setAllGraphics(QWidget *parent, int px, int py, int wdt, int hgt, int mlw, int txh, QPalette lPal, QFont lFont, QString title, QString mtitle)`
This slot is a shortcut for:
1) `void QWidget::setParent(QWidget *parent)`
2) `void SDSLog::setGeometry(int px, int py, int wdt, int hgt, int mlw, int txh)`
3) `void SDSLog::setStyle(QPalette lPal, QFont lFont)`
4) `void SDSLog::setTitles(QString title, QString mtitle)`

`void SDSLog::setDeviceID(QString cld)`
This slot sets in member `QString devID` the device three character code. If its argument is longer than three characters, only the first three are considered.

`void SDSLog::setGeometry(int px, int py, int wdt, int hgt, int mlw, int txh)`
This slot sets the position in the parent widget for the `SDSLog` object, and the positions and dimensions for all its members. The upper left corner of the `SDSLog` object is placed at coordinates `int px` and `int py` in the parent widget, `int wdt` is the common width for the `SDSLog` object and members `QLabel mainLabel` and `QTextEdit messageDisplay`, `int hgt` is the `SDSLog` object's height, `int mlw` is the width for the label of `SDSEdit manualEntry`, and `int txh` is the common height for `QLabel mainLabel` and `SDSEdit manualEntry`. The top left corner of `QLabel mainLabel` is placed at coordinates `x=0, y=0` in the `SDSLog` widget, the

top left corner of `SDSEdit manualEntry` is placed at `x=0, y=hgt-txh` and the top left corner of `QTextEdit messageDisplay` is placed at `x=0, y=txh`. All other dimensions are determined so that all objects fit in the `SDSLog` widget as indicated in figure 4.1.

`void SDSLog::setMaxLineLength(int mxl)`
This slot sets the value stored in member `maxlen`, i.e. the maximum length in characters for the messages' lines.

`void SDSLog::setStyle(QFont lf, QPalette lp)`
This slot sets, for the `SDSLog` widget and its child members, the text font to `QFont lf` and the color palette to `QPalette lp`

`void SDSLog::setTitles(QString title, QString mtitle)`
This slot sets the texts displayed in `QLabel mainLabel` and in `QLabel SDSEdit::vLabel` respectively to `QString title` and `QString mtitle`.

Protected slots

`void SDSLog::lineToLog()`
This slot inserts an user's annotation in the log history, by both displaying it in `QTextEdit messageDisplay` and saving it in `QFile logFile`.

Signals

`void SDSLog::newLogEntry(QString,QString,QColor)`
This signal passes the log message, unless it is an user annotation, jointly with the code for the device that originated the message and the used text color.

Established signal/slot connections

Signal `void QLineEdit::returnPressed()`, emitted by member `QLineEdit field` of `SDSEdit manualEntry`
to slot `void SDSLog::lineToLog()`
This connection ensures that the annotation typed in `SDSEdit manualEntry` is saved in the log history when the user presses the return key.

Chapter 5

Device classes

5.1 Introduction

The concept of *device* changed several times during the design of the classes illustrated in this chapter. The starting point has been the request of an application capable of managing the parallel data acquisition by several backends. This request lead to the design of a base class, to be later reimplemented in subclasses for obtaining objects capable of managing specific backends. In order to design the base class, it has been necessary to figure out how a generic backend can be considered. It resulted that it can be seen as composed of two parts, namely:

1) the **instrument**: it's the backend's *working core*, whose role is to process the incoming signal by sampling and digitalizing it, and to perform the online data processing, if required;

2) the **control server**: it's a computer devoted for running the backend's *control software*, i.e. the software that effectively configures the instrument, starts, stops and checks the progress and status of the data acquisition, and takes the opportune decisions in case of problems; in any backend the *control software* can also accept network connections from a remote computer that runs the application that manages the whole observation;

An antenna is a *device* that is totally different from a backend, since it's totally different its **main task**, i.e. the task that the antenna continuously performs during the observation, namely the tracking of the observed source. What is also true is that the antenna is controlled by a dedicated server that runs the antenna's *control software*, whose description exactly matches

the one above for the backend's *control software* if one reads *antenna* instead of *backend* and *source tracking* instead of *data acquisition*.

This simple consideration lead to the following definition, namely a *device* is a generic part of a telescope that in turn can be considered composed of two parts:

1) the **instrument**: it's the device *working core*, whose role is to continuously perform during the observation the device **main task**;

2) the **control server**: it's a computer devoted for running the device *control software*, i.e. the software that effectively configures the device, starts, stops and checks the progress and status of the main task, and takes the opportune decisions in case of problems; in any device the *control software* can also accept network connections from a remote computer that runs the application that manages the whole observation;

The mentioned device concept lead to design the `SDSDevice` class, i.e. the base class for managing a device, whose subclasses are `SDSBackend` and `SDSAntenna`, the base classes for building objects devoted to control the backends and the antenna respectively.

A device procedure may require a sequence of commands somehow displaced in time. If the function responsible for sending these commands is run in the application's main thread, the application itself freezes until the requested procedure ends. This issue can be solved if sequences of commands to a device are managed by a parallel thread. For this reason class `SDSCommThread` has been designed for providing threads that manage the communications between a device and its managing object in the user's application.

Once these base classes have been implemented, what remained to do was to design and implement the class that provides the object for managing an observation or an observing session. It has been soon realized that this class could be a subclass of `SDSDevice`. Infact, if one considers a device whose instrument is the whole telescope, it turns out that the telescope's main task is the **observation**, and that the telescope is configured and driven by a control server that runs a control software, which in turn is nothing but the application the observer uses for managing its observing session. This lead to the implementation of class `SDSManager` as a `SDSDevice` subclass that immediately resulted fully implemented, since all telescope−dependent tasks and commands are implemented in `SDSBackend` and `SDSAntenna` subclasses.

5.2 SDSDevice class

5.2.1 Class overview

`SDSDevice` is the base class for building objects devoted to control a generic device. A set of functionalities has already been implemented which can be recognised as common to any device, without knowing a priori their kind. These basic functionalities are:

1) the device control has to be enabled/disabled in the observer application accordingly to the observations' requirements;

2) any device has to be set up before the observation, i.e. a set of scientific and technical parameters have to be set;

3) the device main task has to be started at the beginning of the observation and stopped at its end;

4) device messages may arrive at any time, and they have to be processed in such a way that simultaneous messages from different device can be simultaneously collected and processed;

5) device messages must be in real time displayed and stored in a file;

6) the device status has to be stored and displayed in its controlling object;

7) it may be necessary to send manual commands;

8) a secondary window may be useful for displaying those informations that do not need to be displayed in the `SDSDevice` main widget;

Every functionality can be activated both manually by the user and programmatically during the observation. The only exceptions are clearly given by the reception and processing of messages from the device, and by the displaying of the device status.

The control of a device can be manually enabled and disabled by clicking `SDSStatusButton deviceAct`, also devoted to displaying the enabling status, or programmatically by calling slot `virtual void SDSDevice::enableDevice(bool)`, whose boolean argument is `true` if it's requested the enabling, or `false` in case of disabling. The enabling status is also stored by means of a boolean value in member `bool isEnabled`, whose values mean `true = enabled` and `false = disabled`.

Parameters values for the device setup can be manually entered by acting in the related `SDSValue` objects, but can be also read in a file: the `setup file`. `SDSEdit SetupFile` provides a field to manually enter the name of the setup file. A directory can be set as the default location for the setup files, by calling slot `void SDSDevice::setSetupDir(QString sd)`. If no default directory is set, the name of the setup file has to be indicated with its full path. If instead a default directory is set, the name of the setup file can be given accordingly to its position with respect to the default directory, namely:

1) the requested file is in the default directory: its name only can be specified;

2) the requested file is in a subdirectory of the default one: its name has to be given with its relative path with respect to the default directory;

3) the requested file is elsewhere: its name has to be given with its absolute path.

Once the name of a setup file is inserted in `SDSEdit SetupFile`, it's automatically loaded. `QPushButton LoadSetup` opens a system dialog window for browsing the system directories and selecting a setup file. When a file is selected in this way, its name is displayed in `SDSEdit SetupFile` by using the same conventions indicated above. If necessary, the already loaded file can be reloaded by clicking `QPushButton ReloadSetup`.

The device main task can be manually started and stopped by clicking `QPushButton Start` and `QPushButton Stop`, or programmatically by calling slots `virtual void SDSDevice::start()` and `virtual void SDSDevice::stop()`.

All communications with the device control server are performed by member `SDSCommThread *commThread`, i.e. the communication thread (§5.3). Any command, or any sequence of commands separated by a semicolon ";", have to be only composed in the main thread, then they have to be passed to the communication thread so that the application does not freeze while sending to the device several commands and waiting for the device response. All given commands are saved in the device log file. All incoming messages are collected by the communication thread before being passed to the main thread for being analyzed, and this ensures that simultaneous messages from different devices can be immediately collected.

The secondary window `QWidget auxiliaryWindow` is devoted to displaying those informations that do not need to be immediately available, hence can be opened when necessary by clicking `QPushButton auxiliaryWindowShow` and closed at any time.

Log messages are managed by `SDSLog deviceLog` (see § 4.7), while the device status is displayed by `SDSEdit Status`.

Manual commands can be typed in `SDSEdit ManualCommand`, and sent by pressing the return key.

Devices have to be identified by a three characters code, stored by `QString deviceID` and set by slot `void SDSDevice::setDeviceID(QString dID)`.

Finally, the label `QLabel MainLabel` can be used for displaying the device name. Its default `QFrame::Shape` and `QFrame::Shadow` are `QFrame::Panel` and `QFrame::Raised`.

The color palette and the text font for the `SDSDevice` widget and its child objects are set by slot `void SDSDevice::setStyle(QFont,QPalette)`.

A default organization for `SDSDevice` widgets cannot be set at this stage, since it strongly depends on the device type and on the number and class of all necessary device− type related members.

5.2.2 Technical description

Class `SDSDevice` object has been designed by implementing the management of those functionalities that are present in any device. Such implementation has been done by identifying for each functionality what is effectively device independent and what is instead device dependent.

The device enabling and disabling is performed by slot `virtual void SDSDevice::enableDevice(bool)`, declared virtual since the tasks to be effectively performed are device dependent. It has to be implemented in `SDSDevice` subclasses to perform both tasks accordingly to the boolean value of its argument, namely if it's equal to `true` the enabling task has to be done, if it's equal to `false` the disabling one. This solution allows to connect this slot to the signal `void SDSStatusButton::activate(bool)` from `SDSStatusButton deviceAct`, also responsible for displaying the enabling status, so that both the device enabling and disabling can be manually invoked by clicking on the mentioned button. The enabling status is also stored in member `bool isEnabled`, whose values mean `true = enabled` and `false = disable`. As already explained in §4.2, the device enabling and disabling may require tasks that may not be successful. For this reason the requested enabling status is not set, in `SDSStatusButton deviceAct` and `bool isEnabled`, as `virtual void SDSDevice::enableDevice(bool)` is called. This setting will have to be implemented in the `SDSDevice` subclasses, in particular in those slots devoted to control if the enabling and disabling tasks are successfully performed.

Member `SDSEdit SetupFile` provides the field to type and display the name of the configuration file. The connection between its signal `void SDSValue::newValue(QString)` and slot `void SDSDevice::readSetupFile(QString)` ensures that the file is immediately read as its name is the new value for `SDSEdit SetupFile`. Slot `void SDSDevice::readSetupFile(QString)` opens the file whose name is its argument and reads it line by line. Once a line is read, slot `virtual void SDSDevice::readSetupLine(QString)` is called for analyzing each line. It's declared here as virtual since the parameters keywords and their argument syntax clearly depend on the device. This slot has to be implemented for reading those lines that set a parameter that is common to any device of the same kind (e.g. the sampling time for a backend), while those lines that set a parameter that is peculiar of a given device can be read by slot `virtual void SDSDevice::readCustomLine(QString)`. Slot `void SDSDevice::loadSetup()` is called when `QPushButton LoadSetup` is clicked: it opens a system dialog window for browsing the system directories and selecting the file to be loaded. Once a file is selected, its name is set as the new value for `SDSEdit SetupFile` and displayed accordingly to the conventions about the default directory mentioned before. The default directory is set by slot `void SDSDevice::setSetupDir(QString)` and stored in member `QString setupDir`. The file whose name is already displayed in `SDSEdit SetupFile` can be reloaded by clicking `QPushButton ReloadSetup`, whose signal `void QPushButton::clicked()` is connected to slot `void SDSDevice::reloadSetupFile()`, which in turn calls `void SDSDevice::readSetupFile(QString)` by setting as its argument the string displayed in `SDSEdit SetupFile`.

Two virtual slots are devoted to starting and stopping the device main task, namely `virtual void SDSDevice::start()` and `virtual void SDSDevice::stop()`, with the obvious meaning of their names. They can be manually called by clicking `QPushButton Start` and `QPushButton Stop`, thanks to their connection to the signal `void QPushButton::clicked()` emitted by these buttons. Their implementation is obviously deferred to fully implemented `SDSDevice` subclasses devoted for managing specific devices.

The device status is displayed by member `SDSEdit Status`, which has been set as not editable for obvious reasons.

The communication between the `SDSDevice` object and the managed device are managed by member `SDSCommThread *commThread` (see §5.3). A single command or a sequence of commands, separated by the semicolon `";"` character, are sent to the communication thread by slot `void SDSDevice::sendCommand(QString)`, whose argument is the string containing the command(s) to be sent. Slot `virtual void SDSDevice::readMessage(QString)` is

devoted to analyze any incoming message and taking the opportune decisions. Two other slots, `virtual void socketError(QAbstractSocket::SocketError)` and `virtual void socketState(QAbstractSocket::SocketState)`, are devoted to take decision when an error occurs in the socket connection or the connection state changes. A set of signal/slot connections ensure that the mentioned slots are called when necessary, but these connections cannot be instantiated in the `SDSDevice` constructor, since `SDSCommThread *commThread` has to be recasted to a fully implemented `SDSCommThread` subclass in the `SDSDevice` subclass devoted to managing a given device. Slot `void SDSDevice::createConnections()` has to be called in the fully implemented `SDSDevice` subclass for establishing these connections.

The possibility of sending manual commands to a device is managed by member `SDSEdit ManualCommand`. The connection between signal `void QLineEdit::returnPressed()`, from member `QLineEdit field` of `SDSEdit ManualCommand`, and slot `void SDSDevice::sendManualCommand()` ensures that the typed command is sent only if the return key is pressed.

Member `SDSLog deviceLog` manages all log messages (§4.7). The directory for the device log files is set by slot `void SDSDevice::setLogDir(QString ld)`, while the three characters code that identifies the device is set by `void SDSDevice::setDeviceID(QString id)`.

Member `QWidget auxiliaryWindow` is the auxiliary window. It is opened by clicking on `QPushButton auxiliaryWindowShow`, thanks to the connection between signal `void QPushButton::clicked()` and slot `void QWidget::show()`.

Member `QLabel MainLabel` is the only already declared graphic object. Apart of this, no default graphic organization is set. Slot `void SDSDevice::setStyle(QFont,QPalette)` sets the color palette and the text font for the main widget and `QWidget auxiliaryWindow`, while slot `virtual void SDSDevice::setChildrenStyle(QFont,QPalette)` has to be implemented in `SDSDevice` subclasses for setting these properties in other possible child widgets that are hosted in dedicated windows.

Inheritance

Inherits from: `QWidget`
Inherited by: `SDSAntenna`, `SDSBackend`, `SDSManager`

Public members

`QWidget auxiliaryWindow`
This member is a widget, to be hosted in a dedicated window, devoted to display those parameters whose presence is not strictly necessary in the `SDSDevice` main widget. The window hosting this widget can be opened by clicking `QPushButton auxiliaryWindowShow`.

`QPushButton auxiliaryWindowShow`
This button opens the window that hosts `QWidget auxiliaryWindow`.

`SDSCommThread *commThread`
This member is a pointer to the `SDSCommThread` object devoted to manage the socket connection to the device server and all communication to and from it.
All signal/slot connections have to be established by calling slot `void SDSDevice::createConnections()`.

`SDSStatusButton deviceAct`
This button enables/disables the device, by calling slot `virtual void SDSDevice::enableDevice(bool)`, and displays the device enabling status.

`QString deviceID`
This member stores the three characters device code. It's set by slot `void SDSDevice::setDeviceID(QString)`

`SDSLog deviceLog`
This member manages the device log messages. The directory for the log files is set by slot `void SDSDevice::setLogDir(QString)`.

`bool isEnabled`
This member stores the device enabling status by means of a boolean variable: `isEnabled = true` means that the device is enabled, `isEnabled = false` means that the device is disabled.

`QPushButton LoadSetup`
This button calls slot `void SDSDevice::loadSetup()` for opening a system dialog window that

allows the system directories browsing and the setup file selection.

`QLabel MainLabel`
This object is the main label in the `SDSDevice` widget, devoted to display the name of the managed device or the description of the role given to the `SDSDevice` object in the observer's application.

`SDSEdit ManualCommand`
This member provides the field for manually entering a command to be given to the device.

`QPushButton ReloadSetup`
This button calls slot `void SDSDevice::reloadSetupFile()` for reloading the setup file whose name is displayed in `SDSEdit SetupFile`.

`SDSEdit SetupFile`
This member is devoted to manually entering and anyway displaying the name of the selected setup file. Once a file name is entered, the file is automatically loaded. The setup file name has to be specified with its full path if no default location for the setup files has been set. If instead the default location has been set, by calling `void SDSDevice::setSetupDir(QString)`, the name of the file has to be entered as follows:

1) the requested file is in the default directory: its name only can be specified;

2) the requested file is in a subdirectory of the default one: its name has to be given with its relative path with respect to the default directory;

3) the requested file is elsewhere: its name has to be given with its absolute path.

`QPushButton Start`
This button allows to manually start the device main task. When it's clicked it calls slot `virtual void SDSDevice::start()`.

`SDSEdit Status`
This member is a not editable `SDSEdit` object devoted to displaying the device status.

`QPushButton Stop`
This button allows to manually stop the device main task. When it's clicked it calls slot `virtual void SDSDevice::stop()`.

Protected members

`QString setupDir`
This member stores the default directory for the setup files. Its value is set by slot `void SDSDevice::setSetupDir(QString)`.

Public slots

`void SDSDevice::createConnections()`
This slot establishes all signal/slot connections between the `SDSDevice` object and its member `SDSCommThread *commThread`. As explained in §5.3, this slot has to be called in the constructor of the fully implemented `SDSDevice` subclass, after recasting `SDSCommThread *commThread` to the fully implemented `SDSCommThread` subclass.

`void SDSDevice::loadSetup()`
This slot opens a system dialog window for browsing the system directories and selecting the setup file. Once a file is selected, its name is displayed in `SDSEdit SetupFile` and automatically loaded.

`void SDSDevice::readSetupFile(QString sFile)`
This slot opens the selected setup file `sFile`, extracts each line and calls slot `virtual void SDSDevice::readSetupLine(QString sLine)` for reading the extracted lines.

`void SDSDevice::reloadSetupFile()`
This slot reloads the setup file, whose name is already displayed in `SDSEdit SetupFile`.

`void SDSDevice::sendCommand(QString cmd)`
This slot transfers to `SDSCommThread *commThread` the string `cmd` containing the command(s) to be sent to the device. If multiple commands have to be sent, they have to be separated by the semicolon `";"` character. Any string passed as command to `SDSCommThread *commThread` is saved in the device log file.

`void SDSDevice::sendManualCommand()`
This slot is responsible for sending to the device a command that's manually entered by the user in member `SDSEdit ManualCommand`.

`void SDSDevice::setDeviceID(QString dID)`
This slot sets to `dID` the three character code for the device. If the given argument is longer, it's truncated to its first three characters.

`void SDSDevice::setLogDir(QString lDir)`
This slot sets to `lDir` the directory for hosting the device log files. The directory has to be specified with its full path.

`void SDSDevice::setSetupDir(QString sDir)`
This slot sets to `sDir` the default directory for the device setup files. The directory has to be specified with its full path.

`void SDSDevice::setStyle(QFont dFont, QPalette dPal)`
This slot sets to `dPal` the color palette and to `dFont` the text font for the `SDSDevice` widget, for `QWidget auxiliaryWindow`, and for other possible child widgets that are hosted in separate windows by calling, for these latter ones, slot `virtual void SDSDevice::setChildrenStyle(QFont dFont, QPalette dPal)`.

Virtual public slots

`virtual void SDSDevice::enableDevice(bool bl)`
This slot has to be implemented in fully implemented `SDSDevice` subclasses for both enabling and disabling the device control. The enabling algorithm has to be called when the argument

has the boolean value `true`, while the disabling one when the argument has the boolean value `false`.

`virtual void SDSDevice::readCustomLine(QString sLine)`
This slot has to be implemented in a fully implemented subclass of a `SDSDevice` subclass devoted to provide the base class for devices of a well identified type (e.g. the base class for backends). It's meant to be called in the implementation of slot `virtual void SDSDevice::readSetupLine(QString sLine)` for extracting, from a line in a setup file, the value of a parameter which is specific of a specific device.

`virtual void SDSDevice::setChildrenStyle(QFont dFont, QPalette dPal)`
This slot has to be implemented in fully implemented `SDSDevice` subclasses for setting to `dPal` the color palette and to `dFont` the text font for further possible child widgets hosted in dedicated windows.

`virtual void SDSDevice::start()`
This slot has to be implemented in fully implemented `SDSDevice` subclasses for starting the device main task.

`virtual void SDSDevice::stop()`
This slot has to be implemented in fully implemented `SDSDevice` subclasses for stop the device main task.

Virtual protected slots

`virtual void SDSDevice::readMessage(QString msg)`
This slot has to be implemented in fully implemented `SDSDevice` subclasses for analyzing the device message `msg` and taking the opportune decisions.

`virtual void SDSDevice::socketError(QAbstractSocket::SocketError se)`
This slot has to be implemented in fully implemented `SDSDevice` subclasses for taking the opportune decisions whenever the error `QAbstractSocket::SocketError se` occurs in the socket connection to the device server, and accordingly to the error type.

`virtual void socketState(QAbstractSocket::SocketState st)` This slot has to be implemented in fully implemented `SDSDevice` subclasses for taking the opportune decisions whenever the state of the connection to the device server changes to `QAbstractSocket::SocketState st`, and accordingly to the new connection state.

Established signal/slot connection

Signal `void SDSStatusButton::activate(bool)`, emitted by member `SDSStatusButton deviceAct`
to slot `virtual void SDSdevice::enableDevice(bool)`
This connection allows to manually switch the enabling status by clicking `SDSStatusButton deviceAct`.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton Start`
to slot `virtual void SDSdevice::start()`
This connection allows to manually start the device main task by clicking `QPushButton Start`.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton Stop`
to slot `virtual void SDSdevice::stop()`
This connection allows to manually stop the device main task by clicking `QPushButton Stop`.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton LoadSetup`
to slot `void SDSdevice::loadSetup()`
This connection allows the call for the slot that opens the system dialog window for selecting the setup file to be loaded, when `QPushButton LoadSetup` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton reloadSetup`
to slot `void SDSdevice::reloadSetupFile()`
This connection allows the call for the slot that reloads the setup file whose name is already displayed in member `SDSEdit SetupFile`, when `QPushButton reloadSetup` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton auxiliaryWindowShow`
to slot `void QWidget::show()` for member `QWidget auxiliaryWindow`
This connection allows the opening of the window that hosts `QWidget auxiliaryWindow` when `QPushButton auxiliaryWindowShow` is clicked.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit SetupFile`
to slot `void SDSDevice::readSetupFile(QString)`
This connection ensures that the setup file is automatically loaded when its name is set for member `SDSEdit SetupFile`.

Signal `void QLineEdit::returnPressed()`, emitted by member `QLineEdit field` of `SDSEdit ManualCommand`
to slot `void SDSDevice::sendManualCommand()`
This connection ensures that a manually given command is sent to the device only when the return key is pressed, after the command has been typed in `SDSEdit ManualCommand`.

Signal/slot connection to be established in fully implemented subclasses

Signal `void SDSCommThread::newCommandGiven(QString)` from member `SDSCommThread *commThread`
to slot `void SDSLog::addToLog(QString)` for member `SDSLog deviceLog`
established by calling slot `void SDSDevice::createConnections()`
This connection has to be established for displaying and storing in the log file all given commands as they are effectively sent to the device.

Signal `void SDSCommThread::newMessage(QString)` from member `SDSCommThread *commThread`
to slot `virtual void SDSDevice::readMessage(QString)`
established by calling slot `void SDSDevice::createConnections()`
This connection has to be established for passing to the `SDSDevice` object all device messages, so that they can be displayed, stored in the log file and the opportune decisions can be taken.

Signal `void SDSCommThread::socketError(QAbstractSocket::SocketError)` from member `SDSCommThread *commThread`
to slot `virtual void SDSDevice::socketError(QAbstractSocket::SocketError)` established by calling slot `void SDSDevice::createConnections()`
This connection ensures that the notification of an error occurred in the socket connection reaches the `SDSDevice` object, so that the opportune decisions can be taken.

Signal `void SDSCommThread::socketState(QAbstractSocket::SocketState)` from member `SDSCommThread *commThread`
to slot `virtual void SDSDevice::socketState(QAbstractSocket::SocketState)` established by calling slot `void SDSDevice::createConnections()`
This connection ensures that the notification of a status change in the socket connection reaches the `SDSDevice` object, so that the opportune decisions can be taken.

5.3 SDSCommThread class

5.3.1 Class overview

`SDSCommThread` is the base class for building the parallel thread devoted to directly manage the socket connection and all communications with the device' server. Inherited by `QThread`, this class has been designed for providing objects that are member of a `SDSDevice` object.

The socket connection parameters are set by slot `void SDSCommThread::setHostParameters(QString ip,int pt)`, whose arguments are the string containing the `IP` address of the device server and the communication port through which the connection has to be established. Slots `void SDSCommThread::connectToHost()` and `void SDSCommThread::disconnectFromHost()` perform all necessary tasks for respectively establishing and terminating the connection to the device server.

A command, or a sequence of commands separated by semicolons, for the device are set by slot `void SDSCommThread::newCommandQueue(QString)`. Slot `void SDSCommThread::setCommandInterval(int cmi)` allows to set the minimum time interval between two consecutive commands: its argument is the value of this time lag, expressed in milliseconds. By default it's set to `1000 ms`.

Some devices may require to be periodically queried for obtaining their current status, since their control software do not automatically send this information to the connected client.

Such a query command is set by slot `void SDSCommThread::setPeriodicCommand(QString cmd)`. This query command is sent at time intervals equal to the time lag set by slot `setCommandInterval(int cmi)`, unless a specific command has to be given to the device.

5.3.2 Technical description

Class `SDSCommThread` is a partially implemented class whose objects manage the socket connection and all communications to and from the device server. Its algorithms have been designed so that what can be considered independent to the device is performed by fully implemented slots, while all tasks that depend on the device itself have to be implemented in a dedicated `SDSCommThread` subclass by opportunely programming the virtual function of this class.

A `SDSCommThread` object is a `QThread` object, whose main members are `QTimer timer`, which ensures that two consecutive commands are sent at a minimum time distance equal to its timeout interval, and `QTcpSocket *socket`, which is the socket through which the connection is established to the device server.

Private slot `virtual void QThread::run()` is reimplemented in this class for starting `QTimer timer` with a timeout interval equal to the value set by slot `void SDSCommThread::setCommandInterval(int cmi)` and stored in member `int cmdPeriod`. The signal `void QTimer::timeout()` from `QTimer timer` is connected to slot `void SDSCommThread::timerHit()`, which is responsible for checking if there is a command to be sent to the device. Its implementation is a bit involved and it is illustrated in figure 5.1.

The command to be sent to the device can be either the periodic command stored in member `QString periodicCommand`, or a specific command if the content of member `QString newCommand` is different from the string `none`. A command for which it is necessary to wait for it's completion by the device has to be stored in member `QString awaitingCommand`, whose content has to be set by the reimplementation of slot `virtual void SDSCommThread::setAwaitingCommand()` in the fully implemented `SDSCommThread` subclass.

Whenever it is received the string containing either a single command, or a sequence of semicolon separated commands, this string is stored in member `QString commandQueue`. This string is read by slot `void SDSCommThread::nextCommandInQueue()`, which extracts the first command to be sent and stores it in `QString newCommand`.

Slot `void SDSCommThread::writeToSocket(QString cmd)` is responsible for effectively sending the command to the device. If the command to be sent is not the *periodic command*, signal `void SDSCommThread::newCommandGiven(QString)` is emitted for passing to other

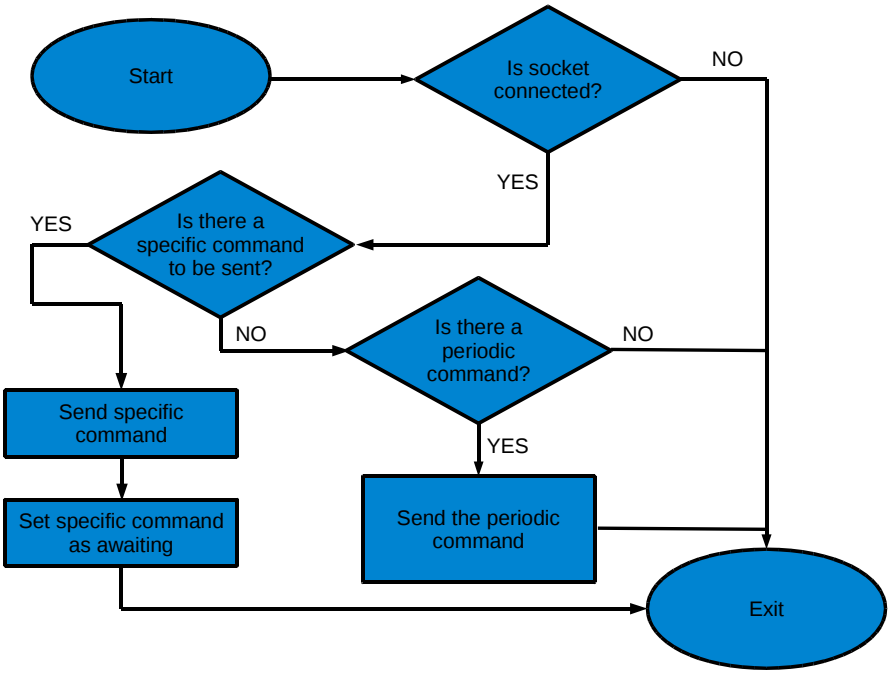

Figure 5.1: Flux diagram for slot `timerHit()`.

object the string containing the command to be sent. Before being written to the socket it may be necessary to reformat the command, accordingly to the syntax required by the device control software for commands received from an external client. A real example may better clarify this point. Let's consider the `PDFB` backend, available at, e.g., the Parkes radio telescope[1]. One of the commands the user can give to this device is:

```
config pdfb3_256_512_256
```

This is how the command has to be composed when directly given through the `PDFB` console. If this command has to be given by a client through a socket connection, it has to be formatted as follows:

```
COMMAND0024 config pdfb3_256_512_256
```

Slot `virtual QString SDSCommThread::composeCommand(QString cmd)` has to be implemented in fully implemented `SDSCommThread` subclasses for performing this kind of reformatting. Its argument, i.e. the argument for `void SDSCommThread::writeToSocket(QString cmd)`, is the

[1] http://www.parkes.atnf.csiro.au/

command to be given, exactly composed as if it were manually typed in the console running on the device server. The returned string is the input string, reformatted for being accepted by the device when received through a socket connection. Slot `composeCommand(QString cmd)` does not need to be reimplemented, if a device does not need such a commands reformatting.

Incoming messages are collected from the socket connection by slot `void SDSCommThread::readSocket()`. This slot emits the signal `void SDSCommThread::newMessage(QString)` for passing to other objects the string containing the received signal, and calls slot `virtual void SDSCommThread::processMessage(QString)`, which has to be implemented in fully implemented `SDSCommThread` subclasses for analyzing the received message and take the opportune decisions. The variety of messages and the decisions to be taken depend on the device control software. Among all possible messages and decisions, it's worth of mentioning messages indicating that the given command has been successfully accomplished and the related decision is to proceed with the next command in the command queue.

Member `QTcpSocket *socket` is a pointer to a `QTcpSocket` object, devoted for establishing the connection to the device server and communicate with it. Its connection parameters, namely the server's `IP` address and the communication port, are set by slot `void SDSCommThread::setHostParameters(QString ip,int pt)` and respectively stored in members `QString hostIP` and `int hostPort`. A signal/slot connection between signal `void QAbstractSocket::stateChanged(QAbstractSocket::SocketState st)`, emitted by `QTcpSocket *socket`,
and slot `void SDSCommThread::state(QAbstractSocket::SocketState st)` ensures that a reaction occurs whenever the state of `QTcpSocket *socket` changes, and the signal `void SDSCommThread::socketState(QAbstractSocket::SocketState st)` is emitted for passing to other objects the information about the new state. Two reactions have been implemented by default. If the new socket state is `QAbstractSocket::ConnectedState`, the `SDSCommThread` thread is started; if instead the new sockets state is `QAbstractSocket::UnconnectedState` the `SDSCommThread` thread is terminated. Further reactions can be disposed by reimplementing slot `virtual void SDSCommThread::socketStateChanged(QAbstractSocket::SocketState st)`, which is always called by `void SDSCommThread::state(QAbstractSocket::SocketState st)`. Similarly, the signal/slot connection between signal `void QAbstractSocket::error(QAbstractSocket::SocketError se)` and slot `void SDSCommThread::error(QAbstractSocket::SocketError se)`, ensures that a reaction occurs whenever an error occurs in the socket connection, and

the signal `void SDSCommThread::socketError(QAbstractSocket::SocketError st)` is emitted for passing to other objects the information about the occurred error. No default reaction has been implemented at this level, hence slot `virtual void SDSCommThread::socketErrorOccurred(QAbstractSocket::SocketError)`, which is always called by `void SDSCommThread::socketError(QAbstractSocket::SocketError st)`, has to be reimplemented if necessary.

Inheritance

Inherits from: `QThread`

Protected members

`QString awaitingCommand`
This member is a `QString` object for storing the last sent command while waiting for a response from the device about its result. It is set to `none` if no command has been sent or if the sent command does not require a response.

`int cmdPeriod`
This member stores the time interval in milliseconds between two consecutive commands. Its default value is `1000 ms`, and can be set by slot `void SDSCommThread::setCommandInterval(int cmi)`.

`QString commandQueue`
This member stores the string for containing all necessary commands for performing a procedure. It is set to `none` when no procedure is in progress or after the last command of a procedure is extracted for being sent to the device. A new command(s) string is set by slot `void SDSCommThread::newCommandQueue(QString cmq)`.

`QString hostIP`
This member stores the string indicating the IP address of the device server. Its value is set by slot `void setHostParameters(QString ip,int pt)`.

`int hostPort`
This member stores the number indicating the device server's communication port through which the socket connection has to be established. Its value is set by slot `void setHostParameters(QString ip,int pt)`.

`QString newCommand`
This member stores the specific command that has to be immediately sent to the device. It remains set to `none` if no specific command has to be sent.

`QString periodicCommand`
This member stores the periodic command that has to be sent to the device. It remains set to `none` if no periodic command is necessary.

`QTcpSocket *socket`
This member is a pointer to the `QTcpSocket` object that communicates through the socket connection with the device server. Its connection parameters are set by slot `void setHostParameters(QString ip,int pt)` and stored in members `QString hostIP` and `int hostPort`.

`QTimer timer`
This member is the repetitive timer for calling at regular intervals slot `void SDSCommThread::timerHit()`, responsible for checking whether there is a command to be sent to the device and for sending it. Its timeout interval is set by slot `void SDSCommThread::setCommandInterval(int cmi)`.

Public slots

`void SDSCommThread::connectToHost()`
This slot is a shortcut for calling `void QTcpSocket::connectToHost(QString hostIP,int hostPort)` for member `QTcpSocket socket`.

`void SDSCommThread::disconnectFromHost()`
This slot terminates the communication thread, by calling `void QThread::terminate()`, closes

the socket connection, by calling `void QTcpSocket::disconnectFromHost()` for member `QTcpSocket socket`, and sets to `none` the value for `awaitingCommand`, `commandQueue` and `newCommand`.

`void SDSCommThread::newCommandQueue(QString ncq)`
This slot replaces the content of `QString commandQueue` with the content of `QString ncq` and, if no command is awaiting response from the device, calls `void SDSCommThread::nextCommandInQueue()` for extracting from the new command queue the first command to send.

`void SDSCommThread::setCommandInterval(int cmi)`
This slot sets to `int cmi` the value for member `int cmdPeriod`, the minimum time interval between two consecutive commands.

`void SDSCommThread::setHostParameters(QString ip, int pt)`
This slot sets the value for `QString hostIP` with the content of string `QString ip`, and the value for `int hostPort` with the value for `int pt`.

`void SDSCommThread::setPeriodicCommand(QString pcm)`
This slot sets to `QString pcm` the value for member `QString periodicCommand`, the command that has to be regularly sent to the device.

Private slots

`void SDSCommThread::run()`
reimplemented from `virtual void QThread::run()`
This slot starts in the thread the repetitive timer `QTimer timer`, with a timeout interval equal to `int cmdPeriod`.

`void SDSCommThread::timerHit()`
This slot is called every `int cmdPeriod` milliseconds for checking if there is a command to be sent to the device, and in case for sending it.

Protected slots

`void SDSCommThread::error(QAbstractSocket::SocketError se)`
This slot is called whenever an error occurs in the socket connection. It emits the signal `void SDSCommThread::socketError(QAbstractSocket::SocketError)` for passing the occurred error to other objects, and it calls `virtual void SDSCommThread::socketErrorOccurred(QAbstractSocket::SocketError se)` for tacking the opportune actions in case of an error occurred in the socket connection.

`void SDSCommThread::nextCommandInQueue()`
This slot extracts from the command queue stored in string `QString commandQueue` the command to be sent to be sent to the device, and stores such command in string `QString newCommand`.

`void SDSCommThread::readSocket()`
This slot is called whenever an new message is ready to be read in `QTcpSocket *socket`. It reads the incoming messages from the socket connection, calls slot `virtual void SDSCommThread::processMessage(QString msg)` for analysing the received message and taking the opportune decisions, and emits signal `void SDSCommThread::newMessage(QString msg)` for passing to other objects the received signal.

`void SDSCommThread::state(QAbstractSocket::SocketState st)`
This slot is called whenever the socket connection state changes. It emits the signal `void SDSCommThread::socketState(QAbstractSocket::SocketState)` for passing to other objects the information about the new connection state, and executes the following default actions if the new state is `QAbstractSocket::ConnectedState` and `QAbstractSocket::UnconnectedState`. In the first case, it starts the `SDSCommThread` thread, while in the second case it terminates the `SDSCommThread` thread and sets to none the string stored in `awaitingCommand`, `commandQueue` and `newCommand`. This slot also calls `virtual void SDSCommThread::socketStateChanged(QAbstractSocket::SocketState st)` for taking further opportune decisions accordingly to the new socket's state.

`void SDSCommThread::writeToSocket(QString cmd)`
This slot emits the signal `void SDSCommThread::newCommandGiven(QString)` for passing to

other objects the command that is set to be sent, if it's different to the periodic command, then calls `virtual void SDSCommThread::composeCommand(QString cmd)` for formatting the command string accordingly to the device control software syntax requirements, and calls `quint64 QIODevice::write(const char *data)` for member `*socket` for sending the command string through the socket.

Virtual protected slots

`virtual QString SDSCommThread::composeCommand(QString cmd)`
This slot has to be implemented for formatting the command string accordingly to the device rules for accepting commands through a socket connection.

`virtual void SDSCommThread::processMessage(QString msg)`
This slot has to be implemented for analysing all messages from the device and taking the opportune decisions.

`virtual void SDSCommThread::setAwaitingCommand()`
This slot has to be implemented for storing in string `awaitingCommand` those commands for which it is necessary to wait for a response from the device before sending another command.

`virtual void SDSCommThread::socketErrorOccurred(QAbstractSocket::SocketError se)`
This slot has to be implemented for triggering the opportune actions whenever an error occurs in the socket connection, and accordingly to the error type.

`virtual void SDSCommThread::socketStateChanged(QAbstractSocket::SocketState st)`
This slot has to be implemented for triggering the opportune actions whenever the state of the socket connection changes, and accordingly to the new connection state.

Signals

`void SDSCommThread::newCommandGiven(QString ncg)`
Whenever a command is given to the device, exception given, for the periodic command, this signal is emitted for passing to other objects the string containing newly given command.

`void SDSCommThread::newMessage(QString msg)`
Whenever a message is received from the device, this signal is emitted for passing the string containing the received message.

`void SDSCommThread::socketError(QAbstractSocket::SocketError se)`
Whenever an error occurs in the socket connection, this signal is emitted for passing the information about the occurred error.

`void SDSCommThread::socketState(QAbstractSocket::SocketState st)`
Whenever the socket connection state changes, this signal is emitted for passing the information about the new connection state.

Established signal/slot connections

Signal `void QAbstractSocket::readyRead()`, emitted by member `QTcpSocket *socket`
to slot `void SDSCommThread::readSocket()`
This connection allows the reading of all incoming messages as soon as they are ready.

Signal `void QAbstractSocket::error(QAbstractSocket::SocketError)`, emitted by member `QTcpSocket *socket`
to slot `void SDSCommThread::error(QAbstractSocket::SocketError)`
This connection ensures that the opportune decisions are taken whenever an error occurs in the socket connection.

Signal `void QAbstractSocket::state(QAbstractSocket::SocketState)`, emitted by member `QTcpSocket *socket`
to slot `void SDSCommThread::stateChanged(QAbstractSocket::SocketState)`
This connection ensures that the opportune decisions are taken whenever the state changes in the socket connection.

5.4 SDSAntenna class

5.4.1 Class overview

`SDSAntenna` is the base class for building an object for configuring and controlling a telescope. This class inherits from `SDSDevice`: members and slots, that have already been declared in `SDSDevice` class, have in this subclass same roles and functions mentioned in section § 5.2, with some obvious exceptions. The large number of graphic members in a `SDSAntenna` object suggested to organize a default layout for both the main widget and the secondary windows. Figures 5.2 and 5.3 display the adopted default layouts, and are used in this section for helping in describing the class and its objects functionalities.

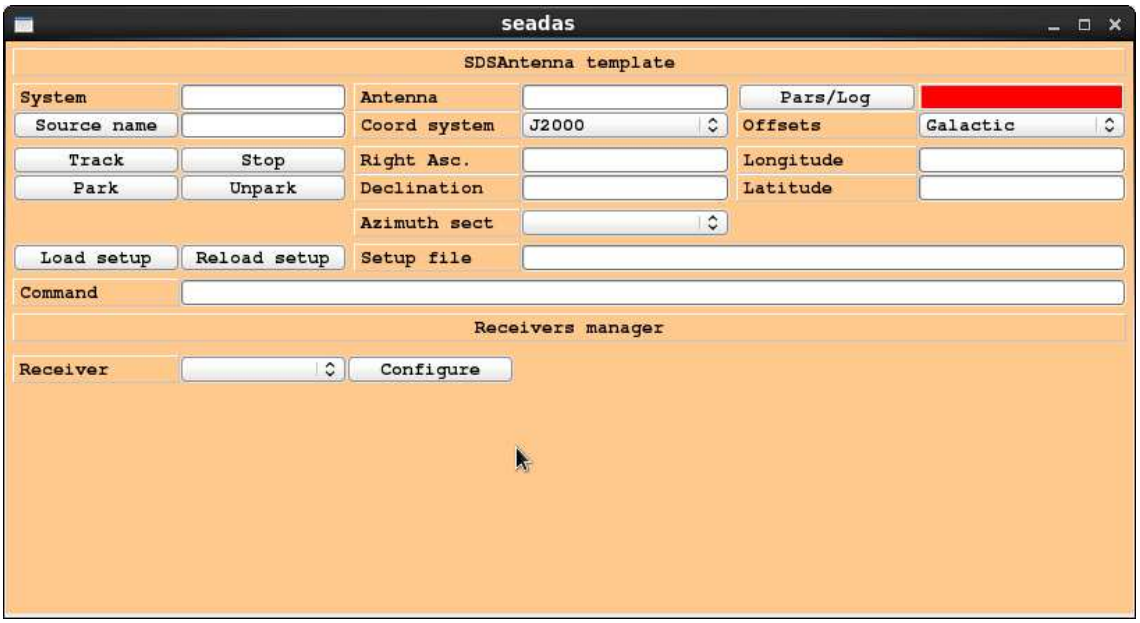

Figure 5.2: `SDSAntenna` main widget's default layout.

The task the antenna has to continuously perform during the observation is the source's `TRACKING`. For this reason member `QPushButton Start` displays the text `Track`. Member `QPushButton Stop`, displaying the text `Stop`, is instead the button for manually stopping this

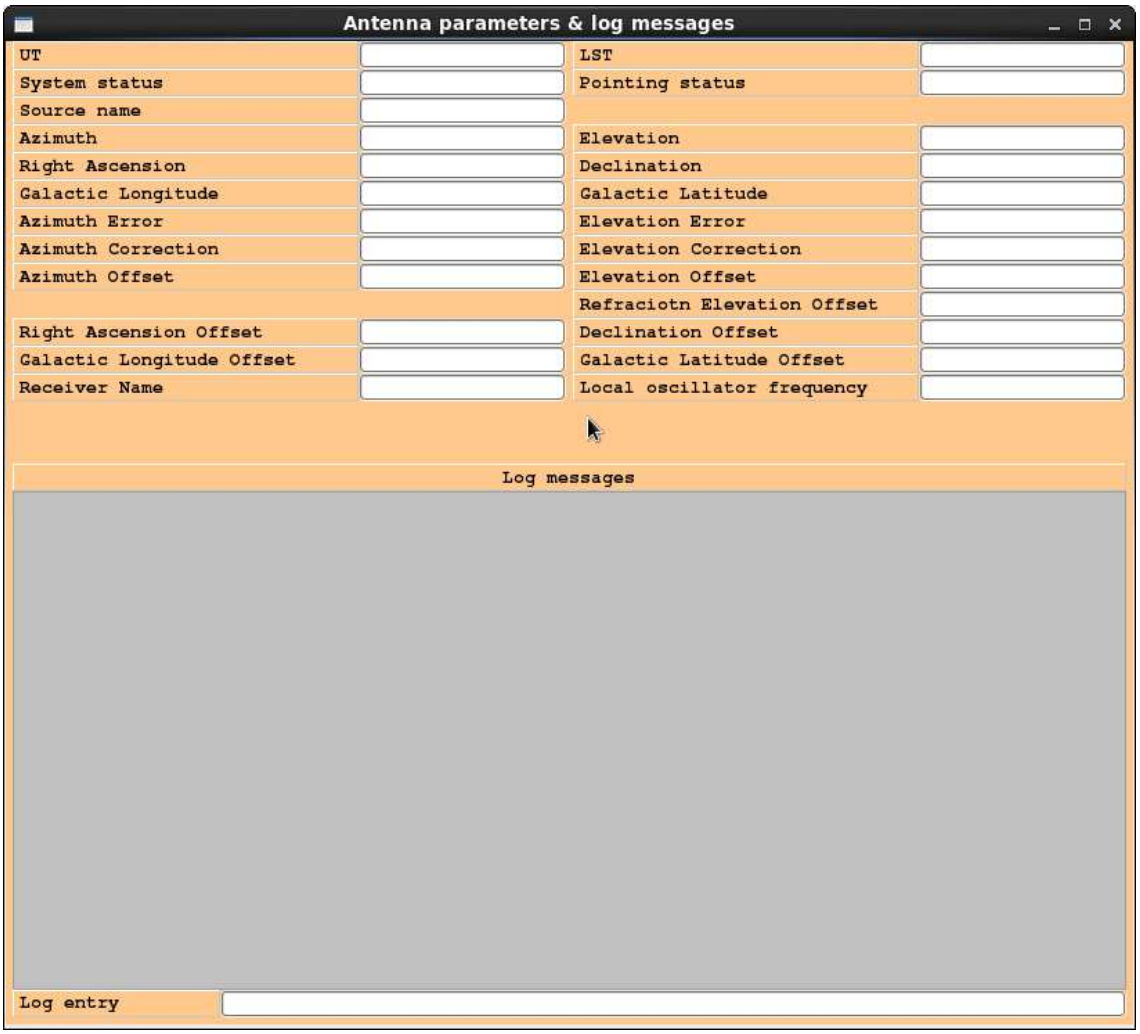

Figure 5.3: `SDSAntenna` secondary window's default layout.

task.

Member `SDSEdit Status`, placed at the top center of the `SDSAntenna` widget and displaying in its label the text `Antenna`, is devoted for displaying whether the antenna is `TRACKING` a source, `SLEWING` to source's coordinates or other similar situations. At its right find place the button `QPushButton auxiliaryWindowShow`, displaying text `Pars/Log`, for showing the secondary window `QWidget auxiliaryWindow`, and the button `SDSStatusButton deviceAct`, the red button displaying no text, for enabling and disabling the control of the antenna.

In the center of the widget can be found the two buttons, members `QPushButton LoadSetup`, displaying text `Load setup,` and `QPushButton ReloadSetup`, displaying text

Reload setup, and the field, member SDSEdit SetupFile, whose label displays text Setup file, for managing the setup file. Slot void SDSAntenna::readSetupLine(QString sl) has been implemented from virtual void SDSDevice::readSetupLine(QString sl) for setting the source name and coordinates, keyword Source, possible offsets in the given coordinates, keyword Offsets, and setting receiver's parameters, keyword Receiver. Just below the setup objects, finds place member SDSEdit ManualCommand, whose label displays text Command.

Parameters object introduced here are members SDSCombo CoordSys, displaying text Coord system, for selecting the coordinate system through which source coordinates are given, SDSCombo OffsetFrame, displaying text Offsets, for selecting the coordinate system for the position offsets. Members SDSEdit Longitude and SDSEdit Latitude, placed below SDSCombo CoordSys, manage respectively the longitude and latitude coordinate in the selected coordinate system, while members SDSEdit LongOffset and SDSEdit LatOffset, placed below SDSCombo OffsetFrame, manage the offset value respectively in the longitude and latitude coordinate in the selected frame for the offsets themselves. Member SDSCombo AzSector, displaying text Azimuth sect, manages allows the selection of the telescope azimuth wrap.

Member SDSEdit SourceName manages the name of the source to be observed. Its label is replaced by member QPushButton SourceNameBt, displaying text Source name, which pops up the SDSCatalogue *sourceCat window, the window for the source catalogue. If the name of a catalogue source is manually typed in SDSEdit SourceName, its catalogue coordinates are automatically loaded and displayed in SDSEdit Longitude and SDSEdit Latitude.

Member SDSEdit SystemStatus, displaying text System, is a not editable SDSEdit object, devoted to display the antenna system status, i.e. whether the antenna system is working without problems or there are issues that may affect the antenna operations and/or performances.

Members QPushButton Park and QPushButtons Unpark, displaying their names as text, are those buttons that allow the antenna manual parking and unparking.

Member SDSCombo Receiver, displaying text Receiver, is devoted to selecting the receiver to be used, by displaying their parameters' widgets, while member QPushButton ConfigureReceiver, displaying text Configure, is the button for manually configuring the antenna with the selected receiver. They are placed under member QLabel ReceiverLabel, displaying text Receiver manager.

Member QWidget auxiliaryWindow, displayed in figure 5.3, hosts SDSLog deviceLog and a set of non editable SDSEdit objects devoted to display the antenna system parameters, whose list is here below:

1 `SysUTC` (label `UT`): the current UTC;

2 `LST` (label `LST`): the local sidereal time;

3 `SystemStatusMn` (label `System status`): the status of the system, as already described for member `SDSedit SystemStatus` in the main widget. `SDSedit SystemStatus` always displays the value displayed in `SystemStatusMn`;

4 `PointingStatusMn` (label `Pointing status`): the antenna pointing status, as already described for member `SDSedit Status` in the main widget. `SDSedit Status` always displays the value displayed in `PointingStatusMn`;

5 `SourceNameMn` (label `Source name`): the source name;

6 `Azimuth` (label `Azimuth`): antenna azimuth;

7 `Elevation` (label `Elevation`): antenna elevation;

8 `RightAscension` (label `Right Ascension`): right ascension of the current pointing;

9 `Declination` (label `Declination`): declination of the current pointing;

10 `GalacticLongitude` (label `Galactic Longitude`): galactic longitude of the current pointing;

11 `GalacticLatitude` (label `Galactic Latitude`): galactic latitude of the current pointing;

12 `AzimuthError` (label `Azimuth Error`): difference between the azimuth values for requested and pointed position;

13 `AzimuthCorrection` (label `Azimuth Correction`): apported correction to the source azimuth;

14 `AzimuthOffset` (label `Azimuth Offset`): value for the pointing offset in the azimuth coordinate;

15 `ElevationError` (label `Elevation Error`): difference between the elevation values for requested and pointed position;

16 `ElevationCorrection` (label `Elevation Correction`): apported correction to the source elevation;

17 `ElevationOffset` (label `Elevation Offset`): value for the pointing offset in the elevation coordinate;

18 `RefractionElevationCorrection` (label `Refraction Elevation Correction`): correction to the source elevation due to the atmosphere's light refraction;

19 `RaOffset` (label `Right Ascension Offset`): value for the pointing offset in the right ascension coordinate;

20 `DeclOffset` (label `Declination Offset`): value for the pointing offset in the declination coordinate;

21 `GalLongitudeOffset` (label `Galactic Longitude Offset`): value for the pointing offset in the galactic longitude coordinate;

22 `GalLatitudeOffset` (label `Galactic Latitude Offset`): value for the pointing offset in the galactic latitude coordinate;

23 `ReceiverCode` (label `Receiver Name`): the code that identifies the receiver in focus;

24 `LOFrequency` (label `Local oscillator frequency`): the local oscillator frequency, in MHz.

Members `SDSSource source`, `SDSReceiver receiver` and `SDSTelescope telescope` store the informations about the selected source, the selected receiver and the telescope parameters.

Slot
`void SDSAntenna::setChildrenStyle(QFont cf,QPalette cp)` has been implemented from `virtual void SDSDevice::setChildrenStyle(QFont cf,QPalette cp)` so that it calls `void QWidget::setFont(QFont cf)` and `void QWidget::setPalette(QPalette cp)` for the `SDSCatalogue *sourceCat` widget, since it is placed in a dedicated window.

5.4.2 Technical description

Class `SDSAntenna` is an `SDSDevice` derived class, for which all functionalities for managing an antenna have been implemented in those tasks and algorithms that can be considered generic for an antenna, while a set of virtual slots has been declared, for implementing in `SDSAntenna` subclasses the antenna peculiarities.

Twelve fully implemented slots, nine public and three protected, perform those tasks that can be considered independent from the antenna system details.

Slot `void SDSAntenna::changeCoordSys(int csi)`, connected to the `void SDSCombo::newIndex(int)` signal emitted by member `SDSCombo CoordSys`, has as argument selected coordinate system item index in `SDSCombo CoordSys` items list. It changes texts displayed in `SDSEdit Longitude` and `SDSEdit latitude` labels, so that they show the coordinate name in the selected frame and, disables `QPushButton SourceNameBt` if it has been selected the horizontal frame, while in all their cases enables this button and loads the catalogue coordinates for the source whose name is displayed in `SDSEdit SourceName`, if the source is present in the implemented catalogue.

As already mentioned, in order to have a source catalogue class `SDSCatalogue` has to be subclassed and member `sourceCat` has to be recasted to the fully implemented `SDSCatalogue` subclass. All signal slot connections have to be set after the mentioned recast. Slot `void SDSAntenna::createCatalogueConnections()` established all necessary connections, namely:

i Signal `void SDSCatalogue::newSourceParameters(QString)` to slot `virtual void SDSAntenna::updateSourceParameters(QString)`: the passed string contains the catalogue selected source's name and coordinates, while the called slot updates these values in members `SourceName`, `Phi`, `Theta` and `CoordSys`;

ii Signal `void QPushButton::clicked()`, emitted by member `SourceNameBt`, to slot `void QWidget::show()` for `sourceCat`: this action opens the catalogue window;

iii Signal `void SDSValue::newValue(QString)`, emitted by member `SourceName`, to slot `virtual void SDSCatalogue::findSourceCoords(QString)`: the passed string is the newly set source name, and the called slot queries the catalogue for finding the coordinates for the source;

Slots `void SDSAntenna::newCoordSys(QString)`, `void SDSAntenna::newSourceName(QString)`, `void SDSAntenna::newLongitude(QString)` and `void SDSAntenna::newLatitude(QString)` are responsible for passing the source parameters whenever their value changes. They are connected to the signal `void SDSValue::newValue(QString)`, emitted by members `SDSCombo CoordSys`, `SDSEdit SourceName`, `SDSEdit Longitude` and `SDSEdit Latitude` respectively, they set the new value in the correspondent member in object `SDSSource source`, then emit the `void SDSAntenna::SourceParameters(SDSSource *)`, so that all source parameters are passed with the updated value. In the case of a catalogue source these slots are redundant, while in the opposite case they represent the only mean to pass source parameters to other objects.

Slot `void SDSAntenna::readSetupLine(QString sl)` has been implemented from `virtual void SDSDevice::readSetupLine(QString sl)` for setting the source name and coordinates, keyword `Source`, possible offsets in the given coordinates, keyword `Offsets`, and setting receiver's parameters, keyword `Receiver`. For the first two mentioned keywords, `Source` and `Offsets`, the analysis of the setup line is already implemented, while for the last one, `Receiver`, the line analysis is deferred to slot `virtual void SDSAntenna::readReceiverLine(QString rl)`. The analysis of all lines that do not start with any so far mentioned keyword is deferred to slot `virtual void SDSDevice::readCustomLine(QString sl)`, accordingly to the basic prescription indicated in §5.2.

Slot `void SDSAntenna::selectReceiver(QString recString)` allows the receiver selection, through the corresponding identification string, in the `SDSCombo ReceiverCombo` items list, by simply calling `void SDSCombo::setValue(QString recString)` for member `SDSCombo ReceiverCombo`.

Slot `void SDSAntenna::setOffsetsFrame(int of)`, connected to signal `void SDSCombo::NewIndex(int)` emitted by member `SDSCombo OffsetsFrame`, it's called whenever the selected item changes in `SDSCombo OffsetsFrame` It sets texts in the labels for `LongOffsets` and `LatOffsets`, so that the coordinate names for the pointing offsets are the correct ones for the selected frame. If the `SDSCombo OffsetsFrame` selected item is `Off`, which corresponds to the item index `0` and means that no pointing offsets are demanded, `SDSEdit LongOffsets` and `SDSEdit LatOffsets` are cleared and hidden. All other selections show these two objects but do not change their value.

Slot `void SDSAntenna::setTrackLimits(float la,float le, float ua, float ue)` sets the values for `SDSAntenna` members that store the antenna limits in azimuth and elevation,

namely `float minAz` and `float maxAz` for the minimum and maximum azimuth value, and `float minEl` and `float maxEl` for the minimum and maximum elevation value. Their arguments set, in the order, the minimum value for the azimuth, the minimum value for the elevation, the maximum value for the azimuth, the maximum value for the elevation.

The antenna position is checked with respect to the track limits by protected slot `void SDSAntenna::checkTrackLimits()`. This slot extracts the current position from members `Azimuth` and `Elevation` and compares them to their minimum and maximum values. If limits are reached, signals `void SDSAntenna::azimuthLimitReached()` or `void SDSAntenna::elevationLimitReached()` are emitted, depending on the reached track limit.

Protected slot `void SDSAntenna::configureReceiver()` is responsible for executing all necessary tasks for placing a receiver in its focal position and configuring the whole antenna accordingly to the selection and the receiver's parameters values. It calls, at first, slot `virtual QString SDSAntenna::getReceiverConfigCommand(int rci)`, by setting its argument equal to the `SDSCombo ReceiverCombo` current index, so that it builds the the string containing all configuration commands, then transfers the command queue to the communication thread.

Protected slot `void SDSAntenna::updateAntennaStatus()` is devoted to displaying the current pointing in the `SDSAntenna` main widget. It sets the values for `SDSEdit Longitude` and `SDSEdit Latitude` with the coordinates values in the system indicated by `SDSCombo CoordSys` and, if a frame is selected in `SDSCombo OffsetsFrame` does the same for the offsets values. This slot is not called anywhere in `SDSAntenna` and it has to be used with caution in order to avoid confusions or interferences with the requested coordinates values.

Eight virtual slots, seven public and one protected, have been declared for performing those tasks whose goal is well identified, but their algorithms depend on the antenna system. Slots `virtual void SDSAntenna::park()` and `virtual void SDSAntenna::unpark()` have to be implemented for respectively parking or unparking the antenna, either by clicking buttons `Park` or `Unpark` to which they are respectively connected, or by calling them in the code as a consequence of some situations that may require so.

Slot `virtual void SDSAntenna::buildReceiverInfos()` has to be implemented for setting in `SDSReceiver receiver` the values for the selected receiver. Its called inside slot `void SDSAntenna::configureReceiver()` after the receiver's configuration commands are sent to the antenna server, but just before the emission of the signal `void SDSAntenna::receiverParameters(SDSReceiver *)` that passes these informations to other objects.

Slot `virtual void SDSAntenna::checkAntennaStatus()` has to be implemented for

analysing the antenna system parameters, i.e. the ones displayed in `QWidget auxiliaryWindow` and take decisions accordingly to the situation. A good policy may be to call this slot inside `virtual void SDSAntenna::getAntennaParameters(QString parstring)`, whose argument is the string received from the antenna system that contains the antenna current parameters, and devoted to firstly display the received parameters' values in the opportune `SDSEdit` and later to trigger all system checks.

Slot `virtual QString SDSAntenna::getReceiverConfigCommand(int recIndex)` has to be implemented for building the queue of commands that is necessary for configuring the selected receiver accordingly to the given parameters. Its argument is the receiver index, which corresponds to the item index if the receiver in the `SDSCombo ReceiverCombo` item list.

Slot `virtual void SDSAntenna::readReceiverLine(QString)` has to be implemented for reading the setup line where receiver parameters are indicated and assigning their values to the opportune `SDSValue` objects.

Slot `virtual void SDSAntenna::updateSourceParameters(QString sourceString)` has to be implemented for reading the source parameters string `sourceString` that comes from `SDSCatalogue *sourceCat` after the selection of a source. This slot has to be virtual since the organization of the values in `QString sourceString` depends on the `SDSCatalogue` fully implemented subclass.

The protected slot `virtual void SDSAntenna::selectReceiver(int recIndex)` has to be implemented for displaying in the main widget those parameters objects that allow the management of the parameters for the selected receiver. Its argument is the receiver index in `SDSCombo ReceiverCombo`, and its connection to signal `void SDSCombo::newIndex(int)` ensures that this slot is called whenever its index changes.

Seven signals have been also implemented, some of them have been already mentioned. `void SDSAntenna::azimuthLimitReached()` and `void SDSAntenna::elevationLimitReached()` are emitted whenever the antenna reaches its azimuth or elevation limit position respectively; `void SDSAntenna::sourceParameters(SDSSource *)` passes the whole set of source parameters, and similarly `void SDSAntenna::receiverParameters(SDSReceiver *)` passes the whole set of parameters for the currently configured receiver; `void SDSAntenna::currentPointingStatus(QString)` and `void SDSAntenna::currentSystemStatus(QString)` pass the string that describe the status displayed in members `SDSEdit Status` and `SDSEdit SystemStatus` respectively, while signal `void SDSAntenna::trackingFailure()` is meant to be emitted in case of a severe failure in

the source tracking.

`SDSDevice` virtual slots that have not yet been implemented need to be implemented in the fully implemented `SDSAntenna` subclass. In particular, slot `void SDSDevice::start()` has to be implemented for pointing the antenna to the source and tracking it, and if necessary, slot `void SDSDevice::stop()` has to be implemented for stopping the source tracking.

The building of a fully implemented object devoted to manage a specific antenna needs the subclassing of class `SDSAntenna` and, in particular, the following steps in the given order:

1) class `SDSCommThread` is subclassed so that all its virtual slots can be implemented, again accordingly to the antenna specific functionalities;

2) member `SDSCommThread *commThread` is recasted to the fully implemented `SDSCommThread` subclass;

3) slot `void SDSDevice::createConnections()` is called for establishing all necessary signal/slot connections between `SDSCommThread *commThread` and its parent object;

4) class `SDSCatalogue` is subclassed so that all its virtual slots can be implemented, accordingly to the catalogue to be built;

5) member `SDSCatalogue *sourceCat` is recasted to the fully implemented `SDSCatalogue` subclass;

6) slot `void SDSDevice::createCatalogueConnections()` is called for establishing all necessary signal/slot connections between `SDSCatalogue *sourceCat` and its parent object;

7) all `SDSDevice` and `SDSAntenna` virtual slots are fully implemented accordingly to the antenna specific functionalities.

Inheritance

Inherits from: `SDSDevice`

Public members

`SDSCombo AzSector`
This member is the `SDSCombo` object for setting and displaying the preferred azimuth wrap.

`QPushButton ConfigureReceiver`
This button is devoted for manually triggering the slot that performs all necessary tasks for placing the selected receiver in focus and configuring the antenna accordingly to the selected receiver and its parameters.

`SDSCombo CoordSys`
This member is the `SDSCombo` object for selecting and displaying the coordinate system.

`SDSEdit Latitude`
This member is the `SDSEdit` object for setting and displaying the latitude coordinate in the selected coordinate system, i.e. Elevation, Declination or Galactic Latitude.

`SDSEdit LatOffset`
This member is the `SDSEdit` object for setting and displaying the pointing offset in the latitude coordinate, in the coordinate system selected for the offsets.

`SDSEdit Longitude`
This member is the `SDSEdit` object for setting and displaying the longitude coordinate in the selected coordinate system, i.e. Azimuth, Right Ascension or Galactic Longitude.

`SDSEdit LongOffset`
This member is the `SDSEdit` object for setting and displaying the pointing offset in the longitude coordinate, in the coordinate system selected for the offsets.

`SDSCombo OffsetsFrame`
This member is the `SDSCombo` object for selecting and displaying the coordinate system for the pointing offsets.

`QPushButton Park`
This member is the button for manually triggering the slot for parking the antenna.

`SDSCombo ReceiverCombo`
This member is the `SDSCombo` object for selecting the receiver and displaying the selection.

`QLabel ReceiverLabel`
This widget is the label for identifying the frame devoted to the receivers management.

`SDSSource source`
This member is the `SDSSource` object for storing and passing the source parameters.

`SDSCatalogue *sourceCat`
This member is a pointer to the `SDSCatalogue` object that manages the sources catalogue. It has to be recasted, in the fully implemented `SDSAntenna` subclass, to the fully implemented `SDSCatalogue` subclass.

`SDSEdit SourceName`
This member is the `SDSEdit` object for setting and displaying the source name.

`QPushButton SourceNameBt`
This button is devoting for opening the `SDSCatalogue *sourceCatalogue` window.

`SDSEdit SystemStatus`
This member is a not editable `SDSEdit` object devoted for displaying the antenna system status.

`QPushButton Unpark`
This button is devoted for manually triggering the slot for unparking the antenna.

Protected members

`SDSEdit Azimuth`
This member is the `SDSEdit` object that displays the antenna current azimuth.

`SDSEdit AzimuthCorrection`
This member is the `SDSEdit` object that displays the apported azimuth correction due to the pointing model.

`SDSEdit AzimuthError`
This member is the `SDSEdit` object that displays the difference between the requested and current azimuth.

`SDSEdit AzimuthOffset`
This member is the `SDSEdit` object that displays the current pointing offset in azimuth.

`SDSEdit Declination`
This member is the `SDSEdit` object that displays the antenna current declination.

`SDSEdit DeclOffset`
This member is the `SDSEdit` object that displays the current pointing offset in declination.

`SDSEdit Elevation`
This member is the `SDSEdit` object that displays the antenna current elevation.

`SDSEdit ElevationCorrection`
This member is the `SDSEdit` object that displays the apported elevation correction due to the pointing model.

`SDSEdit ElevationError`
This member is the `SDSEdit` object that displays the difference between the requested and current elevation.

`SDSEdit ElevationOffset`
This member is the `SDSEdit` object that displays the current pointing offset in elevation.

`SDSEdit GalacticLatitude`
This member is the `SDSEdit` object that displays the antenna current Galactic latitude.

`SDSEdit GalacticLongitude`
This member is the `SDSEdit` object that displays the antenna current Galactic longitude.

`SDSEdit GalLatitudeOffset`
This member is the `SDSEdit` object that displays the current pointing offset in the Galactic latitude.

`SDSEdit GalLongitudeOffset`
This member is the `SDSEdit` object that displays the current pointing offset in the Galactic longitude.

`SDSEdit LOFrequency`
This member is the `SDSEdit` object that displays the frequency of the local oscillator for the receiver in focus.

`SDSEdit LST`
This member is the `SDSEdit` object that displays the antenna system's local sidereal time.

`float minAz`
This member stores the lower limit for the antenna azimuth.

`float minEl`
This member stores the lower limit for the antenna elevation.

`float maxAz`
This member stores the upper limit for the antenna azimuth.

`float maxEl`
This member stores the upper limit for the antenna elevation.

`SDSEdit PointingStatusMn`
This member is the `SDSEdit` object that displays the antenna pointing status in `QWidget auxiliaryWindow`.

`SDSEdit RaOffset`
This member is the `SDSEdit` object that displays the current pointing offset in right ascension.

`SDSEdit ReceiverCode`
This member is the `SDSEdit` object that displays the name of the receiver currently in focus.

`SDSEdit RefractionElevationCorrection`
This member is the `SDSEdit` object that displays the apported elevation correction due to atmosphere light refraction.

`SDSEdit RightAscension`
This member is the `SDSEdit` object that displays the antenna current right ascension.

`SDSEdit SourceNameMn`
This member is the `SDSEdit` object that displays the source name, as stored in the antenna system.

`SystemStatusMn`
This member is the `SDSEdit` object that displays the antenna system status in `QWidget auxiliaryWindow`.

`SDSEdit SysUTC`
This member is the `SDSEdit` object that displays the antenna system's UTC.

Public slots

`void SDSAntenna::changeCoordSys(int cs)`
This slot sets texts for labels of members `SDSEdit Longitude` and `SDSEdit Latitude`, so that they display the names of the coordinates in the selected coordinate system. It disables `QPushButton SourceNameBt`, if the horizontal system is selected, while in all other cases it enables the mentioned button and loads the catalogue coordinates, in the selected system, for the source whose name is displayed by `SDSEdit SourceName`, if the source is in the catalogue. Its argument is the item index in `SDSCombo CoordSys` for the selected coordinate system.

`void SDSAntenna::createCatalogueConnections()`
This slot establishes the signal/slot connections involving member `SDSCatalogue *sourceCat`. The connections it sets are:

i Signal `void SDSCatalogue::newSourceParameters(QString)` to slot `virtual void SDSAntenna::updateSourceParameters(QString)`: the passed string contains the catalogue selected source's name and coordinates, while the called slot updates these values in members `SDSEdit SourceName`, `SDSEdit Longitude`, `SDSEdit Latitude` and `SDSCombo CoordSys`;

ii Signal `void QPushButton::clicked()`, emitted by member `SourceNameBt`, to slot `void QWidget::show()` for `sourceCatalogue`: this action opens the catalogue window;

iii Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit SourceName`, to slot `virtual void SDSCatalogue::findSourceCoords(QString)`: the passed string is the newly set source name, and the called slot queries the catalogue for finding the coordinates for the source;

This slot has to be called, in the fully implemented `SDSAntenna` subclass, after having recasted `SDSCatalogue *sourceCat` to the fully implemented `SDSCatalogue` subclass.

`SDSAntenna::newCoordSys(QString ncs)`
This slot sets in member `SDSSource source` the value `ncs` for the coordinate system, then emits the `void SDSAntenna::sourceParameters(SDSSource *)` signal for passing to other object the full set of updated source's parameters.

`SDSAntenna::newLatitude(QString nth)`
This slot sets in member `SDSSource source` the value `nth` for the source latitude coordinate, then emits the `void SDSAntenna::sourceParameters(SDSSource *)` signal for passing to other object the full set of updated source's parameters.

`void SDSAntenna::newLongitude(QString nph)`
This slot sets in member `SDSSource source` the value `nph` for the source longitude coordinate,

then emits the `void SDSAntenna::sourceParameters(SDSSource *)` signal for passing to other object the full set of updated source's parameters.

`void SDSAntenna::newSourceName(QString nsn)`
This slot sets in member `SDSSource source` the value `nsn` for the source name, then emits the `void SDSAntenna::sourceParameters(SDSSource *)` signal for passing to other object the full set of updated source's parameters.

`void SDSAntenna::readSetupLine(QString sl)`,
reimplemented from `virtual void SDSDevice::readSetupLine(QString sl)`
This slot has been reimplemented for reading the setup lines containing the keywords `Source`, `Offsets` and `Receiver`. The implementation of the `Source` keyword allows the following syntaxes:

i) `Source = [`*Source name*`]`

ii) `Source = [`*Source name*`], [`*Coord sys*`]`

iii) `Source = [`*Source name*`], [`*Coord sys*`], [`*long value*`], [`*lat value*`]`

The first syntax load the catalogue coordinates in the currently selected coordinate system, for the source whose name is *Source name*. The second syntax loads the catalogue coordinates in the coordinate system given by *Coord sys*, again for the source whose name is *Source name*. The third syntax loads the requested *long value* and *lat value* coordinates, meant as the longitude and latitude coordinate in the given system *Coord sys*, for a source whose name is *Source name*. The third syntax obviously is mandatory for sources not present in the catalogue object `SDSCatalogue *sourceCat`. The implementation of the `Offsets` keyword allows the following syntaxes.

i) `Offsets = Off`

ii `Offsets = [`*Coord sys*`], [`*long offset*`], [`*lat offset*`]`

The first syntax selects the item `Off` in `SDSCombo OffsetsFrame`, hence no pointing offsets are set, while the second syntax selects the item *Coord sys* in `SDSCombo OffsetsFrame`, then assigns the values *long offset* and *lat offset* to members `SDSedit LongOffset` and `SDSedit LatOffset` respectively. The implementation of the `Receiver` keyword calls slot `virtual void`

`SDSAntenna::readReceiverLine(QString sl)` by setting as its argument the whole setup line `sl`. If the keyword is none of the ones mentioned above, `readSetupLine()` calls `virtual void SDSAntenna::readCustomLine(QString sl)`, again setting as its argument the whole setup line `sl`.

`void SDSAntenna::selectReceiver(QString rcname)`
This slot allows to select the receiver in member `SDSCombo ReceiverCombo`, by using the string that identifies the receiver in the `ReceiverCombo` items list.

`void SDSAntenna::setChildrenStyle(QFont cf, QPalette cp)`, reimplemented from `virtual void SDSDevice::setChildrenStyle(QFont cf, QPalette cp)`
This slot set to `cf` the text font and to `cp` the color palette for the `SDSCatalogue *sourceCat` widget, since it's located in a dedicated window.

`void SDSAntenna::setOffsetsFrame(int ofi)`
This slot takes as argument the index `ofi` of the selected item in `SDSCombo OffsetsFrame`. If the index value is `0`, which corresponds to item `Off`, members `LongOffsets` and `LatOffsets` are cleared and hidden, while in all other cases they are shown and the text in their label is set to the name of the coordinate they represent in the selected frame for the pointing offsets.

`void SDSAntenna::setTrackLimits(float la, float le, float ua, float ue)`
This slot sets the minimum and maximum accepted values for both azimuth and elevation while tracking a source. Its arguments respectively are the minimum azimuth value, the minimum elevation value, the maximum azimuth value and the maximum elevation value.

Virtual public slots

`virtual void SDSAntenna::antennaPark()`
This slot has to be implemented to composing the necessary commands to park the antenna and transferring the parking command queue to `SDSCommThread *commThread`.

`virtual void SDSAntenna::antennaUnpark()`
This slot has to be implemented to composing the necessary commands to unpark the antenna

and transferring the parking command queue to `SDSCommThread *commThread`.

`virtual void SDSAntenna::checkAntennaStatus()`
This slot has to be implemented for performing all necessary runtime system checks.

`virtual void SDSAntenna::getAntennaParameters(QString parString)`
This slot has to be implemented for reading the string that contains the antenna system parameters and for storing the parameters' values in the `SDSEdit` objects placed in `QWidget auxiliaryWindow`.

`virtual QString SDSAntenna::getReceiverConfigCommand(int recIndex)`
This slot has to be implemented for building and returning all necessary commands for placing the receiver in focus and accordingly configuring the antenna and the receiver itself. Its argument is the index for the receiver code that identifies it in the `SDSCombo ReceiverCombo` items list.

`virtual void SDSAntenna::readReceiverLine(QString rsl)`
This slot has to be implemented for reading the setup line `QString rsl` that contain the parameters for selecting the receiver and setting its parameters.

`virtual void SDSAntenna::updateSourceParameters(QString parString)`
This slot has to be implemented for reading the source parameters string `QString parString` that comes from `SDSCatalogue *sourceCatalogue` after the selection of a source, accordingly to the syntax implemented in the fully implemented `SDSCatalogue` subclass.

Protected slots

`void SDSAntenna::checkTrackLimits()`
This slot checks the current antenna's azimuth and elevation against their limits for tracking a source. Signal `void SDSAntenna::azimuthLimitReached()` is emitted if the antenna reaches either limit in azimuth, while signal `void SDSAntenna::elevationLimitReached()` is emitted if the antenna reaches either limit in elevation.

`void SDSAntenna::configureReceiver()`
This slot performs the task for placing the selected receiver in focus and accordingly configuring the antenna. It gets first the receiver index from `SDSCombo ReceiverCombo`, then calls slot `virtual QString SDSAntenna::getReceiverConfigCommand(int)` for obtaining all necessary commands, transfers these commands to `SDSCommThread *commThread` for being effectively
sent to the antenna system, calls slot `virtual void SDSAntenna::buildReceiverInfos()` for storing the receiver's parameters in member `SDSReceiver receiver`, and emits the signal `void SDSAntenna::receiverParameters(&receiver)` for passing these informations to other objects.

`void SDSAntenna::updateAntennaStatus()`
This slot sets in members `SDSEdit Longitude` and `SDSEdit Latitude`, the longitude and latitude values in the coordinate system indicated in `SDSCombo CoordSys`, then sets in members `SDSEdit LongOffset` and `SDSEdit LatOffset` the longitude and latitude offset values in the coordinate system indicated in `SDSCombo OffsetsFrame`. It has to be used with due caution in order to avoid conflicts with the requested coordinate values and offsets.

Virtual protected slots

`virtual void SDSAntenna::selectReceiver(int rxi)`
This slot has to be implemented for showing in the main widget all necessary objects for configuring the receiver selected in `SDSCombo ReceiverCombo`. Its argument `rxi` is the index for the receiver identification string in the `ReceiverCombo` items list. A signal/slot connection between this slot and the `void SDSCombo::newIndex(int)`, emitted by `SDSCombo ReceiverCombo`, ensures that this slot is called whenever the item index changes in `SDSCombo ReceiverCombo`.

Signals

`void SDSAntenna::azimuthLimitReached()`
This signal is emitted whenever the antenna reaches either limit in azimuth.

`void SDSAntenna::currentPointingStatus(QString)`
This signal is devoted to passing to other object the current antenna pointing status.

`void SDSAntenna::currentSystemStatus(QString)`
This signal is devoted to passing to other object the current antenna system status.

`void SDSAntenna::elevationLimitReached()`
This signal is emitted whenever the antenna reaches either limit in elevation.

`void SDSAntenna::receiverParameters(SDSReceiver *)`
This signal is devoted to passing to other object a reference to a `SDSReceiver` object containing the name and all configuration parameters for the selected receiver.

`void SDSAntenna::sourceParameters(SDSSource *)`
This signal is devoted to passing to other object a reference to a `SDSSource` object, whose members contain the name and the coordinates of the selected source.

`void SDSAntenna::trackingFailure()`
This signal is meant for being emitted whenever a serious tracking problem occurs.

Established signal/slot connections

Signal `void QPushButton::clicked()`, emitted by member `QPushButton ConfigureReceiver`
to slot `virtual void SDSAntenna::configureReceiver()`
This connection allows the manual call to slot `virtual void SDSAntenna::configureReceiver()` when button `QPushButton ConfigureReceiver` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton Park`
to slot `virtual void SDSAntenna::antennaPark()`
This connection allows the manual call to slot `virtual void SDSAntenna::antennaPark()` when button `QPushButton Park` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton Unpark`
to slot `virtual void SDSAntenna::antennaUnpark()`
This connection allows the manual call to slot `virtual void SDSAntenna::antennaUnpark()` when button `QPushButton Unpark` is clicked.

Signal `void SDSCombo::newIndex(int)`, emitted by member `SDSCombo CoordSys`
to slot `void SDSAntenna::changeCoordSys(int)`
This connection ensures that all necessary tasks are performed when the coordinate system is changed in member `SDSCombo CoordSys`.

Signal `void SDSCombo::newIndex(int)`, emitted by member `SDSCombo OffsetsFrame`
to slot `void SDSAntenna::setOffsetsFrame(int)`
This connection ensures that all necessary tasks are performed when the coordinate system for the pointing offsets is changed in member `SDSCombo OffsetsFrame`.

Signal `void SDSCombo::newIndex(int)`, emitted by member `SDSCombo ReceiverCombo`
to slot `virtual void SDSAntenna::selectReceiver(int)`
This connection ensures that the parameters objects are displayed for the selected receiver whenever the selected item changes in member `SDSCombo ReceiverCombo`.

Signal `void SDSCombo::newValue(QString)`, emitted by member `SDSCombo CoordSys`
to slot `void SDSAntenna::newCoordSys(QString)`
This connection allows the immediate propagation of the newly set source coordinate system, as it's given to `SDSCombo CoordSys` as its new value.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit Latitude`
to slot `void SDSAntenna::newLatitude(QString)`
This connection allows the immediate propagation of the newly set source latitude coordinate, as it's given to `SDSEdit Latitude` as its new value.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit Longitude`
to slot `void SDSAntenna::newLongitude(QString)`
This connection allows the immediate propagation of the newly set source latitude coordinate, as it's given to `SDSEdit Longitude` as its new value.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit PointingStatusMn`
to slot `void SDSEdit::setValue(QString)` for member `SDSEdit Status`
This connection ensures that the antenna pointing status is immediately displayed in the `SDSAntenna` main widget.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit SourceName`
to slot `void SDSAntenna::newSourceName(QString)`
This connection allows the immediate propagation of the newly set source name, as it's given to `SDSEdit SourceName` as its new value.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit SystemStatusMn`
to slot `void SDSEdit::setValue(QString)` for member `SDSEdit SystemStatus`
This connection ensures that the antenna system status is immediately displayed in the `SDSAntenna` main widget.

5.5 SDSBackend class

5.5.1 Class overview

Class `SDSBackend` is the base class for building objects that control a backend. This class is inherited from class `SDSDevice` and is implemented in those functionalities that are common to any backend. The large number of objects in a `SDSBackend` widget suggested the implementation of a default graphic organization of this object. Figure 5.4 displays the adopted layout.

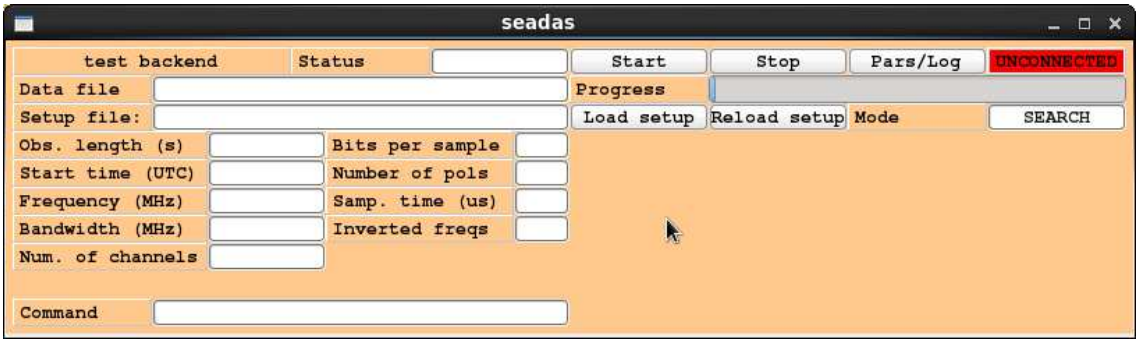

Figure 5.4: `SDSBackend` widget default layout.

All necessary parameters for the data acquisition can be divided in two groups. The first group contains those parameters that are present in the configuration of any backend, which can be called *backend independent parameters*, while the second group contains those parameters that are specific for a given backend, since they are strictly related to the way a given backend processes the signal, hence can be called *backend custom parameters*.

For each *backend independent parameter*, a dedicated `SDSValue` object has been declared as member of a `SDSBackend` object. These parameters are:

01) `SDSEdit Mode`: the backend data acquisition mode;

02) `SDSEdit ObsLength`: the data acquisition length, in seconds;

03) `SDSEdit ObsStartHour`: the data acquisition `UTC` start time;

04) `SDSEdit Frequency`: the central value in MHz for the frequency band;

05) `SDSEdit Bandwidth`: the width in MHz for the frequency band;

06) `SDSEdit ChannelWidth`: the width in MHz for each frequency channel;

07) `SDSEdit NumberOfChannels`: the number of frequency channels;

08) `SDSEdit InvertedFreqs`: a parameter for indicating if the sky frequency band is flipped after the signal downconversion process;

09) `SDSEdit NumberOfPols`: the number of polarizations to represent in the data;

10) `SDSEdit DimensionOfSamples`: same as above, inserted for lexical consistency with some backends;

11) `SDSEdit BitsPerSample`: the number of bits for representing each data sample;

12) `SDSEdit SamplingTime`: the sampling time, in microseconds;

13) `SDSEdit SubintTime`: the subintegration length, in seconds (`pulsar folding` mode only);

14) `SDSEdit ProfileBins`: the number of bins for representing a pulsar profile (`pulsar folding` mode only);

Informations about the receiver, the observing session, the source parameters and the telescope are respectively stored in members `SDSReceiver receiver`, `SDSSession session`, `SDSSource source` and `SDSTelescope telescope`.

Member `SDSEdit DataFile`, displaying text `Data file`, is a non editable `SDSEdit` object for displaying the name of the data file, member `QProgressBar AcquisitionProgress` is a progress bar for displaying the data acquisition progress, introduced at its right by member `QLabel AqPrgLb`, displaying text `Progress`.

`SDSDevice` objects have been placed in the upper part of the widget, with the only exception of `SDSEdit ManualCommand`displaying text `Command`, which is placed near the widget bottom left corner. Objects for managing the *backend independent parameters* find place in the left side of the widget, while the right side, which is empty in figure 5.4, is devoted to hosting the *backend custom parameters*. Member `SDSLog deviceLog` has been located in `QWidget auxiliaryWindow`.

5.5.2 Technical description

The implementation for class `SDSBackend` as a subclass of `SDSDevice` has been obtained through the following steps:

a) the declaration of the `SDSValue` objects for managing the values for all *backend independent parameters*, and of two boolean variables for storing the backend status by means of boolean values;

b) the implementation of slots declared virtual in the parent `SDSDevice` class;

c) the implementation of slots devoted to perform tasks that are necessary for any backend;

d) the declaration of virtual slots that have to be implemented in `SDSBackend` subclasses;

e) the declaration of opportune signals;

`SDSValue` objects devoted to the management of *backend independent parameters* have already been introduced in the previous section. Two boolean variables, namely members `bool isIdle` and `bool isRecording`, respectively store by means of boolean values the backend's *idle* and *recording* status.

`SDSDevice` virtual slots for which an implementation has been done in class `SDSBackend` are the following ones.

Slot `void SDSBackend::enableDevice(bool bl)` (public) is implemented for establishing and closing the socket connection to the backend's server, accordingly to its argument's boolean value: if `bl = true` the connection is established, if `bl = true` the connection is closed;

Slot `void SDSBackend::readSetupLine(QString stl)` (public) has been implemented for setting the value for all *backend independent parameters*, if the setup line `QString stl` contains the keyword for any of these parameters, or calling `virtual void SDSDevice::readCustomLine(QString sl)` in the opposite case.

Slot `void SDSBackend::socketError(QAbstractSocket::SocketError)` (public) has been implemented for displaying in `SDSLog deviceLog` messages that report the occurred error for `QTcpSocket SDSCommThread::*socket`. Errors for which a message has been planned are:

0 Connection refused or timed out

1 Connection closed by remote host

2 Host not found

5 Timeout error

7 Network error

-1 Unidentified error

Items in this list are numbered accordingly to the integer value that correspond to the related item in `enum QAbstractSocket::SocketError`.

Slot `void SDSBackend::socketState(QAbstractSocket::SocketState)` (public) has been reimplemented for reacting in case of changes in the state for `QTcpSocket SDSCommThread::*socket`. States for which actions have already been implemented are:

0 Connection closed

3 Connection established

Again, items in this list are numbered accordingly to the integer value that correspond to the related item in `enum QAbstractSocket::SocketState`. In both implemented cases, slot `SDSStatusButton::setEnabled(bool)` is called for member `SDSStatusButton deviceAct`, the boolean value for member `bool isEnabled` is set accordingly to the new state of the socket

connection, and a message about the new socket state is displayed in `SDSLog deviceLog`. The boolean value assigned to the argument of slot `setEnabled(bool)` and to member `bool isEnabled` is for both `true` if the connection has been established, else `false`.

Fully implemented slots introduced in class `SDSBackend` are the following ones.

Slot `void SDSBackend::setBackendMode(QString bkmode)` (public) is devoted to set the `SDSBackend` object accordingly to the given data acquisition mode `QString bkmode`. Three modes have been implemented, namely `baseband`, `pulsar fold` and `pulsar search` modes. The slot's argument has to take one of the following values, accordingly to the selected mode: `BASEBAND`, `FOLD` and `SEARCH`. This slot shows in the `SDSBackend` widget those parameters, among the *backend independent* ones, which are meaningful for the selected mode, and hides all others. It also calls backend's mode's dedicated virtual slots, which have to be implemented for setting the backend's custom functionalities accordingly to the selected mode.

Slots `void SDSBackend::setIdle()` and `void SDSBackend::setRecording()` set to `true` the boolean value stored in members `bool isIdle` and `bool isRecording` respectively, and to `false` the other mentioned.

Four slots have been implemented here for setting in any `SDSBackend` object informations about the receiver, session, source and telescope parameters. These slots update the values stored in the related information member, but only when the backend is idle. These slots are:

1) `void SDSBackend::setReceiver(SDSReceiver *rec)` updates the receiver's informations stored in `SDSReceiver receiver`;

2) `void SDSBackend::setSession(SDSSession *ssn)` updates the session's informations stored in `SDSSession session`;

3) `void SDSBackend::setSource(SDSSource *src)` updates the source informations stored in `SDSSource source`;

4) `void SDSBackend::setTelescope(SDSTelescope *tel)` updates the telescope's informations stored in `SDSTelescope telescope`;

Virtual slots declared in this class are the ones devoted to setting the backend's custom functionalities for the selected data acquisition mode, namely `virtual void SDSBackend::setBasebandMode()` for the `BASEBAND` mode, `virtual void SDSBackend::setFoldMode()`, for the `pulsar FOLD` mode, and `virtual void SDSBackend::setSearchMode()`, for the `pulsar SEARCH` mode.

One signal has been declared in this class, namely `void SDSBackend::dataAcquisitionTerminated()`. This signal is meant to be emitted whenever the data acquisition terminates, regardless of the reason.

The building of a fully implemented object devoted to manage a specific backend needs the subclassing of class `SDSBackend` and, in particular, the following steps in the given order:

1) class `SDSCommThread` is subclassed so that all its virtual slots can be implemented, again accordingly to the backend specific functionalities;

2) member `SDSCommThread *commThread` is recasted to the fully implemented `SDSCommThread` subclass;

3) slot `void SDSDevice::createConnections()` is called for establishing all necessary signal/slot connections between `*commThread` and its parent object;

4) all `SDSDevice` and `SDSBackend` virtual slots are fully implemented accordingly to the backend's specific functionalities.

Inheritance

Inherits from: `SDSDevice`

Public members

`QLabel AqPrgLb`
This member is the label for providing a short description for member `QProgressBar AcquisitionProgress`.

`QProgressBar AcquisitionProgress`
This member is the progress bar for displaying the data acquisition progress.

`SDSEdit Bandwidth`
This member is the `SDSEdit` object for setting and displaying the frequency band width (`MHz`).

`SDSEdit BitsPerSample`
This member is the `SDSEdit` object for setting and displaying the number of bits for representing each data sample.

`SDSEdit ChannelWidth`
This member is the `SDSEdit` object for setting and displaying the width in MHz of each frequency channel.

`SDSEdit DimensionOfSamples`
This member is the same as `SDSEdit NumberOfPols` (see below), provided for consistency to the terminology of some backends.

`SDSEdit DataFile`
This member is the not editable `SDSEdit` object for displaying the name of the data file.

`SDSEdit Frequency`
This member is the `SDSEdit` object for setting and displaying the central value in MHz for the frequency band.

`SDSEdit InvertedFreqs`
This member is the `SDSEdit` object for setting and displaying the flag that indicates whether the frequency order has been flipped (value `YES`) or not (value `NO`) after the signal down conversion.

`bool isIdle`
This member stores by means of a boolean value the backend's idle status. Its value can be set by slot `void SDSBackend::setIdle()`, and it's equal to `true` if the backend is idle, otherwise to `false`.

`bool isRecording`
This member stores by means of a boolean value the backend's recording status. Its value can be set by slot `void SDSBackend::setRecording()`, and it's equal to `true` if the backend is recording, otherwise to `false`.

`SDSEdit Mode`
This member is the `SDSEdit` object for setting and displaying the data acquisition mode.

`SDSEdit NumberOfChannels`
This member is the `SDSEdit` object for setting and displaying the number of channels the frequency band has been subdivided into.

`SDSEdit NumberOfPols`
This member is the `SDSEdit` object for setting and displaying the number of polarizations to represent in the data. Its values can be `1 = total intensity`, `2 = amplitude only for both polarizations`, `4 = amplitude and phase for both polarizations`.

`SDSEdit ObsLength`
This member is the `SDSEdit` object for setting and displaying the observation length, in seconds.

`SDSEdit ObsStartHour`
This member is the `SDSEdit` object for setting and displaying the desidered `UTC` time for the start of the data acquisition.

`SDSEdit ProfileBins`
This member is the `SDSEdit` object for setting and displaying the number of bins for representing the pulsar profile (`pulsar folding` mode only).

`SDSReceiver receiver`
This member is the `SDSReceiver` object for storing all receiver's parameters. It can be set by calling slot `void SDSBackend::setReceiver(SDSReceiver *rec)`.

`SDSEdit SamplingTime`
This member is the `SDSEdit` object for setting and displaying the signal sampling time in μs (`pulsar search` mode only).

`SDSSession session`
This member is the `SDSSession` object for storing all basic session's information. It can be set by calling slot `void SDSBackend::setSession(SDSSession *ssn)`.

`SDSSource source`
This member is the `SDSSource` object for storing the name and the coordinates for the source to observe. It can be set by calling slot `void SDSBackend::setSource(SDSSource *src)`.

`SDSTelescope telescope`
This member is the `SDSTelescope` object for storing all basic informations about the telescope. It can be set by calling slot `void SDSBackend::setTelescope(SDSTelescope *tel)`.

`SDSEdit SubintTime`
This member is the `SDSEdit` object for setting and displaying the subintegration time in seconds (`pulsar folding` mode only).

Public slots

`void SDSBackend::enableDevice(bool bl)`,
reimplemented from `virtual void SDSDevice::enableDevice(bool bl)`
This slot has been implemented for establishing or closing the connection to the backend's server, accordingly to the value of its boolean argument.

`void SDSBackend::readSetupLine(QString stl)`,
reimplemented from `virtual void SDSDevice::readSetupLine(QString stl)` This slot has been implemented for assigning values to all *backend independent parameter*, by analyzing the argument `QString stl`. If such string does not contain the keyword for any parameter belonging to the mentioned group, slot `virtual SDSDevice::readCustomLine(QString stl)` is called for checking whether the string `stl` contains the keyword for a *backend custom parameter* and setting it in case.

`void SDSBackend::setBackendMode(QString bkmode)`
This slot has been implemented for showing, in the `SDSBackend` widget, the `SDSValue` objects that manage those parameters which are meaningful for the selected data acquisition mode. The possible modes are `BASEBAND` for baseband data acquisition, `FOLD` for pulsar folding mode, and `SEARCH` for pulsar search mode. This slot also calls the virtual slot devoted to perform the backend's custom setup required by the selected data acquisition mode: `virtual void`

SDSBackend::setBasebandMode(), virtual void SDSBackend::setFoldMode() or virtual void SDSBackend::setSearchMode(), respectively for the BASEBAND, pulsar FOLD and pulsar SEARCH mode.

void SDSBackend::setIdle
This slot sets to true the value for member isIdle, and to false the value for member isRecording.

void SDSBackend::setReceiver(SDSReceiver *rec)
This slot sets in member SDSReceiver receiver the parameters for the currently configured receiver, by assigning to it the corresponding values stored in the SDSReceiver object pointed by *rec.

void SDSBackend::setRecording
This slot sets to true the value for member isRecording, and to false the value for member isIdle.

void SDSBackend::setSession(SDSSession *ssn)
This slot sets in member SDSSession session the parameters for the currently set session, by assigning to it the corresponding values stored in the SDSSession object pointed by *src.

void SDSBackend::setSource(SDSSource *src)
This slot sets in member SDSSource source the parameters for the currently set source, by assigning to it the corresponding values stored in the SDSSource object pointed by *ssn.

void SDSBackend::setTelescope(SDSTelescope *src)
This slot sets in member SDSTelescope telescope the telescope's parameters, by assigning to it the corresponding values stored in the SDSTelescope object pointed by *ssn.

void SDSBackend::socketError(QAbstractSocket::SocketError se), reimplemented from virtual void SDSDevice::socketError(QAbstractSocket::SocketError se)
This slot has been reimplemented for generating log messages in case of errors occurred in the socket connection managed by member SDSCommThread *commThread. Default messages are

implemented if the occurred error is:

0 Connection refused or timed out

1 Connection closed by remote host

2 Host not found

5 Timeout error

7 Network error

-1 Unidentified error

Items in the list are numbered accordingly to the corresponding index in the `QAbstractSocket::socketError enum` object.

`void SDSBackend::socketState(QAbstractSocket::SocketState ss)`, reimplemented from `virtual void SDSDevice::socketState(QAbstractSocket::SocketState ss)`
This slot has been implemented for reacting in case of a change of the state for the socket connection managed by `SDSCommThread *commThread`. Default actions have been implemented for the following socket states:

0 Connection closed

3 Connection established

Items in the list are numbered accordingly to the corresponding index in the `QAbstractSocket::socketState enum` object. In both the mentioned cases, a log message is generated for informing about the state change. If the new state is `connection established`, the value for member `isEnabled` is set to `true` and slot `void SDSStatusButton::setEnabled(bool bl)` is called, with the same boolean argument, for member `deviceAct`. If instead the new state is `connection closed`, the same actions are performed with the boolean value `false`.

Virtual protected slots

`virtual void SDSBackend::setBasebandMode()`
This slot has to be implemented in `SDSBackend` subclasses for completing the setup of the backend managing object, if the data acquisition mode is `BASEBAND`.

`virtual void SDSBackend::setFoldMode()`
This slot has to be implemented in `SDSBackend` subclasses for completing the setup of the backend managing object, if the data acquisition mode is `FOLD`, i.e. *pulsar folding* mode.

`virtual void SDSBackend::setSearchMode()`
This slot has to be implemented in `SDSBackend` subclasses for completing the setup of the backend managing object, if the data acquisition mode is `SEARCH`, i.e. *pulsar search* mode.

Signals

`void SDSBackend::dataAcquisitionTerminated()`
This signal has been designed for being emitted whenever the backend's data acquisition terminates, regardless of the reason.

Established signal/slot connections

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit Mode`
to slot `void SDSBackend::setBackendMode(QString)`
This connection ensures that the `SDSBackend` object sets itself accordingly to the selected data acquisition mode, when the stored value changes for member `SDSEdit Mode`.

5.6 SDSManager class

5.6.1 Class overview

Class `SDSManager` class provides the object form managing an observing session. An object of this class is hence devoted for setting all session data and mode, calling the opportune setup

and schedule file, managing the list of observations, starting and stopping the session, and keeping the session log history. Similarly to `SDSAntenna` and `SDSBackend` objects, it inherits from `SDSDevice` and a default organization has been implemented for the `SDSManager` widget. The adopted layout is displayed in figure 5.5.

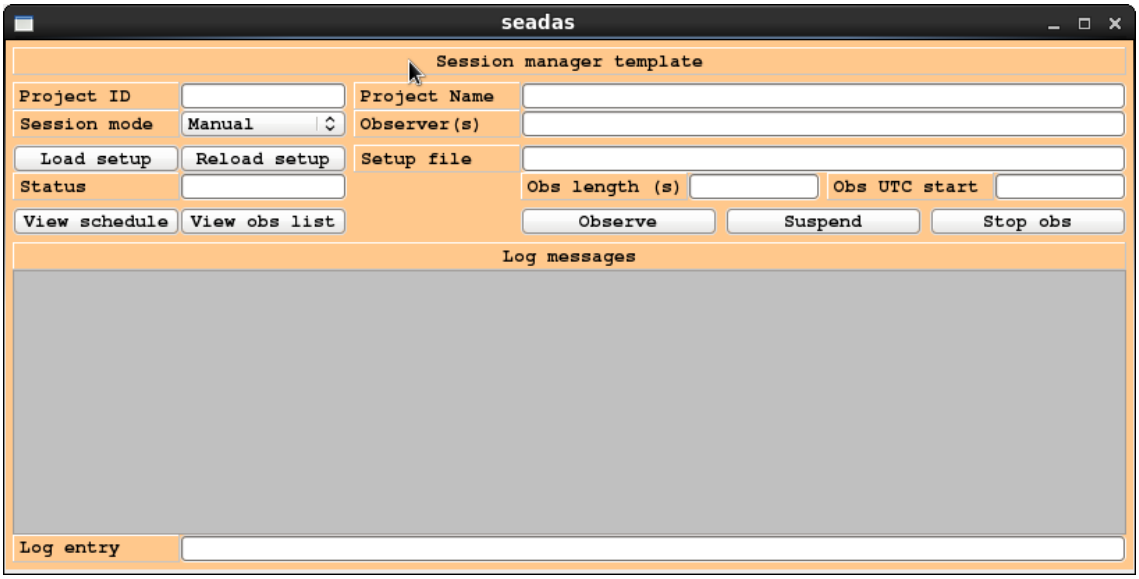

Figure 5.5: SDSManager widget's default layout.

Session data are the *project ID*, i.e. a code assigned to the project by the telescope's *Time Allocation Committee*, the *project name*, i.e. the title of the project, and the name(s) of the observer(s) that are doing the observations. The session mode can be `Manual`, for a single observation, or `Schedule` for an entirely automatized observing session. The `SDSValue` objects that manage these data, respectively `SDSEdit ProjectCode`, `SDSEdit ProjectName`, `SDSEdit ObserverName` and `SDSCombo SessionMode`, are recognizable at the top of the `SDSManager` widget.

The setup file that is loaded by a `SDSManager` object plays the role of a *global setup file*, i.e. in such a file entries are present for loading the configuration for the session, the antenna and all controlled backends. The syntax for setting a parameter is the following one:

```
[keyword] = [value(s)] # [Comment]
```

where `[keyword]` is the keyword that identifies a parameter, `[value(s)]` is the string containing the required value(s), and `[Comment]` is any comment the observer may insert in the setup file. The `#` character is the delimiter between the string containing a setting, at the left, and the

comment, at the right. Any line whose first non blank character is `#` is a commented line. Keywords for the management parameters are:

1) `ProjectID` : the code that identifies the project, usually assigned by the telescope's time allocation committee;
2) `ProjectName` : a short title for the project;
3) `ObserverName` : the name(s) of the observer(s);
4) `Schedule` : the name of the schedule file, indicated accordingly to the prescriptions illustrated in §4.5;
5) `SessionMode` : the session mode, namely `Manual` if manual interaction is desidered, or `Schedule` is a schedule has to be run;
6) `ObsLength` : the length of the observation, in seconds;
7) `ObsStartHour` : the UTC start time of the observation;

Entries for a device have the following syntax:

```
[device identification string] > [keyword] = [value(s)]
```

where `device identification string` is `ANTENNA` for all entries related to the antenna setup, and is the backend identification string *bkstring* (see §5.6.2 for details). At the right side of the `>` character the allowed syntaxes are:

i) an entry for setting up a device parameter, with the same syntax to be used in the device configuration file;
ii) `Setup = [setup file name]` for loading the setup file for the device. The file name has to be indicated accordingly to the prescriptions illustrated in §5.2;
iii) the keyword `ENABLE` for enabling the device;
iv) the keyword `DISABLE` for disabling the device;

The keyword `Source`, whose value(s) are set in the `SDSAntenna` object, can be specified both with and without prepending the string `ANTENNA >`, i.e. the following syntaxes are both valid:

```
ANTENNA > Source = [source parameters]
Source = [source parameters]
```

In `Schedule` mode, the schedule is managed by the window for loading and displaying it, member `SDSScheduleManager schedManager`, and by the window for organizing the observation's list, member `SDSObservationList obsList`. These window can be opened by clicking on the `View schedule` and `View obs list` buttons respectively. Observations in a schedule are organized in single lines, i.e. each line in the schedule file contains all necessary informations for a single observation only. Entries in a schedule line have the same syntax as in the setup file, and are separated by semicolons. Any entry allowed in a setup file is also allowed in a schedule line with the syntax requested by the `global` setup file. Here below is an example for a schedule line:

```
Source = [source parameters] ; Setup = [setup file] ; ObsLength = 7200
```

In the example above it's also present the entry `Setup = [setup file]`, which indicates the setup file to be loaded by the `SDSManager` object. Once the schedule file has been loaded in `SDSScheduleManager schedManager`, single observation lines can be selected by clicking on them with the mouse left button, while a click with the mouse middle button selects the whole displayed schedule. Any selection is displayed in `SDSObservationList obsList` and can be rearranged as indicated in §4.6.

Two `SDSEdit` objects, `SDSEdit ObsLength` and `SDSEdit ObsStartHour`, placed below `SDSEdit SetupFile`, allow to set and display the length, expressed in seconds, and the `UTC` start time, in the `HH:MM:SS` format, for the observation. At their right is placed `SDSEdit Status`, devoted for displaying the observation status.

Buttons displaying text `Observe` and `Stop obs` respectively are `SDSDevice` members `QPushButton Start` and `QPushButton Stop`. The first one starts the single observation or the schedule session, accordingly to the selected session mode, while the second interrupts the current observation and stops the schedule session. Between them button `Suspend`, member `QPushButton Suspend`, is devoted for suspending the schedule mode at the end of the current observation.

In the lower part of the widget it's recognisable member `SDSLog deviceLog`, for managing the session and all devices log messages.

The last `SDSDevice` member present in the default template is `QLabel MainLabel`, located at the top and displaying text `Session manager template`.

5.6.2 Technical description

Class `SDSManager` inherits from class `SDSDevice`, and implements in it all necessary functionalities for managing an observation and an observing session. This class has been designed so that the `SDSAntenna` object that manages the antenna and the `SDSBackend` objects that manage the backends are members of this class.

Member `SDSAntenna *antenna` is the pointer to the object that controls the antenna. Its fully implementation requires:

1) class `SDSAntenna` is subclassed, for fully implementing all antenna functionalities;

2) member `*antenna` is recasted to the fully implemented antenna subclass;

3) public slot `void SDSManager::createAntennaConnections()` is called for establishing the necessary signal/slot connections (see below);

The declaration and fully implementation for the backends objects is a bit involved, because of the requirement of controlling several backends in parallel. For this reason it has been created the `map < QString , SDSBackend * > backend` object, for calling a specific backend by using a `QString` text string, and a vector of pointers to `SDSBackend` objects, `vector vback`, for operating on all backends without knowing a priori the text string that identifies the backend itself. The string that identifies a backend in `map < QString , SDSBackend * > backend` is the same string to be used for identifying the backend in the `SDSManager` setup file. The full implementation of the control of a given backend requires:

1) class `SDSBackend` is subclassed, for fully implementing all functionalities for the backend;

2) an identification string *bkstring* is selected for the backend;

3) the object `backend[`*bkstring*`]` is recasted to the fully implemented `SDSBackend` subclass;

4) slot `void vector::push_back` is called for member `vector vback`, with argument `backend[`*bkstring*`]`, for placing in the backends vector a reference to the backend object;

Once these steps have been done for all backends to be controlled, public slot `void SDSManager::createBackendsConnections()` has to be called for establishing the necessary signal/slot connections between the `SDSManager` object and all backends' objects. Finally, once the antenna object and all backends' objects have been recasted as above, public slot `void SDSManager::connectAntennaToBackends()` has to be called for establishing the necessary signal/slot connections between member `*antenna` and all backends' objects.

Although `SDSAntenna` and `SDSBackend` classes have to be subclassed and fully implemented for having all necessary functionalities, this fact does not require any subclassing for `SDSManager` class. Members for controlling the antenna and the backend have been declared in the header as *public* pointers, but not yet instantiated, and this allows their recasting outside the class.

Session informations are also stored in the information object `SDSSession session`, and are set by slots `void SDSManager::newObserverName(QString nm)`, `void SDSManager::newProjectCode(QString nm)` and `void SDSManager::newProjectName(QString nm)` for the observer(s) name(s), project code and project name respectively. These slots also emit the signal `void SDSManager::sessionParameters(SDSSession *ssn)` with argument a reference to the `SDSSession session` object.

Slot `void SDSManager::readSetupLine(QString sl)` has been reimplemented from `virtual void SDSDevice::readSetupLine(QString sl)` for analyzing each line in the `SDSManager` setup file, accordingly to the syntaxes illustrated in §5.6.1. It also contains a call to `virtual void SDSDevice::readCustomLine(QString sl)`, so that it becomes easy the implementation of further parameters in possible `SDSManager` subclasses.

Slot `void SDSManager::readScheduleLine()` has been implemented for reading the observation's settings in a schedule line, as stored in member `QString scheduleLine`. This slot reads, in the given order, the `SDSManager` setup file, the antenna and backends' setup files, then the remaining entries. Each entry is then analyzed by calling slot `void SDSManager::readSetupLine(QString sl)`, since their syntax is the same as in the `SDSManager` setup file. The only exception is obviously given by the entry that indicates `SDSManager` setup file itself, for which a dedicated implementation has been done.

The core of the `SDSManager` implementation is in those slots and those signal/slot connections that allow the coordination between the antenna's source tracking and the backend data acquisition. Slot `void SDSManager::start()` has been reimplemented from `virtual void SDSDevice::start()` for starting the observation or the schedule session. If the selected session mode is `Schedule`, this slot reads the first line displayed in

SDSObservationList obsList, stores it in QString scheduleLine and calls slot void SDSManager::readScheduleLine(), before calling slot void SDSAntenna::start() for member SDSAntenna *antenna. The value for member bool obsRequest is set to true. Its function is to provide a mean of indicating that an observation is starting and, in particular, that the antenna is slewing to the source coordinates and, once the source tracking starts, backend's data acquisition has to be started. This slot is also responsible for checking whether all observations in the observations list have been done or not. Once the first line in SDSObservationList obsList is stored in QString scheduleLine, a sanity check is done for controlling whether the stored string is empty or not. A sequence of only blank characters is considered an empty string. If such string is empty, a message is generated for informing about the end of the schedule session and member obsRequest is set to false.

Slot void SDSManager::newAntennaStatus(QString nas) is called whenever the antenna pointing status changes, thanks to its connection to signal void SDSValue::newValue(QString) from SDSEdit SDSAntenna::Status. If the new pointing status is nas = TRACKING and obsRequest = true, enabled backends are started by calling slot void SDSAntenna::startBackends(), a slot devoted to this specific task, member bool obsRequest is set to false and member bool isObserving is set to true. This last member allows to indicate, by means of a boolean variable, that an observation is in progress. If instead the new pointing status is nas = SLEWING and isObserving = true, which is the typical situation of a loss of the source tracking, bool isObserving is set to false and backends are stopped.

Slot void SDSmanager::stop() has been reimplemented from virtual void SDSDevice::stop() for interrupting all backends data acquisition, by calling their custom implementation of virtual void SDSDevice::stop(), and, if in schedule mode, by restoring to the observation list the schedule line related to the interrupted observation and setting the value for member bool stopSchedule to true. This last member stores, by means of a boolean variable, the request of the interruption for the schedule session.

Slot void SDSAntenna::backendHasFinished() is called whenever a backend has terminated its data acquisition, thanks to its connection to signal void SDSBackend::dataAcquisitionTerminated() from all implemented backends. This connection is established by calling slot void SDSManager::createBackendsConnections(). If there still is at least one backend whose data acquisition is in progress, nothing happens. If instead all enabled backends are idle, a message is generated for informing about the

completion of the observation. Moreover if the session mode is `Schedule`, if `stopSchedule = true` a second message is generated for informing about the suspension of the schedule, if instead `stopSchedule = false` slot `void SDSManager::start()` is called for starting the next planned observation.

From the point of view of the `SDSManager` widget's graphic properties, slot `void SDSManager::setChildrenStyle(QFont cf,QPalette cp)` has been reimplemented from `virtual void SDSDevice::setChildrenStyle(QFont cf,QPalette cp)` for setting the text font to `QFont cf` and the color palette to `QPalette cp` for members `SDSScheduleManager schedManager` and `SDSObservationList obsList`.

All `SDSDevice` members and virtual function not mentioned so far do not find any implementation or call in this class. Not strictly necessary widget - like members have been hidden, member `SDSCommThread*commThread` has not been recasted and all slots that interact with it are not called anywhere, and not yet cited virtual functions have not been implemented.

Inheritance

Inherits from: `SDSDevice`

Public members

`SDSAntenna *antenna`
This member is a pointer to the `SDSAntenna` object devoted to manage the antenna. Since class `SDSAntenna` is not fully implemented for managing a given antenna, the following steps are required for `SDSAntenna *antenna` to be managed by a `SDSManager` object:

1) class `SDSAntenna` is subclassed, for fully implementing all antenna functionalities;

2) member `*antenna` is recasted to the fully implemented antenna subclass;

3) public slot `void SDSManager::createAntennaConnections()` is called for establishing the necessary signal/slot connections (see below);

Moreover, once all backends have been

fully implemented, slot `void SDSManager::connectAntennaToBackends()` has to be called for establishing all necessary signal/slot connections between `SDSAntenna *antenna` and all fully implemented `SDSBackend` objects.

`map < QString , SDSBackend * > backend`
This object is a map of all fully implemented `SDSBackend` objects versus the strings that identifies each of them, and represents the inclusion as members of all `SDSBackend` objects in `SDSManager` class. The final implementation of all backend's objects require the following steps:

1) class `SDSBackend` is subclassed, for fully implementing all functionalities for the backend;

2) an identification string *bkstring* is selected for the backend;

3) the object `backend[`*bkstring*`]` is recasted to the fully implemented `SDSBackend` subclass;

4) slot `void vector::push_back` is called for member `vector vback`, with argument `backend[`*bkstring*`]`, for placing in the backends vector a reference to the backend object;

5) slot `void SDSManager::createBackendsConnections()` is called for establishing the necessary signal/slot connections to the `SDSManager` object;

Steps 1) to 4) have to be done for each single backend to be controlled, while step 5) has to be done after the implementation of all backends. Moreover, once the antenna object has been fully implemented too, public slot `void SDSManager::connectAntennaToBackends()` has to be called for establishing the necessary signal/slot connections between member `*antenna` and all backends' objects.

`bool isObserving`
This member stores, by means of a boolean variable, the overall telescope situation with respect to the progress observation. `isObserving = true` indicates that the antenna is tracking the source and all required backends are performing their data acquisition, while `isObserving = false` indicates any situation but the above mentioned one.

`SDSEdit ObserverName`
This member is the `SDSEdit` object for setting and displaying the observer(s) name(s). Once its displayed value changes, this information is propagated to all `SDSBackend` objects.

`SDSEdit ObsLength`
This member is the `SDSEdit` object for setting and displaying the observation length, expressed in seconds.

`SDSObservationList obsList`
This member is the `SDSObservationList` object that manages the observation list in `Schedule` mode.

`QPushButton obsListBt`
This member is the button devoted to opening the window that hosts `SDSObservationList obsList`.

`bool obsRequest`
This member stores, by means of a boolean variable, the overall telescope situation with respect to the request of an observation: `obsRequest = true` indicates that the telescope is doing all necessary tasks to set itself up, including its slewing to the source coordinates, before the data acquisition starts.

`SDSEdit ObsStartHour`
This member is the `SDSEdit` object for setting and displaying the observation `UTC` start time, in the `HH:MM:SS` format.

`SDSEdit ProjectCode`
This member is the `SDSEdit` object for setting and displaying the project code, as assigned by the telescope's *Time Allocation Comittee*.

`SDSEdit ProjectName`
This member is the `SDSEdit` object for setting and displaying the project name or short description.

`SDSSession session`
This member is the `SDSSession` object for storing all session's informations.

`SDSCombo SessionMode`
This member is the `SDSCombo` for selecting the session mode, namely `Manual` for a manually controlled observation, and `Schedule` for a schedule managed session.

`SDSScheduleManager schedManager`
This member is the `SDSScheduleManager` object devoted for managing the schedule file.

`QPushButton schedManagerBt`
This member is button devoted to opening the window that hosts `SDSScheduleManager schedManager`.

`bool stopSchedule`
This member indicates, by means of a boolean variable, the suspension of the running schedule. `stopSchedule = true` means that such suspension has been requested, `stopSchedule = false` the opposite case.

`QPushButton Suspend`
This member is the button devoted for calling the suspension of the running schedule at the end of the running observation.

`vector <SDSBackend *> vback`
This member is a `vector` object, whose elements are pointers to `SDSBackend` objects. It allows to perform tasks on all backends objects without knowing their identification string.

Protected members

`QString scheduleLine`
This member stores, in `Schedule` mode, the schedule line that contains the informations about the observation in progress.

Public slots

`void SDSManager::backendHasFinished()`
This slot is called when a backend has terminated its data acquisition. It checks whether all other enabled backends have also terminated their data acquisition for determining whether the observation can be considered completed and, in `Schedule` session mode, starting with the next one.

`void SDSManager::connectAntennaToBackends()`
This slot establishes all necessary signal/slot connections between `SDSAntenna *antenna` and all `SDSBackend` objects, namely:

1) Signal `void SDSAntenna::sourceParameters(SDSSource *)` from member `SDSAntenna *antenna`, to slot `void SDSBackend::setSource(SDSSource *)` for all declared elements of `vector <SDSBackend *> vback`;

2) Signal `void SDSAntenna::receiverParameters(SDSReceiver *)` from member `SDSAntenna *antenna`, to slot `void SDSBackend::setReceiver(SDSReceiver *)` for all declared elements of `vector <SDSBackend *> vback`.

This slot has to be called after member `SDSAntenna *antenna` and all declared elements of `vector <SDSBackend *> vback` have been recasted to the fully implemented subclasses of their base class.

`void SDSManager::createAntennaConnections()`
This slot establishes all necessary signal/slot connections between `SDSAntenna *antenna` and the `SDSManager` object, namely:

1) Signal `void SDSValue::newValue(QString)` from member `SDSEdit SDSAntenna::Status`, to slot `void SDSManager::newAntennaStatus(QString)`

This slot has to be called after member `SDSAntenna *antenna` has been recasted to the fully implemented subclass of `SDSAntenna`.

`void SDSManager::createBackendsConnections()`
This slot establishes all necessary signal/slot connections between the `SDSManager` object and all `SDSBackend` objects, namely:

1) Signal `void SDSBackend::dataAcquisitionTerminated()` from all declared elements of `vector <SDSBackend *> vback`, to slot `void SDSAntenna::backendHasFinished()`

This slot has to be called after all declared elements of `vector <SDSBackend *> vback` have been recasted to the fully implemented subclasses of their base class.

`void SDSManager::newAntennaStatus(QString nas)`
This slot triggers the opportune actions whenever the antenna pointing status changes, accordingly to the overall situation.

`void SDSManager::newObserverName(QString nm)`
This slot sets to `nm` the observer(s) name(s) in member `SDSSession session`, and emits the signal `void SDSManager::emit sessionParameters(&session)` for passing this information to other objects.

`void SDSManager::newProjectCode(QString nm)`
This slot sets to `nm` the project's code in member `SDSSession session`, and emits the signal `void SDSManager::emit sessionParameters(&session)` for passing this information to other objects.

`void SDSManager::newProjectName(QString nm)`
This slot sets to `nm` the project's name in member `SDSSession session`, and emits the signal `void SDSManager::emit sessionParameters(&session)` for passing this information to other objects.

`void SDSManager::readScheduleLine()`
This slot reads the schedule line stored in member `QString scheduleLine` and accordingly sets all observation's parameters.

`void SDSManager::readSetupLine(QString sl)`
This slot analyzes the line `sl` of a setup file and accordingly calls the opportune slot for performing the indicated setting.

`void SDSManager::setChildrenStyle(QFont cf,QPalette cp)`
This slot sets to `QFont cf` and to `QPalette cp` respectively the text font and the color palette for members `SDSObservationList obsList` and `SDSScheduleManager schedManager`.

`void SDSManager::start()`
reimplemented from `virtual void SDSDevice::start()`
This slot starts either a single observation or a schedule session, accordingly to the session mode selected in member `SDSCombo SessionMode`.

`void SDSManager::startBackends()`
This slot triggers the data acquisition for all enabled backends, after having set them in them the informations about the session and the observation's length and `UTC` start time.

`void SDSManager::stop()`
reimplemented from `virtual void SDSDevice::stop()`
This slot stops the data acquisition for all backends that are enabled and still acquiring data and, if the session mode is `Schedule`, stops the session.

`void SDSManager::suspend()`
This slot sets to `true` the value for member `bool stopSchedule`, so that the running schedule session is stopped at the end of the observation in progress.

Signals

`void SDSManager::sessionParameters(SDSSession *)`
This signal passes a reference to an `SDSSession` object for passing to other objects all informations about the observing session.

Established signal/slot connections

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit ObserverName`
to slot `void SDSManager::newObserverName(QString)`
This connection ensures that slot `void SDSManager::newObserverName(QString)` is called whenever the information stored in member `SDSEdit ObserverName` is updated.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit ProjectCode`
to slot `void SDSManager::newProjectCode(QString)`
This connection ensures that slot `void SDSManager::newProjectCode(QString)` is called whenever the information stored in member `SDSEdit ProjectCode` is updated.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit ProjectName`
to slot `void SDSManager::newProjectName(QString)`
This connection ensures that slot `void SDSManager::newProjectName(QString)` is called whenever the information stored in member `SDSEdit ProjectName` is updated.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton Suspend`
to slot `void SDSManager::suspend()`
This connection ensures that slot `void SDSManager::suspend()` is called whenever button `QPushButton Suspend` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton obsListBt`
to slot `void QWidget::show()` for member `SDSObservationList obsList`
This connection ensures that the window hosting member `SDSObservationList obsList` is opened whenever button `QPushButton obsListBt` is clicked.

Signal `void QPushButton::clicked()`, emitted by member `QPushButton schedManagerBt`
to slot `void QWidget::show()` for member `SDSScheduleManager schedManager`
This connection ensures that the window hosting member `SDSScheduleManager schedManager` is opened whenever button `QPushButton schedManagerBt` is clicked.

Signal `void SDSScheduleManager::obsLineSelected(QString)`, emitted by member `SDSScheduleManager schedManager`

to slot `void SDSObservationList::addObsLine(QString)` for member `SDSObservationList obsList`
This connection ensures that slot `void SDSObservationList::addObsLine(QString)` is called whenever a selection is made in the text displayed in member `SDSScheduleManager schedManager`.

Signal/slot connection to be established

Signal `void SDSBackend::dataAcquisitionTerminated()`, emitted by all declared elements of `vector <SDSBackend *> vback`,
to slot `void SDSAntenna::backendHasFinished()`
established by calling slot `void SDSManager::createBackendsConnections()`
This connection has to be established for allowing the `SDSManager` object to determine whether all active backends have terminated their data acquisition hence establishing whether the observation can be considered completed.

Signal `void SDSValue::newValue(QString)`, emitted by member `SDSEdit SDSAntenna::Status`, to slot `void SDSManager::newAntennaStatus(QString)`,
established by calling slot `void SDSManager::createAntennaConnections()`
This connection has to be established for allowing the `SDSManager` object the taking of all opportune decisions in correspondence of a change in the antenna pointing status.

Signal `void SDSAntenna::sourceParameters(SDSSource *)`, emitted by member `SDSAntenna *antenna`, to slot `void SDSBackend::setSource(SDSSource *)` for all declared elements of vector `vector <SDSBackend *> vback`,
established by calling slot `void SDSManager::connectAntennaToBackends()`
This connection has to be established for allowing the `SDSAntenna` object to pass to all backends the informations about the source to be observed.

Signal `void SDSAntenna::receiverParameters(SDSReceiver *)`, emitted by member `SDSAntenna *antenna`
to slot `void SDSBackend::setReceiver(SDSReceiver *)` for all declared elements of `vector <SDSBackend *> vback`,
established by calling slot `void SDSManager::connectAntennaToBackends()`
This connection has to be established for allowing the `SDSAntenna` object to pass to all backends the informations about the selected receiver.