
# Open CASCADE Technology 7.5.0

## Extended Data Exchange (XDE)

November 3, 2020

# Contents

# 1 Introduction

This manual explains how to use the Extended Data Exchange (XDE). It provides basic documentation on setting up and using XDE.

The Extended Data Exchange (XDE) module allows extending the scope of exchange by translating additional data attached to geometric BREP data, thereby improving the interoperability with external software.

Data types such as colors, layers, assembly descriptions and validation properties (i.e. center of gravity, etc.) are supported. These data are stored together with shapes in an XCAF document. It is also possible to add a new types of data taking the existing tools as prototypes.

Finally, the XDE provides reader and writer tools for reading and writing the data supported by XCAF to and from IGES and STEP files.

Figure 1: Shape imported using XDE

The XDE component requires Shape Healing toolkit for operation.

## 1.1 Basic terms

For better understanding of XDE, certain key terms are defined:

- **Shape** – a standalone shape, which does not belong to the assembly structure.
- **Instance** – a replication of another shape with a location that can be the same location or a different one.
- **Assembly** – a construction that is either a root or a sub-assembly.

## 1.2 XDE Organization

The basis of XDE, called XCAF, is a framework based on OCAF (Open CASCADE Technology Application Framework) and is intended to be used with assemblies and with various kinds of attached data (attributes). Attributes can be Individual attributes for a shape, specifying some characteristics of a shape, or they can be Grouping attributes, specifying that a shape belongs to a given group whose definition is specified apart from the shapes.

XDE works in an OCAF document with a specific organization defined in a dedicated XCAF module. This organization is used by various functions of XDE to exchange standardized data other than shapes and geometry.

The Assembly Structure and attributes assigned to shapes are stored in the OCAF tree. It is possible to obtain TopoDS representation for each level of the assembly in the form of *TopoDS_Compound* or *TopoDS_Shape* using the API.

Basic elements used by XDE are introduced in the XCAF sub-module by the package XCAFDoc. These elements consist in descriptions of commonly used data structures (apart from the shapes themselves) in normalized data exchanges. They are not attached to specific applications and do not bring specific semantics, but are structured according to the use and needs of data exchanges. The Document used by XDE usually starts as a *TDocStd_↩ Document*.

## 1.3 Assemblies

XDE supports assemblies by separating shape definitions and their locations. Shapes are simple OCAF objects without a location definition. An assembly consists of several components. Each of these components references one and the same specified shape with different locations. All this provides an increased flexibility in working on multi-level assemblies.

For example, a mechanical assembly can be defined as follows:

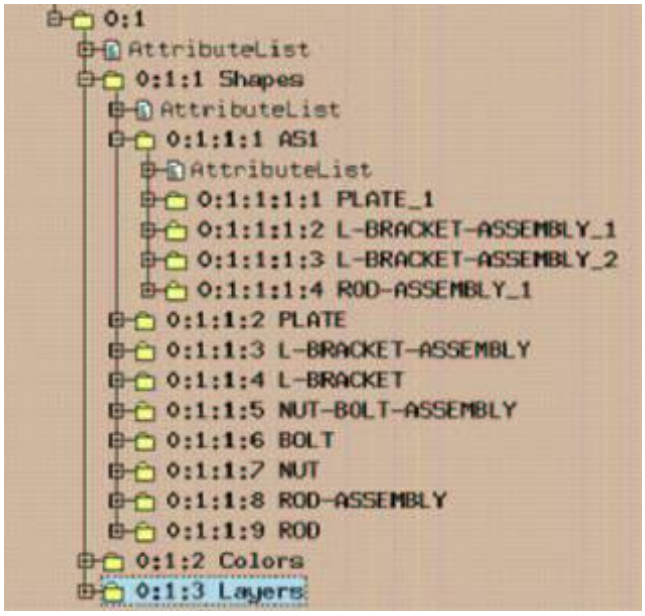


Figure 2: Assembly Description

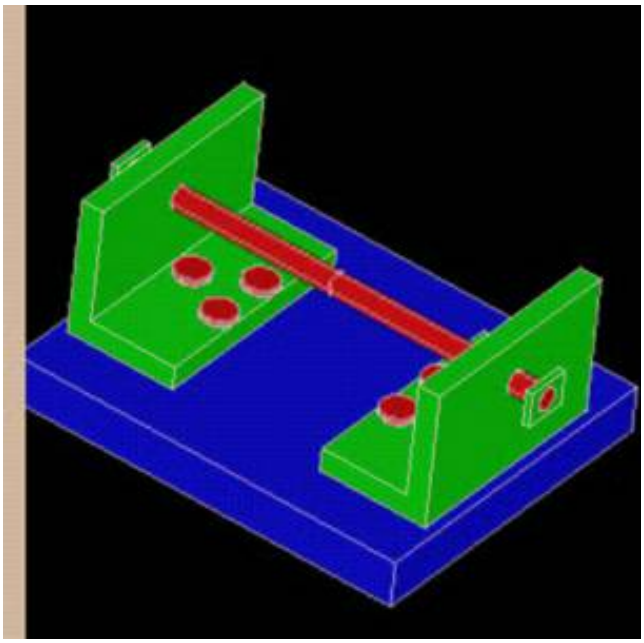

Figure 3: Assembly View

XDE defines the specific organization of the assembly content. Shapes are stored on sub-labels of label 0:1:1. There can be one or more roots (called free shapes) whether they are true trees or simple shapes. A shape can be considered to be an Assembly (such as AS1 under 0:1:1:1 in Figure1) if it is defined with Components (sub-shapes, located or not).

*XCAFDoc_ShapeTool* is a tool that allows managing the Shape section of the XCAF document. This tool is implemented as an attribute and located at the root label of the shape section.

## 1.4 Validation Properties

Validation properties are geometric characteristics of Shapes (volume, centroid, surface area) written to STEP files by the sending system. These characteristics are read by the receiving system to validate the quality of the translation. This is done by comparing the values computed by the original system with the same values computed by the receiving system on the resulting model.

Advanced Data Exchange supports both reading and writing of validation properties, and provides a tool to check them.

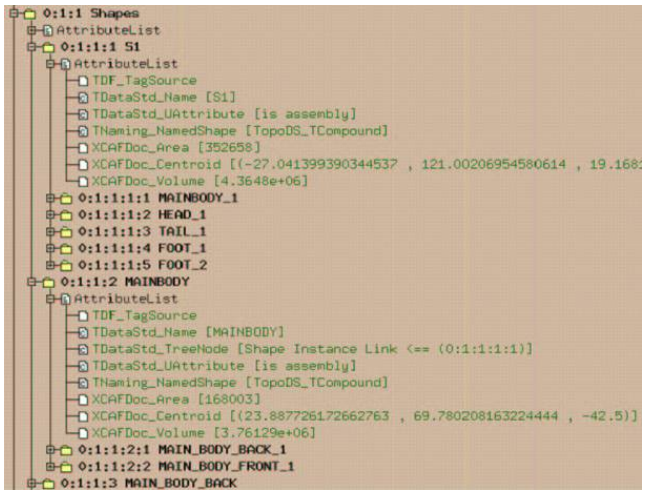


Figure 4: Validation Property Descriptions

Check logs contain deviations of computed values from the values stored in a STEP file. A typical example appears as follows:

| Label | Area defect | Volume defect | dX | dY | DZ | Name |
|---|---|---|---|---|---|---|
| 0:1:1:1 | 312.6 (0%) | -181.7 (0%) | 0.00 | 0.00 | 0.00 | "S1" |
| 0:1:1:2 | -4.6 (0%) | -191.2 (0%) | -0.00 | 0.00 | -0.00 | "MAINBODY" |
| 0:1:1:3 | -2.3 (0%) | -52.5 (0%) | -0.00 | 0.00 | 0.00 | "MAIN_BODY_BACK" |
| 0:1:1:4 | -2.3 (0%) | -51.6 (0%) | 0.00 | 0.00 | -0.00 | "MAIN_BODY_FRONT" |
| 0:1:1:5 | 2.0 (0%) | 10.0 (0%) | -0.00 | 0.00 | -0.00 | "HEAD" |
| 0:1:1:6 | 0.4 (0%) | 0.0 (0%) | 0.00 | -0.00 | -0.00 | "HEAD_FRONT" |
| 0:1:1:7 | 0.4 (0%) | 0.0 (0%) | 0.00 | -0.00 | -0.00 | "HEAD_BACK" |
| 0:1:1:8 | -320.6 (0%) | 10.9 (0%) | -0.00 | 0.00 | 0.00 | "TAIL" |
| 0:1:1:9 | 0.0 (0%) | 0.0 (0%) | -0.00 | -0.00 | 0.00 | "TAIL_MIDDLE" |
| 0:1:1:10 | -186.2 (0%) | 4.8 (0%) | -0.00 | 0.00 | -0.00 | "TAIL_TURBINE" |
| 0:1:1:11 | 0.3 (0%) | -0.0 (0%) | -0.00 | -0.00 | 0.00 | "FOOT" |
| 0:1:1:12 | 0.0 (0%) | -0.0 (0%) | 0.00 | -0.00 | -0.00 | "FOOT_FRONT" |
| 0:1:1:13 | 0.0 (0%) | 0.0 (0%) | -0.00 | 0.00 | 0.00 | "FOOT_BACK" |

In our example, it can be seen that no errors were detected for either area, volume or positioning data.

## 1.5 Names

XDE supports reading and writing the names of shapes to and from IGES and STEP file formats. This functionality can be switched off if you do not need this type of data, thereby reducing the size of the document.

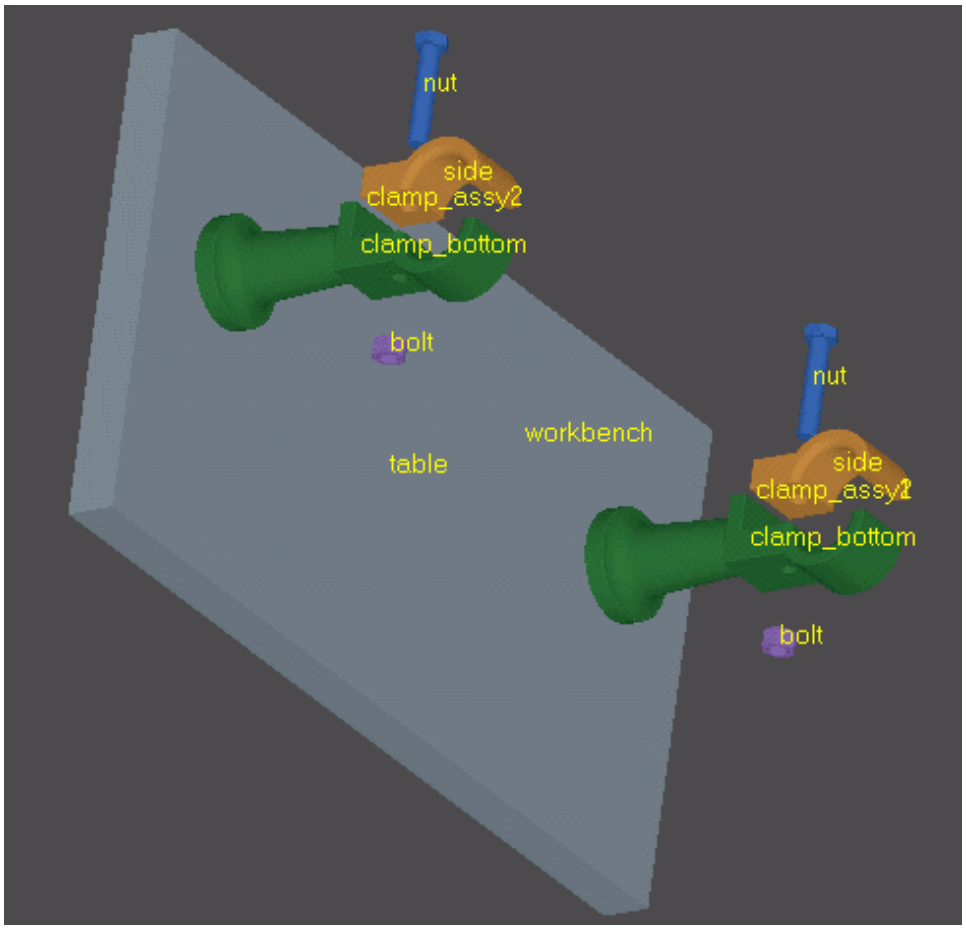


Figure 5: Instance Names

## 1.6 Colors and Layers

XDE can read and write colors and layers assigned to shapes or their subparts (down to the level of faces and edges) to and from both IGES and STEP formats. Three types of colors are defined in the enumeration *XCAF↩ Doc_ColorType*:

- generic color *(XCAFDoc_ColorGen)*
- surface color *(XCAFDoc_ColorSurf)*
- curve color *(XCAFDoc_ColorCurv)*

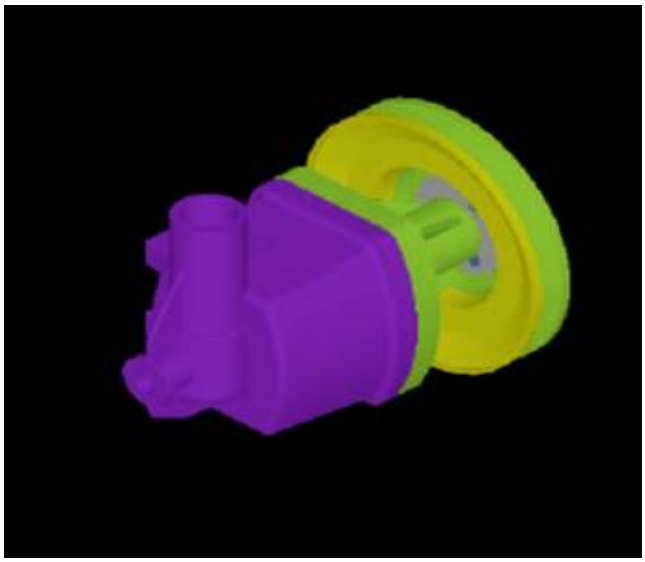

Figure 6: Colors and Layers

## 1.7 Geometric Dimensions & Tolerances (GD&T)

GD&T are a type of Product and Manufacturing Information (PMI) that can be either computed automatically by a CAD system, or entered manually by the user. For detailed information use CAx-IF Recommended Practices for the Representation and Presentation of Product Manufacturing Information (PMI) (AP242)

XDE can read and write GD&T data of the following types:

- dimensions, such as distance, length, radius and so on;
- geometric tolerances;
- datums, i.e theoretically exact geometric references, such as point, line or plane, to which toleranced features are related.

XDE supports two presentations of GD&T data:

- semantic presentation, i.e. data is stored in a machine-consumable way and includes all information required to understand the specification without the aid of any presentation elements;
- tessellated presentation, i.e. data is displayed in a human-readable way.

## 1.8 Clipping planes

XDE supports reading from STEP and storing named planes used for clipping. Currently, XDE supports saving of clipping planes in XBF format only.

XDE provides capabilities for adding, editing and removing clipping planes.

## 1.9 Saved views

XDE supports reading from STEP views. Views allow saving information about camera parameters (position, direction, zoom factor, etc.) and visible shapes, PMIs, used clipping planes and notes. Currently, XDE supports saving of clipping planes in XBF format only.

XDE provides the following view management capabilities:

- add/remove views;
- set view camera parameters;
- set visible shapes, PMIs, used clipping planes and notes.

## 1.10 Custom notes

Custom notes is a kind of application-specific data attached to assembly items, their attributes and sub-shapes. Basically, there are simple textual comments, binary data and other application-specific data. Each note is provided with a timestamp and the user who created it.

Notes API provides the following functionality:

- Returns the total number of notes and annotated items;
- Returns labels for all notes and annotated items;
- Creates notes:
  - Comment note from a text string;
  - Binary data note from a file or byte array;
- Checks if an assembly item is annotated;
- Finds a label for the annotated item;
- Returns all note labels for the annotated item;
- Adds a note to item(s):
  - Assembly item;
  - Assembly item attribute;
  - Assembly item subshape index;
- Removes note(s) from an annotated assembly item; orphan note(s) might be deleted optionally (items without linked notes will be deleted automatically);
- Deletes note(s) and removes them from annotated items;
- Gets / deletes orphan notes.

# 2 Working with XDE

## 2.1 Getting started

As explained in the last chapter, XDE uses *TDocStd_Documents* as a starting point. The general purpose of XDE is:

- Checking if an existing document is fit for XDE;
- Getting an application and initialized document;
- Initializing a document to fit it for XDE;
- Adding, setting and finding data;
- Querying and managing shapes;
- Attaching properties to shapes.

The Document used by XDE usually starts as a *TDocStd_Document*.

### 2.1.1 Environment variables

To use XDE you have to set the environment variables properly. Make sure that two important environment variables are set as follows:

- *CSF_PluginDefaults* points to sources of *%CASROOT%/src/XCAFResources ($CASROOT/src/XCAF↩ Resources)*.
- *CSF_XCAFDefaults* points to sources of *%CASROOT%/src/XCAFResources ($CASROOT/src/XCAF↩ Resources)*.

### 2.1.2 General Check

Before working with shapes, properties, and other types of information, the global organization of an XDE Document can be queried or completed to determine if an existing Document is actually structured for use with XDE.

To find out if an existing *TDocStd_Document* is suitable for XDE, use:

```
Handle(TDocStd_Document) doc...
if ( XCAFDoc_DocumentTool::IsXCAFDocument (doc) ) { .. yes .. }
```

If the Document is suitable for XDE, you can perform operations and queries explained in this guide. However, if a Document is not fully structured for XDE, it must be initialized.

### 2.1.3 Get an Application or an Initialized Document

If you want to retrieve an existing application or an existing document (known to be correctly structured for XDE), use:

```
Handle(TDocStd_Document) aDoc;
Handle(XCAFApp_Application) anApp = XCAFApp_Application::GetApplication();
anApp->NewDocument(;MDTV-XCAF;,aDoc);
```

## 2.2 Shapes and Assemblies

### 2.2.1 Initialize an XDE Document (Shapes)

An XDE Document begins with a *TDocStd_Document*. Assuming you have a *TDocStd_Document* already created, you can ensure that it is correctly structured for XDE by initializing the XDE structure as follows:

```
Handle(TDocStd_Document) doc...
Handle (XCAFDoc_ShapeTool) myAssembly =
XCAFDoc_DocumentTool::ShapeTool (Doc->Main());
TDF_Label aLabel = myAssembly->NewShape()
```

**Note** that the method *XCAFDoc_DocumentTool::ShapeTool* returns the *XCAFDoc_ShapeTool*. The first time this method is used, it creates the *XCAFDoc_ShapeTool*. In our example, a handle is used for the *TDocStd_Document*.

### 2.2.2 Get a Node considered as an Assembly

To get a node considered as an Assembly from an XDE structure, you can use the Label of the node. Assuming that you have a properly initialized *TDocStd_Document*, use:

```
Handle(TDocStd_Document) doc...
Handle(XCAFDoc_ShapeTool) myAssembly = XCAFDoc_DocumentTool::ShapeTool (aLabel);
```

In the previous example, you can also get the Main Item of an XDE document, which records the root shape representation (as a Compound if it is an Assembly) by using *ShapeTool(Doc->Main())* instead of *ShapeTool(a↩ Label)*.

You can then query or edit this Assembly node, the Main Item or another one (*myAssembly* in our examples).

**Note** that for the examples in the rest of this guide, *myAssembly* is always presumed to be accessed this way, so this information will not be repeated.

### 2.2.3 Updating the Assemblies after Filling or Editing

Some actions in this chapter affect the content of the document, considered as an Assembly. As a result, you will sometimes need to update various representations (including the compounds).

To update the representations, use:

```
myAssembly->UpdateAssemblies();
```

This call performs a top-down update of the Assembly compounds stored in the document.

**Note** that you have to run this method manually to actualize your Assemblies after any low-level modifications on shapes.

### 2.2.4 Adding or Setting Top Level Shapes

Shapes can be added as top-level shapes. Top level means that they can be added to an upper level assembly or added on their own at the highest level as a component or referred by a located instance. Therefore two types of top-level shapes can be added:

- shapes with upper level references
- free shapes (that correspond to roots) without any upper reference

**Note** that several top-level shapes can be added to the same component.

A shape to be added can be defined as a compound (if required), with the following interpretations:

- If the Shape is a compound, according to the user choice, it may or may not be interpreted as representing an Assembly. If it is an Assembly, each of its sub-shapes defines a sub-label.
- If the Shape is not a compound, it is taken as a whole, without breaking it down.

To break down a Compound in the assembly structure, use:

```
Standard_Boolean makeAssembly;
// True to interpret a Compound as an Assembly,
// False to take it as a whole
aLabel = myAssembly->AddShape(aShape, makeAssembly);
```

Each node of the assembly therefore refers to its sub-shapes.

Concerning located instances of sub-shapes, the corresponding shapes, (without location) appear at distinct sub-labels. They are referred to by a shape instance, which associates a location.

#### 2.2.5 Setting a given Shape at a given Label

A top-level shape can be changed. In this example, no interpretation of compound is performed:

```
Standard_CString LabelString ...;
// identifies the Label (form ;0:i:j...;)
TDF_Label aLabel...;
// A label must be present
myAssembly->SetShape(aLabel, aShape);
```

#### 2.2.6 Getting a Shape from a Label

To get a shape from its Label from the top-level, use:

```
TDF_Label aLabel...
// A label must be present
if (aLabel.IsNull()) {
  // no such label : abandon
}
TopoDS_Shape aShape;
aShape = myAssembly->GetShape(aLabel);
if (aShape.IsNull()) {
  // this label is not for a Shape
}
```

**Note** that if the label corresponds to an assembly, the result is a compound.

#### 2.2.7 Getting a Label from a Shape

To get a Label, which is attached to a Shape from the top-level, use:

```
Standard_Boolean findInstance = Standard_False;
// (this is default value)
aLabel = myAssembly->FindShape(aShape [,findInstance]);
if (aLabel.IsNull()) {
  // no label found for this shape
}
```

If *findInstance* is True, a search is made for the shape with the same location. If it is False (default value), a search is made among original, non-located shapes.

#### 2.2.8 Other Queries on a Label

Various other queries can be made from a Label within the Main Item of XDE:

#### Main Shapes

To determine if a Shape is recorded (or not), use:

```
if ( myAssembly->IsShape(aLabel) ) { .. yes .. }
```

To determine if the shape is top-level, i.e. was added by the *AddShape* method, use:

```
if ( myAssembly->IsTopLevel(aLabel) ) { .. yes .. }
```

To get a list of top-level shapes added by the *AddShape* method, use:

```
TDF_LabelSequence frshapes;
myAssembly->GetShapes(frshapes);
```

To get all free shapes at once if the list above has only one item, use:

```
TopoDS_Shape result = myAssembly->GetShape(frshapes.Value(1));
```

If there is more than one item, you must create and fill a compound, use:

```
TopoDS_Compound C;
BRep_Builder B;
B.MakeCompound(C);
for(Standard_Integer i=1; i=frshapes.Length(); i++) {
  TopoDS_Shape S = myAssembly->GetShape(frshapes.Value(i));
  B.Add(C,S);
}
```

In our example, the result is the compound C. To determine if a shape is a free shape (no reference or super-assembly), use:

```
if ( myAssembly->IsFree(aLabel) ) { .. yes .. }
```

To get a list of Free Shapes (roots), use:

```
TDF_LabelSequence frshapes;
myAssembly->GetFreeShapes(frshapes);
```

To get the shapes, which use a given shape as a component, use:

```
TDF_LabelSequence users;
Standard_Integer nbusers = myAssembly->GetUsers(aLabel,users);
```

The count of users is contained with *nbusers*. It contains 0 if there are no users.

#### Assembly and Components

To determine if a label is attached to the main part or to a sub-part (component), use:

```
if (myAssembly->IsComponent(aLabel)) { .. yes .. }
```

To determine whether a label is a node of a (sub-) assembly or a simple shape, use:

```
if ( myAssembly->IsAssembly(aLabel) ) { .. yes .. }
```

If the label is a node of a (sub-) assembly, you can get the count of components, use:

```
Standard_Boolean subchilds = Standard_False; //default
Standard_Integer nbc = myAssembly->NbComponents (aLabel [,subchilds]);
```

If *subchilds* is True, commands also consider sub-levels. By default, only level one is checked.

To get component Labels themselves, use:

```
Standard_Boolean subchilds = Standard_False; //default
TDF_LabelSequence comps;
Standard_Boolean isassembly = myAssembly->GetComponents
(aLabel,comps[,subchilds]);
```

#### 2.2.9 Instances and References for Components

To determine if a label is a simple shape, use:

```
if ( myAssembly->IsSimpleShape(aLabel) ) { .. yes .. }
```

To determine if a label is a located reference to another one, use:

```
if ( myAssembly->IsReference(aLabel) ) { .. yes .. }
```

If the label is a located reference, you can get the location, use:

```
TopLoc_Location loc = myAssembly->GetLocation (aLabel);
```

To get the label of a referenced original shape (also tests if it is a reference), use:

```
Standard_Boolean isref = myAssembly->GetReferredShape
(aLabel, refLabel);
```

**Note** *isref* returns False if *aLabel* is not for a reference.

### 2.3 Editing Shapes

In addition to the previously described *AddShape* and *SetShape*, several shape edits are possible.

To remove a Shape, and all its sub-labels, use:

```
Standard_Boolean remsh = myAssembly->RemoveShape(aLabel);
// remsh is returned True if done
```

This operation will fail if the shape is neither free nor top level.

To add a Component to the Assembly, from a new shape, use:

```
Standard_Boolean expand = Standard_False; //default
TDF_Label aLabel = myAssembly->AddComponent (aShape [,expand]);
```

If *expand* is True and *aShape* is a Compound, *aShape* is broken down to produce sub-components, one for each of its sub-shapes.

To add a component to the assembly, from a previously recorded shape (the new component is defined by the label of the reference shape, and its location), use:

```
TDF_Label refLabel ...; // the label of reference shape
TopLoc_Location loc ...; // the desired location
TDF_Label aLabel = myAssembly->AddComponent (refLabel, loc);
```

To remove a component from the assembly, use:

```
myAssembly->RemoveComponent (aLabel);
```

### 2.4 Management of Sub-Shapes

In addition to components of a (sub-)assembly, it is possible to have individual identification of some sub-shapes inside any shape. Therefore, you can attach specific attributes such as Colors. Some additional actions can be performed on sub-shapes that are neither top-level, nor components: To add a sub-shape to a given Label, use:

```
TDF_Label subLabel = myAssembly->AddSubShape (aLabel, subShape);
```

To find the Label attached to a given sub-shape, use:

```
TDF_Label subLabel; // new label to be computed
if ( myAssembly-> FindSubShape (aLabel, subShape, subLabel)) { .. yes .. }
```

If the sub-shape is found (yes), *subLabel* is filled by the correct value.

To find the top-level simple shape (not a compound whether free or not), which contains a given sub-shape, use:

```
TDF_Label mainLabel = myAssembly->FindMainShape(subShape);
```

**Note** that there should be only one shape for a valid model. In any case, the search stops on the first one found.

To get the sub-shapes of a shape, which are recorded under a label, use:

```
TDF_LabelSequence subs;
Standard_Boolean hassubs = myAssembly->GetSubShapes (aLabel,subs);
```

## 2.5 Properties

Some properties can be attached directly to shapes. These properties are:

- Name (standard definition from OCAF)
- Centroid (for validation of transfer)
- Volume (for validation of transfer)
- Area (for validation of transfer) Some other properties can also be attached, and are also managed by distinct tools for Colors and Layers. Colors and Layers are managed as an alternative way of organizing data (by providing a way of identifying groups of shapes). Colors are put into a table of colors while shapes refer to this table. There are two ways of attaching a color to a shape:
- By attaching an item from the table.
- Adding the color directly. When the color is added directly, a search is performed in the table of contents to determine if it contains the requested color. Once this search and initialize operation is done, the first way of attaching a color to a shape is used.

### 2.5.1 Name

Name is implemented and used as a *TDataStd_Name*, which can be attached to any label. Before proceeding, consider that:

- In IGES, every entity can have a name with an optional numeric part called a Subscript Label. For example, *MYCURVE* is a name, and *MYCURVE(60)* is a name with a Subscript Label.
- In STEP, there are two levels: Part Names and Entity Names:
  - Part Names are attached to ;main shapes; such as parts and assemblies. These Part Names are specifically supported by XDE.
  - Entity Names can be attached to every Geometric Entity. This option is rarely used, as it tends to overload the exploitation of the data structure. Only some specific cases justify using this option: for example, when the sending system can really ensure the stability of an entity name after each STEP writing. If such stability is ensured, you can use this option to send an Identifier for external applications using a database. **Note** that both IGES or STEP files handle names as pure ASCII strings.

These considerations are not specific to XDE. What is specific to data exchange is the way names are attached to entities.

To get the name attached to a label (as a reminder using OCAF), use:

```
Handle(TDataStd_Name) N;
if ( !aLabel.FindAttribute(TDataStd_Name::GetID(),N)) {
  // no name is attached
}
TCollection_ExtendedString name = N->Get();
```

Don't forget to consider Extended String as ASCII, for the exchange file.

To set a name to a label (as a reminder using OCAF), use:

```
TCollection_ExtendedString aName ...;
// contains the desired name for this Label (ASCII)
TDataStd_Name::Set (aLabel, aName);
```

### 2.5.2 Centroid

A Centroid is defined by a Point to fix its position. It is handled as a property, item of the class *XCAFDoc_Centroid*, sub-class of *TDF_Attribute*. However, global methods give access to the position itself.

This notion has been introduced in STEP, together with that of Volume, and Area, as defining the Validation Properties: this feature allows exchanging the geometries and some basic attached values, in order to perform a synthetic checking on how they are maintained after reading and converting the exchange file. This exchange depends on reliable exchanges of Geometry and Topology. Otherwise, these values can be considered irrelevant.

A centroid can be determined at any level of an assembly, thereby allowing a check of both individual simple shapes and their combinations including locations.

To get a Centroid attached to a Shape, use:

```
gp_Pnt pos;
Handle(XCAFDoc_Centroid) C;
aLabel.FindAttribute ( XCAFDoc_Centroid::GetID(), C );
if ( !C.IsNull() ) pos = C->Get();
```

To set a Centroid to a Shape, use:

```
gp_Pnt pos (X,Y,Z);
// the position previously computed for the centroid
XCAFDoc_Centroid::Set ( aLabel, pos );
```

### 2.5.3 Area

An Area is defined by a Real, it corresponds to the computed Area of a Shape, provided that it contains surfaces. It is handled as a property, item of the class *XCAFDoc_Area*, sub-class of *TDF_Attribute*. This notion has been introduced in STEP but it is usually disregarded for a Solid, as Volume is used instead. In addition, it is attached to simple shapes, not to assemblies.

To get an area attached to a Shape, use:

```
Standard_Real area;
Handle(XCAFDoc_Area) A;
L.FindAttribute ( XCAFDoc_Area::GetID(), A );
if ( !A.IsNull() ) area = A->Get();

To set an area value to a Shape, use:
Standard_Real area ...;
// value previously computed for the area
XCAFDoc_Area::Set ( aLabel, area );
```

### 2.5.4 Volume

A Volume is defined by a Real and corresponds to the computed volume of a Shape, provided that it contains solids. It is handled as a property, an item of the class *XCAFDoc_Volume*, sub-class of *TDF_Attribute*. This notion has been introduced in STEP. It may be attached to simple shapes or their assemblies for computing cumulated volumes and centers of gravity.

To get a Volume attached to a Shape, use:

```
Standard_Real volume;
Handle(XCAFDoc_Volume) V;
L.FindAttribute ( XCAFDoc_Volume::GetID(), V );
if ( !V.IsNull() ) volume = V->Get();
```

To set a volume value to a Shape, use:

```
Standard_Real volume ...;
// value previously computed for the volume
XCAFDoc_Volume::Set ( aLabel, volume );
```

## 2.6 Colors and Layers

XDE can read and write colors and layers assigned to shapes or their subparts (down to level of faces and edges) to and from both IGES and STEP formats.

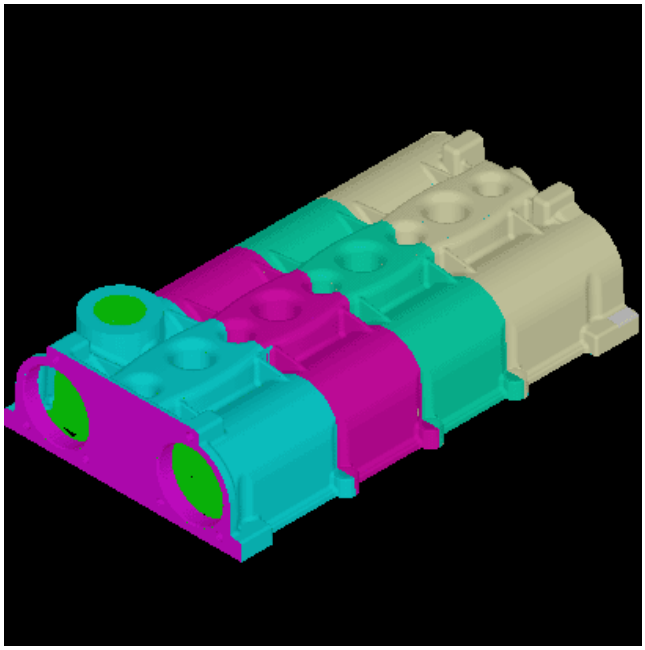

Figure 7: Motor Head

In an XDE document, colors are managed by the class *XCAFDoc_ColorTool*. It works basing on the same principles as ShapeTool works with Shapes. This tool can be provided on the Main Label or on any sub-label. The Property itself is defined as an *XCAFDoc_Color*, sub-class of *TDF_Attribute*.

Colors are stored in a child of the starting document label: it is the second level (0.1.2), while Shapes are at the first level. Each color then corresponds to a dedicated label, the property itself is a Quantity_Color, which has a name and value for Red, Green, Blue. A Color may be attached to Surfaces (flat colors) or to Curves (wireframe colors), or to both. A Color may be attached to a sub-shape. In such a case, the sub-shape (and its own sub-shapes) takes its own Color as a priority.

Layers are handled using the same principles as Colors. In all operations described below you can simply replace **Color** with **Layer** when dealing with Layers. Layers are supported by the class *XCAFDoc_LayerTool*.

The class of the property is *XCAFDoc_Layer*, sub-class of *TDF_Attribute* while its definition is a *TCollection_↩ ExtendedString*. Integers are generally used when dealing with Layers. The general cases are:

- IGES has *LevelList* as a list of Layer Numbers (not often used)
- STEP identifies a Layer (not by a Number, but by a String), to be more general.

Colors and Shapes are related to by Tree Nodes.

These definitions are common to various exchange formats, at least for STEP and IGES.

### 2.6.1 Initialization

To query, edit, or initialize a Document to handle Colors of XCAF, use:

```
Handle(XCAFDoc_ColorTool) myColors =
XCAFDoc_DocumentTool::ColorTool(Doc->Main ());
```

This call can be used at any time. The first time it is used, a relevant structure is added to the document. This definition is used for all the following color calls and will not be repeated for these.

### 2.6.2 Adding a Color

There are two ways to add a color. You can:

- add a new Color defined as *Quantity_Color* and then directly set it to a Shape (anonymous Color)
- define a new Property Color, add it to the list of Colors, and then set it to various shapes. When the Color is added by its value *Quantity_Color*, it is added only if it has not yet been recorded (same RGB values) in the Document.

To set a Color to a Shape using a label, use:

```
Quantity_Color Col (red,green,blue);
XCAFDoc_ColorType ctype ..;
// can take one of these values :
// XCAFDoc_ColorGen : all types of geometries
// XCAFDoc_ColorSurf : surfaces only
// XCAFDoc_ColorCurv : curves only
myColors->SetColor ( aLabel, Col, ctype );
```

Alternately, the Shape can be designated directly, without using its label, use:

```
myColors->SetColor ( aShape, Col, ctype );
// Creating and Adding a Color, explicitly
Quantity_Color Col (red,green,blue);
TDF_Label ColLabel = myColors->AddColor ( Col );
```

**Note** that this Color can then be named, allowing later retrieval by its Name instead of its Value.

To set a Color, identified by its Label and already recorded, to a Shape, use:

```
XCAFDoc_ColorType ctype ..; // see above
if ( myColors->SetColors ( aLabel, ColLabel, ctype) ) {.. it is done .. }
```

In this example, *aLabel* can be replaced by *aShape* directly.

### 2.6.3 Queries on Colors

Various queries can be performed on colors. However, only specific queries are included in this section, not general queries using names.

To determine if a Color is attached to a Shape, for a given color type (ctype), use:

```
if ( myColors->IsSet (aLabel , ctype)) {
  // yes, there is one ..
}
```

In this example, *aLabel* can be replaced by *aShape* directly.

To get the Color attached to a Shape (for any color type), use:

```
Quantity_Color col;
// will receive the recorded value (if there is some)
if ( !myColors->GetColor(aLabel, col) ) {
// sorry, no color ..
}
```

Color name can also be queried from *col.StringName* or *col.Name*. In this example, *aLabel* can be replaced by *aShape* directly.

To get the Color attached to a Shape, with a specific color type, use:

```
XCAFDoc_ColorType ctype ..;
Quantity_Color col;
// will receive the recorded value (if there is some)
if ( !myColors->GetColor(aLabel, ctype, col) ) {
// sorry, no color ..
}
```

To get all the Colors recorded in the Document, use:

```
Quantity_Color col; // to receive the values
TDF_LabelSequence ColLabels;
myColors->GetColors(ColLabels);
Standard_Integer i, nbc = ColLabels.Length();
for (i = 1; i = nbc; i ++) {
  aLabel = Labels.Value(i);
  if ( !myColors->GetColor(aLabel, col) ) continue;
  // col receives the color n0 i ..
}
```

To find a Color from its Value, use:

```
Quantity_Color Col (red,green,blue);
TDF_Label ColLabel = myColors-FindColor (Col);
if ( !ColLabel.IsNull() ) { .. found .. }
```

#### 2.6.4 Editing Colors

Besides adding colors, the following attribute edits can be made:

To unset a Color on a Shape, use:

```
XCAFDoc_ColorType ctype ...;
// desired type (XCAFDoc_ColorGen for all )
myColors->UnSetColor (aLabel,ctype);
```

To remove a Color and all the references to it (so that the related shapes will become colorless), use:

```
myColors->RemoveColor(ColLabel);
```

## 2.7 Geometric Dimensions & Tolerances (GD&T)

XDE can read and write GD&T assigned to shapes or their subparts (down to the level of faces and edges) to and from STEP formats.

In an XDE document, GD&T are managed by the class *XCAFDoc_DimTolTool*. It works basing on the same principles as ShapeTool works with Shapes. This tool can be provided on the Main Label or on any sub-label. The GD&T entities themselves are defined as the following sub-classes of *TDF_Attribute*:

- *XCAFDoc_Dimension* - for dimensions;
- *XCAFDoc_GeomTolerance* - for geometric tolerances;

- *XCAFDoc_Datum* - for geometric tolerance Datums. A GD&T type is identified by the attributes listed above, i.e. *XCAFDoc_DimTolTool* methods working with particular entity types check for presence of the corresponding attributes in passed labels. One can use methods of *XCAFDoc_DimTolTool* beginning with 'Is' for this purpose.

GD&T entities are stored in a child of the starting document label 0.1.4. Each GD&T entity then corresponds to the dedicated label, the property itself is one of access classes:

- *XCAFDimTolObject_DimensionObject* - for dimensions;
- *XCAFDimTolObject_GeomToleranceObject* - for geometric tolerances;
- *XCAFDimTolObject_DatumObject* - for geometric tolerance Datums.

GD&Ts and Shapes are related by Graph Nodes.

These definitions are common to various exchange formats, at least for STEP.

### 2.7.1 Initialization

To query, edit, or initialize a Document to handle GD&Ts of XCAF, use:

```
Handle(XCAFDoc_DimTolTool) myDimTolTool =
XCAFDoc_DocumentTool::DimTolTool(Doc->Main());
```

This call can be used at any time. When it is used for the first time, a relevant structure is added to the document. This definition is used for all later GD&T calls and is not repeated for them.

### 2.7.2 Adding a GD&T

*XCAFDoc_DimTolTool* provides methods to create GD&T 'empty' entities:

- *AddDimension* - for a new dimension;
- *AddGeomTolerance* - for a new geometric tolerance;
- *AddDatum* - for a new geometric tolerance datum.

All methods create a sub-label for the corresponding GD&T entity of the tool master label and attach an attribute specific for the created entity.

Here is an example of adding a new dimension:

```
TDF_Label aDimLabel = myDimTolTool->AddDimension();
if (!aDimLabel.IsNull())
{
  // error processing
}
```

A similar approach can be used for other GD&T types.

### 2.7.3 Editing a GD&T

A newly added GD&T entity is empty. To set its data a corresponding access object should be used as it is demonstrated below, where the dimension becomes a linear distance between two points.

```
Handle(XCAFDoc_Dimension) aDimAttr;
aDimLabel.FindAttribute(XCAFDoc_Dimension::GetID(), aDimAttr);
if (!aDimAttr.IsNull())
{
  Handle(XCAFDimTolObjects_DimensionObject) aDimObject = aDimAttr->GetObject();
```

```
  // set dimension data
  aDimObject->SetType(XCAFDimTolObjects_DimensionType_Location_LinearDistance);
  aDimObject->SetPoint(thePnt1);  // the first reference point
  aDimObject->SetPoint2(thePnt2); // the second reference point
  aDimObject->SetValue(theValue); // the distance value
  //...
  aDimAttr->SetObject(aDimObject);
}
```

A similar approach can be used for other GD&T types.

#### 2.7.4 Linking GD&Ts

To link a GD&T entity with other OCAF labels (e.g. representing shapes) one should use the following methods:

- *SetDimension* - for dimensions;
- *SetGeomTolerance* - for geometric tolerances;
- *SetDatum* - for geometric tolerance datums.

These methods can take a single label or a sequence of labels. All previous links will be removed.

The example below demonstrates linking of a dimension to sequences of shape labels:

```
TDF_LabelSequence aShapes1, aShapes2;
aShapes1.Append(aShape11);
//...
aShapes2.Append(aShape21);
//...
aDGTTool->SetDimension(aShapes1, aShapes2, aDimLabel);
```

In addition, a special method *SetDatumToGeomTol* should be used to link a datum with a geometric tolerance.

#### 2.7.5 Finding GD&Ts and reference shapes

*XCAFDimTolObjects_Tool* class provides basic capabilities for searching GD&Ts linked to shapes. The tool provides sequences of dimensions, geometric tolerances and datums linked with a shape. A series of related datums is also returned for geometric tolerances.

To get reference shapes for a GD&T entity one can use *GetRefShapeLabel* from *XCAFDoc_DimTolTool*.

*XCAFDoc_DimTolTool* provides methods to get lists of all dimensions, geometric tolerances and datums.

#### 2.7.6 Storing custom data

Every GD&T entity in XDE is represented as a label with attached attribute identifying entity type. All specific data is stored in sub-labels in standard OCAF attributes, such as *TDataStd_Integer*, *TDataStd_IntegerArray*, *TDataStd_↩ RealArray* and so on. Sub-label tags are reserved for internal use and cannot be used for storing custom data. The following tag ranges are reserved for GD&T entities:

- 1 - 17 - for dimensions;
- 1 - 17 - for geometric tolerances;
- 1 - 19 - for datums. Custom data can be stored in labels with tags beyond the ranges listed above.

### 2.8 Clipping planes

In an XDE document, Clipping planes are managed by the class *XCAFDoc_ClippingPlaneTool*. It works basing on the same principles as ShapeTool works with Shapes. This tool can be provided on the Main Label or on any

sub-label. Clipping planes are stored in a child of the starting document label 0.1.8, where planes themselves are defined as *TDataXtd_Plane* attribute. *TDataStd_Name* attribute is used for naming.

To query, edit, or initialize a Document to handle clipping planes of XCAF, use:

```
Handle(XCAFDoc_ClippingPlaneTool) myClipPlaneTool =
XCAFDoc_DocumentTool::ClippingPlaneTool(Doc->Main());
```

This call can be used at any time. When it is used for the first time, a relevant structure is added to the document.

To add a clipping plane use one of overloaded methods *AddClippingPlane*, e.g.:

```
gp_Pln aPln = ...
Standard_Boolean aCapping = ...
TDF_Label aClipPlnLbl = myClipPlaneTool->AddClippingPlane(aPln, "Name of plane", aCapping);
if (aClipPlnLbl.IsNull())
{
  // error processing
}
```

To remove a plane use *RemoveClippingPlane* method, e.g.:

```
if (!myClipPlaneTool->RemoveClippingPlane(aClipPlnLbl))
{
  // not removed
}
```

The plane will not be removed if it is referenced in at least one view.

To change the clipping plane and its name use *UpdateClippingPlane* method, e.g.:

```
gp_Pln aPln = ...
myClipPlaneTool->UpdateClippingPlane(aClipPlnLbl, aPln, "New name of plane");
```

Capping property can be changed using *SetCapping* method, e.g.:

```
Standard_Boolean aCapping = ...
myClipPlaneTool->SetCapping(aClipPlnLbl, aCapping);
```

*XCAFDoc_ClippingPlaneTool* can be used to get all clipping plane labels and to check if a label belongs to the *ClippingPlane table*, e.g.:

```
TDF_LabelSequence aClipPlaneLbls;
myClipPlaneTool->GetClippingPlanes(aClipPlaneLbls);
...
for (TDF_LabelSequence::Iterator anIt(aClipPlaneLbls); anIt.More(); anIt.Next())
{
  if (myClipPlaneTool->IsClippingPlane(anIt.Value()))
  {
    // the label is a clipping plane
    gp_Pln aPln;
    TCollection_ExtendedString aName;
    Standard_Boolean aCapping;
    if (!myClipPlaneTool->GetClippingPlane(anIt.Value(), aPln, aName, aCapping))
    {
      // error processing
    }
    ...
  }
}
```

## 2.9 Saved views

In an XDE document, Views are managed by the class *XCAFDoc_ViewTool*. It works basing on the same principles as ShapeTool works with Shapes. This tool can be provided on the Main Label or on any sub-label. Views are stored in a child of the starting document label 0.1.7, where a view itself is defined as *XCAFDoc_View* sub-class of *TDF_Attribute*. Views and selected shapes, clipping planes, GD&Ts and notes are related by Graph Nodes.

To query, edit, or initialize a Document to handle views of XCAF, use:

```
Handle(XCAFDoc_ViewTool) myViewTool =
XCAFDoc_DocumentTool::ViewTool(Doc->Main());
```

This call can be used at any time. When it is used for the first time, a relevant structure is added to the document.

To add a view use *AddView* method and an access *XCAFView_Object* object to set camera parameters, e.g.:

```
TDF_Label aViewLbl = myViewTool->AddView();
if (aViewLbl.IsNull())
{
  // error processing
}
Handle(XCAFDoc_View) aViewAttr;
aViewLbl.FindAttribute(XCAFDoc_View::GetID(), aViewAttr);
if (!aViewAttr.IsNull())
{
  Handle(XCAFView_Object) aViewObject = aViewAttr->GetObject();
  // set view data
  aViewObject->SetType(XCAFView_ProjectionType_Parallel);
  aViewObject->SetViewDirection(theViewDir);
  aViewObject->SetZoomFactor(2.0);
  ...
  aViewAttr->SetObject(aViewObject);
}
```

To set shapes, clipping planes, GD&Ts and notes selected for the view use one of overloaded *SetView* methods of *XCAFDoc_ViewTool*. To set only clipping planes one should use *SetClippingPlanes* method.

```
TDF_LabelSequence aShapes; ...
TDF_LabelSequence aGDTs; ...
myViewTool->SetView(aShapes, aGDTs, aViewLbl);
TDF_LabelSequence aClippingPlanes; ...
myViewTool->SetClippingPlanes(aClippingPlanes, aViewLbl);
```

To remove a view use *RemoveView* method.

To get all view labels and check if a label belongs to the View table use:

```
TDF_LabelSequence aViewLbls;
myViewTool->GetViewLabels(aViewLbls);
...
for (TDF_LabelSequence::Iterator anIt(aViewLbls); anIt.More(); anIt.Next())
{
  if (myViewTool->IsView(anIt.Value()))
  {
    // the label is a view
    ...
  }
}
```

To get shapes, clipping planes, GD&Ts or notes associated with a particular view use the following methods:

- *GetRefShapeLabel* - returns a sequence of associated shape labels;
- *GetRefGDTLabel* - returns a sequence of associated GDT labels;
- *GetRefClippingPlaneLabel* - returns a sequence of associated clipping plane labels;
- *GetRefNoteLabel* - returns a sequence of associated note labels;
- *GetRefAnnotationLabel* - returns a sequence of associated annotated labels.

And vice versa, to get views that display a particular clipping plane, GD&T or note use the following methods:

- *GetViewLabelsForShape* - returns a sequence of view labels associated with a shape;
- *GetViewLabelsForGDT* - returns a sequence of view labels associated with a GD&T;
- *GetViewLabelsForClippingPlane* - returns a sequence of view labels associated with a clipping plane;
- *GetViewLabelsForNote* - returns a sequence of view labels associated with a note;
- *GetViewLabelsForAnnotation* - returns a sequence of view labels associated with an annotated label.

## 2.10 Custom notes

In an XDE document, custom notes are managed by the class *XCAFDoc_NotesTool*. It works basing on the same principles as ShapeTool works with Shapes. This tool can be provided on the Main Label or on any sub-label. The Property itself is defined as sub-class of *XCAFDoc_Note* abstract class, which is a sub-class of *TDF_Attribute* one.

Custom notes are stored in a child of the *XCAFDoc_NotesTool* label, at label 0.1.9.1. Each note then corresponds to a dedicated label. A note may be attached to a document item identified by a label, a sub-shape identified by integer index or an attribute identified by GUID. Annotations are stored in a child of the *XCAFDoc_NotesTool* label, at label 0.1.9.2. Notes binding is done through *XCAFDoc_GraphNode* attribute.

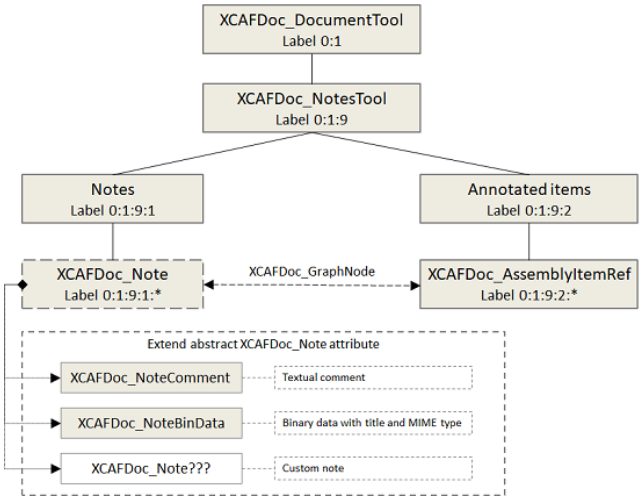


Figure 8: Structure of notes part of XCAF document

### 2.10.1 Initialization

To query, edit, or initialize a Document to handle custom notes of XCAF, use:

```
Handle(XCAFDoc_NotesTool) myNotes =
XCAFDoc_DocumentTool::NotesTool(Doc->Main ());
```

This call can be used at any time. The first time it is used, a relevant structure is added to the document. This definition is used for all later notes calls and will not be repeated for them.

### 2.10.2 Creating Notes

Before annotating a Document item a note must be created using one of the following methods of *XCAFDoc_↩ NotesTool* class:

- *CreateComment* : creates a note with a textual comment;
- *CreateBinData* : creates a note with arbitrary binary data, e.g. contents of a file.

Both methods return an instance of *XCAFDoc_Note* class.

```
Handle(XCAFDoc_NotesTool) myNotes = ...
Handle(XCAFDoc_Note) myNote = myNotes->CreateComment("User", "Timestamp", "Hello, World!");
```

This code adds a child label to label 0.1.9.1 with *XCAFDoc_NoteComment* attribute.

### 2.10.3 Editing a Note

An instance of *XCAFDoc_Note* class can be used for note editing. One may change common note data.

```
myNote->Set("New User", "New Timestamp");
```

To change specific data one needs to down cast *myNote* handle to the appropriate sub-class:

```
Handle(XCAFDoc_NoteComment) myCommentNote = Handle(XCAFDoc_NoteComment)::DownCast(myNote);
if (!myCommentNote.IsNull()) {
  myCommentNote->Set("New comment");
}
```

In order to edit auxiliary note data such as text and attachment position, plane for rendering and tesselated presentation, one should use a transfer object *XCAFNoteObjects_NoteObject* by GetObject and SetObject methods of *XCAFDoc_Note* class. *XCAFNoteObjects_NoteObject* class provides the following functionality:

- HasPlane, GetPlane and SetPlane methods test, get and set plane for note rendering
- HasPoint, GetPoint and SetPoint methods test, get and set note attachment position on the annotated object
- HasPointText, GetPointText, SetPointText methods test, get and set test position
- GetPresentation and SetPresentation methods allow to test for and specify tesselated presentation

After getting, the transfer object can be edited and set back to the note:

```
Handle(XCAFNoteObjects_NoteObject) aNoteObj = myNote->GetObject();
if (!aNoteObj.IsNull())
{
  gp_Pnt aPntTxt (...);
  aNoteObj->SetPointText (aPntTxt);
  TopoDS_Shape aS = ...;
  aNoteObj->SetPresentation (aS);
  myNote->SetObject (aNoteObj);
}
```

### 2.10.4 Adding Notes

Once a note has been created it can be bound to a Document item using the following *XCAFDoc_NotesTool* methods:

- *AddNote* : binds a note to a label;
- *AddNoteToAttr* : binds a note to a label's attribute;
- *AddNoteToSubshape* : binds a note to a sub-shape.

All methods return a pointer to *XCAFDoc_AssemblyItemRef* attribute identifying the annotated item.

```
Handle(XCAFDoc_NotesTool) myNotes = ...
Handle(XCAFDoc_Note) myNote = ...
TDF_Label theLabel; ...
Handle(XCAFDoc_AssemblyItemRef) myRef = myNotes->AddNote(myNote->Label(), theLabel);
Standard_GUID theAttrGUID; ...
Handle(XCAFDoc_AssemblyItemRef) myRefAttr = myNotes->AddNoteToAttr(myNote->Label(), theAttrGUID);
Standard_Integer theSubshape = 1;
Handle(XCAFDoc_AssemblyItemRef) myRefSubshape = myNotes->AddNoteToSubshape(myNote->Label(), theSubshape);
```

This code adds three child labels with *XCAFDoc_AssemblyItemRef* attribute to label 0.1.9.2. *XCAFDoc_GraphNode* attributes are added to the child labels and note labels.

### 2.10.5 Finding Notes

To find annotation labels under label 0.1.9.2 use the following *XCAFDoc_NotesTool* methods:

- *FindAnnotatedItem* : returns an annotation label for a label;
- *FindAnnotatedItemAttr* : returns an annotation label for a label's attribute;
- *FindAnnotatedItemSubshape* : returns an annotation label for a sub-shape.

```
Handle(XCAFDoc_NotesTool) myNotes = ...
TDF_Label theLabel; ...
TDF_Label myLabel = myNotes->FindAnnotatedItem(theLabel);
Standard_GUID theAttrGUID; ...
TDF_Label myLabelAttr = myNotes->FindAnnotatedItemAttr(theLabel, theAttrGUID);
Standard_Integer theSubshape = 1;
TDF_Label myLabelSubshape = myNotes->FindAnnotatedItemSubshape(theLabel, theSubshape);
```

Null label will be returned if there is no corresponding annotation.

To get all notes of the Document item use the following *XCAFDoc_NotesTool* methods:

- *GetNotes* : outputs a sequence of note labels bound to a label;
- *GetAttrNotes* : outputs a sequence of note labels bound to a label's attribute;
- *GetAttrSubshape* : outputs a sequence of note labels bound to a sub-shape.

All these methods return the number of notes.

```
Handle(XCAFDoc_NotesTool) myNotes = ...
TDF_Label theLabel; ...
TDF_LabelSequence theNotes;
myNotes->GetNotes(theLabel, theNotes);
Standard_GUID theAttrGUID; ...
TDF_LabelSequence theNotesAttr;
myNotes->GetAttrNotes(theLabel, theAttrGUID, theNotesAttr);
Standard_Integer theSubshape = 1;
TDF_LabelSequence theNotesSubshape;
myNotes->GetAttrSubshape(theLabel, theSubshape, theNotesSubshape);
```

### 2.10.6 Removing Notes

To remove a note use one of the following *XCAFDoc_NotesTool* methods:

- *RemoveNote* : unbinds a note from a label;
- *RemoveAttrNote* : unbinds a note from a label's attribute;
- *RemoveSubshapeNote* : unbinds a note from a sub-shape.

```
Handle(XCAFDoc_Note) myNote = ...
TDF_Label theLabel; ...
myNotes->RemoveNote(myNote->Label(), theLabel);
Standard_GUID theAttrGUID; ...
myRefAttr = myNotes->RemoveAttrNote(myNote->Label(), theAttrGUID);
Standard_Integer theSubshape = 1;
myNotes->RemoveSubshapeNote(myNote->Label(), theSubshape);
```

A note will not be deleted automatically. Counterpart methods to remove all notes are available, too.

### 2.10.7 Deleting Notes

To delete note(s) use the following *XCAFDoc_NotesTool* methods:

- *DeleteNote* : deletes a single note;

- *DeleteNotes* : deletes a sequence of notes;
- *DeleteAllNotes* : deletes all Document notes;
- *DeleteOrphanNotes* : deletes notes not bound to Document items.

All these methods except for the last one break all links with Document items as well.

## 2.11 Reading and Writing STEP or IGES

Note that saving and restoring the document itself are standard OCAF operations. As the various previously described definitions enter into this frame, they will not be explained any further. The same can be said for Viewing: presentations can be defined from Shapes and Colors.

There are several important points to consider:

- Previously defined Readers and Writers for dealing with Shapes only, whether Standard or Advanced, remain unchanged in their form and in their dependencies. In addition, functions other than mapping are also unchanged.
- XDE provides mapping with data other than Shapes. Names, Colors, Layers, GD&T, Clipping planes, Views, Validation Properties (Centroid, Volume, Area), and Assembly Structure are hierarchic with rigid motion. Currently, Clipping planes and Views writing supported for XBF format only.
- XDE mapping is relevant for use within the Advanced level of Data Exchanges, rather than Standard ones, because a higher level of information is better suited to a higher quality of shapes. In addition, this allows to avoid the multiplicity of combinations between various options. Note that this choice is not one of architecture but of practical usage and packaging.
- Reader and Writer classes for XDE are generally used like those for Shapes. However, their use is adapted to manage a Document rather than a Shape.

The packages to manage this are *IGESCAFControl* for IGES, and *STEPCAFControl* for STEP.

### 2.11.1 Reading a STEP file

To read a STEP file by itself, use:

```
STEPCAFControl_Reader reader;
IFSelect_ReturnStatus readstat = reader.ReadFile(filename);
// The various ways of reading a file are available here too :
// to read it by the reader, to take it from a WorkSession ...
Handle(TDocStd_Document) doc...
// the document referred to is already defined and
// properly initialized.
// Now, the transfer itself
if ( !reader.Transfer ( doc ) ) {
  cout;Cannot read any relevant data from the STEP file;endl;
  // abandon ..
}
// Here, the Document has been filled from a STEP file,
// it is ready to use
```

In addition, the reader provides methods that are applicable to document transfers and for directly querying of the data produced.

### 2.11.2 Writing a STEP file

To write a STEP file by itself, use:

```
STEPControl_StepModelType mode =
STEPControl_AsIs;
// Asis is the recommended value, others are available
// Firstly, perform the conversion to STEP entities
STEPCAFControl_Writer writer;
//(the user can work with an already prepared WorkSession or create a //new one)
Standard_Boolean scratch = Standard_False;
STEPCAFControl_Writer writer ( WS, scratch );
// Translating document (conversion) to STEP
if ( ! writer.Transfer ( Doc, mode ) ) {
  cout;The document cannot be translated or gives no result;endl;
  // abandon ..
}
// Writing the File
IFSelect_ReturnStatus stat = writer.Write(file-name);
```

#### 2.11.3 Reading an IGES File

Use the same procedure as for a STEP file but with IGESCAFControl instead of STEPCAFControl.

#### 2.11.4 Writing an IGES File

Use the same procedure as for a STEP file but with IGESCAFControl instead of STEPCAFControl.

### 2.12 Using an XDE Document

There are several ways of exploiting XDE data from an application, you can:

1. Get the data relevant for the application by mapping XDE/Appli, then discard the XDE data once it has been used.
2. Create a reference from the Application Document to the XDE Document, to have its data available as external data.
3. Embed XDE data inside the Application Document (see the following section for details).
4. Directly exploit XDE data such as when using file checkers.

#### 2.12.1 XDE Data inside an Application Document

To have XCAF data elsewhere than under label 0.1, you use the DocLabel of XDE. The method DocLabel from XCAFDoc_DocumentTool determines the relevant Label for XCAF. However, note that the default is 0.1.

In addition, as XDE data is defined and managed in a modular way, you can consider exclusively Assembly Structure, only Colors, and so on.

As XDE provides an extension of the data structure, for relevant data in standardized exchanges, note the following:

- This data structure is fitted for data exchange, rather than for use by the final application.
- The provided definitions are general, for common use and therefore do not bring strongly specific semantics.

As a result, if an application works on Assemblies, on Colors or Layers, on Validation Properties (as defined in STEP), it can rely on all or a part of the XDE definitions, and include them in its own data structure.

In addition, if an application has a data structure far from these notions, it can get data (such as Colors and Names on Shapes) according to its needs, but without having to consider the whole.