Developer Guide

Integration of new meshing algorithm as plug-in to SALOME Mesh module

Rédaction	Vérification	Approbation
J DOROVSKIKH	V SANDLER	R NEDELEC

Date : 17 Septembre 2010

Avant-propos:

This document is a developer guide that provides a details description of custom meshing plug-in integration procedure.

SOMMAIRE

1. Introduction

This document describes how to add custom meshing tools to the SALOME application.

SALOME Mesh module provides a plug-in mechanism that allows using custom algorithms for mesh computation. Each meshing plug-in is a set of algorithms and (optionally) hypotheses.

Each algorithm is intended to create a set of mesh elements from initial data:

- Given geometry (`TopoDS_Shape`) to be meshed. Hereafter in this document a geometric object is referred as "shape". For 3D algorithms an input can be also an imported 2D closed mesh shell, and in this case no input shape is present.
- Results of other algorithms work (existing mesh elements). Usually (but not always) algorithm of higher dimension uses mesh elements and nodes, generated by algorithms of lower dimensions. For example, `Quadrangle_2D` algorithm builds faces grid, basing on edges discretization, done by some 1D algorithm.
- Meshing parameters, defined via hypotheses, compatible with the algorithm.

This document is a developer guide that provides a details description of new meshing plug-in integration procedure, step by step.

2. Integration of meshing plug-in

2.1 Create your plug-in module

First of all choose a name for your plug-in. Here after in this document we will mention it as <MyPluginName>.

Create on disk a directory structure like for usual SALOME module. In the `src` directory you will create directories for server and, optionally, client libraries of your plug-in. For example, see structure of `NETGENPLUGIN_SRC`.

- Server library provides your algorithms and hypotheses implementation (including corresponding CORBA interfaces).
- Client library provides implementation of graphical user interface for your hypotheses creation.

2.2 Implement server plug-in library

Server library is the main part of your plug-in. Here you have to implement the way your algorithms will generate mesh and the data your hypotheses will keep.

2.2.1 Implement functionality of the algorithms

Inherit your classes of algorithms from SMESH ones, for example:

```
class NETGENPlugin_NETGEN_3D: public SMESH_3D_Algo
```

- 3D algorithm generating volume elements (like, for example, hexahedron) should be inherited from `SMESH_3D_Algo`.
- 2D algorithm generating surface elements (triangle, quadrangles and/or polygons) should be inherited from `SMESH_2D_Algo`.
- 1D algorithms generating segments should be inherited from `SMESH_1D_Algo`.

See examples in:
`SMESH_SRC/src/StdMeshers/StdMeshers_*.*`
`NETGENPLUGIN_SRC/src/NETGENPlugin/NETGENPlugin_*.*`

2.2.1.1 Define algorithm features

First, it is necessary to specify features of your algorithm that define how its `Compute()` method, which is to generate mesh, is called: in what turn, with what input etc. These features are defined as member fields of `SMESH_Algo` and its base classes. These features have to be initialized or their default values can be modified in a constructor of your algorithm class. The following features can be specified:

`_name`	This is an algorithm's type name. It is used as an identifier of the algorithm class internally within SALOME. For example, this name is passed to the SMESH engine when it is requested to create new instance of the algorithm. It must be initialized in the algorithm constructor.
`_compatibleHypothesis`	Array of names of hypotheses types that can be used to define meshing parameters of this algorithm. This array must to be filled - if the algorithm has any parameter - in the algorithm constructor.
`_onlyUnaryInput`	A Boolean flag indicating if the algorithm accepts only one or several shapes as input. If several shapes are acceptable, all the sub-shapes that should be computed with the same parameters are passed to `Compute()` as a compound shape (of type `TopAbs_COMPOUND`). The default value of this flag is `true`, but it can be redefined in the constructor if necessary.
`_requireDiscreteBoundary`	A Boolean flag indicating if the algorithm requires low dimension mesh as input or not. `NETGEN 1D-2D` algorithm can be referenced as a good example of algorithm whose `_requireDiscreteBoundary == false` because it itself can generate 1D mesh and thus it does not need a pre-existing 1D mesh. On the other hand, `MEFISTO` algorithm is an example of the algorithm that specifies `_requireDiscreteBoundary == true` (i.e. with its default value) because it can generate 2D mesh only and thus it requires pre-existing 1D mesh that should be generated by other algorithm fist. This flag's default value is `true`; it can be redefined in the constructor if necessary.
`_requireShape`	A Boolean flag indicating if the 3D algorithm can work without an input shape, i.e. is it able to generate a 3D mesh on an input 2D mesh shell or not. Default `true` value can be redefined in the constructor if necessary.

`_supportSubmeshes`	A Boolean flag indicating if the algorithm, that does not require a lower dimension mesh as input (i.e. its `_requireDiscreteBoundary == false`), can use exiting elements of lower dimension for mesh generation. For example `NETGEN 1D-2D` algorithm uses 1D mesh built by other algorithm, for that its `_supportSubmeshes == true`. And, contrary, `Body fitting` algorithm can't use neither 1D nor 2D pre-existing mesh, and thus its `_supportSubmeshes == false`. This flag's default `false` value can be redefined in the constructor if necessary.

Only `_name` attribute is mandatory to be defined. If other attributes are left default, then an algorithm has following features:

- It has no parameters (as `_compatibleHypothesis` is empty);
- It accepts only one input shape (as `_onlyUnaryInput == true`);
- It requires that boundary of the input shape is meshed by other algorithm (as `_requireDiscreteBoundary == true`);
- It can't work on the mesh shell, without the input shape (`_requireShape == true`);
- It uses input mesh of lower dimension (`_supportSubmeshes == true`).

2.2.1.2 Implement needed methods

Next step consists in implementing of the meshing algorithm. The methods of the algorithm to be implemented are described in this chapter.

2.2.1.2.1 CheckHypothesis()

```
bool CheckHypothesis(SMESH_Mesh&                          aMesh,
                     const TopoDS_Shape&                  aShape,
                     SMESH_Hypothesis::Hypothesis_Status& aStatus);
```

This method is called by the SMESH engine in two cases.

- If a hypothesis or algorithm is assigned/removed to/from `aShape`, in order to find out if all needed hypotheses are assigned and thus it's possible to call `Compute()` method.
- Just before calling `Compute()` in order to let algorithm find out meshing parameters to be used for meshing `aShape`.

This method should check if needed hypotheses are assigned to the shape passed as argument (all hypotheses assigned to a shape are returned by `GetUsedHypothesis()` method) and return `true` or `false` depending on a result of check. In addition it should return a corresponding value via a `Hypothesis_Status` out argument.

If an invalid value of parameter can't be rejected in a method of hypothesis setting this parameter (for example if parameter's validity depends on a shape the hypothesis is assigned to), then parameter's validity should be checked by `CheckHypothesis()`.

The following `Hypothesis_Status`'es can be returned:

- `HYP_OK` – everything is OK;
- `HYP_MISSING` – required hypothesis is missing;

- HYP_ALREADY_EXIST – several suitable hypotheses are assigned and it's not clear which one to use for meshing;
- HYP_BAD_PARAMETER – hypothesis has an invalid parameter value;
- HYP_NOTCONFORM – non-conform mesh would be produced if the assigned hypothesis is used for meshing.

2.2.1.2.2 Compute()

```
bool Compute( SMESH_Mesh& theMesh, const TopoDS_Shape& theShape);
bool Compute( SMESH_Mesh& theMesh, SMESH_MesherHelper& theHelper);
```

This method is called when the user invokes "Compute mesh" command.

This method should implement mesh generation. `Compute()` method has two signatures, the first one computes mesh on `theShape`, the second one computes 3D mesh on `theMesh`, which is in this case a 2D mesh shell. The second `Compute()` can be optionally implemented in 3D algorithms. `theHelper` argument of the second `Compute()` must be obligatory used for addition of nodes and elements to `theMesh`.

While implementing the meshing algorithm, the following important aspects should be respected:

- During mesh generation, the algorithm should periodically check value of its `_computeCanceled` attribute. If its value is `true`, it means that the user cancelled the computation and the algorithm should stop operation.
- The algorithm should take into account *order* of the input mesh. Depending on this order it should generate either liner or quadratic elements. To facilitate this task `SMESH_MesherHelper` class, described below in this document, is intended.
- All elements and nodes added to the mesh by your algorithm must be assigned to the shape being meshed. This is needed to remove theses nodes and elements from the mesh when meshing parameters of your algorithm are modified by the user. In addition to this, 1D algorithm must specify a parameter (U) of node on an edge and 2D algorithm must specify parameters (U, V) of node on a face. To do this it is recommended to use methods of `SMESH_MesherHelper` class described below.
- In case if some error occurs, the algorithm should report a failure reason via one of error() methods (see `SMESH_Algo` base class).
- If the input mesh does not meet requirements of the algorithm, invalid elements and/or nodes can be reported via `addBadInputElement()` method.

Note that `SMESH_Algo` class defines several static methods that can be useful for implementing your meshing algorithm. Refer to the `SMESH_Algo` class for more details. As well, there are several helper classes that can be useful at algorithm implementation, namely `SMESH_MesherHelper`, `SMESH_File`, `SMESH_Block`, `StdMeshers_FaceSide`.

2.2.1.2.2.1 SMESH_MesherHelper

`SMESH_MesherHelper` class is a tool mostly intended for creation of either linear or quadratic elements depending on the order of input mesh. It also provides information on particularity of the shape (its periodicity, presence of seam edges and degenerated edges) and some other methods useful while mesh generation.

The following code shows how `SMESH_MesherHelper` can be used in a 2D algorithm.

```
// create a helper
SMESH_MesherHelper helper( theMesh );
// analyze order of input mesh
_quadraticMesh = helper.IsQuadraticSubMesh( theShape );
// make helper assign new elements and nodes to theShape
helper.SetElementsOnShape( true );
...
// UV position of a new node on a geometrical face is stored
SMDS_MeshNode* newNode1 = helper.AddNode( x,y,z, /*ID=*/0, U,V );
...
// TRI3 or TRI6 element is created depending on this->_quadraticMesh
SMDS_MeshFace* tri = helper.AddFace( newNode1, newNode2, newNode3 );
```

2.2.1.2.2.2 SMESH_File

`SMESH_File` is a cross-platform interface for effective reading of files.

2.2.1.2.2.3 SMESH_Block

`SMESH_Block` is a tool working with the block shape. It provides

- Access to topology of the block.
- Computation of 3D coordinates of a point by its normalized parameters.
- Computation of normalized parameters of a given 3D point.

2.2.1.2.2.4 StdMeshers_FaceSide

`StdMeshers_FaceSide` is a tool sorting nodes located on a boundary of geometrical face and providing information about these nodes useful for 2D meshing.

2.2.1.2.3 Evaluate()

```
bool Evaluate(SMESH_Mesh &         theMesh,
              const TopoDS_Shape & theShape,
              MapShapeNbElems&     theResMap);
```

This method is called when the user invokes “Evaluate” command on a mesh, which is useful to estimate a final amount of mesh elements in the mesh.

This method should roughly estimate number of elements and nodes that will be generated by `Compute()` method on `theShape`. Note that this method should work quickly, since the main goal of evaluation operation is to allow the user previewing of approximate number of mesh nodes/elements that might be produced during the actual mesh computation.

2.2.1.2.4 SetEventListener()

```
void SetEventListener(SMESH_subMesh* subMesh);
```

This method is needed only for algorithms like projection algorithms whose work depends on a mesh generated on another sub-shape (not a lower dimension sub-shape of the sub-shape being meshed).

This method gives algorithm the possibility to setup an object of type `SMESH_subMeshEventListener` to some sub-mesh. This allows keeping the mesh up-to-date at modification of hypotheses in the case where one sub-mesh depends on another but not hierarchically (topological hierarchy is meant here).

For example, a sub-mesh computed by projection algorithm (target sub-mesh) depends on a sub-mesh from which mesh was projected (source sub-mesh) but these sub-meshes are of the same hierarchical level, so that if the source sub-mesh is cleared due to some reason (for example due

to modification of meshing parameters) the target sub-mesh normally is not cleared and the mesh becomes out-of-date.

Setting `SMESH_subMeshEventListener` on the source sub-mesh allow transferring `SMESH_subMesh::CLEAN` event from the source sub-mesh to the target sub-mesh so that the both sub-meshes are cleared together and the mesh remains up-to-date.

The following code shows how to achieve this in `SetEventListener()` method of projection algorithm.

```
// Create an event listener that will transfer important events from the
// source sub-mesh to the target one. The first arg of its constructor
// means that SMESH_subMesh is responsible for deleting this listener.
// The second arg is just an ID allowing to retrieve the listener and
// its data (SMESH_subMeshEventListenerData) from the sub-mesh storing
// them.

SMESH_subMeshEventListener* listener =
  new SMESH_subMeshEventListener(true,“ProjAlgo::SetEventListener”);

// Create a "listener data" object storing the target sub-mesh where the
// listener will transfer events to.
// subMesh here is the target sub-mesh where ProjAlgo is assigned to.

SMESH_subMeshEventListenerData* data =
  SMESH_subMeshEventListenerData::MakeData( subMesh );

// Set the event listener to the source sub-mesh.

subMesh->SetEventListener( listener, data, sourceSubMesh);
```

2.2.1.2.5 SubmeshRestored()

```
void SubmeshRestored(SMESH_subMesh* subMesh);
```

This method lets the algorithm restore needed event listeners after restoring the Study from a file.

If we continue considering the example with some projection algorithm, then in this method the algorithm should find the source sub-mesh and to set event listeners in the same way as in `SetEventListener()`.

2.2.2 Implement functionality of the hypotheses

For hypothesis implementation you have to set up the following attributes in the constructor:

`_name`	Hypothesis type name.
`_param_algo_dim`	An integer value that holds a dimension of mesh elements, which will be generated by your algorithm (that will use this hypothesis). For the optional hypothesis, the value of this attribute must be negative, e.g. -2 means “optional 2D hypothesis”.

While implementing methods that set up the values of meshing parameters, note that such methods have to call `NotifySubMeshesHypothesisModification()` method in case if a parameter value changes (like the following code, which is typical one, does):

```
void StdMeshers_NumberOfSegments::SetNumberOfSegments(int segmentNumber)
throw(SALOME_Exception)
{
  if ( segmentNumber <= 0 )
    throw SALOME_Exception
      (LOCALIZED("number of segments must be positive"));

  if ( segmentNumber != _numberOfSegments ) {
    _numberOfSegments = segmentNumber;
    NotifySubMeshesHypothesisModification();
  }
}
```

See examples in:
`SMESH_SRC/src/StdMeshers/StdMeshers_*.*`
`NETGENPLUGIN_SRC/src/NETGENPlugin/NETGENPlugin_*.*`

The virtual methods described in the following paragraphs should be implemented for your hypothesis class.

2.2.2.1.1 SetParametersByMesh()

```
bool SetParametersByMesh(const SMESH_Mesh*   theMesh,
                         const TopoDS_Shape& theShape);
```

This method is called when a local hypothesis is created and `theMesh` on `theShape` is already computed but using a hypothesis of other type. This method should, if possible, set values of parameters according to mesh existing on `theShape`, that allows the user to attune the parameter without a need to guess its current value. For example if an edge was meshed using “Local Length” hypothesis then a being created "Nb. Segments" hypothesis can find out number of segments by counting elements assigned to `theShape`, which is an edge in this case.

2.2.2.1.2 SetParametersByDefaults()

```
bool SetParametersByDefaults(const TDefaults& dflts,
                             const SMESH_Mesh* theMesh=0);
```

This method allows initializing parameter values of your hypothesis being created according to preferred (defined in “Preferences” dialog) number of segments per edge (`TDefaults::_nbSegments`) and preferred segment length depending on shape size (`TDefaults::_elemLength`).

This method should, if possible, set parameter values according to default values of number of segments (`_nbSegments`) and of segment length (`_elemLength`) held by `TDefaults` structure. For example a hypothesis defining maximal area of surface element could define its parameter value as `_elemLength * _elemLength`.

To help the hypothesis in evaluating parameter values, `TDefaults` structure also may hold a shape to which the hypothesis is assigned.

2.2.2.1.3 SaveTo() and LoadFrom()

```
std::ostream & SaveTo(std::ostream & save);
std::istream & LoadFrom(std::istream & load);
```

These methods are called when a Study is saved and restored. These methods should be implemented if a hypothesis have any parameter to store (there are also hypotheses that affect algorithm behavior by only their presence, without need in any parameter).

Values of parameters should be stored as text in a steam. Pay attention to storing string parameters that may include white spaces (name of groups for example). In case of presence of white spaces in a string it won't be possible to retrieve the whole string from the stream using >> `operator`. One of possible solutions is to store each byte of a string as an integer (as an example of implementation see `resultGroupsToIntVec()` method in `SMESH_SRC/src/StdMeshers/StdMeshers_ImportSource.cxx`).

Note that it's not possible to store a parameter of type `TopoDS_Shape` as an index of a sub-shape in the main shape (shape to be meshed). This is because the hypotheses exist independently of any mesh that holds the shape to be meshed and thus the hypothesis has no access to any mesh when `LoadFrom()` is called. A solution is to implement the discussed methods and to store a study entry of a referred shape at the CORBA interface implementation of hypothesis. Then `LoadFrom()` method will be able to restore a shape by its entry and to pass it to the core implementation of hypothesis.

2.2.3 Define CORBA interfaces for your hypotheses and algorithms

Create an IDL file in the `idl` directory of your module folders structure. Define there your algorithms and hypotheses interfaces. Inherit corresponding SMESH interfaces, for example:

```
interface NETGENPlugin_NETGEN_3D : SMESH::SMESH_3D_Algo
{
};
Interface NETGENPlugin_Hypotheis : SMESH::SMESH_Hypothesis
{
...
```

Algorithm interfaces usually do not require any methods to be defined.

The interface of hypothesis should define API methods for setting up and retrieving values of parameters. These methods will be called from your client (GUI) library or from Python scripts.

See examples in:
`SMESH_SRC/idl/SMESH_BasicHypothesis.idl`

It is recommended to respect the following two rules when designing IDL API of your hypothesis:

- Each method setting up a parameter value should be dedicated to only this parameter, without any Boolean or textual switch specifying which parameter is set by this call. Respecting this rule allows Python Dump functionality to convert several commands that together define all parameters, into one command setting all parameters at once via the Python method wrapping creation of your hypothesis.

- If a method sets up several values at once, i.e. it has several arguments, only the first one must be of type that can be set via SALOME notebook variable (`double, long or short`).

Hereafter is an example of a deprecated (first one) and recommended (second and third ones) style of interface methods.

```
    /*!
     * Sets <start segment length> or <end segment length> parameter
     * value.
     * (This is a deprecated style).
     */
    void SetLength(in double length, in boolean isStartLength);
```

```
    /*!
     * Sets <start segment length> parameter value
     */
    void SetStartLength(in double length);
    /*!
     * Sets <end segment length> parameter value
     */
    void SetEndLength(in double length);
```

And a Python dump of calls of the last two methods will possibly look like following:

```
StartAndEndLength = RegularAlgo.StartAndEndLength( 1.5, 10.1 )
```

2.2.4 Implement CORBA interfaces

Inherit your classes, implementing declared IDL interfaces, from the corresponding classes of `SMESH_I` package and from your CORBA interfaces, for example:

```
class NETGENPlugin_NETGEN_3D_i:
  public virtual POA_NETGENPlugin::NETGENPlugin_NETGEN_3D,
  public virtual SMESH_3D_Algo_i
```

Here `POA_NETGENPlugin::NETGENPlugin_NETGEN_3D` is a name of class generated by compiler from CORBA interface definition of `NETGENPlugin_NETGEN_3D` algorithm.

To learn signatures of the C++ methods you are to implement your IDL file, compile it and check a header file generated by compiler from your IDL file. The file is located in `idl` folder of your build directory and its name coincides with the name of your IDL file plus ".hh" extension.

Include your IDL files. For example, if you have described your interface in file `NETGENPlugin_Algorithm.idl`, put the following two lines at the beginning of your header file:

```
#include <SALOMEconfig.h>
#include CORBA_SERVER_HEADER(NETGENPlugin_Algorithm)
```

The common header file `SALOMEconfig.h` provides macro `CORBA_SERVER_HEADER` that is used to include IDL interfaces to the C++ code.

See examples in:

```
SMESH_SRC/src/StdMeshers_I/StdMeshers_*_i.*
NETGENPLUGIN_SRC/src/NETGENPlugin/NETGENPlugin_*_i.*
```

2.2.5 Implement factory function

The goal of the factory function is to create hypotheses and algorithms instances by request from SMESH module.

```
extern "C"
{
  GenericHypothesisCreator_i* GetHypothesisCreator(const char* aHypType)
  {
    // Hypotheses
    if (strcmp(aHypName, "LocalLength") == 0)
      aCreator = new HypothesisCreator_i<StdMeshers_LocalLength_i>;
    else if (strcmp(aHypName, "NumberOfSegments") == 0)
    …
```

```
    // Algorithms
    else if (strcmp(aHypName, "Regular_1D") == 0)
      aCreator = new HypothesisCreator_i<StdMeshers_Regular_1D_i>;
    else if (strcmp(aHypName, "MEFISTO_2D") == 0)
    …
    return aCreator;
  }
}
```

Note: when this method is called, parameter `aHypType` is initialized by value of `<type>` attribute from your plug-in XML file (see chapter 2.5 for plug-in XML file description) and this is also a value of attribute `_name` of your algorithm/hypothesis class.

See an example in:
`SMESH_SRC/src/StdMeshers_I/StdMeshers_i.cxx`
`NETGENPLUGIN_SRC/src/NETGENPlugin/NETGENPlugin_i.cxx`

2.2.6 Write Makefile.am

Imagine you want to call your server library `MyServerLib.` Then you have to specify variable `lib_LTLIBRARIES` as `libMyServerLib.la`. You can choose any name for your server library, just specify it correctly in your plug-in XML file as `libMyServerLib.so` (how to do it will be described on page 15).

If implementation of your server library is separated into several packages, or you have other reasons to make some of your header files visible outside concrete package (for example, you want to use them in some other module implementation), do not forget to list them in `salomeinclude_HEADERS` section.

List your source files in section `dist_libMyServerLib_la_SOURCES`.

Specify required compilation and linkage flags using `libMyServerLib_la_CPPFLAGS` and `libMyServerLib_la_LDFLAGS` as.

See an examples in:
`SMESH_SRC/src/StdMeshers_I/Makefile.am`
`NETGENPLUGIN_SRC/src/NETGENPlugin/Makefile.am`

2.3 Implement client (GUI) plugin library

This step is required only if your hypotheses/algorithms need specific GUI for their construction.

It is usually required only for hypotheses, which provide some parameters for their construction. Algorithms are usually created without any specific parameters.

Note, that hypothesis can be also created without any parameters; in such a case algorithm behavior depends just on the hypothesis presence/absence.

2.3.1 Implement the required GUI

GUI consists of a set of the dialog boxes which are used to enter hypotheses parameters by the user.

See an examples in:
`SMESH_SRC/src/StdMeshersGUI/StdMeshersGUI_*Creator.*`
`NETGENPLUGIN_SRC/src/GUI/NETGENPluginGUI_*Creator.*`

In order to create your own dialog box:

- If parameters of your hypothesis have one of the following simple types: integer, double or string, you can inherit your `Creator` class from `StdMeshersGUI_StdHypothesisCreator` and redefine the methods `storeParams()`, `stdParams()`, `attuneStdWidget()`, `hypTypeName()`.

- In the other case you should inherit your `Creator` class from `SMESHGUI_GenericHypothesisCreator` and redefine methods `buildFrame()`, `storeParams()`, `retrieveParams()`

Example:
`SMESH_SRC/src/StdMeshersGUI/StdMesherGUI_NbSegmentsCreator.*`

All data from your plug-in XML file (described later in chapter 2.5) is accessible in your GUI via `HypothesisData` class:

```
HypothesisData* data = SMESH::GetHypothesisData( aHypType );
```

See `HypothesisData` class definition details at:
`SMESH_SRC/src/SMESHGUI/SMESHGUI_Hypotheses.h`

2.3.2 Provide graphical and textual resources for GUI

Optionally you can implement the GUI resource files `<MyResourceKey>_images.ts` and `<MyResourceKey>_msg_en.ts`. These files are the part of the Qt internationalization system used in SALOME.

See an example in:
`SMESH_SRC/src/StdMeshersGUI/StdMeshers_*.ts`
`NETGENPLUGIN_SRC/src/GUI/NETGENPlugin_*.ts`

Note:

- `ICON_SMESH_TREE_HYPO_<MyHypType1>` specifies an ID of the icon for the Object Browser for the hypothesis `<MyHypType1>`.

- `ICON_SMESH_TREE_ALGO_<MyAlgType1>` specifies an ID of the icon for the Object Browser for the algorithm `<MyAlgType1>`.

See the chapter 2.5 for more details about meaning of the `MyResourceKey`, `MyHypType1`, `MyAlgType1`.

2.3.3 Implement Hypothesis Creator and factory function

Below is a typical code of the factory function that creates and export new instance of the hypothesis creator class. This method is automatically invoked from SALOME.

```
extern "C"
{
  SMESHGUI_GenericHypothesisCreator*
      GetHypothesisCreator( const QString& aHypType )
  {
    if( aHypType=="NumberOfSegments" )
```

```
      return new StdMeshersGUI_NbSegmentsCreator();
    else
      return new StdMeshersGUI_StdHypothesisCreator( aHypType );
  }
}
```

Here aHypType parameter is used to pass the value of the <type> attribute in your plug0in XML file (see 2.5).

See an example in:
SMESH_SRC/src/StdMeshersGUI/StdMeshersGUI.cxx
SMESH_SRC/src/StdMeshersGUI/StdMeshersGUI_StdHypothesisCreator
SMESH_SRC/src/StdMeshersGUI/StdMeshersGUI_NbSegmentsCreator

2.3.4 Write Makefile.am

Let you want to call your client library MyClientLib, then set variable lib_LTLIBRARIES value to libMyClientLib.la. This is the default name of the library. You can choose any name for your client library; then specify it correctly in your plug-in XML file as libMyClientLib.so.

If you want to export some of your header files (e.g. you want to use them in some other module implementation), do not forget to list them in the salomeinclude_HEADERS section.

List your source files in the section dist_libMyClientLib_la_SOURCES.

Define MOC_FILES variable as a list of files, which will be generated automatically for dialog boxes and widgets using Qt moc compiler (meta data for GUI classes, refer to Qt documentation for more details).

Set nodist_libMyClientLib_la_SOURCES variable value to the $(MOC_FILES) since these files must not be included in a distribution.

Also set nodist_salomeres_DATA to the list of *.qm files; these files will be automatically generated from the corresponding resource *.ts files
.
Specify libMyClientLib_la_CPPFLAGS and libMyClientLib_la_LDFLAGS as required compilation and linkage flags.

See an example in:
SMESH_SRC/src/StdMeshersGUI/Makefile.am
NETGENPLUGIN_SRC/src/GUI/Makefile.am

2.4 Create icons for the Object browser

Icons are image files which are used for the displaying of the hypotheses and algorithms in SALOME Object Browser. If your hypotheses/algorithms do not need specific GUI, but you want to provide icons for object browser, see 2.3.2 chapter.

2.5 Describe your plug-in in dedicated XML resource files

2.5.1 Plug-in services description

You should create an XML file named <MyPluginName>.xml which should describe all the algorithms and hypotheses, implemented by your plug-in package. This description if used:

- By SMESH GUI while defining the mesh or sub-mesh
    - To create your algorithms and hypotheses;
    - To find hypotheses suitable for each algorithm;
    - To know compatible algorithms, etc.

- By SMESH engine
    - To learn how manage specific Python wrappings for algorithms (for generation of Python dump, for example).

See sample of such a file below:

```
<meshers-group name="MyName"
               resources="MyResourceKey"
               server-lib="libMyServerLib.so"
               gui-lib="libMyClientLib.so">
  <hypotheses>
    <hypothesis type="MyHypType1"
                label-id="My beautiful 1D hypothesis"
                icon-id="my_hypo_1_icon.png"
                dim="1"/>
    <hypothesis type="MyHypType2"
                label-id="My beautiful 3D hypothesis"
                icon-id="my_hypo_2_icon.png"
                dim="3"
                need-geom="false"
                auxiliary="true"/>
  </hypotheses>
  <algorithms>
    <algorithm type="MyAlgType1"
               label-id="My beautiful 1D algorithm"
               icon-id="my_algo_1_icon.png"/>
               hypos="MyHypType1,…"
               opt-hypos="MyHypType5,…"
               input="VERTEX"
               output="EDGE"
               dim="1"/>
    <algorithm type="MyAlgType2"
               label-id="My beautiful 3D algorithm"
               icon-id="my_algo_2_icon.png"
               input="QUAD"
                need-geom=”false”
               support-submeshes="true"
               dim="3"/>
  </algorithms>
</meshers-group>
<hypotheses-set-group>
  <hypotheses-set name="Automatic Tetrahedralization"
                  hypos="MaxLength"
                  algos="Regular_1D, MEFISTO_2D, NETGEN_3D"/>
  <hypotheses-set name="Automatic Hexahedralization"
                  hypos="NumberOfSegments"
                  algos="Regular_1D, Quadrangle_2D, Hexa_3D"/>
</hypotheses-set-group>
```

See an example in:

```
SMESH_SRC/resources/StdMeshers.xml
NETGENPLUGIN_SRC/resources/NETGENPlugin.xml
```

Attributes of the <meshers-group> tag:

- Value of the <name> attribute is used to collect hypotheses/algorithms in groups, when they are displayed in the algorithm/hypothesis creation dialog box in GUI. You can also use this attribute for short description of your meshing plug-n (implementing your GUI).
- Value of the <resources> attribute ("MyResourceKey" in above example) is used to access resources (messages and icons) from the GUI (see chapter 2.3.2). It should coincide with the name of plug-in.
- Value of the <server-lib> attribute is a name of your meshing plug-in's server library (see chapter 2.2.6).
- Value of the <gui-lib> attribute is a name of your meshing plug-in's client library (see chapter 2.3.4).

Attributes of the <hypothesis/algorithm> tag:

- Value of the <type> attribute is a unique name of the type of hypothesis/algorithm.
  - It is a value of the _name field of your hypothesis class (see chapter 2.2.1, implementation of the constructor of StdMeshers_LocalLength class: _name = "LocalLength").
  - It is a key to each certain hypothesis class (see chapter 2.2.5, implementation of GetHypothesisCreator() method in the StdMeshers_i.cxx).
  - It is a key to each certain hypothesis GUI (see chapter 2.3.1, for example implementation of StdMeshersGUI_StdHypothesisCreator class, usage of method hypType()).
  - It is a key to algorithm/hypothesis icon in the Object Browser (see chapter 2.3.2).
- Value of the <label-id> attribute is displayed in the GUI in the list of available hypotheses/algorithms ("Create Hypothesis/Algorithm" dialog).
- Value of the <icon-id> attribute is a name of the icon file, which is displayed in the GUI in the list of available hypotheses/algorithms ("Create Hypothesis/Algorithm" dialog).
- Value of the <dim> attribute means algorithm/hypothesis dimension and is used in GUI as algorithms/hypotheses of different dimensions are created separately.
- If <need-geom> attribute is set to "false", an algorithm can be used for creating mesh without underlying geometry.

Specific attributes of the <hypothesis> tag:

- If <auxiliary> attribute is set to "true", a hypothesis can be used in addition to an assigned "main" hypothesis (such auxiliary hypothesis can be selected in "Create Mesh" dialog in a special area).

Specific attributes of <algorithm> tag:

- Value of the <hypos> attribute is a list of types of hypotheses (type of hypothesis is defined by <type> attribute of <hypothesis>), usable by this algorithm.
- Value of the <opt-hypos> attribute is a list of types of optional hypotheses, usable by this algorithm (such as "Propagation", "QuadranglePreference", etc.)
- Value of the <input> attribute means types of mesh elements of a lower dimension, allowed in input mesh for this algorithm.
- Value of the <output> attribute means type of mesh elements, generated by this algorithm.
- If <support-submeshes> attribute is set to "true", this means that the algorithm building all-dim elements supports sub-meshes.

Attributes of the <hypotheses-set> tag:

- Value of the <name> attribute is a name of set.
- Value of the <hypos> attribute is a list of types of hypotheses, which will be created automatically, if user selects this set.
- Value of the <algos> attribute is a list of types of algorithms, which will be created automatically, if user selects this set.

Note: all attributes values from this XML file will be accessible in the GUI via HypothesisData and HypothesesSet classes (see 2.3.1).

2.5.2 GUI resources description

Another important XML resource file to be created is SalomeApp.xml. This file is automatically parsed by SALOME GUI; its goal is to specify:

- Path to the GUI resources: internationalization files and icons.
- Path to the user documentation (optionally) and its position in the *Help* menu (by default references to the documentation files are added as child items in the *Help* → *Mesh module* sub-menu).
- Specify default values of any applicable preferences (optionally). The preferences can be accessed (set/get) via centralized SALOME resource manager. Note, however, that for the current moment mechanism of exporting the preferences for meshing plug-ins to the general Preferences dialog box is not implemented. Though, the preferences can be used internally by the plug-in library if necessary.

The typical contents of SalomeApp.xml file is the following:

```
<document>
 <section name="resources">
  <parameter name="MyPluginName"
value="${MyPluginName_ROOT_DIR}/share/salome/resources/mypluginname"/>
 <section name="smesh_help">
  <parameter name="Plug-ins/My Plugin User's Guide"
value="${MyPluginName_ROOT_DIR}/share/doc/salome/gui/MyPluginName/index.html"/>
 </section>
</document>
```

- Parameter <MyPluginName> of <resources> section specifies path to the resources directory.

- The documentation for the plug-in can be listed in optional <smesh_help> section. You can specify as many documents in this section as you need – all of them will be listed in the *Help → Mesh module* sub-menu of the application. Here, <name> attribute of each parameter, listed in <smesh_help> section, specifies the document's title and, optionally, its position in the *Help → Mesh module* sub-menu (the above example adds menu item *Help → Mesh module → Plug-ins → My Plugin User's Guide*). Attribute <value> specifies absolute path to the documentation file; you can use any environment variables – the usual way is specify the relative sub-directory of the module's root directory, as demonstrated in above example.

2.6 Implement Python API

This step is optional.

2.6.1 Python dump

Creation of your algorithm and hypotheses is dumped into Python script automatically by SALOME, you only need to add some C++ statements in methods of your hypotheses implementing IDL API (discussed in chapter 2.2.4) that set parameter values.

Include a needed header in your MyPluginName_Hypothesis_i.cxx

```
#include <SMESH_PythonDump.hxx>
```

Add code creating python commands. Fox example (note that all sample codes in the section 2.6 are coherent and comments within code are meaningful):

```
void MyPluginName_Hypothesis_i::SetParams(CORBA::Long           nb,
                                          SMESH::double_array&  vals,
                                          GEOM::GEOM_Object_ptr geom)
{
  // set meshing parameters
  ...

  // Python Dump
  SMESH::TPythonDump() << _this() << ".SetParams( "
                       << nb   << ", "
                       << vals << ", "
                       << geom << ")"
}
```

A result Python command will look like following.

```
MyPluginName_Hypothesis.SetParams( 12, [ 2.1, 2.3 ], Face_1 )
```

SMESH::TPythonDump class defines << operator to print objects of most types used in SMESH (see the class definition in SMESH_SRC/src/SMESH_I/SMESH_PythonDump.hxx), so this step should be trivial.

2.6.2 smesh.py Python interface

Mesh module defines `smesh.py` Python interface simplifying creation of meshes in Python scripts. Consider, for example, the following Python command written with use of `smesh.py` interface.

```
mesh.Segment().NumberOfSegments( 12, 2.1 )
```

This command wraps the following separate commands:

1. Creation of an instance of `StdMeshers_Regular_1D` algorithm.
2. Addition of the created algorithm to the mesh.
3. Creation of an instance of `StdMeshers_NumberOfSegments` hypothesis.
4. Addition of the created hypothesis to the mesh.
5. Setting meshing parameters to the hypothesis.

To provide such a wrapping for your algorithm and hypotheses you need to:

- Define a Python class wrapping creation of your algorithm.
- Describe required wrapping in `<MyPluginName>.xml` file.

2.6.2.1 Python class of the algorithm

The Python class wrapping your algorithm is to be created in `<MyPluginName>DC.py` file.

See examples in
`SMESH_SRC/src/SMESH_SWIG/StdMeshersDC.py`
`NETGENPLUGIN_SRC/src/NETGENPlugin/NETGENPluginDC.py`

First, your Python file should import needed stuff from `smesh.py`:

```
from smesh import Mesh_Algorithm
```

Import your server library:

```
import MyPluginName
```

If your Python algorithm is to be created by one of existing methods of `smesh.Mesh` class, like `Triangle()` or `Hexahedron()` (which are actually dynamically added to `smesh.Mesh` class at loading of smesh.py), then you need to define an ID of your class, i.e. a string variable that will be used as an argument of this method to discriminate your algorithm from others:

```
MyPluginID = "MyPluginName_2D"
# this ID will be used like this:
# myAlgo = mesh.Quadrangle( algo=smesh.MyPluginID )
```

This variable is optional if you want that your algorithm to be created by a uniquely named method of smesh.Mesh class.

Inherit your class from `smesh.Mesh_Algorithm`:

```
class MyAlgorithm (Mesh_Algorithm):
```

Define tree obligatory attributes in your class:

- `meshMethod` – a string specifying a method of `smesh.Mesh` class by which your algorithm will be created (this method does not actually exist in `the smesh.Mesh` class; instead it will be dynamically added to it in runtime).

- `algoType` – a string serving as ID of your class (already mentioned above in this paragrath - MyPluginID).

- `docHelper` – a doc string of your class that will be used at generation of documentation of `meshMethod`.

```
class MyAlgorithm (Mesh_Algorithm):

   meshMethod = "MyQuadAlgorithm"
   algoType   = MyPluginID
   docHelper  = """Creates MyPluginName_Algorithm, which generates
                   quadrangles on arbitrary faces"""
```

As result of the example code above, your algorithm can be created by `smesh.Mesh` like this:

```
quadAlgo = mesh.MyQuadAlgorithm(algo=smesh.MyPluginID)
# or simply
quadAlgo = mesh.MyQuadAlgorithm()
```

Define a constructor and methods creating your hypothesis (and probably defining meshing parameters at once) like following.

```
def __init__(self, mesh, geom ):
    # This method creates an instance of your algorithm and assigns it
    #  to the mesh

    Mesh_Algorithm.__init__(self)
    self.Create( mesh, geom, "algoTypeName" ) # the last arg must be
    ## equal to the attribute _name of your algorithm core class
    ## (described in chapter 2.2.1.1)

def Parameters( nb, vals, geom ):
    # This method creates an instance of your hypothesis, assigns it to
    # the mesh and sets parameter values

    hyp = self.Hypothesis( "hypTypeName",
                           [nb, vals, geom],
                           "libMyPluginName.so")
    # where the last arg must be equal to the attribute _name of your
    # hypothesis core class (described in chapter 2.2.1.1).
    # The second arg will be used to give a name to your new hypothesis.
    # The third arg names your server plug-in library.

    hyp.SetParams( nb, vals, geom ) # initialize meshing parameters
```

As a result of the code above, your hypothesis can be created and initialized by the following command

```
myHyp = quadAlgo.Parameters( 12, [ 2.1, 2.3 ], Face_1 )
```

Update your `Makefile.am` to install the Python file where you define your class of algorithm:

```
dist_salomescript_DATA = <MyPluginName>DC.py
```

2.6.2.2 XML description of Python wrapping

In order to let SALOME know how to wrap calls relating to your plug-in into calls of `smesh.py` Python interface, you have to provide description of this wrapping in `<MyPluginName>.xml` file (described in chapter 2.5).

Suppose we want to get the following wrapping commands:

```
quadAlgo = mesh.MyQuadAlgorithm()
myHyp = quadAlgo.Parameters( 12, [ 2.1, 2.3 ], Face_1 )
```

An XML description of this wrapping should be added in the `<algorithm>` tag as its child:

```
<algorithm type = "algoTypeName"
           ...
           dim = "2">
   <python-wrap>
      <algo>algoTypeName=MyQuadAlgorithm( algo=smesh.MyPluginID )</algo>
      <hypo>
         hypTypeName=Parameters(SetParams(1),SetParams(2),SetParams(3))
      </hypo>
  </python-wrap>
</algorithm>
```

An `<algo>` tag describes how an algorithm of given type name (here "`algoTypeName`") is created:

- By what method of `smesh.Mesh` class (here "`MyQuadAlgorithm()`").
- With what algorithm ID as argument (here "`algo=smesh.MyPluginID`").

Algorithm ID argument "`(algo=...)`" can be omitted in our case as `MyQuadAlgorithm()` is a unique method name of `smesh.Mesh`, we could simply write "`algoTypeName=MyQuadAlgorithm()`".

`<hypo>` tag describes how a hypothesis of given type name (here "`hypTypeName`", which is equal to `_name` attribute of C++ class of your hypothesis and to "`type`" attribute of `<hypothesis>` tag) is created:

- By what method of the Python algorithm (here "`Parameters()`").
- With what arguments; each argument in this description specifies a method of hypothesis IDL API (here "`SetParams`") and one-based index of the argument of this method to use (here `"(1),(2),(3)"`). In our example the first argument ("`SetParams(1)`") means that the first argument of `MyAlgorithm.Parameters()` Python method is equal to the first argument of `MyPlugin_Hypothesis_i::SetParams()` method (implementing IDL API).

In "snapshot" mode of Python dump, consecutive commands changing a parameter of hypothesis are erased from a dump script if some intermediate parameter values are not used in the final mesh. Consider the following Python commands.

```
myHyp.SetParams(12, [ 2.1, 2.3 ], Face_1 )
myHyp.SetParams(10, [ 2.1, 2.3 ], Face_1 )
mesh.Compute()
```

The first command has no effect to the final mesh state and in “snapshot” mode it will be erased. But if a hypothesis command not simply changes a value of parameter but adds more values to a complex parameter, then repeated calls of such a command should not be erased.

The following XML description says SMESH Engine not to erase repeated calls of `AddParam1()` and `AddPara2()` methods of your hypothesis (`<python-wrap>` tag is added as a child to `<hypothesis>` tag):

```
<hypothesis type="hypTypeName" dim="2">
   <python-wrap>
      <accumulative-methods>
          AddParam1, AddParam2
      </accumulative-methods>
   </python-wrap>
</hypothesis>
```

2.7 Usage of notebook variables

This step is optional.

To enable usage of notebook variables by your plug-in you need to

- Pass names of variables used to set meshing parameters from GUI side to Engine side and backward (this is implemented in GUI part of your plug-in).
- Specify which arguments of a command being dumped can be set via notebook variables (this is done in C++ classes implementing IDL API of your hypotheses).

2.7.1.1 GUI part

Notebook variables can be entered in spinbox GUI controls only. So, if your Hypothesis Creator is inherited from `StdMeshersGUI_StdHypothesisCreator`, i.e. GUI widgets for hypothesis parameters are created automatically, then notebook variables can be used for parameters of simple types: `double`, `long` and `short`. If your Hypothesis Creator is inherited from `SMESHGUI_GenericHypothesisCreator` then you are to use classes `SalomeApp_IntSpinBox` and `SMESHGUI_SpinBox` to implement GUI controls for parameters that can be set via notebook variables.

When hypothesis edition starts, your Hypothesis Creator should initialize text of a spin widget by a name of variable, if it was used to set up a parameter. You can get this variable name by calling `SMESH_Hypothesis::GetVarParameter( string methodName )`, which returns an empty string if no variable was used to set the parameter. Here “`methodName`” is a name of the method setting the parameters, it is “`SetParams`” in the case of our sample:

```
hyp->GetVarParameter( "SetParams" );
```

If your Hypothesis Creator is inherited from `StdMeshersGUI_StdHypothesisCreator`, then the code defining contents of the spin widget will look like following.

```
SMESHGUI_GenericHypothesisCreator::StdParam item;
if ( !initVariableName( hyp, item, "SetParams" ))
  item.myValue = h->GetParam1(); // only the 1st arg can variable!
```

When values of parameters are passed from GUI widgets to your hypothesis, as hypothesis edition ends, your Hypothesis Creator should (the order of calls is important!):

- Pass a text from a widget to the hypothesis, which will store it if it's a name of variable; along with the text of widget, a name of the hypothesis method setting a parameter should be specified; for this, call `SMESH_Hypothesis::SetVarParameter( text, method )` method inherited by your hypothesis.

- Set a value of the parameter to your hypothesis.

If your Hypothesis Creator is inherited from `StdMeshersGUI_StdHypothesisCreator`, then the code doing this will look like this:

```
hyp->SetVarParameter( params[0].text(), "SetParams" );
hyp->SetParams( params[0].myValue.toInt(), vals, geom );
```

2.7.1.2 IDL API of hypotheses

In C++ code implementing IDL API of your hypothesis, you should wrap a parameter that can be defined via a notebook variable in an object of type `SMESH::TVar`. After such a modification, the code provided in chapter 2.6.1 will look as following.

```
void MyPluginName_Hypothesis_i::SetParams(CORBA::Long           nb,
                                          SMESH::double_array&  vals,
                                          GEOM::GEOM_Object_ptr geom)
{
  // set meshing parameters
  ...

  // Python Dump
  SMESH::TPythonDump() << _this() << ".SetParams( "
                       << SMESH::TVar( nb ) << ", "// !!!!!!!!!!!!!
                       << vals << ", "
                       << geom << ")"
}
```

2.8 Documentation

This step is optional.

You can provide a documentation of your meshing plug-in in any appropriate form. However, if you provide the documentation as HTML files, SALOME can automatically locate it and include into the *Help* menu of the SALOME GUI desktop.

Usual approach for documentation generation is to use doxygen program for generation of the HTML documentation from plain text files

SALOME will automatically search for the `index.html` file in the following directory:
`${MyPluginName_ROOT_DIR}/share/doc/salome/MyPluginName`

If the file is present, the reference to it is added to the *Help* menu. The position of this item in Help menu can be customized in SalomeApp.xml file. The usual approach is to put it to the *Help* → *Mesh module* → *Plug-ins* submenu (see paragraph 2.5.2).

Optionally, it is possible to generate documentation of the methods dynamically added to the `smesh.Mesh` class. To do this, you have to:

- Add doxygen-style documentation to your module's Python API (paragraph 2.6.2).

- Add rules for generation of documentation of dynamic methods to the Makefile.am responsible for your documentation generation. This can be done by means of `collect_mesh_methods.py` script supplied with the SMESH module. This script generates dummy `smesh.py` file in the build directory, with description of all dynamic methods. This file should be then listed as an additional input of doxygen. Be careful to avoid of copying dummy `smesh.py` file to the installation directory of your plug-in by mistake; it might break using of SMESH module's Python API in runtime.

See example in
`NETGENPLUGIN_SRC/doc/salome/gui/NETGENPLUGIN`

2.9 Build your plug-in

Configure and build your plug-in in the usual way (by invocation of build_configure, configure, make and make install commands). Check that all libraries (`libMyServerLib.so` and, optionally, `libMyClientLib.so`) are built; resource files are properly installed, etc.

2.10 Set up environment

Set environment variable `MyPluginName_ROOT_DIR` to your plug-in installation directory path. For example:

```
export MyPluginName_ROOT_DIR=/home/user/SALOME/INSTALL/MyPluginName
```

Note that you do not need to list your plug-in anywhere. It will be automatically detected by SALOME in runtime during the application initialization. You only need to specify `MyPluginName_ROOT_DIR` as described above.

SALOME will automatically locate your XML file, searching for it in the following directory (note that last component of the path is a name of plugin in lower case):

```
${MyPluginName_ROOT_DIR}/share/salome/resources/mypluginname
```

2.11 Run SALOME

Run SALOME application, create new study, load Mesh module. Via menu *Mesh → Create Mesh* invoke "Create Mesh" dialog box and look at the available algorithms list. If everything is done properly, you should see your algorithms in this list.

Try to create a new hypothesis and check, if your hypotheses are available. Define complete set of algorithms and hypotheses; click "OK" in the mesh creation dialog. Compute the created mesh.

Check the result of computation.

Références documentaires

Documents de référence

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

	Titre	Référence
[1]	Contrat Marché C434C71440: TMA PAL/SALOME 2008-2010 : Année 2010	C434C71440 / OC2D07081

Historique des révisions

Les versions successives du présent document sont :

Version en vigueur

Versions antérieures

Version	Rédacteur	Date	Objet de la révision
V1M8	V SANDLER	30/08/2012	Add chapter for documentation. Minor corrections.
V1M7	E AGAPOV	23/08/2012	Update for SALOME 6.6
V1M6	J DOROVSKIKH	17/09/2010	Update for SALOME 5.1.4
V1M5	J DOROVSKIKH	20/06/2007	Update for SALOME 3.2.6
V1M4	J DOROVSKIKH	07/02/2007	Update for SALOME 3.2.5
V1M3	J DOROVSKIKH	30/03/2006	Additional remarks
V1M2	M KAZAKOV	14/02/2006	Minor revision and remarks
V1M1	J DOROVSKIKH	14/02/2006	Draft version for validation
V1M0	J DOROVSKIKH	24/01/2006	Initial version