CUSTOMER: CEA DEN **PROJECT NAME:** PAL/SALOME 2006

PROJECT REF.: 4 0001 26 209. **DELIVERABLE:** L3/4 F Q1 2006

TITLE

USE OF SMDS API (SMESH SALOME MODULE)

TYPE

REFERENCE GUIDE

DOC. NAME: 20070620_SALOME_3.2.0_SMDS_REFERENCE_GUIDE.DOC

History

Version	Date	Author	Description	Status
0.0	24/01/2006	JFA	Initial version	D
0.1	14/02/2006	JFA	Draft version for validation	D
0.2	14/02/2006	MKO	Minor revision and remarks	P
0.3	07/02/2007	JFA	Update for Salome 3.2.5	D
0.4	20/06/2007	JFA	Update for Salome 3.2.6	D

D: Draft, **P**: Proposed for validation

Validation (if necessary)

Name	Date	Visa	Additional Remarks
	__/__/2006		

CONTENTS

1. INTRODUCTION

This document describes Salome SMDS package as a part of Salome SMESH module. Data structures, algorithms and interfaces, implemented in this package, will be documented here in order to provide easy and fast way to get acquainted with Salome SMDS for beginners and to point out some details for experienced developers.

The SMDS package implements basic structures and algorithms for keeping and processing Mesh.

It provides:

- Organization of a mesh as a transient object in a regular way, with internal hierarchy and quick access to any data.
- Mesh editing: add/remove/change mesh elements and nodes.
- Mesh statistics (number of nodes/elements, Ids).
- Advanced algorithms to analyze 3D Mesh elements.

2. SHORT MESH OVERVIEW

Mesh is a topological structure, constructed from simple parts (mesh elements). See Figure 1 for example of a mesh:

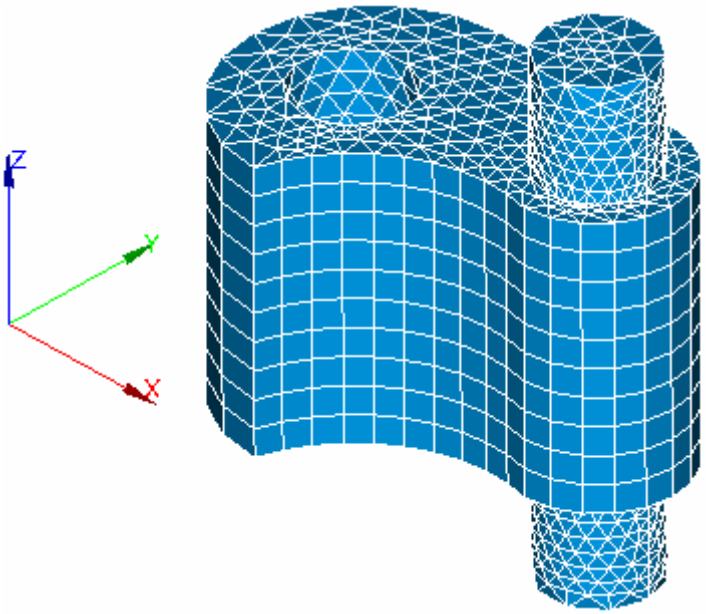

Figure 1. An example of a mesh

Depending on their dimensions, mesh elements are separated on:

- Nodes, which are 0D mesh elements. General info about a node is its coordinates in 3D space.
- Edges, which are 1D mesh elements. Edge is commonly defined by its two end nodes.
- Faces, which are 2D mesh elements. Face is commonly defined by its corner nodes.
- Volumes, which are 3D mesh elements. Volume is commonly defined by its corner nodes.

Depending on their complexity and usage frequency, mesh elements are separated on:

- Regular mesh elements. SMDS supports:
  1. 0D: Nodes.
  2. 1D: Linear edges.
  3. 2D: Triangles (three corner nodes) and quadrangles (four corner nodes). See Figure 2.

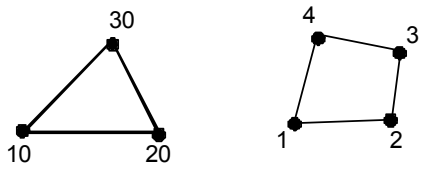

Figure 2. Triangles

  4. 3D: Tetrahedrons (four corner nodes), pyramids (five corner nodes), prisms (six corner nodes) and hexahedrons (eight corner nodes). See Figure 3.

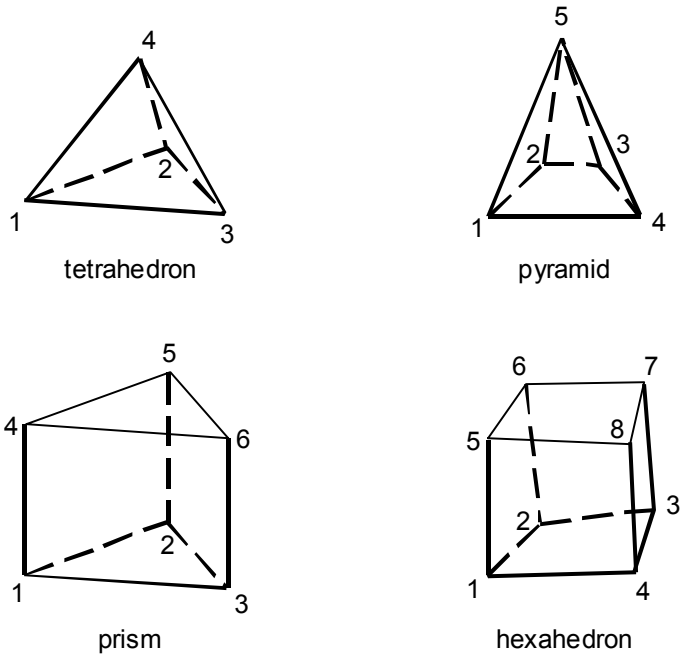

Figure 3. Tetrahedrons

- Polygonal mesh elements. SMDS supports:
  1. 2D: Polygons. Polygon is defined by sequence of its corner nodes. Any quantity of corner nodes can be given.
  2. 3D: Polyhedral volumes. Polyhedral volume is defined by its boundary faces, each of those is set as a sequence of its corner nodes.
- Quadratic mesh elements. Additionally to corner nodes each quadratic element has one middle node between each two neighbor corner nodes. SMDS supports:

1. 1D: Quadratic edges. In addition to two corner nodes, quadratic edge has one middle node.
2. 2D: Quadratic triangles and quadrangles.
3. 3D: Quadratic tetrahedrons, pyramids, prisms and hexahedrons.

Sometimes word «elements» mean all mesh elements (0D, 1D, 2D and 3D), in other cases only 1D, 2D and 3D elements are considered as «elements», and in those cases 0D elements are specially noted as nodes. Though in most places «mesh elements» mean all dimension elements, but «elements» mean only edges, faces and volumes.

It is possible to define a mesh of mixed type – e.g., combining triangles and quadrangles, as well as elements of different dimension (edges and volumes together with faces). The connectivity of mesh data is maintained in both ways: Element to Node and Node to Element; this allows creating simple algorithms of mesh traversal and locating the neighborhood of any node or element.

All nodes and elements are quickly accessed by ID integer value (unique for every node or element). Alternatively, the application can use mesh iterators. There are 4 types of mesh iterators, each one taking into account only one type of mesh entity (nodes, edges, faces or volumes).

Multiple meshes can be hold by an application at the same time. Each mesh has its own life cycle:

- Creation. Mesh is created as an object in memory and managed by its pointer.
- Filling with mesh elements. See chapter 3.2 for more about mesh elements creation.
- Mesh editing. You can add/remove/edit mesh elements along all the mesh living period.
- Mesh exploring. One can retrieve information about mesh and its entities and use it as he likes. For example, it can be used for visualization (graphic or textual) or in advanced computational algorithms.
- Mesh saving on disk. One can store mesh information on disk in his preferred format(s). SMDS package does not provide any special tool for this.
- Mesh destruction. When one finishes his work with any certain mesh, he deletes corresponding mesh object from memory.

In SMDS mesh is represented by class *SMDS_Mesh*. Mesh elements are represented by classes, inheriting common parent class *SMDS_MeshElement.*

3. CREATION OF MESH AND FILLING IT WITH MESH ELEMENTS.

3.1. Creation of meshes and sub-meshes

SMDS_Mesh class has two constructors. Public constructor has no arguments, it is used to create new meshes, having no parent meshes.

```
SMDS_Mesh();
```

Newly created mesh has neither elements, no sub-meshes and has the following default flags set:

```
myHasConstructionEdges = false
myHasConstructionFaces = false
myHasInverseElements = true
```

New NodeIDFactory and ElementIDFactory is created for the new mesh and is stored in its fields.

Private constructor requires one argument of type SMDS_Mesh* for parent mesh passage. It is intended for sub-meshes creation and is called from AddSubMesh() public method:

```
private:
  SMDS_Mesh(SMDS_Mesh * parent);
```

```
public:
  SMDS_Mesh *AddSubMesh();
```

Calling AddSubMesh() method of existing mesh Mesh1 creates new mesh Mesh2, which is a sub-mesh of Mesh1, i.e. it is inserted in sub-meshes of Mesh1 and has Mesh1 as its parent. Mesh2 is created empty (without elements and sub-meshes), has default flags set, and shares with Mesh1 NodeIDFactory and ElementIDFactory.

3.2. Filling mesh with elements

Mesh elements are created by calling the methods of the mesh object. During creation of element a unique integer identificator (ID) is automatically generated for it or can be set by user. For example, having a 3D point {x,y,z}, a node can be created like this:

```
SMDS_Mesh* aMesh;
double x,y,z;
int anID;
// ... initialization of aMesh, point coordinates, ID ...
SMDS_MeshNode* aNode1 = aMesh->AddNode(x, y, z);
// or
SMDS_MeshNode* aNode2 = aMesh->AddNode(x, y, z, anID);
```

After that the node ID can be retrieved like this:

```
int aNodeID = aNode1->GetID();
```

And the node itself can be retrieved by its ID like this:

```
SMDS_MeshNode* aNode = aMesh->FindNode(aNodeID);
```

3.3. Creation of a Node

A node is created by means of the method *AddNode* that takes as parameters the 3D Cartesian coordinates of a point:

```
virtual SMDS_MeshNode* AddNode(double x, double y, double z);
```

A node with predefined ID is created by means of the method *AddNodeWithID* that takes as parameters the 3D Cartesian coordinates of a point and node ID:

```
virtual SMDS_MeshNode* AddNodeWithID(double x, double y, double z, int
ID);
```

3.4. Creation of an Edge

3.4.1. Linear edge creation

A regular (linear) edge is created by means of the method *AddEdge and AddEdgeWithID* that takes as parameters the IDs or pointers of two existing nodes. The order of the nodes does not matter.

```
virtual SMDS_MeshEdge* AddEdgeWithID(int n1, int n2, int ID);

virtual SMDS_MeshEdge* AddEdgeWithID(const SMDS_MeshNode * n1,
                                     const SMDS_MeshNode * n2,
                                     int ID);

virtual SMDS_MeshEdge* AddEdge(const SMDS_MeshNode * n1,
                               const SMDS_MeshNode * n2);
```

3.4.2. Quadratic edge creation

A quadratic edge is created by means of the method *AddEdge and AddEdgeWithID* that takes as parameters the IDs or pointers of three existing nodes. The order of two first nodes does not matter, the fird node is the middle node.

```
virtual SMDS_MeshEdge* AddEdgeWithID(int n1, int n2, int n12, int ID);
virtual SMDS_MeshEdge* AddEdgeWithID(const SMDS_MeshNode * n1,
                                     const SMDS_MeshNode * n2,
                                     const SMDS_MeshNode * n12,
                                     int ID);
virtual SMDS_MeshEdge* AddEdge(const SMDS_MeshNode * n1,
                               const SMDS_MeshNode * n2,
                               const SMDS_MeshNode * n12);
```

3.5. Creation of a Face

3.5.1. Regular face creation

A regular face is created by means of the overloaded methods *AddFace and AddFaceWithID* that take as parameters the IDs or pointers of three (for triangle) or four (for quadrangle) existing nodes. The cyclic order of nodes is important, because it defines the orientation of the face.

```
virtual SMDS_MeshFace* AddFaceWithID(int n1, int n2, int n3, int ID);
virtual SMDS_MeshFace* AddFaceWithID(const SMDS_MeshNode * n1,
                                     const SMDS_MeshNode * n2,
                                     const SMDS_MeshNode * n3,
                                     int ID);
virtual SMDS_MeshFace* AddFace(const SMDS_MeshNode * n1,
                               const SMDS_MeshNode * n2,
                               const SMDS_MeshNode * n3);

virtual SMDS_MeshFace* AddFaceWithID(int n1, int n2, int n3, int n4,
int ID);
virtual SMDS_MeshFace* AddFaceWithID(const SMDS_MeshNode * n1,
                                     const SMDS_MeshNode * n2,
                                     const SMDS_MeshNode * n3,
                                     const SMDS_MeshNode * n4,
                                     int ID);
virtual SMDS_MeshFace* AddFace(const SMDS_MeshNode * n1,
                               const SMDS_MeshNode * n2,
                               const SMDS_MeshNode * n3,
                               const SMDS_MeshNode * n4);
```

Also there are methods for regular face on edges creation. Currently these methods are not used by SMESH module, because it doesn't support meshes with descendants. To properly use these methods, one must call at first method setConstructionEdges() with *true* to enable meshes with descendants. The same is true for volumes on faces also.

```
virtual SMDS_MeshFace* AddFaceWithID(const SMDS_MeshEdge * e1,
                                     const SMDS_MeshEdge * e2,
                                     const SMDS_MeshEdge * e3, int
ID);
virtual SMDS_MeshFace* AddFace(const SMDS_MeshEdge * e1,
                               const SMDS_MeshEdge * e2,
                               const SMDS_MeshEdge * e3);

virtual SMDS_MeshFace* AddFaceWithID(const SMDS_MeshEdge * e1,
                                     const SMDS_MeshEdge * e2,
                                     const SMDS_MeshEdge * e3,
```

```
                                    const SMDS_MeshEdge * e4, int
ID);
virtual SMDS_MeshFace* AddFace(const SMDS_MeshEdge * e1,
                               const SMDS_MeshEdge * e2,
                               const SMDS_MeshEdge * e3,
                               const SMDS_MeshEdge * e4);
```

3.5.2. Polygon creation

A polygonal face is created with methods *AddPolygonalFace* and *AddPolygonalFaceWithID* that take as first parameter a sequence of IDs or pointers of existing nodes. The cyclic order of nodes is important, because it defines the orientation of the face.

```
virtual SMDS_MeshFace* AddPolygonalFaceWithID (std::vector<int>
nodes_ids, const int ID);

virtual SMDS_MeshFace* AddPolygonalFaceWithID (std::vector<const
SMDS_MeshNode*> nodes, const int ID);

virtual SMDS_MeshFace* AddPolygonalFace (std::vector<const
SMDS_MeshNode*> nodes);
```

3.5.3. Quadratic face creation

A quadratic face is created by means of the overloaded methods *AddFace and AddFaceWithID* that take as parameters the IDs or pointers of six (for quadratic triangle) or eight (for quadratic quadrangle) existing nodes.

To add quadratic triangle, use the following methods, where n1, n2 and n3 is a corner nodes, and nij will be middle node between nodes ni and nj:

```
virtual SMDS_MeshFace* AddFaceWithID(int n1, int n2, int n3,
                                     int n12,int n23,int n31, int ID);

virtual SMDS_MeshFace* AddFaceWithID(const SMDS_MeshNode * n1,
                                     const SMDS_MeshNode * n2,
                                     const SMDS_MeshNode * n3,
                                     const SMDS_MeshNode * n12,
                                     const SMDS_MeshNode * n23,
                                     const SMDS_MeshNode * n31,
                                     int ID);

virtual SMDS_MeshFace* AddFace(const SMDS_MeshNode * n1,
                               const SMDS_MeshNode * n2,
                               const SMDS_MeshNode * n3,
                               const SMDS_MeshNode * n12,
                               const SMDS_MeshNode * n23,
                               const SMDS_MeshNode * n31);
```

To add quadratic quadrangle, use the following methods, where n1, n2, n3 and n4 is a corner nodes, and nij will be middle node between nodes ni and nj:

```
virtual SMDS_MeshFace* AddFaceWithID(int n1, int n2, int n3, int n4,
                                int n12,int n23,int n34,int n41, int ID);

virtual SMDS_MeshFace* AddFaceWithID(const SMDS_MeshNode * n1,
                                     const SMDS_MeshNode * n2,
                                     const SMDS_MeshNode * n3,
                                     const SMDS_MeshNode * n4,
                                     const SMDS_MeshNode * n12,
                                     const SMDS_MeshNode * n23,
                                     const SMDS_MeshNode * n34,
```

```
                                    const SMDS_MeshNode * n41,
                                    int ID);

virtual SMDS_MeshFace* AddFace(const SMDS_MeshNode * n1,
                               const SMDS_MeshNode * n2,
                               const SMDS_MeshNode * n3,
                               const SMDS_MeshNode * n4,
                               const SMDS_MeshNode * n12,
                               const SMDS_MeshNode * n23,
                               const SMDS_MeshNode * n34,
                               const SMDS_MeshNode * n41);
```

3.6. Creation of a Volume

3.6.1. Regular volume creation

A regular volume is created by means of the overloaded methods *AddVolume and AddVolumeWithID* that take as parameters the IDs or pointers of four (for tetrahedron), five (for pyramid), six (for prism) or eight (for hexahedron) existing nodes. The cyclic order of nodes is important, because it defines the orientation of the volume. See also Figure 3 to know appropriate order of nodes for volumes creation.

```
virtual SMDS_MeshVolume* AddVolumeWithID(int n1, int n2, int n3, int
n4, int ID);

virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshNode * n1,
                                         const SMDS_MeshNode * n2,
                                         const SMDS_MeshNode * n3,
                                         const SMDS_MeshNode * n4,
                                         int ID);

virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshNode * n1,
                                   const SMDS_MeshNode * n2,
                                   const SMDS_MeshNode * n3,
                                   const SMDS_MeshNode * n4);

virtual SMDS_MeshVolume* AddVolumeWithID(int n1, int n2, int n3, int
n4, int n5, int ID);

virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshNode * n1,
                                         const SMDS_MeshNode * n2,
                                         const SMDS_MeshNode * n3,
                                         const SMDS_MeshNode * n4,
                                         const SMDS_MeshNode * n5,
                                         int ID);

virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshNode * n1,
                                   const SMDS_MeshNode * n2,
                                   const SMDS_MeshNode * n3,
                                   const SMDS_MeshNode * n4,
                                   const SMDS_MeshNode * n5);

virtual SMDS_MeshVolume* AddVolumeWithID(int n1, int n2, int n3, int
n4, int n5, int n6, int ID);

virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshNode * n1,
                                         const SMDS_MeshNode * n2,
                                         const SMDS_MeshNode * n3,
                                         const SMDS_MeshNode * n4,
                                         const SMDS_MeshNode * n5,
                                         const SMDS_MeshNode * n6,
                                         int ID);
```

```
virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshNode * n1,
                                   const SMDS_MeshNode * n2,
                                   const SMDS_MeshNode * n3,
                                   const SMDS_MeshNode * n4,
                                   const SMDS_MeshNode * n5,
                                   const SMDS_MeshNode * n6);

virtual SMDS_MeshVolume* AddVolumeWithID(int n1, int n2, int n3, int
n4,int n5, int n6, int n7, int n8, int ID);

virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshNode * n1,
                                         const SMDS_MeshNode * n2,
                                         const SMDS_MeshNode * n3,
                                         const SMDS_MeshNode * n4,
                                         const SMDS_MeshNode * n5,
                                         const SMDS_MeshNode * n6,
                                         const SMDS_MeshNode * n7,
                                         const SMDS_MeshNode * n8,
                                         int ID);

virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshNode * n1,
                                   const SMDS_MeshNode * n2,
                                   const SMDS_MeshNode * n3,
                                   const SMDS_MeshNode * n4,
                                   const SMDS_MeshNode * n5,
                                   const SMDS_MeshNode * n6,
                                   const SMDS_MeshNode * n7,
                                   const SMDS_MeshNode * n8);
```

There are also methods for volumes on faces creation. See note about faces on edges creation above to properly use them.

```
virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshFace * f1,
                                         const SMDS_MeshFace * f2,
                                         const SMDS_MeshFace * f3,
                                         const SMDS_MeshFace * f4,
int ID);

virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshFace * f1,
                                   const SMDS_MeshFace * f2,
                                   const SMDS_MeshFace * f3,
                                   const SMDS_MeshFace * f4);

virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshFace * f1,
                                         const SMDS_MeshFace * f2,
                                         const SMDS_MeshFace * f3,
                                         const SMDS_MeshFace * f4,
                                         const SMDS_MeshFace * f5,
int ID);

virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshFace * f1,
                                   const SMDS_MeshFace * f2,
                                   const SMDS_MeshFace * f3,
                                   const SMDS_MeshFace * f4,
                                   const SMDS_MeshFace * f5);

virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshFace * f1,
                                         const SMDS_MeshFace * f2,
                                         const SMDS_MeshFace * f3,
```

```
                                        const SMDS_MeshFace * f4,
                                        const SMDS_MeshFace * f5,
                                        const SMDS_MeshFace * f6,
int ID);

virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshFace * f1,
                                   const SMDS_MeshFace * f2,
                                   const SMDS_MeshFace * f3,
                                   const SMDS_MeshFace * f4,
                                   const SMDS_MeshFace * f5,
                                   const SMDS_MeshFace * f6);
```

3.6.2. Polyhedral volume creation

A polyhedral volume is created by means of the overloaded methods *AddPolyhedralVolume and AddPolyhedralVolumeWithID* that take as a first parameter a sequence of nodes (IDs or pointers), as the second parameter sequence of quantities (integer numbers).

```
virtual SMDS_MeshVolume* AddPolyhedralVolumeWithID
                         (std::vector<int> nodes_ids,
                          std::vector<int> quantities,
                          const int        ID);

virtual SMDS_MeshVolume* AddPolyhedralVolumeWithID
                         (std::vector<const SMDS_MeshNode*> nodes,
                          std::vector<int> quantities,
                          const int        ID);

virtual SMDS_MeshVolume* AddPolyhedralVolume
                         (std::vector<const SMDS_MeshNode*> nodes,
                          std::vector<int>       quantities);
```

Through the <quantities> one should pass quantities of nodes in each face of volume. For example, let the volume being created must have N faces. Then length of the <quantities> array will be N, quantities[i] will be a quantity of nodes in the i-th face. Length of the <nodes> array will be quantities[1] + quantities[2] + ... + quantities[N]. The <nodes> array will contain sequentially all the nodes of each face: at first quantities[1] nodes for the first face, then quantities[2] nodes for the second face, and so on. As each node of volume normally belongs to several faces of it, it will be contained in the <nodes> several times.

3.6.3. Quadratic volume creation

A quadratic volume is created by means of the overloaded methods *AddVolume and AddVolumeWithID* that take as parameters appropriate quantity of nodes (IDs or pointers). To know necessary quantity of nodes for some type of volume creation, one should add quantity of edges (Qe) to quantity of corner nodes (Qc). Results will be the following (nij means medium node between nodes ni and nj):

- For tetrahedron Qc = 4, Qe = 6, so use methods, taking ten nodes:

```
virtual SMDS_MeshVolume* AddVolumeWithID(int n1, int n2, int n3, int
n4, int n12,int n23,int n31, int n14,int n24,int n34, int ID);

virtual SMDS_MeshVolume* AddVolumeWithID(const SMDS_MeshNode * n1,
                                         const SMDS_MeshNode * n2,
                                         const SMDS_MeshNode * n3,
                                         const SMDS_MeshNode * n4,
                                         const SMDS_MeshNode * n12,
                                         const SMDS_MeshNode * n23,
                                         const SMDS_MeshNode * n31,
```

```
                                        const SMDS_MeshNode * n14,
                                        const SMDS_MeshNode * n23,
                                        const SMDS_MeshNode * n34,
                                        int ID);

virtual SMDS_MeshVolume* AddVolume(const SMDS_MeshNode * n1,
                                   const SMDS_MeshNode * n2,
                                   const SMDS_MeshNode * n3,
                                   const SMDS_MeshNode * n4,
                                   const SMDS_MeshNode * n12,
                                   const SMDS_MeshNode * n23,
                                   const SMDS_MeshNode * n31,
                                   const SMDS_MeshNode * n14,
                                   const SMDS_MeshNode * n23,
                                   const SMDS_MeshNode * n34);
```

- For pyramid Qc = 5, Qe = 8, so use methods, taking 13 nodes:

```
n1, n2, n3, n4, n5,
n12, n23, n34, n41, n15, n25, n35, n45.
```

- For prism Qc = 6, Qe = 9, so use methods, taking 15 nodes:

```
n1, n2, n3, n4, n5, n6,
n12, n23, n31, n45, n56, n64, n14, n25, n36
```

- For hexahedron Qc = 8, Qe = 12, so use methods, taking 20 nodes:

```
n1, n2, n3, n4, n5, n6, n7, n8,
n12, n23, n34, n41, n56, n67, n78, n85, n15, n26, n37, n48
```

Note: The cyclic order of nodes is important, because it defines the orientation of the volume.

4. MESH STRUCTURE EXPLORATION METHODS

To iterate through all mesh elements of certain type, there are four iterators:

```
SMDS_NodeIteratorPtr nodesIterator() const;

SMDS_EdgeIteratorPtr edgesIterator() const;

SMDS_FaceIteratorPtr facesIterator() const;

SMDS_VolumeIteratorPtr volumesIterator() const;
```

To iterate through all mesh elements of all types, except nodes, there is elementsIterator() method:

```
SMDS_ElemIteratorPtr elementsIterator() const;
```

To know type of any mesh element, GetElementType() method is implemented:

```
SMDSAbs_ElementType GetElementType( const int id, const bool iselem )
const;
```

To find node or element by its ID there are following two methods:

```
const SMDS_MeshNode *FindNode(int idnode) const;

const SMDS_MeshElement *FindElement(int IDelem) const;
```

To find an element by its nodes there are following methods:

```
const SMDS_MeshEdge *FindEdge(int idnode1, int idnode2) const;

const SMDS_MeshFace *FindFace(int idnode1, int idnode2, int idnode3)
const;

const SMDS_MeshFace *FindFace(int idnode1, int idnode2, int idnode3,
int idnode4) const;

const SMDS_MeshFace *FindFace(std::vector<int> nodes_ids) const;

static const SMDS_MeshEdge* FindEdge(const SMDS_MeshNode * n1,
                                     const SMDS_MeshNode * n2);

static const SMDS_MeshFace* FindFace(const SMDS_MeshNode *n1,
                                     const SMDS_MeshNode *n2,
                                     const SMDS_MeshNode *n3);

static const SMDS_MeshFace* FindFace(const SMDS_MeshNode *n1,
                                     const SMDS_MeshNode *n2,
                                     const SMDS_MeshNode *n3,
                                     const SMDS_MeshNode *n4);

static const SMDS_MeshFace* FindFace(std::vector<const SMDS_MeshNode
*> nodes);
```

Information about quantity of mesh entities can be retrieved with methods:

```
int NbNodes() const;

int NbEdges() const;

int NbFaces() const;

int NbVolumes() const;

int NbSubMesh() const;
```

To obtain first/last used mesh element ID, one can call:

```
int MaxNodeID() const;

int MinNodeID() const;

int MaxElementID() const;

int MinElementID() const;
```

For debug purpose, mesh contents can be dumped on standard output with methods:

```
void DumpNodes() const;

void DumpEdges() const;

void DumpFaces() const;

void DumpVolumes() const;

void DebugStats() const;
```

5. MESH ELEMENTS EDITION AND DELETION

5.1. Edition

To edit mesh element (but not node), one can use the following two methods of Mesh object:

```
static bool ChangeElementNodes(const SMDS_MeshElement * elem,
                               const SMDS_MeshNode    * nodes[],
                               const int                nbnodes);

static bool ChangePolyhedronNodes(const SMDS_MeshElement * elem,
std::vector<const SMDS_MeshNode*> nodes, std::vector<int> quantities);
```

Avoid direct using of methods ChangeNodes() on elements if you use information about nodes inverse elements. Note, that such information is used by SMDS_VolumeTool class.

To edit a node, its own interface has to be used (method setXYZ(), see 6.2.1 for it).

One can do automatic renumbering of mesh elements with following method of Mesh object:

```
virtual void Renumber (const bool isNodes,
                       const int startID = 1, const int deltaID = 1);
```

Order of mesh elements will not be changed by this method, only their IDs will be changed to correspond renumbering conditions (start ID and delta ID).

5.2. Deletion

For elements and nodes removal there are some methods implemented:

```
virtual void RemoveElement(const SMDS_MeshElement *        elem,
std::list<const SMDS_MeshElement *>& removedElems, std::list<const
SMDS_MeshElement *>& removedNodes,
                           const bool removenodes = false);

virtual void RemoveElement(const SMDS_MeshElement * elem, bool
removenodes = false);

virtual void RemoveNode(const SMDS_MeshNode * node);

virtual void RemoveEdge(const SMDS_MeshEdge * edge);

virtual void RemoveFace(const SMDS_MeshFace * face);

virtual void RemoveVolume(const SMDS_MeshVolume * volume);
```

Any mesh element (node, edge, face or volume) can be removed from the mesh.

No special care is needed when removing a node. There is no risk that other elements remain referring to the removed node. The inverse connections are built automatically and the node has a list of all the elements that refer to this node, all those elements are removed too. It provides maintenance of the data consistency in a safe way.

The operation of the deletion of a node has two options:

- Only free - the node will be removed only if it is free (has no inverse elements). If this option is off then inverse elements will be removed as well.
- Delete free nodes – it means that all nodes becoming free after the deletion the inverse elements of the given node will be also removed from the mesh (default mode). If this option is off then no other nodes are deleted from the mesh.

The operation of deletion of a mesh element other than a node has an option of deletion of nodes. With this option all the nodes constituting this element are removed if they become free.

To remove current mesh from its parent mesh, use method:

```
virtual bool RemoveFromParent();
```

To remove any sub-mesh of current mesh, pass it to the method:

```
virtual bool RemoveSubMesh(const SMDS_Mesh * aMesh);
```

6. MESH ELEMENTS CLASSES

All mesh objects, that can be stored in Mesh (and Mesh class itself, as it can be placed as sub-mesh), inherit common parent class SMDS_MeshObject. SMDS_MeshObject is absolutely general, it defines only virtual destructor.

Mesh is represented by class *SMDS_Mesh,* directly inheriting *SMDS_MeshObject* class.

Mesh elements are represented by classes, inheriting common parent class *SMDS_MeshElement*, which also inherits *SMDS_MeshObject* class.

Inheritance tree is the following:

- *SMDS_MeshObject*
    - *SMDS_Mesh*
    - *SMDS_MeshGroup*
    - *SMDS_MeshIDFactory*
        - *SMDS_MeshElementIDFactory*
    - *SMDS_MeshElement*
        - *SMDS_MeshNode*
        - *SMDS_MeshEdge*
            - *SMDS_QuadraticEdge*
        - *SMDS_MeshFace*
            - *SMDS_FaceOfEdges*
            - *SMDS_FaceOfNodes*
            - *SMDS_QuadraticFaceOfNodes*
            - *SMDS_PolygonalFaceOfNodes*
        - *SMDS_MeshVolume*
            - *SMDS_VolumeOfEdges*
            - *SMDS_VolumeOfNodes*
                - *SMDS_PolyhedralVolumeOfNodes*
            - *SMDS_QuadraticVolumeOfNodes*

6.1. SMDS_MeshElement class

```
class SMDS_WNT_EXPORT SMDS_MeshElement:public SMDS_MeshObject
{
  public:
   SMDS_ElemIteratorPtr nodesIterator() const;
   SMDS_ElemIteratorPtr edgesIterator() const;
   SMDS_ElemIteratorPtr facesIterator() const;

   virtual SMDS_ElemIteratorPtr elementsIterator(SMDSAbs_ElementType
type) const;

   virtual int NbNodes() const;
   virtual int NbEdges() const;
   virtual int NbFaces() const;

   int GetID() const;

   // Return the type of the current element
   virtual SMDSAbs_ElementType GetType() const = 0;
   virtual bool IsPoly() const { return false; };
   virtual bool IsQuadratic() const;

   virtual bool IsMediumNode(class SMDS_MeshNode* node) const;

   friend std::ostream & operator <<(std::ostream & OS, const
SMDS_MeshElement *);
   friend bool SMDS_MeshElementIDFactory::BindID(int
ID,SMDS_MeshElement*elem);

  protected:
   SMDS_MeshElement(int ID=-1);
   virtual void Print(std::ostream & OS) const;

  private:
   int myID;
};
```

Method GetType() is abstract. It is implemented in child classes and thus returns corresponding element types. Possible element types are:

- SMDSAbs_All,
- SMDSAbs_Node,
- SMDSAbs_Edge,
- SMDSAbs_Face,
- SMDSAbs_Volume

To distinguish polygonal elements from the others, virtual method IsPoly() is defined in SMDS_MeshElement class and redefined in classes representing polygonal elements.

To distinguish quadratic elements from the regular ones, virtual method IsQuadratic() is defined in SMDS_MeshElement class and redefined in classes representing quadratic elements.

In order to be able to differ corner nodes of element from medium nodes, located on edges, virtual method IsMediumNode(const SMDS_MeshNode* node) is defined in SMDS_MeshElement class.

Method Print() is called by friend operator<<() for element dumping into a stream. Print() method is virtual and it is reimplemented in each child class, which add to dump its own data.

6.2. Regular elements classes

6.2.1. Class for nodes

```
class SMDS_MeshNode:public SMDS_MeshElement
{
  public:
   SMDS_MeshNode(double x, double y, double z);
   ...
   SMDSAbs_ElementType GetType() const;
   int NbNodes() const;

   void setXYZ(double x, double y, double z);
   double X() const;
   double Y() const;
   double Z() const;

   void AddInverseElement(const SMDS_MeshElement * ME);
   void RemoveInverseElement(const SMDS_MeshElement * parent);
   void ClearInverseElements();
   bool emptyInverseElements();
   SMDS_ElemIteratorPtr GetInverseElementIterator(SMDSAbs_ElementType
type = SMDSAbs_All) const;
   int NbInverseNodes(SMDSAbs_ElementType type = SMDSAbs_All) const;

   void SetPosition(const SMDS_PositionPtr& aPos);
   const SMDS_PositionPtr& GetPosition() const;

   friend bool operator<(const SMDS_MeshNode& e1, const SMDS_MeshNode&
e2);

  protected:
   SMDS_ElemIteratorPtr elementsIterator(SMDSAbs_ElementType type)
const;

  private:
   double myX, myY, myZ;
   SMDS_PositionPtr myPosition;
   NCollection_List<const SMDS_MeshElement*> myInverseElements;
};
```

GetType() method returns SMDSAbs_Node.

NbNodes() method returns 1.

Operator < compares nodes by they IDs.

Inverse Connections.

Normally the data structure supports the links between nodes and elements built of them in one direction, concretely an element contains links to nodes. For performing some tasks it is needed to support inverse links from a node to elements that refer to it.

The class *SMDS_MeshNode* supports the mechanism of inverse connections. It has the methods that allow to establish, edit and inquire the list of inverse connections.

Position of a Node Related to the Underlying Geometry.

A data structure supports the relation of a node with the underlying geometry from the initial CAD model. For this purpose an object of the class *SMDS_Position* can be attached to a node.

SMDS_Position is an abstract class that defines the interface allowing to retrieve the ID of related subshape, the type of position that identifies the type of this subshape and the true 3D Cartesian coordinates obtained from initial geometry.

There are four classes inheriting *SMDS_Position*:

1. *SMDS_SpacePosition* is used to characterize a mesh node with a 3D point in space not related to any underlying geometry.
2. *SMDS_VertexPosition* is used to characterize a mesh node with a CAD vertex.
3. *SMDS_EdgePosition* is used to characterize a mesh node with a CAD edge. An object of this class can be used to store the U parameter of point on curve.
4. *SMDS_FacePosition* is used to characterize a mesh node with a CAD face. An object of this class can be used to store the U and V parameters of a point on surface.

Although these classes are not abstract and can be used directly to create objects to be attached to nodes, a user can derive from them his own classes appropriate to his CAD model. Furthermore this definition is certainly necessary if the 3D coordinates of a position are supposed to be used, because the classes *VertexPosition*, *EdgePosition* and *FacePosition* have little information to retrieve true 3D coordinates of the position.

6.2.2. Class for linear edges

```
class SMDS_MeshEdge:public SMDS_MeshElement
{
  public:
   SMDS_MeshEdge(const SMDS_MeshNode* node1,
                       const SMDS_MeshNode* node2);
         bool ChangeNodes(const SMDS_MeshNode* node1,
                          const SMDS_MeshNode* node2);
   ...
   SMDSAbs_ElementType GetType() const;
   int NbNodes() const;
   int NbEdges() const;
   friend bool operator<(const SMDS_MeshEdge& e1, const SMDS_MeshEdge&
e2);

  protected:
   SMDS_ElemIteratorPtr elementsIterator(SMDSAbs_ElementType type)
const;

  private:
   const SMDS_MeshNode* myNodes[2];
};
```

GetType() method returns SMDSAbs_Edge.

NbNodes() method returns 2.

NbEdges() method returns 1.

Operator < compares edges by their nodes IDs. If one edge E1 has nodes N1 and N2 (N1 < N2), another edge E2 has nodes N3 and N4 (N3 < N4), then E1 is recognized less than E2, if {N1 < N3} or {N1 == N3 and N2 < N4}.

6.2.3. Class for regular faces

Basic class for all 2D elements reimplements only method GetType(). It returns SMDSAbs_Face:

```
class SMDS_MeshFace:public SMDS_MeshElement
{
  public:
   SMDSAbs_ElementType GetType() const;
};
```

Class for regular faces:

```
class SMDS_FaceOfNodes:public SMDS_MeshFace
{
  public:
        SMDS_FaceOfNodes(const SMDS_MeshNode* node1,
                         const SMDS_MeshNode* node2,
                         const SMDS_MeshNode* node3);
        SMDS_FaceOfNodes(const SMDS_MeshNode* node1,
                         const SMDS_MeshNode* node2,
                         const SMDS_MeshNode* node3,
                         const SMDS_MeshNode* node4);
        bool ChangeNodes(const SMDS_MeshNode* nodes[],
                         const int            nbNodes);
        ...
        int NbNodes() const;
        int NbEdges() const;
        int NbFaces() const;

  protected:
        SMDS_ElemIteratorPtr elementsIterator(SMDSAbs_ElementType
type) const;

  private:
        const SMDS_MeshNode* myNodes[4];
        int                  myNbNodes;
};
```

NbNodes() method returns 3 for triangles and 4 for quadrangles (depends on real state).

NbEdges() method returns the same value, as NbNodes().

NbFaces() method returns 1.

6.2.4. Class for regular volumes

Basic class for all 3D elements reimplements only method GetType(). It returns SMDSAbs_Volume:

```
class SMDS_MeshVolume:public SMDS_MeshElement
{
  public:
```

```
   SMDSAbs_ElementType GetType() const;
};
```

Class for regular volumes

```
class SMDS_VolumeOfNodes:public SMDS_MeshVolume
{
  public:
    SMDS_VolumeOfNodes(const SMDS_MeshNode * node1,
                       const SMDS_MeshNode * node2,
                       const SMDS_MeshNode * node3,
                       const SMDS_MeshNode * node4);
    SMDS_VolumeOfNodes(const SMDS_MeshNode * node1,
                       const SMDS_MeshNode * node2,
                       const SMDS_MeshNode * node3,
                       const SMDS_MeshNode * node4,
                       const SMDS_MeshNode * node5);
    SMDS_VolumeOfNodes(const SMDS_MeshNode * node1,
                       const SMDS_MeshNode * node2,
                       const SMDS_MeshNode * node3,
                       const SMDS_MeshNode * node4,
                       const SMDS_MeshNode * node5,
                       const SMDS_MeshNode * node6);
    SMDS_VolumeOfNodes(const SMDS_MeshNode * node1,
                       const SMDS_MeshNode * node2,
                       const SMDS_MeshNode * node3,
                       const SMDS_MeshNode * node4,
                       const SMDS_MeshNode * node5,
                       const SMDS_MeshNode * node6,
                       const SMDS_MeshNode * node7,
                       const SMDS_MeshNode * node8);
    bool ChangeNodes(const SMDS_MeshNode* nodes[],
                     const int            nbNodes);
    ...
    int NbNodes() const;
    int NbEdges() const;
    int NbFaces() const;

  protected:
    SMDS_ElemIteratorPtr  elementsIterator(SMDSAbs_ElementType type)
const;
    const SMDS_MeshNode** myNodes;
    int                   myNbNodes;
};
```

6.3. Polygonal elements classes

6.3.1. Class for polygons

Class SMDS_PolygonalFaceOfNodes inheriting from SMDS_MeshFace represents a polygon.

```
class SMDS_PolygonalFaceOfNodes:public SMDS_MeshFace
{
 public:
  SMDS_PolygonalFaceOfNodes (std::vector<const SMDS_MeshNode *>
nodes);

  virtual SMDSAbs_ElementType GetType() const;
```

```
  virtual bool IsPoly() const { return true; };

  bool ChangeNodes (std::vector<const SMDS_MeshNode *> nodes);
  bool ChangeNodes (const SMDS_MeshNode* nodes[],
                    const int            nbNodes);

  virtual int NbNodes() const;
  virtual int NbEdges() const;
  virtual int NbFaces() const;

  ...

 private:
  std::vector<const SMDS_MeshNode *> myNodes;
};
```

6.3.2. Class for polyhedral volumes

Class SMDS_PolyhedralVolumeOfNodes inheriting from SMDS_VolumeOfNodes represents a polyhedral volume.

```
class SMDS_PolyhedralVolumeOfNodes:public SMDS_VolumeOfNodes
{
 public:
  SMDS_PolyhedralVolumeOfNodes (std::vector<const SMDS_MeshNode *>
nodes,
                                std::vector<int> quantities);

  virtual SMDSAbs_ElementType GetType() const;
  virtual bool IsPoly() const { return true; };

  bool ChangeNodes (const std::vector<const SMDS_MeshNode *> nodes,
                    const std::vector<int> quantities);

  virtual int NbNodes() const;
  virtual int NbEdges() const;
  virtual int NbFaces() const;

  int NbFaceNodes (const int face_ind) const;
  // 1 <= face_ind <= NbFaces()

  const SMDS_MeshNode* GetFaceNode (const int face_ind, const int
node_ind) const;
  // 1 <= face_ind <= NbFaces()
  // 1 <= node_ind <= NbFaceNodes()

  ...

  virtual const SMDS_MeshNode* GetNode (const int ind) const;
  SMDS_ElemIteratorPtr uniqueNodesIterator() const;
  int NbUniqueNodes() const;

 protected:
  SMDS_ElemIteratorPtr elementsIterator (SMDSAbs_ElementType type)
const;

 private:
  std::vector<const SMDS_MeshNode *> myNodesByFaces;
  std::vector<int> myQuantities;
};
```

6.4. Quadratic elements classes

6.4.1. Class for quadratic edges

Class SMDS_QuadraticEdge inheriting from SMDS_MeshEdge represents a quadratic edge.

```
class SMDS_QuadraticEdge:public SMDS_MeshEdge
{
  public:
   SMDS_QuadraticEdge(const SMDS_MeshNode * node1,
                          const SMDS_MeshNode * node2,
                          const SMDS_MeshNode * node12);

  virtual bool IsQuadratic() const { return true; }
  virtual bool IsMediumNode(class SMDS_MeshNode* node) const;

  SMDS_NodeIteratorPtr interlacedNodesIterator() const;
  ...
};
```

The method interlacedNodesIterator() intended for iteration on nodes in the order they are encountered walking along the edge.

6.4.2. Class for quadratic faces

Class SMDS_QuadraticFaceOfNodes inheriting SMDS_MeshFace represents a quadratic face.

```
class SMDS_QuadraticFaceOfNodes:public SMDS_MeshFace
{
public:
  SMDS_QuadraticFaceOfNodes (const SMDS_MeshNode * n1,
                             const SMDS_MeshNode * n2,
                             const SMDS_MeshNode * n3,
                             const SMDS_MeshNode * n12,
                             const SMDS_MeshNode * n23,
                             const SMDS_MeshNode * n31);
  SMDS_QuadraticFaceOfNodes(const SMDS_MeshNode * n1,
                            const SMDS_MeshNode * n2,
                            const SMDS_MeshNode * n3,
                            const SMDS_MeshNode * n4,
                            const SMDS_MeshNode * n12,
                            const SMDS_MeshNode * n23,
                            const SMDS_MeshNode * n34,
                            const SMDS_MeshNode * n41);

  virtual bool IsQuadratic() const { return true; }
  virtual bool IsMediumNode(class SMDS_MeshNode* node) const;

  SMDS_NodeIteratorPtr interlacedNodesIterator() const;
  ...
};
```

The method interlacedNodesIterator() intended for iteration on face nodes in the order they are encountered walking along edges.

6.4.3. Class for quadratic volumes

To maintain quadratic volumes, there is class SMDS_QuadraticVolumeOfNodes, inheriting SMDS_MeshVolume.

```
class SMDS_QuadraticVolumeOfNodes : public SMDS_MeshVolume
{
public:
  // tetrahedron with 10 nodes
  SMDS_QuadraticVolumeOfNodes (const SMDS_MeshNode * n1,
                               const SMDS_MeshNode * n2,
                               const SMDS_MeshNode * n3,
                               const SMDS_MeshNode * n4,
                               const SMDS_MeshNode * n12,
                               const SMDS_MeshNode * n23,
                               const SMDS_MeshNode * n31,
                               const SMDS_MeshNode * n14,
                               const SMDS_MeshNode * n23,
                               const SMDS_MeshNode * n34);
  // pyramid with 13 nodes
  SMDS_QuadraticVolumeOfNodes(const SMDS_MeshNode * n1,
  ...
  //  Pentahedron with 15 nodes
  SMDS_QuadraticVolumeOfNodes(const SMDS_MeshNode * n1,
  ...
  // Hexahedron with 20 nodes
  SMDS_QuadraticVolumeOfNodes(const SMDS_MeshNode * n1,
  ...
  virtual bool IsQuadratic() const { return true; }
  virtual bool IsMediumNode(class SMDS_MeshNode* node) const;
  ...
};
```

7. LIST OF SMDS HEADER FILES

- Mesh Object

  `SMDS_MeshObject.hxx`
- Mesh

  `SMDS_Mesh.hxx`
- Groups

  `SMDS_MeshGroup.hxx`
- Mesh Elements
  - Basic

    `SMDSAbs_ElementType.hxx`
    `SMDS_MeshElement.hxx`
  - Nodes

    `SMDS_MeshNode.hxx`
  - Edges

    `SMDS_MeshEdge.hxx`
    `SMDS_QuadraticEdge.hxx`
  - Faces

    `SMDS_MeshFace.hxx`
    `SMDS_FaceOfEdges.hxx`
    `SMDS_FaceOfNodes.hxx`
    `SMDS_PolygonalFaceOfNodes.hx`
    `SMDS_QuadraticFaceOfNodes.hxx`
  - Volumes

    `SMDS_MeshVolume.hxx`

```
SMDS_VolumeOfFaces.hxx
SMDS_VolumeOfNodes.hxx
SMDS_PolyhedralVolumeOfNodes.hxx
SMDS_QuadraticVolumeOfNodes.hxx
```

- IDs Factory

```
SMDS_MeshIDFactory.hxx
SMDS_MeshElementIDFactory.hxx
```

- Iterators

```
SMDS_Iterator.hxx
SMDS_ElemIterator.hxx
SMDS_IteratorOfElements.hxx
```

- Volume Tool

```
SMDS_VolumeTool.hxx
```

- SMDS Iterator (specific SMDS iterator, iterating over abstract set of values like STL containers)

```
SMDS_Iterator.hxx
```

- Position

```
SMDS_TypeOfPosition.hxx
SMDS_Position.hxx
SMDS_VertexPosition.hxx
SMDS_EdgePosition.hxx
SMDS_FacePosition.hxx
SMDS_SpacePosition.hxx
```

8. NOT USED / NOT FULLY IMPLEMENTED SMDS FUNCTIONALITIES

- SMDS_Mesh::boundaryFaces(), SMDS_Mesh::boundaryEdges() methods are declared, but not implemented.
- SMDS_VolumeOfNodes::GetType() method implementation duplicates SMDS_MeshVolume::GetType() method.
- Position classes are defined in SMDS package, are used by SMDS_MeshNode class (only interface and field), but does not really usefull in SMDS, because there is no link to any shape. May be move them to SMESHDS? Unreal, because in SMESHDS there is no class, inheriting SMDS_MeshNode.
- SMDS_MeshGroup implemented in SMDS, but is not really usefull here (no links on groups in SMDS_Mesh or in any other classes from SMDS package). May be move it to SMESHDS? Or move GroupBase and Group from SMESHDS here?
- Methods AddSubMesh(), SMDS_Mesh(parent), RemoveSubMesh(), RemoveFromParent(), NbSubMesh() and field std::list<SMDS_Mesh*> myChildren present in SMDS_Mesh class, but there are no methods to iterate through sub-meshes. And all these functionalities are no used by SMESHDS or any other SMESH package!
- Functionality for meshes with descendants support is not fully implemented and is not used in SMESH module.
- SMDS_Mesh::myHasInverseElements flag is not used.