SALOME TUTORIAL
USER'S GUIDE

Contents

1. INTRODUCTION

This document represents a tutorial on SALOME platform. The tutorial provides an introduction to the developing of an application based on SALOME platform. It does not cover all the aspects of software application development; however it describes basic workflow which should be applied by the developers in order to implement new SALOME component(s).

This tutorial is intended for the programmers with extended experience in C++ and/or Python languages, but unfamiliar with SALOME platform. In addition, some knowledge of CORBA technology, numerical computation cycle (pre-processing, processing, post-processing), and Qt library is required.

1.1 HOW TO USE THIS TUTORIAL

The tutorial is an interactive presentation which describes, step by step, how to build a custom SALOME-based application (SALOME component) "from scratch". It can be used as a training material organized in series of "lessons" where every lesson introduces a certain issue of SALOME platform. Further lessons are based on the previous ones so it is proposed to study them one after another to avoid misunderstanding.

The tutorial is mainly intended to help a new developer, fully unfamiliar with SALOME platform, to understand its main concepts and develop a customized SALOME component. Also, it can be used by experienced users of SALOME platform as a reference material. The tutorial can be observed a recommended practices warehouse in this case.

The chapter 6 of the tutorial “SALOME Concepts” presents main concepts of SALOME platform. Having met an unknown word (notion, concept) in the tutorial, please, refer to this chapter for explanations.

1.2 PRE-REQUISITES

This tutorial is applicable to the SALOME platform version 7.3.0 and newer. For other pre-requisites (3rd-party products) please refer to the SALOME requirements.

Note: SALOME platform distribution under Linux (SALOME Install Wizard) includes pre-compiled binaries of SALOME Tutorial modules, described in this document. If you choose installation of all SALOME modules into single directory (this option is available in SALOME Install Wizard), you will not be able to complete this tutorial as modules installed with SALOME will have more priority and, thus, they will be used in run-time instead of manually built ones.

1.3 COHESION OF SAMPLE COMPONENTS

Since the main goal of the tutorial is to provide the guide for creation of a new component, it is separated into 3 main parts:

- ATOMIC: light-weight component.
- ATOMGEN: Python component with CORBA engine.
- ATOMSOLV: C++ component with engine

It is proposed to study the tutorial in this order (Light-weight, then Python, then C++ with engine) because some topics are common and they will be explained only once in previous chapters. Every chapter consists of explanations and the code in C++/Python that builds up a component.

The components developed with SALOME platform are mostly used for pre- and post-processing applications. Usually the data is prepared by one component, then it is processed by various solvers of other components that represent application of different physical algorithms to the data, and the results of calculations might be displayed by yet another component! Such interaction between components on the basis of data processing is known as coupling of the components. This tutorial will simulate a simple model of components coupling; the following interaction between the components will be developed during the tutorial:

- The first component to be developed is a light-weight component named ***ATOMIC***. Its main goal is to prepare the data for the calculations. It allows the user to input the data using dialog boxes and export the data to a file. In the ATOMIC component it is impossible to visualize this data in any way, neither it is intended to perform any algorithmic processing of the data. These tasks will be carried out by the other components. The data is a list of records, each containing 3 floating point values (Cartesian coordinates), and a string value (name). It represents molecules and atoms. An XML format is used for the exporting of the data.

- The second component is a Python component named ***ATOMGEN***. It performs the processing of the data. It reads the data from the XML file and performs "algorithmic processing" to the data. After processing, the "molecules" are increased in number and oriented along some curve. It can be thought as "spatial algorithmic processing" of molecules. Then the data is stored in an internal data structure so it is accessible by other components via CORBA interface.

- The third component is a C++ component with CORBA engine named ***ATOMSOLV***. It allows further processing of the data and displaying of the results. This component introduces concepts of the viewer, view management, selection management, etc. It is possible to open a 3D view window in this component and display the "atoms" and "molecules". The "atoms" are represented by spheres in 3D space. ATOMSOLV also performs further processing of data, this time not spatial, but let's say "thermal". After the processing, different properties are assigned to the "atoms", that is reflected in their 3D representation – the atoms are differently colored. It will be also possible to change the color and display mode of the "atoms" manually. An important issue is how the data ("atoms") is retrieved from the ATOMGEN component. It is done by means of CORBA interface directly from the Python component's engine.

2. SALOME BUILD PROCEDURE

In this section we introduce the structure of a SALOME component source directories and a formal procedure of building the application. SALOME platform is rather complex software, and its build procedure requires examination.

The description of build procedure given in this chapter is mainly aimed for Linux platform, however Windows is also supported by build procedure.

2.1 GENERAL DESCRIPTION OF THE BUILD PROCEDURE

The build procedure used for SALOME modules is based on the CMake tool (http://www.cmake.org). CMake is a cross-platform build system which works on Linux, Windows and other operating systems.

CMake build system is used to define build rules for the application that allows developing of the cross-platform build procedures. They allow hiding a lot of details of Makefile's from the developer. With CMake tool, the build procedure of each SALOME module consists of the following steps:

Step	*Description*	*Input*	*Output*
cmake	Read script files and produce input files for the native build system of the platform where it runs on. It can create GNU Makefiles, KDevelop project files, XCode project files, MS Visual Studio project files, etc. When invoked, cmake command performs configuration of the build procedure, making all necessary products and tools detection, compilation/installation options set-up etc.	CMakeLists.txt files *.cmake files *.in files	CMakeCache.txt Makefile other files
make	Compile all sources generating libraries, executables, etc. in the build directory	Makefile Source code files	Libraries, executables, scripts, resources, documentation, etc. in the build directory
make install	Install the application to the target directory (by default, /usr).	Libraries, executables, etc. in the build directory	Libraries, executables, etc. in the target directory

The CMake tool based procedure uses set of input files which define the build rules. The CMake build system is implemented in such a way to simplify as much as possible the defining of the build rules. It represents a “top-level” layer above the GNU make utility, providing simplified rules for such objects like libraries (static and shared), executables, python scripts, resources, documentation files, etc. It also provides integration with other build tools, like compilers, linkers, documentation generation utilities; supports different programming languages (C++, Fortran, Python, etc.) and other. Also it is possible to define own build rules; this is done by creating custom scripts following CMake syntax. The input for CMake build system is usually a set of `CMakeLists.txt` and `*.cmake` files.

CMake input files are used to describe the build procedure, in particular:

- Test platform;
- Test system configuration;

- Detect pre-requisites;
- Specify targets (libraries, executables);
- Generate build rules (for example, standard UNIX makefiles on Linux, MSVC solutions, etc).

2.2 TYPICAL SOURCES PACKAGE ORGANIZATION

Let's take a look at the typical SALOME module sources package. Usually source package of SALOME module consist of the set of files and sub-directories, including source code (C++, Fortran, Python, etc.), build procedure files (`CMakeLists.txt`), check procedures (`*.cmake`), resources files (images, translation files, etc.), test scripts and other staff.

Unpack attached archive ***light-00.tar.gz*** with ATOMIC module initial source tree to your working directory. It represents typical sources directory tree:

`ATOMIC_SRC`	- root directory
`ATOMIC_SRC/adm_local`	- administrative directory
`ATOMIC_SRC/adm_local/cmake_files`	- CMake-related administrative files
`ATOMIC_SRC/bin`	- directory for scripts and other tools
`ATOMIC_SRC/bin/VERSION.in`	- version file
`ATOMIC_SRC/resources`	- directory for common resources
`ATOMIC_SRC/resources/SalomeApp.xml`	- configuration file ("full" SALOME)
`ATOMIC_SRC/resources/LightApp.xml`	- configuration file ("light" SALOME)
`ATOMIC_SRC/resources/ATOMIC.png`	- main module icon
`ATOMIC_SRC/src`	- directory for source code
`ATOMIC_SRC/src/ATOMICGUI`	- GUI library source code directory
`ATOMIC_SRC/src/ATOMICGUI/ATOMICGUI.h`	- GUI library header file
`ATOMIC_SRC/src/ATOMICGUI/ATOMICGUI.cxx`	- GUI library implementation file
`ATOMIC_SRC/src/ATOMICGUI/resources`	- GUI library custom resources
`ATOMIC_SRC/ATOMIC_version.h.in`	- Module header file template
`ATOMIC_SRC/CMakeLists.txt`	- root CMake system file

Some secondary files are not listed. Typically, each sub-directory also contains own `CMakeLists.txt` file.

In addition, other modules can provide more complex sources package structure including additional directories, for example `doc` for the documentation sources, `idl` for CORBA interfaces files, etc.

2.3 CUSTOMIZE BUILD PROCEDURE

Project's root directory provides main CMake configuration that allows build all targets into one set of binaries and libraries. Each sub-directory also includes CMake configuration file (`CMakeLists.txt`) that specifies targets being build.

The file CMakeLists.txt in root directory of the SALOME module provides basic build rules to be used in other `CMakeLists.txt` files. It sets main properties of project: name, version, pre-requisites, installation paths, programming languages being used by the project, tree of sub-directories, etc.

A lot of files used by the build procedure of each SALOME module are located in SALOME KERNEL module (that is referenced by the `KERNEL_ROOT_DIR` environment variable), namely in its `salome_adm` sub-folder. Similarly, the `GUI_ROOT_DIR` environment variable is used for the graphical user interface (GUI) module of SALOME; this module also provides a set of configuration utilities (`*.cmake` files) in its `adm_local` folder.

Some modules can need some external packages in order to compile and run properly. The usual approach is to write a special `*.cmake` file for the purpose of finding a certain piece of software and to set it's libraries, include files and definitions into appropriate variables so that they can be used in the build process of another project. It is possible to include the standard CMake detection

modules (`FindXyz.cmake` files, located in the standard CMake installation directory) or, if CMake does not provide a search procedure for some required software, it is necessary to create `*.cmake` module for each pre-requisite.

Imagine we plan to use **gd** library for GIF/JPEG pictures creation in ATOMIC module. Then we must check at the configuration step, that this library is available, properly installed and compatible with our source code. To do this, we'll need to add the corresponding check to our build

- Create `*.cmake` file implementing the procedure that checks the availability of the gd library and sets required compilation / linkage options.

Take a look at the attached ready-to-use `FindGd.cmake` file. It provides a typical example of such procedure implementation. The CMake tool includes a large set of helper macro-commands which can be used in the check procedures. In our case, we check presence and availability of the `gd.h` header file and `libgd.so` library using pre-defined CMake macro-commands `FIND_PATH` and `FIND_LIBRARY`. If everything goes well, we set the global variable `GD_FOUND` to `TRUE`, otherwise print the warning message. In addition, we specify `GD_LIBRARIES` and `GD_INCLUDE_DIRS` variables that can be then used in the `CMakeLists.txt` files when it's necessary to link against the **gd** library.

- Add check procedure into `main CMakeLists.txt` file and print `GD_FOUND` variable value in the summary section:

```
MESSAGE(STATUS "Check for gd ...")

FIND_PACKAGE(Gd)
...
MESSAGE(STATUS "Find gd - ${GD_FOUND}")
```

- Include `GD_LIBRARIES and GD_INCLUDE_DIRS` variables in the `CMakeLists.txt` where it's necessary to link against the **gd** library. This can be done in `${GD_LIBRARIES}` and `${GD_INCLUDE_DIRS}`. For example

```
...
INCLUDE_DIRECTORIES(${GUI_INCLUDE_DIR} ${GD_INCLUDE_DIRS})
...
TARGET_LINK_LIBRARIES(mylibrary ${GUI_LIBRARIES} ${GD_LIBRARIES})
...
```

2.4 BUILD THE MODULE

We start developing the component with a stub that contains only basic directory structure, `CMakeLists.txt` files, several resources files and two source files with minimal code required for GUI library building. The archive file ***light-00.tar.gz*** with ATOMIC module initial source tree has to be unpacked to the working directory. To build the module it's necessary to perform the following actions:

- Check prerequisites: Linux operating system, C/C++ compiler, CMake tool availability, etc.
- Set SALOME build environment (environment scripts `env_build.sh` and `env_build.csh` are the part of the SALOME installation procedure), for example:

```
[%] source env_build.sh
```

- Set ATOMIC module environment (this is optional for build procedure, but helps giving further instructions) - set paths to the ATOMIC module source, build and installation directories, either using file **`atomic env build.sh`** (note: you have to edit it before usage to set correct paths):

```
[%] source atomic_env_build.sh
```

or by using shell commands directly:

```
[%] export ATOMIC_SRC_DIR=your_sources_path/ATOMIC_SRC
[%] export ATOMIC_BUILD_DIR=your_build_path/ATOMIC_BUILD
[%] export ATOMIC_ROOT_DIR=your_install_path/ATOMIC
```

Only the environment variable `ATOMIC_ROOT_DIR` is actually required - for using ATOMIC module in SALOME and for compilation of other modules, depending on the ATOMIC (this is not necessary for ATOMIC compilation and can be done later, but we tell it here for better understanding).

- Build and install ATOMIC module:

  Change directory to the `${ATOMIC_SRC_DIR}` (if not yet there) and go one level up:

  ```
  [%] cd ${ATOMIC_SRC_DIR}
  [%] cd ..
  ```

  Here, we suppose that a source directory and a build directory are on the same level in the directories hierarchy. Create build directory `${ATOMIC_BUILD_DIR}`, and `cd` to this directory:

  ```
  [%] mkdir ${ATOMIC_BUILD_DIR}
  [%] cd ${ATOMIC_BUILD_DIR}
  ```

  Invoke `cmake` command to prepare build directory for compilation:

  ```
  [%] cmake –DCMAKE_BUILD_TYPE=<Mode>
      -DCMAKE_INSTALL_PREFIX=${ATOMIC_ROOT_DIR} ../${ATOMIC_SRC_DIR}
  ```

  Here, `<Mode>` is a build mode (`Release` or `Debug`), `ATOMIC_ROOT_DIR` is a destination folder to install ATOMIC module of SALOME. By default (if `CMAKE_INSTALL_PREFIX` option is not given), ATOMIC module will be configured for installation to the `/usr` directory that requires root permissions to complete the installation.

  Invoke `make` and `make install` commands to build and install ATOMIC module:

  ```
  [%] make
  [%] make install
  ```

2.5 RUNNING SALOME WITH ENABLED ATOMIC MODULE

To launch SALOME session with the ATOMIC module, it's necessary to perform the following steps:

- Set runtime environment:

  Set paths to products and standard SALOME modules (environment scripts `env_products.sh` and `env_products.csh` are the part of the SALOME installation procedure):

  ```
  [%] source env_products.sh
  ```

  Supplement environment with ATOMIC installation directory path:

  ```
  [%] export ATOMIC_ROOT_DIR=your_install_path
  ```

- Launch SALOME.

To launch Salome with some modules available, for example GEOM and ATOMIC, use `runSalome` command with `--modules` option:

```
[%] ${KERNEL_ROOT_DIR}/bin/salome/runSalome --modules=GEOM,ATOMIC
```

This command starts SALOME GUI session. The «Modules» toolbar will contain two items: «Geometry» and «Atomic».

- Quit SALOME GUI

  Use menu *File* → *Exit* to close SALOME GUI desktop.

Now you know how to build SALOME modules and run SALOME. Let's use this knowledge to study ATOMIC: light-weight component.

3. ATOMIC: LIGHT-WEIGHT COMPONENT

This chapter is dedicated to so-called "light-weight" components of SALOME platform. What is a SALOME component is explained in 6.1.2 chapter. The "lightness" of a light-weight component consists in exclusion of SALOME platform services that are based on CORBA technology. Light-weight components do not have a CORBA-based engine and do not use any CORBA-based services of SALOME platform (implemented in SALOMEDS, NamingService, Container, and other packages of KERNEL module).

Here we have to say a few words about different modes of building and running of SALOME applications. 2 core modules of SALOME platform - KERNEL and GUI - can be compiled and run in 2 configurations (can be understood as *modes*): **full** and **light**.

- Light configuration means that all CORBA-based services are disabled. To build the modules in "light" configuration `-DSALOME_LIGHT_ONLY=ON` option must be passed to the `cmake` command (see paragraph 2 for details). To run SALOME in the light configuration, the command `runLightSalome.csh` (or `runLightSalome.sh`) from GUI module is used.

- Building in full configuration enables all CORBA services (`-DSALOME_LIGHT_ONLY=OFF` parameter of `cmake` command, this option is used by default). To run SALOME in the full configuration a command `runSalome` from KERNEL module is used.

A light-weight component can work with SALOME KERNEL and GUI modules built in both light and full configurations. In other words, a light-weight component can be a part of a multi-component SALOME application, and the other components do not have to be necessarily light-weight. But if all components are light-weight (in particular, if there is only 1 light-weight component in an application), then it is preferable to use KERNEL and GUI modules in light configuration. This will increase the application performance since a number of unused CORBA-based services will not be started.

Studying this chapter step by step, we shall create a simple light-weight component ***ATOMIC***. Beginning with the next section – « Instantiating a GUI module » - we start the development of our component. First we create a GUI module class, then create its internal data structure (Component with data section), implement persistence of the data (Implementing persistence section), learn how to display the data in Object Browser (Working with Object Browser section), set and retrieve selected objects and display a popup menu (Selection management section). At last, we shall implement several new functions wrapped into Operation objects.

Having completed this chapter of the tutorial, the sample light-weight component should look like shown at the Figure 1.

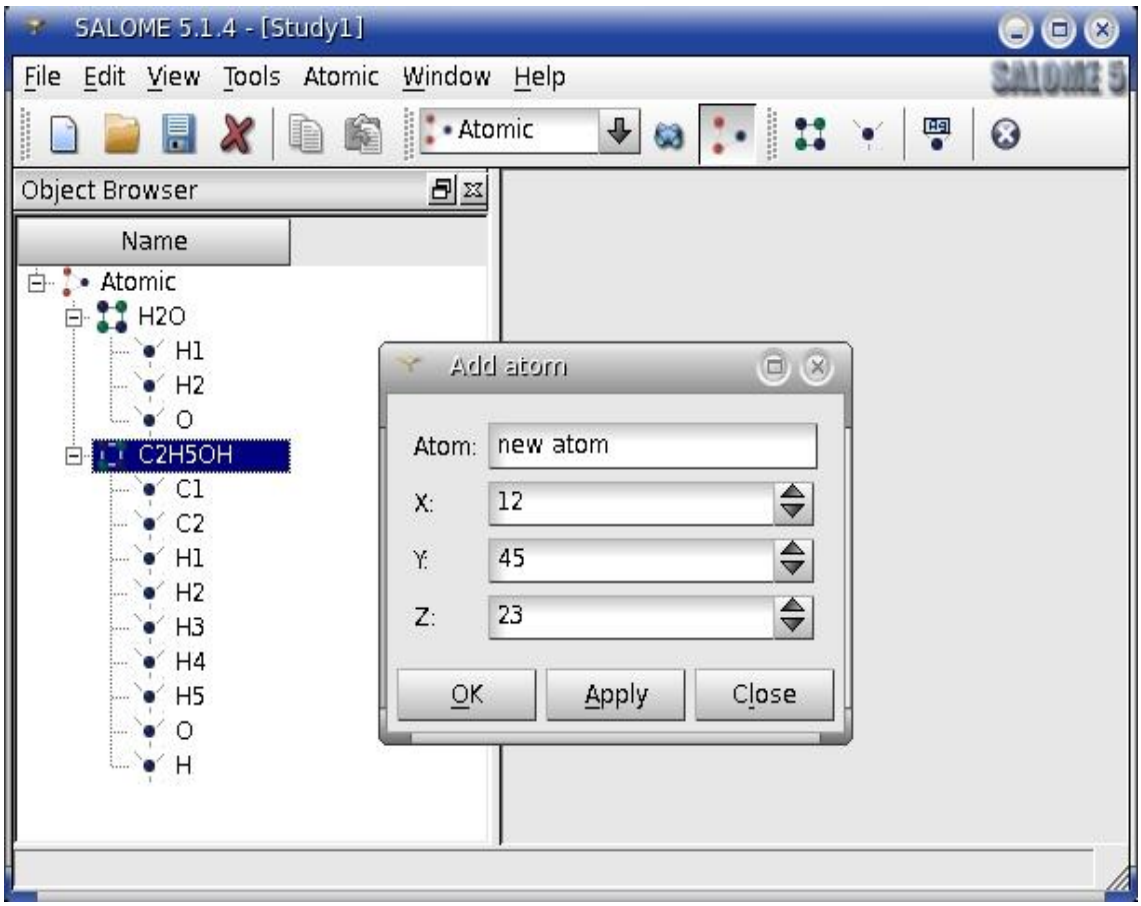

Figure 1. ATOMIC module

The ***ATOMIC*** component will allow creating "molecule" objects (compound) and "atom" objects (single objects with a "molecule" as a parent). "Atom" objects will have 3 integer attributes (X, Y, Z coordinates). "Molecules" and their "atoms" will be shown in Object Browser. Additionally we shall implement the following functions: renaming and removal of "molecules" and "atoms", import and export of data ("molecules" and "atoms") to/from file in XML format.

3.1 INSTANTIATING A GUI MODULE

We shall start the development of a new GUI module class for ATOMIC component with a build stub. The stub contains structure of directories, necessary utility files (`CMakeLists.txt`), and 2 source files: `ATOMICGUI.h` and `ATOMICGUI.cxx` - starting point for our GUI module development.

Please, unpack the stub archive file ATOMIC module source files (initial stub) to your working directory. Set environment, create ATOMIC build directory, run `cmake`, `make` and `make install` commands. For details on SALOME build procedure please refer to the paragraph 2.

An application can be started now, new Study can be created and ATOMIC module can be loaded. But it has no controls and does absolutely nothing.

Let's take a look at `ATOMIC.h` and `ATOMIC.cxx` (comments are excluded):

ATOMIC.h:

```
class ATOMICGUI: public LightApp_Module
{
  Q_OBJECT
public:
  ATOMICGUI();
};
```

ATOMIC.CXX:

```
ATOMICGUI::ATOMICGUI()
: LightApp_Module( "ATOMICGUI" )
{
}
extern "C" {
  ATOMICGUI_EXPORT CAM_Module* createModule() {
    return new ATOMICGUI();
  }
}
```

So we have an empty module and an "`extern C`" function for its instantiation. A global "`extern C`" function that creates a GUI module instance is used in SALOME for dynamic loading of a component at run time. It should be always declared using exactly the same "signature":

```
extern "C" { CAM_Module* createModule(); }
```

A component is loaded using the following sequence of actions:

- Resource file(s) are parsed. Resource file(s) contain a lot of customizable information which is used during application start up and run. The information contains instruction which components should be activated, which CORBA servers started (in full configuration), and many other important things. A general rule for resource files location is: first files listed in `LightAppConfig` (light-weight configuration) or `SalomeAppConfig` variable value (file names must be separated by ';' or ':' symbol) are parsed, then a user-dependent resource file `~/.SalomeApprc.<version>` is parsed. Commands `runSalome` and `runLightSalome` extend this variable automatically using according `<module_name>_ROOT_DIR` values.

- Components to be activated in the current SALOME session are listed in resource files (section "`launch`", parameter "`modules`"). Names of GUI modules library files (`.so` or `.dll`) are either dynamically constructed at run time (on Linux the algorithm is: "`lib`"+<component_name>+"`.so`") or they are explicitly indicated in the resource file.

- When a certain module is activated (selected in components combo box or a corresponding tool button is pressed), its library is loaded into memory, an "`extern C`" function `createModule()` is dynamically located in the library and called to retrieve a GUI module object.

- As soon as GUI module object is constructed, it receives control of execution, creates necessary menu items, view windows, etc. A component is ready to operate.

Below we shall examine the functions that are called on GUI module object when it is activated and deactivated:

- **`initialize()`**: Called when a component is activated for the first time. Function `initialize()` is called only once for each GUI module object created. It is the best place for code that creates actions, menu items, tool buttons, operations, popup menu items, etc.

- **`activateModule()`**: Called every time a component is activated (a corresponding item is selected in components combo box or a tool button in components tool bar is pressed). Usually this function displays menu items and tool buttons (which were already created in `initialize()`) and installs customized selection managers (not used in ATOMIC component).

- **`deactivateModule()`**: Called every time when a component is deactivated (another component is activated or a study is closed). Usually this function hides menus and tool buttons of a component and deactivates customized selection managers.

Let's assume we have implemented these functions and created 2 actions for atoms and molecules creation. We shall also create the corresponding menu items and tool buttons and connect their actions to one common slot (which will do nothing for the moment). The GUI module class of ATOMIC component will now look in the following way:

ATOMICGUI.h
ATOMICGUI.cxx
    initialize()
    activateModule()
    deactivateModule()
    onOperation()
    extern "C" createModule()

```
#if !defined(ATOMICGUI_H)
#define ATOMICGUI_H

#include <LightApp_Module.h>

/*!
 * Class       : ATOMICGUI
 * Description : GUI module class for ATOMIC component
 */
class ATOMICGUI: public LightApp_Module
{
  Q_OBJECT
  enum { agCreateConf, agAddAtom };

public:
  ATOMICGUI();
  virtual void initialize ( CAM_Application* );

public slots:
  virtual bool activateModule   ( SUIT_Study* );
  virtual bool deactivateModule ( SUIT_Study* );

private slots:
  void         onOperation();
};

#endif // ATOMICGUI_H
```

```
using namespace std;

#include "ATOMICGUI.h"

#include <LightApp_Application.h>

#include <SUIT_ResourceMgr.h>
#include <SUIT_Session.h>
#include <SUIT_Desktop.h>

/*! Constructor */
ATOMICGUI::ATOMICGUI()
: LightApp_Module( "ATOMICGUI" )
{
}

/*! Initialization function.
    Called only once on first activation of GUI module.
*/
void ATOMICGUI::initialize ( CAM_Application* app )
{
  LightApp_Module::initialize( app ); // call parent implementation
```

```
  QWidget* parent = application()->desktop();
  SUIT_ResourceMgr* resMgr = SUIT_Session::session()->resourceMgr();

  // create actions
  createAction( agCreateConf, tr( "TOP_CREATE_CONF" ),
      resMgr->loadPixmap( "ATOMIC", tr( "ICON_ATOMIC_CONF" ) ),
      tr( "MEN_CREATE_CONF" ), tr( "STB_CREATE_CONF" ), 0, parent,
      false, this, SLOT( onOperation() ) );
  createAction( agAddAtom,    tr( "TOP_ADD_ATOM" ),
      resMgr->loadPixmap( "ATOMIC", tr( "ICON_ATOM" ) ),
      tr( "MEN_ADD_ATOM" ),    tr( "STB_ADD_ATOM" ),    0, parent,
      false, this, SLOT( onOperation() ) );

  // init popup menus
  int aAtomicMnu = createMenu( tr( "MEN_ATOMIC" ), -1, -1, 50 );
  createMenu( agCreateConf, aAtomicMnu, 10 );
  createMenu( separator(),  aAtomicMnu, -1, 10 );
  createMenu( agAddAtom,    aAtomicMnu, 10 );

  // create toolbar
  int tbId = createTool( tr( "MEN_ATOMIC" ) );
  createTool( agCreateConf, tbId );
  createTool( agAddAtom,    tbId );
}

/*! Activation function. Called on every activation of a GUI module.
*/
bool ATOMICGUI::activateModule ( SUIT_Study* study )
{
  bool isDone = LightApp_Module::activateModule( study );
  if ( !isDone ) return false;

  setMenuShown( true );
  setToolShown( true );

  return isDone;
}

/*! Deactivation function.
    Called on every deactivation of a GUI module.
*/
bool ATOMICGUI::deactivateModule ( SUIT_Study* study )
{
  setMenuShown( false );
  setToolShown( false );

  return LightApp_Module::deactivateModule( study );
}

/*! slot connected to all functions of the component
    (create molecule, add atom, etc.)
*/
void ATOMICGUI::onOperation()
{
  if( sender() && sender()->inherits( "QAction" ) )
  {
    int id = actionId( ( QAction* )sender() );
    printf( "An operation with ID = %d was called\n", id );
  }
}

#if defined(WNT)
```

```
#define ATOMICGUI_EXPORT __declspec(dllexport)
#else   // WNT
#define ATOMICGUI_EXPORT
#endif  // WNT

/*! GUI module instantiation function */
extern "C" {
  ATOMICGUI_EXPORT CAM_Module* createModule() {
    return new ATOMICGUI();
  }
}
```

The whole application at this point of our development can be downloaded using this link, unzipped and built.

Please, build the application and run it. Now we can see that the Desktop has Atomic menu with sub items and a new tool bar:

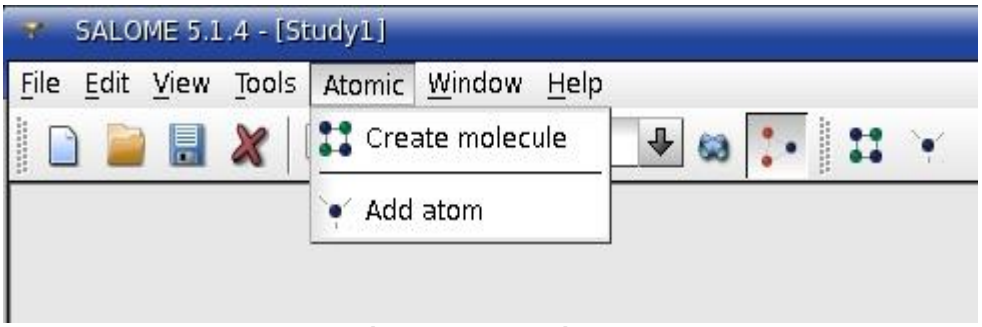

Figure 2. « Atomic » menu

Please, pay attention at using a new object for string and bitmap resources allocation described in SALOME concepts chapter: Resource manager. Resource manager uses resource file pointed by `LightAppConfig` / `SalomeAppConfig` variable for location of directories with resources. In our case, `LightApp.xml` (or `SalomeApp.xml`) file contains the following lines that tell Resource manager to use a certain directory for location of ATOMIC resources:

```
<section name="resources" >
  <parameter name="ATOMIC"
   value="${ATOMIC_ROOT_DIR}/share/salome/resources/atomic"/>
</section>
```

This directory (`${ATOMIC_ROOT_DIR}/share/salome/resources/atomic`) after building of ATOMIC component contains `.qm` files (prepared by `lrelease` tool from Qt toolkit) for bitmap and string resource allocation. File names must follow certain rule:

- Bitmap resources file name: `<Resource_name>_images.qm` [`ATOMIC_images.qm`]
- Textual resources file name: `<Resource_name>_msg_<language>.qm` [`ATOMIC_msg_en.qm`]

For more information on using `qm` files and internationalization support by Qt toolkit, please, refer to Qt API reference on Qt web site: `QTranslator` class.

At this point we have built a GUI module for a light-weight component with a set of commands accessible through menus and tool buttons. The goal of our component is to prepare data for its further processing. In the next section we shall develop an internal data model for our component and implement its persistence.

3.2 COMPONENT WITH DATA

A component stores its data by means of Data Model - a very important concept of SALOME platform. Data Model is a somewhat manager of data within a component. It plays a role of interface for accessing the data: retrieval, removal, and modification. It also implements persistence of data: saving to external file(s) and reconstruction of internal data structure from the file(s).

The data itself can be organized in any way a programmer desires. In our simple case we will use a list of values, more complicated data structures can be implemented using complex tools for data storage and retrieval (as CORBA-based `SALOMEDS` library or even SQL servers), but the common gateway for accessing the data will be Data Model.

Data Model represents arbitrary internal data in a tree-like structure. It is done through `root()` method of `CAM_DataModel` class. It returns object of the highest level of a component, usually this object represents the component itself. Here we have to say a few words about what kind of object is returned by Data Model. This object is a called a Data Object. Its primary mission is to provide a common view of arbitrary data. It is a proxy-object - it hides the real implementation of data and provides a generic interface to accessing it by other objects. For example, Object Browser "knows" how to display Data Objects, Selection Manager "knows" how to select Data Objects, and only Data Object itself "knows" which real piece of data was accessed (displayed, selected, etc.) through it.

OK, let's return to our ATOMIC component and develop a data structure for it. First of all, we have to develop a data itself. In our simple case, we shall use a simple list of values to represent a set of molecules. Each value - is an object of a class that we are going to develop. Let's assume that we have done it :), and here it is:

```
class ATOMICGUI_AtomicMolecule
{
private:

  class Atom
  {
  public:
    Atom();
    Atom(const QString& name, const double x, const double y, const
double z);
    QString name() const { return myName; }
    double  x()    const { return myX;    }
    double  y()    const { return myY;    }
    double  z()    const { return myZ;    }

    int     id()   const { return myId;   }

  private:
    QString    myName;
    double     myX;
    double     myY;
    double     myZ;
    int        myId;
    static int myMaxId;

    friend class ATOMICGUI_AtomicMolecule;
  };

public:
  ATOMICGUI_AtomicMolecule( const QString& name = QString::null );
  virtual ~ATOMICGUI_AtomicMolecule();

  void addAtom( const QString& atom, const double x, const double y,
const double z );
```

```
  void deleteAtom( const int index );

  int         id     () const { return myId; }
  QString     name   () const { return myName; }
  int         count  () const { return myAtoms.count(); }

  int         atomId   ( const int index ) const;
  QString     atomName ( const int index ) const;
  double      atomX    ( const int index ) const;
  double      atomY    ( const int index ) const;
  double      atomZ    ( const int index ) const;

  void        setName( const QString& name, const int index = -1 );

private:
  QString          myName;
  QList<Atom>      myAtoms;
  int              myId;
  static int       myMaxId;
};
```

This class represents one molecule that consists of several atoms. The data structure of ATOMIC component consists of a set of molecules. How do we do it and where do we store this set? As we have described above, the best place for it would be a Data Model. We develop `ATOMICGUI_DataModel` class, and its header file will look like this:

```
class ATOMICGUI_DataModel : public LightApp_DataModel
{
  Q_OBJECT

public:
  ATOMICGUI_DataModel ( CAM_Module* );
  virtual ~ATOMICGUI_DataModel();

  bool   createMolecule ();
  bool   addAtom (const QString& moleculeID, const QString& atomName,
                  const double x, const double y, const double z);

private:
  QList<ATOMICGUI_AtomicMolecule> myMolecules;
};
```

The private member field `myMolecules` is basically all data structure of our component. `createMolecule()` and `addAtom()` methods allow to add new objects to the data structure. Please, take the source file of the current state of ATOMIC component, study it carefully and build it. We have implemented a virtual function of GUI module class which creates Data Model. This function will be called automatically when new study is created. Also Data Model will receive a number of callbacks for saving and restoring of it data - we shall implement them later. We also added functionality to `onOperation()` slot of `ATOMICGUI` class, so that pressing "Create molecule" tool button (or selecting a menu item) creates a real object in our data structure. In the future we shall replace this code (we shall use Operation object). Also it is not currently possible to add Atoms to molecules since we do not know how to identify a molecule to be used (wait until we learn Selection management). *"Everything is good in its season"!*

Now we are going to develop a Data Object for our model. As it was mentioned above, Data Object plays a role of proxy - it hides the real implementation of data and provides a generic interface for accessing it by other objects. For ATOMIC component we shall develop one Data Object class to represent both Molecule and Atom objects. Another successor of Data Object that we will develop represents a root object - parent of all our Molecules and Atoms.

```
class ATOMICGUI_DataObject : public virtual LightApp_DataObject
{
public:
  ATOMICGUI_DataObject (SUIT_DataObject*,
        ATOMICGUI_AtomicMolecule* = 0, int = -1);
  virtual ~ATOMICGUI_DataObject();

  virtual QString entry() const;

  QString    name()    const;
  QPixmap    icon(const int = NameId)    const;
  QString    toolTip(const int = NameId) const;

  ATOMICGUI_AtomicMolecule* molecule() const { return myMolecule; }
  int        atomIndex()  const { return myIndex; }

  bool       isMolecule() const;
  bool       isAtom() const;

private:
  ATOMICGUI_AtomicMolecule*  myMolecule;
  int                        myIndex;
};

class ATOMICGUI_ModuleObject : public ATOMICGUI_DataObject,
                               public LightApp_ModuleObject
{
public:
  ATOMICGUI_ModuleObject ( CAM_DataModel*, SUIT_DataObject* = 0 );

  virtual QString name()    const;
  QPixmap         icon(const int = NameId)    const;
  QString         toolTip(const int = NameId) const;
};
```

Method `entry()` of Data Object is inherited from `LightApp_DataObject` class. This method must return a unique string for every object. It is used for unique object "key" generation which in turn is used for object compare (equal, less, etc.). We shall implement it as follows:

```
QString ATOMICGUI_DataObject::entry() const
{
  QString id = "root";
  if ( myMolecule )
    id = QString::number( myMolecule->id() );
  if ( myIndex >= 0 )
    id += QString( "_%1" ).arg( QString::number(
          myMolecule->atomId( myIndex ) ) );
  return QString( "ATOMICGUI_%1" ).arg( id );
}
```

Such implementation returns an unique string for all 3 kinds of ATOMIC objects: root object, molecule, and atom.

The next step is building a tree of Data Objects and setting the root of this tree as a root of the Data Model. The best place for this functionality is virtual method `build()` of Data Model class. Let's implement it:

```
void ATOMICGUI_DataModel::build()
{
  CAM_ModuleObject* modelRoot =
    dynamic_cast<CAM_ModuleObject*>( root() );
  if( !modelRoot )  {  // root is not set yet
    LightApp_Study* study =
```

```
      dynamic_cast<LightApp_Study*>( module()->
                    application()->activeStudy());
    modelRoot = createModuleObject( study->root() );
    setRoot( modelRoot );
  }
  // create 'molecule' objects under model root object
  // and 'atom' objects under 'molecule'-s
  for ( int i = 0; i < myMolecules.count(); i++ ) {
    ATOMICGUI_DataObject* molObj =
      new ATOMICGUI_DataObject ( modelRoot, &myMolecules[i] );
    for ( int j = 0; j < myMolecules[ i ].count(); j++ ) {
      /*ATOMICGUI_DataObject* atomObj = */
      new ATOMICGUI_DataObject ( molObj, &myMolecules[i], j );
    }
  }
}
```

A root object is an instance of `ATOMICGUI_ModuleObject` class, its children are molecule objects, their children - atom objects. We shall also implement several utility methods in our Data Model:

```
bool renameObj ( const QString& entry, const QString& newName );
bool deleteObjs ( const QStringList& listOfEntries );
ATOMICGUI_DataObject* findObject   ( const QString& entry );
ATOMICGUI_DataObject* findMolecule ( const QString& entry );
```

All these methods operate with entries. Having implemented `entry()` method of Data Object it is much easier now to operate with these entries - unique identifiers of Data Objects of all types: root Data Object, molecules, and atoms.

Please, take source files of the current state of ATOMIC component. Study them to understand the following issues:

- How the real data (classes declared in `ATOMICGUI_Data.h` file) is "wrapped" with Data Object interface.
- How a tree of Data Objects is constructed.
- How a certain molecule or atom is accessed (modified) using its Data Object.

As it is still not possible to create atoms of a certain molecule (we simply do not know yet how to select a certain molecule), we have added a temporary code that creates 3 atoms for any new molecule in `createMolecule()` method.

Calling `root()->dump()` in `ATOMICGUI_DataModel::build()` method displays in the terminal the current structure of Data Objects. After we learn how to display the objects in Object Browser, there will be no need in this `dump()`.

In the next section we shall implement persistence of our data.

3.3 IMPLEMENTING PERSISTENCE

Data Model has a number of virtual functions that are called by Study when it is being saved to a file or restored from a file. The functions are: `save()`, `saveAs()`, `open()`. The algorithm of saving of a Study is the following: Study iterates active components, retrieves their Data Models and calls `save()` or `saveAs()` functions. In `save()` and `saveAs()` a Data Model saves its data to a temporary file(s) in arbitrary format and returns (in out-parameter) a list of file names which contain the saved data. The first file name must be a name of a directory, the next file names (any number of them) - names of files in this directory. If we go deeper into persistence implementation details, we'll see that contents of these files will be serialized into a binary stream, this stream will be saved into one single file (or multiple files, depends on settings), and afterwards

the original file(s) created by Data Model will be deleted. During opening of a Study the algorithm is exactly the opposite: from a single file (or multiple files) a stream is created, it restores the temporary files for Data Models (in the same way as Data Model saved them, only location of these files may change), and finally `open()` function of a Data Model is called receiving the temporary file name(s) on its input. It must restore the internal data structure from the files.

Such scheme of persistence is very flexible. It allows for saving of a Study that contains data of multiple components into one single file. Each component follows its own rules for saving of its own data; the information that it presents is only a file name where it saved the data.

Let's take a look at ATOMIC component. The following virtual functions must be redefined in Data Model:

```
virtual bool open   ( const QString&, CAM_Study*, QStringList );
virtual bool save   ( QStringList& );
virtual bool saveAs ( const QString&, CAM_Study*, QStringList& );

virtual bool isModified () const;
virtual bool isSaved    () const;
```

The functions `isModified()` and `isSaved()` are called by application to enable/disable "Save" and "Save As" menu item and tool button.

We also have to add 2 functions to do the real saving of our "atomic" data structure to a file and restoring it from a file. We chose XML format and Qt DOM library for these needs.

```
bool  importFile ( const QString&, CAM_Study* = 0 );
bool  exportFile ( const QString& = QString::null );
```

Now let's take a look at implementation of `open()`, `save()`, and `saveAs()` functions:

```
bool ATOMICGUI_DataModel::open( const QString& URL,
                CAM_Study* study, QStringList listOfFiles )
{
  LightApp_Study* aDoc = dynamic_cast<LightApp_Study*>( study );
  if ( !aDoc )
    return false;

  LightApp_DataModel::open( URL, aDoc, listOfFiles );

  // The first list item contains path to a temporary
  // directory, where the persistent files was placed
  if ( listOfFiles.count() > 0 ) {
    QString aTmpDir ( listOfFiles[0] );

    // This module operates with a single persistent file
    if ( listOfFiles.size() == 2 ) {
      myStudyURL = URL;
      QString aFullPath = SUIT_Tools::addSlash( aTmpDir ) +
                  listOfFiles[1];
      return importFile( aFullPath, aDoc );
    }
  }

  return false;
}

bool ATOMICGUI_DataModel::save( QStringList& listOfFiles )
{
  bool isMultiFile = false;

  LightApp_DataModel::save( listOfFiles );
```

```
  LightApp_Study* study = dynamic_cast<LightApp_Study*>(
              module()->application()->activeStudy() );

  QString aTmpDir(study->GetTmpDir( myStudyURL.toLatin1(),
              isMultiFile ).c_str());

  QString aFileName = SUIT_Tools::file( myStudyURL, false ) +
              "_ATOMICGUI.xml";
  QString aFullPath = aTmpDir + aFileName;
  bool ok = exportFile( aFullPath );

  listOfFiles.append( aTmpDir );
  listOfFiles.append( aFileName );

  return ok;
}

bool ATOMICGUI_DataModel::saveAs ( const QString& URL,
             CAM_Study* study, QStringList& listOfFiles )
{
  myStudyURL = URL;
  return save( listOfFiles );
}
```

The member field `myStudyURL` is used to store the file name of last used persistent file. In `save()` we create an XML file, export our data into this file, and return in the out parameter the directory and the file name.

In the `open()` function we construct the file name using the first (directory name) and the second (file name) members of the input list, and call `importFile()` to restore the data structure from this file.

All we have to do now - is implement `importFile()` and `exportFile()` functions. This is already done in the sources of the current state of ATOMIC component. Please, save the source files and build the component. Now it is possible to save the study and restore it. We can see in the terminal that `dump()` after opening of a study outputs the same, previously saved data structure.

At this point we finish to develop the `ATOMICGUI_DataModel` class - now has all functionality that is supposed to have.

In the next section we learn how to use Object Browser for displaying the data structure, so we will not have to use `dump()` any more.

3.4 WORKING WITH OBJECT BROWSER

Object Browser is a view window based on QTreeView class, which can display the data structure of components based on Data Objects. Object Browser is shared among all components; it is created for every new (opened) Study. Components display their "portions" of data in Object Browser under their "root objects". On the Figure 3 below we can see 2 root objects of 2 components: Geometry component and Mesh component.

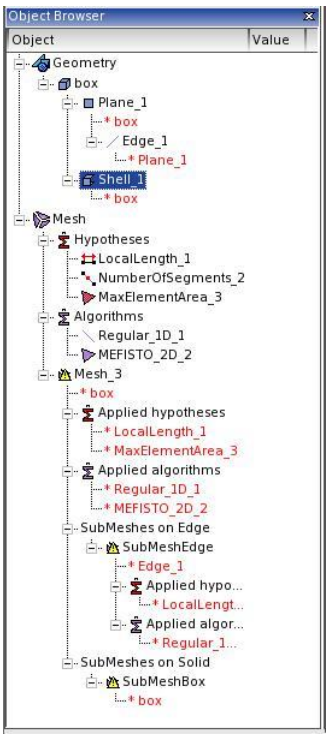

Figure 3. Object Browser

Object Browser supports selection of objects (setting and retrieving selected objects), and displaying a popup menu on them. Data Objects can provide different text for different columns in Object Browser. Standard set of columns includes 4 additional columns: "Value", "Entry", "IOR", and "Reference entry" (see Figure 4), but any number of additional columns can be added by a component.

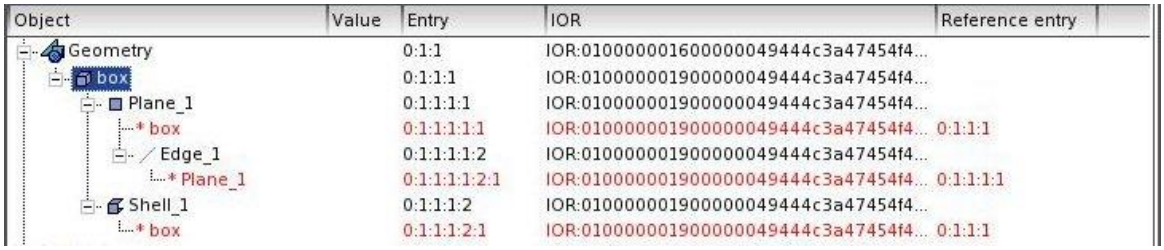

Figure 4. Object browser custom columns

Data Objects also provide an icon and a tool tip to be displayed in Object Browser. If we take a look again at Data Object of ATOMIC component, it has got methods that are used by object browser:

```
QString    name()    const;
QPixmap    icon()    const;
QString    toolTip() const;
```

OK, now we come to the first question: how does a component opens and accesses the Object Browser?

- To make the Object Browser available for a component, a component's GUI module class must redefine virtual method `windows()` and add a "flag" of Object Browser to out-parameter map. That's all!

```
void ATOMICGUI::windows ( QMap<int, int>& aMap ) cons
{
  aMap.insert( LightApp_Application::WT_ObjectBrowser,
               Qt::LeftDockWidgetArea );
}
```

Now Object Browser window is opened every time a component is activated. If user closes Object Browser window, then it can be opened again using menu *View* → *Windows* → *Object Browser*.

- A component accesses Object Browser through methods of `LightApp_Application::objectBrowser()`. So from a GUI module class: `getApp()->objectBrowser()`.

After the Object Browser is active, it automatically scans Data Models of active components, takes their root Data Objects (using method "`root()`"), and displays them in the list view. Please, take the source files of ATOMIC component with available Object Browser. The data structure is shown in Object Browser. In the next section we are going to learn how to retrieve selected objects from Object Browser and how to install different popup menus on different objects.

3.5 SELECTION MANAGEMENT

At this point of the tutorial, the data structure of ATOMIC component is fully implemented, it is possible to create ATOMIC entities, save and restore a Study, display it in the Object Browser. In this section we learn how to manage selection of objects - molecules and atoms - in Object Browser. Management of selection in other types of viewers is very similar and follows the same schema as we study here.

Selection management is handled by a class `LightApp_SelectionMgr` or its successors. So far, we have 2 "representations" of every entity of data structure: "core" object stored privately in Data Model (instance of `ATOMICGUI_AtomicMolecule` class) and a corresponding Data Object (instance of `ATOMICGUI_DataObject` class). Selection manager uses yet another "representation" of data. It is called a Data Owner object. Usually it is not necessary to redefine Data Owner class in a custom component, a standard `LightApp_DataOwner` class is sufficient. Data Owner is a very simple class; it stores only an entry of a selected Data Object. As we remember, "entry" is a unique "key" of a piece of data. We implemented the method `entry()` in `ATOMICGUI_DataObject` class. It returns different string keys for different objects (and the same string keys for the same objects). Data Owner objects can be treated as entries wrapped into instances of a class. The wrapping of a string entry into a class is done because some components may extend the Data Owner class so it stores not only entries, but object type information, object properties information, etc. In any case, Data Owner always represents a selected entity.

Selection may be performed not only in Object Browser, but in other windows that can display data in a custom way (3D views, 2D plots, etc.). Every view window has its own "selector" - a class that performs "conversion" of data from the format used by this viewer to Data Owner and reverse. This is the main purpose of Data Owner - store selected entities independently from the source of selection. Let's demonstrate how it works in terms of Object Browser:

- When selection event is emitted in the list view of Object Browser, the global Selection Manager receives this event and requests the selector of Object Browser to return the selected objects. Selector of Object Browser scans its Data Objects and creates a Data Owner for every selected Data Object. Entry of a new Data Owner is set equal to the entry of the corresponding Data Object.

- When it is needed to select objects in Object Browser (synchronize selection, reflect to selection in a dialog box, etc.), then the global Selection Manager receives a list of Data Owners to be selected. It passes it to the selector of Object Browser. The selector finds Data Objects with the same entries as given Data Owners and selects them.

When we say "global Selection Manager" we mean an object of `LightApp_SelectionMgr` class which is stored in `LightApp_Application` class instance. When we create an Object Browser, its selector is also automatically created and registered in the Selection Manager. After the registration, the selector will receive external selection events from the Selection Manager, and the Selection Manager will receive the selected objects (list of Data Owners) from the selector.

OK, let's implement a method in `ATOMICGUI` class, which will return a list of strings - entries of the selected objects.

```
void ATOMICGUI::selected( QStringList& entries, const bool multiple )
{
  LightApp_SelectionMgr* mgr = getApp()->selectionMgr();
  if( !mgr )
    return;

  SUIT_DataOwnerPtrList anOwnersList;
  mgr->selected( anOwnersList );

  for ( int i = 0; i < anOwnersList.size(); i++ )
  {
    const LightApp_DataOwner* owner =
       dynamic_cast<const LightApp_DataOwner*>
       ( anOwnersList[ i ].get() );
    QStringList es = owner->entry().split( "_" );
    if ( es.count() > 1 && es[ 0 ] == "ATOMICGUI" &&
         es[ 1 ] != "root" )
    {
      entries.append( owner->entry() );
      if( !multiple )
        break;
    }
  }
}
```

In this method we receive from the Selection Manager the list Data Owners, iterate them and then select only "our" entries. We also exclude "`ATOMICGUI_root`" entry as it is the root entry of the Data Model and does not correspond to any molecule or atom.

Now we shall implement creation of atoms of a certain molecule. We remove the code which creates 3 atoms of any new molecule from Data Model class and add the following code to `onOperation()` slot of `ATOMICGUI` class:

```
if ( id == agAddAtom ) {
  QStringList entries;
  selected( entries, false );
  ATOMICGUI_AddAtomDlg dlg ( getApp()->desktop() );
  int res = dlg.exec();
  ATOMICGUI_DataModel* dm = dynamic_cast<ATOMICGUI_DataModel*>
                            ( dataModel() );
  if( dm && res == QDialog::Accepted && dlg.acceptData( entries ) ) {
    QString name;
    double x, y, z;
    dlg.data( name, x, y, z );
    dm->addAtom( entries.first(), name, x, y, z );
    getApp()->updateObjectBrowser();
  }
}
```

`ATOMICGUI_AddAtomDlg` is a very simple dialog that allows user to input a name of a new atom and 3 coordinates: X, Y, and Z. Please, download the source files of the current version of ATOMIC with selection management, and enjoy atoms creation under selected molecules!

Another aspect of selection management which we are going to study in this section is management of popup menus. Popup menu is shown when user clicks a right mouse button in any view window including Object Browser. This view window in terms of popup management is called "a client window". If there are selected objects, a popup menu should contain commands that use

these objects. If there are no selected objects - popup contains commands applicable for the client window (change it background color, refresh view, etc.).

A GUI module class can add its commands to the popup menu. It can be done in 2 ways described in details in next sections.

3.5.1 Popup menu handling with `contextMenuPopup()` method

Redefine virtual function `void contextMenuPopup()`. This function is called by `LightApp_Application` class on popup menu request event. The module can analyze the type of client window - parameter "client" (for Object Browser it will be equal to "ObjectBrowser" string), the current selection, and fill the given popup menu with necessary items. Let's implement this function in ATOMICGUI class:

```
void ATOMICGUI::contextMenuPopup( const QString& client,
                  QMenu* menu, QString& /*title*/ )
{
  if ( client == "ObjectBrowser" ) {
    QStringList entries;
    selected( entries, false );
    if ( entries.size() ) {
      QStringList es = entries.first().split( "_" );
      if ( es.count() == 2 && es[ 0 ] == "ATOMICGUI" ) {
        // selected object belongs to ATOMICGUI
        // and it is a molecule object
        manu->addAction( action( agAddAtom ) );
      }
    }
  }
}
```

Please, replace `ATOMICGUI.h` and `ATOMICGUI.cxx` files in your version with these ones: ATOMICGUI.h with contextMenuPopup(), ATOMICGUI.cxx with contextMenuPopup() and recompile the component (`make` command). Popup menu shown in Object Browser with a molecule object selected now has 1 item added by our module: "Add Atom" action.

3.5.2 Popup menu manager

The approach described in previous paragraph is very simple and straightforward. It is suitable for components with simple logic of popup menu construction (small number of object types and/or small number of commands). The second approach to add items to popup menu is more advanced. It consists in redefinition of a class which performs analysis of selected objects in terms of their properties that influence popup menu construction. For example: property "type of object" determines if a certain item should be added to popup or not; property "visibility of object" determines which one of the items "Display" or "Hide" should be added to the popup. A custom class that will perform analysis of properties of an object must inherit `LightApp_Selection` class. GUI module returns an instance of this class in its virtual method `createSelection()`.

Determining of the properties of objects that affect popup menu items is the first step. The second step is generation of logical rules that check the values of properties (returned by a successor of `LightApp_Selection` class) and decide - if a certain item must be added to popup menu or not. The object that stores these rules is called a Popup Manager - instance of `QtxPopupMgr` class.

In GUI module of ATOMIC component we will have the following set of rules for its actions:

```
QString rule = "client='ObjectBrowser' and selcount=1 and
                type='Molecule'";
popupMgr()->setRule( action( agAddAtom ),  rule );

rule = "client='ObjectBrowser' and selcount=1 and type='Root'";
popupMgr()->setRule( action( agCreateMol ),rule );

rule = "($type in {'Molecule' 'Atom'}) and client='ObjectBrowser'
```

```
        and selcount=1";
popupMgr()->setRule( action( agRename ),   rule );

rule = "($type in {'Molecule' 'Atom'}) and client='ObjectBrowser'
        and selcount>0";
popupMgr()->setRule( action( agDelete ),   rule );
```

As you can notice, we add 2 new commands - Rename and Delete. We are not implementing these operations yet, for the moment we only add them to popup menu.

Let's take a closer look at the logical rules. A rule `"client='ObjectBrowser' and selcount=1 and type='Molecule'"` in natural language would be: "a client window must be Object Browser AND there should be exactly 1 object selected AND the type of the object should be 'Molecule'". This rule is set for "Add Atom" action.

A construct "`$type in {'Molecule' 'Atom'}`" means "value of 'type' property must be equal to one of the following: 'Molecule' or 'Atom'". Another rule could look like this: "`'Molecule' in $type`", and that would mean "there should be at least one object, for which the value of a property 'type' would be equal to 'Molecule'". The rules can be combined with "and" and "or" operators.

The function `setRule()` of the `QtxPopupMgr` class has 2 significant parameters (as we see in the code above).

Let's return to `LightApp_Selection` class. We must create a successor of this class and redefine its virtual methods `init()`, `count()`, and `param()` in order to compute the value of parameter `"type"` of the logical rules. The other parameters `"client"` and `"selcount"` are computed by `LightApp_Selection` class, and we can simply use them in our logical rules.

The `init()` method is called when a "request popup" event comes (user clicks a right mouse button). In this method the class must initialize its internal fields (lists, maps, etc. for calculation of parameters in `parameter()` method. `parameter()` returns a value of a certain parameter of an certain object (object index is given). And the last method `count()` returns the number of objects. Let's take a look at implementation of these methods in ATOMICGUI_Selection class:

```
void ATOMICGUI_Selection::init( const QString& client,
        LightApp_SelectionMgr* mgr )
{
  if ( mgr ) {
    SUIT_DataOwnerPtrList sel;
    mgr->selected( sel);
    SUIT_DataOwnerPtrList::const_iterator anIt  = sel.begin(),
                                          aLast = sel.end();
    for ( ; anIt != aLast; anIt++ ) {
      QString type = "Unknown";
      SUIT_DataOwner* owner = (SUIT_DataOwner*)( (*anIt).get() );
      LightApp_DataOwner* sowner =
           dynamic_cast<LightApp_DataOwner*>( owner );
      QStringList es = sowner->entry().split( "_" );
      if ( es.count() > 0 && es[ 0 ] == "ATOMICGUI" ) {
        if ( es.count() > 1 ) {
          if( es[ 1 ] == "root" )
            type = "Root";
          else
            type = "Molecule";
          if ( es.count() > 2 )
            type = "Atom";
        }
      }
      myTypes.append( type );
    }
  }
```

```
  LightApp_Selection::init( client, mgr );
}

QVariant ATOMICGUI_Selection::parameter( const int ind, const
QString& p ) const
{
  if ( p == "type" )
    return myTypes[ ind ];
  return LightApp_Selection::parameter( ind, p );
}

int ATOMICGUI_Selection::count() const
{
  return myTypes.count();
}
```

The `ATOMICGUI_Selection` class has a member field `myTypes` of `QStringList` type. It stores the types of selected objects. How the types are "calculated" using the entries of the objects - is shown in `init()` method. Method `param()` is very simple - it returns type of an object stored in `myTypes`.

Now we must compile all these changes together. Please, download the version of ATOMIC source file with advanced popup management.

At this point of our tutorial we have studied almost all topics related to development of a light-weight component. In the next and last section of light-component chapter we improve the way in which we implemented `onOperation()` slot of ATOMICGUI class. A new conceptual object of SALOME platform will be introduced and studied - the Operation. We also implement several new functionalities using Operations - import and export of data, renaming and removal of molecules and atoms.

3.6 OPERATIONS

Operation is a manager of an action inside a component GUI. By "action" we understand any functionality a GUI module of a component provides to a user. Examples of actions may be the following: creation of a sphere in Geometry component, Atom creation in ATOMIC component, mesh calculation in Mesh component.

Using an Operation for action management gives the following advantages:

- Action can be canceled, suspended, and resumed during its execution.
- An Operation instance can control which other Operations can be executed simultaneously with this Operation. It is implemented using method

  `bool isValid(SUIT_Operation* theOtherOperation) const.`

  Before starting a new operation (*operation_A*), an application calls `isValid()` method of the operation being executed (*operation_B*) passing it *operation_A* as a parameter. If *operation_B* returns false, then *operation_A* is not started, it must wait untill *operation_B* finishes its execution. This mechanism can be overridden, though, with yet another virtual method of SUIT_Operation class:

  `bool SUIT_Operation::isGranted() const.`

  If this method returns true, then the operation is started any way, ignoring `isValid()` return value.
- Operation has support for transaction mechanism.

For example, if user closes a study during execution of an operation, the Operation receives "Abort" signal (`abortOperation()` virtual function is called). The operation stops all algorithmic processing, closes the dialog windows it opened, aborts transaction, and frees all resources. Without using the Operation object it would be problematic to perform such smart deactivation of the action.

Base class for Operation object is `SUIT_Operation`, then it is inherited in `LightApp` package - `LightApp_Operation`. `SUIT_Operation` has the following virtual methods to be redefined in successors:

`bool isReadyToStart() const`	returns true if all initialization steps are done (location of resources, creation of dialog, etc.) and the operation is ready to be started.
`void stopOperation()`	called when operation must be stopped. It is called from abort() and commit() functions. Usually all dialog boxes are closed and resources are freed in this function.
`void startOperation()`	called when operation is started. It should start processing, display dialog boxes, etc. in this function.
`void abortOperation()`	called when operation is aborted. It should urgently stop all processing and free all resources.
`void commitOperation()`	called when operation is committed (normally finished). It is called from non-virtual commit() function which is usually called by the operation itself when everything is done.
`void resumeOperation()`	called when operation is started again after suspension.
`void suspendOperation()`	called when operation must be suspended. It should store the current state of processing, hide dialog boxes, etc.
`bool hasTransaction() const`	returns true if the operation uses any transaction mechanism (for keeping track of undo/redo or other means).
`bool abortTransaction()`	aborts transaction
`bool openTransaction()`	opens new transaction
`bool commitTransaction( const QString& = QString::null )`	commits transaction. The parameter is the name of a transaction.

`LightApp_Operation` class adds access to GUI module instance, Desktop, Selection manager from within the Operation, and adds 2 virtual methods for working with a dialog window:

```
LightApp_Dialog* dlg() const;
void setDialogActive( const bool );
```

If an Operation works with a dialog window, then its method `dlg()` should return it. It will be automatically shown in `start()`, and hidden in `abort()` or `commit()`.

OK, now we are going to develop Operations for the ATOMIC component. Custom Operation objects must be created in the virtual method of the GUI module class :

```
LightApp_Operation* createOperation( const int operationID ) const;
```

Let's create Operations for Atom and Molecule creation. First of all we create 2 classes - implementations of our custom Operations: `ATOMICGUI_CreateMolOp` and `ATOMICGUI_AddAtomOp`. We shall also create a special base class for Operations of ATOMIC component. Implementations of these classes will be the following (we already have the corresponding functionality in `onOperation()` slot of `ATOMICGUI` class, here we moved it to the new Operation classes):

```
ATOMICGUI_CreateMolOp:
ATOMICGUI_CreateMolOp::ATOMICGUI_CreateMolOp()
: ATOMICGUI_Operation()
{
}
ATOMICGUI_CreateMolOp::~ATOMICGUI_CreateMolOp()
{
}
void ATOMICGUI_CreateMolOp::startOperation()
{
  if( dataModel() && dataModel()->createMolecule() )
    commit();
  else
    abort();
}

ATOMICGUI_AddAtomOp:
ATOMICGUI_AddAtomOp::ATOMICGUI_AddAtomOp()
  : ATOMICGUI_Operation(),
    myDlg( 0 )
{
}
ATOMICGUI_AddAtomOp::~ATOMICGUI_AddAtomOp()
{
  if ( myDlg )
    delete myDlg;
}
LightApp_Dialog* ATOMICGUI_AddAtomOp::dlg() const
{
  if ( !myDlg )
    const_cast<ATOMICGUI_AddAtomOp*>( this )->myDlg =
      new ATOMICGUI_AddAtomDlg( module()->getApp()->desktop() );

  return myDlg;
}
void ATOMICGUI_AddAtomOp::onApply()
{
  QStringList entries;
  atomModule()->selected( entries, false );
  ATOMICGUI_AddAtomDlg* d =
    dynamic_cast<ATOMICGUI_AddAtomDlg*>( dlg() );
  if( dataModel() && d && d->acceptData( entries ) )
  {
    QString name;
    double x, y, z;
    d->data( name, x, y, z );
    dataModel()->addAtom( entries.first(), name, x, y, z );
    module()->getApp()->updateObjectBrowser();
  }
}
```

And the following modifications must be done to `ATOMICGUI` class:

```
void ATOMICGUI::onOperation()
{
```

```
  if( sender() && sender()->inherits( "QAction" ) )
  {
    int id = actionId( ( QAction* )sender() );
    startOperation( id );
  }
}
LightApp_Operation* ATOMICGUI::createOperation( const int id ) const
{
  switch( id )
  {
  case agCreateMol:
    return new ATOMICGUI_CreateConfOp();

  case agAddAtom:
    return new ATOMICGUI_AddAtomOp();

  default:
    return 0;
  }
}
```

The `onOperation()` slot of `ATOMICGUI` class simply calls `startOperation()` method passing the Operation identifier. The Operation objects are stored on the base level of GUI module (in `LightApp_Module` class) and starting the Operation multiple times does not create multiple Operation objects (can be understood as caching). But if the Operation is started for the first time, it is created by `createOperation()` method, which we have redefined in `ATOMICGUI` class in order to create custom Operations of our component.

The modifications are reflected in this version of ATOMIC source files. It also became possible to improve "Add atom" operation: `ATOMICGUI_AddAtomDlg` has 3 buttons now: Apply, Ok, and Close, so it is possible to create several atoms during one single operation pressing Apply button. Please, notice, that signals of the Operation dialog (`ATOMICGUI_AddAtomDlg` for example) are connected to virtual slots of base Operation class `ATOMICGUI_Operation`. Child Operations need to redefine these virtual functions to add customized processing.

ATOMIC component is almost finished. Finally, we are going to implement several other Operators for the following actions: import of data, export of data, renaming of atoms and molecules, and removal. The core functionality for these actions already exists in Data Model. All we have to do is create 4 new Operator classes, and add their creation to `createOperation()` method.

Let's take a look at new Operation for import and export (we shall combine these 2 actions into 1 Operator for simplicity):

```
ATOMICGUI_ImportExportOp::ATOMICGUI_ImportExportOp( const bool import
)
: ATOMICGUI_Operation(),
  myIsImport( import )
{
}
ATOMICGUI_ImportExportOp::~ATOMICGUI_ImportExportOp()
{
}
void ATOMICGUI_ImportExportOp::startOperation()
{
  ATOMICGUI_DataModel* dm = dataModel();
  if ( !dm )
  {
    abort();
    return;
  }

  QStringList filtersList;
```

```
  filtersList.append( tr( "XML_FILES" ) );

  // Select a file to be imported
  QString aFileName =
    module()->getApp()->getFileName( myIsImport, QString::null,
                                     filtersList.join( ";;" ),
                                     tr( myIsImport ?
                                     "ATOMICGUI_IMPORT_XML" :
                                     "ATOMICGUI_EXPORT_XML" ), 0 );

  if( !aFileName.isEmpty() )
  {
    if( ( myIsImport && dm->importFile( aFileName ) ) ||
        ( !myIsImport && dm->exportFile( aFileName ) ) )
    {
      commit();
      return;
    }
    else
      SUIT_MessageBox::warning ( application()->desktop(),
                   tr( "WRN_WARNING" ),
                   tr( myIsImport ?
                   "WRN_IMPORT_FAILED" :
                   "WRN_EXPORT_FAILED" ),
                   tr( "BUT_OK" ) );
  }
  abort();
}
```

The Operator retrieves a file name and calls `import()` or `export()` function of Data Model to perform the task.

The `onOperation()` slot of `ATOMICGUI` class will look like this:

```
LightApp_Operation* ATOMICGUI::createOperation( const int id ) const
{
  switch( id )
  {
  case agImportXML:
    return new ATOMICGUI_ImportExportOp( true );

  case agExportXML:
    return new ATOMICGUI_ImportExportOp( false );

  case agCreateConf:
    return new ATOMICGUI_CreateMolOp();

  case agAddAtom:
    return new ATOMICGUI_AddAtomOp();

  case agRename:
    return new ATOMICGUI_RenameOp();

  case agDelete:
    return new ATOMICGUI_DeleteOp();

  default:
    return 0;
  }
}
```

Please, download the source files of the final version of ATOMIC component using this link. As an

exercise, you can develop new Operations for ATOMIC component. "Delete All" or even "Copy / Paste" of atoms could be a very good practice and examination of the obtained knowledge!

3.7 IMPLEMENTING DUMP PYTHON

Dump python operation allows storing the state of SALOME study in form of the Python script. Resulting script is the fast way for restoring of the content of SALOME study.

The implementation of the dump python mechanism is different for the SALOME light-weight component and component with CORBA engine:

- To implement dump python in the light-weight component it is necessary to redefine virtual method `dumpPython()` declared in the `LightApp_DataModel` class. The signature of this method is:

```
virtual bool dumpPython( const QString&, CAM_Study*, bool,
QStringList& );
```

- To implement the dump python mechanism in the SALOME component with CORBA engine it is necessary implement virtual function `DumpPython()` of the module engine, a successor of the `Engines::EngineComponent` CORBA interface. The signature of this method is:

```
sequence<octet> DumpPython(in Object, in boolean, in boolean,
out boolean);
```

Take into account that component engine should be also inherited from the `SALOMEDS::Driver` interface and provide (at least empty) implementation of persistence methods. This requirement is stipulated by the architectural features of the SALOME data server.

3.7.1 Different approaches of the dump python mechanism implementation

Generally there are two main approaches for implementation of the dump python mechanism:

1. Each method of the custom SALOME component that publishes any data in the SALOME Study, also records some additional information to the component. This information can be later used for generation of the Python script, more precisely the part of the script that concerns the component. In the simple case it can be a string representation of the Python command with all required parameters and returning value(s) that reproduces the component's function being invoked. This information can be stored in arbitrary way, for example directly in the component's data model. Other approach is to use SALOMEDS attributes to store Python commands directly in SALOME study. Take into account that Dump python data should be persistent, i.e. it should be stored/retrieved during the study saving/loading. The described approach is called "historical dump".

   Advantages:
   - The dump python functionality in most cases can be trivially implemented.

   Disadvantages:
   - Main disadvantage of the historical dump is a problem of backward compatibility. Since the Python command is generated and stored directly at the moment of the function invocation, the maintenance of the studies (for example, in case of significant changing component API) becomes complicated task.

2. Dump python method analyzes current content of the SALOME Study, namely the part related to the component, and generates a Python script basing on the information retrieved from the study and the corresponding data stored in the component data model. This approach is called "snapshot dump" since it allows generation of minimal and sufficient script that reproduces the current content of the study, avoiding generation any intermediate commands (like data edition or removal commands). Usually this approach

does not imply storing any additional information in the Study since it dumps the current state only.

Advantages:
- No need to modify the existent data model of the custom component.
- No need to store any additional data in the study.
- No any “backward compatibility” problem.

Disadvantages:
- For complex data model, an implementation of the dump python function can be rather complicated task, especially in case of complex relations between data objects.

Thus, taking into account advantages and disadvantages of both approaches it is recommended to use the “historical” approach when creating a new component, since all required functionality can be initially included to the data model being implemented. The second can be applied when adding support of dump python mechanism to the existent component.

In fact, it’s possible to mix both approached when the most complicated objects in the custom component store the additional information thus simplifying the implementation of the dump python method and less complex objects are written in a Python script basing on analysis of their content, that allows minimizing of the modifications in the existing code.

3.7.2 Adding “snapshot dump” in ATOMIC module

The simplicity of the data model of the ATOMIC component allows applying “snapshot dump” approach to it.

First step of the dump python mechanism implementation in our component is creation of the Python interface, since currently ATOMIC component has no way to create and publish objects using Python interpreter. Firstly, let’s change return type of the `createMolecule()` method of the `ATOMICGUI_DataModel` class from `bool` to `QString`; now this method will return study entry (unique identifier) of the created molecule. Also, we will remove temporary debug code that automatically added three atoms to the just created molecule in `createMolecule()` function:

```
QString ATOMICGUI_DataModel::createMolecule()
{
  // add new molecule
  ATOMICGUI_AtomicMolecule mol;
  myMolecules.append( mol );
  // obtain its id (entry)
  QString id = QString( "ATOMICGUI_%1" ).arg( mol.id() );
  // update object browser
  update();
  // return entry of the created molecule
  return id;
}
```

The entry returned by the `createMolecule()` function will be required later in Python module in order to access the corresponding C++ data object.

Let’s adds new `AtomicPy` python module in our component. For wrapping C++ classes into Python we will use sip third-party open-source software (by Riverbank Computing Ltd). This seems to be natural choice, since sip is one of the SALOME pre-requisites; it provides a simple way to generate Python wrapping for C++ code, especially Qt-based one. `AtomicPy` library will contain `AtomicMolecule` class:

```
class TCreateMoleculeEvent: public SALOME_Event
{
public:
  QString myName;
  typedef QString TResult;
  TResult myResult;
```

```
  TCreateMoleculeEvent(const QString& name) :
   myResult(""), myName(name) {}
  virtual void Execute()
  {
    if ( !SUIT_Session::session() )
      return;
    LightApp_Application* app =
      dynamic_cast<LightApp_Application*>(
      SUIT_Session::session()->activeApplication() );
    if(!app)
      return;

    LightApp_Module* module =
     dynamic_cast<LightApp_Module*>( app->module("Atomic") );
    if(!module)
      return;

    ATOMICGUI_DataModel* model =
      dynamic_cast<ATOMICGUI_DataModel*>(module->dataModel());
    if(!model)
      return;

    myResult = model->createMolecule();
    model->renameObj(myResult,myName);
  }
};

AtomicMolecule::AtomicMolecule( const QString& name )
{
  myId = ProcessEvent( new TCreateMoleculeEvent( name ) );
}
```

Above code creates a molecule and publishes it in the study. The constructor `AtomicMolecule(const QString& name)` performs in such a way the same action as “Create molecule” function from GUI interface .

Below code adds new atom to the molecule, like the command “Add atom” from the GUI interface:

```
class TAddAtomEvent: public SALOME_Event
{
public:
  QString myId;
  QString myName;
  double myX, myY, myZ;

  TAddAtomEvent(const QString& id,
    const QString& name,
    const double x,
    const double y,
    const double z) : myId(id), myName(name), myX(x), myY(y), myZ(z)
{}
  virtual void Execute()
  {
    if ( !SUIT_Session::session() )
      return;
    LightApp_Application* app =
      dynamic_cast<LightApp_Application*>(
      SUIT_Session::session()->activeApplication() );
    if(!app)
      return;

    LightApp_Module* module =
```

```
      dynamic_cast<LightApp_Module*>( app->module("Atomic") );
    if(!module)
      return;

    ATOMICGUI_DataModel* model =
      dynamic_cast<ATOMICGUI_DataModel*>(module->dataModel());
    if(!model)
      return;

    model->addAtom(myId,myName,myX,myY,myZ);
  }
};
void AtomicMolecule::addAtom( const QString& atom,
                              const double x,
                              const double y,
                              const double z ){
  ProcessVoidEvent( new TAddAtomEvent( myId, atom, x, y, x ) );
}
```

Note, that all the functions modifying the contents of the study are wrapped by the events using SALOME events mechanism. This is important since all the Python commands executed in the SALOME embedded Python interpreter are serialized in order to avoid concurrent access to the Python interpreter from different threads that might lead to the application crashes.

To wrap our `AtomicMolecule` class into Python module we should describe it in the SIP specification file:

```
%Module AtomicPy

%Import QtGuimod.sip

%ExportedHeaderCode
#include <AtomicPy.h>
%End

class AtomicMolecule /NoDefaultCtors/
{
public:
  AtomicMolecule( const QString& name ) /Transfer/;
  void addAtom( const QString& atom, const double x,
                const double y, const double z );

private:
  AtomicMolecule(AtomicMolecule&);
};
```

Now we are ready for the implementation of the `dumpPython()` method:

```
bool ATOMICGUI_DataModel::dumpPython( const QString& theURL,
              CAM_Study* theStudy,
              bool isMultiFile,
              QStringList& theListOfFiles  ) {
  QString aScript = "from AtomicPy import *\n";
  QString aPrefix = "";
  if(isMultiFile) {
    aScript += "def RebuildData(theStudy):\n";
    aPrefix = "\t";
  }

  for ( int i = 0; i < myMolecules.count(); i++ ) {
    aScript += aPrefix + QString("mol_%1 =
```

```
      AtomicMolecule('").arg(i) + myMolecules[ i ].name()+"')\n";
    for ( int j = 0; j < myMolecules[ i ].count(); j++ ) {
      aScript += aPrefix + QString("mol_%1.addAtom('").arg(i) +
                 myMolecules[ i ].atomName( j );
      aScript += QString("', %1, %2, %3)\n")
                 .arg(myMolecules[i].atomX( j ))
                 .arg(myMolecules[i].atomY( j ))
                 .arg(myMolecules[i].atomZ( j ));
    }
  }

  if(isMultiFile) {
    aScript += aPrefix+"pass\n";
  }

  LightApp_Study* study = dynamic_cast<LightApp_Study*>( theStudy );
  if(!study)
    return false;

  std::string aTmpDir = study->GetTmpDir(
              theURL.toLatin1().constData(), isMultiFile );
  std::string aFile = aTmpDir + "atomic_dump.tmp";

  std::ofstream outfile(aFile.c_str());
  outfile.write (aScript.toLatin1().data(),aScript.size());
  outfile.close();

  theListOfFiles.append(aTmpDir.c_str());
  theListOfFiles.append("atomic_dump.tmp");

  return true;
}
```

Our function iterates through the list of molecules stored in the data model and generates Python script, line by line. After that it stores generated script into temporary file and puts path to temporary file and name of the file into output parameter `theListOfFiles`.

Note, that way the Python script is created is different for “single-file” and “multiple-file” modes. Mainly this concerns the tabulations and preface part of generated script.

Now our ATOMIC component can generate Python script, which can restore state of its data model:

```
from AtomicPy import *
def RebuildData(theStudy):
  mol_0 = AtomicMolecule('H20')
  mol_0.addAtom('H1', 0, 0, 0)
  mol_0.addAtom('H2', 1, 1, 1)
  mol_0.addAtom('O', 0.5, 0.5, 0.5)
  pass
```

Please, use this link to download the latest source files of the fully functional version of ATOMIC component with implemented dump python mechanism.

This is the end of the chapter dedicated to development of a light-weight component. If you are ready to continue with our tutorial and learn about other types of components - please, proceed to the next chapter - ATOMGEN: Python component.

4. ATOMGEN: PYTHON COMPONENT

In this chapter we learn how to develop a component in Python programming language (SALOME platform supports 2 languages for development of custom components: C++ and Python). Using Python for new component development is easier in many ways, because a lot of implementation details are hidden from a developer, Python modules (files in Python language) are brief and clear.

A very important issue of this chapter is a concept of CORBA Engine. Engine is a part of a component, built upon CORBA technology, which usually performs algorithmic data processing. Engine can be understood as a stand-alone piece of software within a component, whose services may be used by a component it belongs to as well as by other external components.

Throughout this chapter we shall develop a Python component named ***ATOMGEN***, with CORBA engine and graphic user interface. This component is able to perform pseudo-algorithmic processing of the data prepared by ATOMIC (light-weight C++ component developed in the first chapter of the tutorial). The data processing in ATOMGEN is in some sense a "spatial analysis" of molecules and atoms: for every molecule it will create 5 new molecules, atoms of new molecules will have different coordinates (translated by a constant distance along X, Y, and Z axes). ATOMGEN is also able to read XML file with data prepared by ATOMIC component and export its data to an XML file (just like ATOMIC).

Having completed this chapter, the ATOMGEN component should look like shown on the picture below:

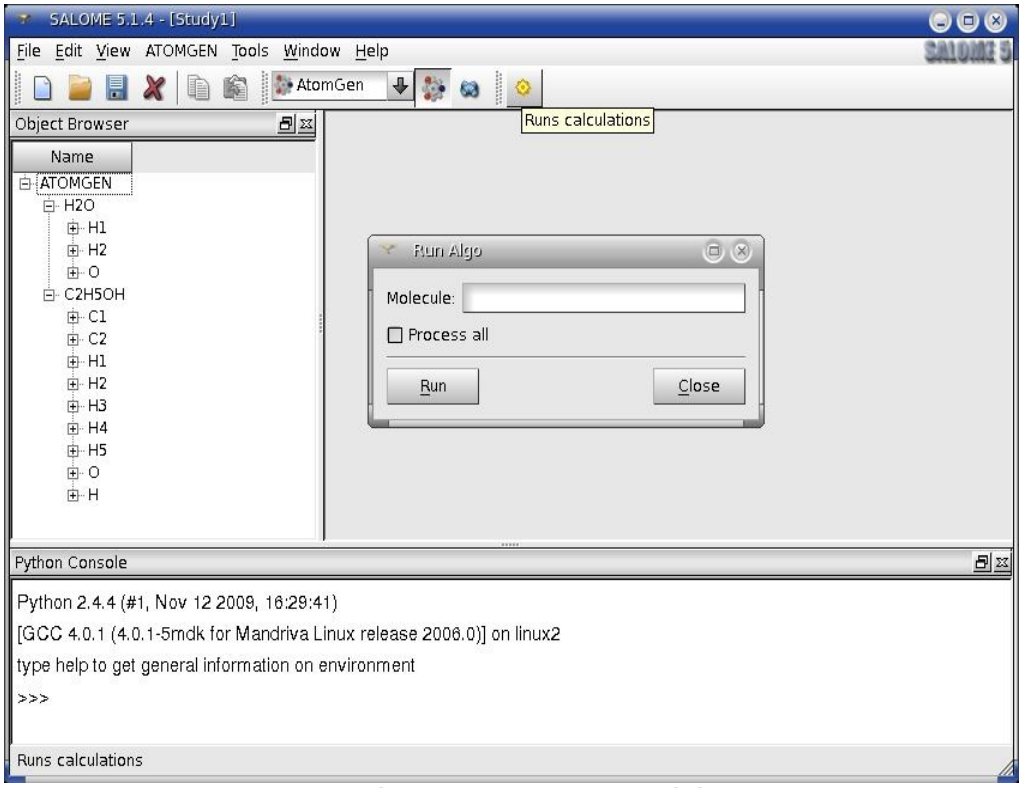

Figure 5. ATOMGEN module

Proceed to the next paragraph to start learning ATOMGEN module.

4.1 COMPONENT WITH CORBA ENGINE

In this section we introduce the notion of engine - a CORBA component that provides a number of services for other objects. CORBA engines are constructed by SALOME components that perform analytical processing used by this component and by other external objects (from within other components, from Python console, etc.). A possibility to use functionality of a component externally is the main advantage of using a CORBA engine.

In our sample ATOMGEN Python component we want to perform analytical processing of data prepared by previously developed ATOMIC component, and provide the next following component - C++ component - with the results of processing. It means that we will use 2 advantages of using an Engine in our application:

- Data processing code is separated from GUI code.
- Interaction with another component is more convenient and transparent (passing results of processing to the next component is done by means of intra-CORBA communication – communication between Engines of components).

Engine is basically a CORBA object which inherits `Engines::EngineComponent` interface declared in `SALOME_Component.idl` file of KERNEL module of SALOME platform. Let's take a look at `Engines::EngineComponent` interface:

readonly attribute `string instanceName`	The name of the instance of the component
readonly attribute `string interfaceName`	The name of the interface of the component
`void ping()`	Determines whether the server has already been loaded or not
`long getStudyId()`	Get study associated to component instance return -1: not initialised (Internal Error) 0: multistudy component instance >0: study id associated to this instance
`void destroy()`	Deactivates the engine TO BE USED BY CONTAINER ONLY (Container housekeeping) use remove_impl from Container instead!
`Container GetContainerRef()`	Returns the container that the engine refers to.
`void setProperties(in FieldsDict dico)`	Gives a sequence of (key=string, value=any) to the engine. Base class component stores the sequence in a map. The map is cleared before. This map is for use by derived classes.
`FieldsDict getProperties()`	Returns a previously stored map (key=string, value=any) as a sequence.
`TMPFile DumpPython(in Object theStudy, in boolean isPublished, in boolean isMultiFile, out boolean isValidScript)`	Returns a python script, which is being played back reproduces the data model of component. Is redefined by components that support such feature.
The following methods are used by Supervisor component for managing a component remotely (starting a certain method of an engine, killing, suspending, etc.)	
`boolean Kill_impl()`	Returns True if the engine has been killed.
`boolean Stop_impl()`	Returns True if the activity of the engine has been stopped (action can't be resumed).
`boolean Suspend_impl()`	Returns True if the activity of the engine has

	been suspended (action can be resumed).
`boolean Resume_impl()`	Returns True if the activity of the engine has been resumed.
`long CpuUsed_impl()`	Returns the Cpu used time (long type does not work in Python!)
`void Names (in string aGraphName, in string aNodeName)`	This method is used by the SUPERVISOR component. It sets the names of the graph and of the node.

Another interface - **Container** (declared in the same file `SALOME_Component.idl`) is used for instanciating of components (to be more precise - engines of components). Main methods of Container interface are:

`boolean load_component_Library(in string componentName)`	Loads a new component class (dynamic library). Returns true if load successfull or already done, false otherwise.
`Component create_component_instance(in string componentName, in long studyId)`	Creates a new servant instance of a component. Component library must be loaded prior to call of this method.
`Component find_component_instance(in string registeredName, in long studyId)`	Finds a servant instance of a component. - `registeredName` is the name of the component in Registry or Name Service, without instance suffix number
`Component load_impl(in string nameToRegister, in string componentName)`	Find a servant instance of a component, or create a new one. Loads the component library if needed. Only applicable to multi study components. - `nameToRegister` is the name of the component which will be registered in Registry (or Name Service) - `componentName` is the name of the constructed library of the component (not used any more, give empty string)
`void remove_impl(in Component component_i)`	Stops the component servant, and deletes all related objects

So, basically, an Engine is a CORBA object and a Container is its manager-object which creates it, publishes in the naming service, and destroys it.

In the next section we are going to write an IDL file for ATOMGEN engine and a Python file with implementation of the methods declared in the IDL file.

4.2 ENGINE: INTERFACE AND IMPLEMENTATION

Development of any CORBA object (and our Engine is a CORBA object) is accomplished in 2 steps:

- Writing the interface in IDL language (Interface Definition Language).
- Development of the implementation of methods declared in the IDL file using one of the following programming languages: C, C++, Java, Smalltalk, COBOL, Ada, Lisp, PL/1, Python. In SALOME applications we use Python and C++.

Deep study of IDL language is beyond the scope of this tutorial. If you are unfamiliar with IDL, please, read the corresponding literature. These links may help:

- Mastering the Interface Definition Language (IDL) from Teach Yourself CORBA In 14 Days tutorial
- Introduction to CORBA IDL from Orbix Programmer's Guide

OK, let's start coding. In IDL file of ATOMGEN component we would like to declare the following CORBA interfaces and methods:

`interface Atom {`
`  string getName();`
`  double getX();`
`  double getY();`
`  double getZ();`
`};`
	Interface of the most elementary piece of data of ATOMGEN component - atoms. An atom has methods for retrieval of name and coordinates. Setting and storing of these properties will be done only in implementation in Python.

`interface Molecule {`
`  string getName();`
`  long getNbAtoms();`
`  Atom getAtom(in long theIndex);`
`};`
	Interface of molecule - compound data type of ATOMGEN (similar to ATOMIC data).

`typedef sequence<Molecule>` `MoleculeList;`	Declaration of sequence of molecules type.

`interface ATOMGEN_Gen :`
`Engines::EngineComponent {`
`  void setCurrentStudy (in`
`SALOMEDS::Study);`
`  boolean importXmlFile (in string`
`theFileName);`
`  boolean exportXmlFile (in string`
`theFileName);`
`  boolean processData (in`
`MoleculeList theData);`
`  MoleculeList getData (in long`
`studyID);`
`};`
	`ATOMGEN_Gen` - interface of ATOMGEN component Engine - inherits `Engines::EngineComponent`
	`setCurrentStudy()` method is required for connecting to the current data source (study object)
	`importXMLFile()` method performs import of data prepared by ATOMIC or ATOMGEN components
	`exportXMLFile()` method performs export of current data to an XML file
	`processData()` method performs "spacial processing" of molecules and atoms: molecules are increased in number, new atoms assigned new coordinates
	`getData()` method returns the list of molecules of a certain study

Let's assume we have written the methods above in IDL file `ATOMGEN.idl`. IDL files are placed in the `idl` subdirectory, located under main source directory of the component. Please, download the first version of ATOMGEN component with `ATOMGEN.idl` file in `idl` subdirectory.

Now we have to create a Python module with implementation of the declared IDL interfaces. We shall create an `src` subdirectory (under main source directory) and add a package called ATOMGEN. Then we are going to create 3 files with code in Python:

`ATOMGEN.py`	Implementation of `ATOMGEN_Gen` interface declared in `ATOMGEN.idl`

ATOMGEN_Data.py	Implementation of Atom and Molecule interfaces declared in ATOMGEN.idl
ATOMGEN_XmlParser.xml	Implementation of XML import and export functionality (used from within ATOMGEN.py Python module)

Let's take a look at the code of ATOMGEN Python module (from next version of ATOMGEN component):

- General initialization

```
# necessary import clauses
import ATOMGEN_ORB
import ATOMGEN_ORB__POA
import SALOME_ComponentPy

# initializing ORB(CORBA)
from omniORB import CORBA
myORB = CORBA.ORB_init([''], CORBA.ORB_ID)

# initializing Portable Object Adapter (CORBA)
from omniORB import PortableServer
myPOA = myORB.resolve_initial_references("RootPOA");

# define a function to convert CORBA object to servant
def ObjectToServant(object):
  return myPOA.reference_to_servant(object)
```

- ATOMGEN class - implements ATOMGEN_Gen interface

```
# ATOMGEN class inherits a stub generated by omniIDL
# for ATOMGEN_Gen interface (ATOMGEN_ORB__POA.ATOMGEN_Gen class).
class ATOMGEN( ATOMGEN_ORB__POA.ATOMGEN_Gen,
               SALOME_ComponentPy.SALOME_ComponentPy_i):

# Constructor
# _naming_service is inherited from the SALOME_ComponentPy class.
  def __init__ (self, orb, poa, contID, containerName,
                instanceName, interfaceName):
    SALOME_ComponentPy.SALOME_ComponentPy_i.__init__(self, orb, poa,
           contID, containerName, instanceName, interfaceName, 0)
    self._naming_service =
           SALOME_ComponentPy.SALOME_NamingServicePy_i(self._orb)
    # self.study keeps reference to the current study (None for now)
    self.study = None
    # self.studyData stores molecules data of several studies.
    # it is a map with an integer key (study ID) and values of type
    # ListOfMolecule.
    # data of multiple studies can be stored using this map
    self.studyData = {}
    pass

  # Returns data (MoleculeList) of the given study (by study ID)
  def getData( self, studyID )
    if self.studyData.has_key(study._get_StudyId()):
      returnself.studyData[study._get_StudyId()]
    return None

  # Sets current study and clears the internal data
  # that is bound with new study ID
```

```
  def setCurrentStudy( self, study ):
    self.study = study
    if self.study and not self.getData(self.study._get_StudyId()):
      self.studyData[self.study._get_StudyId()] = []
    pass

  # Performs import of data from an XML file using method
  # readXmlFile declared in ATOMGEN_XmlParser Python module
  # After import the data is saved in the internal data structure
  def importXmlFile( self, fileName ):
    if self.study:
      from ATOMGEN_XmlParser import readXmlFile
      new_data = readXmlFile( fileName )
      for mol in new_data:
        for i in range(mol.getNbAtoms()):
          mol.atoms[ i ]._this()
        mol = mol._this()
      data = self.getData(self.study._get_StudyId())
      data += new_data
      return True
    return False

  # Exports data to an XML file using method writeXmlFile
  # declared in ATOMGEN_XmlParser Python module
  def exportXmlFile( self, fileName ):
    if self.study:
      from ATOMGEN_XmlParser import writeXmlFile
      studyID = self.study._get_StudyId()
      writeXmlFile( fileName, self.studyData[ studyID ] )
      return True
    return False

  # Artificial "spacial processing" of data.
  # For every existing molecule 5 (nb_steps)
  # new molecules are created.
  # Coordinates of atoms are shifted by 10, 5, and 3 respectively
  # multiplied by new molecule number.
  def processData( self, data ):
    if not self.study: return False
    nb_steps = 5
    new_data = []
    dx = 10.0
    dy = 5.0
    dz = 3.0
    for i in range( nb_steps ):
      for mol in data:
        new_mol = self._translateMolecule(
            mol, dx*(i+1), dy*(i+1), dz*(i+1))
        new_data.append( new_mol )
        for j in range(new_mol.getNbAtoms()):
          new_mol.atoms[ j ]._this()
        new_mol = new_mol._this()
    data = self.getData(self.study._get_StudyId())
    data += new_data
    return True

  # Creates new molecule, coordinates of atoms are shifted
  # by given values.
  def _translateMolecule(self, mol, dx, dy, dz):
    mol = ObjectToServant( mol )
    from ATOMGEN_Data import Molecule, Atom
    new_mol = Molecule(mol.name + " translated")
    print mol.name
```

```
    for atom in mol.atoms:
      new_mol.addAtom( Atom(atom.name, atom.x + dx,
                            atom.y + dy, atom.z + dz) )
    return new_mol
```

The other Python modules – `ATOMGEN_Data.py` and `ATOMGEN_XmlParser.py` do not require additional explanations - `Atom` and `Molecule` interfaces have the simplest implementation possible, and XML parser functions are simple utility functions in Python.

Please build current ATOMGEN component. It does not have any GUI connected to the engine yet, so use of the component is very limited.

In the next section we will learn how to implement persistence of data using SALOMEDS package of KERNEL module. This is a very powerful tool for persistence implementation and definitely worth learning.

4.3 ADVANCED DATA STORAGE

SALOME platform has a number of services that can be used by custom components. One of these services is SALOMEDS package of KERNEL module (DS stands for Data Structure). SALOMEDS provides a framework for implementation of persistent data structure. It can be used using CORBA interface of SALOMEDS package (see `SALOMEDS.idl` and `SALOMEDS_Attributes.idl` files) or without CORBA through a proxy package SALOMEDSClient. Classes of SALOMEDSClient package have the same interface as declared in `SALOMEDS.idl` (and implemented in SALOMEDS package), but the functionality of SALOMEDSClient classes can be used without CORBA, but only from C++ using pointers.

Since in this chapter we deal with a Python component, we have only one option to use SALOMEDS: through the CORBA interface. The table below explains the purpose of the main interfaces declared in `SALOMEDS.idl` file (and the corresponding classes of SALOMEDSClient package which have almost the same name: `SALOMEDS::Study` becomes `SALOMEDSClient_Study`, etc.).

`Study`	`Study` is a warehouse of data. It can be understood as a document, the data storage of the upper level. Study contains data of multiple components, it's a single document for all components. Most of operations on a Study object are handled by `StudyManager` and `StudyBuilder` interfaces.
`SObject`	`SObject` is the main constituent of SALOMEDS-based data structure. If you are familiar with CAF (Cascade Application Framework) - the analogy of `SObject` would be `TDF_Label` class. It can be understood as a branch of a tree of data, or as a record in a database table. Usually it does not store the data itself, it uses child Attributes - successors of `SALOMEDS::GenericAttribute` - for storing specific data, properties of the object.
`GenericAttribute`	`GenericAttribute` is base class for Attributes - an actual piece of data. Attributes are connected to `SObjects`, they can be observed as properties of `SObjects`.
`StudyManager`	Interface to manipulate Studies. You will find in this interface the methods to create, open, close, and save a `Study`. Since a SALOME session is multi-document, you will also find methods that allow navigating through a collection of studies currently present in a session.
`StudyBuilder`	The purpose of `StudyBuilder` interface is to add and/or remove objects and attributes. An instance of `StudyBuilder` is linked to a

	`Study`. A command management is also provided for undo/redo functionality.
`SComponent`	`SComponent` is basically an `SObject` (it inherits `SObject`) of a certain type. It represents a component itself in a multi-component document (a `Study` object). It is a parent for component's data, its `SObjects` and Attributes. As a comment in `SALOMEDS.idl` file says, "the SComponent interface establishes in the study a permanent association to the components integrated into SALOME platform".
`Driver`	`Driver` interface represents a common tool that allows components of SALOME application to perform the following data-related tasks: publish the objects created by a certain component in the `Study` (declare them persistent and available for other components) save/load the data created by a component transform the transient (run-time) references to `SObjects` into persistent references when saving (or loading) a study and vice versa. copy/paste common functionality. Copy/paste can be called by any component in order to copy/paste its object created in the study These functionalities are called by `StudyManager` for performing tasks on a `Study` object. Any component must implement `Driver` interface itself (as we do in ATOMGEN component) or provide an object which implements it for the component's data.

OK, let's return to ATOMGEN component. We want to use the functionality of SALOMEDS for implementation of persistence for our data. *(Yes, we already know how to export the data into an XML file, and we learned in the previous chapter how to enclose this XML file into a persistent Study file. Using SALOMEDS for this task will be done only by way of example, to display yet another way of persistence implementation)*.

We shall create number of new methods in `ATOMGEN.py` module:

```
def Save( self, component, URL, isMultiFile )
def Load( self, component, stream, URL, isMultiFile )
def IORToLocalPersistentID(self, sobject, IOR, isMultiFile, isASCII)
def LocalPersistentIDToIOR(self, sobject, persistentID,
                           isMultiFile, isASCII)
def Close( self, component )
def CanPublishInStudy( self, IOR )
def PublishInStudy( self, study, sobject, object, name )
```

All these methods are overridden from Driver interface. They will be called by StudyBuilder and StudyManager objects when a study is saved, loaded, closed, etc.

Let's take a closer look at `PublishInStudy()` method. It is intended to register (or "save" in other words) an object of "local" type, component dependent, in SALOMEDS-based data structure. It usually means creation of one or several SObjects and child Attributes. It can be understood as conversion of local data structure to SALOMEDS-based data structure.

```
def PublishInStudy( self, study, sobject, object, name ):
    if study and object and object._narrow(ATOMGEN_ORB.Molecule):
        builder = study.NewBuilder()
        builder.NewCommand()
        # get or create component object
        father = study.FindComponent(self._ComponentDataType)
        if father is None:
```

```
            builder
            father = builder.NewComponent(self._ComponentDataType)
            attr = builder.FindOrCreateAttribute(father,
                                                 "AttributeName")
            attr.SetValue(self._ComponentDataType)
            builder.DefineComponentInstance(father, self._this())
            pass
        # publish molecule
        sobject = builder.NewObject(father)
        attr = builder.FindOrCreateAttribute(sobject,
               "AttributeName")
        if not name:
            name = object.getName()
        attr.SetValue(name)
        attr = builder.FindOrCreateAttribute(sobject, "AttributeIOR")
        attr.SetValue(ObjectToString(object))
        # publish atoms
        for i in range(object.getNbAtoms()):
            atom = object.getAtom( i )
            sobject1 = builder.NewObject(sobject)
            attr = builder.FindOrCreateAttribute(sobject1,
                  "AttributeName")
            attr.SetValue(atom.getName())
            attr = builder.FindOrCreateAttribute(sobject1,
                  "AttributeIOR")
            attr.SetValue(ObjectToString(atom))
            sobject2 = builder.NewObject(sobject1)
            attr = builder.FindOrCreateAttribute(sobject2,
                  "AttributeName")
            attr.SetValue("x")
            attr = builder.FindOrCreateAttribute(sobject2,
                  "AttributeReal")
            attr.SetValue(atom.getX())
            sobject2 = builder.NewObject(sobject1)
            attr = builder.FindOrCreateAttribute(sobject2,
                  "AttributeName")
            attr.SetValue("y")
            attr = builder.FindOrCreateAttribute(sobject2,
                  "AttributeReal")
            attr.SetValue(atom.getY())
            sobject2 = builder.NewObject(sobject1)
            attr = builder.FindOrCreateAttribute(sobject2,
                  "AttributeName")
            attr.SetValue("z")
            attr = builder.FindOrCreateAttribute(sobject2,
                  "AttributeReal")
            attr.SetValue(atom.getZ())
        builder.CommitCommand()
        return sobject
    return None
```

As we look in the code above, we can see that we expect to receive an object of `ATOMGEN_ORB.Molecule` type at the input. Then we create a new `StudyBuilder` object and open a transaction on it. After that we find or create (only once for the first time) a `SComponent` object - father object for SObjects of ATOMGEN component. After that we create an SObject that corresponds to a given molecule object. We create 2 attributes for the molecule `SObject`: `AttributeName` (to store the name of molecule) and `AttributeIOR`. `AttributeIOR` stores a unique key of any CORBA object. In our case, the `SObject` exists on SALOMEDS CORBA server, and it has got its own IOR. In order to locate the servant on the side of our engine, we need to store its IOR in the `SObject` - that is done with the help of `AttributeIOR` child object.

Finally, we iterate the atoms of the molecule. For every atom we create an `SObject` and a number of attributes to store name, IOR, and coordinates of the atom.

Method `Load()` of ATOMGEN class performs the opposite task: it iterates the SALOMEDS data structure and creates `ATOMGEN_ORB.Molecule` and `ATOMGEN_ORB.Atom` objects for SObjects of the corresponding level (children of ATOMGEN `SComponent` correspond to molecules, grandchildren - to atoms). Please, pay attention at using special iterator classes (interfaces declared in `SALOMEDS.idl` file) for traversing the SALOMEDS data structure.

```
def Load( self, component, stream, URL, isMultiFile ):
    global __entry2IOR__
    __entry2IOR__.clear()
    import StringIO, pickle
    study = component.GetStudy()
    iter = study.NewChildIterator(component)
    data = []
    while iter.More():
        sobject = iter.Value()
        iter.Next()
        found, attr = sobject.FindAttribute("AttributeName")
        if not found: continue
        from ATOMGEN_Data import Molecule, Atom
        mol = Molecule(attr.Value())
        __entry2IOR__[sobject.GetID()] = ObjectToString(mol._this())
        iter1 = study.NewChildIterator(sobject)
        while iter1.More():
            sobject1 = iter1.Value()
            iter1.Next()
            found, attr = sobject1.FindAttribute("AttributeName")
            if not found: continue
            name = attr.Value()
            x = y = z = None
            iter2 = study.NewChildIterator(sobject1)
            while iter2.More():
                sobject2 = iter2.Value()
                iter2.Next()
                found, attr1 =
                  sobject2.FindAttribute("AttributeName")
                if not found: continue
                found, attr2 =
                  sobject2.FindAttribute("AttributeReal")
                if not found: continue
                if attr1.Value() == "x": x = attr2.Value()
                if attr1.Value() == "y": y = attr2.Value()
                if attr1.Value() == "z": z = attr2.Value()
            if None not in [x, y, z]:
                atom = Atom(name, x, y, z)
                mol.addAtom(atom)
                __entry2IOR__[sobject1.GetID()] =
                     ObjectToString(atom._this())
            pass
        data.append(mol)
    self.studyData[ study._get_StudyId() ] = data
    return 1
```

At this point we finish to study SALOMEDS and its use in ATOMGEN component. Please, download the source files of the current version of ATOMGEN with advanced data structure.

In the next chapter we create a graphic user interface for ATOMGEN component to finalize its development. Please, continue with GUI for Python component, if you are ready!

4.4 GUI FOR PYTHON COMPONENT

In distinction from C++ components, whose GUI modules are unique subclasses of a common parent class (`CAM_Module`), all Python components use the same GUI module class. This class is a C++ class `SALOME_PYQT_Module`. Surprised? How can a C++ class be a GUI module for a Python component? And even for all Python components, with different menu items, view windows, selection management policies, etc.? In this section we are going to explain how it works.

In the previous chapter about light-weight component, in the section "Instantiating a GUI module", we have pointed that a GUI library file name can be also specified in the resource file (`SalomeApp.xml` or `LightApp.xml` file). For C++ components the usual practice is not to indicate the name of GUI library file explicitly. The default algorithm of library file name construction is "lib" + component_name + ".so", and usually GUI libraries of C++ components have exactly the same file name, so there is no need to indicate them explicitly. But for a Python component it is not the case, and we must add the following lines to the resource file SalomeApp.xml (Python components are loaded only in full configuration of SALOME, it is a current limitation, so we examine only `SalomeApp.xml` file and omit `LightApp.xml`):

```
<section name="ATOMGEN" >
  <parameter name="name"    value="AtomGen" />
  <parameter name="icon"    value="ATOMGEN.png" />
  <parameter name="library" value="SalomePyQtGUI" />
</section>
```

After we have indicated a parameter `"library"` in resource section of ATOMGEN component, the GUI library file to be loaded on the first activation of ATOMGEN component will be `libSalomePyQtGUI.so`. `libSalomePyQtGUI.so` is a library built in `SALOME_PYQT/SALOME_PYQT_GUI` package of GUI module of SALOME platform. It contains `SALOME_PYQT_Module` class - a successor of `SalomeApp_Module` and a utility class `SALOME_PYQT_PyInterp`. `SALOME_PYQT_Module` implements all main virtual methods of GUI modules that allow for customization of GUI of a certain component:

```
void initialize();
bool activateModule();
bool deactivateModule();
void windows() const;
void viewManagers() const;
void contextMenuPopup(), etc.
```

Implementation of these methods is a little bit tricky. `SALOME_PYQT_Module` gets a name of a component that uses it, and constructs a name of a Python script using the following rule: "component_name"+"GUI.py" ("ATOMGENGUI.py" in our case). `SALOME_PYQT_Module` has a Python interpreter wrapped into `SALOME_PYQT_PyInterp` class, and implementation of the methods (`initialize()`, `activateModule()`, etc.) simply calls a method with the same name in a Python module constructed above. So `SALOME_PYQT_Module` constructed for ATOMGEN component in its `initialize()` method will call a Python method `initialize()` from `ATOMGENGUI.py`.

The implementation of this schema is little bit more complicated. Let's take a look at `deactivatedModule()` method, for example:

```
bool SALOME_PYQT_Module::deactivateModule( SUIT_Study* theStudy )
{
  MESSAGE( "SALOME_PYQT_Module::deactivateModule" );

  if ( menuMgr() )
    disconnect( menuMgr(), SIGNAL( menuHighlighted( int, int ) ),
                this,      SLOT( onMenuHighlighted( int, int ) ) );

  // remove menus & toolbars created from XML file if required
  if ( myXmlHandler )
```

```
    myXmlHandler->clearActions();

  // deactivate menus, toolbars, etc
  setMenuShown( false );
  setToolShown( false );

  // DeactivateReq: request class for internal deactivate() operation
  class DeactivateReq : public PyInterp_LockRequest
  {
  public:
    DeactivateReq( PyInterp_base*      _py_interp,
           SUIT_Study*         _study,
           SALOME_PYQT_Module* _obj )
      : PyInterp_LockRequest( _py_interp, 0, true ), myStudy ( _study
),
        myObj   ( _obj   ) {}
  protected:
    virtual void execute()
    {
      myObj->deactivate( myStudy );
    }
  private:
    SUIT_Study*         myStudy;
    SALOME_PYQT_Module* myObj;
  };

  // Posting the request
  PyInterp_Dispatcher::Get()->Exec( new DeactivateReq( myInterp,
theStudy, this ) );

  return SalomeApp_Module::deactivateModule( theStudy );
}
```

First of all, the method does all usual deactivation things: disconnect the signals and hide menus and tool buttons. Then it creates an object of `PyInterp_LockRequest` type and passes it to `PyInterp_Dispatcher`. This mechanism is used in order to synchronize the calls to Python interpreter. User actions are asynchronous by their nature (user can start an action, then - yet another action before the first one is finished, and so on) and if we pass them to Python interpreter in the same asynchronous order (or disorder), then actions easily lock each other and the Python interpreter hangs. This problem is solved with the help of special class `PyInterp_Dispatcher` which creates a queue of requests of the Python interpreter and calls the actions one after another.

Let's see what happens in `deactivatedModule()` up to the end. The code that will be synchronously executed is in the protected virtual method `execute()` of `DeactivateReq` (`PyInterp_LockRequest` subclass object):

```
virtual void execute()
{
  myObj->deactivate( myStudy );
}
```

Here, `myObj` is an object of `SALOME_PYQT_Module` type, so we have to see `SALOME_PYQT_Module::deactivate()` method now (as it is called in `execute()`):

```
void SALOME_PYQT_Module::deactivate( SUIT_Study* theStudy )
{
  // check if the sub interpreter is initialized and Python module is
imported
  if ( !myInterp || !myModule ) {
    // Error! Python sub interpreter should be initialized and module
should be imported first!
```

```
    return;
  }
  // then call Python module's deactivate() method
  if(PyObject_HasAttrString(myModule , "deactivate")){
    PyObjWrapper res(PyObject_CallMethod(myModule, "deactivate",
"""));
    if( !res ) {
      PyErr_Print();
    }
  }
}
```

After checking the Python interpreter and GUI module for existence, method `deactivate` is located and invoked in a Python script:

```
PyObject_CallMethod( myModule, "deactivate", "" )
```

OK, we finally come to the code executed in ATOMGEN component. Let's see the deactivate method of `ATOMGENGUI.py`:

```
def deactivate():
    print "ATOMGENGUI::deactivate"
    # connect selection
    global myStudy
    studyId = myStudy._get_StudyId()
    selection = __study_data_map__[ studyId ][ "selection" ]
    selection.ClearIObjects()
    QObject.disconnect( selection,
         SIGNAL( "currentSelectionChanged()" ), selectionChanged )
    global myRunDlg
    if myRunDlg:
        myRunDlg.close()
        myRunDlg = None
    myStudy = None
    pass
```

This method performs the local de-activation, in particular, it clears selection and closes the dialog box if it was open.

Once again, let's retrace the call stack:

1. `SALOME_PYQT_Module::deactivateModule()` is called asynchronously (in general) in response to an external event (component deactivation)
2. `SALOME_PYQT_Module::deactivate()` is called synchronously using an internal event queue
3. deactivate from ATOMGENGUI is called

This procedure of calls from `SALOME_PYQT_Module` to Python script is followed by all methods of GUI module for a Python component. The following methods of ATOMGENGUI.py are called from `SALOME_PYQT_Module` in response to corresponding events:

- initialize
- windows
- views
- activate
- deactivate
- activeStudyChanged
- createPopupMenu
- OnGUIEvent

Please, refer to ATOMGENGUI.py file to know how these methods are implemented. It is not complicated, so we will not examine them in details.

There are 2 more topics related to ATOMGEN GUI that we are going discuss here: using pyuic compiler for Qt-based dialog boxes development and creation of menu items and tool buttons.

As you have noticed, in ATOMGENGUI package of ATOMGEN component we have a file with "ui" extension: rundlg.ui. This file was prepared with Qt Designer tool, it is in special format (similar to XML) that describes a layout of a dialog box. In ATOMGENGUI.py file we have the following code:

```
import ui_rundlg
class RunDlg(QDialog, ui_rundlg.Ui_RunDlg):
......
```

The Python module ui_rundlg, mentioned in this code, is a module automatically generated from rundlg.ui file. Class _rundlg.Ui_RunDlg is a class of dialog box described in rundlg.ui file. In the source directory we have only this ui file, but after we build ATOMGEN component, the build directory will contain ui_rundlg.py file (ATOMGEN_BUILD/bin/salome subdirectory) generated by pyuic compiler. So we can derive a new class from rundlg.RunDlg:

```
class RunDlg(QDialog, ui_rundlg.Ui_RunDlg)
```

Deriving from classes generated by the pyuic compiler is very common in development of GUI for Python components. QDialog is the parent class, that allows RunDlg to be Qt dialog window widget that contains all components declared in ui file. In constructor on RunDlg it is called setupUi method of Ui_RunDlg to initialize and insert all content widgets into the dialog box.

The last topic that we are going to learn in this section is creation of menu items and tool buttons in GUI modules written in Python. GUI module of a C++ component can create menu items and tool buttons using only createMenu(), and createTool() methods. Python components also can use these methods of SalomePyQt Python module. As we can see in activate method of ATOMGENGUI module a menu items are created using the following calls:

```
a = sgPyQt.createAction( __CMD_IMPORT_XML__,
                         tr( "MEN_IMPORT_XML" ),
                         tr( "TOP_IMPORT_XML" ),
                         tr( "STB_IMPORT_XML" ) )
fileMnu = sgPyQt.createMenu( QApplication.translate( "ATOMGENGUI",
                         "MEN_FILE" ), -1, -1 )
sgPyQt.createMenu( __CMD_IMPORT_XML__, fileMnu, 10 )
```

But we can also see that menu *ATOMGEN* with *Run* item is not present in the code of ATOMGENGUI. Identifier of the "Run" command is declared there (__CMD_RUN_ALGO__ = 4002), moreover, the command is properly handled in ATOMGENGUI (onRunAlgo() method). This menu item and the tool button are declared in a special XML file which is located in the resource directory of ATOMGEN component: ATOMGEN_en.xml

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE application PUBLIC "" "desktop.dtd">
<application title="ATOMGEN component" date="15/11/2005"
             author="SALOME team" appId="SALOME" >
<desktop>

<!-- ### MENUBAR ###  -->
<menubar>
 <menu-item label-id="ATOMGEN" item-id="90" pos-id="3">
  <popup-item item-id="4002" label-id="Run" icon-id=""
   tooltip-id="Runs calculations" accel-id="" toggle-id=""
```

```
    execute-action=""/>
  </menu-item>
</menubar>

<!-- ### TOOLBARS ###  -->
<toolbar label-id="ATOMGEN">
  <toolbutton-item item-id="4002" label-id="Run"
      icon-id="atomgen_run.png" tooltip-id="Runs calculations"
      accel-id="" toggle-id="" execute-action=""/>
</toolbar>

</desktop>
</application>
```

This XML file describes menu items and tool buttons for ATOMGEN component. In order to use this way of menu items creation, the XML file must be named in the following way: `component_name + "_" + language_id + ".xml"`. `language_id` is 2 letters language identifier used in the current session of SALOME (`"en"`, `"fr"`, `"ru"`, etc. - same postfix is used for naming the `po`-files in SALOME). This way of menu items and tool buttons creation (via XML file) was adopted in previous versions of SALOME platform (series 1.x, 2.x), now it is left only for Python components for compatibility reasons.

With this section we would like to finish the development of ATOMGEN Python component. Please, use this link to download the latest source files of the fully functional version of ATOMGEN. Study them carefully, and after that return to our tutorial to study the next chapter which will explain how to develop a component in C++ with engine and advanced visualization for our molecular data!

4.5 DUMP PYTHON MECHANISM

As it has been discussed in chapter 3.7.1 there are basically two different approaches for implementing of Dump python functionality in SALOME components – to make “historical” or “snapshot” dump. For the ATOMGEN component “historical dump” approach seems to suit better than “snapshot” one, because engine of this component implements methods, which can create and publish in the study more then one object at a time. As described in the paragraph 3.7.1, in case of the “historical dump” approach each command that creates and publishes data in the study should store additional information related to the Dump python functionality in the component. To store this additional information we will use `AttributeTableOfString` attribute class that will be created on the root `SObject` of the component’s data tree. This approach allows us to avoid keeping the data related to the Dump python functionality somewhere in additional data structures at the engine side. An additional advantage of SALOME attribute usage is that it is persistent (as all other SALOMEDS attributes) – it is saved/restored to the study file automatically by SALOME data server.

Let’s add required changes in the implementation of the component’s methods:

1. Changes in the `importXmlFile()` method:

```
    def importXmlFile( self, fileName ):
        """
        Imports atomic data from external XML file
        and publishes the data in the active study
        """
        if self.study:
            # import file
            from ATOMGEN_XmlParser import readXmlFile
            new_data = readXmlFile( fileName )
            entries = self.appendData( new_data )
            if len(entries)  > 0 :
                cmd = "[" + ", ".join(entries) +
                      "] = " + __pyEngineName__
                cmd += ".importXmlFile('" + fileName + "')"
                attr = self._getTableAttribute()
```

```
                if attr is not None:
                    attr.PutValue(cmd,attr.GetNbRows()+1,1)
            res = []
            for m in new_data:
                res.append(m._this())
            return res
        return []
```

2. Changes in the `processData()` method:

```
    def processData( self, data ):
        """
        Perform some specific action on the atomic data
        """
        if not self.study: return []
        nb_steps = 5
        new_data = []
        dx = 10.0
        dy = 5.0
        dz = 3.0
        for i in range( nb_steps ):
            for mol in data:
                new_mol = self._translateMolecule( mol, i,
                       dx * (i+1), dy * (i+1), dz * (i+1) )
                new_data.append( new_mol )
        entries = self.appendData( new_data )
        if len(entries) > 0 :
            lst = []
            for m in data:
                ior = ObjectToString(m)
                so = self.study.FindObjectIOR(ior)
                lst.append(so.GetID())

            cmd =  "[" + ", ".join(entries) + "] = "+__pyEngineName__
            cmd += ".processData([" + ", ".join(lst) + "])"
            attr = self._getTableAttribute()
            if attr is not None:
                attr.PutValue(cmd, attr.GetNbRows()+1,1)
        res = []
        for m in new_data:
            res.append(m._this())
        return res
```

3. Changes in the `exportXmlFile()` method:

```
    def exportXmlFile( self, fileName ):
        """
        Exports atomic data from the active study to
        the external XML file
        """
        if self.study:
            from ATOMGEN_XmlParser import writeXmlFile
            studyID = self.study._get_StudyId()
            writeXmlFile( fileName, self.studyData[ studyID ] )
            cmd = __pyEngineName__+ ".exportXmlFile('" +
                  fileName + "')"
            attr = self._getTableAttribute()
            if attr is not None:
                attr.PutValue(cmd,attr.GetNbRows()+1,1)
            return True
        return False
```

Now methods store python commands necessary to generation python script in the study.

Also `processData()` and `importXmlFile()` methods return the list of references to the created objects.

Important notice: for identification of the objects in the stored python command objects' entries (unique identifiers) are used. These identifiers will be replaced by the unique python names in the `DumpPython` method (see below).

Finally, let's impement `DumpPython` method. This method iterates through the stored python commands, for each command generates unique valid object name and replaces object's entry by the generated name in the resulting Python command:

```
    def DumpPython(self, theStudy, isPublished, isMultiFile):
        script = []
        prefix = ""
        if isMultiFile :
            script.append("import salome")
            script.append("\n")
            prefix = "\t"
        script.append("import ATOMGEN\n")
        script.append(__pyEngineName__ + " =
           salome.lcc.FindOrLoadComponent(\"FactoryServerPy\",
           \"ATOMGEN\")")

        if isMultiFile :
            script.append("def RebuildData(theStudy):\n")
            script.append(prefix+__pyEngineName__ +
                ".setCurrentStudy(theStudy)\n")
        else:
            script.append(__pyEngineName__ +
                ".setCurrentStudy(theStudy)\n")

        attr = self._getTableAttribute()
        if attr is not None:
            for idx in range(attr.GetNbRows()):
                s = prefix + attr.GetValue(idx+1,1)
                script.append(s)

        if isMultiFile :
          script.append(prefix+"pass")
        else:
          script.append("\n")
        script.append("\0")

        all = "\n".join(script)
        self._getPyNames()
        studyID = self.study._get_StudyId()

        for k in self.entry2PyName[studyID].keys() :
            all = all.replace(k,self.entry2PyName[studyID][k])

        return (all,1)
```

Please, use this link to download the latest source files of the fully functional version of ATOMGEN componet with implemented dump python mechanism.

5. ATOMSOLV: C++ COMPONENT WITH ENGINE

This chapter is dedicated to C++ component with a CORBA engine. This type of component is the most frequently used in `industrial` SALOME-based applications as it provides the most of advantages of SALOME platform. Both KERNEL and GUI modules must be compiled in full configuration to include all services of SALOME platform.

Studying this chapter step by step, we shall develop the next new component for processing of molecules and atoms - ***ATOMSOLV*** component. Currently the cycle of our sample data processing is the following:

- the data (molecules and atoms) are created in ATOMIC component (light-weight) and saved in XML format;
- ATOMGEN component reads the data from the XML file and performs their "spacial analysis": number of molecules and atoms is increased.

ATOMSOLV component (to be developed in this chapter) retrieves the data from ATOMGEN component and performs its further analysis. It assigns properties to molecules (we think of these properties as "temperature" of molecules). And ATOMSOLV finally displays the results: atoms are displayed as spheres in 3D space; the color of the spheres reflects the value of temperature of a molecule the atoms belong to.

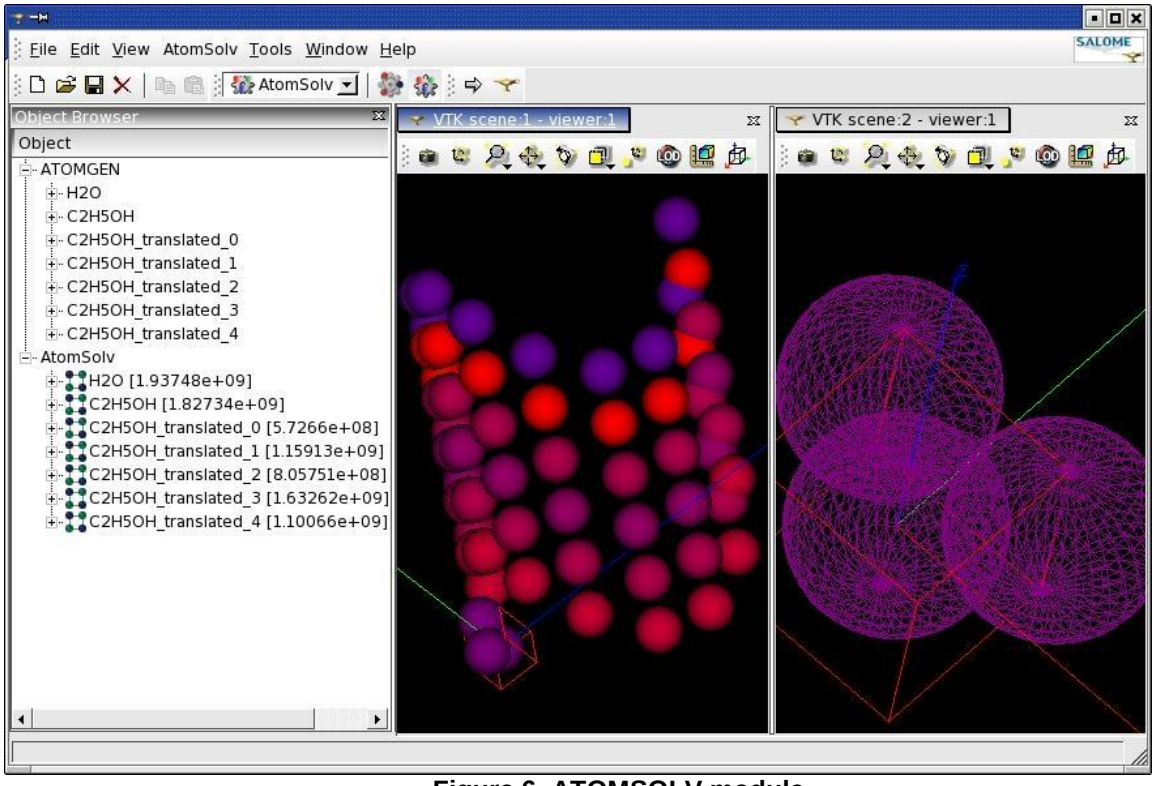

Figure 6. ATOMSOLV module

Having completed this chapter we shall have the the full demonstration of data processing cycle: data preparation (modeling), data analysis, and visualization of results.

The new concepts of SALOME platform that we are going to learn in this chapter mainly include graphical capabilities of SALOME: notions of view window, view manager, selection in 3D viewers, user-defined preferences, etc.

5.1 ENGINE: INTERFACE AND IMPLEMENTATION

Interface Engine of ATOMSOLV component is rather simple. Its objective is to store data retrieved from ATOMGEN component and perform additional processing of the data. In order to store the results of additional processing we must extent `ATOMGEN_ORB::Molecule` interface. To keep it simple, we will use *struct* to store the original molecule and its new property: floating point value of temperature:

```
module ATOMSOLV_ORB
{
  struct TMolecule
  {
    ATOMGEN_ORB::Molecule molecule;
    double temperature;
  };
};
```

The interface of ATOMSOLV engine:

```
module ATOMSOLV_ORB
{
  typedef sequence<TMolecule> TMoleculeList;

  interface ATOMSOLV_Gen : Engines::EngineComponent
  {
    boolean setData( in long studyID, in TMoleculeList theData );
    boolean getData( in long studyID, out TMoleculeList outData );
    boolean processData( in long studyID );
  };
};
```

Let's take a look at the implementation of the ATOMSOLV engine interface:

```
class ATOMSOLV: public POA_ATOMSOLV_ORB::ATOMSOLV_Gen, public
Engines_Component_i
{
public:
  ATOMSOLV(CORBA::ORB_ptr, PortableServer::POA_ptr,
PortableServer::ObjectId *, const char *, const char *);
  virtual ~ATOMSOLV();

  bool setData( long studyID,
                const ATOMSOLV_ORB::TMoleculeList& theData );
  bool getData( long studyID,
                ATOMSOLV_ORB::TMoleculeList_out outData );
  bool processData( long studyID );

private:
  std::map<long, ATOMSOLV_ORB::TMoleculeList*> myData;
};
```

The implementation class `ATOMSOLV` inherits `POA_ATOMSOLV_ORB::ATOMSOLV_Gen` class which is an automatically generated stub for `ATOMSOLV_Gen` CORBA interface. And it inherits `Engines_Component_i` class as the base component (engine) implementation class.

The private member field `myData` (std::map) is used to store lists of molecules for multiple studies. The key in the map is a studyID, and the value is a list of TMolecules (molecules with temperature).

The processing is done very simple - temperature assigned to molecules is a randomly generated floating point value:

```
bool ATOMSOLV::processData( long studyID )
{
```

```
  if ( myData.find( studyID ) != myData.end() ) {
    ATOMSOLV_ORB::TMoleculeList* data = myData[ studyID ];
    for ( int i = 0, n = data->length(); i < n; i++ )
      (*data)[i].temperature = rand();
  }
}
```

Please, use this link to download the version of ATOMSOLV with implementation of engine. It is not possible to really work with this version as it has no GUI, so, please, use it only as a code reference.

5.2 INSTANTIATING A GUI MODULE

GUI module for a component with engine must be derived from `SalomeApp_Module` class, which presumes use of an engine. Pure virtual method `SalomeApp_Module::engineIOR()` must return IOR of CORBA engine of a component. The engine is usually loaded by GUI module on the first invocation (in virtual method `initialize()`) and registered in LifeCycleCORBA. Let's take a look at methods of ATOMSOLVGUI that work with engine:

```
virtual QString engineIOR() const;
static void InitATOMSOLVGen( SalomeApp_Application* );
static ATOMSOLV_ORB::ATOMSOLV_Gen_var GetATOMSOLVGen();
```

```
void ATOMSOLVGUI::InitATOMSOLVGen( SalomeApp_Application* app )
{
  if ( !app )
    myEngine = ATOMSOLV_ORB::ATOMSOLV_Gen::_nil();
  else {
    Engines::EngineComponent_var comp =
       app->lcc()->FindOrLoad_Component( "FactoryServer",
       "ATOMSOLV" );
    ATOMSOLV_ORB::ATOMSOLV_Gen_ptr atomGen =
       ATOMSOLV_ORB::ATOMSOLV_Gen::_narrow(comp);
    ASSERT( !CORBA::is_nil( atomGen ) );
    myEngine = atomGen;
  }
}
ATOMSOLV_ORB::ATOMSOLV_Gen_var ATOMSOLVGUI::GetATOMSOLVGen()
{
  if ( CORBA::is_nil( myEngine ) ) {
    SUIT_Application* suitApp =
        SUIT_Session::session()->activeApplication();
    SalomeApp_Application* app =
        dynamic_cast<SalomeApp_Application*>( suitApp );
    InitATOMSOLVGen( app );
  }
  return myEngine;
}
QString ATOMSOLVGUI::engineIOR() const
{
  CORBA::String_var anIOR =
         getApp()->orb()->object_to_string( GetATOMSOLVGen() );
  return QString( anIOR.in() );
}
```

Method `GetATOMSOLVGen()` is made static to allow access to ATOMSOLV engine from different GUI classes. It is also possible to acquire a pointer to `ATOMSOLV_Gen` calling directly `FindOrLoad_Component()` method of `LifeCycleCORBA` interface that can be aquired from `SalomeApp_Application` instance.

Let's implement a method to retrieve data from ATOMGEN engine and store it in ATOMSOLV engine. It will be a slot connected to "Retrive data" action of ATOMSOLVGUI class:

```
1.  void ATOMSOLVGUI::OnRetrieveData()
2.  {
3.    ATOMSOLV_ORB::ATOMSOLV_Gen_var engine = GetATOMSOLVGen();
4.    SalomeApp_Application* app = getApp();
5.    if ( !CORBA::is_nil( engine ) && app ) {
6.      // acquire ATOMGEN engine: use LifeCycleCORBA service for it
7.      Engines::EngineComponent_var comp =
8.        app->lcc()->FindOrLoad_Component("FactoryServerPy","ATOMGEN");
9.      ATOMGEN_ORB::ATOMGEN_Gen_var atomGen =
10.       ATOMGEN_ORB::ATOMGEN_Gen::_narrow( comp );
11.     SalomeApp_Study* appStudy =
12.       dynamic_cast<SalomeApp_Study*>( app->activeStudy() );
13.     if ( !CORBA::is_nil( atomGen ) && appStudy ) {
14.       const int studyID = appStudy->id();
15.       // load study data if it is not done yet by ATOMGEN component
16.       if ( _PTR( Study ) studyDS = appStudy->studyDS() ) {
17.         if ( _PTR( SComponent ) atomGenSComp =
18.              studyDS->FindComponent( "ATOMGEN" ) ) {
19.           _PTR( StudyBuilder ) builder = studyDS->NewBuilder();
20.           std::string atomGenIOR =
21              app->orb()->object_to_string( atomGen );
22.           builder->LoadWith( atomGenSComp, atomGenIOR );
23.         }
24.       }
25.       // retrieve data from ATOMGEN
26.       ATOMGEN_ORB::MoleculeList_var inData =
27.          atomGen->getData( studyID );
28.       // "convert" Molecules to TMolecules,
29.       // set default temperature '0'
30.       const int n = inData->length();
31.       ATOMSOLV_ORB::TMoleculeList_var outData =
32.            new ATOMSOLV_ORB::TMoleculeList();
33.       outData->length( n );
34.       for ( int i = 0; i < n; i++ ) {
35.         ATOMSOLV_ORB::TMolecule_var tmol =
36            new ATOMSOLV_ORB::TMolecule();
37.         tmol->molecule =
38.           ATOMGEN_ORB::Molecule::_duplicate( inData[i] );
39.         tmol->temperature = 0;
40.         outData[ i ] = tmol;
41.       }
42.       // store the data in ATOMSOLV engine
43.       engine->setData( studyID, outData );
44.
45.       // update object browser so new data objects appear in it
46.       app->updateObjectBrowser();
47.     }
48.   }
49. }
```

As we see, first of all we acquire reference to both engines: of ATOMSOLV and ATOMGEN components (lines 3, 9). Then we must get the data from ATOMGEN that corresponds to the currently opened study. We obtain the integer study ID (14), and then use SALOMEDS services for loading of the study by the component ATOMGEN (16-22). It is necessary to do it, because internal data structure of ATOMGEN *may not be initialized* in case if ATOMGEN was not previously loaded and therefore it has not "connected" to the current study. We do this "connection" of ATOMGEN engine to study manually, calling `StudyBuilder::LoadWith()` method (22). After that we retrieve the data and convert it to format of ATOMSOLV (list TMolecules instead of Molecules), setting default temperature of 0 (25-41). Finally, we store the data in ATOMSOLV engine (43) and update the object browser in order to see the new data under ATOMSOLV root.

As you already know, in order to see the data structure in Object Browser, we have to create Data Model and Data Object classes, build a tree of Data Objects inside the Data Model (in its method `build()`), and connect the Data Model to component GUI. Let's assume, we have already done it, and take a look at the most interesting methods of Data Model and Data Object:

```
void ATOMSOLVGUI_DataModel::build()
{
  ATOMSOLVGUI_ModuleObject* modelRoot =
   dynamic_cast<ATOMSOLVGUI_ModuleObject*>( root() );
  if( !modelRoot )  {  // root is not set yet
    modelRoot = new ATOMSOLVGUI_ModuleObject( this, 0 );
    setRoot( modelRoot );
  }

  // create 'molecule' objects under model root object
  // and 'atom' objects under 'molecule'-s
  ATOMSOLV_ORB::ATOMSOLV_Gen_var engine =
    ATOMSOLVGUI::GetATOMSOLVGen();
  if ( !CORBA::is_nil( engine ) ) {
    const int studyID = getStudy()->id();
    ATOMSOLV_ORB::TMoleculeList_var molecules;
    if ( !engine->getData( studyID, molecules ) )
      return;

    for ( int i = 0, n = molecules->length(); i < n; i++ ) {
      ATOMSOLVGUI_DataObject* molDO =
         new ATOMSOLVGUI_DataObject ( modelRoot, i );
      const ATOMSOLV_ORB::TMolecule& mol = molecules[i];
      const int atoms = mol.molecule->getNbAtoms();
      for ( int j = 0; j < atoms; j++ )
        new ATOMSOLVGUI_DataObject ( molDO, i, j );
    }
  }
}
```

```
QString ATOMSOLVGUI_DataObject::entry() const
{
  QString id = "root";
  if ( myMoleculeIndex > -1 ) {
    id = QString::number( myMoleculeIndex );
    if ( myAtomIndex >= 0 )
    id += QString( "_%1" ).arg( myAtomIndex );
  }
  return QString( "ATOMSOLVGUI_%1" ).arg( id );
}

QString ATOMSOLVGUI_DataObject::name() const
{
  ATOMSOLV_ORB::TMolecule tmolecule = getTMolecule();
  ATOMGEN_ORB::Molecule_var mol = tmolecule.molecule;
  if ( !CORBA::is_nil( mol ) ) {
    if ( !isAtom() )
      return QString( "%1 [%2]" ).arg( mol->getName() ).arg(
           tmolecule.temperature );
    else if ( myAtomIndex < mol->getNbAtoms() )
      return mol->getAtom( myAtomIndex )->getName();
  }
  return QString("-Error-");
}

ATOMSOLV_ORB::TMolecule ATOMSOLVGUI_DataObject::getTMolecule() const
{
  ATOMSOLV_ORB::ATOMSOLV_Gen_var engine =
```

```
    ATOMSOLVGUI::GetATOMSOLVGen();
  LightApp_RootObject* rootObj =
    dynamic_cast<LightApp_RootObject*> ( root() );
  if ( rootObj && !CORBA::is_nil( engine ) ) {
    const int studyID = rootObj->study()->id();
    if ( studyID > 0 ) {
      ATOMSOLV_ORB::TMoleculeList_var molecules;
      if ( engine->getData( studyID, molecules ) &&
       myMoleculeIndex > -1 &&
       myMoleculeIndex < molecules->length() )
    return molecules[ myMoleculeIndex ];
    }
  }
  return ATOMSOLV_ORB::TMolecule();
}
```

Data Model builds a tree of Data objects in the following way: first of all it creates a root object in case it was not created before. Then it gets the list of `TMolecules` from engine, iterates it, and builds Data Objects for every `TMolecule`, and for every atom of `TMolecule`.

Data Object stores 2 integer indexes: index of `TMolecule(myMoleculeIndex)`, and index of atom within `TMolecule(myAtomicIndex)`. If `myAtomicIndex` is equal to '-1' then the Data Object corresponds to a molecule object, if `myAtomicIndex` is a valid index (>= 0), then the Data Object corresponds to an atom.

Please, download the source files of the current version of ATOMSOLV component, compile them, and start the application. ATOMGEN component must be also made available. It means that `ATOMGEN_ROOT_DIR` variable must be correctly set and ATOMGEN component must be added to the list of active components (with `--modules=ATOMSOLV,ATOMGEN` command line parameter, for example). After the application is started, switch to ATOMGEN and import an XML file with data prepared by ATOMIC component (for example, sample.xml from ATOMIC component, it is located in resources directory of ATOMIC). Now activate ATOMSOLV component and select *AtomSolv → Retrieve data* command. AtomSolv root object with molecules and atoms must appear in Object Browser - they were retrieved from ATOMGEN engine using CORBA technology!

If we choose *AtomSolv → Process data* command now, the molecules will be assigned new temperature properties (see `ATOMSOLVGUI::OnProcessData()` and `ATOMSOLV::processData()` methods). It is reflected in the Data Objects that correspond to the molecules: the number in square brackets is changed.

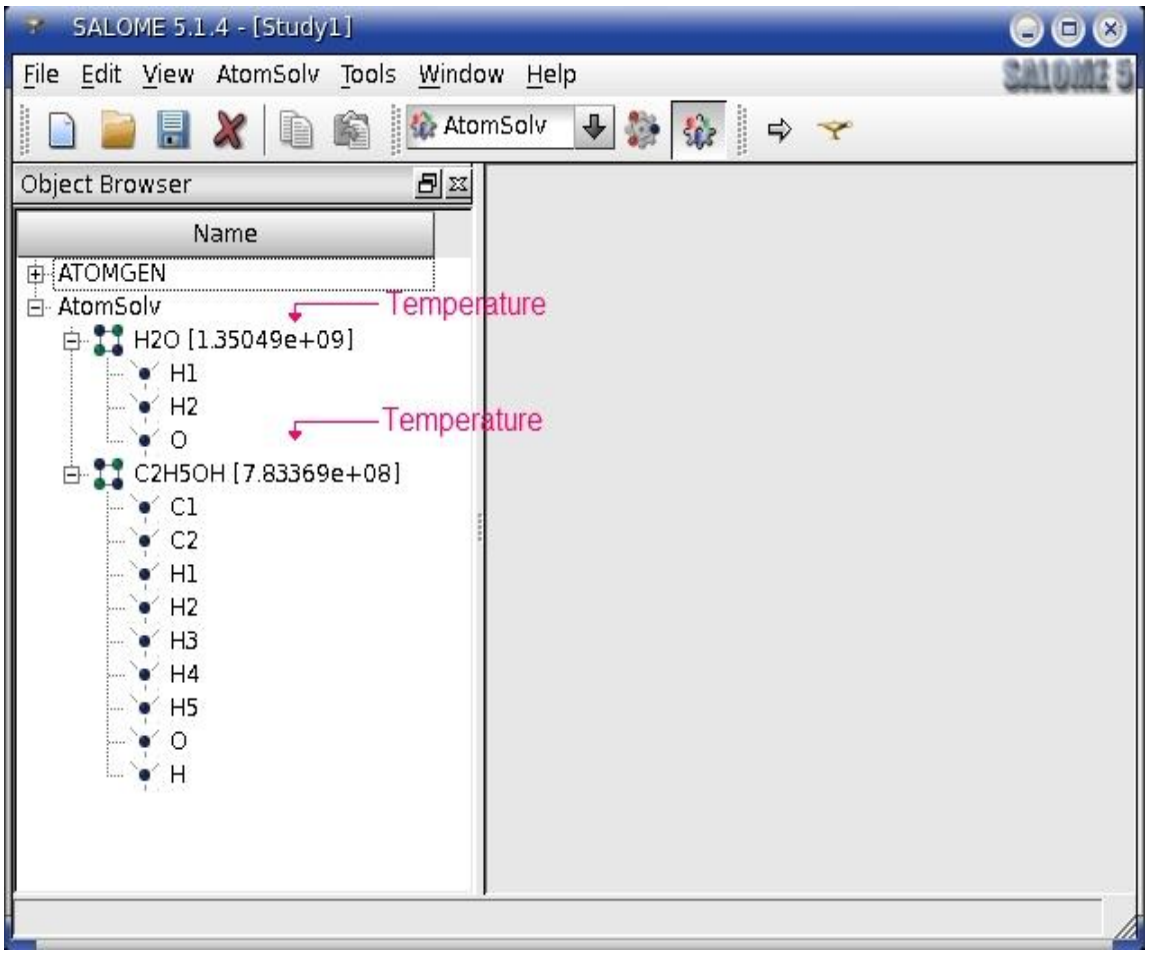

Figure 7. Post-processing with ATOMSOLV

In the next section we will learn how to display the atoms in 3D viewer.

5.3 GRAPHICAL CAPABILITIES

In this section we are going to learn how to display objects in 3D viewer. In our application the objects that are most suitable for 3D presentation are atoms as they have such property as Cartesian coordinates. But first of all, we must implement retrieval of objects selected in Object Browser (so we know *what* to display), and add the corresponding actions to popup menu (so we know *when* to display). We shall implement method `ATOMSOLVGUI::selected()` in a very similar to ATOMIC component way:

```
void ATOMSOLVGUI::selected (QStringList& entries, const bool
multiple)
{
  LightApp_SelectionMgr* mgr = getApp()->selectionMgr();
  if( !mgr )
    return;

  SUIT_DataOwnerPtrList anOwnersList;
  mgr->selected( anOwnersList );

  for ( int i = 0; i < anOwnersList.size(); i++ )
  {
    const LightApp_DataOwner* owner =
      dynamic_cast<const LightApp_DataOwner*>(
      anOwnersList[ i ].get() );
    QStringList es = owner->entry().split( "_" );
    if ( es.count() > 1 && es[ 0 ] == "ATOMSOLVGUI" &&
```

```
          es[ 1 ] != "root" )
      {
        if ( !entries.contains( owner->entry() ) )
          entries.append( owner->entry() );
        if( !multiple )
          break;
      }
    }
  }
```

We would like to add the following actions to the popup menu of atoms (displayable objects): Display, Erase, Representation mode - Points, Representation mode - Wireframe, Representation mode - Surface, Change color, Change Transparency. So called "representation mode" (or "display mode") is the way the object looks in 3D viewer:

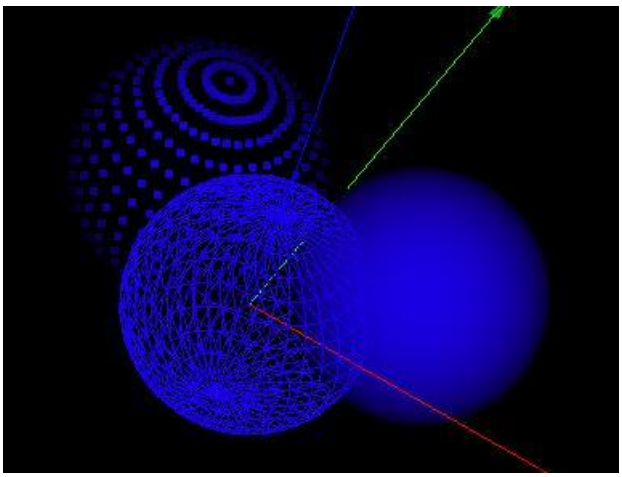

Figure 8. Representation modes

We would like to be able to change this property, so we create the corresponding actions. Color and transparency are also 3D presentation properties, they can be also modified:

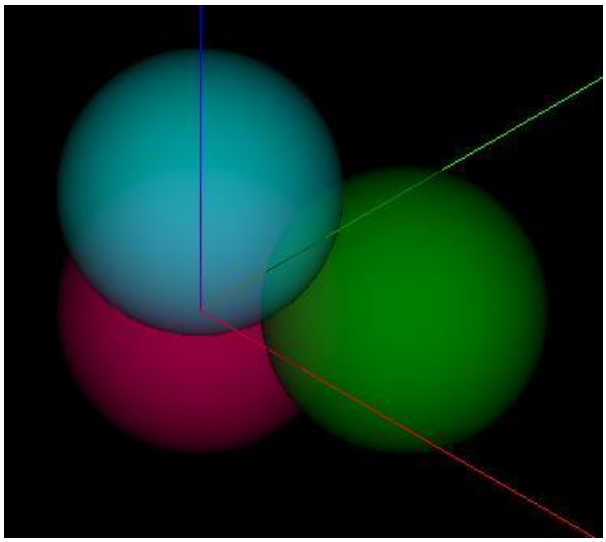

Figure 9. Colored atoms

The logical rules for adding Display / Erase commands to popup menu will be the following:

```
mgr->setRule( action( Display ), "true in $canBeDisplayed and
              activeModule='ATOMSOLV' and !isVisible" );
mgr->setRule( action( Erase ), "true in $canBeDisplayed and
              activeModule='ATOMSOLV' and isVisible" );
```

The parameter `canBeDisplayed` is analyzed by the `LightApp_Selection` class when a popup menu is constructed. `LightApp_Selection` requests from a GUI module a special object called Displayer (ancestor of `LightApp_Displayer` class). Displayer has a virtual function `canBeDisplayed()` which is called with entry of an object and type of viewer as input parameter. If it returns true, then `canBeDisplayed` popup menu parameter becomes true for this object. Let's create `ATOMSOLVGUI_Displayer` class and implement its `canBeDisplayed()` method:

```
bool ATOMSOLVGUI_Displayer::canBeDisplayed( const QString& entry,
                            const QString& viewer_type ) const
{
  QStringList es = entry.split( "_" );
  bool result = ( es.count() == 3 && es[ 0 ] == "ATOMSOLVGUI" &&
                  viewer_type == SVTK_Viewer::Type() );
  return result;
}
```

The code **`es.count() == 3 && es[ 0 ] == "ATOMSOLVGUI"`** guarantees that the entry belongs to an atom (only atom objects has 3 parts divided with underscore, and the first part is "ATOMSOLVGUI" string). The second part of the logical condition **`viewer_type == SVTK_Viewer::Type()`** means that our atoms can be displayed only in VTK viewer.

Here we have to say a few words about different types of viewers adopted in SALOME platform. GUI module of SALOME currently contains the following packages responsible for displaying objects in various ways:

GLViewer	2D presentation of objects, developed specially for SALOME platform using core Open GL libraries
OCCViewer, SOCC	3D presentation based on Open CASCADE technology (AIS, V3d packages from Open CASCADE library: http://www.opencascade.org)
Plot2d, SPlot2d	2D presentation based on QWT toolkit (open source Qt-based library: http://qwt.sourceforge.net), mainly aimed to display graphs and curves in 2D
VTKViewer, SVTK	3D presentation based on Visualization TookKit (open source toolkit by Kitware, Inc.: www.vtk.org)

All visualization packages of GUI module presented above contain several obligatory classes inherited from classes of SUIT package that allow for *abstraction* from a certain way the object visualization is implemented in this package. The interface of all visualization packages is the same, the implementation, of course, differs. In the table below we will try to describe the base classes and their objectives:

`SUIT_ViewWindow`	View window is a frame, inherited from `QMainWindow` (Qt library), that contains the visualization scene. Objects are displayed inside a view window. The visualization packages inherit their custom view windows from `SUIT_ViewWindow` and fill it with custom widgets in order to display objects in a certain way. OCCViewer, for example, places a V3d_View (Open CASCADE library) inside its view window to display a 3D scene. View window is able to save its contents as an image (`dumpView()` and `dumpViewToFormat()` virtual functions) and it is able to save and restore its parameters (values of zoom, pan, degree rotation, and other custom properties of the scene) - `get/setVisualParameters()` virtual functions. These functions are redefined in every custom view window to perform the corresponding functionality. As view window is a basic frame, it receives the basic window events:

	mouse moves, clicks, keyboard presses, etc. One of the objectives of view window is to pass these events further - it is done through various signals emitted by `SUIT_ViewWindow` class: `mousePressed()`, `mouseReleased()`, `weeling()`, `keyPressed()`, etc.

`SUIT_ViewModel`	View model is a creator of View windows. "View model - View window" pair follows "Factory method" pattern - view windows are created by virtual method of view model `createView()`. Visualization packages of GUI module redefine `SUIT_ViewModel` class to be able to create custom view windows.
	View models have a pair of methods:
	`static QString Type();`
	`virtual QString getType() const;`
	These methods must return a type descriptor of a view ("OCCViewer", "VTKViewer", etc.). This type is used in context popup menu ("client" parameter will be equal to this type), and in many other places in the code where it is needed to determine the type of a view.

`SUIT_ViewManager`	The name of the class shows its main purpose: it manages the views (view windows). It contains a view model as a member field for creation of a view and various methods for accessing the managed views (`getActiveView()`, `getViewsCount()`, etc.). `SUIT_ViewManager` is a "gateway" class for working with view windows from application or another side. `STD_Application` class (parent class for `LightApp_Application` and `SalomeApp_Application`) stores view managers and it is possible to retrieve a view manager of a certain type using methods of the application.

The classes presented above from SUIT package play the role of an abstraction layer that generalizes ***where*** the objects are displayed (view window). Now we have to observe the objects themselves.

Different viewers naturally work with different internal graphic presentations. OCCViewer, for example, uses `AIS_InteractiveObject` class for presentation of an object in the graphic scene; VTKViewer uses `vtkActor` for the same purpose. The goal of SALOME platform is to support a unified way of working with presentation objects of different types. This is done using an abstraction layer declared in Prs package of GUI module. This package contains only 2 files (`SALOME_Prs.h/cxx`), they contain declaration and implementation of several classes described in the table below:

SALOME_Prs	`SALOME_Prs` is an abstraction of an object presentation. Generalization of ***what*** to display (the object). It is a pure interface that represents "something" that can be displayed in `SALOME_View` (another abstraction).

SALOME_OCCPrs, SALOME_VTKPrs, SALOME_Prs2d	Direct successors of `SALOME_Prs`. The implementation of these classes is very simple, they delegate the call to `SALOME_View`. For example:
	`void SALOME_OCCPrs::DisplayIn( SALOME_View* v ) const`
	`{`
	`  if ( v ) v->Display( this );`
	`}`

SALOME_View	`SALOME_View` is an abstraction of ***how*** to display (type of viewer). `SALOME_View` interface (it is more an interface as the base implementation leaves the methods empty) is inherited by View Models of the visualization packages (`SOCC_ViewModel`, `SVTK_ViewModel`, etc.). These view models know how to display `SALOME_Prs` of a certain type. Naturally, `SVTK_ViewModel` redefines only 1 virtual `Display()` method of `SALOME_View` interface - the one that takes a `SALOME_VTKPrs` as a parameter. It will know exactly how to treat `SALOME_VTKPrs` object, how to get the internal displayable object from it (`vtkActor`) and display it in the active VTK view window.
SALOME_Displayer	`SALOME_Displayer` interface plays a role of a display manager. It is used by application (`LightApp_Application` or `SalomeApp_Application`), GUI module of a component (`LightApp_Module` or `SalomeApp_Module`) when it needs to display or erase the objects. Displayer is the only class that usually needs to be redefined by a component in order to display its objects. Custom Displayer class must be derived from `LightApp_Displayer`. Usually there is only one virtual method of `LightApp_Displayer` to be redefined in a custom Displayer class for displaying of an object in all types of viewers: `virtual SALOME_Prs* buildPresentation( const QString& entry, SALOME_View* = 0 );`

Let's create our own Displayer class (in fact, we have already done it above) and redefine `buildPresentation()` method:

```
SALOME_Prs* ATOMSOLVGUI_Displayer::buildPresentation( const QString&
entry, SALOME_View* view )
{
  const int studyID = getStudyID();
  if ( studyID == -1 )
    return 0;

  SVTK_Prs* prs = dynamic_cast<SVTK_Prs*>(
     LightApp_Displayer::buildPresentation( entry, view ) );

  if ( !prs ) return 0;

  double temperature;
  ATOMGEN_ORB::Atom_var atom = getAtom( entry, studyID,
                                        temperature );

  if ( !CORBA::is_nil( atom ) ) {
    double center[ 3 ];
    center[ 0 ] = atom->getX();
    center[ 1 ] = atom->getY();
    center[ 2 ] = atom->getZ();

    vtkSphereSource* vtkObj = vtkSphereSource::New();
    vtkObj->SetRadius( radius );
    vtkObj->SetCenter( center );
    vtkObj->SetThetaResolution( (int)( vtkObj->GetEndTheta() *
         quality_coefficient ) );
    vtkObj->SetPhiResolution( (int)( vtkObj->GetEndPhi() *
         quality_coefficient ) );

    vtkPolyDataMapper* vtkMapper = vtkPolyDataMapper::New();
    vtkMapper->SetInput( vtkObj->GetOutput() );
```

```
    vtkObj->Delete();

    SALOME_Actor* actor = SALOME_Actor::New();
    actor->SetMapper( vtkMapper );
    actor->setIO( new SALOME_InteractiveObject( entry.toLatin1(),
            "ATOMSOLV" ) );
    setTemperature( actor, temperature );

    actor->SetRepresentation( 2 ); // 2 == surface mode

    vtkMapper->Delete();

    prs->AddObject( actor );
  }

  return prs;
}
```

The main lines in the code above are highlighted in bold font: creation of `vtkActor` (`SALOME_Actor` is its ancestor), and filling of `SALOME_Prs` object by this `vtkActor` (`SVTK_Prs` to be more specific). Such `SALOME_Prs` object will be possible to display in VTK viewer. `SVTK_ViewModel`, as an ancestor of `SALOME_View` will receive this `SALOME_Prs`, downcast it to `SVTK_Prs`, get `vtkActor` from it - and display it in the active view window. That's how it works in SALOME.

As we decided to work with VTK viewer (`canBeDisplayed()` method of `ATOMSOLV_Displayer` returns "true" only if type of viewer is "VTKViewer"), we must create a VTK view window when our component GUI starts. This is done using virtual method of `LightApp_Module` `viewManagers()`. We must return in the out parameter of this method the list of viewer types that we want to use. In our case, the implementation will be the following:

```
void ATOMSOLVGUI::viewManagers( QStringList& theViewMgrs ) const
{
  theViewMgrs.append( SVTK_Viewer::Type() );
}
```

OK, now we have to return to the new commands that we were going to implement in ATOMSOLV GUI: changing of 3D representation mode (points, wireframe, surface mode), changing of color and transparency. We are going to create new actions in ATOMSOLVGUI and create the logical rules for these actions, so they are correctly added to the popup menu:

```
1.  createAction( Shading, tr( "TLT_SHADING" ), QIcon(),
2.  tr("MEN_SHADING" ), tr( "STS_SHADING" ), 0, aParent, true, this,
3.      SLOT( OnDisplayerCommand() ) );
4.  createAction( Wireframe, tr( "TLT_WIREFRAME" ), QIcon(),
5.      tr( "MEN_WIREFRAME" ), tr( "STS_WIREFRAME" ), 0,
6.      aParent, true, this, SLOT( OnDisplayerCommand() ) );
7.  createAction( PointsMode, tr( "TLT_POINTSMODE" ), QIcon(),
8.      tr( "MEN_POINTSMODE" ), tr( "STS_POINTSMODE" ), 0, aParent,
9.      true, this, SLOT( OnDisplayerCommand() ) );
10. createAction( Color, tr( "TLT_COLOR" ), QIcon(),
11.     tr( "MEN_COLOR" ), tr( "STS_COLOR" ), 0, aParent, false,
12.     this, SLOT( OnDisplayerCommand() ) );
13. createAction( Transparency, tr( "TLT_TRANSPARENCY" ), QIcon(),
14.     tr( "MEN_TRANSPARENCY" ), tr( "STS_TRANSPARENCY" ), 0,
15.     aParent, false, this, SLOT( OnDisplayerCommand() ) );
16. QtxPopupMgr* mgr = popupMgr();
17. int dispmodeId = mgr->insert( tr( "MEN_DISPLAY_MODE" ), -1, -1 );
18. mgr->insert( action(  PointsMode ), dispmodeId, -1 );
19. mgr->insert( action(  Wireframe ), dispmodeId, -1 );
20. mgr->insert( action(  Shading ), dispmodeId, -1 );
```

```
21. mgr->insert( action( Color ), -1, 0 );
22. mgr->insert( action( Transparency ), -1, 0 );
23. mgr->setRule( action( PointsMode ), "client='VTKViewer' and
24.    selcount>0 and isVisible", true );
25. mgr->setRule( action( Wireframe ), "client='VTKViewer' and
26.    selcount>0 and isVisible", true );
27. mgr->setRule( action( Shading ), "client='VTKViewer' and
28.    selcount>0 and isVisible", true );
29. mgr->setRule( action( Color ), "client='VTKViewer' and
30.    selcount>0 and isVisible", true );
31. mgr->setRule( action( Transparency ), "client='VTKViewer' and
32,    selcount>0 and isVisible", true );
33. mgr->setRule( action( PointsMode ), "$displaymode={'Points'}",
34.    QtxPopupMgr::ToggleRule );
35. mgr->setRule( action( Wireframe ), "$displaymode={'Wireframe'}",
36.    QtxPopupMgr::ToggleRule );
37. mgr->setRule( action( Shading ), "$displaymode={'Surface'}",
38.    QtxPopupMgr::ToggleRule );
```

The last section of the code (lines 33-38) is rather interesting: we make the representation mode actions to be toggle actions ("checkable" menu items) and set logical rules for the toggle status. (If `QtxPopupMrg::setRule()` function is called with the last parameter equal to `QtxPopupMgr::ToggleAction`, then it sets the rule for the toggle status, and not for command visibility as it would be with default last parameter).

Let's take a closer look at the rules themselves. In natural language they would mean: "values of *displaymode* parameter must be equal to the list that contains 1 element ('Points', 'Wireframe', or 'Surface')". Such rule differs from the rule "displaymode='Points'" (or 'Wireframe', or 'Surface'), because in case when 2 elements with different representation modes will be selected, the first rule would return "false" result for both actions (because value of displaymode parameter will be equal to the list containing 2 different elements), although the second rule would return "true" for both actions.

The new parameter "displaymode" that we are using in the logical rules is not computed anywhere for us (as "client" or "selcount" parameters). We have to create a custom Selection class and compute this parameter in it. We already got acquainted with this mechanism of parameters computation in ATOMIC component (see Selection section of Light-weight component chapter), so here we will just present the function of our custom Selection class (`ATOMSOLVGUI_Selection`), which computes the new "displaymode" parameter:

```
QString ATOMSOLVGUI_Selection::displayMode( const int index ) const
{
  SALOME_View* view = LightApp_Displayer::GetActiveView();
  QString viewType = activeViewType();
  if ( view && viewType == SVTK_Viewer::Type() ) {
    if ( SALOME_Prs* prs = view->CreatePrs(
         entry( index ).toLatin1() )){
      SVTK_Prs* vtkPrs = dynamic_cast<SVTK_Prs*>( prs );
      vtkActorCollection* lst = vtkPrs ? vtkPrs->GetObjects() : 0;
      if ( lst ) {
        lst->InitTraversal();
        vtkActor* actor = lst->GetNextActor();
        if ( actor ) {
          SALOME_Actor* salActor =
            dynamic_cast<SALOME_Actor*>( actor );
          if ( salActor ) {
            int dm = salActor->GetRepresentation();
            if ( dm == 0 )
              return "Points";
            else if ( dm == 1 )
              return "Wireframe";
            else if ( dm == 2 )
```

```
              return "Surface";
            } // if ( salome actor )
          } // if ( actor )
        } // if ( lst == vtkPrs->GetObjects() )
      }
    }
    return "Undefined";
  }
```

The last thing that we must implement is the slot in `ATOMSOLVGUI` class to which all visualization actions are connected (Display, Erase, Representation modes, Color, Transparency). We will also implement changing of representation mode, color, and transparency in `ATOMSOLVGUI_Displayer` class.

```
1.   void ATOMSOLVGUI::OnDisplayerCommand()
2.   {
3.     const QObject* obj = sender();
4.     if ( obj && obj->inherits( "QAction" ) ) {
5.       const int id = actionId ( (QAction*)obj );
6.       switch ( id ) {
7.       case Display : {
8.         QStringList entries;
9.         selected ( entries, true );
10.        ATOMSOLVGUI_Displayer d;
11.        for ( QStringList::const_iterator it = entries.begin(),
12.              last = entries.end(); it != last; it++ )
13.          d.Display( it->toLatin1(), /*updateviewer=*/false, 0 );
14.        d.UpdateViewer();
15.      } break;
16.      case Erase   : {
17.        QStringList entries;
18.        selected ( entries, true );
19.        ATOMSOLVGUI_Displayer d;
20.        for ( QStringList::const_iterator it = entries.begin(),
21.              last = entries.end(); it != last; it++ )
22.          d.Erase( *it, /*forced=*/true, /*updateViewer=*/false, 0
);
23.        d.UpdateViewer();
24.      } break;
25.      case Shading   : {
26.        QStringList entries;
27.        selected ( entries, true );
28.        ATOMSOLVGUI_Displayer().setDisplayMode(entries, "Surface"
);
29.      } break;
30.      case Wireframe   : {
31.        QStringList entries;
32.        selected ( entries, true );
33.        ATOMSOLVGUI_Displayer().setDisplayMode(entries, "Wireframe");
34.      } break;
35.      case PointsMode   : {
36.        QStringList entries;
37.        selected ( entries, true );
38.        ATOMSOLVGUI_Displayer().setDisplayMode( entries, "Points"
);
39.      } break;
40.      case Color   : {
41.        QStringList entries;
42.       selected ( entries, true );
43.        QColor initialColor( "white" );
44.        if ( entries.count() == 1 )
45.          initialColor=ATOMSOLVGUI_Displayer().getColor(entries[0]);
46.        QColor color = QColorDialog::getColor( initialColor,
```

```
47.           getApp()->desktop() );
48.       if ( color.isValid() )
49.         ATOMSOLVGUI_Displayer().setColor( entries, color );
50.     } break;
51.     case Transparency   : {
52.       QStringList entries;
53.       selected ( entries, true );
54.       ATOMSOLVGUI_TransparencyDlg( getApp()->desktop(),
55.                                    entries ).exec();
56.     } break;
57.     default: printf( "ERROR: Action with ID = %d was not found
58.                      in ATOMSOLVGUI\n", id ); break;
59.     }
50.   }
61. }
```

As we see in the code above, to display an object we call basic method of Displayer class: `Display()` (lines 7-14). This method is not implemented in `ATOMSOLVGUI_Displayer`, but the core implementation in the parent class (`LightApp_Displayer`) will call virtual function `buildPresentation()` which is redefined in `ATOMSOLVGUI_Displayer`.

Please, download the current version of ATOMSOLV component source files, compile it and run. Don't forget to start ATOMGEN component as well and import a valid XML file with data in it (sample.xml from ATOMIC component resources). Then retrieve the data in ATOMSOLV, select atoms in Object Browser and display them in the viewer. Pay attention to the methods of `ATOMSOLV_Displayer`: how it assigns new color to presentations of atoms when "Process Data" is called (`updateActor()`, `setTemperature()` methods), how it sets new representation mode (`setDisplayMode()` method), and transparency (`setTransparency()`).
For setting transparency we use a separate modal dialog box `ATOMSOLVGUI_TransparencyDlg`, which calls `setTransparency()` method of Displayer every time user moves the transparency control (slider bar).

As an exercise, we would propose you to upgrade ATOMSOLV component and make it possible to display atoms in OCCViewer (another type of 3D viewer). Don't forget to add modifications in the following methods:

- `ATOMSOLVGUI::viewManagers()` - add a new type of view manager.
- `ATOMSOLVGUI_Displayer::canBeDisplayed()` -- must return true not only for VTK viewer
- `ATOMSOLVGUI_Displayer::buildPresentation()` -- creation of `SOCC_Prs` object in case the given view is of OCCViewer type. The method already contains commented code for building `SOCC_Prs` object.
- Get/set methods for representation mode, color, transparency must take into account the type of viewer! Try to use methods of `SALOME_View` interface for simplification.
- `ATOMSOLVGUI::OnProcessData()` must be modified to work not only with vtkActor-s. Redisplaying of objects must be done in a more general way then present implementation.

At this point we would like to finish the section about graphical capabilities of SALOME and switch to the next section about user-defined preferences in a SALOME application.

5.4 PREFERENCES

In the previous section we implemented visualization of atoms as 3D spheres with hard-coded values of radius and initial representation mode. Such approach is not flexible and does not meet requirements of an industrial application. Even our ATOMSOLV component is not an industrial application, we would like it to be as good as possible, and in the present section we will learn how

a user can edit values of radius and default representation mode at run time and `ATOMSOLV_Displayer` can retrieve these values and use them.

Such values that can be edited by user are called *preferences*. Their edition is done in Preferences dialog box shown when a menu item *File → Preferences* is selected (see Figure 10).

A SALOME component can easily add its own editable values to this dialog box. The values can be retrieved using a global Resource Manager object. For example, the value of "Multi file save" check-box in the dialog box above can be retrieved using the following call:

```
SUIT_Session::session()->resourceMgr()->booleanValue("Study",
                "multi_file", false);
```

"Study" and "multi_file" are descriptors of the "Multi file save" preference (they were indicated during its creation), "false" is the default value, it will be returned in case Resource Manager fails to find the preference with given descriptors.

In ATOMSOLV component we would like to add the following parameters to be edited by user: floating point parameter radius, and representation mode parameter that can be equal to one of the 3 predefined values: "Points", "Wireframe", "Surface". How can this be done?

As we see in the dialog box above, the list-box on the left contains the names of all available components. But only untill the GUI modules of these components are loaded. As soon as GUI modules are loaded, they are requested for their preferences. If a component does not have any preferences (in case of ATOMGEN and ATOMSOLV component - it is true), the name of the component is removed from the list-box. So if we load ATOMGEN and ATOMSOLV components, and open Preferences dialog box after that -- we will not see the names of our components in the list-box.

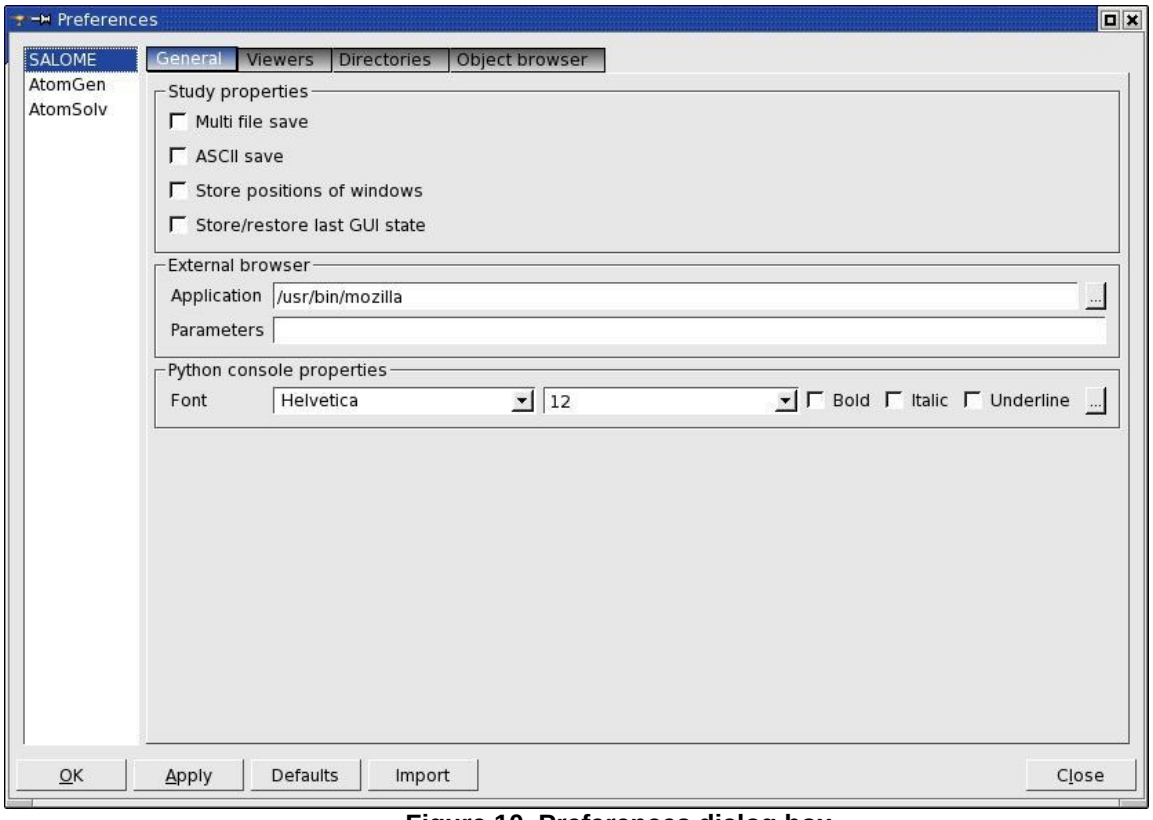

Figure 10. Preferences dialog box

To add user-editable parameters to the Preferences dialog box a component GUI module must redefine virtual method `createPreferences()`. `ATOMSOLVGUI` class will do it in the following way:

```
1.  void ATOMSOLVGUI::createPreferences()
2.  {
3.    int tabId = addPreference( tr( "ATOMSOLV_PREFERENCES" ) );
4.    int groupId = addPreference( tr( "PRESENTATION_PREF_GROUP" ),
5.                  tabId );
6.    setPreferenceProperty( groupId, "columns", 1 );
7.    // Representation mode preference
8.    int dispModeId = addPreference( tr( "DISPLAY_MODE_PREF" ),
9.                 groupId, LightApp_Preferences::Selector,
10.                "ATOMSOLV", "Representation" );
11.   QList<QVariant> intDispModes;
12.   QStringList strDispModes;
13.   intDispModes.append( 0 );
14.   strDispModes.append( tr( "MEN_POINTSMODE" ) );
15.   intDispModes.append( 1 );
16.   strDispModes.append( tr( "MEN_WIREFRAME" ) );
17.   intDispModes.append( 2 );
18.   strDispModes.append( tr( "MEN_SHADING" ) );
19.   setPreferenceProperty( dispModeId, "strings", strDispModes );
20.   setPreferenceProperty( dispModeId, "indexes", intDispModes );
21.   // Radius preference
22.   int radisusId = addPreference( tr( "RADIUS_PREF" ), groupId,
23.           LightApp_Preferences::DblSpin, "ATOMSOLV", "Radius" );
24.   setPreferenceProperty( radisusId, "min", .001 );
25.   setPreferenceProperty( radisusId, "max", 1000 );
26.   setPreferenceProperty( radisusId, "precision", 3 );
27. }
```

Adding preferences is very simple. We use only 1 method to create a separate tab for preferences of ATOMSOLV component (line 3), the same method to create a group (QGroupBox) for our preferences (line 4), and the same method for creation of editable parameters as well (lines 8, 22). The method is `addPreference()`, it is inherited from `LightApp_Module` class.

If `addPreference()` is called with 1 parameter it creates a tab, with 2 - it creates a group (a tab ID must be passed as second parameter), with more then 2 - it creates a control for editing of a certain value. The type of control is passed as a third parameter (different control types are described in the table below), 4th and 5th parameters are descriptors of the value being edited. Later retrieval of the value using Resource Manager must use these descriptors.

`LightApp_Preferences::Space`	Pseudo parameter type, it has NO control for edition. It might be useful when a large number of controls are added, and they are aligned in grid, and it is needed to leave empty space instead of a control in the grid - in this case a control with Space type can be created.
`LightApp_Preferences::Bool`	Boolean parameter, control for edition is a check box.
`LightApp_Preferences::Color`	Color parameter, control for edition is a push button that displays the color, when pressed it opens a standard system dialog box for color selection.
`LightApp_Preferences::String`	String parameter, control for edition is a line edit control.
`LightApp_Preferences::Selector`	Parameter with predefined list of values. Control for edition is a combo box.
`LightApp_Preferences::DblSpin`	Floating point parameter, control for edition is a

	modified spin box for floating point values.
`LightApp_Preferences::IntSpin`	Integer parameter, control for edition is a standard spin box.
`LightApp_Preferences::Double`	Floating point parameter, control for edition is a line edit control.
`LightApp_Preferences::Integer`	Integer parameter, control for edition is a line edit control.
`LightApp_Preferences::GroupBox`	Pseudo parameter type, creates a group box that can be used for grouping of other controls.
`LightApp_Preferences::Font`	Font parameter, control for edition is complex, it allows to set the name of font family, size of font, set bold, italic, and underline attributes. It also allows to open a standard system "Select font" dialog box and select the font there.
`LightApp_Preferences::DirList`	Directory list parameter, control for edition is a list of strings (directories) with possibility to add, remove, and change the order of strings in it.
`LightApp_Preferences::File`	File parameter, control for edition is a line edit control and a button that opens a standard system "Open file" dialog.
`LightApp_Preferences::User`	Pseudo parameter type, no control for edition, it must be used in successors of Preferences class for new preferences type s in the future.

After we have added the necessary parameters, they must be adjusted. For example, by default the preferences are grouped in 2 columns on a tab. In our case 1 column would look better, and we add the following line in `createPreferences()` method:

```
setPreferenceProperty( groupId, "columns", 1 );
```

It makes the controls in the previously created group with ID = `groupId` to be aligned in 1 column, one under another.

For Selector control it is necessary to install the list of values. It is possible to set up 2 lists: one list of displayable values ("strings"), and another list of values to be returned ("indexes"). In our case it is very helpful, because representation mode "Surface" has ID of 2, for example. We set up 2 properties for representation mode preference (with `dispModeId`) -- see lines 11-18 in the code above.

For the Radius parameter we will set up 3 properties: minimum value, maximum value, and precision. How it is done - shown on lines 24-26 in the code above.

The next important task is to track changes of our preferences. User may open Preferences dialog box at any time and modify values of our parameters. ATOMSOLV component must reflect to these changes, and draw next atoms with new radius, for example, in case the radius was changed. Tracking these changes is very easy: we have to redefine one more virtual method in `ATOMSOLVGUI: preferencesChanged()`:

```
void ATOMSOLVGUI::preferencesChanged( const QString& group, const
QString& param )
{
```

```
  if ( group == "ATOMSOLV" && param == "Representation" )
    ATOMSOLVGUI_Displayer::setDefaultRepresentation(
       getApp()->resourceMgr()->integerValue( group, param ) );
  else if ( group == "ATOMSOLV" && param == "Radius" )
    ATOMSOLVGUI_Displayer::setDefaultRadius(
       getApp()->resourceMgr()->doubleValue( group, param ) );
}
```

We implemented 4 static methods in `ATOMSOLVGUI_Displayer`: `setDefaultRepresentation()/setDefaultRepresentation()` and `setDefaultRadius()/setDefaultRadius()`. If a corresponding parameter is changed in Preferences dialog box - we call a "set" method of Displayer, and inside the Displayer we naturally use a "get" method to retrieve the value.

Please, download the last final version of ATOMSOLV source files with editable preferences.

At this point we would like to finish our *Getting started with SALOME platform* tutorial. We believe that now you are ready for development of new SALOME-based components on your own. If you have questions or/and suggestions - please, we will be happy to hear them from you on SALOME platform web forum: http://www.salome-platform.org/forum/!

6. SALOME CONCEPTS

6.1 KERNEL CONCEPTS

6.1.1 Build configurations

2 core modules of SALOME platform - KERNEL and GUI - can be compiled and run in 2 configurations (can be understood as *versions*): **full** and **light**.

- Light configuration means that all CORBA-based services are disabled. To build the modules in light configuration `-DSALOME_LIGHT_ONLY=ON` parameter must be passed to the `cmake` command (see chapter "SALOME build procedure" for details). To run SALOME in light configuration a command `runLightSalome.csh` (or `runLightSalome.sh`) from GUI module is used.

- Building in full configuration enables all CORBA services (`-DSALOME_LIGHT_ONLY=OFF` parameter of `cmake` command, this option is used by default). To run SALOME in full configuration a command `runSalome` from KERNEL module is used.

Light-weight components can work with SALOME built in both light and full configurations. In other words, a light-weight component can be a part of a multi-component SALOME application, and the other components do not have to be necessarily light-weight. But if all components are light-weight (in particular, if there is only 1 light-weight component in an application), then it is preferable to use KERNEL and GUI modules in light configuration. This will increase the application performance since a number of unused CORBA-based services will not be started.

Full-weight components can be compiled and run only if KERNEL and GUI are built in full configuration.

6.1.2 Component

A component is the base concept of SALOME platform. It can be understood as a separate software application - a piece of software which is dedicated to do a certain functionality. Examples of components are:

- GEOM component - allows user to create geometrical data using various algorithms of creation and modification of geometrical primitives

- Post-Pro component - allows user to display results of numerical computations using sophisticated visualization technologies

- YACS component - allows user to perform complicated data processing using other components and embedded Python

- etc.

SALOME toolkit introduces a multi-component approach. It means that a SALOME application consists of 1 or more components that operate at the same time sharing several **common** objects: **GUI objects** such as *desktop* with main menu, *Objects Browser*, *Python console* panel, etc., and **non-GUI objects** built on CORBA technology. Being completely different in functionality, SALOME components have a lot in common - architecture, internal structure of data and algorithms, common source code at the base level. Custom components are developed with high degree of code reuse which in turn greatly improves the quality of new custom components.

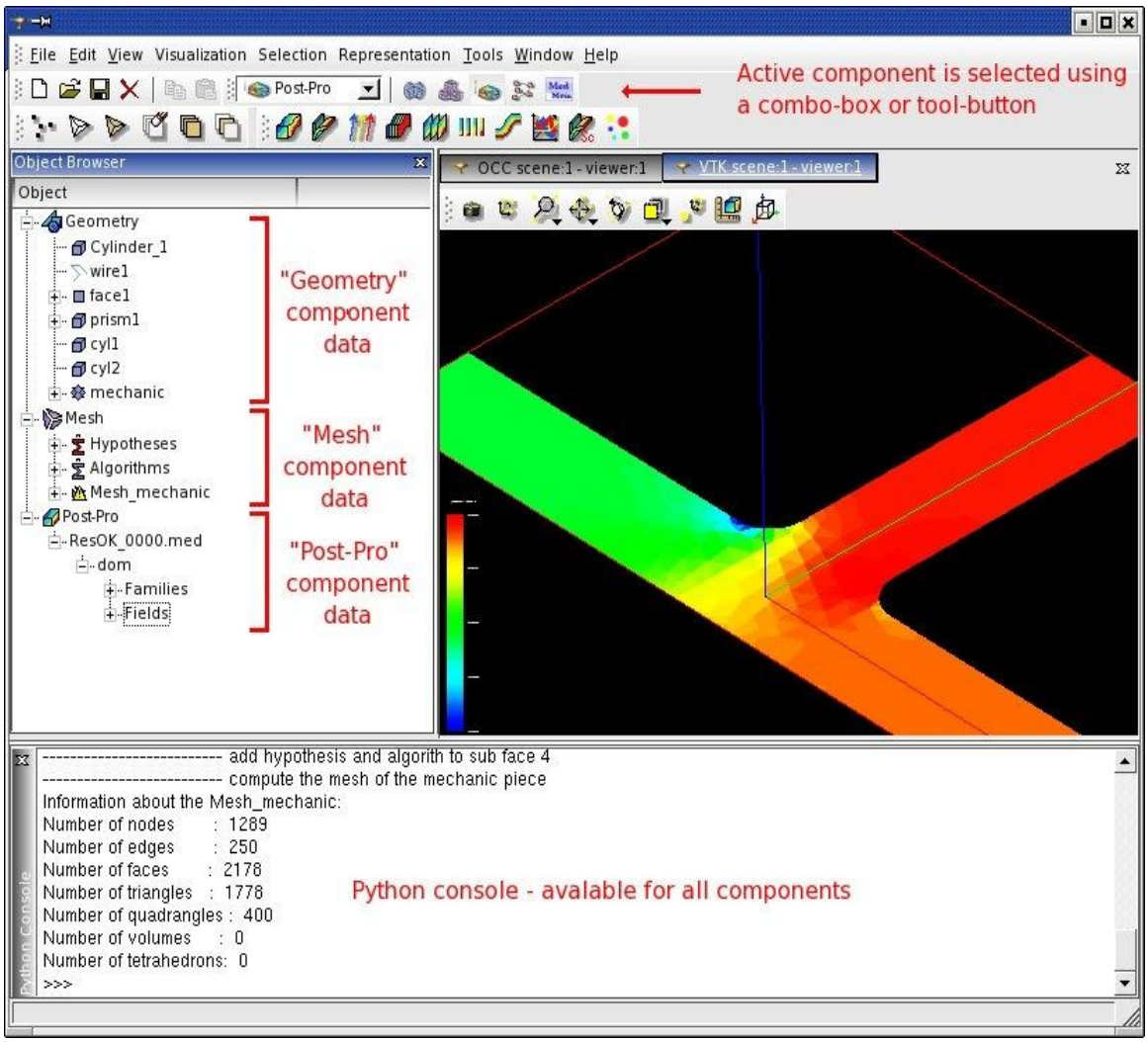

Figure 11. A multi-component SALOME application

A component's architecture usually consists of a GUI part and an algorithmic part embedded into one single object with CORBA interface that is called an engine. If a component does not have CORBA engine and does not use other CORBA services (provided by SALOME platform KERNEL module or other components), then such component is called a light-weight component. Components with CORBA engine can be said full-weight components, but usually they are referred just as "components".

Another partition of components is done on the basis of programming language used for a component development. Currently components of SALOME platform can be written in C++ and in Python language.

6.1.3 C++ component

A component written in C++ programming language.

Light-weight C++ components should inherit corresponding classes from `LightApp` package of SALOME GUI: component's GUI module class inherits `LightApp_Module`, component's operation class inherits `LightApp_Operation`, etc.

CORBA engines of C++ components should implement `Engines::EngineComponent` interface declared in SALOME_Component.idl file of KERNEL module. GUI classes of components with CORBA engine should inherit classes of `SalomeApp` package of GUI module: GUI module class - `SalomeApp_Module`, etc.

Please, refer to “ATOMIC: light-weight component” and “ATOMSOLV: C++ component with engine” chapters of the tutorial for further details.

6.1.4 CORBA engine

A part of a component, which is built using CORBA technology and implements `Engines::EngineComponent` interface declared in SALOME_Component.idl file of KERNEL module. Engine of a component usually performs algorithmic data processing. Its services may be used by a component it belongs to as well as by other components. For example, "Post-Pro" component uses services of "MED" component engine for importation of data files in med format.

6.1.5 Light-weight component

Light-weight component is a SALOME component without CORBA engine. If a component does not implement any services accessible via CORBA technology by other components, and if a component does not use CORBA-based services of SALOME KERNEL (implemented in SALOMEDS, NamingService, Container, and other packages), then such component should be built upon light-weight architecture.

Light-weight component consists of GUI module and functional packages which are accessible only from its GUI module and not accessible from within other components.

Light-weight components can work with SALOME built in both light and full configurations. In other words, a light-weight component can be a part of a multi-component SALOME application, and the other components do not have to be necessarily light-weight. But if all components are light-weight (in particular, if there is only 1 light-weight component in an application), then it is preferable to use KERNEL and GUI modules in light configuration. This will increase the application performance since a number of unused CORBA-based services will not be started.

Currently only C++ light-weight components are supported by SALOME platform. In future, it is planned to add support for Python light-weight components.

6.1.6 Numerical computations cycle

Software data processing which is usually performed in 3 phases:

1. Pre-processing phase: preparation of data, design of the mathematical model of a physical object or phenomenon.
2. Processing phase: numerical computations carried out by a special software (solver), application of algorithms to a previously developed mathematical model;
3. Post-processing phase: visual representation of computation results (graphs, colored shapes in 3D, etc.).

6.1.7 Python component

A component written in Python programming language. It may be either with CORBA engine or without.
Please, refer to “ATOMGEN: Python component” chapter of the tutorial for details.

6.1.8 SALOME data structure

SALOMEDS (SALOME data structure) is a library that provides support for a multi-component document of SALOME platform. Components can use SALOMEDS to publish their data inside a SALOMEDS document (Study object). Publishing the data in a common document gives the following advantages for a custom component:

- The data becomes available for other components (for processing, visualization, etc.), it can accessed using SALOMEDS tools and services.
- The data becomes automatically persistent (can be saved and restored), as persistence is already implemented in SALOMEDS library.

SALOMEDS also provides the mechanism of data persistence for components that do not publish their data in a common SALOMEDS data structure. This mechanism is described in Implementing persistence section of the tutorial. Briefly, SALOMEDS provides the following: a component saves its data in arbitiary format to an external file and returns the name of this file to SALOMEDS. SALOMEDS serializes this file into a binary stream and includes it into the common Study file on

save operation. When the data must be restored, exactly the same file is created by SALOMEDS for the component, and the component itself is responsible for loading it.

6.1.9 Study

Study represents a SALOME platform document that contains data of multiple components. The data is organized in a tree-like structure within the Study. SALOMEDS library supports persistence of Study.

6.2 GUI CONCEPTS

6.2.1 Data model

Data model is a manager of data within a component GUI. It plays a role of interface for accessing the data: retrieval, removal, and modification. It also implements persistence of data: saving to external file(s) and reconstruction of internal data structure from the file(s).

Data Model represents arbitrary internal data in a tree-like structure. It is done through `root()` method of `CAM_DataModel` class. It returns object of the highest level of component's data, usually this object represents the component itself. This objects (instance of Data Object class) has child objects, they also have child objects, and so forth.

Data model class is usually redefined by a component (inherits `LightApp_DataModel` or `SalomeApp_DataModel` classes).

6.2.2 Data object

It is a unitary piece of data within a component GUI. Its primary mission is to provide a common view to an arbitrary data. It is a proxi-object - it hides the real implementation of data and provides a generic interface to accessing it by other objects. For example, Object Browser "knows" how to display Data Objects, Selection Manager "knows" how to select Data Objects, and only Data Object itself "knows" which real piece of component's data was accessed (displayed, selected, etc.) through it.

Data object supports tree-like structure: it has a parent Data object (or null, if it is a root-level object), and arbitrary number of child objects.

Data object class is usually redefined by a component (inherits `LightApp_DataObject` or `SalomeApp_DataObject` classes).

6.2.3 Data owner

Data owner is an abstract representation of a piece of data. It is mainly used for selection management - Data owner represents a selected entity independently from the source of selection (Object Browser, 3D viewer, 2D chart). Setting and retrieval of selection always operates with the list of Data owners.

Data owner contains a unique string identifier called "entry" which is used for locating the real object in the component data structure.

Usually it is not necessary to declare a custom Data owner class in a component, the base implementation is in `LightApp_DataOwner` class.

6.2.4 Desktop

Desktop represents a main frame of a SALOME application. It contains a menu bar, tool bars, and central area for GUI controls of components: Object Browser, Python console, 3D/2D viewers, etc.

Base desktop class is SUIT_Desktop, it defines methods to access active window inside the desktop frame, managers of tool bars and main menu. STD package contains 3 successors of base Desktop class: `STD_SDIDesktop`, `STD_MDIDesktop`, and `STD_TabDesktop`. SDI and MDI desktop classes implement single and multiple document interfaces within the desktop frame. TabDesktop allows for "tabbing" of windows within the desktop frame. By default, if it is not overridden in a component GUI, SALOME applications use TabDesktop. Tabbed windows can be

split vertically or horizontally, it is possible to store positions of windows, how they are split, etc. and restore afterwards. Example of application with `STD_TabDesktop`:

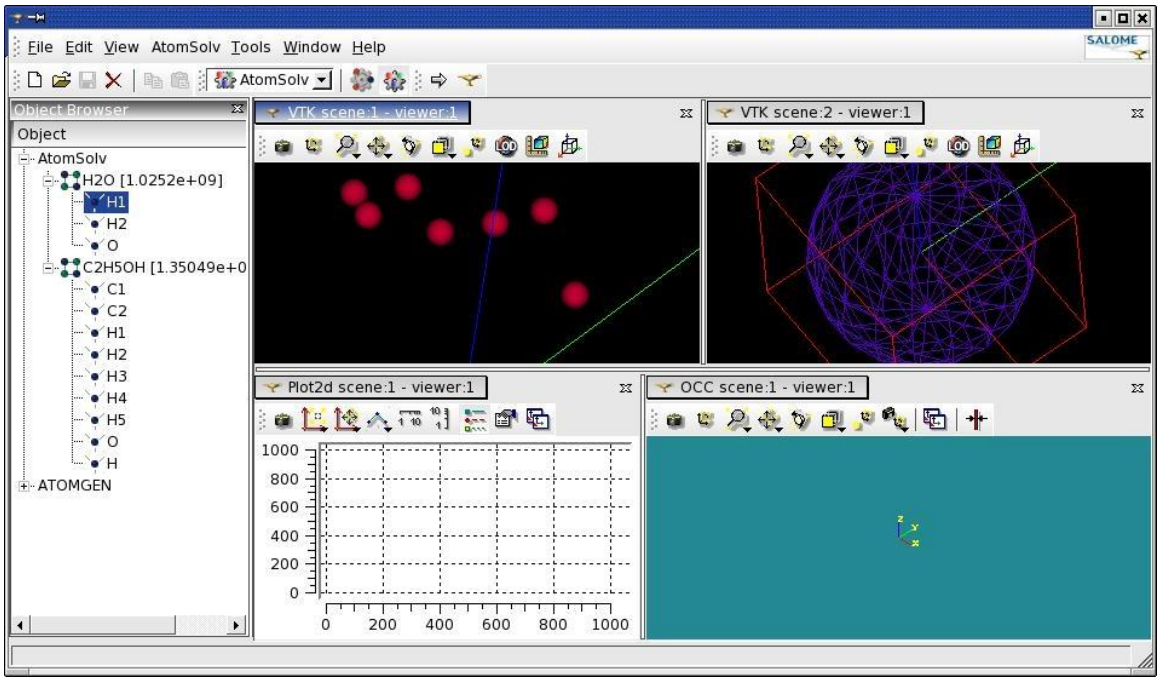

Figure 12. Tabbed desktop window

6.2.5 GUI module

A part of a component, which is responsible for graphical representation and behavior of a component. An approximate list of responsibilities of GUI module is:

- Creation of main menu items and toolbar buttons, connection them to the component's functions.
- Definition of types of shared GUI objects that the component will use (viewers, object browser, etc.).
- Filling popup menu on object selection event.
- Starting of common services as selection management, popup menu management, etc.

In general, a GUI module coordinates the behavior of a component especially when it relates to interaction with user.

GUI module class of C++ component must inherit `LightApp_Module` (for light architecture) or `SalomeApp_Module` class.

GUI module of Python component is implemented in a more complex way. Please, refer to GUI for Python component section of the tutorial for details.

6.2.6 Operation

Operation is a manager of an action inside a component GUI. By "action" we understand any functionality a GUI module of a component provides to a user. Examples of actions may be the following: creation of a sphere in Geometry component, Atom creation in ATOMIC component, graph execution in Supervisor component.

Using an Operation for action management gives the following advantages:

- Action can be canceled, suspended, and resumed during its execution.

- Operation instance can control which other Operations can be executed simultaneously with this Operation. It is implemented using method

```
bool SUIT_Operation::isValid(SUIT_Operation* theOtherOperation)
const.
```

Before starting a new operation (*operation_A*), an application calls isValid() method of the operation being executed (*operation_B*) passing it *operation_A* as a parameter. If *operation_B* returns false, then *operation_A* is not started, it must wait untill *operation_B* finishes its execution. This mechanism can be overridden, though, with yet another virtual method of SUIT_Operation class:

```
bool SUIT_Operation::isGranted() const.
```

If this method returns true, then the operation is started any way, ignoring isValid() return value.

- Operation has support for transaction mechanism.

For example, if user closes a study during execution of an operation, the Operation receives "Abort" signal (`abortOperation()` virtual function is called). The operation stops all algorithmical processing, closes the dialog windows it opened, aborts transaction, and frees all resources. Without using the Operation object it would be problematic to perform such smart deactivation of the action.

Base class for Operation object is `SUIT_Operation`, then it is inherited in `LightApp` package - `LightApp_Operation`. Custom operations of a component should inherit `LightApp_Operation` class.

6.2.7 Resource manager

It is a class that provides access to various resources at run-time. Values of integer, floating point, boolean, string and even complex (`QFont`, `QColor`) types can be set and retrieved from the Resource manager. Resource manager can be statically accessed from any place in the code using the following call:

```
SUIT_Session::session()->resourceMgr();
```

Between the sessions Resource manager stores resources in external files in XML or INI formats (XML by default). When SALOME application starts, Resouce manager locates these files and loads resources from them. The resouces files are:

- In light configuration of SALOME:

1. User resource file ~/.LightApprc.<version>, where <version> is the version number of SALOME (currently 3.2.0).

2. LightApp.xml files in directories listed in LightAppConfig environmental variable. For example, if LightAppConfig equals to

```
"/work/ATOMGEN_BUILD/share/salome/resources:/work/ATOMSOLV_BUILD/sha
re/salome/resources:/work/GUI_BUILD/share/salome/resources"
```

Resource manager will try to load 3 LightApp.xml files from the 3 listed directories.

- In full configuration of SALOME:

3. User resource file ~/.SalomeApprc.<version>, where <version> is the version number of SALOME (currently 3.2.0).
4. SalomeApp.xml files in directories listed in SalomeAppConfig environmental variable. For example, if SalomeAppConfig equals to

```
"/work/ATOMGEN_BUILD/share/salome/resources:/work/ATOMSOLV_BUILD/sha
re/salome/resources:/work/GUI_BUILD/share/salome/resources"
```

Resource manager will try to load 3 SalomeApp.xml files from the 3 listed directories.

Certain resources can be modified by user using Preferences dialog box (see Preferences section of the tutorial for details). Modified resources are written by the Resouce manager to the user resource file (.LightApprc.<version> or .SalomeApprc.<version> in the user home directory).

6.2.8 Selection management

Selection management in a SALOME application is handled by a class `LightApp_SelectionMgr` or its successors. Selection manager uses very "light" representation of data: Data Owner objects. Data Owner stores only an entry (unique string identifier, unique "key" of a piece of data) of a selected entity. Every window (view, dialog box, etc.) that displays data and supports selection of objects must implement a so called Selector class - a class that performs "conversion" of data internally used by this view to Data Owner and reverse. Also every window that supports selection must register itself as a selection source by the Selection Manager. Such registration is done for support of selection syncronization: entities that are selected in one window become selected in other windows.

The selection syncronization follows the next scheme:

- When a selection event happens in a window, it emits a signal. This signal is caught by the global Selection Manager.
- Selection Manager requests for selected objects from the Selector of the signal emitter.
- Selector creates a list of Data owners that correspond to the selected in its window entities.
- Selection Manager receives the list of Data owners; then it iterates the other registered Selectors and programmatically sets the selection in them passing the list of Data owners.
- Having received the list of Data owners, Selectors try to select the corresponding objects in their windows.

6.2.9 View manager

The name of this object shows its main purpose: it manages the views (View windows). It contains a View model as a member field for creation of a view and various methods for accessing the managed views (`getActiveView()`, `getViewsCount()`, etc.).

`SUIT_ViewManager` is a "gateway" class for working with view windows from application or another side. `STD_Application` class (parent class for `LightApp_Application` and `SalomeApp_Application`) stores view managers and it is possible to retrieve a view manager of a certain type using methods of the application.

6.2.10 View model

View model is a creator of View windows. "View model - View window" pair follows "Factory method" pattern - view windows are created by virtual method of view model `createView()`. Visualization packages of GUI module redefine `SUIT_ViewModel` class to be able to create custom view windows.

View models have a pair of methods:

```
static QString Type();
virtual QString getType() const;
```

These methods must return a type descriptor of a view ("OCCViewer", "VTKViewer", etc.). This type is used in context popup menu ("client" parameter will be equal to this type), and in many other places in the code where it is needed to determine the type of a view.

6.2.11 View window

View window is a frame, inherited from `QMainWindow` (Qt library), that contains the visualization scene. Objects are displayed inside a view window. The visualization packages inherit their custom view windows from `SUIT_`ViewWindow and fill it with custom widgets in order to display objects in a certain way. OCCViewer, for example, places a `V3d_View` (Open CASCADE library) inside its view window to display a 3D scene.

View window is able to save its contents as an image (`dumpView()` and `dumpViewToFormat()` virtual functions) and it is able to save and restore its parameters (values of zoom, pan, degree rotation, and other custom properties of the scene) - `get/setVisualParameters()` virtual functions. These functions are redefined in every custom view window to perform the corresponding functionality.

As view window is a basic frame, it receives the basic window events: mouse moves, clicks, keyboard presses, etc. One of the objectives of view window is to pass these events further - it is done through various signals emitted by `SUIT_ViewWindow` class: `mousePressed()`, `mouseReleased()`, `weeling()`, `keyPressed()`, etc.

7. ATTACHMENTS

File name	Description	Paragraph #
ATOMIC: light-weight component		
light-00.tar.gz	Empty stub for light-weight component development. Contains basic sources directory structure and stub for the GUI module class with no meaningful content.	2.2 3.1
FindGd.cmake	This file provides macro-procedure used to check presence and availability of **gd** library.	2.2
light-01.tar.gz	Light-weight component with implemented methods `initialize()`, `activateModule()`, `deactivateModule()`, and a few customized actions and menu items created.	3.1
light-02.tar.gz	Light-weight component with redefined data model class connected to GUI module. New data classes developed (molecules, atoms), data model class is almost empty.	3.2
light-03.tar.gz	Light-weight component with redefined data model and data object classes. Data model implements `build()` virtual method for creation of a tree-type structure of data objects.	3.2
light-04.tar.gz	Light-weight component with fully developed data model: it implements persistence of its data and builds tree of data objects.	3.3
light-05.tar.gz	Light-weight component with Object Browser displaying the custom data objects of the component.	3.4
light-06.tar.gz	Light-weight component with support of selection of elements in Object Browser. It is possible to analyze selection and retrieve entries of the selected objects.	3.5
light-07.tar.gz	Light-weight component with advanced popup menus management (use of `QtxPopupMgr`, logical rules for actions, redefinition of Selection class).	3.5.2
light-08.tar.gz	Light-weight component with redefined Operation class and 2 custom operations. Example of using operations: one operation uses a dialog box, another - does not.	3.6
light-09.tar.gz	Fully functional ATOMIC component with customized data model, data objects, persistent data structure, Object Browser capabilities, selection management, popup menus management, and use of operations.	3.6
light-10.tar.gz	Fully functional ATOMIC component with implemented dump python functionality.	3.7
ATOMGEN: Python component		

py-01.tar.gz	An empty component with one IDL file that contains definition of interface of ATOMGEN engine.	4.2
py-02.tar.gz	Python component with engine: interface is declared in IDL, implementation written in Python. No GUI is developed yet.	4.2
py-03.tar.gz	Python component with advanced data structure: persistence of internal data is implemented using SALOMEDS package of KERNEL.	4.3
py-04.tar.gz	Fully functional ATOMGEN component: engine implementation, persistent data structure and GUI written in Python.	4.4
py-05.tar.gz	Fully functional ATOMGEN component with implemented dump python functionality.	4.5
ATOMSOLV: C++ component with engine		
c-01.tar.gz	C++ component with implemented engine: interface is declared in IDL file, implementation code in C++.	5.1
c-02.tar.gz	C++ component with implemented engine and GUI module. Objects are displayed in Object Browser, but visual presentation in 3D viewer is not implemented yet.	5.2
c-03.tar.gz	C++ component with implemented engine and GUI module, atoms are visualized in 3D viewer (VTK viewer).	5.3
c-04.tar.gz	Fully functional ATOMSOLV component: engine is implemented, GUI with visualization of atoms in 3D, visualization parameters are taken from user defined preferences.	5.4