Object Libraries

Application Framework User's Guide

Version 6.3 / September 2008

NOTICE FOR USERS:
This User Guide is a general instruction for Open CASCADE study. It may be incomplete and even contain occasional mistakes, particularly in examples, samples, etc.
Open CASCADE S.A.S. bears no responsibility for such mistakes. If you find any mistakes or imperfections in this document, or if you have suggestions for improving this document, please, contact us and contribute your share to the development of Open CASCADE Technology: bugmaster@opencascade.com

Tour Opus 12

77, Esplanade du Général de Gaulle

92914 PARIS LA DEFENSE

FRANCE

Table of Contents

1. Introduction

This manual explains how to use the Open CASCADE Application Framework (OCAF). It provides basic documentation on using OCAF. For advanced information on OCAF and its applications, see our offerings on our web site at www.opencascade.com/support/training.html

OCAF (the Open CASCADE Application Framework) is a RAD (Rapid Application Development) framework used for specifying and organizing application data. To do this, OCAF provides:

- Ready-to-use data common to most CAD/CAM applications,
- A scalable extension protocol for implementing new application specific data,
- An infrastructure
  - To attach any data to any topological element
  - To link data produced by different applications (*associativity of data*)
  - To register the modeling process - the creation history, or parametrics, used to carry out the modifications.

Using OCAF, the application designer concentrates on the functionality and its specific algorithms. In this way, he avoids architectural problems notably implementing Undo-redo and saving application data.

In OCAF, all of the above are already handled for the application designer, allowing him to reach a significant increase in productivity.

In this respect, OCAF is much more than just one toolkit among many in the CAS.CADE Object Libraries. Since it can handle any data and algorithms in these libraries - be it modeling algorithms, topology or geometry - OCAF is a logical supplement to these libraries.

The table below contrasts the design of a modeling application using object libraries alone and using OCAF.

Development tasks	Without OCAF	With OCAF
Creation of geometry Algorithm Calling the modeling libraries	To be created by the user	To be created by the user
Data organization Including specific attributes and modeling process	To be created by the user	Simplified
Saving data in a file Notion of *document*	To be created by the user	Provided
Document-view management	To be created by the user	Provided
Application infrastructure New, Open, Close, Save and Save As File menus	To be created by the user	Provided
Undo-Redo Robust, multi-level	To be created by the user	Provided
Application-specific dialog boxes	To be created by the user	To be created by the user

Table 1: Services provided by OCAF

The relationship between OCAF and the Open CASCADE Object Libraries can be seen in figure 1 below.

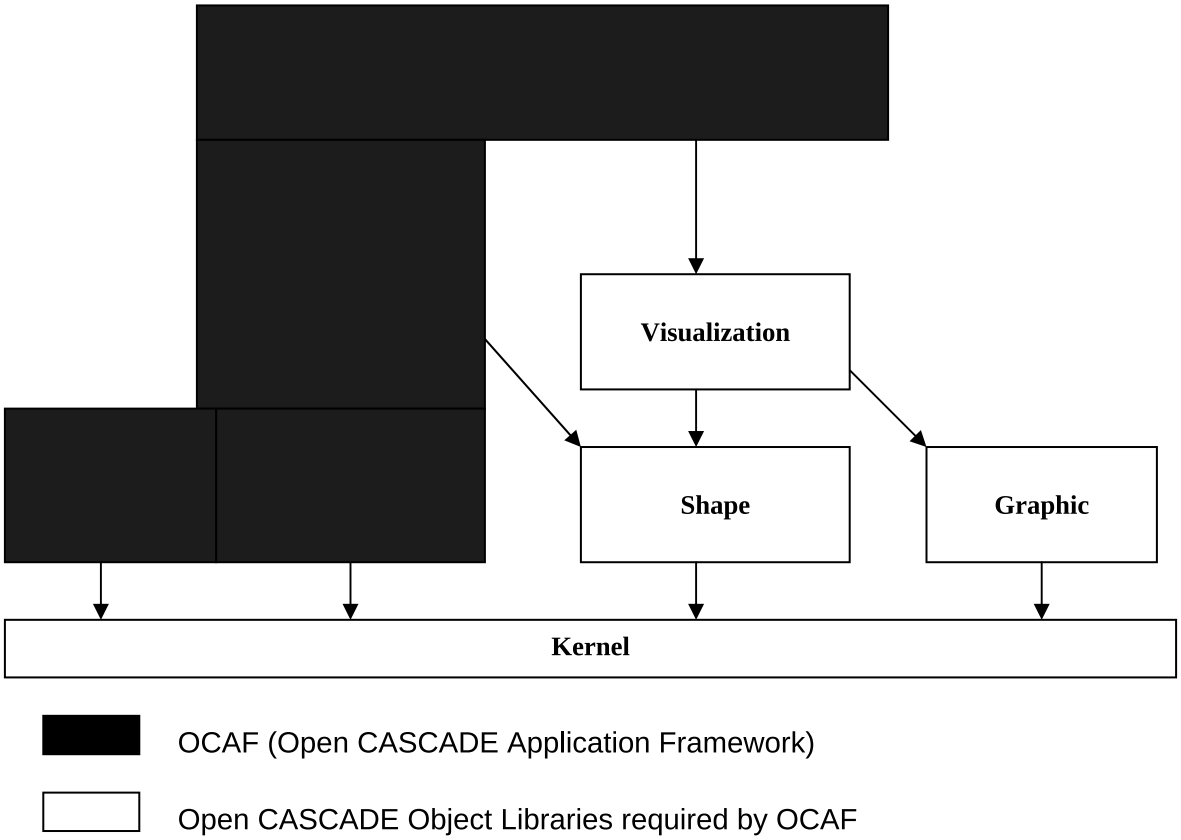

Visualization
Shape
Graphic
Kernel
OCAF (Open CASCADE Application Framework)
Open CASCADE Object Libraries required by OCAF

Figure 1. Architecture of OCAF

The subsequent chapters of this document explain the concepts and show how to use the services of OCAF.

2. Basic Concepts

2. 1 Overview

In most existing geometric modeling systems, the data structure is shape driven. They usually use a brep model, where solids and surfaces are defined by a collection of entities such as faces, edges etc., and by attributes such as application data. These attributes are attached to the entities. Examples of application data include:

- color,
- material,
- information that a particular edge is blended.

A shape, however, is inevitably tied to its underlying geometry. And geometry is highly subject to change in applications such as parametric modeling or product development. In this sort of application, using a brep (boundary representation) data structure proves to be a not very effective solution. A solution other than the shape must be found, i.e. a solution where attributes are attached to a deeper invariant structure of the model. Here, the topology itself will be one attribute among many.

In OCAF, data structure is reference key-driven. The reference key is implemented in the form of labels. Application data is attached to these labels as attributes. By means of these labels and a tree structure they are organized in, the reference key aggregates all user data, not just shapes and their geometry. These attributes have similar importance; no attribute is master in respect of the others.

The reference keys of a model - in the form of labels - have to be kept together in a single container. This container is called a document.

topology-driven approach **reference key-driven approach**

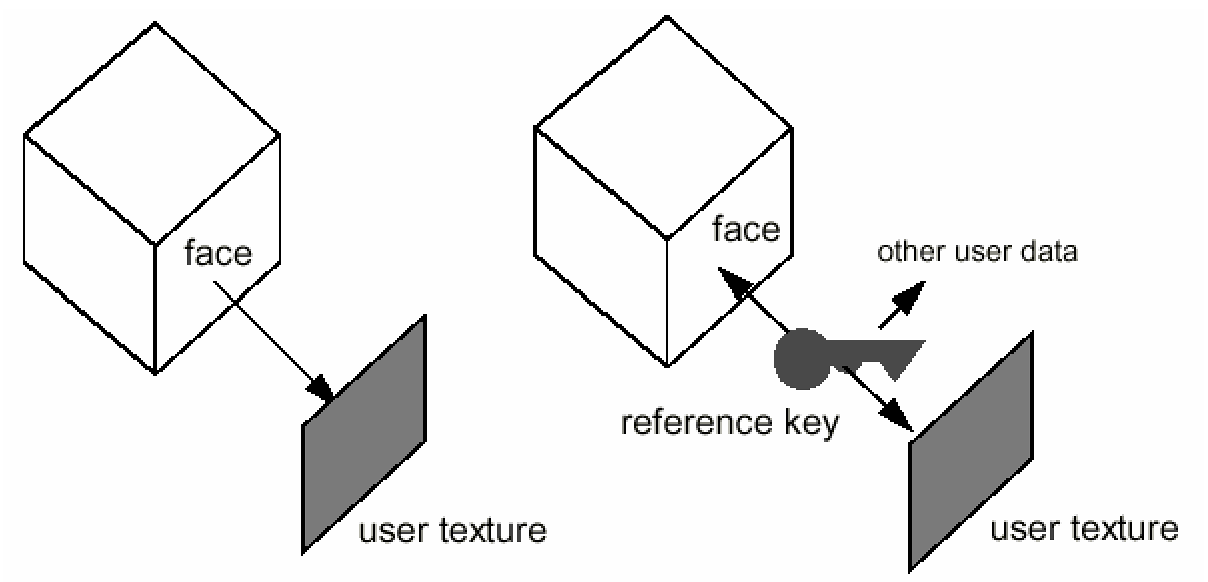


Figure 2. Topology-driven vs. reference key-driven approaches

2. 2 Applications and documents

OCAF documents are in turn managed by an OCAF application, which is in charge of:

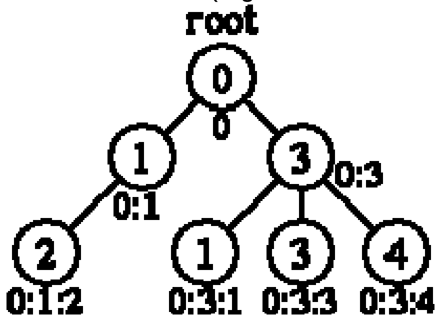
- Creating new documents
- Saving documents and opening them
- Initializing document views.

Apart from their role as a container of application data, documents can refer to each other; Document A, for example, can refer to a specific label in Document B. This functionality is made possible by means of the reference key.

2. 3 The document and the data framework

Inside a document, there is a data framework, a model, for example. This is a set of labels organized in a tree structure characterized by the following features:

- The first label in a framework is the root of the tree
- Each label has a tag expressed as an integer value
- Sub-labels of a label are called its children
- Each label which is not the root has one father – label from an upper level of the framework
- Labels which have the same father are called brothers
- Brothers cannot share the same tag
- A label is uniquely defined by an entry expressed as a list of tags (entry) of fathers from the root: this list of tags is written from right to left: tag of label, tag of its father, tag of father of its father,..., 0 (tag of the root label)



For example, in the above figure, a simple framework model is represented. Inside the circles are the tags of corresponding labels. Under the circles are lists of tags. The root label always has zero tag.

Children of the root label are middle-level labels with tags 1 and 3. These labels are brothers.

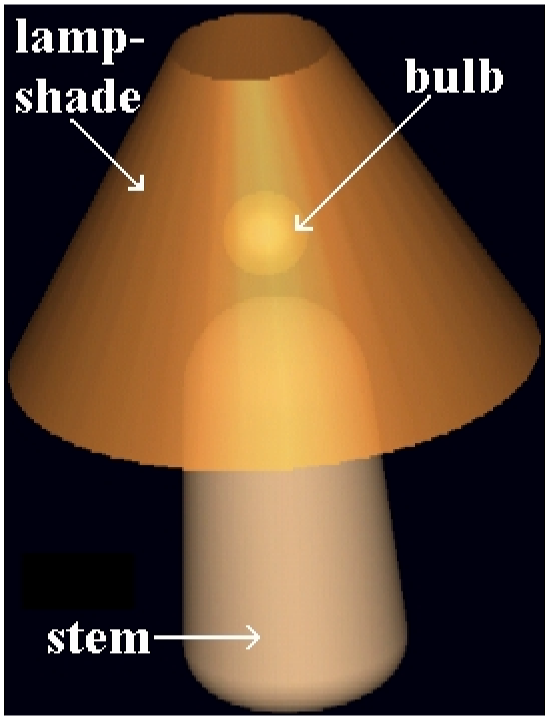
List of tags of the right-bottom label is «0:3:4»: this label has tag 4, its father (with entry «0:3») has tag 3, father of father has tag 0 (the root label always has «0» entry).

For example, an application for designing table lamps will first allocate a label for the lamp unit (the lamp is illustrated below). The root label never has brother labels, so, for a lot of lamps in the framework allocation, one of the root label sub-labels for the lamp unit is used. By doing so, you would avoid any confusion between table lamps in the data framework. Parts of the lamp have different material, color and other attributes, so, for each sub-unit of the lamp a child label of the lamp label with specified tags is allocated:

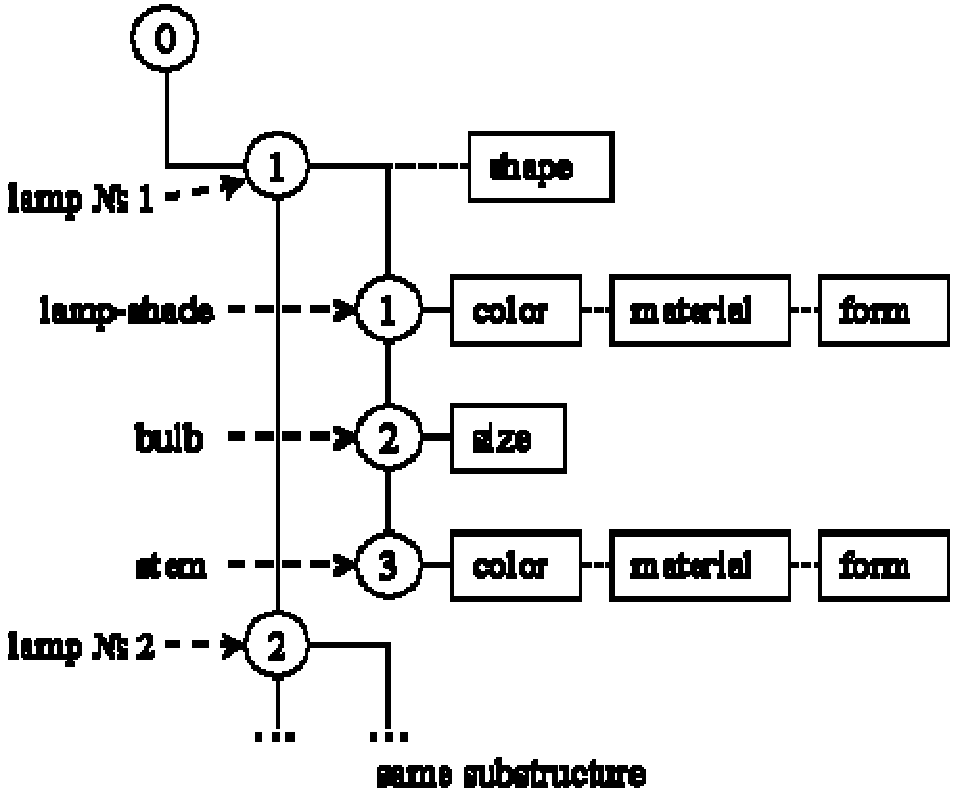
- a lamp-shade label with tag 1
- a bulb label with tag 2
- a stem label with tag 3

Label tags are chosen at will. They are just identifiers of the lamp parts. Now you can refine all units: set to the specified label geometry, color, material and other information about the lamp or it’s parts. This information is placed into special attributes of the label: the pure label contains no data – it is only a key to access data.

The thing to remember is that tags are private addresses without any meaning outside the data framework. It would, for instance, be an error to use part names as tags. These might change or be removed from production in next versions of the application, whereas the exact form of that part might be what you wanted to use in your design, the part name could be integrated into the framework as an attribute.



So, after the user changes the lamp design, only corresponding attributes are changed, but the label structure is maintained. Lamp shape must be recreated by new attribute values and attributes of the lamp shape must refer to a new shape.



The previous figure shows the table-lamps document structure: each child of the root label contains a lamp shape attribute and refers to the sub-labels, which contain some design information about corresponding sub-units.

The data framework structure allows to create more complex structures: each lamp label sub-label may have children labels with more detailed information about parts of the table lamp and its components.

Note that the root label can have attributes too, usually global attributes: the name of the document, for example.

As in the case of the table lamp example above, OCAF documents aggregate a battery of ready-to-use attributes, which represent typical data used in CAD. This data includes not only the Shape attribute, but a wide range of Standard attributes corresponding to the following types (see paragraph 6):

- Geometric attributes
- General attributes
- Relationship attributes
- Auxiliary attributes

2. 3. 1 Documents

Documents offer access to the data framework and manage the following items:

- Manage the notification of changes
- Update external links

- Manage the saving and restoring of data
- Store the names of software extensions.
- Manage command transactions
- Manage Undo and Redo options.

2. 3. 2 Shape attribute

The shape attribute implements the functionality of the Open CASCADE topology manipulation:

- reference to the shapes
- tracking of shape evolution

2. 3. 3 Standard attributes

Several ready-to-use base attributes already exist. These allow operating with simple common data in the data framework (for example: integer, real, string, array kinds of data), realize auxiliary functions (for example: tag sources attribute for the children of the label counter), create dependencies (for example: reference, tree node)....

2. 3. 4 Visualization attributes

These attributes allow placing viewer information to the data framework, visual representation of objects and other auxiliary visual information, which is needed for graphical data representation.

2. 3. 5 Function services

Where the document manages the notification of changes, a function manages propagation of these changes. The function mechanism provides links between functions and calls to various algorithms.

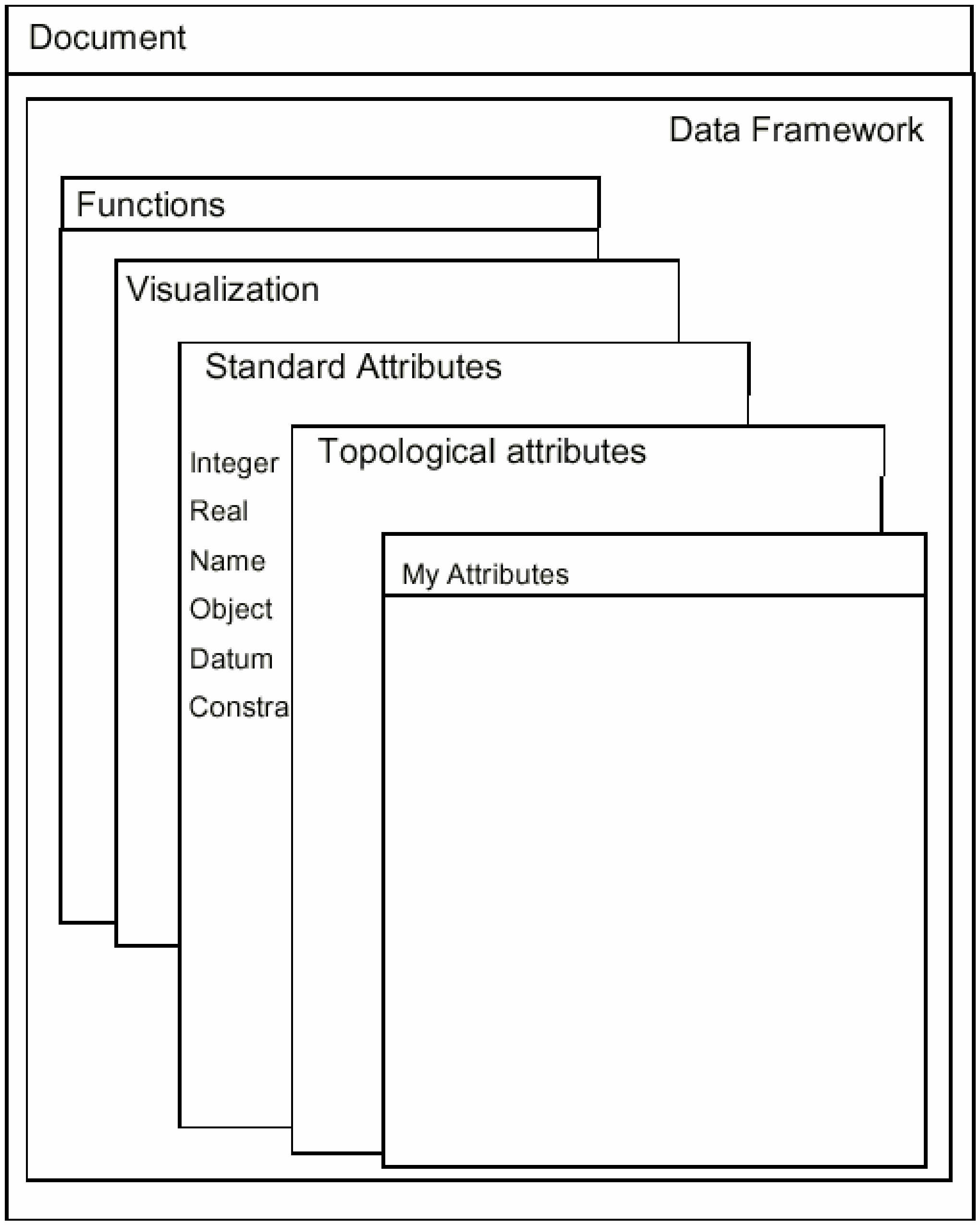


Figure 3. Document structure

3. Data Framework Services

3. 1 Overview

The data framework offers a single environment in which data from different application components can be handled.

This allows you to exchange and modify data simply, consistently, with a maximum level of information, and with stable semantics.

The building blocks of this approach are:

- The tag
- The label
- The attribute

As it has been mentioned earlier, the first label in a framework is the root label of the tree. Each label has a tag expressed as an integer value, and a label is uniquely defined by an entry expressed as a list of tags from the root, 0:1:2:1, for example.

Each label can have a list of attributes, which contain data, and several attributes can be attached to a label. Each attribute is identified by a GUID, and although a label may have several attributes attached to it, it must not have more than one attribute of a single GUID.

The sub-labels of a label are called its children. Conversely, each label, which is not the root, has a father. Brother labels cannot share the same tag.

The most important property is that a label's entry is its persistent address in the data framework.

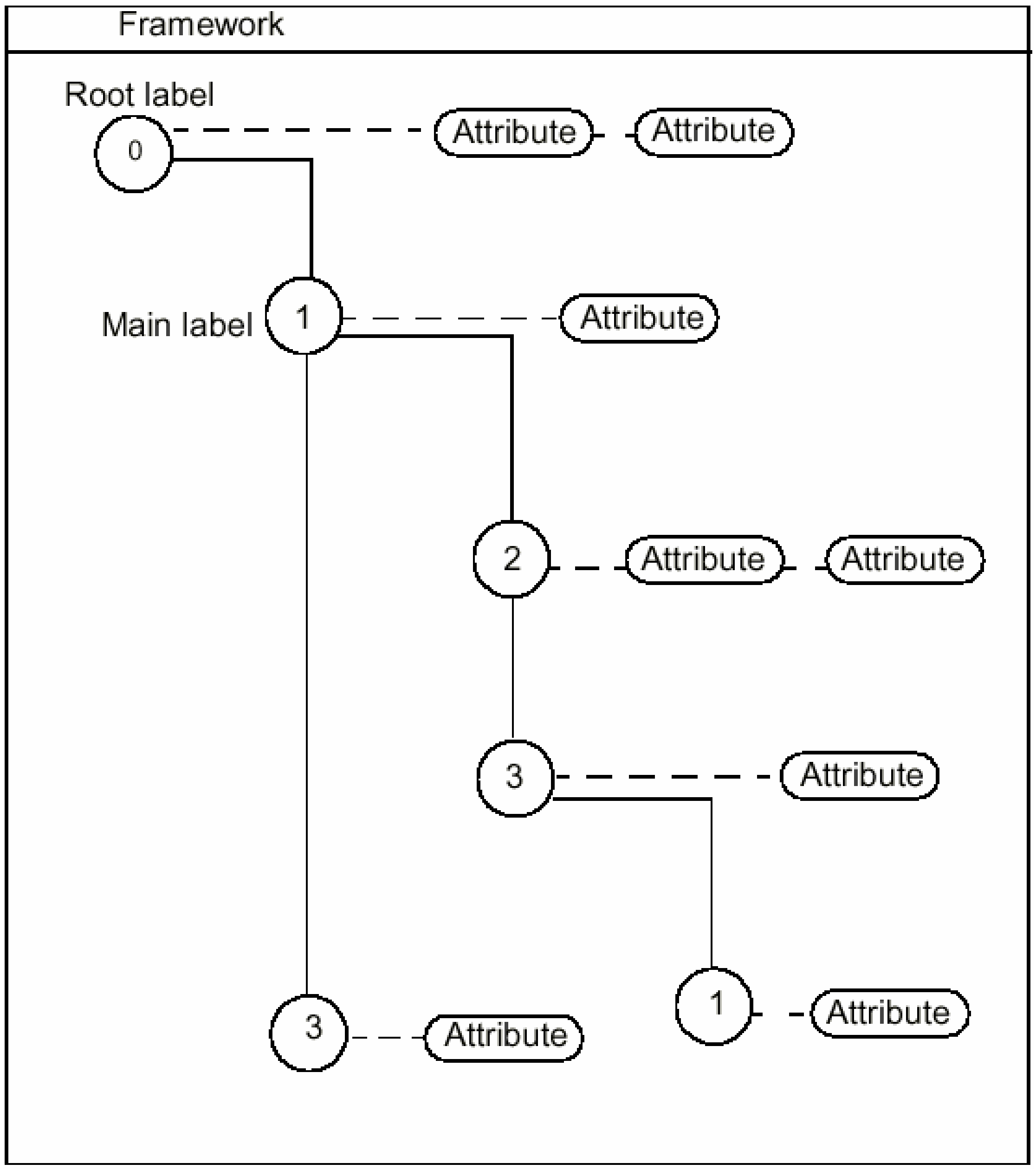


Figure 4. Contents of a document

3. 2 The Tag

A tag is an integer, which identifies a label in two ways:

- Relative identification
- Absolute identification.

In relative identification, a label's tag has a meaning relative to the father label only. For a specific label, you might, for example, have four child labels identified by the tags 2, 7, 18, 100. In using relative identification, you ensure that you have a safe scope for setting attributes.

In absolute identification, a label's place in the data framework is specified unambiguously by a colon-separated list of tags of all the labels from the one in question to the root of the data framework. This list is called an entry. TDF_Tool::TagList allows you to retrieve the entry for a specific label.

In both relative and absolute identification, it is important to remember that the value of an integer has no intrinsic semantics whatsoever. In other words, the natural sequence that integers suggest, i.e. 0, 1, 2, 3, 4 ... - has no importance here. The integer value of a tag is simply a key.

The tag can be created in two ways:

- Random delivery
- User-defined delivery

As the names suggest, in random delivery, the tag value is generated by the system in a random manner. In user-defined delivery, you assign it by passing the tag as an argument to a method.

3. 2. 1 Creating child labels using random delivery of tags

To append and return a new child label, you use TDF_TagSource::NewChild. In the example below, the argument *level2, which is passed to NewChild,* is a TDF_Label.

Example

```
TDF_Label child1 = TDF_TagSource::NewChild (level2);
TDF_Label child2 = TDF_TagSource::NewChild (level2);
```

3. 2. 2 Creation of a child label by user delivery from a tag

The other way to create a child label from a tag is by user delivery. In other words, you specify the tag, which you want your child label to have.

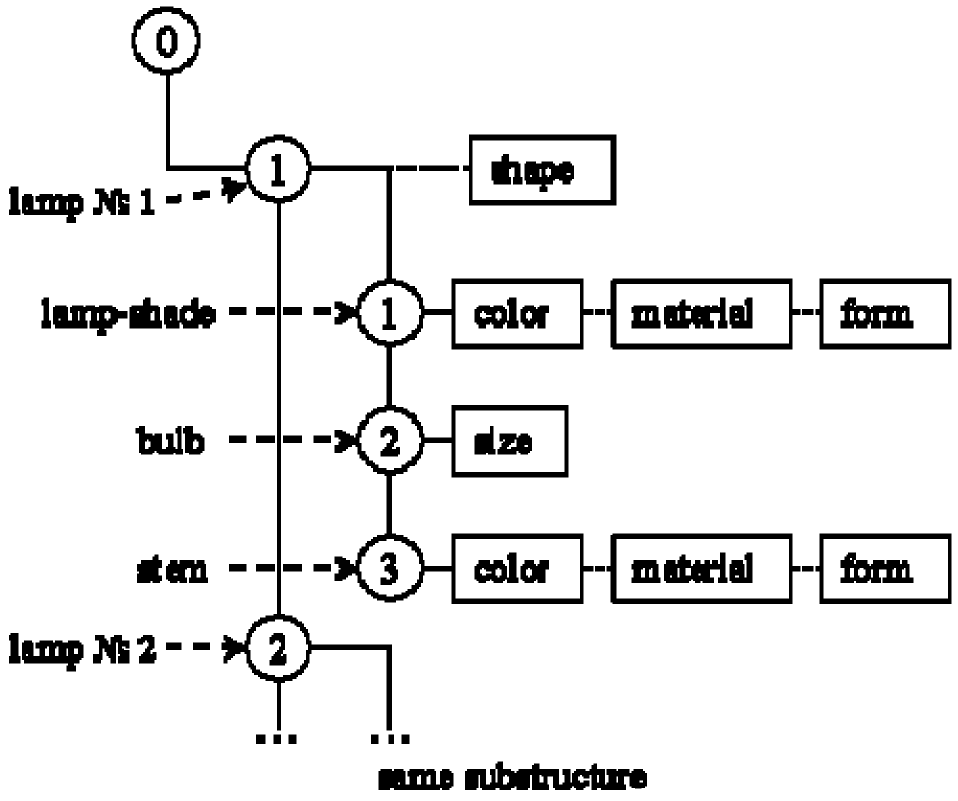
To retrieve a child label from a tag which you have specified yourself, you need to use TDF_Label::FindChild and TDF_Label::Tag as in the example below. Here, the integer 3 designates the tag of the label you are interested in, and the Boolean false is the value for the argument *create*. When this argument is set to *false*, no new child label is created.

Example

```
TDF_Label achild = root.FindChild(3,Standard_False);
if (!achild.IsNull()) {
Standard_Integer tag = achild.Tag();
}
```

3. 3 The Label

The tag gives a persistent address to a label. The label – the semantics of the tag – is a place in the data framework where attributes, which contain data, are attached. The data framework is, in fact, a tree of labels with a root as the ultimate father label (refer to the following figure):



Label can not be deleted from the data framework, so, the structure of the data framework that has been created can not be removed while the document is opened. Hence any kind of reference to an existing label will be actual while an application is working with the document.

3. 3. 1 Label creation

Labels can be created on any labels, compared with brother labels and retrieved. You can also find their depth in the data framework (depth of the root label is 0, depth of child labels of the root is 1 and so on), whether they have children or not, relative placement of labels, data framework of this label. It is the class TDF_Label that offers the above services.

3. 3. 2 Creating child labels

To create a new child label in the data framework using explicit delivery of tags, use TDF_Label::FindChild.

Example

```
//creating a label with tag 10 at Root
TDF_Label lab1 = aDF->Root().FindChild(10);
```

```
//creating labels 7 and 2 on label 10
TDF_Label lab2 = lab1.FindChild(7);

TDF_Label lab3 = lab1.FindChild(2);
```

You could also use the same syntax but add the Boolean *true* as a value of the argument "*create"*. This ensures that a new child label will be created if none is found. Note that in the previous syntax, this was also the case since *create* is *true* by default.

Example

```
TDF_Label level1 = root.FindChild(3,Standard_True);
TDF_Label level2 = level1.FindChild(1,Standard_True);
```

3. 3. 3 Retrieving child labels

You can retrieve child labels of your current label by iteration on the first level in the scope of this label.

Example

```
TDF_Label current;
//
for (TDF_ChildIterator it1 (current,Standard_False); it1.More(); it1.Next()) {
achild = it1.Value();
//
// do something on a child (level 1)
//
}
```

You can also retrieve all child labels in every descendant generation of your current label by iteration on all levels in the scope of this label.

Example

```
for (TDF_ChildIterator itall (current,Standard_True); itall.More(); itall.Next()) {
achild = itall.Value();
//
// do something on a child (all levels)
//
}
```

Using TDF_Tool::Entry with TDF_ChildIterator you can retrieve the entries of your current label's child labels as well.

Example

```
void DumpChildren(const TDF_Label& aLabel)
{
TDF_ChildIterator it;
TCollection_AsciiString es;
for (it.Initialize(aLabel,Standard_True); it.More(); it.Next()){
TDF_Tool::Entry(it.Value(),es);
cout << as.ToCString() << endl;
}
}
```

3. 3. 4 Retrieving the father label

Retrieving the father label of a current label.

Example

```
TDF_Label father = achild.Father();
isroot = father.IsRoot();
```

3. 4 The Attribute

The label itself contains no data. All data of any type whatsoever - application or non-application - is contained in attributes. These are attached to labels, and there are different types for different types of data. OCAF provides many ready-to-use standard attributes such as integer, real, constraint, axis and plane. There are also attributes for topological naming, functions and visualization. Each type of attribute is identified by a GUID.

The advantage of OCAF is that all of the above attribute types are handled in the same way. Whatever the attribute type is, you can create new instances of them, retrieve them, attach them to and remove them from labels, «forget» and «remember» the attributes of a particular label.

3. 4. 1 Retrieving an attribute from a label

To retrieve an attribute from a label, you use TDF_Label::FindAttribute. In the example below, the GUID for integer attributes, and *INT*, a handle to an attribute are passed as arguments to FindAttribute for the current label.

Example

```
if(current.FindAttribute
        (TDataStd_Integer::GetID(),INT))
{
    // the attribute is found
}
else {
    // the attribute is not found
}
```

3. 4. 2 Identifying an attribute using a GUID

You can create a new instance of an attribute and retrieve its GUID. In the example below, a new integer attribute is created, and its GUID is passed to the variable *guid* by the method ID inherited from TDF_Attribute.

Example

```
Handle(TDataStd_Integer) INT = new TDataStd_Integer();
Standard_GUID guid = INT->ID();
```

3. 4. 3 Attaching an attribute to a label

To attach an attribute to a label, you use TDF_Label::Add. Repetition of this syntax raises an error message because there is already an attribute with the same GUID attached to the current label.

TDF_Attribute::Label for *INT* then returns the label *attach* which *INT* is attached to.

Example

```
current.Add (INT); // INT is now attached to current
current.Add (INT); // causes failure
TDF_Label attach = INT->Label();
```

3. 4. 4 Testing the attachment to a label

You can test whether an attribute is attached to a label or not by using TDF_Attribute::IsA with the GUID of the attribute as an argument. In the example below, you test whether the current label has an integer attribute, and then, if that is so, how many attributes are attached to it. TDataStd_Integer::GetID provides the GUID argument needed by the method IsAttribute.

TDF_Attribute::HasAttribute tests whether there is an attached attribute, and TDF_Tool::NbAttributes returns the number of attributes attached to the label in question, *current* here,

Example

```
// Testing of attribute attachment
//
if (current.IsA(TDataStd_Integer::GetID())) {
    // the label has an Integer attribute attached
}
if (current.HasAttribute()) {
    // the label has at least one attribute attached
    Standard_Integer nbatt = current.NbAttributes();
    // the label has nbatt attributes attached
}
```

3. 4. 5 Removing an attribute from a label

To remove an attribute from a label, you use TDF_Label::Forget with the GUID of the deleted attribute. To remove all attributes of a label, TDF_Label::ForgetAll.

Example

```
current.Forget(TDataStd_Integer::GetID());
// integer attribute is now not attached to current label
current.ForgetAll();
// current has now 0 attributes attached
```

3. 4. 6 Specific attribute creation

If the set of existing and ready to use attributes implementing standard data types does not cover the needs of a specific data presentation task, the user can build his own data type and the corresponding new specific attribute implementing this new data type.

There are two ways to implement a new data type: create a new attribute (standard approach), or use the notion of User Attribute by means of a combination of standard attributes (alternative way)

In order to create a new attribute in the standard way do the following:

- Create a class inherited from TDF_Attribute and implement all purely virtual and necessary virtual methods:
    - **ID()** – returns a unique GUID of a given attribute
    - **Restore(attribute)** – sets fields of this attribute equal to the fields of a given attribute of the same type
    - **Paste(attribute, relocation_table)** – sets fields of a given attribute equal to the field values of this attribute ; if the attribute has references to some objects of the data framework and relocation_table has this element, then the given attribute must also refer to this object .
    - **NewEmpty() –** returns a new attribute of this class with empty fields
    - **Dump(stream) –** outputs information about a given attribute to a given stream debug (usually outputs an attribute of type string only)
- Create the persistence classes for this attribute according to the file format chosen for the document (see below).

Methods NewEmpty, Restore and Paste are used for the common transactions mechanism (Undo/Redo commands). If you don't need this attribute to react to undo/redo commands, you can write only stubs of these methods, else you must call the Backup method of the TDF_Attribute class every time attribute fields are changed.

If you use a standard file format and you want your new attributes to be stored during document saving and retrieved to the data framework whenever a document is opened, you must do the following:

- If you place an attribute to a new package, it is desirable (although not mandatory) if your package name starts with letter «T» (transient), for example: attribute TMyAttributePackage_MyAttribute in the package TMyAttributePackage
- Create a new package with name «P[package name]» (for example PMyAttributePackage) with class PMyAttributePackage_MyAttribute inside. The new class inherits the PDF_Attribute class and contains fields of attributes, which must be saved or retrieved («P» - persistent).
- Create a new package with name «M[package name]» (for example MMyAttributePackage) with classes MMyAttributePackage_MyAttributeRetrievalDriver and MMyAttributePackage_MyAttributeStorageDriver inside. The new classes inherit MDF_ARDriver and MDF_ASDriver classes respectively and contain the translation functionality: from T... attribute to P... and vice versa (M - middle) (see the realization of the standard attributes).
- M... package must contain AddStorageDrivers(aDriverSeq : ASDriverHSequence from MDF) and AddRetrievalDrivers(aDriverSeq : ASDriverHSequence from MDF) methods, which append to the given sequence <aDriverSeq> of drivers a sequence of all new attribute drivers (see the previous point), which will be used for the attributes storage/retrieval.
- Use the standard schema (StdSchema unit) or create a new one to add your P-package and compile it.

If you use the XML format, do the following:

- Create a new package with the name Xml[package name] (for example XmlMyAttributePackage) containing class XmlMyAttributePackage_MyAttributeDriver. The new class inherits XmlMDF_ADriver class and contains the translation functionality: from transient to persistent and vice versa (see the realization of the standard attributes in the packages XmlMDataStd, for example). Add package method AddDrivers which adds your class to a driver table (see below).

- Create a new package (or do it in the current one) with two package methods: Factory, which loads the document storage and retrieval drivers; and AttributeDrivers, which calls the methods AddDrivers for all packages responsible for persistence of the document.
- Create a plug-in implemented as an executable (see example XmlPlugin). It calls a macro PLUGIN with the package name where you implemented the method Factory.

If you use the binary format, do the following:

- Create a new package with name Bin[package name] (for example BinMyAttributePackage) containing a class BinMyAttributePackage_MyAttributeDriver. The new class inherits BinMDF_ADriver class and contains the translation functionality: from transient to persistent and vice versa (see the realization of the standard attributes in the packages BinMDataStd, for example). Add package method AddDrivers which adds your class to a driver table (see below).
- Create a new package (or do it in the current one) with two package methods: Factory, which loads the document storage and retrieval drivers; and AttributeDrivers, which calls the methods AddDrivers for all packages responsible for persistence of the document.
- Create a plug-in implemented as an executable (see example BinPlugin). It calls a macro PLUGIN with the package name where you implemented the method Factory.

See «Saving the document» on page 23 and «Opening the document from a file» on page 25 for the description of a document save/open mechanisms.

If you decided to use the alternative way (create a new attribute by means of UAttribute and a combination of other standard attributes), do the following:

- Set a TDataStd_UAttribute with a unique GUID attached to a label. This attribute defines the semantics of the data type (identifies the data type).
- Create child labels and allocate all necessary data through standard attributes at the child labels.
- Define an interface class for access to the data of the child labels.

Choosing the alternative way of implementation of new data types allows to forget about creating persistence classes for your new data type. Standard persistence classes will be used instead.Besides, this way allows separating the data and the methods for access to the data (interfaces). It can be used for rapid development in all cases when requirements to application performance are not very high.

Let's study the implementation of the same data type in both ways by the example of transformation represented by gp_Trsf class. The class gp_Trsf defines the transformation according to the type (gp_TrsfForm) and a set of parameters of the particular type of transformation (two points or a vector for translation, an axis and an angle for rotation, and so on).

1. The first way: creation of a new attribute. The implementation of the transformation by creation of a new attribute is represented in the Appendix.

2. The second way: creation of a new data type by means of combination of standard attributes. Depending on the type of transformation it may be kept in data framework by different standard attributes. For example, a translation is defined by two points. Therefore the data tree for translation looks like this:

- Type of transformation (gp_Translation) as TDataStd_Integer;
- First point as TDataStd_RealArray (three values: X1, Y1 and Z1);

- Second point as TDataStd_RealArray (three values: X2, Y2 and Z2).

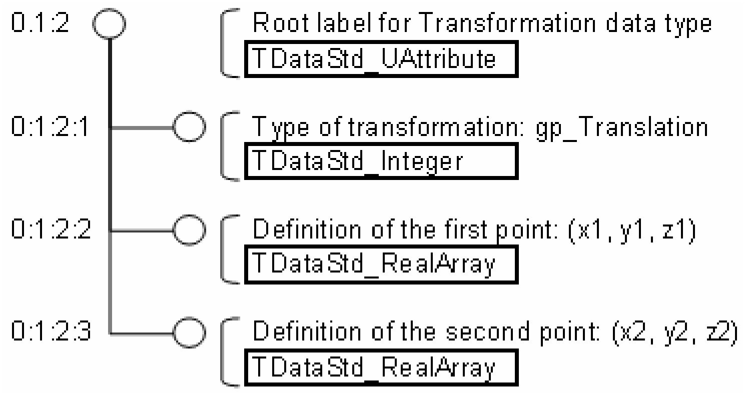

0:1:2
Root label for Transformation data type
TDataStd_UAttribute
0:1:2:1
Type of transformation: gp_Translation
TDataStd_Integer
0:1:2:2
Definition of the first point: (x1, y1, z1)
TDataStd_RealArray
0:1:2:3
Definition of the second point: (x2, y2, z2)
TDataStd_RealArray

Figure 5. Data tree for translation

If the type of transformation is changed to rotation, the data tree looks like this:

- Type of transformation (gp_Rotation) as TDataStd_Integer;
- Point of axis of rotation as TDataStd_RealArray (three values: X, Y and Z);
- Axis of rotation as TDataStd_RealArray (three values: DX, DY and DZ);
- Angle of rotation as TDataStd_Real.

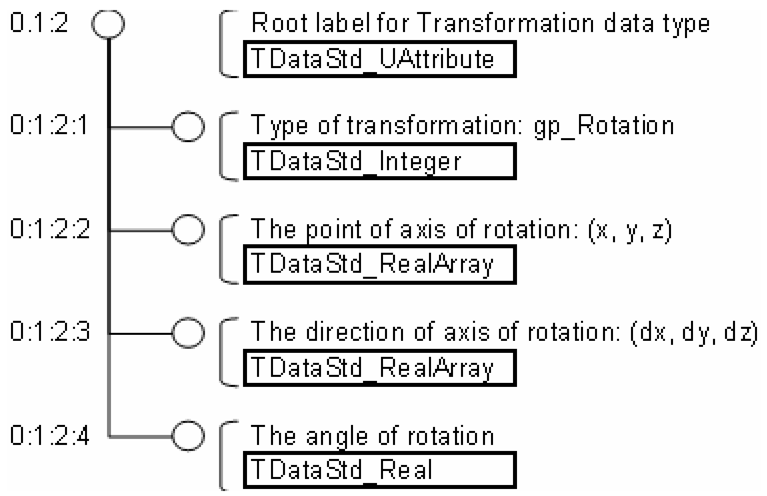

0:1:2
Root label for Transformation data type
TDataStd_UAttribute
0:1:2:1
Type of transformation: gp_Rotation
TDataStd_Integer
0:1:2:2
The point of axis of rotation: (x, y, z)
TDataStd_RealArray
0:1:2:3
The direction of axis of rotation: (dx, dy, dz)
TDataStd_RealArray
0:1:2:4
The angle of rotation
TDataStd_Real

Figure 6. Data tree for rotation

The attribute TDataStd_UAttribute with the chosen unique GUID identifies the data type. The interface class initialized by the label of this attribute allows access to the data container (type of transformation and the data of transformation according to the type).

4. Standard Document Services

4. 1 Overview

Standard documents offer ready-to-use documents containing a TDF-based data framework. Each document can contain only one framework.

The documents themselves are contained in the instantiation of a class inheriting from TDocStd_Application. This application manages the creation, storage and retrieval of documents.

You can implement undo and redo in your document, and refer from the data framework of one document to that of another one. This is done by means of external link attributes, which store the path and the entry of external links.

To sum up, standard documents alone provide access to the data framework. They also allow you to:

- Update external links
- Manage the saving and opening of data
- Manage the undo/redo functionality.

4. 2 The Application

As a container for your data framework, you need a document, and your document must be contained in your application. This application will be a class inheriting from TDocStd_Application.

4. 2. 1 Creating an application

To create an application, use the following syntax.

Example

```
Handle(TDocStd_Application) app
        = new MyApplication_Application ();
```

Note that MyApplication_Application is a class, which you have to create and which will inherit from TDocStd_Application.

4. 2. 2 Creating a new document

To the application which you declared in the previous example (4.2.1), you must add the document *doc* as an argument of TDocStd_Application::NewDocument.

Example

```
Handle(TDocStd_Document) doc;
app->NewDocument("NewDocumentFormat", doc);
```

4. 2. 3 Retrieving the application to which the document belongs

To retrieve the application containing your document, you use the syntax below.

Example

```
app = Handle(TDocStd_Application)::DownCast
        (doc->Application());
```

4. 3 The Document

The document contains your data framework, and allows you to retrieve this framework, recover its main label, save it in a file, and open or close this file.

4. 3. 1 Accessing the main label of the framework

To access the main label in the data framework, you use TDocStd_Document::Main as in the example below. The main label is the first child of the root label in the data framework, and has the entry 0:1.

Example

```
TDF_Label label = doc->Main();
```

4. 3. 2 Retrieving the document from a label in its framework

To retrieve the document from a label in its data framework, you use TDocStd_Document::Get as in the example below. The argument *label* passed to this method is an instantiation of TDF_Label.

Example

```
doc = TDocStd_Document::Get(label);
```

4. 3. 3 Saving the document

If in your document you use only standard attributes (from the packages TDF, TDataStd, TNaming, TFunction, TPrsStd and TDocStd), you just do the following steps:

- In your application class (which inherits class TDocStd_Application) implement two methods:
    - Formats (TColStd_SequenceOfExtendedString& theFormats), which append to a given sequence <theFormats> your document format string, for example, «NewDocumentFormat» – this string is also set in the document creation command
    - ResourcesName(), which returns a string with a name of resources file (this file contains a description about the extension of the document, storage/retrieval drivers GUIDs...), for example, «NewFormat»
- Create the resource file (with name, for example, «NewFormat») with the following strings:

Example

```
formatlist:NewDocumentFormat
```

```
NewDocumentFormat: New Document Format Version 1.0
NewDocumentFormat.FileExtension: ndf
NewDocumentFormat.StoragePlugin: bd696000-5b34-11d1-b5ba-00a0c9064368
NewDocumentFormat.RetrievalPlugin: bd696001-5b34-11d1-b5ba-00a0c9064368
NewDocumentFormatSchema: bd696002-5b34-11d1-b5ba-00a0c9064368
NewDocumentFormat.AttributeStoragePlugin:57b0b826-d931-11d1-b5da-00a0c9064368
NewDocumentFormat.AttributeRetrievalPlugin:57b0b827-d931-11d1-b5da-00a0c9064368
```

- Create the resource file «Plugin» with GUIDs and corresponding plugin libraries, which looks like this:

Example

```
! Description of available plugins
! ********************************
!
b148e300-5740-11d1-a904-080036aaa103.Location: libFWOSPlugin.so
!
! standard document drivers plugin
!
bd696000-5b34-11d1-b5ba-00a0c9064368.Location: libPAppStdPlugin.so
bd696001-5b34-11d1-b5ba-00a0c9064368.Location: libPAppStdPlugin.so
!
!
! standard schema plugin
!
bd696002-5b34-11d1-b5ba-00a0c9064368.Location: libPAppStdPlugin.so
!
! standard attribute drivers plugin
!
57b0b826-d931-11d1-b5da-00a0c9064368.Location: libPAppStdPlugin.so
57b0b827-d931-11d1-b5da-00a0c9064368.Location: libPAppStdPlugin.so
```

In order to set the paths for these files it is necessary to set the environments: CSF_PluginDefaults and CSF_NewFormatDefaults. For example let's set the files in the directory "MyApplicationPath/MyResources":

```
setenv CSF_PluginDefaults MyApplicationPath/MyResources
setenv CSF_NewFormatDefaults MyApplicationPath/MyResources
```

Once these steps are taken you may run your application, create documents and Save/Open them. These resource files already exist in the OCAF (format «Standard»).

If you use your specific attributes from packages, for example, P-, M- and TMyAttributePackage, see «Specific attribute creation» on page 20; you must take some additional steps for the new plugin implementation:

- Add our «P» package to the standard schema. You can get an already existing (in Open CASCADE sources) schema from StdSchema unit and add your package string to the cdl-file: «package PMyAttributePackage».

- Next step consists of implementation of an executable, which will connect our documents to our application and open/save them. Copy the package PAppStdPlugin and change its name to MyTheBestApplicationPlugin. In the PLUGIN macros type the name of your factory which will be defined in the next step.
- Factory is a method, which returns drivers (standard drivers and our defined drivers from the "M" package) by a GUID. Copy the package where the standard factory is defined (it is PAppStd in the OCAF sources). Change its name to

MyTheBestSchemaLocation. The Factory() method of the PappStd package checks the GUID set as its argument and returns the corresponding table of drivers. Set two new GUIDs for your determined storage and retrieval drivers. Append two "if" declarations inside the Factory() method which should check whether the set GUID coincides with GUIDs defined by the Factory() method as far as our storage and retrieval drivers are concerned. If the GUID coincides with one of them, the method should return a table of storage or retrieval drivers respectively.

- Recompile all. Add the strings with GUIDs – in accordance with your plugin library GUID - to the «Plugin» file.

4. 3. 4 Opening the document from a file

To open the document from a file where it has been previously saved, you use TDocStd_Application::Open as in the example below. The arguments are the path of the file and the document saved in this file.

Example

```
app->Open("/tmp/example.caf", doc);
```

4. 4 External Links

External links refer from one document to another. They allow you to update the copy of data framework later on.

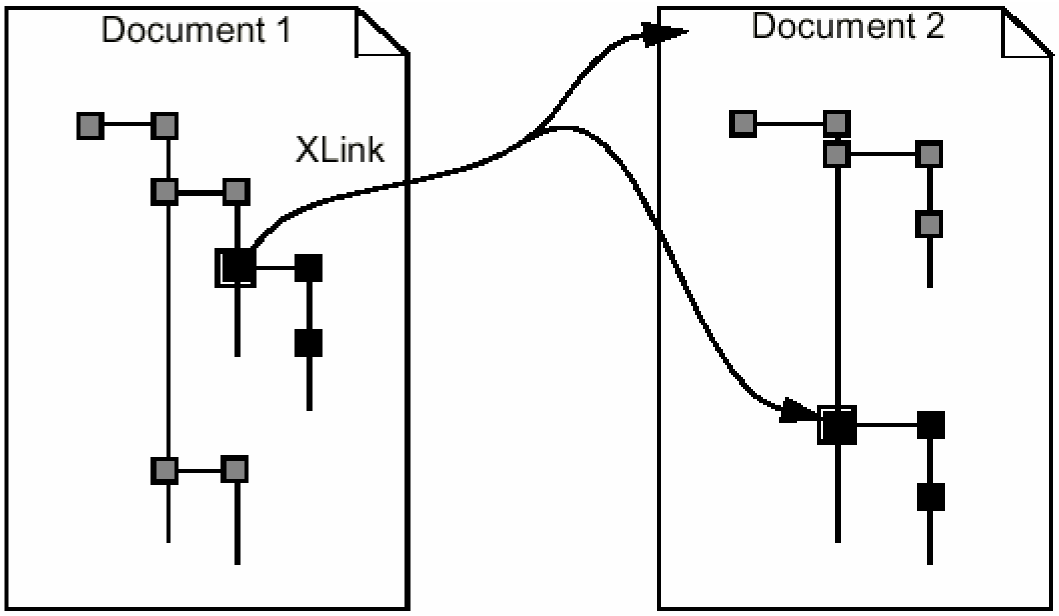


Figure 7. External links between documents

Note that documents can be copied with or without a possibility of updating an external link.

4. 4. 1 Copying the document

With a possibility of updating it later

To copy a document with a possibility of updating it later, you use TDocStd_XLinkTool::CopyWithLink.

Example

```
Handle(TDocStd_Document) doc1;
Handle(TDocStd_Document) doc2;

TDF_Label source = doc1->GetData()->Root();
TDF_Label target = doc2->GetData()->Root();
TDocStd_XLinkTool XLinkTool;

XLinkTool.CopyWithLink(target,source);
```

Now the target document has a copy of the source document. The copy also has a link in order to update the content of the copy if the original changes.

In the example below, something has changed in the source document. As a result, you need to update the copy in the target document. This copy is passed to TDocStd_XLinkTool::UpdateLink as the argument *target*.

Example

```
XLinkTool.UpdateLink(target);
```

With no link between the copy and the original

You can also create a copy of the document with no link between the original and the copy. The syntax to use this option is TDocStd_XLinkTool::Copy; the copied document is again represented by the argument *target*, and the original – by *source.*

Example

```
XLinkTool.Copy(target, source);
```

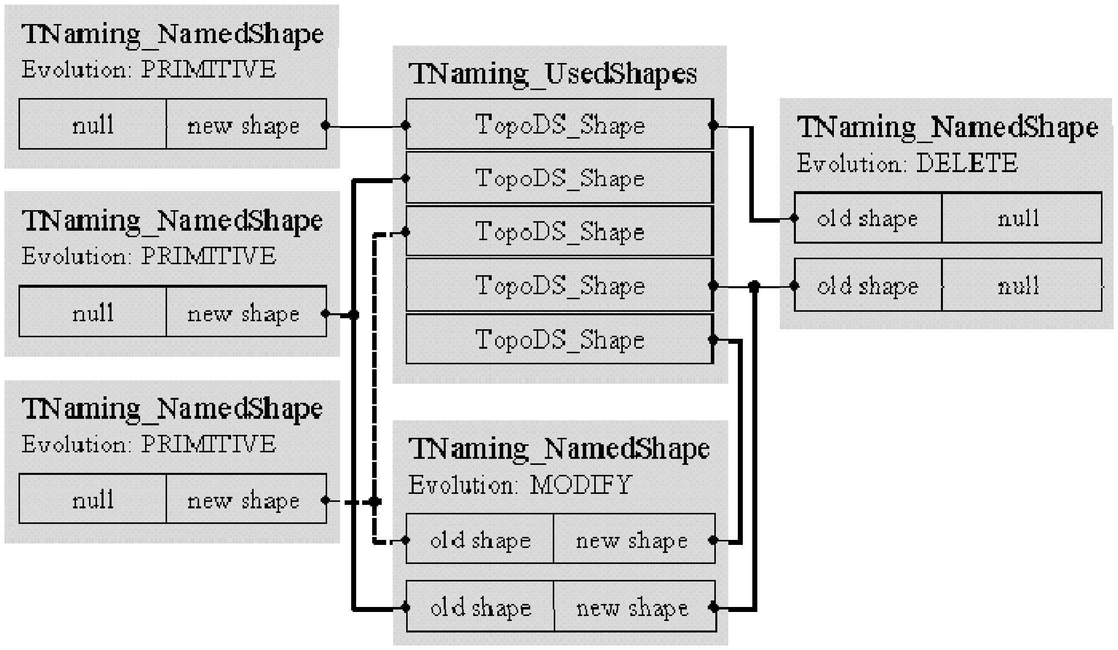
5. OCAF Shape Attributes

5. 1 Overview

OCAF shape attributes are used for topology objects and their evolution access. All topological objects are stored in one TNaming_UsedShapes attribute at the root label of the data framework. This attribute contains a map with all topological shapes used in a given document.

The user can add the TNaming_NamedShape attribute to other labels. This attribute contains references (hooks) to shapes from the TNaming_UsedShapes attribute and an evolution of these shapes. The TNaming_NamedShape attribute contains a set of pairs of hooks: to the "Old" shape and to a "New" shape (see the following figure). It allows not only to get the topological shapes by the labels, but also to trace the evolution of the shapes and to correctly update dependent shapes by the changed one.

If a shape is newly created, then the old shape of a corresponding named shape is an empty shape. If a shape is deleted, then the new shape in this named shape is empty.

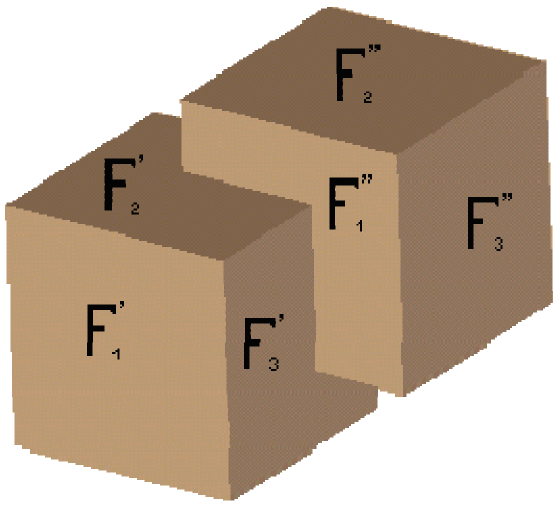


Different algorithms may dispose sub-shapes of the result shape at the individual labels depending on whether it is necessary to do so:

- If a sub-shape must have some extra attributes (material of each face or color of each edge). In this case a specific sub-shape is placed to a separate label (usually to a sub-label of the result shape label) with all attributes of this sub-shape.
- If the topological naming algorithm is needed, a necessary and sufficient set of sub-shapes is placed to child labels of the result shape label. As usual, for a basic solid and closed shells, all faces of the shape are disposed.

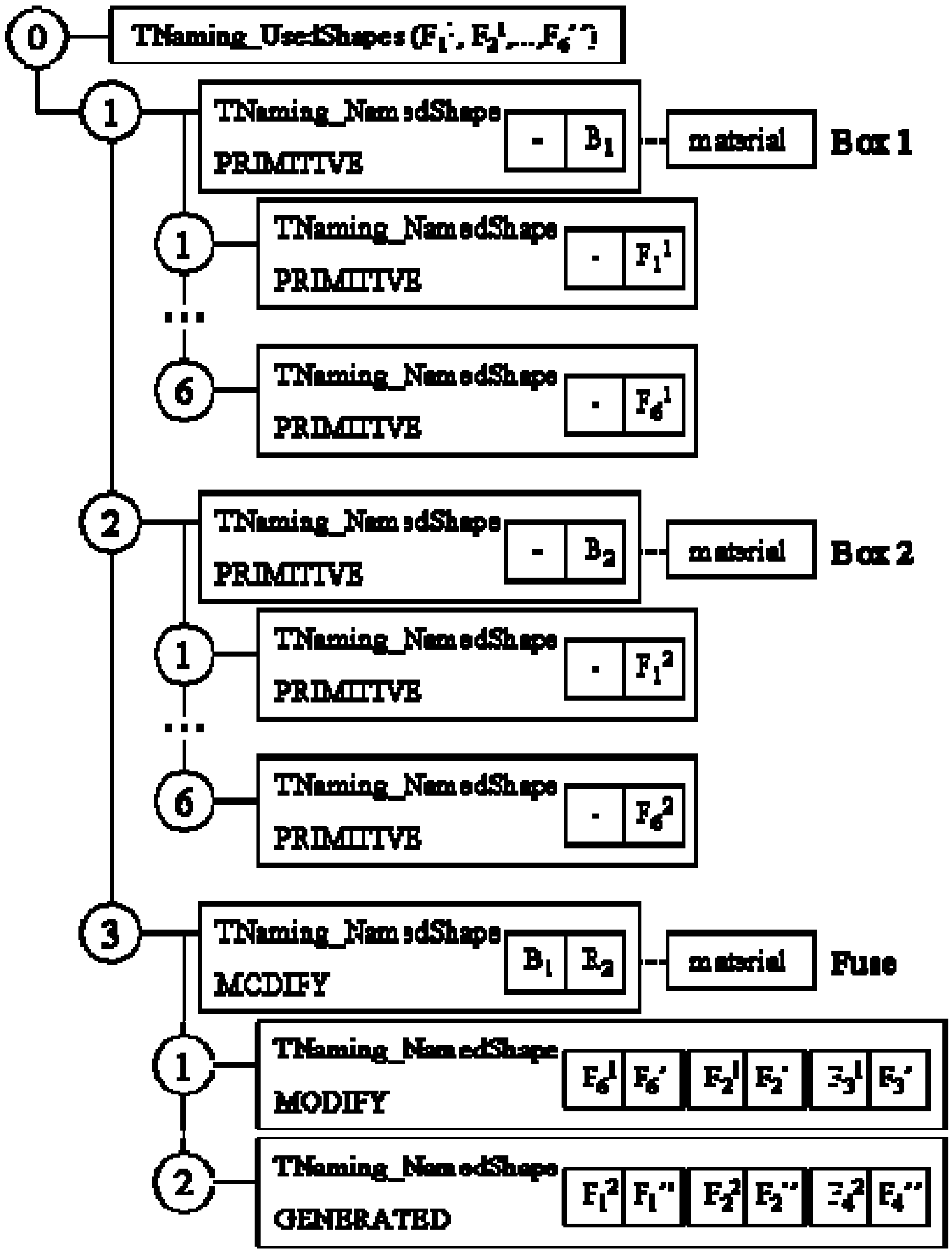
TNaming_NamedShape may contain a few pairs of hooks with the same evolution. In this case the topology shape, which belongs to the named shape is a compound of new shapes.

Consider the following example. Two boxes (solids) are fused into one solid (the result one). Initially each box was placed to the result label as a named shape, which has evolution PRIMITIVE and refers to the corresponding shape of the TNaming_UsedShapes map. The box result label has a material attribute and six child labels containing named shapes of Box faces.



After the fuse operation a modified result is placed to a separate label as a named shape, which refers to the old shape – one of the boxes, as well as to the new shape – the shape resulting from the fuse operation – and has evolution MODIFY (see the following figure).

Named shapes, which contain information about modified faces, belong to the fuse result sub-labels: sub-label with tag 1 – modified faces of the first box, sub-label with tag 2 – generated faces of the box 2.



This is necessary and sufficient information for the functionality of the right naming mechanism: any sub-shape of the result can be identified unambiguously by name type and set of labels, which contain named shapes:

- face F1' as a modification of F_1^1 face
- face F1'' as generation of F_1^2 face
- edges as an intersection of two contiguous faces
- vertices as an intersection of three contiguous faces

After any modification of source boxes the application must automatically rebuild the naming entities: recompute the named shapes of the boxes (solids and faces) and fuse the resulting named shapes (solids and faces) that reference to the new named shapes.

5. 2 Services provided

5. 2. 1 Registering shapes and their evolution

When using TNaming_NamedShape to create attributes, the following fields of an attribute are filled:

- A list of shapes called the «old» and the «new» shapes A new shape is recomputed as the value of the named shape. The meaning of this pair depends on the type of evolution.
- The type of evolution: a term of the TNaming_Evolution enumeration:
    - PRIMITIVE – newly created topology, with no previous history
    - GENERATED – as usual, this evolution of a named shape means, that the new shape is created from a low-level old shape (a prism face from an edge, for example)
    - MODIFY – the new shape is a modified old shape
    - DELETE – the new shape is empty; the named shape with this evolution just indicates that the old shape topology is deleted from the model
    - SELECTED – a named shape with this evolution has no effect on the history of the topology; it is

used for the selected shapes that are placed to the separate label

Only pairs of shapes with equal evolution can be stored in one named shape.

5. 2. 2 Using naming resources

The class TNaming_Builder allows you to create a named shape attribute. It has a label of a future attribute as an argument of the constructor. Respective methods are used for the evolution and setting of shape pairs. If for the same TNaming_Builder object a lot of pairs of shapes with the same evolution are given, then these pairs would be placed in the resulting named shape. After the creation of a new object of the TNaming_Builder class, an empty named shape is created at the given label.

Example

```
// a new empty named shape is created at «label»
TNaming_Builder builder(label);
// set a pair of shapes with evolution GENERATED
builder.Generated(oldshape1,newshape1);
// set another pair of shapes with the same evolution
builder.Generated(oldshape2,newshape2);
// get the result – TNaming_NamedShape attribute
Handle(TNaming_NamedShape) ns = builder.NamedShape();
```

5. 2. 3 Reading the contents of a named shape attribute

You can use TNaming_NamedShape class to get evolution of this named shape (method TNaming_NamedShape::Evolution()) and «value» of the named shape – compound of new shapes of all pairs of this named shape (method TNaming_NamedShape::Get()).

More detailed information about the contents of the named shape or about the modification history of a topology can be obtained with the following:

- TNaming_Tool provides a common high-level functionality for access to the named shapes contents:
    - GetShape(Handle(TNaming_NamedShape)) method returns a compound of new shapes of the given named shape
    - CurrentShape(Handle(TNaming_NamedShape)) method returns a compound of the shapes – last modifications (latest versions) of the shapes from the given named shape
    - NamedShape(TopoDS_Shape,TDF_Label) method returns a named shape, which contains a given shape as a new shape. Given label is any label from the data framework – it just gives access to it
- TNaming_Iterator given access to the named shape hooks pairs.

Example

```
// create an iterator for a named shape
TNaming_Iterator iter(namedshape);
// iterate while some pairs are not iterated
while(iter.More()) {
    // get the new shape from the current pair
    TopoDS_Shape newshape = iter.NewShape();
    // get the old shape from the current pair
    TopoDS_Shape oldshape = iter.OldShape();
    // do something...

    // go to the next pair
    iter.Next();
}
```

5. 2. 4 Selection Mechanism

One of user interfaces for topological naming resources is the TNaming_Selector class. You can use this class to:

- Store a selected shape on a label
- Access the named shape
- Update this naming

Selector places a new named shape with evolution SELECTED to the given label. By the given context shape (main shape, which contains a selected sub-shape), its evolution and naming structure the selector creates a «name» of the selected shape – unique description how to find a selected topology.

After any modification of a context shape and updating of the corresponding naming structure, you must call the TNaming_Selector::Solve method. If the naming structure is right, then the selector automatically updates the selected shape in the corresponding named shape, else it fails.

5. 2. 5 Exploring shape evolution

The class TNaming_Tool provides a toolkit to read current data contained in the attribute.

If you need to create a topological attribute for existing data, use the method NamedShape.

Example

```
class MyPkg_MyClass
{
public: Standard_Boolean SameEdge
(const Handle(CafTest_Line)& L1,
const Handle(CafTest_Line)& L2);
};

Standard_Boolean CafTest_MyClass::SameEdge
(const Handle(CafTest_Line)& L1,
const Handle(CafTest_Line)& L2)
{
    Handle(TNaming_NamedShape) NS1 = L1->NamedShape();
    Handle(TNaming_NamedShape) NS2 = L2->NamedShape();
    return BRepTools::Compare(NS1,NS2);
}
```

6. Standard Attributes

6. 1 Overview

Several ready-to-use attributes already exist. These allow you to create and modify attributes for many basic data types, and are available in the packages TDataStd or TDF. Each attribute is one of four types:

- Geometric attributes
- General attributes
- Relationship attributes
- Auxiliary attributes

Available attributes include:

- **Geometric attributes**
    - **Axis** – simply identifies, that the concerned TNaming_NamedShape attribute with an axis shape inside belongs to the same label
    - **Constraint –** contains information about a constraint between geometries: used geometry attributes, type, value (if exists), plane (if exists), «is reversed», «is inverted» and «is verified» flags
    - **Geometry** - simply identifies, that the concerned TNaming_NamedShape attribute with a specified-type geometry belongs to the same label
    - **Plane** - simply identifies, that the concerned TNaming_NamedShape attribute with a plane shape inside belongs to the same label
    - **Point** - simply identifies, that the concerned TNaming_NamedShape attribute with a point shape inside belongs to the same label
    - **Shape** - simply identifies, that the concerned TNaming_NamedShape attribute belongs to the same label

- **General attributes**
    - **Comment** – contains a string – some comment for a given label (or attribute)
    - **Expression** – contains an expression string and a list of used variables attributes
    - **Integer** – contains an integer value
    - **IntegerArray** – contains an array of integer values
    - **Name –** contains a string – some name of a given label (or attribute)
    - **Real –** contains a real value
    - **RealArray –** contains an array of real values
    - **Relation –** contains a relation string and a list of used variables attributes
    - **Variable –** simply identifies, that a variable belongs to this label; contains the «is constraint» flag and a string of used units («mm», «m»...)
    - **UAttribute –** attribute with a user-defined GUID. As a rule, this attribute is used as a marker, which is independent of attributes at the same label (note, that attributes with the same GUIDs can not belong to the same label)
- **Relationship attributes**
    - **Reference** – contains reference to the label of its own data framework
    - **TreeNode** – this attribute allows to create an internal tree in the data framework; this tree consists of nodes with the specified tree ID; each node contains references to the father, previous brother, next brother, first child nodes and tree ID.
- **Auxiliary attributes**

- **Directory** – hi-level tool attribute for sub-labels management
- **TagSource –** this attribute is used for creation of new children: it stores the tag of the last-created child of the label and gives access to the new child label creation functionality.

All of these attributes inherit class TDF_Attribute, so, each attribute has its own GUID and standard methods for attribute creation, manipulation, getting access to the data framework.

6. 2 Services common to all attributes

6. 2. 1 Accessing GUIDs

To access the GUID of an attribute, you can use two methods:

- Method **GetID**: this is the static method of a class. It returns the GUID of any attribute, which is an object of a specified class (for example, TDataStd_Integer returns the GUID of an integer attribute). Only two classes from the list of standard attributes do not support these methods: TDataStd_TreeNode and TDataStd_Uattribute, because the GUIDs of these attributes are variable.
- Method **ID**: this is the method of an object of an attribute class. It returns the GUID of this attribute. Absolutely all attributes have this method: only by this identifier you can discern the type of an attribute.

6. 2. 2 Conventional Interface of Standard Attributes

It is usual to create standard named methods for the attributes:

- Method Set(label, [value]) – it is the static method, which allows to add an attribute to a given label. If an attribute is characterized by one value this method may set it.
- Method Get() returns the value of an attribute if it is characterized by one value.
- Method Dump(Standard_OStream) outputs debug information about a given attribute to a given stream.

7. Visualization Attributes

7. 1 Overview

Standard visualization attributes implement the Application Interactive Services (see Open CASCADE Visualization User's Guide) in the context of Open CASCADE Application Framework. Standard visualization attributes are AISViewer and Presentaion and belong to the TPrsStd package.

7. 2 Services provided

7. 2. 1 Defining an interactive viewer attribute

The class TPrsStd_AISViewer allows you to define an interactive viewer attribute. There may be only one such attribute per one data framework and it is always placed to the root label. So, it could be set or found by any label («access label») of the data framework. Nevertheless the default architecture can be easily extended and the user can manage several Viewers per one framework by himself.

To initialize the AIS viewer as in the example below, use method Find.

Example

```
// «access» is any label of the data framework
Handle(TPrsStd_AISViewer) viewer = TPrsStd_AISViewer::Find(access)
```

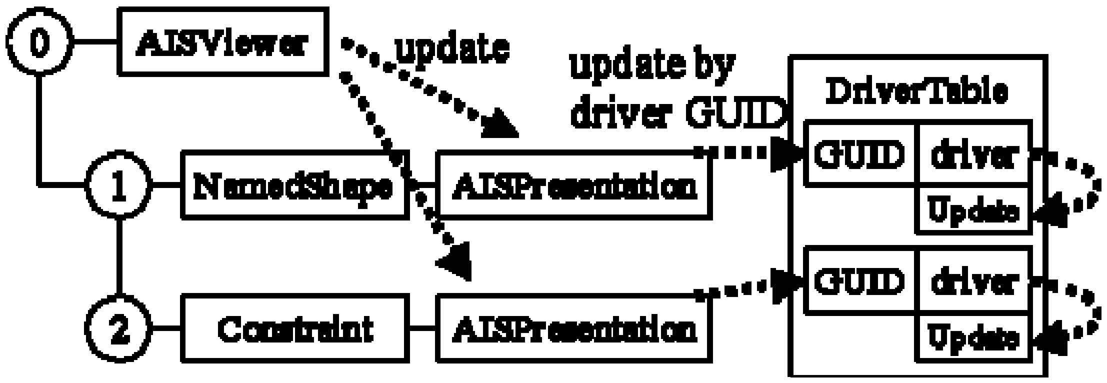
7. 2. 2 Defining a presentation attribute

The class TPrsStd_AISPresentation allows you to define the visual presentation of document labels contents. In addition to various visual fields (color, material, transparency, «isDisplayed», etc.), this attribute contains its driver GUID. This GUID defines the functionality, which will update the presentation every time when needed.

7. 2. 3 Creating your own driver

The abstract class TPrsStd_Driver allows you to define your own driver classes. Simply redefine the Update method in your new class, which will rebuild the presentation.

If your driver is placed to the driver table with the unique driver GUID, then every time the viewer updates presentations with a GUID identical to your driver's GUID, the Update method of your driver for these presentations must be called:



As usual, the GUID of a driver and the GUID of a displayed attribute are the same.

7. 2. 4 Using a container for drivers

You frequently need a container for different presentation drivers. The class TPrsStd_DriverTable provides this service. You can add a driver to the table, see if one is successfully added, and fill it with standard drivers.

To fill a driver table with standard drivers, first initialize the AIS viewer as in the example above, and then pass the return value of the method InitStandardDrivers to the driver table returned by the method Get. Then attach a TNaming_NamedShape to a label and set the named shape in the presentation attribute using the method Set. Then attach the presentation attribute to the named shape attribute, and the AIS_InteractiveObject, which the presentation attribute contains, will initialize its drivers for the named shape. This can be seen in the example below.

Example

```
DriverTable::Get() -> InitStandardDrivers();
// next, attach your named shape to a label
TPrsStd_AISPresentation::Set(NS};
// here, attach the AISPresentation to NS.
```

8. Function Services

8. 1 Overview

Function services aggregate data necessary for regeneration of a model. The function mechanism - available in the package TFunction - provides links between functions and any execution algorithms, which take their arguments from the data framework, and write their results inside the same framework.

When you edit any application model, you have to regenerate the model by propagating the modifications. Each propagation step calls various algorithms. To make these algorithms independent of your application model, you need to use function services.

Take, for example, the case of a modeling sequence made up of a box with the application of a fillet on one of its edges. If you change the height of the box, the fillet will need to be regenerated as well.

8. 2 Services provided

8. 2. 1 Finding functions, their owners and roots

The class TFunction_Function is an attribute, which stores a link to a function driver in the data framework. In the static table TFunction_DriverTable correspondence links between function attributes and drivers are stored.

You can write your function attribute, a driver for such attribute (which updates the function result in accordance to a given map of changed labels), and set your driver with the GUID to the driver table.

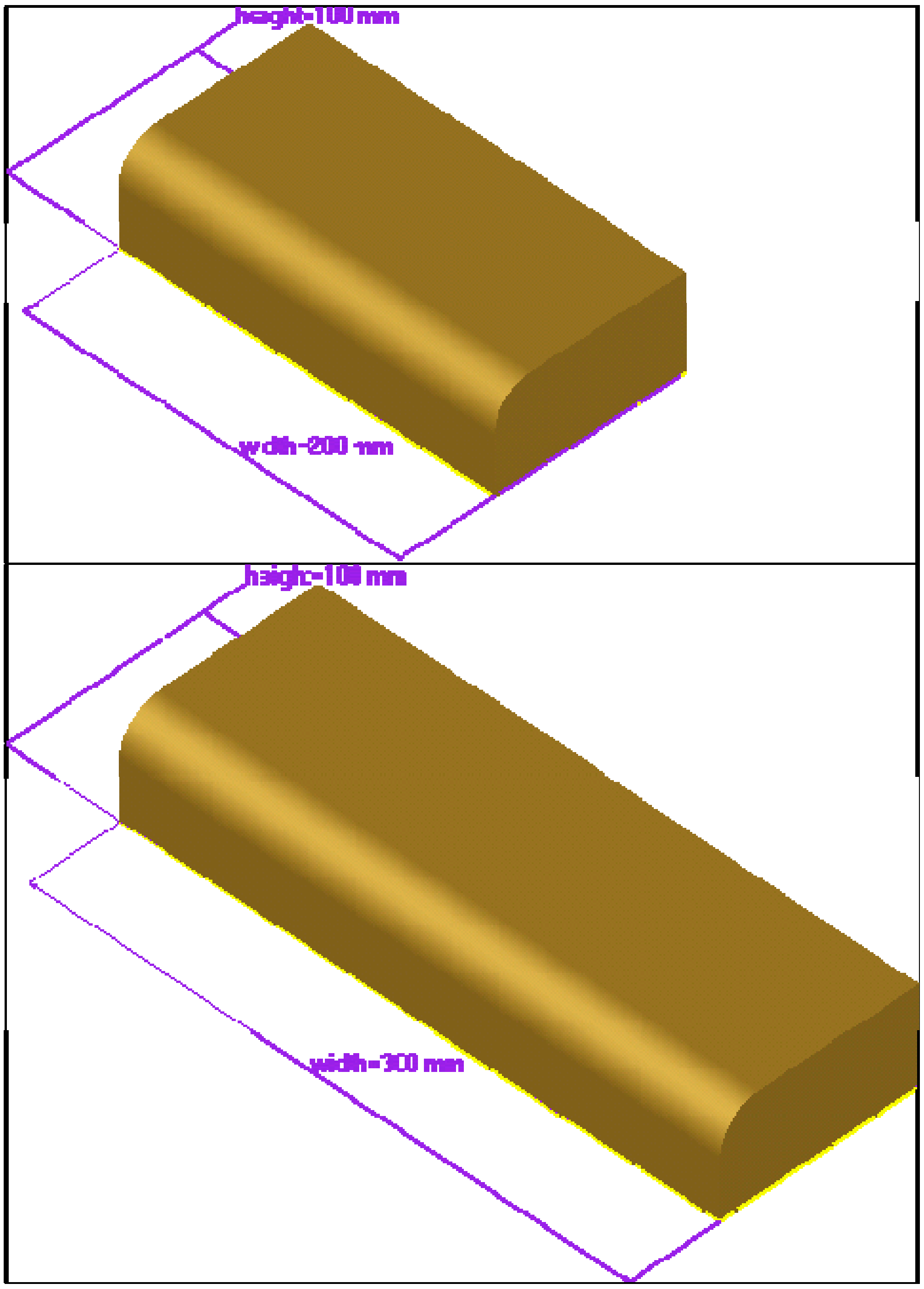
Then the solver algorithm of a data model can find the Function attribute on a corresponding label and call the Execute driver method to update the result of the function.

8. 2. 2 Storing and accessing information about function status

For updating algorithm optimization, each function driver has access to the TFunction_Logbook object that is a container for a set of touched, impacted and valid labels. Using this object a driver gets to know which arguments of the function were modified.

8. 2. 3 Propagating modifications

An application must implement its functions, function drivers and the common solver for parametric model creation. For example, let’s check the following model (see the following illustration):



- user creates a rectangular planar face F with height 100 and width 200
- user creates prism P using face F as a basis
- user creates fillet L at the edge of the prism
- user changes the width of F from 200 to 300:
    - the solver for the function of face F starts
    - the solver detects that an argument of the face “F” function has been modified
    - the solver calls the driver of the face F function for a regeneration of the face
    - the driver rebuilds face F and adds the label of the face “width” argument to

the logbook as touched and the label of the function of face F as impacted

- solver detects the function of P – it depends on the function of F
- the solver calls the driver of the prism P function
- the driver rebuilds prism P and adds the label of this prism to the logbook as impacted
- the solver detects the function of L – it depends on the function of P
- the solver calls the L function driver
- the driver rebuilds fillet L and adds the label of the fillet to the logbook as impacted

9. GLOSSARY

application

A document container holding all documents containing all application data.

application data

Data produced by an application, as opposed to data referring to it.

associativity of data

Ability to propagate modifications made to one document to other documents, which refer to such document. Modification propagation is:

- unidirectional, that is, from the referenced to the referencing document(s), or
- bi-directional, from the referencing to the referenced document and vice-versa.

See also: compound document

attribute

A container for application data. An attribute is attached to a label in the hierarchy of the data framework.

child

A label created from another label, which by definition, is the father label.

compound document

A set of interdependent documents, linked to each other by means of external references. These references provide the associativity of data.

data framework

A tree-like data structure which in OCAF, is a tree of labels with data attached to them in the form of attributes.

This tree of labels is accessible through the services of the **TDocStd_Document** class.

document

A container for a data framework which grants access to the data, and is, in its turn, contained by an application. A document also allows you to:

- Manage modifications, providing Undo and Redo functions
- Manage command transactions
- Update external links
- Manage save and restore options
- Store the names of software extensions.

driver

An abstract class, which defines the communications protocol with a system.

entry

An ASCII character string containing the tag list of a label.

Example

```
0:3:24:7:2:7
```

external links

References from one data structure to another data structure in another document.
To store these references properly, a label must also contain an external link attribute.

See also: scope

father

A label from which other labels have been created. The other labels are, by definition, the children of this label.

framework

A group of co-operating classes which enable a design to be re-used for a given category of problem. The framework guides the architecture of the application by breaking it up into abstract classes, each of which has different responsibilities and collaborates in a predefined way.

Application developer creates a specialized framework by:

- defining new classes which inherit from these abstract classes
- composing framework class instances
- implementing the services required by the framework.

In C++, he implements application behavior in the virtual functions redefined in these derived classes. This is known as overriding.

See also: data framework

GUID

Global Universal ID. A string of 37 characters intended to uniquely identify an object.

Example

```
2a96b602-ec8b-11d0-bee7-080009dc3333
```

label

A point in the data framework, which allows data to be attached to it by means of attributes.

This point has a name in the form of an entry, which identifies its place in the data framework.

modified

A modified label contains attributes whose data has been modified.

See also: valid

reference key

An invariant reference, which may refer to any type of data used in an application.

In its transient form, it is a label in the data framework, and the data is attached to it in the form of attributes.

In its persistent form, it is an entry of the label.

It allows an application to recover any entity in the current session or in a previous session.

resource file

A file containing a list of each document’s schema name and the storage and retrieval plug-ins for that document.

root

The starting point of the data framework.

This point is the top label in the framework. It is represented by the [0] entry and is created at the same time with the document you are working on.

scope

The set of all the attributes and labels which depend on a given label. The scope of that label.

tag list

The list of integers which identifies the place of a label in the data framework.
This list is displayed in an entry.

topological naming

Systematic referencing of topological entities so that these entities can still be identified after the models they belong to have gone through several steps in modeling. In other words, topological naming allows you to track entities through the steps in the modeling process.

This referencing is needed when a model is edited and regenerated, and can be seen as a mapping of labels and name attributes of the entities in the old version of a model to those of the corresponding entities in its new version.

Note that if the model’s topology changes in the modeling process, this mapping may not fully coincide. A Boolean operation, for example, may split edges.

topological tracking

Following a topological entity in a model through the steps taken to edit and regenerate that model.

valid

A set of valid labels in a data framework is a set of already recomputed labels in the scope of regeneration sequence and including the label containing a feature which is to be recalculated.

Consider the case of a box to which you first add a fillet, then a protrusion feature. For recalculation purposes, only valid labels of each construction stage are used. In recalculating a fillet, they are only those of the box and the fillet, not the protrusion feature which was added afterwards.

See also: modified

10. APPENDIX

10.1 Implementation of attribute Transformation (CDL file).

```
class Transformation from MyPackage inherits Attribute from TDF

    ---Purpose: This attribute implements a transformation data container.

uses

     Attribute       from TDF,
     Label           from TDF,
     GUID            from Standard,
     RelocationTable from TDF,
     Pnt             from gp,
     Ax1             from gp,
     Ax2             from gp,
     Ax3             from gp,
     Vec             from gp,
     Trsf            from gp,
     TrsfForm        from gp

is

    ---Category: Static methods
    --          ===============

    GetID (myclass)
    ---C++: return const &
    ---Purpose: The method returns a unique GUID of this attribute.
    --          By means of this GUID this attribute may be identified
    --          among other attributes attached to the same label.
    returns GUID from Standard;

    Set (myclass; theLabel : Label from TDF)
    ---Purpose: Finds or creates the attribute attached to <theLabel>.
    --          The found or created attribute is returned.
    returns Transformation from MyPackage;


    ---Category: Methods for access to the attribute data
    --           ========================================

    Get (me)
    ---Purpose: The method returns the transformation.
    returns Trsf from gp;


    ---Category: Methods for setting the data of transformation
    --           ==============================================

    SetRotation (me : mutable;
                       theAxis : Ax1 from gp;
                       theAngle : Real from Standard);
```

```
---Purpose: The method defines a rotation type of transformation.

SetTranslation (me : mutable;
                          theVector : Vec from gp);
---Purpose: The method defines a translation type of transformation.

SetMirror (me : mutable;
               thePoint : Pnt from gp);
---Purpose: The method defines a point mirror type of transformation
--          (point symmetry).

SetMirror (me : mutable;
               theAxis : Ax1 from gp);
---Purpose: The method defines an axis mirror type of transformation
--          (axial symmetry).

SetMirror (me : mutable;
               thePlane : Ax2 from gp);
---Purpose: The method defines a point mirror type of transformation
--          (planar symmetry).

SetScale (me : mutable;
              thePoint : Pnt from gp;
              theScale : Real from Standard);
---Purpose: The method defines a scale type of transformation.

SetTransformation (me : mutable;
                             theCoordinateSystem1 : Ax3 from gp;
                             theCoordinateSystem2 : Ax3 from gp);
---Purpose: The method defines a complex type of transformation
--          from one co-ordinate system to another.


---Category: Overridden methods from TDF_Attribute
--           ===================================

ID (me)
---C++: return const &
---Purpose: The method returns a unique GUID of the attribute.
--          By means of this GUID this attribute may be identified
--          among other attributes attached to the same label.
returns GUID from Standard;

Restore (me: mutable;
             theAttribute : Attribute from TDF);
---Purpose: The method is called on Undo / Redo.
--          It copies the content of <theAttribute>
--          into this attribute (copies the fields).

NewEmpty (me)
---Purpose: It creates a new instance of this attribute.
--          It is called on Copy / Paste, Undo / Redo.
returns mutable Attribute from TDF;

Paste (me;
          theAttribute : mutable Attribute from TDF;
```

```
            theRelocationTable : mutable RelocationTable from TDF);
   ---Purpose:: The method is called on Copy / Paste.
   --          It copies the content of this attribute into
   --          <theAttribute> (copies the fields).

   Dump(me; anOS : in out OStream from Standard)
   ---C++: return &
   ---Purpose: Prints the content of this attribute into the stream.
   returns OStream from Standard is redefined;


   ---Category: Constructor
   --          ===========

   Create
   ---Purpose: The C++ constructor of this atribute class.
   --          Usually it is never called outside this class.
   returns mutable Transformation from MyPackage;


fields

   -- Type of transformation
   myType : TrsfForm from gp;

   -- Axes (Ax1, Ax2, Ax3)
   myAx1 : Ax1 from gp;
   myAx2 : Ax2 from gp;
   myFirstAx3 : Ax3 from gp;
   mySecondAx3 : Ax3 from gp;

   -- Scalar values
   myAngle : Real from Standard;
   myScale : Real from Standard;

   -- Points
   myFirstPoint : Pnt from gp;
   mySecondPoint : Pnt from gp;


end Transformation;
```

10.2 Implementation of attribute Transformation, (CPP file).

```
#include <MyPackage_Transformation.ixx>

//=======================================================================
//function : GetID
//purpose  : The method returns a unique GUID of this attribute.
//           By means of this GUID this attribute may be identified
//           among other attributes attached to the same label.
//=======================================================================
const Standard_GUID& MyPackage_Transformation::GetID()
{
  static Standard_GUID ID("4443368E-C808-4468-984D-B26906BA8573");
  return ID;
```

```
}

//=======================================================================
//function : Set
//purpose  : Finds or creates the attribute attached to <theLabel>.
//           The found or created attribute is returned.
//=======================================================================
Handle(MyPackage_Transformation)       MyPackage_Transformation::Set(const       TDF_Label&
theLabel)
{
  Handle(MyPackage_Transformation) T;
  if (!theLabel.FindAttribute(MyPackage_Transformation::GetID(), T))
  {
    T = new MyPackage_Transformation();
    theLabel.AddAttribute(T);
  }
  return T;
}

//=======================================================================
//function : Get
//purpose  : The method returns the transformation.
//=======================================================================
gp_Trsf MyPackage_Transformation::Get() const
{
  gp_Trsf transformation;
  switch (myType)
  {
    case gp_Identity:
    {
      break;
    }
    case gp_Rotation:
    {
      transformation.SetRotation(myAx1, myAngle);
      break;
    }
    case gp_Translation:
    {
      transformation.SetTranslation(myFirstPoint, mySecondPoint);
      break;
    }
    case gp_PntMirror:
    {
      transformation.SetMirror(myFirstPoint);
      break;
    }
    case gp_Ax1Mirror:
    {
      transformation.SetMirror(myAx1);
      break;
    }
    case gp_Ax2Mirror:
    {
      transformation.SetMirror(myAx2);
      break;
    }
    case gp_Scale:
    {
      transformation.SetScale(myFirstPoint, myScale);
      break;
    }
    case gp_CompoundTrsf:
    {
      transformation.SetTransformation(myFirstAx3, mySecondAx3);
      break;
    }
    case gp_Other:
    {
      break;
    }
  }
  return transformation;
}

//=======================================================================
//function : SetRotation
//purpose  : The method defines a rotation type of transformation.
//=======================================================================
void MyPackage_Transformation::SetRotation(const gp_Ax1& theAxis, const Standard_Real
theAngle)
{
  Backup();
  myType = gp_Rotation;
  myAx1 = theAxis;
  myAngle = theAngle;
}

//=======================================================================
//function : SetTranslation
//purpose  : The method defines a translation type of transformation.
//=======================================================================
void MyPackage_Transformation::SetTranslation(const gp_Vec& theVector)
{
  Backup();
  myType = gp_Translation;
  myFirstPoint.SetCoord(0, 0, 0);
  mySecondPoint.SetCoord(theVector.X(), theVector.Y(), theVector.Z());
}

//=======================================================================
//function : SetMirror
//purpose  : The method defines a point mirror type of transformation
//           (point symmetry).
//=======================================================================
void MyPackage_Transformation::SetMirror(const gp_Pnt& thePoint)
{
  Backup();
  myType = gp_PntMirror;
  myFirstPoint = thePoint;
}
```

```
//=======================================================================
//function : SetMirror
//purpose  : The method defines an axis mirror type of transformation
//           (axial symmetry).
//=======================================================================
void MyPackage_Transformation::SetMirror(const gp_Ax1& theAxis)
{
  Backup();
  myType = gp_Ax1Mirror;
  myAx1 = theAxis;
}

//=======================================================================
//function : SetMirror
//purpose  : The method defines a point mirror type of transformation
//           (planar symmetry).
//=======================================================================
void MyPackage_Transformation::SetMirror(const gp_Ax2& thePlane)
{
  Backup();
  myType = gp_Ax2Mirror;
  myAx2 = thePlane;
}

//=======================================================================
//function : SetScale
//purpose  : The method defines a scale type of transformation.
//=======================================================================
void MyPackage_Transformation::SetScale(const gp_Pnt& thePoint, const Standard_Real
theScale)
{
  Backup();
  myType = gp_Scale;
  myFirstPoint = thePoint;
  myScale = theScale;
}

//=======================================================================
//function : SetTransformation
//purpose  : The method defines a complex type of transformation
//           from one co-ordinate system to another.
//=======================================================================
void MyPackage_Transformation::SetTransformation(const gp_Ax3& theCoordinateSystem1,
                                                 const gp_Ax3& theCoordinateSystem2)
{
  Backup();
  myFirstAx3 = theCoordinateSystem1;
  mySecondAx3 = theCoordinateSystem2;
}

//=======================================================================
//function : ID
//purpose  : The method returns a unique GUID of the attribute.
//           By means of this GUID this attribute may be identified
//           among other attributes attached to the same label.
//=======================================================================
const Standard_GUID& MyPackage_Transformation::ID() const
{
  return GetID();
}

//=======================================================================
//function : Restore
//purpose  : The method is called on Undo / Redo.
//           It copies the content of <theAttribute>
//           into this attribute (copies the fields).
//=======================================================================
void MyPackage_Transformation::Restore(const Handle(TDF_Attribute)& theAttribute)
{
  Handle(MyPackage_Transformation)                theTransformation                =
Handle(MyPackage_Transformation)::DownCast(theAttribute);
  myType = theTransformation->myType;
  myAx1 = theTransformation->myAx1;
  myAx2 = theTransformation->myAx2;
  myFirstAx3 = theTransformation->myFirstAx3;
  mySecondAx3 = theTransformation->mySecondAx3;
  myAngle = theTransformation->myAngle;
  myScale = theTransformation->myScale;
  myFirstPoint = theTransformation->myFirstPoint;
  mySecondPoint = theTransformation->mySecondPoint;
}

//=======================================================================
//function : NewEmpty
//purpose  : It creates a new instance of this attribute.
//           It is called on Copy / Paste, Undo / Redo.
//=======================================================================
Handle(TDF_Attribute) MyPackage_Transformation::NewEmpty() const
{
  return new MyPackage_Transformation();
}

//=======================================================================
//function : Paste
//purpose  : The method is called on Copy / Paste.
//           It copies the content of this attribute into
//           <theAttribute> (copies the fields).
//=======================================================================
void MyPackage_Transformation::Paste(const Handle(TDF_Attribute)& theAttribute,
                                      const Handle(TDF_RelocationTable)& ) const
{
  Handle(MyPackage_Transformation)                theTransformation                =
Handle(MyPackage_Transformation)::DownCast(theAttribute);
  theTransformation->myType = myType;
  theTransformation->myAx1 = myAx1;
  theTransformation->myAx2 = myAx2;
  theTransformation->myFirstAx3 = myFirstAx3;
  theTransformation->mySecondAx3 = mySecondAx3;
  theTransformation->myAngle = myAngle;
  theTransformation->myScale = myScale;
  theTransformation->myFirstPoint = myFirstPoint;
  theTransformation->mySecondPoint = mySecondPoint;
```

```
}

//=======================================================================
//function : Dump
//purpose  : Prints the content of this attribute into the stream.
//=======================================================================
Standard_OStream& MyPackage_Transformation::Dump(Standard_OStream& anOS) const
{
  anOS<<"Transformation: ";
  switch (myType)
  {
    case gp_Identity:
    {
      anOS<<"gp_Identity";
      break;
    }
    case gp_Rotation:
    {
      anOS<<"gp_Rotation";
      break;
    }
    case gp_Translation:
    {
      anOS<<"gp_Translation";
      break;
    }
    case gp_PntMirror:
    {
      anOS<<"gp_PntMirror";
      break;
    }
    case gp_Ax1Mirror:
    {
      anOS<<"gp_Ax1Mirror";
      break;
    }
    case gp_Ax2Mirror:
    {
      anOS<<"gp_Ax2Mirror";
      break;
    }
    case gp_Scale:
    {
      anOS<<"gp_Scale";
      break;
    }
    case gp_CompoundTrsf:
    {
      anOS<<"gp_CompoundTrsf";
      break;
    }
    case gp_Other:
    {
      anOS<<"gp_Other";
      break;
    }
  }
  return anOS;
}

//=======================================================================
//function : MyPackage_Transformation
//purpose  : A constructor.
//=======================================================================
MyPackage_Transformation::MyPackage_Transformation():myType(gp_Identity)
{

}
```