EURIWARE

Référence : RE/OCC/OC2D09025/0A-001/0.4

Architecture specification

SALOME GUI Architecture

Rédaction	Vérification	Approbation
V SANDLER	S ANIKIN	S MOZOKHIN

Date : 07 Juin 2010

SOMMAIRE

1. General information

1.1 Introduction

SALOME GUI module or SUIT (**S**ALOME **U**ser **I**nterface **T**oolkit) represents a user interface framework for SALOME platform. It reflects the last trends of the GUI developments and best practices of the software development. SUIT is developed as a set of packages implementing reusable software components, allowing building multi-scale CAD applications. With SUIT it is possible both to implement new CAD applications (completely independent from SALOME itself) and develop and integrate fully SALOME-compliant modules - “light-weight” (without any CORBA connection, also called “light”) and “full-scale” (distributed, CORBA-based), using C++ or Python programming languages.

Among other important features SUIT proposes flexible, powerful and safe mechanisms of interaction with other SALOME modules (both distributed and “light”), resources management, viewers and selection handling, exception/signals processing, bringing to the top the multi-desktop dockable-windowed user interface which improves usability of SALOME GUI.

This document contains brief description of most important technical issues related to the SALOME GUI architecture design.

1.2 This document

This document specifies the general architecture of SALOME GUI module including interaction with KERNEL, SALOME servers and modules. This document does not specify the functionality of KERNEL module or other SALOME servers and modules.

The terms and abbreviations used in the text of the document are explained in the Glossary.

1.3 Glossary

Application	Class that is a central notion of SALOME GUI library; it actually defines GUI architecture and behavior (2.3).
Configuration file	File that contains SALOME settings and/or user preferences (2.8.2).
⇒ global **configuration file**	Stores global SALOME settings; not modifiable by the user.
⇒ user’s **configuration file**	Stores user preferences and SALOME settings changed by the user during the GUI session.
⇒ section	Represents a group of settings or preferences, e.g. launch parameters, language settings, resources list etc.
⇒ setting/preference	Elementary unit of the **configuration file** representing some particular feature of SALOME GUI (examples: show splash on start-up flag, a language used, background color for some viewer, trihedron size, etc).

Data Model	Represents a portion of data related to some **Module**; contains a tree of **Data Objects** (2.6).
Data Object	Elementary unit of the **Data Model**; implements presentation methods like `name()`, `icon()`, `toolTip()`, etc. (2.6).
Data Owner	Abstract interface for any selectable entity (tree view item, mesh element, geometrical object, table row, etc.) (2.7).
Desktop	Application desktop window. Embeds all the application GUI elements, like menus, toolbars, dockable windows, dialog boxes, etc (2.4).
Displayer	Abstract class that can be re-implemented by SALOME module to handle Show / Hide operations for presentable objects (2.9.4).
Exception & signals handler	Mechanism that implements centralized exception and signals handling inside the GUI session, allowing user a chance to save results of his work if some operation completed abnormally (2.2).
IAPP	**I**nterface **Appl**icative = synonym of **GUI** (graphical user interface).
Log window	Simple output log window (2.12.3).
Library	Shared binary library that implements some functionality. The naming system for libraries is different for different platforms. On Windows, library name looks like `SalomeApp.dll` while on Linux it would mostly named as `libSalomeApp.so`. To avoid misunderstanding, libraries in this document are name without platform-dependent prefix and suffix, for example `SalomeApp`.
Module	Implements a block of custom functions which present particular functionality of the SALOME module (2.5).
Multi-desktop user interface	One separate top-level desktop window is created for each study (2.4).
Multi-document user interface (MDI)	A type of the GUI when single top-level window is used for the application. All the documents (zero or more) can be opened simultaneously. All the documents share the same top-level application window (2.4).
Neutral point	The state of the application, when there is no active module (2.3).
Notebook	A functionality of SALOME that allows operating with named variables (integer, floating point or string values) instead of the direct numerical values. The SALOME Notebook is a part of the Dump Python functionality; its goal is a parameterization of resulting Python scripts (2.14).

Object browser	Reusable GUI element that displays graphically SALOME objects data tree; allows selecting of the objects, invoking of some operations for them via context popup menu, etc. (2.12.1).
PyQt	Free open-source library that provides Python bindings for the Qt toolkit implemented with help of *SIP* (Riverbank Computing Inc).
Python console	Embedded python console that allows executing a Python commands (2.12.2) within the SALOME GUI application.
`QApplication`	Qt application – a part of Qt library: a class that manages the GUI application's control flow and main settings; dispatches events from window system (2.2).
Resource manager	Responsible for parsing of configuration files, loading of translation resources, loading of images and other resource files, reading/modifying of user preferences (2.8).
SALOME module	Software entity which is integrated in the SALOME platform implementing some dedicated services that are needed to reach the general objective of SALOME platform (e.g. Geometry module provides basic functionalities to create, import and edit CAD models).
⇒ C++ **SALOME module**	**SALOME module**, implemented mainly by using C++ language.
⇒ Python **SALOME module**	Python-based **SALOME module** – usually contains no line of C++ code (paragraph 6).
⇒ Distributed **SALOME module**	CORBA-based SALOME module, implementing distributed model. Usually it contains (at least) two libraries: ⇒ Engine library, that implements module functionality; this functionality can be accessed independently from GUI via CORBA interface (and in Python scripts through the Python bindings). ⇒ GUI library, that implements user front-end of the module; this library is integrated to the SALOME GUI and available in the GUI deskop as a set of menu actions, dialog boxes etc.
⇒ "Lightweight" **SALOME module**	**SALOME module** that does not have CORBA engine (paragraph 7).
Selection manager	The key feature of **Selection manager** is synchronization of selection among all GUI elements, which implement selection capability (2.7).
Selection filter	Implements logical check when the objects being selected should meet some predefined conditions (2.7).
Selector	This class hides implementation details of a selection mechanism for particular widget; and **Selection manager**

	interacts with the widget only through the **Selector** interface (2.7).
Session	Class that controls all the life cycle of the SALOME GUI session. It allows to start/stop GUI session; manages Applications objects; provides an access to common **Resource manager** (2.1), etc.
Single document user interface (SDI)	A type of the GUI when single top-level window is used for the application. Only one document can be opened at the moment (2.4).
SIP	**S**imple **I**nterface **P**rotocol – free open-source tool for automatic generation of the Python bindings for C and C++ libraries developed and distributed by Riverbank Computing Ltd.
Study	Represents a document within the GUI and acts as an abstraction layer between actual document data (e.g. remote data available through CORBA) and data presentation in the GUI elements (2.6).
SUIT	**S**ALOME **U**ser **I**nterface **T**oolkit = SALOME GUI module.
SWIG	**S**implified **W**rapper and **I**nterface **G**enerator – free open-source code development tool that designed to make it easy for scientists and engineers to add scripting language interfaces to programs and libraries written in C and C++. In SALOME it is used for generation of the Python wrappings to access GUI functionality from external or internal Python interpreter.
View manager	Handles all the 3D or 2D **View windows** of the given type (OCC, VTK, Plot2d, etc) (2.9).
View model	Responsible for appearance and behavior of the **View window** (provide methods for displaying/erasing of objects, process user actions, like mouse clicks and keystrokes, processes selection of objects in the **View window**, handles popup menus, etc) (2.9).
View window	View frame that embeds 3D or 2D viewer in GUI (2.9).
Visual state	A state of the application GUI, including number and position of the 3D/2D views and their visual properties (background, camera position, etc) and presentation objects. The visual state is recorded and restored by the specific GUI functions (2.15).

2. Major functional parts

The simplified diagram of the SUIT-based SALOME GUI is figured in the APPENDIX 1. This paragraph describes the main components of GUI and interaction between them.

2.1 Session

The SALOME GUI introduces multiple-desktop application interface style. This means that each study has its own top-level desktop window. But SUIT as toolkit library provides a possibility to develop applications with different types of the interface (2.4):

- Single desktop - only one study can be opened at the moment.
- Multi-document interface (usually abbreviated as MDI): one desktop per several studies. This kind of interface was implemented for old versions of SALOME GUI, before version 3.0).
- Multiple-desktop interface - one separate desktop per study. This type of the main application window is chosen for SALOME since version 3.0.

The main class of the GUI library that defines the general behavior of the GUI is **Application** (see paragraph 2.3 below).

The GUI session is represented by `SUIT_Session` class (SUIT package). This class provides a way to start/close **Application** objects. The application library, passed as command-line parameter of the executable (`SUITApp` for standalone session or `SALOME_Session_Server` for distributed session) to its `main()` function, is loaded dynamically by the only `SUIT_Session` class instance, that eliminates need of direct linkage of the executables with specific GUI library and its dependency libraries (see Figure 1). This application library is cached by `SUIT_Session`.

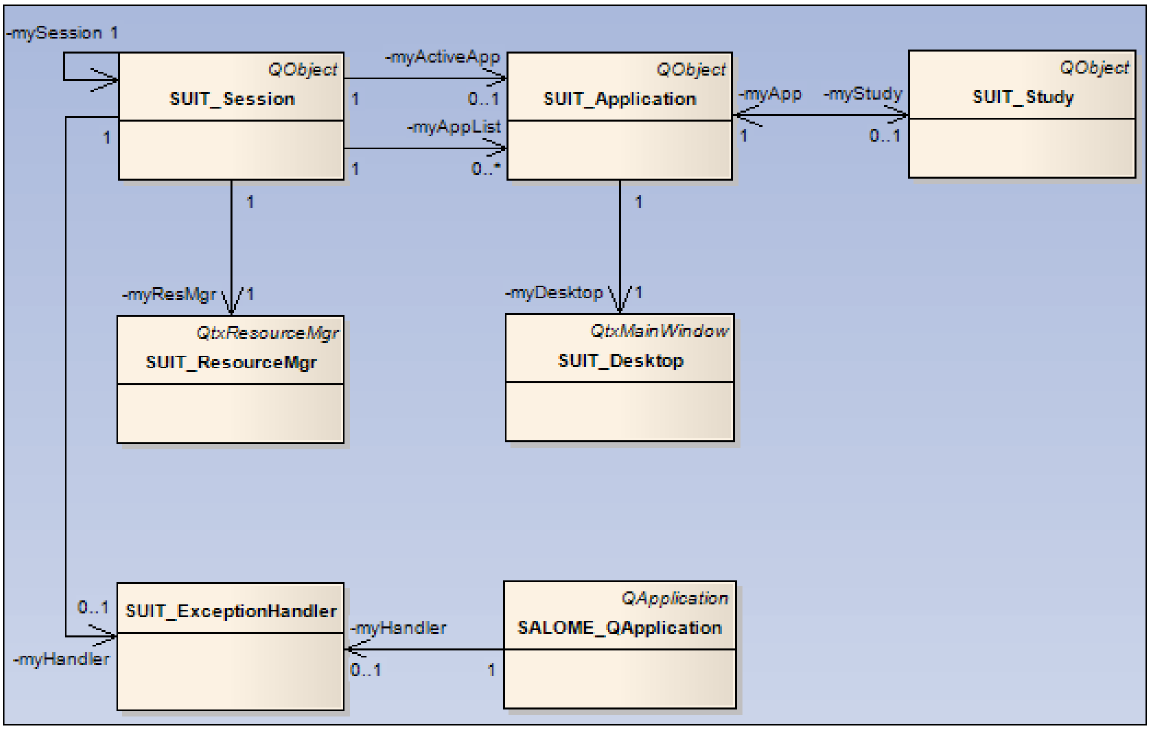


Figure 1. Main GUI classes

On the start-up, `SALOME_Session_Server` (or `SUITApp`) creates the only instance of `SUIT_Session`, passing it as parameter *SalomeApp* library name (implementation of SALOME GUI module for distributed application mode):

```
...
// Create GUI session
SALOME_Session* aGUISession = new SALOME_Session();
// Load SalomeApp dynamic library
SUIT_Application* aGUIApp =
      aGUISession->startApplication("SalomeApp",0,0);
```

```
...
```

In the above fragment of code `SALOME_Session` is a successor of `SUIT_Session` class.
The only instance of the GUI session object is available via the static method `session()` of the `SUIT_Session` class:

```
SUIT_Session* session = SUIT_Session::session();
```

Only one **Application** object can be active at the moment. `SUIT_Session` stores all **Application** objects in the list. Access to the currently active **Application** object is provided by `activeApplication()` method:

```
SUIT_Application* app = SUIT_Session::session()->activeApplication();
```

The full list of the currently existing **Application** instances can be obtained using `applications()` method:

```
QList<SUIT_Application*> apps = SUIT_Session::session()->applications();
```

To finish the GUI session and close all study desktop windows (e.g. with `StopSession()` method of Session CORBA interface) `SALOME_Session_Server` calls `closeSession()` method:

```
SUIT_Session* session = SUIT_Session::session();
session->closeSession(SUIT_Session::DONT_SAVE);
```

In addition, `SUIT_Session` class provides an access to the global **Resource manager** (see 2.8) instance:

```
SUIT_ResourceMgr* resMgr = SUIT_Session::session()->resourceMgr();
```

2.2 Exceptions and signals handling

The main goal of the centralized exception and signal handling is to give the user a chance to save results of his work, if some operation completed abnormally with signal (OS-generated event related to such critical things as memory access violation or floating-point error) or exception not handled locally (in the scope of the corresponding GUI function).

Base interfaces designed in SUIT for exception handling (see Figure 1 above), are:

- `SALOME_QApplication` – class (defined in the Session package) derived from `QApplication` class. Redefines virtual `notify()` method, which is an entry point for all GUI events, such as mouse clicks or keystrokes. This class provides a method `setHandler(SUIT_ExceptionHandler*)` that allows installing custom handler of exceptions/signals.
- `SUIT_ExceptionHandler` – signal and exception handler class (its virtual `handle()` method is called internally by the `SALOME_QApplication::notify()` method).
- `SUIT_Session` – its method `handler()` returns currently used exception handler instance obtained by calling global `getExceptionHandler()` function exported by the application library (if it is present).

The *SalomeApp* library exports `getExceptionHandler()` function, that returns pointer to the `SalomeApp_ExceptionHandler` (a successor of the `SUIT_ExceptionHandler` class) object. This pointer is passed to the `SALOME_QApplication` class instance during the application initialization. The code fragments below demonstrate this technique:

SalomeApp_ExceptionHandler.cxx:

```
extern "C" SUIT_ExceptionHandler* getExceptionHandler()
{
  // boolean flag specifies if it's necessary to handle
  // Floating Point Error signal (SIGFPE)
  return new SalomeApp_ExceptionHandler(true);
}
```

SUIT_Session.cxx:

```
typedef SUIT_ExceptionHandler* (*APP_GET_HANDLER_FUNC)();
...
SUIT_Application* SUIT_Session::startApplication(
                  const QString& name, int /*argc*/, char** /*argv*/)
{
...
  if (!myHandler)
  {
    APP_GET_HANDLER_FUNC crtHndlr = 0;
#ifdef WIN32
    crtHndlr =
(APP_GET_HANDLER_FUNC)::GetProcAddress((HINSTANCE)libHandle,
                                       "getExceptionHandler");
#else
    crtHndlr = (APP_GET_HANDLER_FUNC)dlsym(libHandle,
                                           "getExceptionHandler");
#endif
    if (crtHndlr)
      myHandler = crtHndlr();
  }
...
}

SUIT_ExceptionHandler* SUIT_Session::handler() const
{
  return myHandler;
}
```

SALOME_Session_Server.cxx:

```
bool SALOME_QApplication::notify(QObject* receiver, QEvent* e)
{
  return myHandler ? myHandler->handle(receiver, e):
                     QApplication::notify(receiver, e);
}

SUIT_ExceptionHandler* SALOME_QApplication::handler() const
{ return myHandler; }

void SALOME_QApplication::setHandler(SUIT_ExceptionHandler* h)
{ myHandler = h; }

...
int main(int argc, char **argv)
{
...
  SALOME_QApplication qappl(argc, argv);
  SALOME_Session* session = new SALOME_Session();
  SUIT_Application* app = session->startApplication("SalomeApp",0,0);
  qappl.setHandler(session->handler());
...
}
```

`SalomeApp_ExceptionHandler` is the class that actually implements handling:

- It handles signals by converting them to the `Standard_Failure` OpenCASCADE exception (existing OCC mechanism is used).
- It catches both exceptions of common types (`Standard_Failure` – created as result of signal handling or generated by application code for some other reason, `std::exception`, `SALOME::SALOME_Exception`) and unknown exception types.

As a result, any signal or exception raised during the GUI event processing (either client-side or propagated from some remote service through CORBA) and not handled locally is caught by the handler. Message box informing the user about a critical error is shown, and the user is recommended to save his data immediately and then re-start the application.

2.3 Application

Application is the central notion of SUIT that actually defines GUI architecture and behavior. In the *SalomeApp* library this notion is implemented by `SalomeApp_Application` class (derived from `LightApp_Application`). Its key feature, inherited from the `CAM_Application` class, is modularity (see Figure 2).

The `SalomeApp_Application` class is a part of the *SalomeApp* library. This library is loaded dynamically (as a plug-in) by the `SUIT_Session` at the final stage of application initialization (see 2.1). Therefore, application executable does not depend on the library and can be launched very quickly, providing (if necessary) some visual feedback of its activity to the user (for instance, splash screen displaying start-up progress indicator can be shown).

Application controls all the life-cycle of the software (see Chapter 5). In particular, **Application** deals with the following objects (some of them are described in more detail by the next paragraphs):

- **Module** (derived from either `SalomeApp_Module` or `LightApp_Module`) – encapsulates a block of logically connected functions, implementing the functionality of some **SALOME module** (examples: Geometry, Mesh, Post-Pro). All **Modules** activated by the user are stored in the **Application** instance.
- **Active Module**: the **Module** the user is currently working with in the certain study. One or zero of the available modules can be active at any moment. **Application** implements module activation and deactivation procedures to allow user to switch between available **SALOME modules**. The application state when there is no active module is called hereafter "**neutral point**".
- **Study** (`SalomeApp_Study`, `LightApp_Study` class) – the document containing data, the **Application** works with.
- **Data Model** (`SalomeApp_DataModel`, `LightApp_DataModel` or successors) – represent parts of data specific for some **SALOME module** (for instance, link to OCAF tree or to some mesh model can be used internally).
- **View managers** – implement particular viewer types (OCC, VTK, Plot2d, etc).
- **Selection manager** – an interface that provides the list of objects currently selected by the user and performs selection synchronization between different GUI elements.
- **Resource manager** – provides an access to the different settings and user preferences including pictures and internationalization files.

The `SalomeApp_Application` class implements multi-desktop interface (see 2.4). Separate `SalomeApp_Application` object with attached **Desktop** window is created for each **Study** (document). By default (when no **Module** is active), desktop window contains the following child windows:

- Object browser
- Python console
- Menu bar and standard toolbar

All these child windows are dockable. User may rearrange them according to his preferences, and this state will be restored next time the application is started (user **configuration file** is used to store position of the dockable windows, see paragraph 2.8.2 for more details about configuration files).

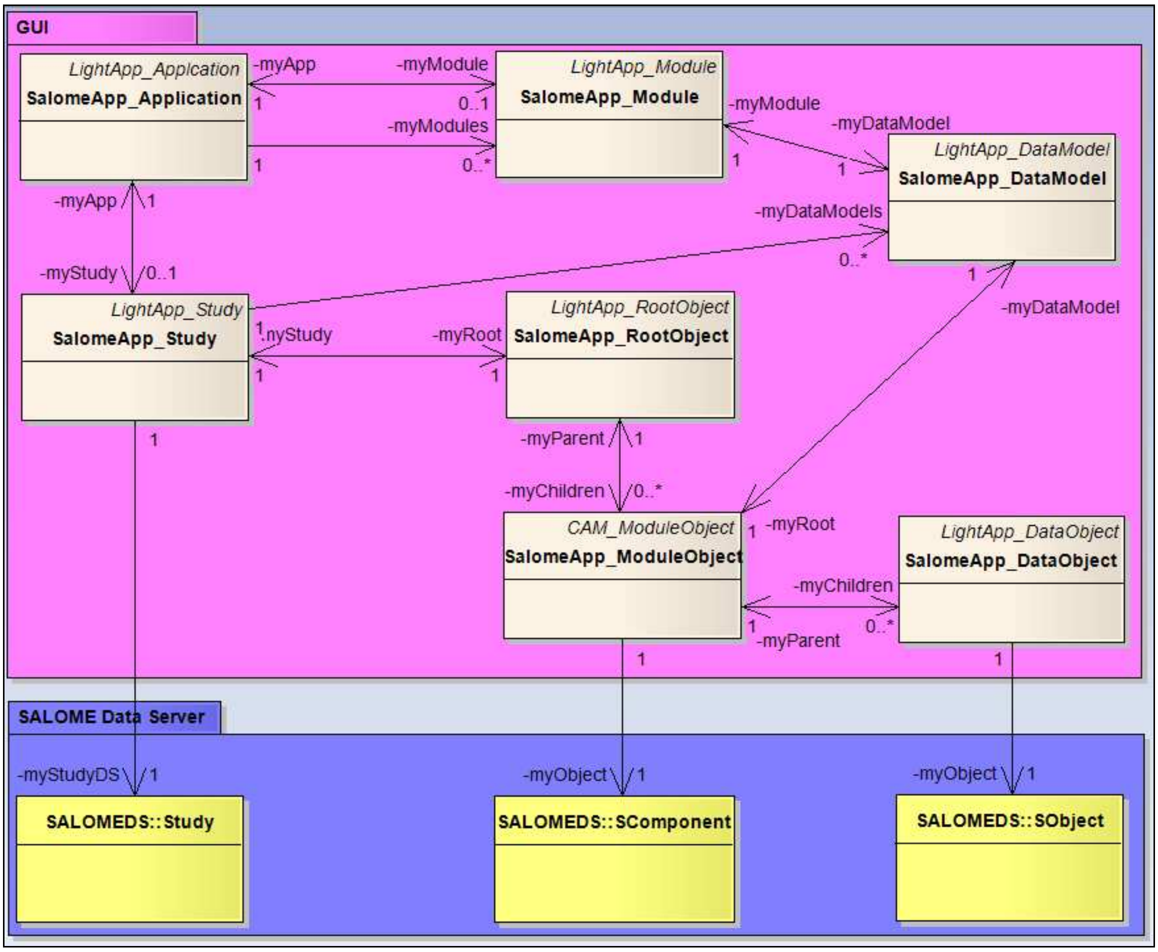


Figure 2. Module architecture

Active study can be obtained from the **Application** by using `activeStudy()` method:

```
SalomeApp_Study* study =
    dynamic_cast< SalomeApp_Study*>(app->activeStudy());
```

To work with modules, SalomeApp_Application class provides the following methods:

```
CAM_Module*   activeModule() const;
CAM_Module*   module(const QString&) const;
ModuleList    modules() const;
void          modules(QList<CAM_Module*>&) const;
void          modules(QStringList&, const bool loaded = true) const;

virtual void addModule(CAM_Module*);
virtual void loadModules();
virtual CAM_Module* loadModule(const QString&,
                               const bool showMsg = true);

virtual bool activateModule(const QString&);
```

```
QString moduleName(const QString&) const;
QString moduleTitle(const QString&) const;
QString moduleIcon(const QString&) const;
bool    isModuleAccessible(const QString&) const;
```

2.4 Desktop

Desktop is the class that defines look-n-feel of the main application window (see Figure 2). SUIT provides several classes which implement different styles of the application main window interface:

- `STD_SDIDesktop` – the simplest desktop class that implements SDI (Single Document Interface) approach, where only one document can be active at the moment and it is embedded into the single desktop window.
- `STD_MDIDesktop` – the class that implement standard MDI (Multiple Document Interface) approach, when the only desktop window is shared between all the documents. Only one document is active at the moment and all the operations (menu commands, etc) are applied to this active document. With such an approach common GUI elements are "shared" between all the documents.
- `STD_TabDesktop` – the class that implements modern multi-desktop approach. With this approach, each opened document has its own separate desktop window.

SALOME GUI is currently using multi-desktop interface. However, custom SUIT-based application can use any of above mentioned classes for implementing of the GUI interface depending on particular application needs.

Desktop object is usually created and attached to the **Application** object at the **Application**'s constructor, e.g.:

```
LightApp_Application::LightApp_Application()
  : CAM_Application(false)
{
  STD_TabDesktop* desk = new STD_TabDesktop();
  setDesktop(desk);
  ...
}
```

Desktop class provides methods to access main menu and toolbar managers associated with the **Application** window (2.10):

```
QtxActionMenuMgr* menuMgr() const;
QtxActionToolMgr* toolMgr() const;
```

Acive view window and full list of all child view windows can be obtained with the corresponding methods (2.9):

```
virtual SUIT_ViewWindow* activeWindow() const;
virtual QList<SUIT_ViewWindow*> windows() const;
```

2.5 Module

Module is the class that actually implements a block of custom functions which represent the functionality of the **SALOME module**.

Depending on the implementation, **SALOME module** can be:

- Regarding the implementation language - binary (written in C++) or scripted (implemented with Python).

- Regarding architecture aspect - distributed (CORBA-based) or standalone (“light”).
- Regarding GUI aspect – with GUI front-end or without it (only engine part, available in batch mode only).

In order to be able to communicate with **Python modules**, a generic *SalomePyQtGUI* (for CORBA-driven) and *SalomePyQtGUILight* (for “light” Python modules) libraries are provided (see paragraph 6 for details about **Python modules**).

SALOME module may or may not have a CORBA engine, depending on the particular needs. If a **module** has CORBA engine, then its GUI library is fully responsible for loading the engine into the CORBA **Container** and activating it. Normally, engine activation is performed on the **Module** initialization (method `initialize()`) or on its first activation (method `activateModule()`).

For those **modules** which do not have CORBA engine, SALOME GUI provides special embedded pseudo-engine in order to simplify persistence operations. Such **SALOME modules** are called “lightweight” (or simply “light”) **modules** (see Chapter 7).

List of the **SALOME modules** available for the current SALOME session is defined through the configuration file (see 2.8 below for details) or specified via the command line parameter of the launching script (`runSalome`, `runLightSalome.sh`).

When module is loaded (it is done automatically on the first activation), its `initialize()` method is called. This method can be used to perform first time initialization of the module. For example, it is a most appropriate place to create menus and toolbars:

```
virtual void initialize(CAM_Application*);
```

On the module activation its `activateModule()` virtual function is called, on the deactivation – `deactivateModule()` method correspondingly:

```
virtual bool activateModule(SUIT_Study*);
virtual bool deactivateModule(SUIT_Study*);
```

Module is also responsible for the creation of the **Data Model**. All SALOME **Modules** should be inherited from the `SalomeApp_Module` class which presents base implementation of **Module** object (see Figure 2) for CORBA-based **modules**, or from `LightApp_Module` class which provides basic implementation for “light” **modules**.

When some **Module** is activated, it can customize the set of dockable windows available to user with help of **Application** services (for instance, if a module required Log window, it can show it when the module activated and hide when it is deactivated). In particular, **Module** may add some specific dockable windows to GUI (they will be hidden by **Application** on **Module** deactivation). And similarly the **Module** defines the list of compatible viewers. For this purpose **Module** class should re-implement the following methods:

```
virtual void windows(QMap<int, int>&) const;
virtual void viewManagers(QStringList&) const;
```

Note: since SALOME GUI implements a multi-desktop user interface (providing a separate desktop per each study) there can be several different **Modules** active at the moment. Each study can have own active **Module** or be at the “**neutral point**”. Switching between the **Modules** is independent process for each study.

Module customizes menus and toolbars according to its functionality. See paragraph 2.9.4 below for more information about menu management. **Module** also may customize the objects selection behavior via installing own **Selectors** to the Object browser and/or **Viewers**. **Module** is also responsible for the context popup menu processing and preferences exporting.

2.6 Study and data model

Study represents a document within GUI and acts as an abstraction layer between actual document data (probably, remote data available through CORBA) and data presentation (in the Object browser). In particular, it contains a tree of **Data Object** instances (see Figure 2).

`SalomeApp_Study` class (derived from `LightApp_Study`) interacts with one or more **Data Model** objects. Each **Data Model** (derived from `SalomeApp_DataModel` base class) represents portion (**Data Object** sub-tree) of data related to some **Module**.

`SalomeApp_Study` class uses *SALOME Data Server* as persistent storage. `LightApp_Study` class does not implement any specific persistence storage – it provides thus a more generic level of the module implementation. As consequence, each “light” module has to implement its own way to save/restore the data. However, `LightApp_Study` provides common-usage HDF-based persistence driver, fully compatible with *SALOMEDS* persistence driver. It means that study saved in the CORBA SALOME session can be opened in the “light” SALOME session and vice versa.

When a **Study** is loaded from the persistent form (via `open()` method invoking `SALOMEDS::StudyManager::Load()` function), it creates **Data Object** structure corresponding to the `SALOMEDS::Study` structure (with named items, icons, references). **Module** data are not loaded yet. However, this structure is sufficient to display study contents in the Object browser.

Then, if there is some active **Module**, or when a **Module** is activated, the **Module** is asked for **Data Model** object, and then **Data Model**’s `open()` method is called.

- If the **Module** has CORBA engine, then `open()` should simply use `SALOMEDS::StudyBuilder::LoadWith()` function to load the **Module**’s data into the CORBA Container where the module’s engine resides, passing object references to `SComponent` and the module engine as arguments.
- Otherwise (for “light” modules), `open()` `method` should invoke `SALOMEDS::StudyBuilder::LoadWith()` function, passing object references to `SComponent` and `SalomeApp::Engine` interface (see 7) as arguments. Then `open()` can load relevant data files using information returned by `SalomeApp_Engine_i::GetListOfFiles()`.

Depending on **Module** needs, **Data Model**’s `open()` method may fill **Data Model** with custom **Data Objects**. Finally, it may replace old **Data Object** corresponding to the **Module**’s sub-tree root in the **Study** with new **Data Model**’s root **Data Object**. Alternatively, **Data Model** may reuse existing sub-tree root as its root **Data Object**, and simply modify existing **Data Objects** by changing required attributes (IORs, for instance).

When **Study**’s `save()` method is invoked, it calls each registered **Data Model**’s `save()` method:

- If corresponding **Module** has CORBA engine and its own persistent files, then regular *SALOMEDS* storage mechanism will be used (**Data Model**’s `save()` should do nothing, it’s a matter of module’s engine to prepare the persistent data and pass this data to the *SALOMEDS* server).
- Otherwise (for “light” modules), **Data Model**’s `save()` method should use `SalomeApp_Engine_i::SetListOfFiles()` to prepare **Module**’s persistent files for putting into a study file.

Finally, `SALOMEDS::StudyManager::Save()` CORBA method is called to create a single HDF file (or other persistent form according to current settings).

It should be noted that **Data Object** tree might have structure different from SALOMEDS study contents. This way **Study** and **Data Models** can hide some portions of data from the user, for instance.

Data Object is a unitary entity representing small portion of data in the **Study**. Main point here is that each **Data Object** represents only that piece of module data that is published (i.e. exported) by the module in the study data tree. In this aspect, data objects are only the presentation attributes of the internal module data. It's a matter of the module what data to export to the study and in what form. For example, some **Data Object** can be linked with the certain data structure in the module's internal data model (like remote CORBA servant object).

As presentable entity, the **Data Object** has different attributes that affect on its behavior and appearance in the Object browser**:** name, entry (unique ID), icon, text color, etc. Some attributes like "entry" or "IOR" have special meaning since they are used to simplify data interaction between modules.

In *SalomeApp* library, the **Data Object** is presented by the `SalomeApp_DataObject` class. This class serves as a GUI interface for the `SALOMEDS::SObject` that represents the same data entity at the level of CORBA study in the data server (*SALOMEDS_Server*).

SALOME modules interact with each other by means of **Data Objects**. To simplify this interaction, each **Data Object** is uniquely identified by the "entry" attribute. In addition, `SalomeApp_DataObject` class provides function to get the value of IOR attribute if it is assigned to the underlying `SALOMEDS::SObject` instance.

2.7 Selection management

One of base services provided by SALOME GUI is common centralized selection mechanism. Its key feature is synchronization of selection among all GUI elements, which implement selection capability.

Main actors of common selection mechanism are (see **Error! Reference source not found.**):

- `SalomeApp_SelectionMgr (derived from SUIT_SelectionMgr)` – class responsible for selection synchronization (selection manager). It maintains a list of `SUIT_Selector` objects and `SUIT_SelectionFilter` objects (see below). Atomic unit of selection is represented by `SalomeApp_DataOwner` class.
- `SUIT_Selector` – specific class should be derived for every widget capable of selection (tree views, viewers, tables, etc.). This class hides implementation details of a particular widget, and selection manager interacts with the widget only through `SUIT_Selector` interface.
- `LightApp_DataOwner` (successor of `SUIT_DataOwner`) – abstract interface for any selectable entity (tree view item, mesh element, geometrical object, table row, etc.). Additionally `LightApp_DataSubOwner` class is used to exchange additional selection information (for example, indices of the mesh elements of selected mesh object).
- `SalomeApp_Filter` (derived from `SUIT_SelectionFilter`) – implements logical check when the objects being selected should meet some predefined conditions.

To connect some widget to the common selection mechanism, its **Selector** object should be created and registered in **Selection manager**.

When selection event occurs in some widget, it should inform its **Selector** object about this. **Selector** object constructs corresponding **DataOwner** objects, puts them to a list and passes the list to the **Selection manager**.

Having been informed by some of its registered **Selector** objects about selection event, **Selection manager** passes the selected **DataOwner** objects to all other registered selectors. Currently installed **Selection filters** are first applied to the new selection in order to ensure that only valid objects pass the selection. **Selection filters** are installed to the **Selection manager** by the **Module** when it is required by some GUI action (e.g. in some dialog box). Each **Selector** can process the list of selected owners according to the logic of corresponding widget (for instance, the owners might be highlighted in a 3D view or Object browser, or corresponding row might get selected in a table, or keyboard focus might be moved to some specific input field, etc.). Finally, **Selection manager** emits `selectionChanged()` signal to inform all application components interested in selection about selection change.

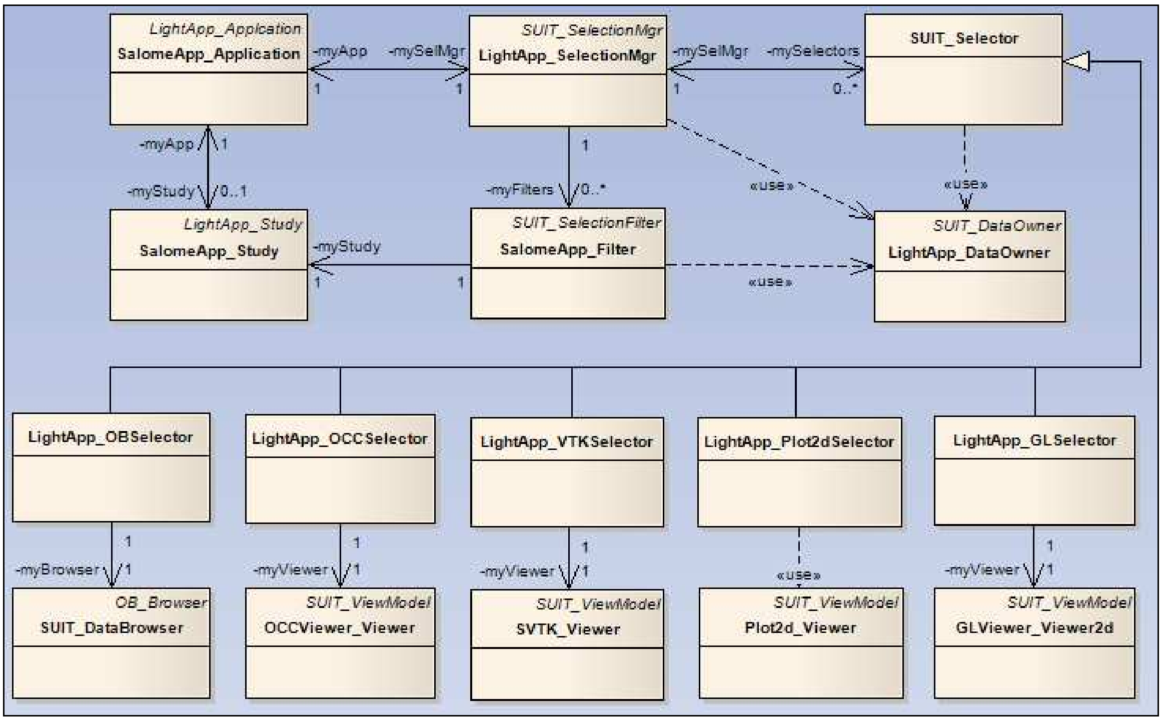


Figure 3. Selection management

At any moment, it is possible to obtain a complete list of **DataOwner** objects currently selected in all registered **Selector** objects through **Selection manager** interface. To fill the list, **Selection manager** asks each **Selector** for its selected owners, and forms a sequence of unique **DataOwner** objects. Uniqueness test is based on the virtual `QString keyString()` method of the **DataOwner** class. For `LightApp_DataOwner` class, for example, this method returns value of the “entry” attribute since it uniquely identifies the **Data Object** inside the **Study**.

SalomeApp library reuses implementations of **Selector** and **DataOwner** interface for common GUI components (Object browser, OCC viewer, VTK viewer, see Figure 3) from LightApp package (which is also an implementation of “light” layer of SALOME GUI). However, any **Module** can replace default any **Selector** by a custom one (using **Selector**’s type), to implement some specific selection logic. For example, if Geometry module requires specific selection behavior for the Object browser items, it can create and install own **Selector** object to the Object browser on the module activation. In such a way, the **Module** is obligatory to remove or deactivate own **Selectors** when it is deactivated.

Presence of common selection mechanism does not prevent custom modules from implementing their own additional selection rules. Moreover, any **Module** might not rely on common selection at all.

2.8 Resources management

2.8.1 Resources manager

Resource manager `SUIT_ResourceMgr` (derived from `QtxResourceMgr` class, a part of Qtx package included into SUIT) is responsible for the following tasks:

- parsing SALOME **configuration files**
- loading Qt translators for GUI resources and maintaining directory lists for images and other resource files search
- Reading/writing user preferences (numbers, colors, boolean flags and other values)

Resource manager (see Figure 1) is initialized when the application is started with the application name and name of environment variable determining list of directories divided by separator symbol - semicolon (';') on Windows or colon (":") on Linux. These folders are then used by **Resource manager** to search the **configuration files**.

For SALOME GUI, this variable's name is *SalomeAppConfig* (*LightAppConfig* for "light" layer of SALOME GUI). Custom SALOME-based application should put the path to its **configuration file** at the left-most position in the directory list specified through this variable.

2.8.2 Configuration files

At the application start-up, the **Resource manager** tries to load several **configuration files**. One of these files is modifiable user configuration file, other files – read-only global files.

The name of the **global configuration file** has form `<application name>.<format>`, the name of **user's configuration file** is different for different operating systems: under Windows the file name is the same as for global configuration files, under Linux it is in form `<application_name>rc.<application_version>`. For example, for SALOME 5.1.3 the files are **SalomeApp.xml** and **.SalomeApprc.5.1.3**.

Resource manager searches **global configuration files** in the folders listed by the variable passed to the constructor, while **user configuration file** is taken from the user's home directory. Note that user's file might not exist at the application start-up, it is automatically created by the **Resource manager** when it saves preferences; the **user's configuration** file is the only file that is re-written and it contains only values changed by the user during the application exploitation. **Global configuration files** are not modifiable by definition; their goal is to specify default values of the application preferences.

Resource manager first loads **user's configuration file** and then – **global configuration files** in the order defined by the list of folders specified in the environment variable. This means that **user's configuration file** has higher priority than **global files**; the values from the **user file** override values defined in the **global files**.

Configuration file may be either in the **INI**-format (textual format, where name of section is rounded by square brackets - `[section_name]` - and contents of section presents one or more lines in form `<name> = <value>`) or in the **XML** format.

When loading the configuration file **Resource manager** uses **current format** (by default, it is the first format in the list of supported formats). Current format can be read and changed by `currentFormat()` and `setCurrentFormat()` methods accordingly. SALOME GUI uses XML format, which is turned on by calling `setCurrentFormat("xml")`.

The structure of XML document must be the following: the root tag is defined by the *"doc_tag"* option, section tag – *"section_tag"* option, parameter tag – *"parameter_tag"* option. The mentioned options can be customized by the application by using `setOption()` method of

Resource manager class. Each section has *name* attribute, and each child of section has *name* and *value* attributes.

In general, the XML document looks like this:

```
<document>
  <section name="section_name">
    <parameter name="parameter_name" value="parameter_value"/>
  </section>
</document>
```

Below is an example of the SALOME configuration file in the XML format (incomplete, only fragment of actual configuration file):

```
<document>
 <section name="desktop" >
  <parameter name="geometry" value="80%x80%+10%+10%"/>
 </section>
 <section name="launch">
  <parameter name="gui"         value="yes"/>
  <parameter name="splash"      value="yes"/>
  <parameter name="file"        value="no"/>
  <parameter name="key"         value="no"/>
  <parameter name="interp"      value="no"/>
  <parameter name="logger"      value="no"/>
  <parameter name="xterm"       value="no"/>
  <parameter name="portkill"    value="no"/>
  <parameter name="killall"     value="no"/>
  <parameter name="pinter"      value="no"/>
  <parameter name="noexcepthandler"  value="no"/>
  <parameter name="modules"     value="GEOM,SMESH,VISU,MED,YACS"/>
  <parameter name="embedded"
             value="SalomeAppEngine,moduleCatalog,study,registry"/>
  <parameter name="standalone"
             value="pyContainer,supervContainer,cppContainer "/>
</section>
 <section name="language">
  <parameter name="language"     value="en"/>
  <parameter name="translators"
             value="%P_msg_%L.qm|%P_icons.qm|%P_images.qm"/>
 </section>
 <section name="resources">
    <parameter name="SUIT"
               value="${GUI_ROOT_DIR}/share/salome/resources/gui"/>
    <parameter name="STD"
               value="${GUI_ROOT_DIR}/share/salome/resources/gui"/>
    <parameter name="Plot2d"
               value="${GUI_ROOT_DIR}/share/salome/resources/gui"/>
    <parameter name="SPlot2d"
               value="${GUI_ROOT_DIR}/share/salome/resources/gui"/>
    <parameter name="GLViewer"
               value="${GUI_ROOT_DIR}/share/salome/resources/gui"/>
    <parameter name="OCCViewer"
               value="${GUI_ROOT_DIR}/share/salome/resources/gui"/>
    <parameter name="VTKViewer"
               value="${GUI_ROOT_DIR}/share/salome/resources/gui"/>
</section>
<section name="SMESH">
  <parameter name="plugins"     value="StdMeshers,NETGENPlugin"/>
 </section>
```

```
 <section name="Style">
  <parameter name="use_salome_style" value="true" />
 </section>
 <section name="ObjectBrowser" >
  <parameter name="auto_hide_search_tool" value="true" />
  <parameter name="auto_size"             value="false" />
  <parameter name="auto_size_first"       value="true"  />
  <parameter name="resize_on_expand_item" value="false"  />
  <parameter name="visibility_column_id_1" value="false"  />
  <parameter name="visibility_column_id_2" value="false" />
  <parameter name="visibility_column_id_3" value="false" />
  <parameter name="visibility_column_id_4" value="false" />
 </section>
 <section name="PyConsole">
  <parameter name="font"                value="Helvetic,12" />
</section>
 <section name="OCCViewer" >
  <parameter name="background"     value="35, 136, 145" />
  <parameter name="iso_number_u"   value="1" />
  <parameter name="iso_number_v"   value="1" />
  <parameter name="trihedron_size" value="100" />
  <parameter name="navigation_mode" value="0"/>
 </section>
 <section name="VTKViewer" >
  <!-- VTK viewer preferences -->
  <parameter name="background"     value="0, 0, 0" />
  <parameter name="trihedron_size" value="105" />
  <parameter name="relative_size"  value="true" />
 </section>
 <section name="MRU" >
  <parameter name="show_mru" value="true"/>
  <parameter name="max_count" value="50"/>
  <parameter name="visible_count" value="7"/>
  <parameter name="insert_mode" value="0"/>
  <parameter name="link_type" value="0"/>
  <parameter name="show_clear" value="true"/>
 </section>
 <section name="FileDlg" >
  <parameter name="QuickDirList" value="${DATA_DIR}" />
  <parameter name="ShowCurDirInitial" value="true" />
 </section>
</document>
```

Settings of each **SALOME module** are defined in the corresponding section of the module's own **configuration file**. For example, all meshing plug-ins for Mesh **module** are listed in the <*plugins*> parameter of the <*SMESH*> section.

The most important sections and parameters of the configuration files are the following:

- Section *launch* defines the parameters of application launching:
  - *gui* : display GUI immediately after application start-up (*"yes"/"no"*); if this parameter is set to *"no"*, it is necessary to call manually the Session CORBA interface's method `GetInterface()` (see 3 and 2.1);
  - *splash* : display splash screen on application start-up (*"yes"/"no"*);
  - *modules*: a list of **modules**, to be used in the SALOME session, separated by comma (",") symbol;
  - *embedded*: a list of embedded SALOME servers/containers to be started, separated by comma;
  - *standalone*: a list of standalone SALOME servers/containers to be started, separated by comma;

  - *xterm*: open separate terminal window for each standalone SALOME server/container;
  - *logger* : start SALOME Logger server;
  - *portkill*: before the starting SALOME kill the servers launched on the current naming service port;
  - *killall*: before the starting new SALOME session kill all currently running sessions.
- Section *resources* defines the list of directories where SALOME resources files are searched (see also next paragraph for more details about resources).
- Section *language* defines the language settings.
- Section *desktop* contains different parameters of the desktop behavior like position, height, width and current state (maximized, minimized, etc) of the main window.
- Section <*module-name*> contains the parameters which refer to the particular SALOME **module**; here the <module-name> is a symbolic name of the SALOME **module**, e.g. **GEOM**, **SMESH**, **VISU**, etc. See paragraphs 4 and 6 below for details.

2.8.3 Working with resource manager

The main section of the **configuration file** is *resources* section. The name is defined by **Resource manager**'s option *"res_section_name"* and if this option is not set, default name "resources" is used.

In order to change the translator of UI text messages, application can use `loadLanguage(prefix, name)` method. Here, `prefix` is the name of setting determining file's folder. This setting is taken from *resources* section. The `name` parameter specifies language to be used for resources, for example "en" means English. If `name` is empty, it is defined by the value of the *language* parameter of the *language* section. The *language* section is defined by the option *"lang_section_name"*. If this option is not set, *language* section's name is "language". If parameter *language* is not found, "en" (English) value is used by default for the application language.

On the application start-up, the resource manager parses the *resources* section of the **configuration files** and for all components and packages mentioned in this section loads the translation files (*.qm) according to the chosen language.

The default name of string translation file is:

```
<path defined by prefix>\<prefix>_msg_<language name>.qm
```

Custom translators can be loaded by defining **Resource manager**'s parameter *translators* of the *language* section. This option contains set of file names templates separated by '|' symbol. In the name template you can use %P, %A, %L macro-substitutions. The resources manager will automatically replace %P "prefix", %A – by application name, %L – by language name. By default**,** application sets *translators* parameter to

"%P_msg_%L.qm|%P_images.qm|%P_icons.qm".

Translator also can be loaded manually by invoking Resource manager's method `loadTranslator(prefix,name)`. Here `prefix` is name of the package; `name` is explicit name of file.

Usually, the name of library or package is used as "prefix". For example, "SMESH" prefix is used for the SMESH **module** resources (**SMESH**_msg_en.qm, **SMESH**_images.qm, etc).

Translators are the part of the Qt application internationalization system. All the translators loaded by the application, are put to the stack. The translators at the top of the stack have higher priority when translating resources than translators from the bottom of the stack. In order to bring some translator to the top of the stack, it's possible to use `raiseTranslators(prefix)` method. To unload the translator from the application, the method `removeTranslators(prefix)` can be used.

The global application **Resource manager** instance can be obtained via static function of the `SUIT_Session` class (see 2.1 and **Figure 1**):

```
SUIT_ResourceMgr* resMgr = SUIT_Session::session()->resourceMgr();
```

To create the pixmap from the graphical image file (in PNG, JPEG or other format supported by Qt), use `loadPixmap(prefix,name)` method of the **Resource manager**, where `prefix` is the name of the package (e.g. **SALOME module** name) and `name` is a file name of the image file:

```
QPixmap px = resMgr->loadPixmap("GEOM","box.png");
```

To find some other resource file use `path(section,prefix,name)` method:

```
QString path(const QString&, const QString&, const QString&) const;
```

2.8.4 Preferences management

Access to the user preferences is also provided via the **Resource manager**. The preferences are automatically loaded when application is starting up and saved when application is closed normally (i.e. not killed by `killSalome.py` script). Each user preference is defined by its section and name. The **Resource manager** provides a lot of methods to read/write preferences:

```
bool    hasSection(const QString&) const;
bool    hasValue(const QString&, const QString&) const;

bool    value(const QString&, const QString&, int&) const;
bool    value(const QString&, const QString&, double&) const;
bool    value(const QString&, const QString&, bool&) const;
bool    value(const QString&, const QString&, QColor&) const;
bool    value(const QString&, const QString&, QFont&) const;
bool    value(const QString&, const QString&,
              QLinearGradient&) const;
bool    value(const QString&, const QString&,
              QRadialGradient&) const;
bool    value(const QString&, const QString&,
              QConicalGradient&) const;
bool    value(const QString&, const QString&, QString&,
              const bool = true) const;

int integerValue(const QString&, const QString&,
                 const int) const;
double doubleValue(const QString&, const QString&,
                   const double) const;
bool booleanValue(const QString&, const QString&,
                  const bool = false) const;
QFont fontValue(const QString&, const QString&,
                const QFont& = QFont()) const;
QColor colorValue(const QString&, const QString&,
                  const QColor& = QColor()) const;
QString stringValue(const QString&, const QString&,
                    const QString& = QString()) const;
QByteArray byteArrayValue(const QString&, const QString&,
           const QByteArray& = QByteArray()) const;
QLinearGradient linearGradientValue(const QString&, const QString&,
           const QLinearGradient& = QLinearGradient()) const;
```

```
QRadialGradient radialGradientValue(const QString&, const QString&,
           const QRadialGradient& = QRadialGradient()) const;
QConicalGradient conicalGradientValue(const QString&, const
   QString&, const QConicalGradient& = QConicalGradient()) const;

void setValue(const QString&, const QString&, const int);
void setValue(const QString&, const QString&, const double);
void setValue(const QString&, const QString&, const bool);
void setValue(const QString&, const QString&, const QFont&);
void setValue(const QString&, const QString&, const QColor&);
void setValue(const QString&, const QString&, const QString&);
void setValue(const QString&, const QString&, const QByteArray&);
void setValue(const QString&, const QString&,
              const QLinearGradient&);
void setValue(const QString&, const QString&,
              const QRadialGradient&);
void setValue(const QString&, const QString&,
              const QConicalGradient&);

void    remove(const QString&, const QString&);
void    removeSection(const QString&);
```

The first parameter of all these methods is a resource section name, and second is a name of the setting.

For string preferences there is a possibility to perform automatic substitution of the environment variables. If the last (boolean) parameter of the `value()` method is *true*, all occurrences of environment variables or setting names will be replaced by its real values. If this parameter is *false*, the method returns pure value with no substitutions made for the environment variables or setting names. Other methods use automatic substitution always.

To substitute the value of some setting in another setting's value, `$(<name>)` syntax should be used. If there is a cyclic reference, for example:

A = $(B);C
B = $(A);D

and `value(“A”)` is called, it returns “$(A);D;C”: result of name substitution does not subject to further substitutions.

In addition, SALOME GUI provides the common preferences edition dialog box. This dialog box has separated tab page for each SALOME **module**. The **Module** class, implementing some SALOME **module** (see 2.4 above), which wants to export any preferences to the common dialog, should redefine `createPreferences()` virtual function and then use `addPreference()` and `setPreferenceProperty()` methods, like in the following example:

```
void GeometryGUI::createPreferences()
{
  int tabId = addPreference(tr(“Settings”));

  int genGroup = addPreference(tr("General"), tabId);
  setPreferenceProperty(genGroup, "columns", 2);

  addPreference(tr("Shading color"), genGroup,
                LightApp _Preferences::Color,
                "Geometry", “shading_color");

  int step = addPreference(tr("Spin box step”)), genGroup,
                           LightApp_Preferences::IntSpin,
                           “Geometry", "SettingsGeomStep");
```

```
  setPreferenceProperty(step, "min", 1;
  setPreferenceProperty(step, "max", 10000);
}
```

In the above sample code, we create tab page *Settings* on the *Geometry* main page (this page is automatically created for each module). Then we create group box *General* on the *Settings* tab page. In the *General* group box, we create *Shading color* color button specifying that this setting should be stored in the parameter *shading_color* of the section *Geometry* of the configuration file. In addition, we create spin box widget (for integer values), named *Spin box step* and specify that this setting should be stored as parameter *SettingsGeomStep* of the *Geometry* section of the configuration file. Also, for this spin box widget we assign constrains – minimum allowed value should be `1` and maximum allowed value is `10000`.

This common *Preferences* dialog box allows editing of such types of settings, like strings, color values, font preferences, integer and floating point parameters, boolean values, etc. Refer to the `LightApp_Preferences` class for more details.

2.9 View management

2.9.1 General description

SALOME GUI provides the flexible way to customize the window's appearance and behavior. Multi-desktop interface (one desktop per study) allows having only one instance of the Object browser, Python console and message output window (called Log window) per study. All these GUI components are implemented as dockable windows and can be easily shown/hidden, e.g. via main menu.

In contrast to the mentioned user components, the 3d/2d views are implemented as child windows of the desktop main window's workspace. The look-n-feel depends on the **Desktop** class used by the **Application** (see 2.4). For `STD_SDIDesktop` and `STD_MDIDesktop` classes all the views occupy an internal working area of the workspace and can be maximized, minimized, tiled (vertically or horizontally) or cascaded to a pile. For `STD_TabDesktop` the views are organized to the single tabbed widget with separate tab page assigned to each view. Several views can occupy the same tab page, in such a case they are stacked inside a page (with only one visible at the moment). It is possible to split tab page horizontally or vertically and move any view to another existing tab page (see Figure 4 and Figure 5).

The view is activated by the clicking its title bar or somewhere in its area (if it is not visible). Usually all the current operations operate with the currently active view. Some operations, however, can automatically activate specific view (or of specific type) if necessary.

User opens 3D/2D view window (OCC, VTK, Plot2d, etc.) by invoking the corresponding item from the main menu *Window* which contains the *New Window* sub-menu with items for each viewer type:

Window
  → *New Window*
    → *OCC view*
    → *VTK view*
    → *Plot2d view*
    → *<other view>*

The list of available 2d/3d view windows is defined by the **Application** class. For example, here is how LightApp_Application class fills in the *New Window* sub-menu:

```
void LightApp_Application::createActions()
```

```
{
  ...
  int windowMenu = createMenu("Window", -1, MenuWindowId, 100);
  int newWinMenu = createMenu("New Window", windowMenu, -1, 0);
  createActionForViewer(NewGLViewId, newWinMenu,
                        QString::number(0), Qt::ALT+Qt::Key_G);
  createActionForViewer(NewPlot2dId, newWinMenu,
                        QString::number(1), Qt::ALT+Qt::Key_P);
  createActionForViewer(NewOCCViewId, newWinMenu,
                        QString::number(2), Qt::ALT+Qt::Key_O);
  createActionForViewer(NewVTKViewId, newWinMenu,
                        QString::number(3), Qt::ALT+Qt::Key_K);
  createActionForViewer(NewQxSceneViewId, newWinMenu,
                        QString::number(4), Qt::ALT+Qt::Key_S);
  ...
}
```

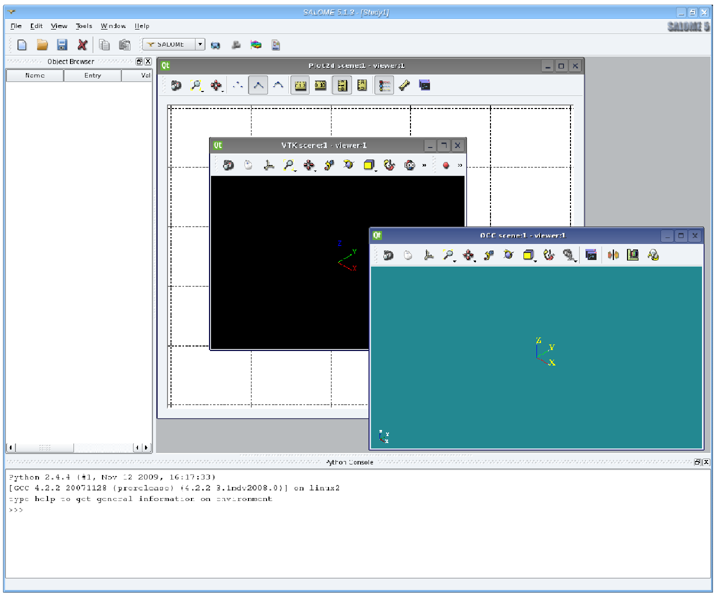


Figure 4. MDI desktop

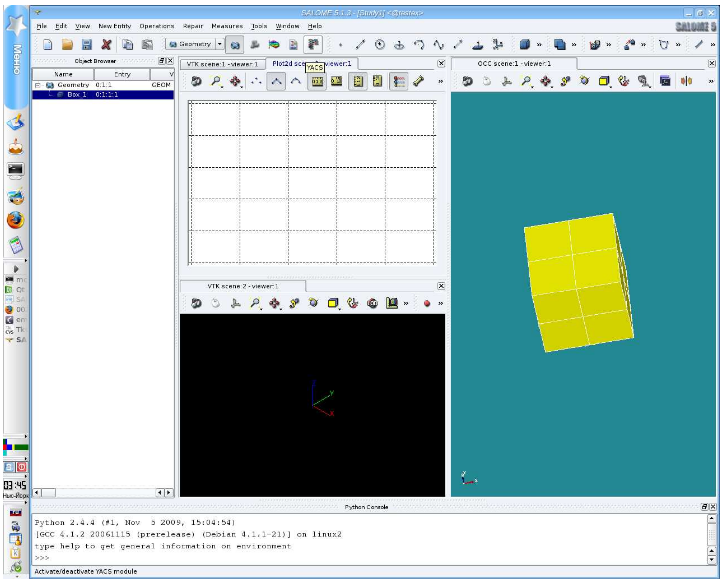


Figure 5. Tabbed desktop

To operate with the viewers, SUIT provides a mechanism of **View managers** (see Figure 6):

- `SUIT_ViewManager` - **View manager** class which handles all the view windows of the given type (OCC, VTK, Plot2d, etc). `SalomeApp_Application` maintains the list of **View managers** to provide access to all the 3D/2D viewers. Also, currently active view manager (that one, which view is currently active) can be also obtained from the **Application** object. `SUIT_ViewManager` creates and owns an instance of `SUIT_ViewModel` (see below).
- `SUIT_ViewModel` – this class is responsible for the appearance and behavior of the viewer (provides methods for displaying/erasing of objects, processes user actions, like mouse clicks and keystrokes, processes selection of objects in the view, handles popup menus, etc). **View model** creates **View windows** (instances of `SUIT_ViewWindow`) by request from the `SUIT_ViewManager`.
- `SUIT_ViewWindow` – base class for all specific view windows implementations; in general it is a view frame which embeds specific 3D or 2D viewer.

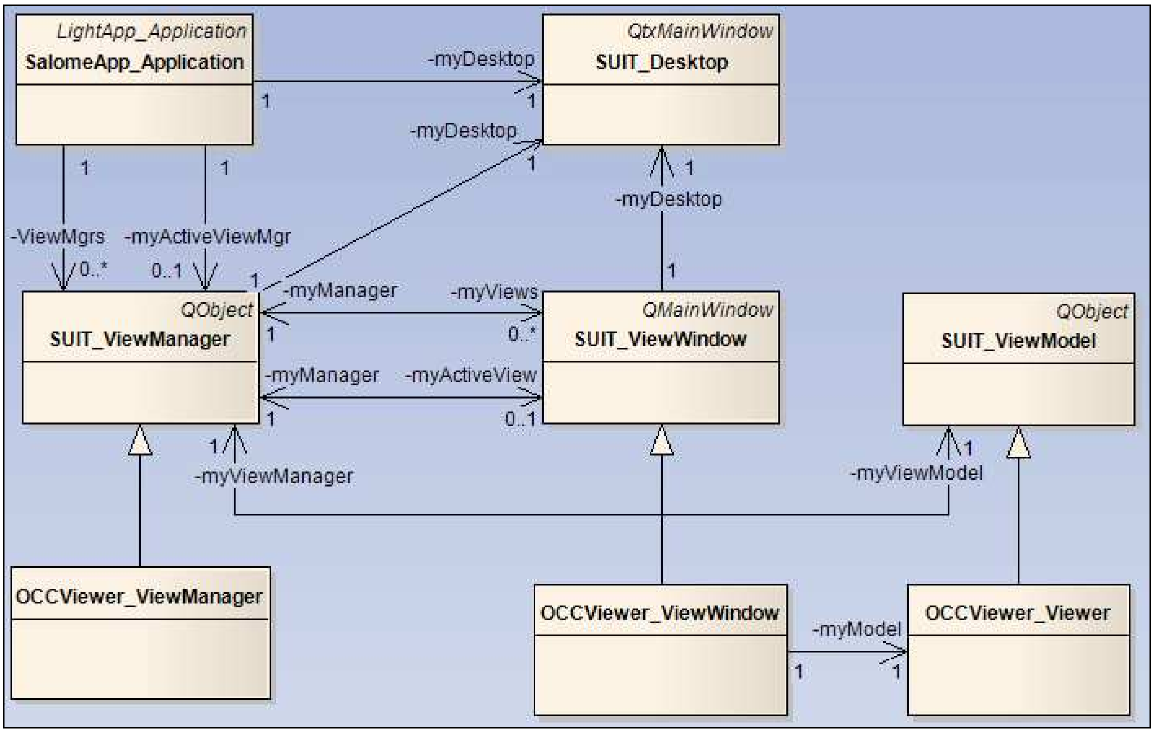


Figure 6. View management

Note that above mentioned classes provide only abstract implementation of the 3D / 2D viewers which should be used as base classes for specific viewer being implemented. SALOME GUI includes several ready-to-use classes (developed in separate source packages) which can be used in the application:

- *OCCViewer*: 3D-viewer, based on the Open CASCADE Technology.
- *VTKViewer*: 3D-viewer based on the Kitware's VTK toolkit.
- *GLViewer*: Common usage OpenGL 2D-viewer, based on the Open CASCADE Technology.
- *Plot2d*: Qwt-based 2D-viewer designed to display 2d curves and point sets.
- *QxScene*: 2D viewer used for supervision schemes in YACS module

Any viewer (OCC, VTK, Plot2d, etc) implements its own view management by inheriting the classes `SUIT_ViewManager`, `SUIT_ViewModel` and `SUIT_ViewWindow` (only OCC viewer classes are shown on Figure 6).

To create new view window there is a `createView()` method of the `SUIT_ViewManager` class. To retrieve an active view, method `getActiveView()` can be used.

`SalomeApp_Application` handles the list of the **View managers**. Each **View manager** maintains the list of views, one or zero of which can be active one (the latest activated child view).

`SalomeApp_Application` provides some methods to work with view managers:

- `activeViewManager()` – to get currently active view manager (the view manager, that owns the currently active view window);
- `findViewManager(type)` – to get view manager of given `type` (since several view managers can be registered in the `SALOMEApp_Application` for the same view type, this method returns the first one found);
- `getViewManager(type, create)` – most useful method which creates a view manager of a requested `type` if no appropriate one is found. The new view window is created automatically when necessary.

Usually, the module can operate with views of specific type only. For example, Mesh and Post-Pro modules use VTK viewer for displaying meshes, while Geometry module mainly operates with OCC viewer. Each module can specify a view type (or even list of types) which should be automatically created (if required) and activated when the module is activated by the user. For this purpose, the module class should inherit and re-implement the following method:

```
void viewManagers(QStringList&) const;
```

2.9.2 Display/Erase operations

Display/Erase operations are performed in the context of the **Module**. Active **Module** customizes main menu and popup menu for objects selected in a view or in the Object browser. When popup menu is created for any data object(s) selected in the viewer (or in the Object browser), active **Module** adds *Display / Display Only / Erase* commands to it. When the command is invoked the **Module** processes it in the corresponding slot. To display any presentable data in the viewer it is necessary first to get the viewer. It is possible to use one of three methods (see previous paragraph):

```
SUIT_ViewManager* activeViewManager();
SUIT_ViewManager* viewManager(const QString& type);
SUIT_ViewManager* getViewManager(const QString& type, bool create);
```

List of the view managers, supported by the application, can be obtained with method:

```
QStringList viewManagersTypes() const;
```

2.9.3 Interactive Object

The management of the presentation in 2D/3D viewers is made by means of interactive objects. SALOME_InteractiveObject class (OBJECT package) represents any presentable object within the 2D/3D viewer. Its main goal is to provide a way of specific data object addressing in the SALOME application. In SALOME, any data object is identified by the component it belongs to and unique identifier, named “entry”. In terms of SALOME data server, entry is a string representation of the object address in the data tree; it’s a sequence of integer number, separated by the colon symbol, for example “0:1:1:1”. In other SALOME-based applications, not using SALOMEDS as data storage, any alternative way to provide unique entry strings can be applied.

```
class SALOME_InteractiveObject
{
  SALOME_InteractiveObject(const char* anEntry,
```

```
                            const char* aComponentDataType,
                            const char* aName);

  void setEntry(const char* anEntry);
  const char* getEntry();
  Standard_Boolean hasEntry();

  void setName(const char* aName);
  const char* getName();

  virtual Standard_Boolean isSame(
        const Handle(SALOME_InteractiveObject)& anIO);

  Standard_Boolean hasReference();
  const char* getReference();
  void setReference(const char* aReference);

  void setComponentDataType(const char* ComponentDataType);
  const char* getComponentDataType();
  Standard_Boolean isComponentType(const char* ComponentDataType);
};
```

2.9.4 Displayer

Displayer is an abstract class that can be re-implemented by SALOME module to handle show / hide operations for presentable objects. It provides methods used to show, hide the object in the 3D/2D view, check if the object is currently displayed in specified view, and check if selected object can be displayed by the module in the specific view:

```
void Display(const QString& entry, const bool update,
             SALOME_View* view);
void Redisplay(const QString& entry, const bool update);
void Erase(const QString& entry, const bool forced,
           const bool update, SALOME_View* view);
void EraseAll(const bool forced, const bool updateViewer,
              SALOME_View* view) const;
bool IsDisplayed(const QString& entry, SALOME_View* view) const;
void UpdateViewer() const;
bool canBeDisplayed(const QString& entry) const;
```

If the module implements own **Displayer** type, it has to override it from LightApp_Displayer class and implement the following methods:

```
virtual bool canBeDisplayed(const QString& entry,
                            const QString& viewer_type) const;
virtual SALOME_Prs* buildPresentation(const QString& entry,
                                      SALOME_View* view);
```

The method `canBeDisplayed()` should return *true* if the object with specified `entry` can be displayed in the view of specified `type`.
The method `buildPresentation()` is used to create new presentation for the object with the specified `entry` in the chosen `view` window.

The Displayer object is exported by the module with the virtual `displayer()` method which should be re-implemented by the module:

```
virtual LightApp_Displayer* displayer();
```

2.9.5 OCC viewer

OCC viewer is based on the Open CASCADE Technology. This kind of viewer is most suitable for displaying of the CAD models, therefore it is a main viewer used by SALOME Geometry (GEOM) module. OCC viewer operates with AIS interactive objects which are displayed on the AIS interactive context (see OCCT documentation). Several view windows can be connected to the same context and, thus, represent the same graphical scene. This kind of viewer supports such view operations as panning, zooming, rotation, fit all, fit area, etc. It is possible to change background color of the scene and dump visible view contents to the image file.

The OCCViewer package provides basic implementation of the OCC 3D viewer:

```
class OCCViewer_Viewer: public SUIT_ViewModel
{
  void getSelectedObjects(AIS_ListOfInteractive& theList);
  void setObjectsSelected(const AIS_ListOfInteractive& theList);
  void setSelected(const Handle(AIS_InteractiveObject)& theIO);

  QColor backgroundColor() const;
  void setBackgroundColor(const QColor&);
  void toggleTrihedron();
  bool isTrihedronVisible() const;
  virtual void setTrihedronShown(const bool show);
  double trihedronSize() const;
  virtual void setTrihedronSize(const double size);

  int interactionStyle() const;
  void setInteractionStyle(const int style);

  void enableSelection(bool isEnabled);
  bool isSelectionEnabled() const;

  void enableMultiselection(bool isEnable);
  bool isMultiSelectionEnabled() const;

  int getSelectionCount() const;
  bool isStaticTrihedronDisplayed();

  bool highlight(const Handle(AIS_InteractiveObject)& IO,
                 bool hilight, bool update);
  bool unHighlightAll(bool update);
  bool isInViewer(const Handle(AIS_InteractiveObject)& IO,
                  bool onlyInViewer);
  bool isVisible(const Handle(AIS_InteractiveObject)& IO);

  void setColor(const Handle(AIS_InteractiveObject)& IO,
                const QColor& color, bool update);
  void switchRepresentation(const Handle(AIS_InteractiveObject)& IO,
                            int mode, bool update);
  void setTransparency(const Handle(AIS_InteractiveObject)& IO,
                       float transparency, bool update);
  void setIsos(const int u, const int v);
  void isos(int& u, int& v) const;
};
```

The SOCC package provides `SALOME_InteractiveObject`-based implementation of the OCC viewer:

```
class SOCC_Viewer: public OCCViewer_Viewer, public SALOME_View
```

```
{
  bool highlight(const Handle(SALOME_InteractiveObject)& IO,
                 bool highlight, bool update);
  bool isInViewer(const Handle(SALOME_InteractiveObject)& IO,
                  bool onlyInViewer);

  void setColor(const Handle(SALOME_InteractiveObject)& IO,
                const QColor& color, bool update);
  void switchRepresentation(const Handle(SALOME_InteractiveObject)& IO,
                            int mode, bool update);
  void setTransparency(const Handle(SALOME_InteractiveObject)& IO,
                       float transparency, bool update);

  virtual void Display(const SALOME_OCCPrs* prs);
  virtual void Erase(const SALOME_OCCPrs* prsnst bool update);
  virtual void EraseAll(const bool update);
  virtual SALOME_Prs* CreatePrs(const char* entry);
  virtual void BeforeDisplay(SALOME_Displayer* d);
  virtual void AfterDisplay (SALOME_Displayer* d);
  virtual void LocalSelection(const SALOME_OCCPrs* prs, const int);
  virtual void GlobalSelection(const bool update) const;
  virtual bool isVisible(const Handle(SALOME_InteractiveObject)& IO);
  virtual void Repaint();
};
```

2.9.6 VTK viewer

VTK viewer is 3D-viewer based on the Kitware's VTK toolkit library. In SALOME, this type of viewer is mostly used in SALOME Mesh (SMESH) and Post-Pro (VISU) modules to display meshes.

The VTKViewer package provides basic implementation of the VTK 3D viewer, introducing a lot of useful classes and utilities.

```
class VTKViewer_ViewWindow : public SUIT_ViewWindow
{
  void setBackgroundColor(const QColor& color);
  QColor backgroundColor() const;

  vtkRenderer* getRenderer();
  VTKViewer_RenderWindow* getRenderWindow();
  VTKViewer_RenderWindowInteractor* getRWInteractor();
  bool isTrihedronDisplayed();

  void Repaint(bool theUpdateTrihedron);
  void onAdjustTrihedron();
  void GetScale(double theScale[3]);
  void SetScale(double theScale[3]);
  void AddActor(VTKViewer_Actor* actor, bool update);
  void RemoveActor(VTKViewer_Actor* actor, bool update);
};
```

```
class VTKViewer_Viewer: public SUIT_ViewModel
{
  void enableSelection(bool isEnabled);
  bool isSelectionEnabled() const;

  void enableMultiselection(bool isEnable);
  bool isMultiSelectionEnabled() const;
```

```
  int  getSelectionCount() const;

  QColor backgroundColor() const;
  void   setBackgroundColor(const QColor& color);
};
```

The SVTK package provides `SALOME_InteractiveObject`-based implementation of the VTK viewer:

```
class SVTK_ViewWindow : public SUIT_ViewWindow
{
  Selection_Mode SelectionMode() const;
  virtual void SetSelectionMode(Selection_Mode ьode);

  virtual void setBackgroundColor(const QColor& color);
  QColor backgroundColor() const;

  bool isTrihedronDisplayed();
  bool isCubeAxesDisplayed();

  virtual void highlight(const Handle(SALOME_InteractiveObject)& IO,
                         bool highlight, bool update);
  virtual void unHighlightAll();
  bool isInViewer(const Handle(SALOME_InteractiveObject)& IO);
  bool isVisible(const Handle(SALOME_InteractiveObject)& IO);
  Handle(SALOME_InteractiveObject) FindIObject(const char* entry);

  virtual void Display(const Handle(SALOME_InteractiveObject)& IO,
                       bool immediatly);
  virtual void DisplayOnly(const Handle(SALOME_InteractiveObject)& IO);
  virtual void Erase(const Handle(SALOME_InteractiveObject)& IO,
                     bool immediatly);
  virtual void DisplayAll();
  virtual void EraseAll();
  virtual void Repaint(bool updateTrihedron);

  virtual void SetScale(double scale[3]);
  virtual void GetScale(double scale[3]);

  virtual void AddActor(VTKViewer_Actor* actor, bool update);
  virtual void RemoveActor(VTKViewer_Actor* actor, bool update);

  virtual void AdjustTrihedrons(const bool forced);
  VTKViewer_Trihedron* GetTrihedron();

  SVTK_CubeAxesActor2D* GetCubeAxes();
  vtkFloatingPointType GetTrihedronSize() const;

  virtual void SetTrihedronSize(const vtkFloatingPointType size,
                                const bool update);
  virtual void SetIncrementalSpeed(const int value, const int mode);
  virtual void SetProjectionMode(const int mode);
  virtual void SetInteractionStyle(const int style);
  virtual void SetSpacemouseButtons(const int btn1, const int btn2,
                                    const int btn3);
};
```

```
class SVTK_Viewer : public SVTK_ViewModelBase, public SALOME_View
```

```
{
  QColor backgroundColor() const;
  void setBackgroundColor(const QColor& color);

  vtkFloatingPointType trihedronSize() const;
  bool trihedronRelative() const;
  void setTrihedronSize(const vtkFloatingPointType size,
                        const bool update);

  int projectionMode() const;
  void setProjectionMode(const int mode);

  int interactionStyle() const;
  void setInteractionStyle(const int style);

  int incrementalSpeed() const;
  int incrementalSpeedMode() const;
  void setIncrementalSpeed(const int value, const int mode);

  int spacemouseBtn(const int btn) const;
  void setSpacemouseButtons(const int btn1, const int btn2,
                            const int btn3);

  void enableSelection(bool isEnabled);
  bool isSelectionEnabled() const;

  void enableMultiselection(bool isEnable);
  bool isMultiSelectionEnabled() const;

  int  getSelectionCount() const;

  void Display(const SALOME_VTKPrs* prs);
  void Erase(const SALOME_VTKPrs* prs, const bool update);
  void EraseAll(const bool update);
  SALOME_Prs* CreatePrs(const char* entry);
  virtual void BeforeDisplay(SALOME_Displayer* d);
  virtual void AfterDisplay(SALOME_Displayer* d);
  virtual bool isVisible(const Handle(SALOME_InteractiveObject)& IO);
  virtual void Repaint();
};
```

2.9.7 Plot2d viewer

Plot2d is a Qwt-based 2D viewer that is applicable for displaying of the point sets and curves at the 2D chart. In SALOME, it is mainly used by Post-Pro (VISU) and Mesh (SMESH) modules. The main GUI source package for this viewer is Plot2d.

For curves representation, the Plot2d package provides `Plot2d_Curve` class. Each curve is represented as a set of the 2D-points and has such visual attributes like line width, marker type and size, color, titles, units. In addition, each 2D curve can be bound to one of two available vertical axes (left or right). Each point is defined by two coordinates (x,y) and optional description.

```
typedef struct
{
  double x;
  double y;
  QString text;
} Plot2d_Point;
```

```
typedef QList<Plot2d_Point> pointList;

class Plot2d_Curve
{
  void      setHorTitle(const QString& title);
  QString   getHorTitle() const;
  void      setVerTitle(const QString& title);
  QString   getVerTitle() const;

  void      setHorUnits(const QString& units);
  QString   getHorUnits() const;
  void      setVerUnits(const QString& units);
  QString   getVerUnits() const;

  void      addPoint(double x, double y, const QString& t);
  void      insertPoint(int, double, double, const QString& t);
  void      deletePoint(int index);
  void      clearAllPoints();
  pointList getPointList() const;

  void      setData(const double* xdata, const double* xdata,
            long length, const QStringList& tdata);
  double*   horData() const;
  double*   verData() const;

  void      setText(const int index, const QString& desc);
  QString   text(const int index) const;

  int       nbPoints() const;
  bool      isEmpty() const;

  void      setAutoAssign(bool enable);
  bool      isAutoAssign() const;

  void      setColor(const QColor& color);
  QColor    getColor() const;

  void      setMarker(Plot2d::MarkerType marker);
  Plot2d::MarkerType getMarker() const;

  void      setLine(Plot2d::LineType type, const int width);
  Plot2d::LineType   getLine() const;
  int       getLineWidth() const;

  void      setYAxis(QwtPlot::Axis axis);
  QwtPlot::Axis      getYAxis() const;

  double             getMinX() const;
  double             getMinY() const;
  double             getMaxX() const;
  double             getMaxY() const;
};
```

The class `Plot2d_ViewFrame` represents 2D-plot area inside the view window and provides a set of methods to operate with curves, to fit the view contents, set/get chart title, show hide legend, show/hide grid, switch to the linear or logarithmic mode (independently for each axis) and other:

```
typedef QList<Plot2d_Curve*> curveList;
```

```
typedef QMultiHash<QwtPlotCurve*, Plot2d_Curve*> CurveDict;

class Plot2d_ViewFrame
{
  void    setTitle(const QString& title);
  QString getTitle() const;
  void    displayCurve(Plot2d_Curve* curve, bool update);
  void    displayCurves(const curveList& curves, bool update);
  void    eraseCurve(Plot2d_Curve* curve, bool update);
  void    eraseCurves(const curveList& curves, bool update);
  int     getCurves(curveList& clist);
  const   CurveDict& getCurves();
  bool    isVisible(Plot2d_Curve* curve);
  void    updateCurve(Plot2d_Curve* curve, bool update);
  void    updateLegend(const Plot2d_Prs* prs);

  void    fitAll();
  void    fitArea(const QRect& area);
  void    fitData(const int mode,
                  const double xMin, const double xMax,
                  const double yMin, const double yMax,
                  const double y2Min, const double y2Max);

  void    getFitRanges(double& xMin, double& xMax,
                       double& yMin, double& yMax,
                       double& y2Min, double& y2Max);

  void    getFitRangeByCurves(double& xMin, double& xMax,
                              double& yMin, double& yMax,
                              double& y2Min, double& y2Max);

  void    setCurveType(int curveType, bool update);
  int     getCurveType() const;
  void    setCurveTitle(Plot2d_Curve* curve, const QString& title);
  void    showLegend(bool show, bool update);
  void    setLegendPos(int pos);
  int     getLegendPos() const;
  void    setMarkerSize(const int size, bool update);
  int     getMarkerSize() const;
  void    setBackgroundColor(const QColor& color);
  QColor  backgroundColor() const;
  void    setXGrid(bool xMajorEnabled, const int xMajorMax,
                   bool xMinorEnabled, const int xMinorMax,
                   bool update);
  void    setYGrid(bool yMajorEnabled, const int yMajorMax,
                   bool yMinorEnabled, const int yMinorMax,
                   bool y2MajorEnabled, const int y2MajorMax,
                   bool y2MinorEnabled, const int y2MinorMax,
                   bool update);

  void    setHorScaleMode(const int mode, bool update);
  int     getHorScaleMode() const { return myXMode; }
  void    setVerScaleMode(const int mode, bool update);
  int     getVerScaleMode() const;

  bool    isModeHorLinear();
  bool    isModeVerLinear();
  bool    isLegendShow();
};
```

While Plot2d package provides base implementation of the curve viewer, SPlot2d source package is an additional super-package above Plot2d that introduces `SALOME_InteractiveObject` handling used in SALOME GUI:

```
class SPlot2d_Curve : public Plot2d_Curve
{
  virtual bool hasIO() const;
  virtual Handle(SALOME_InteractiveObject) getIO() const;
  virtual void setIO(const Handle(SALOME_InteractiveObject)&);

  virtual bool hasTableIO() const;
  virtual Handle(SALOME_InteractiveObject) getTableIO() const;
  virtual void setTableIO(const Handle(SALOME_InteractiveObject)&);
};
```

```
class SPlot2d_Viewer : public Plot2d_Viewer, public SALOME_View
{
  bool isInViewer(const Handle(SALOME_InteractiveObject)& IObject);
  void Display(const Handle(SALOME_InteractiveObject)& IObject,
               bool update);
  void DisplayOnly(const Handle(SALOME_InteractiveObject)& IObject);
  void Erase(const Handle(SALOME_InteractiveObject)& IObject,
             bool update);

  void Display(const SALOME_Prs2d* prs);
  void Erase(const SALOME_Prs2d* prs, const bool update);
  virtual void EraseAll(const bool update);
  virtual void Repaint();
  virtual SALOME_Prs*  CreatePrs(const char* entry);
  virtual void BeforeDisplay(SALOME_Displayer* d);
  virtual void AfterDisplay (SALOME_Displayer* d);
  virtual bool isVisible(const Handle(SALOME_InteractiveObject)& IO);

  SPlot2d_Curve* getCurveByIO(
    const Handle(SALOME_InteractiveObject)& IO, Plot2d_ViewFrame* view);
  Plot2d_ViewFrame* getActiveViewFrame();
  Handle(SALOME_InteractiveObject) FindIObject(const char* Entry);
};
```

2.9.8 OpenGL 2D viewer

This is a common usage OpenGL 2D viewer, based on the Open CASCADE Technology. Not used directly in SALOME but can be appropriate for external SUIT-based applications or new SALOME modules.

2.9.9 Supervision graph viewer

This type of the view window has been used for SALOME Supervision module. Since version 5.1.0 it is no more used and is to be removed in future.

2.9.10 QxScene viewer

The QxScene view is used by the YACS module to display / edit calculation schemas. It embeds on the Qt's QGraphicsView class and, thus, can be used by any other module to display arbitrary 2D graphic objects implemented with the Qt's `QGraphicView`-related functionality:

```
class QxScene_ViewWindow: public SUIT_ViewWindow
```

```
{
  void               setBackgroundColor(const QColor&);
  QColor             backgroundColor() const;

  void               setScene(QGraphicsScene* scene);
  QGraphicsScene*    getScene();
  void               setSceneView(QGraphicsView* sceneView);
  QGraphicsView*     getSceneView();
};
```

2.10 Menu and toolbars management

General idea of redesigned menu and toolbar management is to make it maximally simple and flexible. Therefore, XML-based menu definition files are no longer used.

Application object creates standard menus, such as “File”, “Edit”, “View”, “Tools”, “Window” and “Help”. It also creates a standard toolbar including buttons for common menu commands.

When some **Module** is activated, menus and toolbars are rebuilt completely. **Module** can customize standard menus and toolbars and create its own ones.

There is no support for XML-based menu definition files in SALOME GUI as it was before. Instead all the GUI actions should be hard-coded in the module GUI implementation. The usual place where to do it is the successor of the `SalomeApp_Module` class. This class defines a family of `createAction()`, `createMenu()` and `createTool()` methods in order to create actions and associate them with the main menu, toolbars and popup menus. This can be done in the `initialize()` method of the **Module** (see **Figure 1**).

The created menus and toolbars are not shown immediately after creation. To do it `activate()` method of the **Module** class should call `setMenuShown(true)` and `setToolShown(true)` methods.
And vice versa, `deactivate()` method should hide the menus and toolbars by calling `setMenuShown(false)` and `setToolShown(false)`.

The processing of the context popup menus is possible by two ways. The simplest way is overriding of the `contextMenuPopup()` method, then filling in the popup menu with commands manually according to the current selection and menu context (Object browser, viewer, etc.) and then processing of the chosen action in the corresponding slot.

Another way is the using of the popup menu management mechanism. First it is necessary in the `initialize()` method of the **Module** to create all actions which should be presented in the context popup menu, then push all these actions to the popup manager and define the lexical rule for each action. Each time when the popup menu is requested these rules define, should some action appear in the menu or not and should it be checked on or off (for switching actions).
Then it is necessary to override `createSelection()` method which should create the instance of `SalomeApp_Selection` class (or its successor). This class analyzes the current selection and defines some variables which are engaged in the lexical rules definition.
For example:

```
void VisuGUI::initialize(CAM_Application* app)
{
  ...
  QPixmap aPixmap = VISU::GetResourceMgr()->loadPixmap
                    ("VISU",tr("ICON_IMPORT_MED"));
  createAction(VISU_IMPORT_FROM_FILE, tr("IMPORT_FROM_FILE"),
               QIcon(aPixmap), tr("MEN_IMPORT_FROM_FILE"),
               "", (Qt::CTRL + Qt::Key_I), aParent, false,
```

```
                this, SLOT(OnImportFromFile()));
  QtxPopupMgr* mgr = popupMgr();
  mgr->insert(action(VISU_IMPORT_FROM_FILE), -1, -1, -1);
  mgr->setRule(action(VISU_IMPORT_FROM_FILE),
    "client='ObjectBrowser' and selcount=1 and type='VISU::TVISUGEN'");
  ...
}
```

In the above example the “Import file” action will appear in the popup menu only if the context popup menu is invoked in the Object browser for the VISU component (top-level VISU root) item. The <client> and <selcount> variables are automatically defined by the LightApp_Selection class, while <type> variable is defined by the `VisuGUI_Selection` class methods.

It is also possible to use directly `LightApp_Selection` class instance – it defines a lot of helpful variables, but if its functionality is not enough for the module needs, it is possible to customize the default behavior by implementing the additional class inheriting from `LightApp_Selection` or from `QtxPopupSelection` as it is done for VISU module.

2.11 Event management

In order for VISU module CORBA engine to be able to interact with GUI elements (VTK and Plot2d views) in accordance with Qt and Xlib rules, mechanism of custom events was introduced in SALOME v2.0.0. This mechanism allows execution of custom code (provided through custom object derived from `SALOME_Event` class) in the main GUI thread context, while this code can be invoked by some secondary thread of the Session server process (for instance, by CORBA execution thread).

Apart from flexibility, robustness and some other benefits, the event-based approach leads to some architectural limitations related to the embedded Python console. First of all, event mechanism made it necessary to execute all Python commands and scripts in a special thread, so as not to block main GUI thread and let it keep processing events (`SALOME_Event` objects in particular).

In its turn, moving GUI Python interpreter to a secondary thread results in situations like the following:

Python-based module is active	
Main GUI thread	**GUI Python interpreter thread**
	Execution of a big Python script invoking `SALOME_Event` mechanism begins → Python lock is acquired. For the script to complete, main GUI thread should not be blocked.
User activates “File” > “New study” menu command	
Callback is invoked to inform PyQt module about study creation	
An attempt to acquire Python lock is made → application HANGS UP!	Thread waits for main GUI thread to process some `SALOME_Event` …

To resolve this problem, some alternatives should be studied carefully (some of them may be used in combination with the others):

- During new study initialization, do not use any additional threads and perform new sub-interpreter initialization in the main GUI thread.
- Serialize requests to Python interpreter from main GUI thread with help of special static (application-global) event queue. If some request implies transient arguments (like pointer to a study or to `QMenu`), and Python interpreter is busy at the moment of request arrival, then such requests can be ignored.
- Probably, block user input events when embedded Python interpreter is busy.

Solutions 1 and 2 are supposed to change the example above as follows:

Python-based module is active	
Main GUI thread	**GUI Python interpreter thread**
	Execution of a big Python script invoking `SALOME_Event` mechanism begins → Python lock is acquired. For the script to complete, main GUI thread should not be blocked.
User activates "File" > "New study" menu command	
Request to initialize new Python sub-interpreter is put to the queue.	
Request to call PyQt module method is put to the queue. Event processing continues.	
	Script execution finishes.
	Python event queue is checked. If it is not empty, the first request is retrieved from the queue. Python command is executed.

Nevertheless, **there is a restriction imposed on Python scripts which can be executed in GUI Python console**: these scripts should not create Python threads, so as not to hang up Python interpreter thread.

2.12 Reused GUI elements

Some GUI elements like Object browser, Python console and Message output (Log) window are implemented as separate SUIT packages. As opposite to old versions of SALOME GUI, in new GUI these objects are done as dockable window and thanks to the multi-desktop interface each study window has only one instance of Object browser, python console and message window.

LightApp_Application and its successor SalomeApp_Application provide methods to add arbitrary widgets as dockable windows of the application's desktop. Any class, inerited from Qt's QWidget class, can be inserted as an additional dock widget to the desktop window (see Figure 7):

```
QWidget* getWindow(const int, const int = -1);
QWidget* dockWindow(const int) const;
```

```
void removeDockWindow(const int);
void insertDockWindow(const int, QWidget*);
void placeDockWindow(const int, Qt::DockWidgetArea);
```

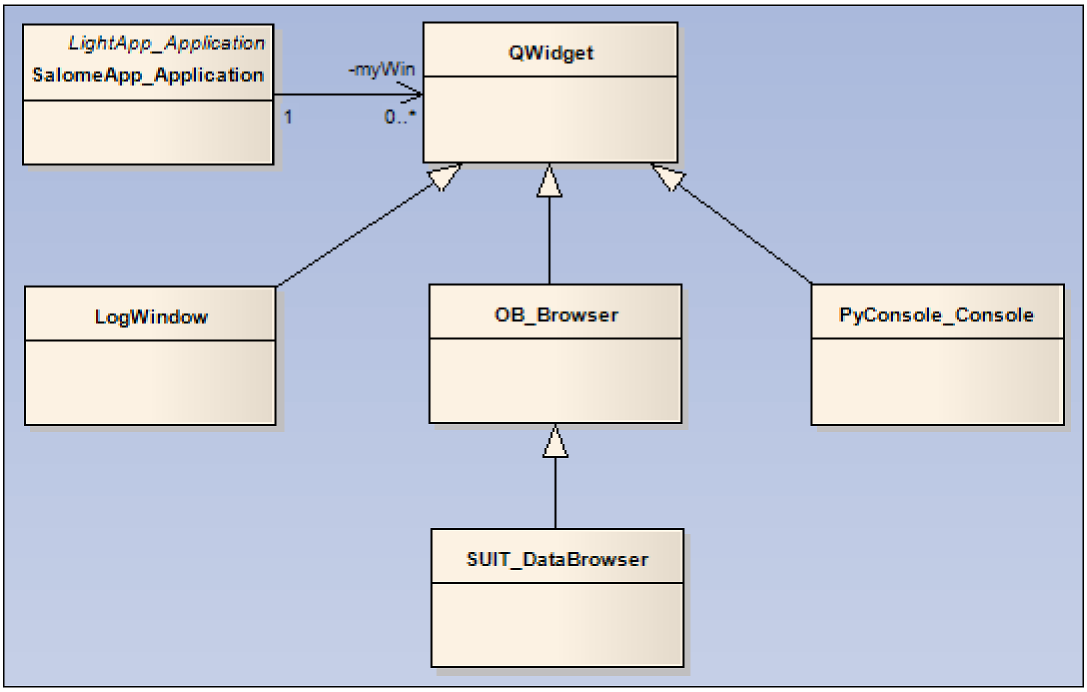


Figure 7. Dockable widgets

2.12.1 Object browser

Object browser is a dockable window which displays study data tree. It is implemented in the ObjBrowser and SUIT packages (see Figure 7).

Since version 5.0 the SALOME Object browser is implementation is based on the Qt's model/view technology. This approach allows displaying the same data (in general, tree-like) in different views in the arbitrary form, depending on specific view need. In addition, Qt provides means to easily sort and filter items displayed in the specific view.

The ObjBrowser package provides basic implementation of the data tree viewer, operating with arbitrary `QAbstractItemModel`-based data model. The tree view widget, used internally, is implemented by the `QtxTreeView` class:

```
class OB_Browser : public QWidget
{
  QAbstractItemModel*    model() const;
  void                   setModel(QAbstractItemModel* m);

  QAbstractItemDelegate* itemDelegate() const;
  void                   setItemDelegate(QAbstractItemDelegate* d);

  bool                   rootIsDecorated() const;
  void                   setRootIsDecorated(const bool enable);

  bool                   sortMenuEnabled() const;
  void                   setSortMenuEnabled(const bool enable);

  QtxSearchTool*         searchTool() const;
  bool                   isSearchToolEnabled() const;
```

```
  void                    setSearchToolEnabled(const bool enable);

  int                     autoOpenLevel() const;
  void                    setAutoOpenLevel(const int auto);
  void                    openLevels(const int level);

  int                     numberOfSelected() const;
  QModelIndexList         selectedIndexes() const;
  const QItemSelection    selection() const;

  virtual void            select(const QModelIndex& i, const bool on,
                                 const bool keepSelection);
  virtual void            select(const QModelIndexList& l, const bool on,
                                 const bool keepSelection);

  bool                    isOpen(const QModelIndex& i) const;
  virtual void            setOpen(const QModelIndex& i, const bool open);

  void                    adjustWidth();
  void                    adjustFirstColumnWidth();
  void                    adjustColumnsWidth();

  unsigned long           getModifiedTime() const;
  void                    setModified();

  QtxTreeView*            treeView() const;
};
```

In SALOME GUI, the elementary unit of the Object browser is represented by `SUIT_DataObject` class. The root object is set by the study – it is a root of study's **Data model** (see 2.6). Then Object browser just iterates through the data structure of the study and displays it in the tree view (see paragraph 2.6 above for more details about data objects).

The class `SUIT_DataBrowser` provides a specific implementation of the Object browser, based on the tree of the `SUIT_DataObject` items. The tree model is implemented by the `SUIT_TreeModel` class:

```
class SUIT_TreeModel : public QAbstractItemModel,
                       public SUIT_AbstractModel
{
  SUIT_DataObject*        root() const;
  void                    setRoot(SUIT_DataObject* r);

  virtual QVariant        data(const QModelIndex& i, int role) const;
  virtual bool            setData(const QModelIndex& i,
                                  const QVariant& data, int role);
  virtual Qt::ItemFlags   flags(const QModelIndex& i) const;
  virtual QVariant        headerData(int section, Qt::Orientation o,
                                     int role) const;

  virtual QModelIndex     index(int row, int column,
                                const QModelIndex& parent) const;
  virtual QModelIndex     parent(const QModelIndex& i) const;

  virtual int             columnCount(const QModelIndex& i) const;
  virtual int             rowCount(const QModelIndex& i) const;
  virtual void            registerColumn(const int group_id,
                                         const QString& name,
                                         const int custom_id);
```

```
  virtual void           unregisterColumn(const int group_id,
                                          const QString& name);
  virtual void           setColumnIcon(const QString& name,
                                       const QPixmap& icon);
  virtual QPixmap        columnIcon(const QString& name) const;
  virtual void           setAppropriate(const QString& name,
                                        const Qtx::Appropriate appr);
  virtual Qtx::Appropriate appropriate(const QString& name) const;

  SUIT_DataObject*       object(const QModelIndex& i) const;
  QModelIndex            index(const SUIT_DataObject* o, int col) const;

  bool                   autoDeleteTree() const;
  void                   setAutoDeleteTree(const bool on);

  bool                   autoUpdate() const;
  void                   setAutoUpdate(const bool on);

  virtual bool           customSorting(const int col) const;
  virtual bool           lessThan(const QModelIndex& left,
                                  const QModelIndex& right) const;

  QAbstractItemDelegate* delegate() const;

  virtual void           updateTree(const QModelIndex& i);
  virtual void           updateTree(SUIT_DataObject* o);
};
```

```
class SUIT_DataBrowser : public OB_Browser, public SUIT_PopupClient
{
  SUIT_DataObject* root() const;
  void             setRoot(SUIT_DataObject* r);

  bool             autoUpdate() const;
  void             setAutoUpdate(const bool on);

  void             updateTree(SUIT_DataObject* o, const bool autoOpen);

  int              updateKey() const;
  void             setUpdateKey(const int key);

  DataObjectList   getSelected() const;
  void             getSelected(DataObjectList& l) const;

  void             setSelected(const SUIT_DataObject* o, const bool on);
  void             setSelected(const DataObjectList& l, const bool on);

  void             setAutoSizeFirstColumn(const bool on);
  void             setAutoSizeColumns(const bool on);
  void             setResizeOnExpandItem(const bool on);
};
```

To work with Object browser `LightApp_Application` class and its successor `SalomeApp_Application` provide the following methods:

- `objectBrowser()` - returns pointer to `SUIT_DataBrowser` class instance;
- `updateObjectBrowser(const bool updateModels = true)` – updates Object browser (does nothing if study is not opened/created). The optional parameter (*true* by default) forces updating of all **Data models** being loaded at the moment.

In most cases there is no need to have a direct access to the Object browser unless you plan to implement own **Selector** class for it.

To get the selected objects from the Object browser use `getSelected()` methods; to set the selection, use `setSelected()` methods.

To get the root entry (in order to iterate through the children) you may use `root()` method, and to update its contents call `updateTree()`.

The Object browser allows two different ways of operating, depending on the value of the `autoUpdate` flag of the `SUIT_TreeModel` class. If the `autoUpdate` flag is set to *true*, the Object browser is automatically updated when new `SUIT_DataObject` is inserted to the tree or when the data object is removed from the tree (destroyed). If the `autoUpdate` flag is set to *false*, the Object browser is not updated automatically, and it is necessary to call its `updateTree()` method (as it is done by the `updateObjectBrowser()` method of the `LightApp_Application` class). The second approach is suitable when the modification of the data tree is performed asynchronously or when the `SUIT_DataObject` tree model is synchronized with another data tree (for example, in SALOME it is a data tree of SALOMEDS study, based on the `SObject` CORBA interface). To optimize the performance of the synchronization process, a special template function `synchronize()` is implemented in `SUIT_TreeSync.h` file that allows effective synchronization of any different data trees. In SALOME, this function is used in `SUIT_TreeModel` class (to synchronize tree of `SUIT_DataObject` items with Qt's tree of the `QModelIndex` entities) and in `SalomeApp_DataModel` class (to synchronize tree of the `SUIT_DataObject` items with the tree of the `SALOMEDS_SObject` items coming from CORBA study stored in SALOME data server).

2.12.2 Python console

The PyConsole package provides an implementation of the embedded Python interpreter window (see Figure 7).

```
class PyConsole_Console : public QWidget, public SUIT_PopupClient
{
  PyConsole_Interp*    getInterp();

  QFont                font() const;
  virtual void         setFont(const QFont& f);

  bool                 isSync() const;
  void                 setIsSync(const bool sync);

  void                 exec(const QString& command);
  void                 execAndWait(const QString& command);
};
```

Method `pythonConsole()` of the `LightApp_Application` class provides an access to Python console window. It returns the pointer to `PyConsole_Console` class.

To execute a Python command in the embedded Python console, use its `exec(const QString& command)` method.

Python console executes Python commands in a secondary thread, to support usage of SALOME events (see 2.11 above).

2.12.3 Message output window

The class `LogWindow`, implemented in the LogWindow source package, provides simple implementation of the message window (see Figure 7) that can be used output different run-time messages. It allows putting simple HTML-formatted text in the message output window, optionally adding the separator in order to mark up different sections of the output log. The separator is a string value and can be changed by `setSeparator()` method. It is possible to save current output log to the file. Usual Copy/Paste/Clear operations are also available through the context popup menu of the window:

```
class LogWindow : public QWidget, public SUIT_PopupClient
{
  QString      banner() const;
  QString      separator() const;

  void         setBanner(const QString& b);
  void         setSeparator(const QString& s);

  void         putMessage(const QString& m, const int mode);
  virtual void putMessage(const QString& m, const QColor& c,
                          const int mode);
  void         clear(const bool clearHistory);

  bool         saveLog(const QString& fileName);
};
```

`SalomeApp_Application` class provides `logWindow()` method to get access to message `LogWindow` class instance. To put the message to the output window it is necessary to call `putMessage()` method of the `LogWindow` class.

2.13 General dialog box class

The LightApp source package includes `LightApp_Dialog` class that can be used to implement specific dialog boxes in some SALOME module. This ready-to-use class implements a big number of helpful methods which allow simplify the implementation of specific dialog boxes for almost any kind of the functionality, providing unified look-n-feel of all the dialog boxes inside the module. The dialog box can be modal or modeless, resizable or no, custom number of the buttons, arranged in a different ways. In addition, `LightApp_Dialog` provides standard widget for the named objects selection; the widget includes text label, line edit and selection button sub-widgets. `LightApp_Dialog` class provides easy way to operate with these objects: add, remove, rename, enable/disable, show/hide, select, find, change pixmap, etc.

```
class LightApp_Dialog : public QtxDialog
{
public:
  bool isExclusive() const;
  void setExclusive(const bool exclusive);

  bool isAutoResumed() const;
  void setAutoResumed(const bool autoResume);

  void showObject(const int id);
  void hideObject(const int id);
  void setObjectShown(const int id, const bool show);
  bool isObjectShown(const int id) const;
  void setObjectEnabled(const int id, const bool enable);
  bool isObjectEnabled(const int id) const;
```

```
  QWidget* objectWg(const int id, const int wgId) const;
  void selectObject(const QString& name, const int type,
                    const QString& id, const bool update);
  void selectObject(const QStringList& names, const TypesList& types,
                    const QStringList& ids, const bool update);
  QString objectText(const int id) const;
  void setObjectText(const int id, const QString& text);

  void selectObject(const int id, const QString& name, const int type,
                    const QString& selid, const bool update);
  void selectObject(const int id, const QStringList& names,
                    const TypesList& types, const QStringList& selIds,
                    const bool update);
  bool hasSelection(const int id) const;
  void clearSelection(const int id);
  void selectedObject(const int id, QStringList& list) const;
  QString selectedObject(const int id) const;
  void objectSelection(SelectedObjects& objs) const;

  void activateObject(const int id);
  void deactivateAll();

protected:
  int createObject(const QString& label, QWidget* parent, const int id);
  void setObjectPixmap(const QPixmap& pixmap);
  void setObjectPixmap(const QString& section, const QString& file);
  void renameObject(const int id, const QString& name);
  void setObjectType(const int id, const int type1, ...);
  void setObjectType(const int id, const TypesList& types);
  void addObjectType(const int id, const int type1, const int, ...);
  void addObjectType(const int id, const TypesList& types);
  void addObjectType(const int id, const int type);
  void removeObjectType(const int id);
  void removeObjectType(const int id, const TypesList& types);
  void removeObjectType(const int id, const int type);
  bool hasObjectType(const int id, const int type) const;
  void objectTypes(const int id, TypesList& types) const;
  QString& typeName(const int type);
  const QString typeName(const int type) const;
  virtual QString selectionDescription(const QStringList& names,
           const TypesList& types, const NameIndication ni) const;
  virtual QString countOfTypes(const TypesList& types) const;
  NameIndication nameIndication(const int id) const;
  void setNameIndication(const int id, const NameIndication ni);
  bool multipleSelection(const int id) const;

  void setReadOnly(const int id, const bool on);
  bool isReadOnly(const int id) const;
};
```

2.14 Notebook

Since version 5.1.1 SALOME provides Notebook functionality. The SALOME Notebook allows operating with named variables instead of the direct numerical values. The functionality of the Notebook is closely associated with the Dump Python mechanism of SALOME. By defining variables and using them in the script or GUI, the user makes easier parameterization of the Python script.

SALOME GUI provides two classes which should be used in the dialog boxes of some SALOME module for each functionality that allows using Notebook variables: `SalomeApp_IntSpinBox` (for integer values) and `SalomeApp_DoubleSpinBox` (for floating point values). In addition to the usual functionality of the spin box widget, these classes allow user to enter the name of the Notebook variable.

Below is an example of the typical usage of the Notebook spin box classes:

```
bool PrimitiveGUI_BoxDlg::isValid(QString& msg)
{
  bool ok = true;
  ok = GroupDimensions->SpinBox_DX->isValid(msg, !IsPreview()) && ok;
  ok = GroupDimensions->SpinBox_DY->isValid(msg, !IsPreview()) && ok;
  ok = GroupDimensions->SpinBox_DZ->isValid(msg, !IsPreview()) && ok;

  ok = fabs(GroupDimensions->SpinBox_DX->value()) >
            Precision::Confusion() && ok;
  ok = fabs(GroupDimensions->SpinBox_DY->value()) >
            Precision::Confusion() && ok;
  ok = fabs(GroupDimensions->SpinBox_DZ->value()) >
            Precision::Confusion() && ok;
  return ok;
}

bool PrimitiveGUI_BoxDlg::execute(ObjectList& objects)
{
  bool res = false;
  GEOM::GEOM_Object_var anObj;
  GEOM::GEOM_I3DPrimOperations_var anOper =
        GEOM::GEOM_I3DPrimOperations::_narrow(getOperation());

  double x = GroupDimensions->SpinBox_DX->value();
  double y = GroupDimensions->SpinBox_DY->value();
  double z = GroupDimensions->SpinBox_DZ->value();

  anObj = anOper->MakeBoxDXDYDZ(x, y, z);
  if (!anObj->_is_nil() && !IsPreview())
  {
    QStringList aParameters;
    aParameters << GroupDimensions->SpinBox_DX->text();
    aParameters << GroupDimensions->SpinBox_DY->text();
    aParameters << GroupDimensions->SpinBox_DZ->text();
    anObj->SetParameters(aParameters.join(":").toLatin1().constData());
  }
  res = true;

  if (!anObj->_is_nil())
    objects.push_back(anObj._retn());

  return res;
}
```

2.15 Visual State

SALOME allows saving of the application GUI visual state, including number and position of the 3D/2D views and their visual properties (background, camera position, etc) and presentation objects, to the persistence file as an additional data record. This visual state of the GUI is displayed in the data tree as a separate named item and can be restored (activated) at any moment by the user by means of the corresponding GUI action (for example, via context popup menu command). In addition, the user can specify in the application preferences that visual state should be automatically stored at the study saving and restored at the study opening.

The mechanism of the visual state recording/restoring includes several steps. Some of these steps are performed by the SALOME GUI module, other steps should be implemented in each SALOME module that supports this functionality (mainly this refers to the visualization of specific module data).

The main class that implements visual state object is `SalomeApp_VisualState`. It records the data to the study by means of the specific `AttributeParameter` attribute of the underlying SALOMEDS study and the helper class `SALOMEDS_IParameters` which provide methods to store arbitrary named data:

```
interface AttributeParameter : GenericAttribute
{
  exception InvalidIdentifier {};

  void      SetInt(in string ID, in long value);
  long      GetInt(in string ID) raises(InvalidIdentifier);
  void      SetReal(in string ID, in double value);
  double    GetReal(in string ID) raises(InvalidIdentifier);
  void      SetString(in string ID, in string value);
  string    GetString(in string ID) raises(InvalidIdentifier);
  void      SetBool(in string ID, in boolean value);
  boolean   GetBool(in string ID) raises(InvalidIdentifier);
  void      SetRealArray(in string ID, in DoubleSeq value);
  DoubleSeq GetRealArray(in string ID) raises(InvalidIdentifier);
  void      SetIntArray(in string ID, in LongSeq value);
  LongSeq   GetIntArray(in string ID) raises(InvalidIdentifier);
  void      SetStrArray(in string ID, in StringSeq value);
  StringSeq GetStrArray(in string ID) raises(InvalidIdentifier);
  boolean   IsSet(in string ID, in long ptype);
  boolean   RemoveID(in string ID, in long ptype);
  AttributeParameter GetFather();
  boolean   HasFather();
  boolean   IsRoot();
  void      Clear();
  StringSeq GetIDs(in long ptype);
};
```

```
class SALOMEDSClient_IParameters
{
  virtual int append(const std::string& listName,
                     const std::string& value);
  virtual int nbValues(const std::string& listName);
  virtual std::vector<std::string> getValues(
                     const std::string& listName);
  virtual std::string getValue(const std::string& listName, int index);
  virtual std::vector<std::string> getLists();
  virtual void setParameter(const std::string& entry,
                            const std::string& parameterName,
                            const std::string& value);
  virtual std::string getParameter(const std::string& entry,
                                   const std::string& parameterName);
  virtual std::vector<std::string> getAllParameterNames(
                            const std::string& entry);
  virtual std::vector<std::string> getAllParameterValues(
                            const std::string& entry);
```

```
  virtual int getNbParameters(const std::string& entry);
  virtual std::vector<std::string> getEntries();
  virtual void setProperty(const std::string& name,
                           const std::string& value);
  virtual std::string getProperty(const std::string& name);
  virtual std::vector<std::string> getProperties();
  virtual std::vector<std::string> parseValue(const std::string& value,
                const char separator, bool fromEnd = true);
  virtual std::string encodeEntry(const std::string& entry,
                                  const std::string& compName);
  virtual std::string decodeEntry(const std::string& entry);
  virtual void setDumpPython(_PTR(Study) study,
                             const std::string& theID);
  virtual bool isDumpPython(_PTR(Study) study,
                            const std::string& theID);
  virtual std::string getDefaultVisualComponent();
};
```

The access to the `IParameters` object is obtained via the `ClientFactory` interface:

```
SalomeApp_Study* study =
          dynamic_cast<SalomeApp_Study*>(myApp->activeStudy());

int savePoint = 1; // unique identifier of the visual state
_PTR(AttributeParameter) ap = study->studyDS()->GetCommonParameters(
          study->getVisualComponentName(), savePoint);
_PTR(IParameters) ip = ClientFactory::getIParameters(ap);
```

When the visual state is saved, the `SalomeApp_VisualState` class initializes the `IParameters` helper by new unique visual state identifier. Then, it stores general parameters like list of opened 2D/3D views, their titles and current layout, currently active view. For each view window, its specific `getVisualParameters()` method. Each view window class can override this method to store any required visual state parameters like color, trihedron size, etc, font, camera position, etc. Finally, `SalomeApp_VisualState` iterates through all the currently active modules calling their `storeVisualParameters()` method passing visual state identifier as parameter. Each module is obligatory to dump its visual state parameters to the save point using `IParameters` helper class.

The procedure of visual state restoring is very similar to the procedure of the saving. First, all required view windows are created, arranged and named according to the recorded state. For each view window, its specific `setVisualParameters()` method is called. All modules being active at the moment of the visual state recording, are (re)activated, and for each module its specific `storeVisualParameters()` method is called.

3. Session interface implementation

The main purpose of **Session** interface is to start and stop GUI session from the external process, giving an access to the SALOME GUI in a batch mode (from terminal window or from batch scripts) or from other SALOME processes (via CORBA bus). **Session** interface provides the following main methods:

- `GetInterface()` – launches GUI session (if it is not launched yet).
- `StopSession()` – stops the GUI session, closing its desktop window.
- `GetStatSession()` – gets Session state: GUI active/idle, number of opened studies, etc.
- `GetActiveStudyId()` – gets an ID of the currently active study;

- `GetComponent(in string theLibraryName)` – gets a reference to the module engine which is loaded in the same process as Session itself, for example Post-Pro (VISU).
- `restoreVisualState(in long theSavePoint)` – restore visual state with specified identifier.
- `emitMessage(in string message)` – send specific message to the Session server. The processing of this function is implementation-dependant.

An access to the GUI **Session** object is provided via `SUIT_Session` class (see paragraph 2.1).

4. SALOME Launching

To work with some **SALOME module** during the GUI session it is necessary to define environment variable `<MODULE>_ROOT_DIR` to point to the **module** binaries distribution and then modify configuration file (see 2.8 above for more details about **Resource manager**). Here and below `<MODULE>` is a **module** symbolic name, e.g. for SALOME Geometry **module** its symbolic name is GEOM.

Sample configuration file for module MODULE:

```
...
  <section name="launch">
    <parameter name="modules" value="MODULE"/>
  </section>
  <section name="resources">
    <parameter name="MODULE"
     value="${MODULE_ROOT_DIR}/share/salome/resources/module"/>
  </section>
  <section name="MODULE">
    <parameter name="name" value="Module"/>
    <parameter name="icon" value="Module.png"/>
    <parameter name="library" value="libModuleGUI.so"/>
    <parameter name="singleton" value="false"/>
  </section>
...
```

The parameter *modules* of the section *launch* defines the list of SALOME **modules** which should be included into the launching SALOME session.

The section *resources* defines for each **module** the directory or list of directories where SALOME GUI should look for resource files. Usually it is a **module**'s share/salome/resources/<module> folder.

The module specific configuration section may contain any component-specific preferences, but several parameters in this section have special meaning, and some of them are obligatory:

- The *name* parameter specifies the module presentable name (the one displayed in the "Components" toolbar).
- The *icon* parameter specifies the icon for the module.
- The optional *library* parameter specifies the name of the module's GUI library; by default (if the *library* parameter is not specified) the name of the module's GUI library is the same as module symbolic name. For example, for module GEOM the library name is libGEOM.so for UNIX/Linux and GEOM.dll for Windows. For custom library name, it's enough to specify library name with removed OS-specific prefix and suffix.
- The optional *singleton* parameter can be used for the module which can be used for only one study. By default, this parameter is set to *false*.

The default list of the modules used in new SALOME session (defined via configuration files) can be overridden by the command line option of the `runSalome` / `runLightSalome.sh` script, for example:

```
runSalome --modules=GEOM,MED,VISU
runLightSalome.sh --modules=LIGHT,MYMODULE
```

5. Working cycle of SALOME GUI

This paragraph provides a description of the typical working cycle of the SALOME application, specifying step-by-step procedure of GUI session from launching of the application to creating/loading of the study and then to exiting the application, making an emphasis on interaction between major functional parts of SALOME GUI and SALOME KERNEL.

The SALOME GUI desktop is shown on the Figure 8:

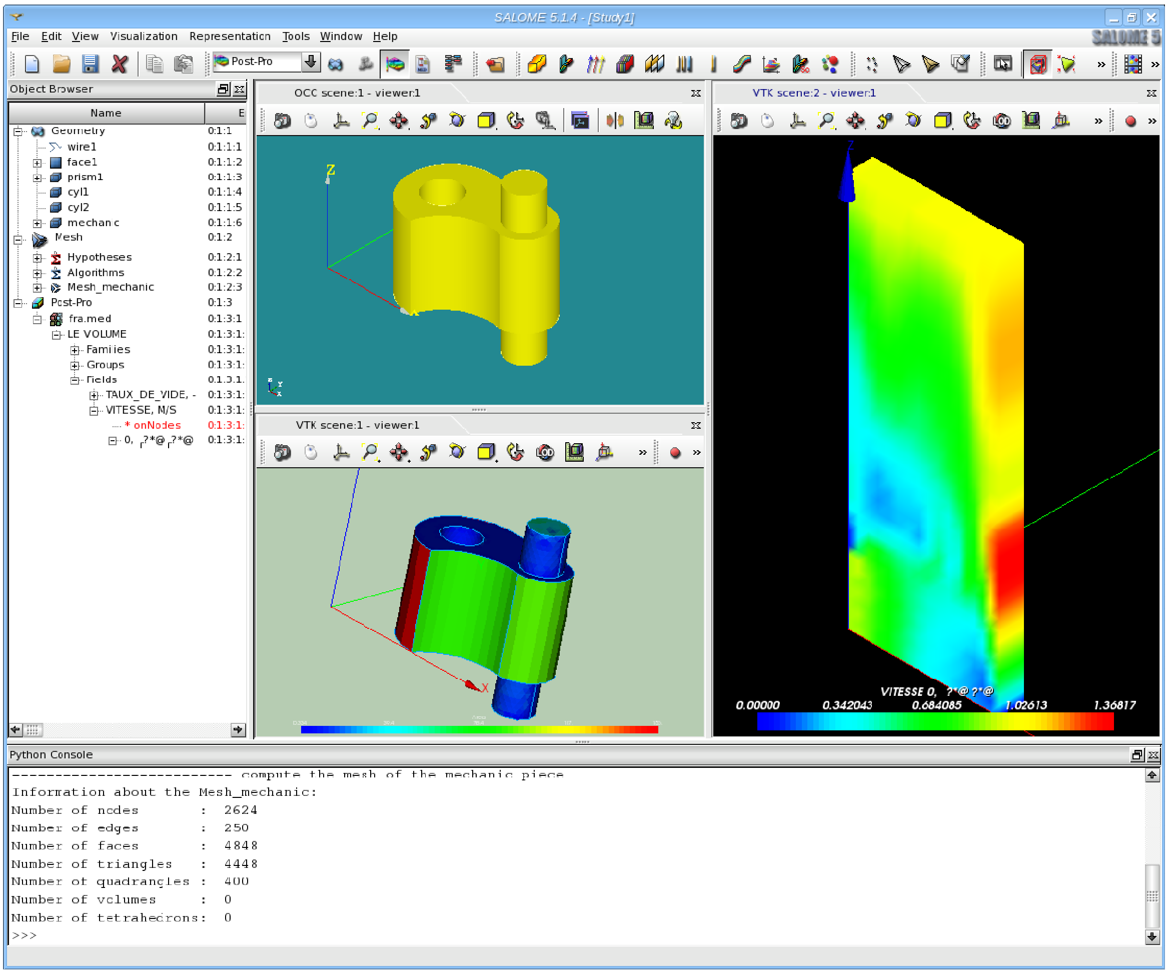


Figure 8. SALOME GUI desktop window

5.1 Typical GUI session

Step 1 : Launch the application

Since SALOME application implies some input parameters which define parameters of launching, the current launch procedure should be kept as is:

1) set SALOME environment (define all environment variables which are necessary for successful running of the application);

2) launch SALOME by running `runSalome` script (start SALOME servers, see below).

The input parameters for the launch are given as:

- Set of **configuration files** (XML files specifying different settings of SALOME application, see 2.8.2), listed below in the priority ascending order:
    - **global** read-only files are situated in the modules resources directories (one file per module); these files specify default SALOME options, including parameters of launching (default list of SALOME **modules**, emdedded/standalone servers settings, splash screen settings etc).
    - **optional** configuration files: it is possible to specify arbitrary number of additional configuration files; these files can be situated in any directory and contain specific users defaults. These files should be named SalomeApp.xml (for distributed SALOME session) or LightApp.xml (for "light" SALOME session) and directories containing these files should be specified within the corresponding environment variable: SalomeAppConfig or LightAppConfig (these variables represent a list of directories separated by ":" symbol); otherwise these optional configuration files are not taken into account.
    - **user** preferences file is situated in the user's home directory); includes user preferences, changed within GUI session. Though it is possible to edit this file manually, it is not recommended since it can be changed during the next SALOME session.
- Command line parameters which override the launch parameters read from the configuration file(s).

⇒ Read **configuration files** (launch options)

⇒ parse command line options

⇒ take the list of SALOME **modules** to be used in the GUI session

⇒ run standalone servers

⇒ run `Session_Server`

- create `SALOME_QApplication` instance
- load main GUI resources using **Resource Manager**
- start separate thread for starting of embedded servers
- show **Splash** window with progress bar to show servers loading process
- load and activate embedded servers (if there are any)
- create `Session` CORBA interface

- activate GUI desktop: `Session` → `GetInterface()`
    - create `SUIT_Session` instance
    - load *SalomeApp* library (libSalomeApp.so / SalomeApp.dll)
    - create `SalomeApp_Application` instance (no study opened yet)
    - install and activate exception & signals handler
    - run main application event loop and start processing GUI events

Step 2 : New study

⇒ `SUIT_Session` → create new `SalomeApp_Application` instance (reuse existing if first study is being created)

⇒ `SalomeApp_Application` → create empty `SalomeApp_Study`

Step 2 : Open study

⇒ `SUIT_Session` → create new `SalomeApp_Application` instance (reuse existing if first study is being created)

⇒ `SalomeApp_Application` → create `SalomeApp_Study`

⇒ `SalomeApp_Study` → load study data

- `SALOMEDS_StudyManager` → `Open()`
- `SalomeApp_Study` → read `SALOMEDS_Study` contents
- create tree of `SalomeApp_DataObject` instances and display it in the Object browser

Step 3 : Activate a module *Module*

⇒ load *Module* library (GUI library of the module, libModule.so / Module.dll)

⇒ create (and cache in the **Application** object) instance of the `SalomeApp_Module` class

- `SalomeApp_Module` → load module CORBA engine (for distributed modules)
- `SalomeApp_Module` → create `SALOMEApp_DataModel` instance
- `SalomeApp_DataModel` → `open()`
    - `SALOMEDS_StudyBuilder` → `LoadWith()` (if SALOME **module** has

CORBA engine)

- ask SALOMEDS_StudyBuilder for data stream files and process them (if there is no CORBA engine – for “light” modules)[1]

▪ SalomeApp_Module → customize menus/toolbars, show/hide dockable windows

Step 4 : Edit study: the working session

⇒ SalomeApp_DataModel → update()

⇒ SalomeApp_DataModel → update itself from the SALOMEDS_Study

▪ rebuild and repaint Object browser

Step 5 : Save study

⇒ SalomeApp_DataModel → save()

▪ SALOMEDS_StudyManager → Save() (if SALOME **module** has CORBA engine)

▪ create data stream files and put them to the SALOMEDS_StudyManager for saving (if **module** does not have CORBA engine – for “light” modules)[2]

Step 6 : Close study

⇒ SalomeApp_Study → close()

▪ SALOMEDS_StudyManager → Close() (if **module** has CORBA engine)

▪ SUIT_Session → delete SalomeApp_Application instance (or just clear it if last study is closed)

- SalomeApp_Application → delete SalomeApp_Study instance

- close all viewers and windows and close study’s desktop (if not last)

Step 7 : Close application

⇒ SUIT_Session → closeSession()

⇒ SALOME_QApplication quits

[1] to reuse a SALOMEDS feature to store all data in one file or in multi-file mode.
[2] see previous remark

6. Python modules

In order to minimize expenses concerned with migrating of existing Python SALOME **modules** on new GUI architecture SALOME 3 and newer tries to keep as much as possible the compatibility with SALOME 2.x Python API. This concern *SalomePyQt*, *SALOME_Swig* and *SalomePy* libraries which are used from the python code to access SALOME GUI functionality, and *SalomePyQtGUI* library which represents a base GUI library for distributed Python **modules**. This significantly reduces the migrating expenses for the Python **components**.
In addition, since version 5.1.2 SALOME provides an additional GUI library: *SalomePyQtGUILight* that can be used to develop “light” Python modules (not having CORBA engine).

6.1 PyQT GUI libraries

SALOME GUI module provides two special libraries which provide general GUI for Python-developed **modules**. These libraries implement callback functions from C++ to Python which allow to process user actions in Python **modules** (creation of **Study**, activation of the **Study**, activation/deactivation of the **Module**, invoking of the menu command, etc). There are two such libraries:

- *SalomePyQtGUI* – used for CORBA-based modules (libSalomePyQtGUI.so / SalomePyQtGUI.dll).
- *SalomePyQtGUILight* – used for “light” Python modules (libSalomePyQtGUILight.so / SalomePyQtGUILight.dll).

To provide the common GUI functionality for all Python **modules** PyQt GUI libraries are implemented with the same rules as any other **modules** GUI:

- classes `SALOME_PYQT_ModuleLight` (inherits `LightApp_Module` class) and `SALOME_PYQT_Module` (inherits `SALOME_PYQT_ModuleLight` and `SalomeApp_Module` classes) are responsible for interacting between SALOME GUI and Python **modules**;
- Class `SALOME_PYQT_PyInterp` (derived from `PyInterp_Interp`) is common Python sub-interpreter shared between all Python **modules** (GUI part);
- *SalomePyQtGUI* and *SalomePyQtGUILight* libraries export instances of the `SALOME_PYQT_Module` and `SALOME_PYQT_ModuleLight` classes correspondingly by request of the `SalomeApp_Application` class.

The paragraph 4 describes the actions which should be done in order to launch SALOME with custom **module**(s). The same concerns the pure Python **modules**. All Python **modules** are listed in the *launch* section of the configuration file in *modules* parameter exactly in the same way as other (C++) SALOME **modules**:

```
...
<section name=”launch”>
  ...
  <param name=”modules” value=”GEOM,SMESH,VISU,PYCALCULATOR,PYHELLO”/>
  ...
</section>
...
```

The *library* parameter of each *module* preferences section should contain “SalomePyQtGUI” value for CORBA-based modules or “SalomePyQtGUILight” value for “light” modules (see 4). It means that for the Python **modules** which do not have own GUI libraries *SalomePyQtGUI* or *SalomePyQtGUILight* library should be used. However, it is possible to implement own GUI library in a similar way as standard SALOME PyQt GUI libraries to suit specific module needs.

As it was mentioned in 2.10 menu resource files (XML-based) are not supported in SALOME GUI since version 3.0. All menu/toolbars actions should be hard-coded in the **module** sources. The only exception to this rule is made to the Python **modules**. The support of XML-based menu resource files is kept as before. The SALOME PyQt GUI libraries automatically search menu definition file and parse it creating main menus, toolbars and popup menus. The format of menu definition files is not changed too.

Note: new Python-based **modules** should avoid using of XML-defined menus, because this way of menu definition is obsolete and can be removed in future versions of SALOME. New Python API should be used instead in order to create menu, toolbars, etc.

Some **modules** can use direct access to main menu via *SalomePyQt* Python API module:

```
static QMenuBar*   getMainMenuBar();
static QPopupMenu* getPopupMenu(const MenuName);
```

Such **modules** add menu items to the main menu manually, usually by using `addAction()` PyQt function. Since application does not clear automatically all “alien” menu items, the **module** is obliged to remove these menu items by its own. Note that if XML-based files are used for creation of menus the clearing is performed automatically.

In order to provide the compatibility with previous SALOME GUI and to minimize the expenses of migration of the already implemented SALOME Python **modules** (like ALLIANCES application, SINERGY, TECHOBJ, etc.) on new GUI, the name and the behavior of callback functions were not changed, except those functions which previously were not used at all (stub functions). Such functions are removed in the new GUI:

- `setWorkSpace(sipType_QWidget workspace)` – passes the workspace object to the python **module**; called each time the module is activated or active study is switched.
- `setSettings()` – called each time when **module** is activated.
- `activeStudyChanged(int studyId)` – called each time when active study is changed by user; `studyId` is a unique study identifier.
- `definePopup(char* context, char* object, char* parent)` – called when context popup menu is requested to define the popup menu context; this function should return list of the values of `context`, `object`, `parent` parameters, modified according to the current selection.
- `customPopup(sipType_QMenu menu, char* context, char* object, char* parent)` – called when popup menu is created from the XML menu definition files (see above) and invoked by the user.
- `OnGUIEvent(int commandId)` – called when user activates some GUI action (from main menu, toolbar or context popup menu); `commandId` is a unique GUI action identifier.

In parallel with old API the new one is provided that is preferable for newly designed Python modules:

- `initialize()` – called when component Module instance is just created and is being initialized.
- `activate()` – called when component is activated; should return Python boolean *True* value if module activation is finished correctly and *False* otherwise.
- `deactivate()` – called when component is deactivated.
- `windows()` – should fill in and return the list of compatible dockable GUI elements (Object browser, Python Console, Log window) and their desired position in the GUI desktop.
- `views()` – should fill in and return the list of compatible viewers which should be automatically opened/activated at the module activation.

- `createPopupMenu(sipType_QMenu menu, char* context)` – called when context popup menu is requested; `menu` is a sip wrapper for the QMenu class, `context` is a popup menu context name ("Object browser", "OCCViewer", etc).
- `createPreferences()` – this method called when the application's common Preferences dialog box is first time invoked; this method should be used by the module to export own preferences to the Preferences dialog box.
- `preferenceChanged(char* section, char* parameter)` – called each time when module's preference item `parameter` of resources section `section` has been changed by the user.
- `activeViewChanged(int viewId)` – called when any 3D/2D view is activated; `viewId` is an unique viewer identifier.
- `viewCloned(int viewId)` – called when any 2D/3D view is cloned; `viewId` is an unique viewer identifier.
- `viewClosed(int viewId)` – called when any 2D/3D view is closed; `viewId` is an unique viewer identifier.

There are also some callback functions specific for "light" modules only:

- `saveFiles(char* dir)` – called when study is saved; `dir` is a path to the temporary directory where the Python module should put own persistence files; the function should return list of persistence file names.
- `openFiles(sipType_QStringList files)` – called when study is loaded; `files` is a list of strings: the first item specifies a path to the temporary directory where the persistence files are unpacked (rest of the items in the list); the Python module should read these files to restore the study data. This function should return Python boolean value *True* in case of success and *False* otherwise.

The following callback functions are used for CORBA-based Python modules only:

- `engineIOR()` – should return (usually loading and initialize before if neded) the module's CORBA engine string identifier.

6.1.1 Caveats

- Multi-desktop environment

Since version 3.0 SALOME GUI has introduced multi-desktop interface. Python **modules** should take this into attention. It means that if any desktop-referenced variable is used in the code this variable should be reinitialized in `setSettings()` and/or `activeStudyChanged()` methods. It is recommended to re-retrieve desktop widget each time when it is needed by calling of the corresponding SALOME Python API methods.

- Workspace

SALOME GUI uses tabbed widget to stack all viewer windows. Moreover, SUIT library provides different types of desktop, and the default desktop type can be changed in future or even become customizable. So, the Python module GUI should not rely on the value passed by `setWorkspace(sipType_Widget workspace)` method – it might even pass 0 (zero) value in future. This method seems not to have significant utility. It is considered as obsolete and will be removed in later version of SALOME GUI.

6.2 Python interface libraries

SALOME GUI includes some libraries to provide an access to the GUI from the Python **modules**. Python **modules** usually use these libraries to get access to SALOME desktop, to the selection, to the main menu of the application, to show "File Open"/"File Save" dialog boxes, to set/get SALOME settings, etc.

Different GUI functionality is wrapped by SWIG and SIP utilities.

There are three different libraries implemented by SALOME GUI:

- *SALOME_Swig* library (_libSALOME_Swig.so / SALOME_Swig.dll) – SWIG-wrapping library for SALOME GUI, it is accessible via Python module `libSALOME_Swig`. This module provides the following methods:
  - `getActiveStudyId()`, `getActiveStudyName()` - get currently active study identifier and its name (URL),
  - `updateObjBrowser()` - update Object browser,
  - `SelectedCount()`, `getSelected()` - get total number of selected objects and retrieve entry of each currently selected object.
  - `AddIObject()`, `RemoveIObject()`, `ClearIObjects()` – modify the selection.
  - `Display()`, `DisplayOnly()`, `Erase()`, `DisplayAll()`, `EraseAll()`, `IsInCurrentView()`, `UpdateView()` - display/erase operations for the currently active viewer.
  - `FitAll()`, `ResetView()`, `ViewTop()`, `ViewBottom()`, `ViewLeft()`, `ViewRight()`, `ViewFront()`, `ViewBack()` – different view operations.

- *SalomePyQt* library (SalomePyQt.so / SalomePyQt.dll) – SIP-wrapping library for SALOME GUI, created using PyQt bindings for the Qt library. This library is accessible viau the Python module `SalomePyQt` and provides a lot of the helpful methods, some of them (not all) are listed below:
  - `getDesktop()`, `getMainFrame()` - get access to the currently active desktop widget and its main working area widget.
  - `getMainMenuBar()`, `getPopupMenu()` - get access to the main menu bar and submenus.
  - `getSelection()` – retrieve Selection object.
  - `putInfo()` – print status message to the desktop's status bar.
  - `getActiveComponent()` – get currently active module name.
  - `updateObjBrowser()` - update Object browser,
  - `getFileName()`, `getOpenFileNames()`, `getExistingDirectory()` – show "Open File", "Save File", "Open Files" or "Choose Directory" standard dialog boxes.
  - `createObject()`, `setName()`, `setIcon()`, `setToolTip()`, `removeObject()`, etc… - set of functions to create presentable data object tree; these functions should be used by the "light" Python module to fullfil the Object browser with the module data tree.
  - `helpContext()` – invoke help browser.
  - `dumpView()` – dump currently active view's contents to the image file.
  - `createMenu()`, `createTool()`, `createAction()`, and other – menu/toolbars management functions.
  - `addSetting()`, `removeSetting()`, `hasSetting()`, etc… - get/set user preferences.
  - `addGlobalPreference()`, `addPreference()`, `setPreferenceProperty()`, … - preferences management (for common application's Preferences dialog box).
  - `getViews()`, `getActiveView()`, `getViewTitle()`, `findViews()`, `activateView()`, `createView()`, `cloneView()`, `splitView()`, `moveView()`, etc – large set of different view management functions.

- *SalomePy* library (libSalomePy.so / SalomePy.dll) – this library mainly provides Python interface to get access to the VTK viewer; it is implemented using vtkPython library API (a part of VTK toolkit). This library is accessible via `SalomePy` Python module:
  - `getRenderer()` - get VTK renderer object.

- `getRenderWindow()` - get VTK render window object.
- `getRenderWindowInteractor()` – get VTK render window interactor object.
- `showTrihedron()` – show/hide trihedron.
- `fitAll()`, `setView()`, `resetView()` – view management; obsolete functions, please use SalomePyQt module for this purpose.

To provide the compatibility with the previous SALOME GUI all these libraries are re-implemented according to the new GUI, providing new API and keeping the old Python API in order to minimize migrating problems.

Newly designed Python modules should avoid using old-style API.

7. "Light" modules

Since version 3.0 SALOME GUI enables supporting of CORBA-independent modules. Such modules might not use CORBA at all, having an internal data structure, which can be written in C++ or Python. These modules always work in the same process as SALOME GUI and from the user's point of view have no difference with standard CORBA-based modules.

For persistence needs, special CORBA pseudo-engine (`SalomeApp::Engine`) is created and activated by SALOME_Session_Server executable on the application start-up. This interface is derived from the `Engines::Component` and `SALOMEDS::Driver`. It is used for homogenous SALOMEDS operation during storage and retrieval of data belonging to the SALOME modules having no CORBA engine.

The `SalomeApp_Engine_i` class provides an access to the pseudo-engine instance via static method `GetInstance()`:

```
static SalomeApp_Engine_i* GetInstance();
```

The `SalomeApp_Engine_i` class implements `Load()` and `Save()` methods:

```
CORBA::Boolean Load(SALOMEDS::SComponent_ptr theComponent,
                    const SALOMEDS::TMPFile& theFile,
                    const char* theURL, bool isMultiFile);
SALOMEDS::TMPFile* Save(SALOMEDS::SComponent_ptr theComponent,
                        const char* theURL,
                        bool isMultiFile);
```

- The method `Load()` unpacks a binary stream received as an argument with help of `SALOMEDS_Tool` class. Then it updates internal files map <study_id> → <component_data_type> → <list_of_files>.
- The method `Save()` puts data files of the given module to a binary stream using internal map <study_id> → <component_data_type> → <list_of_files>. The files map should be prepared through the local C++ API (see below).

Local C++ API of `SalomeApp_Engine_i` servant includes two methods:

```
typedef std::vector<std::string> ListOfFiles;
ListOfFiles GetListOfFiles(const int theStudyId,
                           const char* theComponentName);
void SetListOfFiles(const ListOfFiles theListOfFiles,
                    const int theStudyId,
                    const char* theComponentName);
```

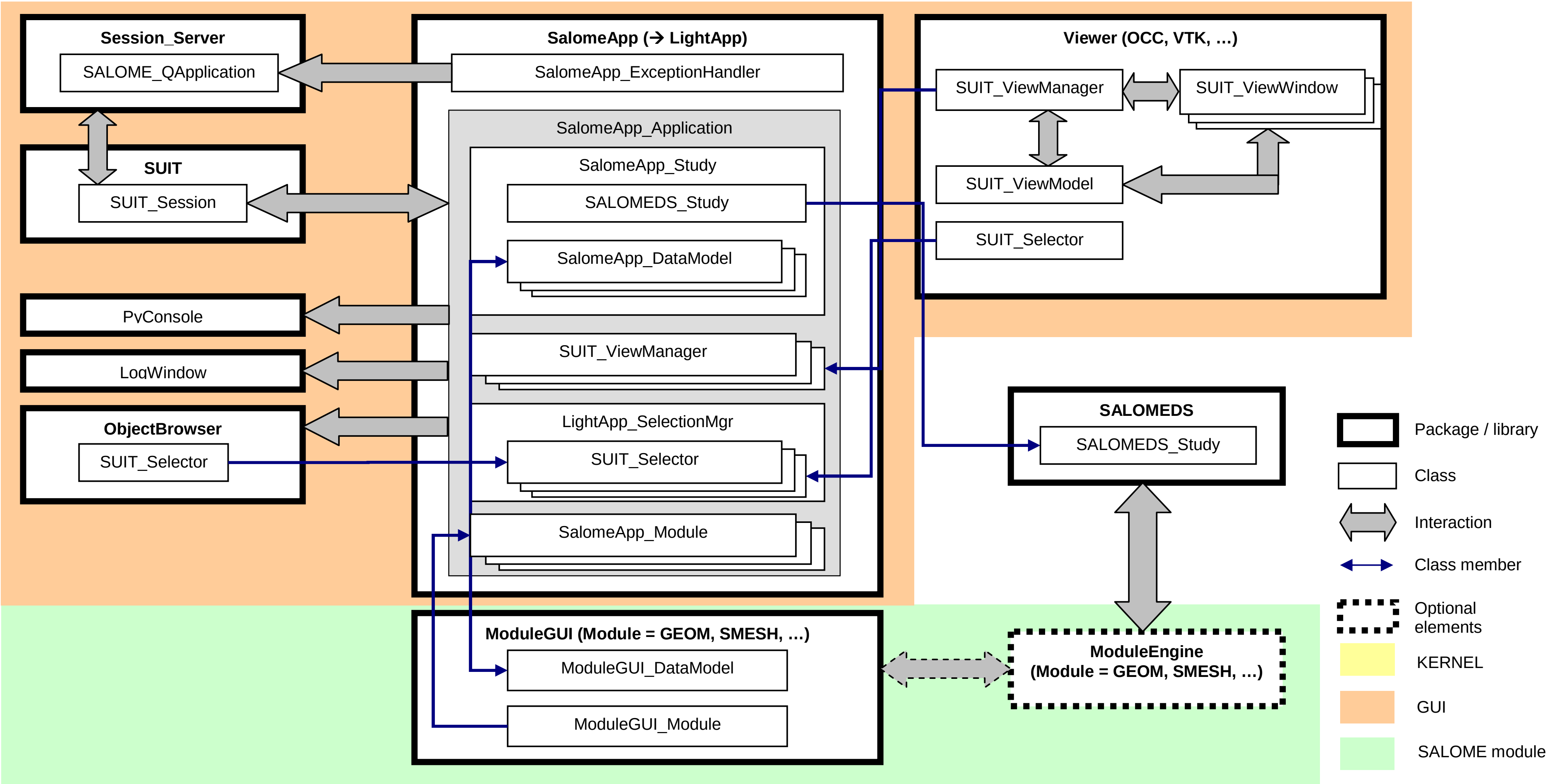
- The method `GetListOfFiles()` returns list of paths to the persistent files retrieved from the given study. **Data Model** uses this method to start loading its data files, after they have been retrieved from a study.
- The method `SetListOfFiles()` passes the list of paths to persitsent files to be put into the given study. **Data Model** uses this method after saving its data files to put them into a study.

To load and activate CORBA pseudo-engine parameter *embedded* of the *launch* section of the configuration file should contain “SalomeAppEngine” value (see 2.8.2).

8. Batch mode

From the batch mode point of view SALOME GUI did not undergo significant changes. As previously, an access to the GUI is provided by Session interface, which allows starting/stopping the GUI, to get the number of the opened studies and the currently active study (see paragraph 3 above).

APPENDIX 1: General structure of SALOME GUI



Références documentaires

Documents de référence

Les documents cités dans le présent document ou utiles à la compréhension de son contenu sont :

Titre	Référence
Contrat MCO SALOME 2009-2010	PRO/GCVP-P/092801/V1 OC2D09025C

Historique des révisions

Les versions successives du présent document sont :

	Version	Rédacteur	Date	Objet de la révision
Version en vigueur	0.4	V SANDLER	07/06/2010	Update for SALOME version 5.1.3
Versions antérieures	0.3	V SANDLER	08/07/2005	Adding UML diagrams and glossary
	0.2	V SANDLER	29/06/2005	Update for SALOME version 3.0.0
	0.1	S ANIKIN, V SANDLER	16/03/2005	From scratch