# OCAF

# Introduction

This manual explains how to use the Open CASCADE Application Framework (OCAF). It provides basic documentation on using OCAF.

## Purpose of OCAF

OCAF (the Open CASCADE Application Framework) is an easy-to-use platform for rapidly developing sophisticated domain-specific design applications. A typical application developed using OCAF deals with two or three-dimensional (2D or 3D) geometric modeling in trade-specific Computer Aided Design (CAD) systems, manufacturing or analysis applications, simulation applications or illustration tools.

Developing a design application requires addressing many technical aspects. In particular, given the functional specification of your application, you must at least:

- Design the architecture of the application — definition of the software components and the way they cooperate;
- Define the data model able to support the functionality required — a design application operates on data maintained during the whole end-user working session;
- Structure the software in order to:
  - synchronize the display with the data — commands modifying objects must update the views;
  - support generalized undo-redo commands — this feature has to be taken into account very early in the design process;

**Table of Contents**

- Implement the function for saving the data — if the application has a long life cycle, the compatibility of data between versions of the application has to be addressed;
- Build the application user interface.

Architectural guidance and ready-to-use solutions provided by OCAF offer you the following benefits:

- You can concentrate on the functionality specific for your application;
- The underlying mechanisms required to support the application are already provided;
- The application can be rapidly be prototyped thanks to the coupling the other Open CASCADE Technology modules;
- The final application can be developed by industrializing the prototype — you don't need to restart the development from scratch.
- The Open Source nature of the platform guarantees the long-term usefulness of your development.

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 their logical supplement.

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

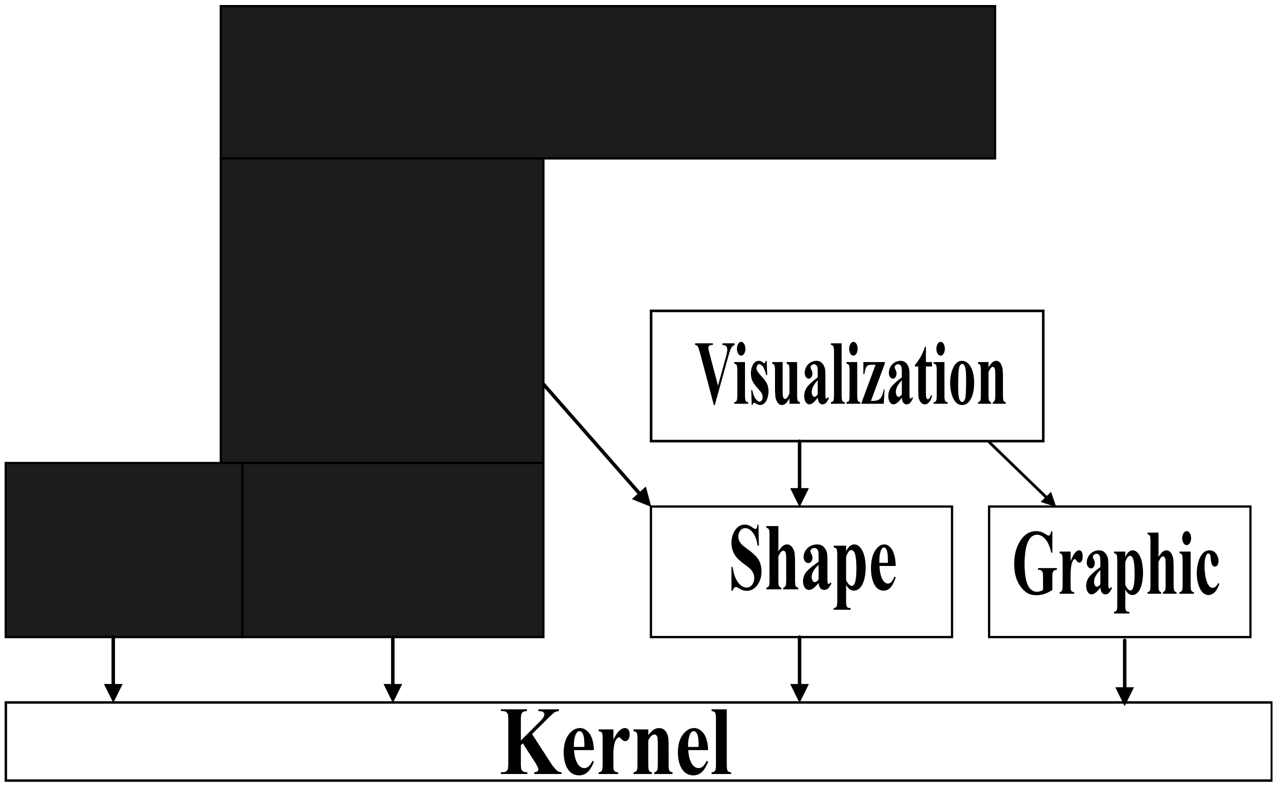
**Table 1: Services provided by OCAF**

| Development tasks | Comments | 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 |

OCAF uses other modules of Open CASCADE Technology — the Shape is implemented with the geometry supported by the Modeling Data module and the viewer is the one provided with the Visualization module. Modeling functions can be implemented using the Modeling Algorithms module.

The relationship between OCAF and the Open CASCADE Technology (**OCCT**) Object Libraries can be seen in the image below.



■ OCAF (Open CASCADE Application Framework)

□ OCCT Object Libraries required by OCAF

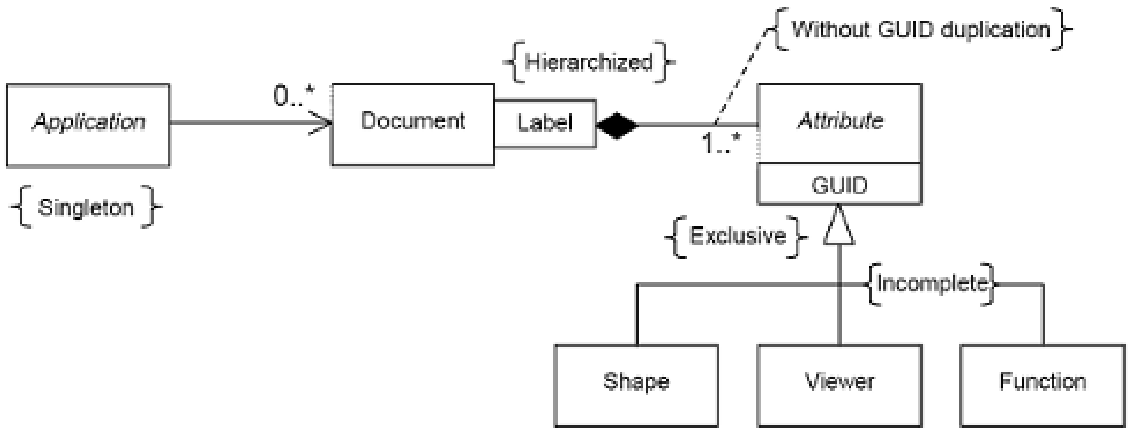
**OCCT Architecture**

In the image, the OCAF (Open CASCADE Application Framework) is shown with black rectangles and OCCT Object Libraries required by OCAF are shown with white rectangles.

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

## Architecture Overview

OCAF provides you with an object-oriented Application-Document-Attribute model consisting of C++ class libraries.



**The Application-Document-Attribute model**

### Application

The *Application* is an abstract class in charge of handling documents during the working session, namely:

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

### Document

The document, implemented by the concrete class *Document*, is the container for the application data. Documents offer access to the data framework and serve the following purposes:

- 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.

Each document is saved in a single flat ASCII file defined by its format and extension (a ready-to-use format is provided with OCAF).

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.

### Attribute

Application data is described by **Attributes**, which are instances of classes derived from the *Attribute* abstract class, organized according to the OCAF Data Framework.

The **OCAF Data Framework** references aggregations of attributes using persistent identifiers in a single hierarchy. A wide range of attributes come with OCAF, including:

- **Standard attributes** 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)....;
- **Shape attributes** contain the geometry of the whole model or its elements including reference to the shapes and tracking of shape evolution;
- Other geometric attributes such as **Datums** (points, axis and plane) and **Constraints** (*tangent-to, at-a-given-distance, from-a-given-angle, concentric,* etc.)
- User attributes, that is, attributes typed by the application
- **Visualization 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.
- **Function services** — the purpose of these attributes is to rebuild objects after they have been modified (parameterization of models). While 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.

In addition, application-specific data can be added by defining new attribute classes; naturally, this changes the standard file format. The only functions that have to be implemented are:

- Copying the attribute
- Converting it from and persistent data storage

## Reference-key model

In most existing geometric modeling systems, the data are topology driven. They usually use a boundary representation (BRep), where geometric models are defined by a collection of faces, edges and vertices, to which application data are attached. Examples of data include:

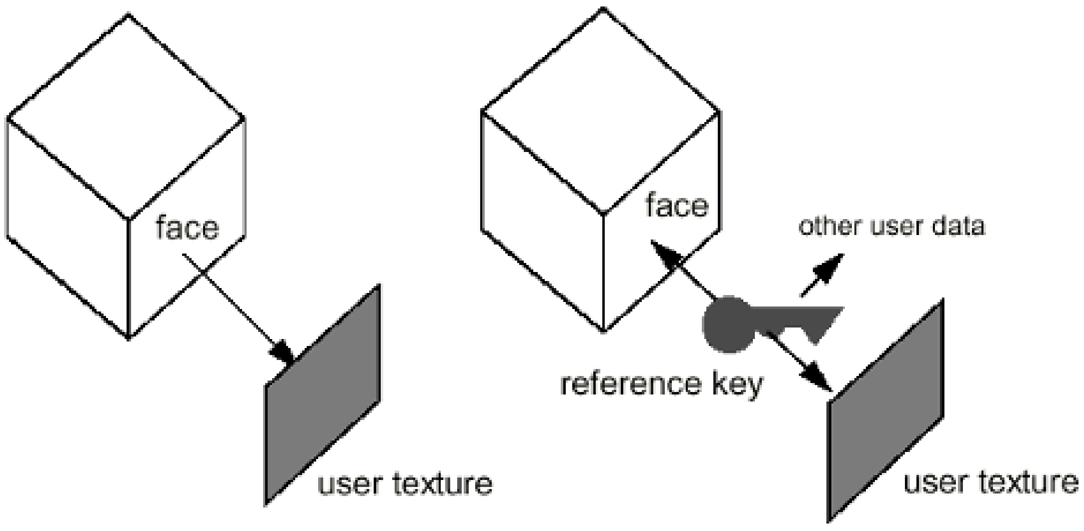
- a color;
- a material;

* information that a particular edge is blended.

When the geometric model is parameterized, that is, when you can change the value of parameters used to build the model (the radius of a blend, the thickness of a rib, etc.), the geometry is highly subject to change. In order to maintain the attachment of application data, the geometry must be distinguished from other data.

In OCAF, the data are reference-key driven. It is a uniform model in which reference-keys are the persistent identification of data. All **accessible** data, including the geometry, are implemented as attributes attached to reference-keys. The geometry becomes the value of the Shape attribute, just as a number is the value of the Integer and Real attributes and a string that of the Name attribute.

On a single reference-key, many attributes can be aggregated; the application can ask at runtime which attributes are available. For example, to associate a texture to a face in a geometric model, both the face and the texture are attached to the same reference-key.



**Topology driven versus reference-key driven approaches**

Reference-keys can be created in two ways:

* At programming time, by the application
* At runtime, by the end-user of the application (providing that you include this capability in the application)

As an application developer, you generate reference-keys in order to give semantics to the data. For example, a function building a prism may create three reference-keys: one for the base of the prism, a second for the lateral faces and a third for the top face. This makes up a semantic built-in the application's prism feature. On the other hand, in a command allowing the end-user to set a texture to a face he/she selects, you must create a reference-key to the selected face if it has not previously been referenced in any feature (as in the case of one of the lateral faces of the prism).

When you create a reference-key to selected topological elements (faces, edges or vertices), OCAF attaches to the reference-key information defining the selected topology — the Naming attribute. For example, it may be the faces to which a selected edge is common to. This information, as well as information about the evolution of the topology at each modeling step (the modified, updated and deleted faces), is used by the naming algorithm to maintain the topology attached to the reference-key. As such, on a parametrized model, after modifying the value of a parameter, the reference-keys still address the appropriate faces, even if their geometry has changed. Consequently, you change the size of the cube shown in the figure above, the user texture stay attached to the right face.

**Note** As Topological naming is based on the reference-key and attributes such as Naming (selection information) and Shape (topology evolution information), OCAF is not coupled to the underlying modeling libraries. The only modeling services required by OCAF are the following:

- Each algorithm must provide information about the evolution of the topology (the list of faces modified, updated and deleted by the algorithm)
- Exploration of the geometric model must be available (a 3D model is made of faces bounded by close wires, themselves composed by a sequence of edges connected by their vertices)

Currently, OCAF uses the Open CASCADE Technology modeling libraries.

To design an OCAF-based data model, the application developer is encouraged to aggregate ready-to-use attributes instead of defining new attributes by inheriting from an abstract root class.
There are two major advantages in using aggregation rather than inheritance:

- As you don't implement data by defining new classes, the format of saved data provided with OCAF doesn't change; so you don't have to write the Save and Open functions
- The application can query the data at runtime if a particular attribute is available

**Summary**

- OCAF is based on a uniform reference-key model in which:
    - Reference-keys provide persistent identification of data;
    - Data, including geometry, are implemented as attributes attached to reference-keys;
    - Topological naming maintains the selected geometry attached to reference-keys in parametrized models;
- In many applications, the data format provided with OCAF doesn't need to be extended;

- OCAF is not coupled to the underlying modeling libraries.

# The Data Framework

## Data Structure

The OCAF Data Framework is the Open CASCADE Technology realization of the reference-key model in a tree structure. It offers a single environment where data from different application components can be handled. This allows exchanging and modifying data simply, consistently, with a maximum level of information and stable semantics.

The building blocks of this approach are:

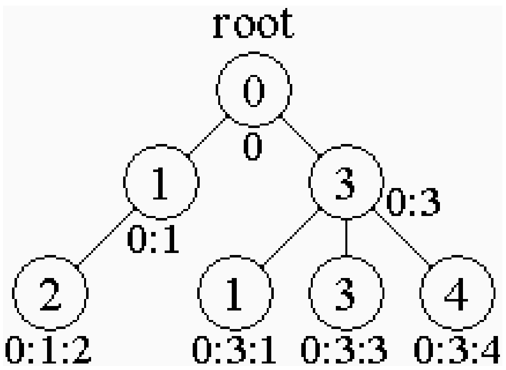
- The tag
- The label
- The attribute

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.



**A simple framework model**

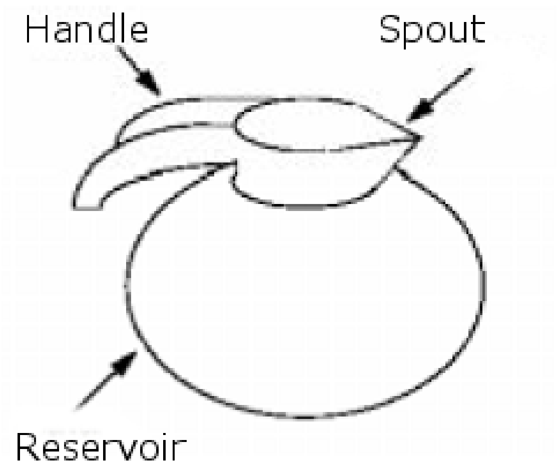
In this image the circles contain tags of the corresponding labels. The lists of tags are located under the circles. The root label always has a zero tag.

The children of a 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).

## Examples of a Data Structure

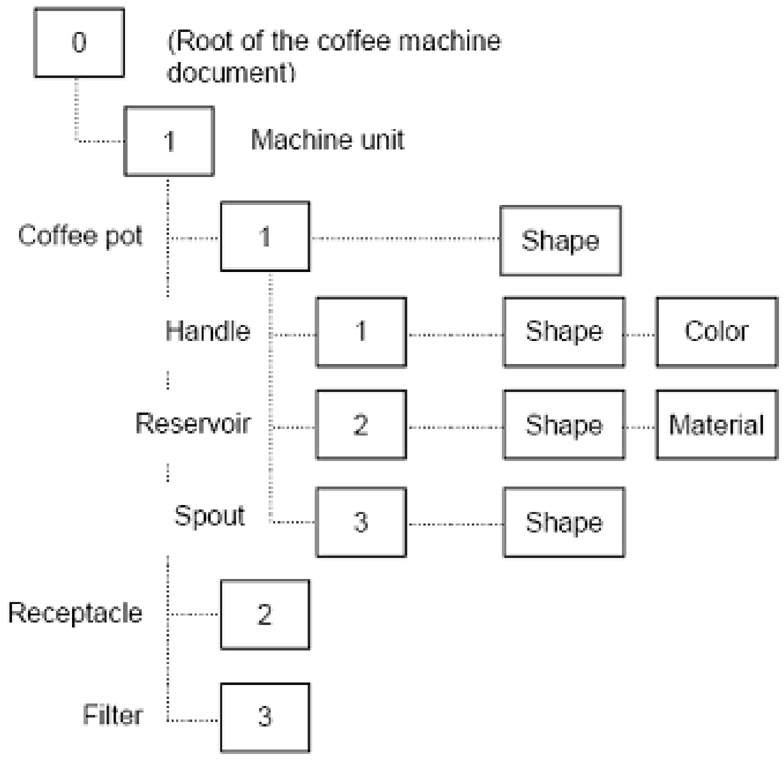
Let's have a look at the example:



**The coffee machine**

In the image the application for designing coffee machines first allocates a label for the machine unit. It then adds sub-labels for the main features (glass coffee pot, water receptacle and filter) which it refines as needed (handle and reservoir of the coffee pot and spout of the reservoir).

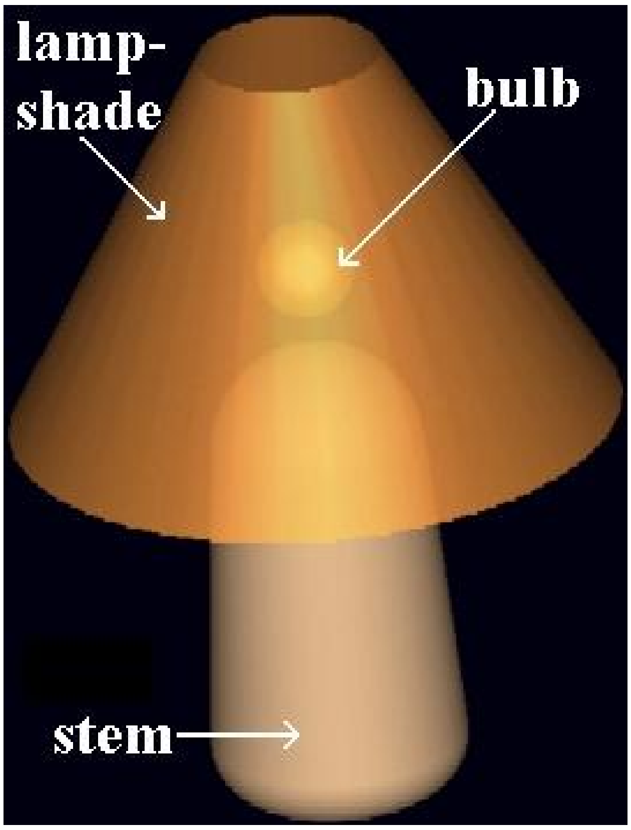
You now attach technical data describing the handle — its geometry and color — and the reservoir — its geometry and material. Later on, you can modify the handle's geometry without changing its color — both remain attached to the same label.



**The data structure of the coffee machine**

The nesting of labels is key to OCAF. This allows a label to have its own structure with its local addressing scheme which can be reused in a more complex structure. Take, for example, the coffee machine. Given that the coffee pot's handle has a label of tag [1], the entry for the handle in the context of the coffee pot only (without the machine unit) is [0:1:1]. If you now model a coffee machine with two coffee pots, one at the label [1], the second at the label [4] in the machine unit, the handle of the first pot would have the entry [0:1:1:1] whereas the handle of the second pot would be [0:1:4:1]. This way, we avoid any confusion between coffee pot handles.

Another example is the application for designing table lamps. The first label is allocated to the lamp unit.



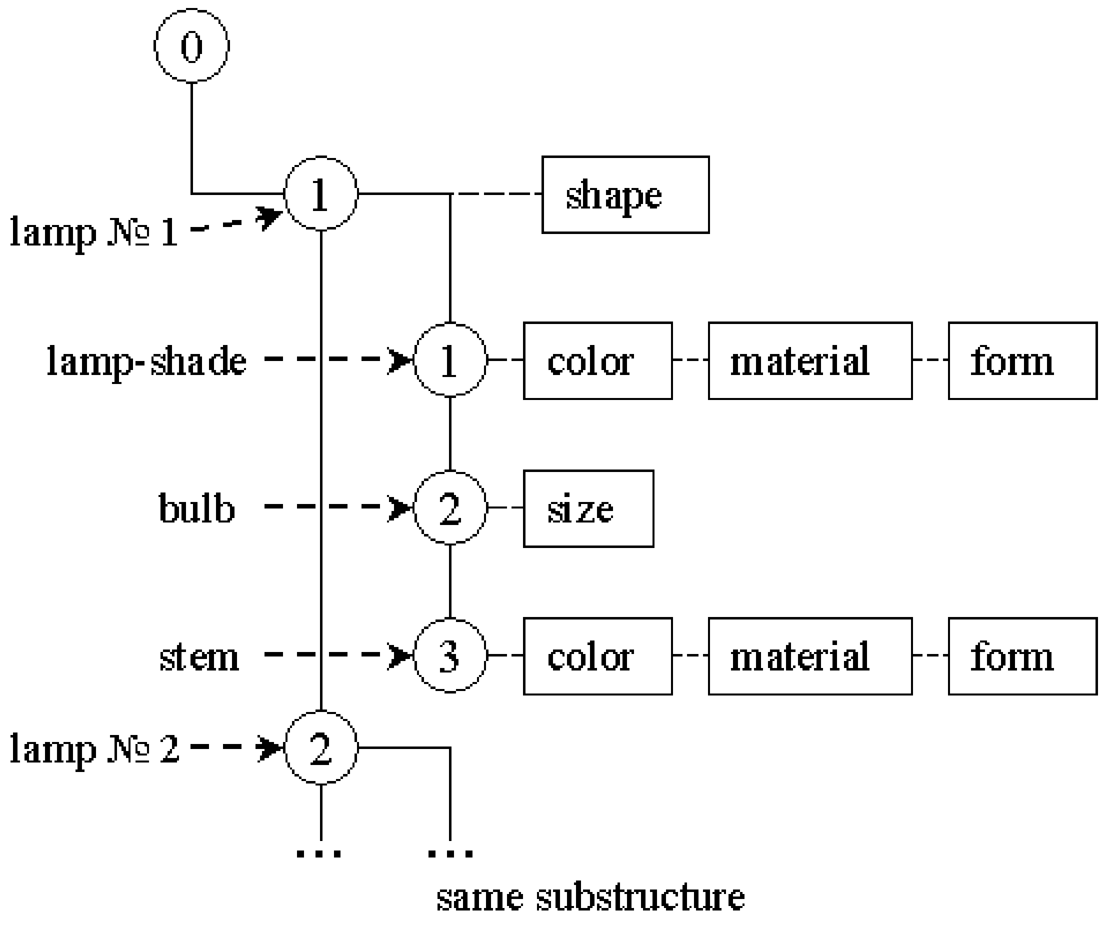
The root label cannot have brother labels. Consequently, various lamps in the framework allocation correspond to the sub-labels of the root label. This allows avoiding any confusion between table lamps in the data framework. Different lamp parts have different material, color and other attributes, so a child label of the lamp with the specified tags is allocated for each sub-unit of the lamp:

- 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 only identifiers of the lamp parts. Now you can refine all units: by setting geometry, color, material and other information about the lamp or its parts to the specified label. This information is placed into special attributes of the label: the pure label contains no data – it is only a key to access data.

Remember 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 reused 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. The 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.

## 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 retrieving 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.

### 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*.

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

### 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.

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

## 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.

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.

## 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. The class *TDF_Label* offers the above services.

## Creating child labels

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

```
//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.

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

## Retrieving child labels

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

```
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.

```
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.

```
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;
  }
}
```

### Retrieving the father label

Retrieving the father label of a current label.

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

## 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.

### 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.

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

## 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*.

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

## 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* to which *INT* is attached.

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

Note. There is an exception from this rule for some sub-set of Standard attributes. See for details chapter 6.Standard Attributes.

## 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, e.g. *current*.

```
// 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
}
```

## 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*.

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

### 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, 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)

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.

To enable possibility to save / restore the new attribute in XML format, do the following:

1. 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).
2. 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.
3. 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.

To enable possibility to save / restore the new attribute in binary format, do the following:

1. 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.
2. 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.
3. 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** and **Opening the document from a file** for the description of document save/open mechanisms.

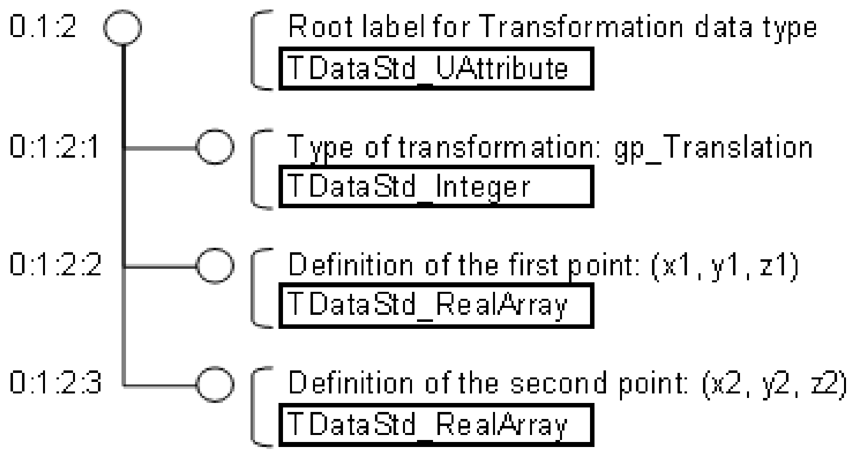
If you decided to use the alternative way (create a new attribute by means of *UAttribute* and a combination of other standard attributes), do the following:

1. 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).
2. Create child labels and allocate all necessary data through standard attributes at the child labels.
3. 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).

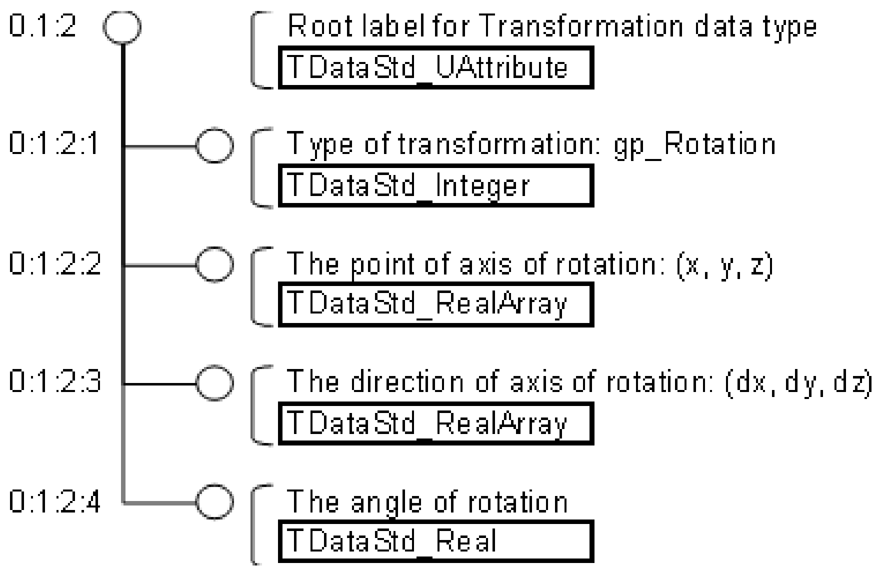
1. The first way: creation of a new attribute. The implementation of the transformation by creation of a new attribute is represented in the **Samples**.
2. The second way: creation of a new data type by means of combination of standard attributes. Depending on the type of transformation it may be kept 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

**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

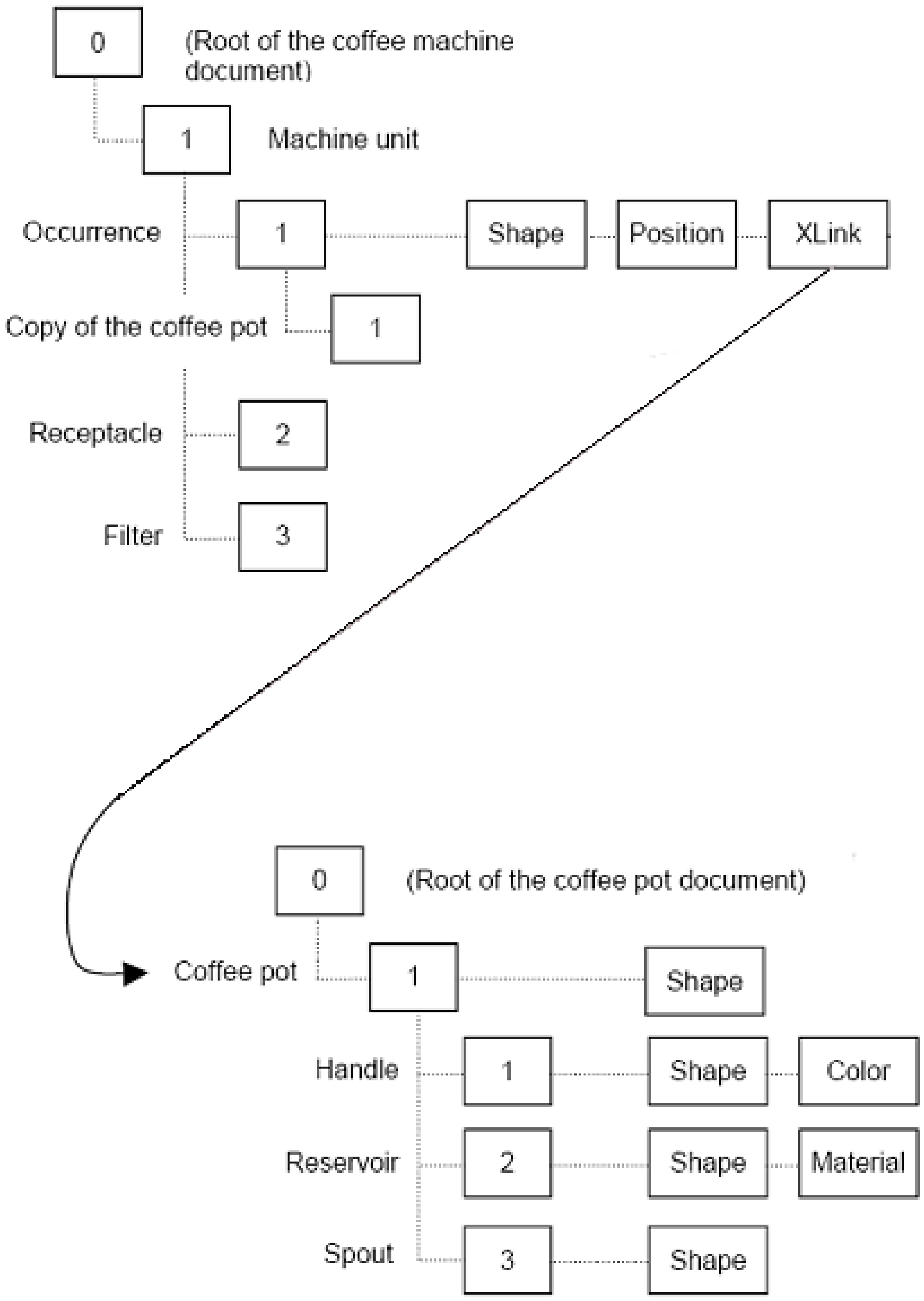
**Data tree for rotation**

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).

## Compound documents

As the identification of data is persistent, one document can reference data contained in another document, the referencing and referenced documents being saved in two separate files.

Lets look at the coffee machine application again. The coffee pot can be placed in one document. The coffee machine document then includes an *occurrence* — a positioned copy — of the coffee pot. This occurrence is defined by an XLink attribute (the external Link) which references the coffee pot of the first document (the XLink contains the relative path of the coffee pot document and the entry of the coffee pot data [0:1] ).



**The coffee machine compound document**

In this context, the end-user of the coffee machine application can open the coffee pot document, modify the geometry of, for example, the reservoir, and overwrite the document without worrying about the impact of the modification in the coffee machine document. To deal with this situation, OCAF provides a service which allows the application to check whether a document is up-to-date. This service is based on a modification counter included in each document: when an external link is created, a copy of the referenced document counter is associated to the XLink in the referencing document. Providing that each modification of the referenced document increments its own counter,
we can detect that the referencing document has to be updated by comparing the two counters (an update function importing the data referenced by an XLink into the referencing document is also provided).

## Transaction mechanism

The Data Framework also provides a transaction mechanism inspired from database management systems: the data are modified within a transaction which is terminated either by a Commit if the modifications are validated or by an Abort if the modifications are abandoned — the data are then restored to the state it was in prior to the transaction. This mechanism is extremely useful for:

- Securing editing operations (if an error occurs, the transaction is abandoned and the structure retains its integrity)
- Simplifying the implementation of the **Cancel** function (when the end-user begins a command, the application may launch a transaction and operate directly in the data structure; abandoning the action causes the transaction to Abort)
- Executing **Undo** (at commit time, the modifications are recorded in order to be able to restore the data to their previous state)

The transaction mechanism simply manages a backup copy of attributes. During a transaction, attributes are copied before their first modification. If the transaction is validated, the copy is destroyed. If the transaction is abandoned, the attribute is restored to its initial value (when attributes are added or deleted, the operation is simply reversed).

Transactions are document-centered, that is, the application starts a transaction on a document. So, modifying a referenced document and updating one of its referencing documents requires two transactions, even if both operations are done in the same working session.

# Standard Document Services

## 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 *TDocStd_Application* (or its descendant). 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.

## 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 *TDocStd_Application* or a class inheriting from it.

### Creating an application

To create an application, use the following syntax.

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

### 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*.

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

Here "NewDocumentFormat" is identifier of the format of your document. OCCT defines several standard formats, distinguishing by a set of supported OCAF attributes, and method of encoding (e.g. binary data or XML), described below. If your application defines specific OCAF attributes, you need to define your own format for it.

### Retrieving the application to which the document belongs

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

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

## 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.

### 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.

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

### 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*.

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

### Defining storage format

OCAF uses a customizable mechanism for storage of the documents. In order to use OCAF persistence to save and read your documents to / from the file, you need to define one or several formats in your application.

For that, use method TDocStd_Application::DefineFormat(), for instance:

```
app->DefineFormat ("NewDocumentFormat", "New format for OCAF documents",
       "ndf",
                   new NewDocumentFormat_RetrievalDriver(),
                   new NewDocumentFormat_StorageDriver());
```

This example defines format "NewDocumentFormat" with a default file extension "ndf", and instantiates drivers for reading and storing documents from and to that format. Either of the drivers can be null, in this case the corresponding action will not be supported for that format.

OCAF provides several standard formats, each covering some set of OCAF attributes:

| Format | Persistent toolkit | OCAF attributes covered |
|---|---|---|
| Legacy formats (read only) | | |

| | | |
|---|---|---|
| OCC-StdLite | TKStdL | TKLCAF |
| MDTV-Standard | TKStd | TKLCAF + TKCAF |
| Binary formats | | |
| BinLOcaf | TKBinL | TKLCAF |
| BinOcaf | TKBin | TKLCAF + TKCAF |
| BinXCAF | TKBinXCAF | TKLCAF + TKCAF + TKXCAF |
| TObjBin | TKBinTObj | TKLCAF + TKTObj |
| XML formats | | |
| XmlLOcaf | TKXmlL | TKLCAF |
| XmlOcaf | TKXml | TKLCAF + TKCAF |
| XmlXCAF | TKXmlXCAF | TKLCAF + TKCAF + TKXCAF |
| TObjXml | TKXmlTObj | TKLCAF + TKTObj |

For convenience, these toolkits provide static methods *DefineFormat()* accepting handle to application. These methods allow defining corresponding formats easily, e.g.:

```
BinDrivers::DefineFormat (app); // define format "BinOcaf"
```

Use these toolkits as an example for implementation of persistence drivers for custom attributes, or new persistence formats.

The application can define several storage formats. On save, the format specified in the document (see *TDocStd_Document::StorageFormat()*) will be used (save will fail if that format is not defined in the application). On reading, the format identifier stored in the file is used and recorded in the document.

## Defining storage format by resource files

The alternative method to define formats is via usage of resource files. This method was used in earlier versions of OCCT and is considered as deprecated since version 7.1.0. This method allows loading persistence drivers on demand, using plugin mechanism.

To use this method, create your own application class inheriting from *TDocStd_Application*, and override method *ResourcesName()*. That method should return a string with a name of resource file, e.g. "NewDocumentFormat", which will contain a description of the format.

Then create that resource file and define the parameters of your format:

```
ndf.FileFormat: NewDocumentFormat
NewDocumentFormat.Description: New Document Format Version 1.0
NewDocumentFormat.FileExtension: ndf
NewDocumentFormat.StoragePlugin: bb5aa176-c65c-4c84-862e-6b7c1fe16921
NewDocumentFormat.RetrievalPlugin: 76fb4c04-ea9a-46aa-88a2-25f6a228d902
```

The GUIDs should be unique and correspond to the GUIDs supported by relevant plugin. You can use an existing plugins (see the table above) or create your own.

Finally, make a copy of the resource file "Plugin" from *$CASROOT/src/StdResource* and, if necessary, add the definition of your plugin in it, for instance:

```
bb5aa176-c65c-4c84-862e-6b7c1fe16921.Location: TKNewFormat
76fb4c04-ea9a-46aa-88a2-25f6a228d902.Location: TKNewFormat
```

In order to have these resource files loaded during the program execution, it is necessary to set two environment variables: *CSF_PluginDefaults* and *CSF_NewFormatDefaults*. For example, set the files in the directory *MyApplicationPath/MyResources*:

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

## Saving a document

To save the document, make sure that its parameter *StorageFormat()* corresponds to one of the formats defined in the application, and use method *TDocStd_Application::SaveAs*, for instance:

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

## Opening the document from a file

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

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

For binary formats only the part of the stored document can be loaded. For that the *PCDM_ReadingFilter* class could be used. It is possible to define which attributes must be loaded or omitted, or to define one or several entries for sub-tree that must be loaded only. The following example opens document *doc*, but reads only "0:1:2" label and its sub-labels and only *TDataStd_Name* attributes on them.

```
Handle(PCDM_ReaderFilter) filter = new PCDM_ReaderFilter("0:1:2");
filter->AddRead("TDataStd_Name");
app->Open("example.cbf", doc, filter);
```

Also, using filters, part of the document can be appended into the already loaded document from the same file. For an example, to read into the previously opened *doc* all attributes, except *TDataStd_Name* and *TDataStd_Integer*:

```
Handle(PCDM_ReaderFilter) filter2 = new
        PCDM_ReaderFilter(PCDM_ReaderFilter::AppendMode_Protect);
filter2->AddSkipped("TDataStd_Name");
filter2->AddSkipped("TDataStd_Integer");
```

```
app->Open("example.cbf", doc, filter2);
```

*PCDM_ReaderFilter::AppendMode_Protect* means that if the loading algorithm finds already existing attribute in the document, it will not be overwritten by attibute from the loading file. If it is needed to substitute the existing attributes, the reading mode *PCDM_ReaderFilter::AppendMode_Overwrite* must be used instead.

*AddRead* and *AddSkipped* methods for attributes should not be used in one filter. If it is so, *AddSkipped* attributes are ignored during the read.

Appending to the document content of already loaded file may be performed several times with the same or different parts of the document loaded. For that the filter reading mode must be *PCDM_ReaderFilter::AppendMode_Protect* or *PCDM_ReaderFilter::AppendMode_Overwrite*, which enables the "append" mode of document open. If the filter is empty or null or skipped in arguments, it opens document with "append" mode disabled and any loading limitations.

## Cutting, copying and pasting inside a document

To cut, copy and paste inside a document, use the class *TDF_CopyLabel*.

In fact, you must define a *Label*, which contains the temporary value of a cut or copy operation (say, in *Lab_Clipboard*). You must also define two other labels:

- The data container (e.g. *Lab_source*)
- The destination of the copy (e.g. *Lab_ Target* )

```
Copy = copy (Lab_Source => Lab_Clipboard)
Cut = copy + Lab_Source.ForgetAll() // command clear the contents of
      LabelSource.
Paste = copy (Lab_Clipboard => Lab_target)
```

So we need a tool to copy all (or a part) of the content of a label and its sub-label, to another place defined by a label.

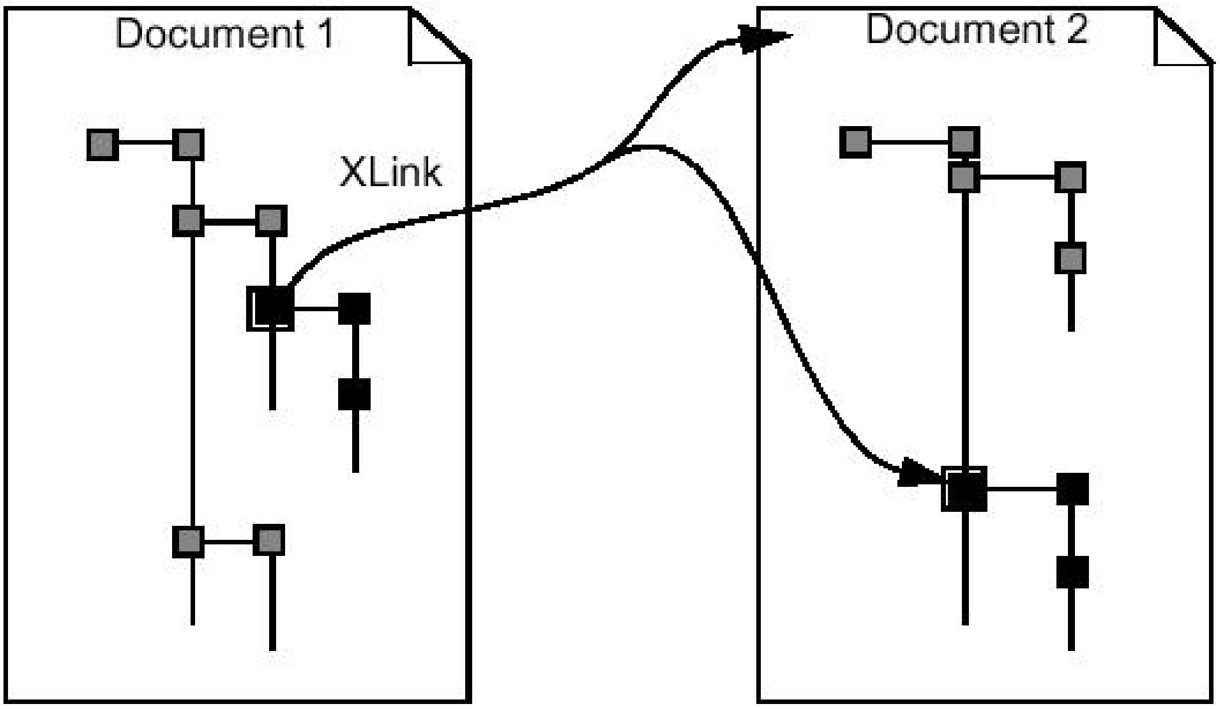
```
TDF_CopyLabel aCopy;
TDF_IDFilter aFilter (Standard_False);

//Don't copy TDataStd_TreeNode attribute

 aFilter.Ignore(TDataStd_TreeNode::GetDefaultTreeID());
 aCopy.Load(aSource, aTarget); aCopy.UseFilter(aFilter);
       aCopy.Perform();

// copy the data structure to clipboard

return aCopy.IsDone(); }
```

The filter is used to forbid copying a specified type of attribute.

You can also have a look at the class *TDF_Closure*, which can be useful to determine the dependencies of the part you want to cut from the document.

## External Links

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



**External links between documents**

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

### Copying the document

#### With the possibility of updating it later

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

```
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*.

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

### Without any 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.*

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

# OCAF Shape Attributes

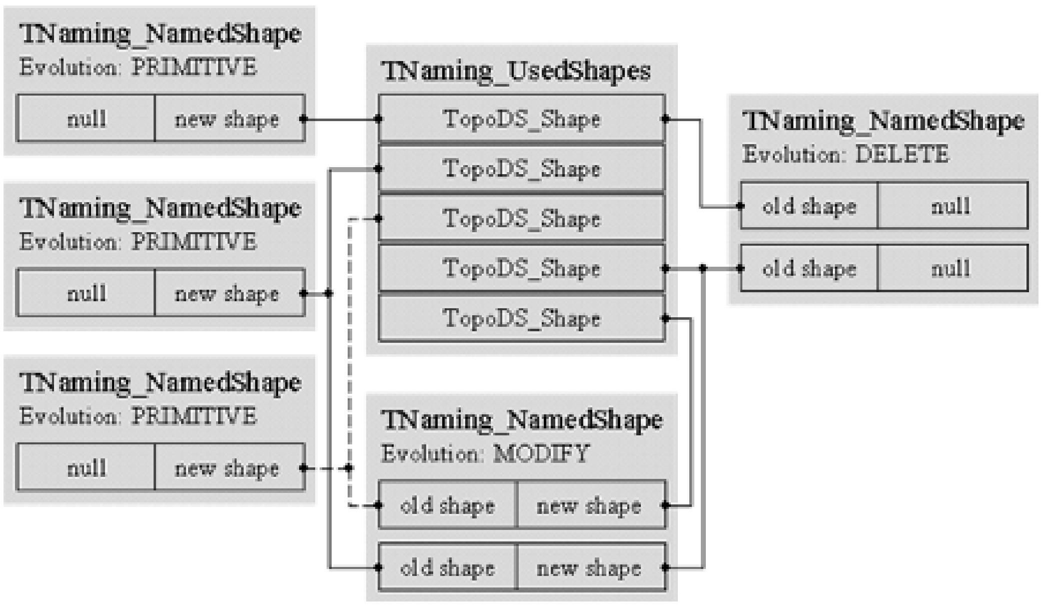
## Overview

A topological attribute can be seen as a hook into the topological structure. It is possible to attach data to define references to it.

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.



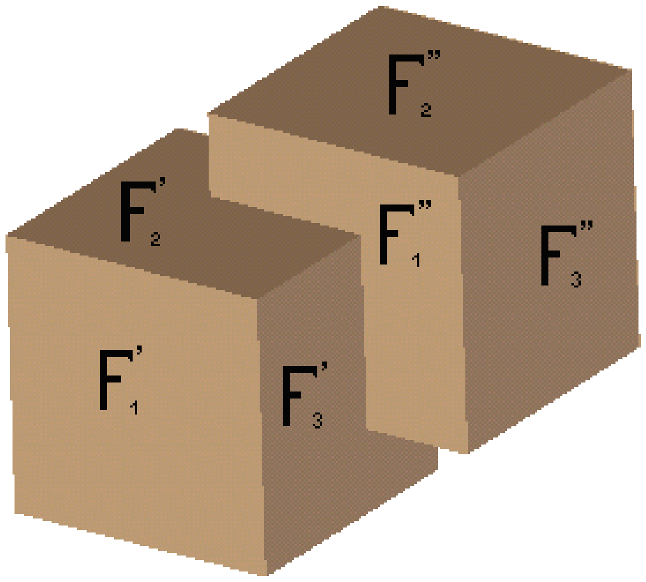
## Shape attributes in data framework.

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.

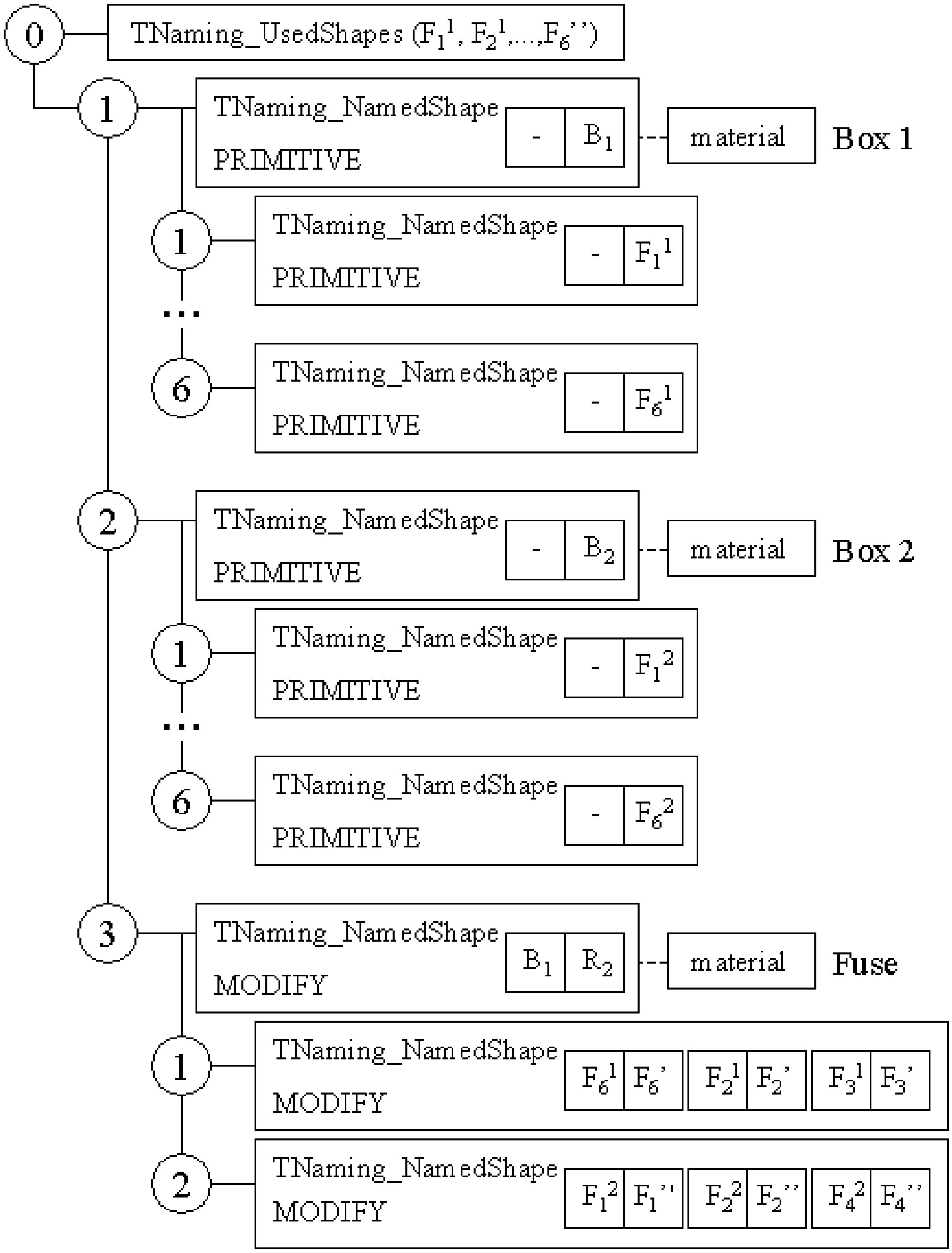


**Resulting box**

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) and to the new 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 from box 1,
- sub-label with tag 2 – modified faces from 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 face F11
- face F1" as generation of face F12
- 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.

## 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, which is a term of the *TNaming_Evolution* enumeration used for the selected shapes that are placed to the separate label:
    - 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.

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

## Using naming resources

The class *TNaming_Builder* allows creating 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.

```
// 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();
```

## Reading the contents of a named shape attribute

You can use the method *TNaming_NamedShape::Evolution()* to get the evolution of this named shape and the method *TNaming_NamedShape::Get()* to get a compound of new shapes of all pairs of this named shape.

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:
  - The method *GetShape(Handle(TNaming_NamedShape))* returns a compound of new shapes of the given named shape;
  - The method *CurrentShape(Handle(TNaming_NamedShape))* returns a compound of the shapes, which are latest versions of the shapes from the given named shape;
  - The method *NamedShape(TopoDS_Shape,TDF_Label)* returns a named shape, which contains a given shape as a new shape. A given label is any label from the data framework – it just gives access to it.
- *TNaming_Iterator* gives access to the named shape and hooks pairs.

```
// 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();
}
```

## Topological naming

The Topological Naming mechanism is based on 3 components:

- History of the used modeling operation algorithm;
- Registering of the built result in Data Framework (i.e. loading the necessary elements of the extracted history in OCAF document);
- Selection / Recomputation of a "selected" sub-shape of the algorithm result.

To get the expected result the work of the three components should be synchronized and the rules of each component should be respected.

## Algorithm history

The "correct" history of a used modeling operation serves the basis of naming mechanism. It should be provided by the algorithm supporting the operation. The history content depends on the type of the topological result. The purpose of the history is to provide all entities for consistent and correct work of the Selection / Recomputation mechanism. The table below presents expected types of entities depending on the result type.

| Result type | Type of sub-shapes to be returned by history of algorithm | Comments |
|---|---|---|
| Solid or closed shell | Faces | All faces |
| Open shell or single face | Faces and edges of opened boundaries only | All faces plus all edges of opened boundaries |
| Closed wire | Edges | All edges |
| Opened wire | Edges and ending vertexes | All edges plus ending vertexes of the wire |
| Edge | Vertexes | Two vertexes are expected |
| Compound or CompSolid | To be used consequentially the above declared rule applied to all sub-shapes of the first level | Compound/CompSolid to be explored level by level until any the mentioned above types will be met |

The history should return (and track) only elementary types of sub-shapes, i.e. Faces, Edges and Vertexes, while other so-called aggregation types: Compounds, Shells, Wires, are calculated by Selection mechanism automatically.

There are some simple exceptions for several cases. For example, if the Result contains a seam edge – in conical, cylindrical or spherical surfaces – this seam edge should be tracked by the history and in addition should be defined before the types. All degenerated entities should be filtered and excluded from consideration.

## Loading history in data framework

All elements returned by the used algorithm according to the aforementioned rules should be put in the Data Framework (or OCAF document in other words) consequently in linear order under the so-called **Result Label**.

The "Result Label" is *TDF_label* used to keep the algorithm result *Shape* from *TopoDS* in *NamedShape* attribute. During loading sub-shapes of the result in Data Framework should be used the rules of chapter **Registering shapes and their evolution**. These rules are also applicable for loading the main shape, i.e. the resulting shape produced by the modeling algorithm.

### Selection / re-computation mechanism

When the Data Framework is filled with all impacted entities (including the data structures resulting from the current modeling operation and the data structures resulting from the previous modeling operations, on which the current operation depends) any sub-shape of the current result can be **selected**, i.e. the corresponding new naming data structures, which support this functionality, can be produced and kept in the Data Framework.

One of the user interfaces for topological naming is the class *TNaming_Selector*. It implements the above mentioned sub-shape "selection" functionality as an additional one. I.e. it can be used for:

- Storing the selected shape on a label – its **Selection**;
- Accessing the named shape – check the kept value of the shape
- Update of this naming – recomputation of an earlier selected shape.

The selector places a new named shape with evolution **SELECTED** to the given label. The selector creates a **name** of the selected shape, which is a unique description (data structure) of how to find the selected topology using as resources:

- the given context shape, i.e. the main shape kept on **Result Label**, which contains a selected sub-shape,
- its evolution and
- naming structure.

After any modification of a context shape and update of the corresponding naming structure, it is necessary to call method *TNaming_Selector::Solve*. If the naming structure, i.e. the above mentioned **name**, is correct, the selector automatically updates the selected sub-shape in the corresponding named shape, else it fails.

## Exploring shape evolution

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

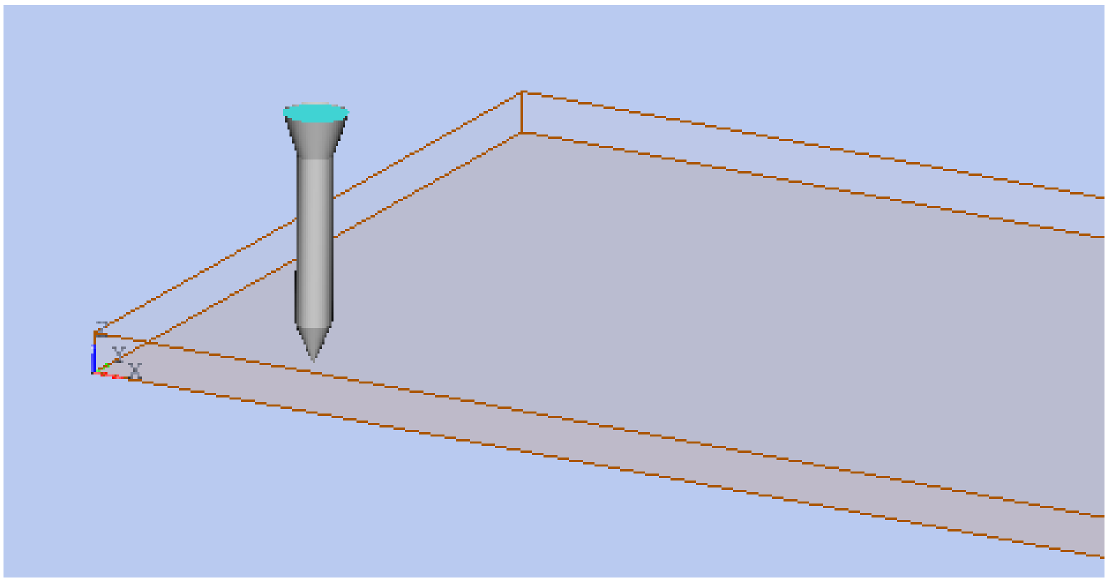
If you need to create a topological attribute for existing data, use the method *NamedShape*.

```
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);
}
```

## Example of topological naming usage

**Topological naming** is a mechanism of Open CASCADE aimed to keep reference to the selected shape. If, for example, we select a vertex of a solid shape and "ask" the topological naming to keep reference to this vertex, it will refer to the vertex whatever happens with the shape (translations, scaling, fusion with another shape, etc.).

Let us consider an example: imagine a wooden plate. The job is to drive several nails in it:



**A nail driven in a wooden plate**

There may be several nails with different size and position. A **Hammer** should push each **Nail** exactly in the center point of the top surface. For this the user does the following:

- Makes several Nails of different height and diameter (according to the need),
- Chooses (selects) the upper surface of each Nail for the Hammer.

The job is done. The application should do the rest – the Hammer calculates a center point for each selected surface of the Nail and "strikes" each Nail driving it into the wooden plate.

What happens if the user changes the position of some Nails? How will the Hammer know about it? It keeps reference to the surface of each Nail. However, if a Nail is relocated, the Hammer should know the new position of the selected surface. Otherwise, it will "strike" at the old position (keep the fingers away!)…

Topological naming mechanism should help the Hammer to obtain the relocated surfaces. The Hammer "asks" the mechanism to "resolve" the selected shapes by calling method *TNaming_Selection::Solve()* and the mechanism "returns" the modified surfaces located at the new position by calling *TNaming_Selector::NamedShape()*.

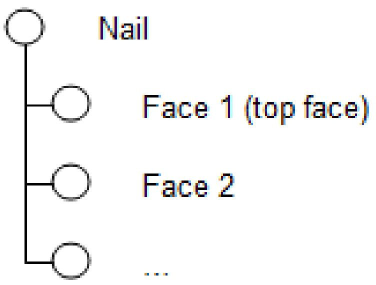
The topological naming is represented as a “black box” in the example above. Now it is time to make the box a little more “transparent”.

The application contains 3 functions:

- **Nail** – produces a shape representing a nail,
- **Translator** – translates a shape along the wooden plate,
- **Hammer** – drives the nail in the wooden plate.

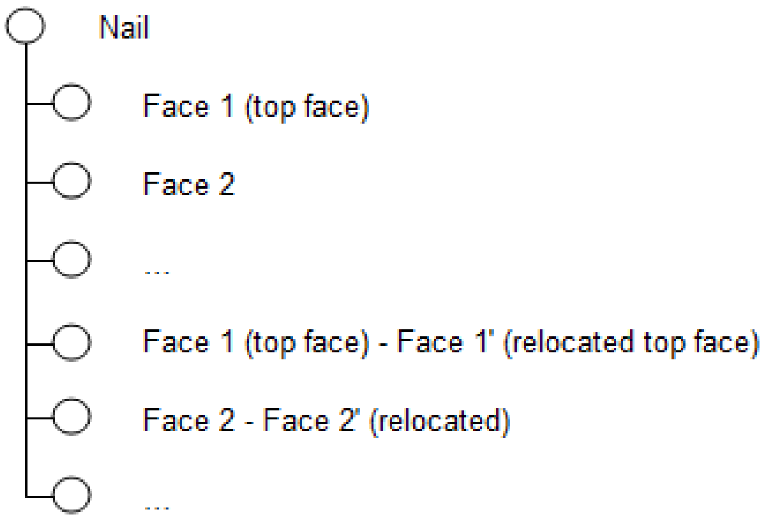
Each function gives the topological naming some hints how to “re-solve” the selected sub-shapes:

- The Nail constructs a solid shape and puts each face of the shape into sub-labels:



**Distribution of faces through sub-labels of the Nail**

- The **Translator** moves a shape and registers modification for each face: it puts a pair: “old” shape – “new” shape at a sub-label of each moving Nail. The “old” shape represents a face of the Nail at the initial position. The “new” shape – is the same face, but at a new position:


Nail
Face 1 (top face)
Face 2
...
Face 1 (top face) - Face 1' (relocated top face)
Face 2 - Face 2' (relocated)
...

**Registration of relocation of faces of a Nail**

How does it work?

- The Hammer selects a face of a Nail calling *TNaming_Selector::Select()*. This call makes a unique name for the selected shape. In our example, it will be a direct

reference to the label of the top face of the Nail (Face 1).
- When the user moves a Nail along the wooden plate, the Translator registers this modification by putting the pairs: “old” face of the Nail – new face of the Nail into its sub-labels.
- When the Hammer calls *TNaming::Solve()*, the topological naming “looks” at the unique name of the selected shape and tries to re-solve it:
    - It finds the 1st appearance of the selected shape in the data tree – it is a label under the Nail function *Face 1*.
    - It follows the evolution of this face. In our case, there is only one evolution – the translation: *Face 1* (top face) – *Face 1’* (relocated top face). So, the last evolution is the relocated top face.
- Calling the method *TNaming_Selector::NamedShape()* the Hammer obtains the last evolution of the selected face – the relocated top face.

The job is done.

P.S. Let us say a few words about a little more complicated case – selection of a wire of the top face. Its topological name is an “intersection” of two faces. We remember that the **Nail** puts only faces under its label. So, the selected wire will represent an “intersection” of the top face and the conic face keeping the “head” of the nail. Another example is a selected vertex. Its unique name may be represented as an “intersection” of three or even more faces (depends on the shape).

# Standard Attributes

## Overview

Standard attributes are ready-to-use attributes, which allow creating and modifying attributes for many basic data types. They are available in the packages *TDataStd, TDataXtd* and *TDF*. Each attribute belongs to one of four types:

- Geometric attributes;
- General attributes;
- Relationship attributes;
- Auxiliary attributes.

## 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;
- **PatternStd** – identifies one of five available pattern models (linear, circular, rectangular, circular rectangular and mirror);
- **Position** – identifies the position in 3d global space.

## General attributes

- **AsciiString** – contains AsciiString value;
- **BooleanArray** – contains an array of Boolean;
- **BooleanList** – contains a list of Boolean;
- **ByteArray** – contains an array of Byte (unsigned char) values;
- **Comment** – contains a string – the comment for a given label (or attribute);
- **Expression** – contains an expression string and a list of used variables attributes;
- **ExtStringArray** – contains an array of *ExtendedString* values;
- **ExtStringList** – contains a list of *ExtendedString* values;
- **Integer** – contains an integer value;
- **IntegerArray** – contains an array of integer values;
- **IntegerList** – contains a list of integer values;
- **IntPackedMap** – contains a packed map of integers;
- **Name** – contains a string – the name of a given label (or attribute);
- **NamedData** – may contain up to 6 of the following named data sets (vocabularies): *DataMapOfStringInteger, DataMapOfStringReal, DataMapOfStringString, DataMapOfStringByte, DataMapOfStringHArray1OfInteger* or *DataMapOfStringHArray1OfReal*;
- **NoteBook** – contains a *NoteBook* object attribute;
- **Real** – contains a real value;
- **RealArray** – contains an array of real values;
- **RealList** – contains a list of real values;
- **Relation** – contains a relation string and a list of used variables attributes;
- **Tick** – defines a boolean attribute;
- **Variable** – simply identifies, that a variable belongs to this label; contains the flag *is constraint* 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;
- **ReferenceArray** – contains an array of references;
- **ReferenceList** – contains a list of references;
- **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** – high-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 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.

## Attributes supporting several attributes of the same type on the same label

By default only one attribute of the same type on the same label is supported. For example, you can set only one TDataStd_Real attribute on the same label. This limitation was removed for some predefined sub-set of standard attributes by adding so called 'user defined ID' feature to the attribute. The listed below attributes received this new feature:

- **TDataStd_AsciiString**
- **TDataStd_Integer**
- **TDataStd_Name**
- **TDataStd_Real**
- **TDataStd_BooleanArray**
- **TDataStd_BooleanList**
- **TDataStd_ByteArray**
- **TDataStd_ExtStringArray**
- **TDataStd_ExtStringList**
- **TDataStd_IntegerArray**
- **TDataStd_IntegerList**
- **TDataStd_RealArray**
- **TDataStd_RealList**
- **TDataStd_ReferenceArray**

* **TDataStd_ReferenceList**

See for details paragraph 6.4.

## Services common to all attributes

### Accessing GUIDs

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

* Method *GetID* 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* 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.

To find an attribute attached to a specific label, you use the GUID of the attribute type you are looking for. This information can be found using the method *GetID* and the method *Find* for the label as follows:

```
Standard_GUID anID = MyAttributeClass::GetID();
Standard_Boolean HasAttribute = aLabel.Find(anID,anAttribute);
```

### Conventional Interface of Standard Attributes

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

* Method *Set(label, [value])* 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.

## The choice between standard and custom attributes

When you start to design an application based on OCAF, usually it is necessary to choose, which attribute will be used for allocation of data in the OCAF document: standard or newly-created?

It is possible to describe any model by means of standard OCAF attributes. However, it is still a question if this description will be efficient in terms of memory and speed, and, at the same time, convenient to use.

This depends on a particular model.

OCAF imposes the restriction that only one attribute type may be allocated to one label. It is necessary to take into account the design of the application data tree. For example, if a label should possess several double values, it is necessary to distribute them through several child sub-labels or use an array of double values.

Let us consider several boundary implementations of the same model in OCAF tree and analyze the advantages and disadvantages of each approach.

## Comparison and analysis of approaches

Below are described two different model implementations: one is based on standard OCAF attributes and the other is based on the creation of a new attribute possessing all data of the model.

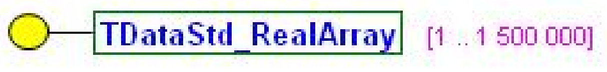
A load is distributed through the shape. The measurements are taken at particular points defined by (x, y and z) coordinates. The load is represented as a projection onto X, Y and Z axes of the local coordinate system at each point of measurement. A matrix of transformation is needed to convert the local coordinate system to the global one, but this is optional.

So, we have 15 double values at each point of measurement. If the number of such points is 100 000, for example, it means that we have to store 1 500 000 double values in the OCAF document.

The first approach consists in using standard OCAF attributes. Besides, there are several variants of how the standard attributes may be used:

- Allocation of all 1 500 000 double values as one array of double values attached to one label;
- Allocation of values of one measure of load (15 values) as one array of double values and attachment of one point of measure to one label;
- Allocation of each point of measure as an array of 3 double values attached to one label, the projection of load onto the local coordinate system axes as another array of 3 double values attached to a sub-label, and the matrix of projection (9 values) as the third array also attached to a sub-label.

Certainly, other variants are also possible.

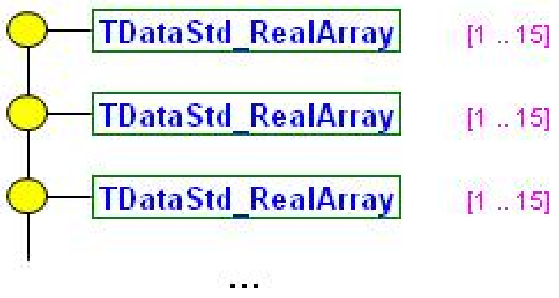


**Allocation of all data as one array of double values**

The first approach to allocation of all data represented as one array of double values saves initial memory and is easy to implement. But access to the data is difficult because the values are stored in a flat array. It will be necessary to implement a class with several methods giving access to particular fields like the measurement points, loads and so on.

If the values may be edited in the application, it means that the whole array will be backed-up on each edition. The memory usage will increase very fast! So, this approach may be considered only in case of non-editable data.

Let's consider the allocation of data of each measurement point per label (the second case). In this case we create 100 000 labels – one label for each measurement point and attach an array of double values to these labels:

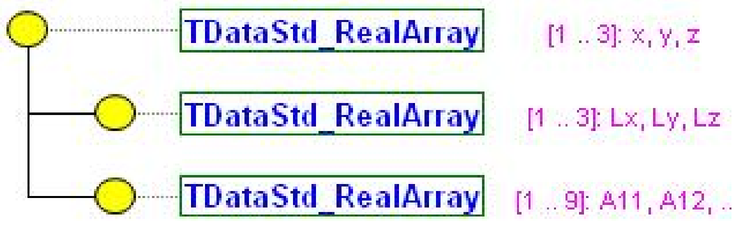


**Allocation of data of each measurement point as arrays of double values**

Now edition of data is safer as far as memory usage is concerned. Change of value for one measurement point (any value: point coordinates, load, and so on) backs-up only one small array of double values. But this structure (tree) requires more memory space (additional labels and attributes).

Besides, access to the values is still difficult and it is necessary to have a class with methods of access to the array fields.

The third case of allocation of data through OCAF tree is represented below:



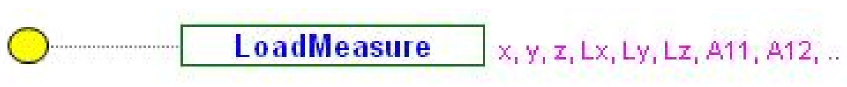
**Allocation of data into separate arrays of double values**

In this case sub-labels are involved and we can easily access the values of each measurement point, load or matrix. We don't need an interface class with methods of access to the data (if it exists, it would help to use the data structure, but this is optional).

On the one hand, this approach requires more memory for allocation of the attributes (arrays of double values). On the other hand, it saves memory during the edition of data by backing-up only the small array containing the modified data. So, if the data is fully modifiable, this approach is more preferable.

Before making a conclusion, let’s consider the same model implemented through a newly created OCAF attribute.

For example, we might allocate all data belonging to one measurement point as one OCAF attribute. In this case we implement the third variant of using the standard attributes (see picture 3), but we use less memory (because we use only one attribute instead of three):



**Allocation of data into newly created OCAF attribute**

The second variant of using standard OCAF attributes still has drawbacks: when data is edited, OCAF backs-up all values of the measurement point.

Let’s imagine that we have some non-editable data. It would be better for us to allocate this data separately from editable data. Back-up will not affect non-editable data and memory will not increase so much during data edition.

### Conclusion

When deciding which variant of data model implementation to choose, it is necessary to take into account the application response time, memory allocation and memory usage in transactions.

Most of the models may be implemented using only standard OCAF attributes. Some other models need special treatment and require implementation of new OCAF attributes.

## Standard Attributes with User Defined GUID

The listed above attributes allow to set at the same Label as many attributes of the same type as you want thanks to specific user's ID. Let's consider it on the example of the TDataStd_Real attribute. The previous version of the attribute allowed to set the attribute using static method Set in next way:

```
static Handle(TDataStd_Real) Set (const TDF_Label& label, const
       Standard_Real value);
```

This is a default form which is kept by the attribute. It uses the default GUID for the attribute identification - TDataStd_Real::GetID(). In case if you want to use the new feature (user defined Real attribute), for example to define several attributes which should keep a value of the same type - Standard_Real, but to be associated with different user's notions (or objects) the new static method Set should be used. In our example we will define two Real attributes which presents two customer's objects - Density and Volume and will be put on the same Label.

```
#define DENSITY Standard_GUID("12e9454b-6dbc-11d4-b9c8-0060b0ee2810")
#define VOLUME  Standard_GUID("161595c0-3628-4737-915a-c160ce94c6f7")

TDF_Label aLabel = ...;

// Real attribute type with user defined GUID associated with user's
      object "Density"
TDataStd_Real::Set(aLabel, DENSITY, 1.2);

// Real attribute type with user defined GUID associated with user's
      object "Volume"
TDataStd_Real::Set(aLabel, VOLUME, 185.5);

 To find an user defined Real attribute just use a corresponding GUID:
Handle (TDataStd_Real) anAtt;
aLabel.FindAttribute (DENSITY, anAtt);
```

## Creation Attributes with User Defined GUID.

You can create a new instance of an attribute with user define GUID and add it to label in two ways.

1. Using static method Set(). For example:

```
TDF_Label aLabel = ...;
Standard_Integer aValue = ...;
Standard_GUID aGuid = TDataStd_Integer::GetID();
TDataStd_Integer::Set(aLabel, aGuid, aValue);
```

1. Using the default constructor

```
Handle(TDataStd_Integer) anInt = new TDataStd_Integer();
anInt->SetID(aGuid);
aLabel.Add(anInt);
anInt->Set(aValue);
```

# Visualization Attributes

## Overview

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

## Services provided

### 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*.

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

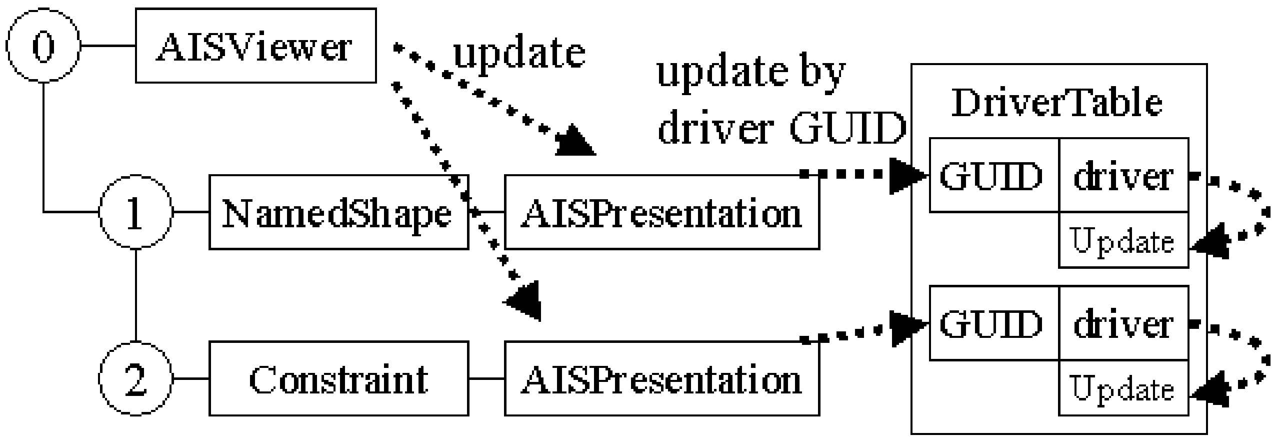
## Defining a presentation attribute

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.

### 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.

### 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.

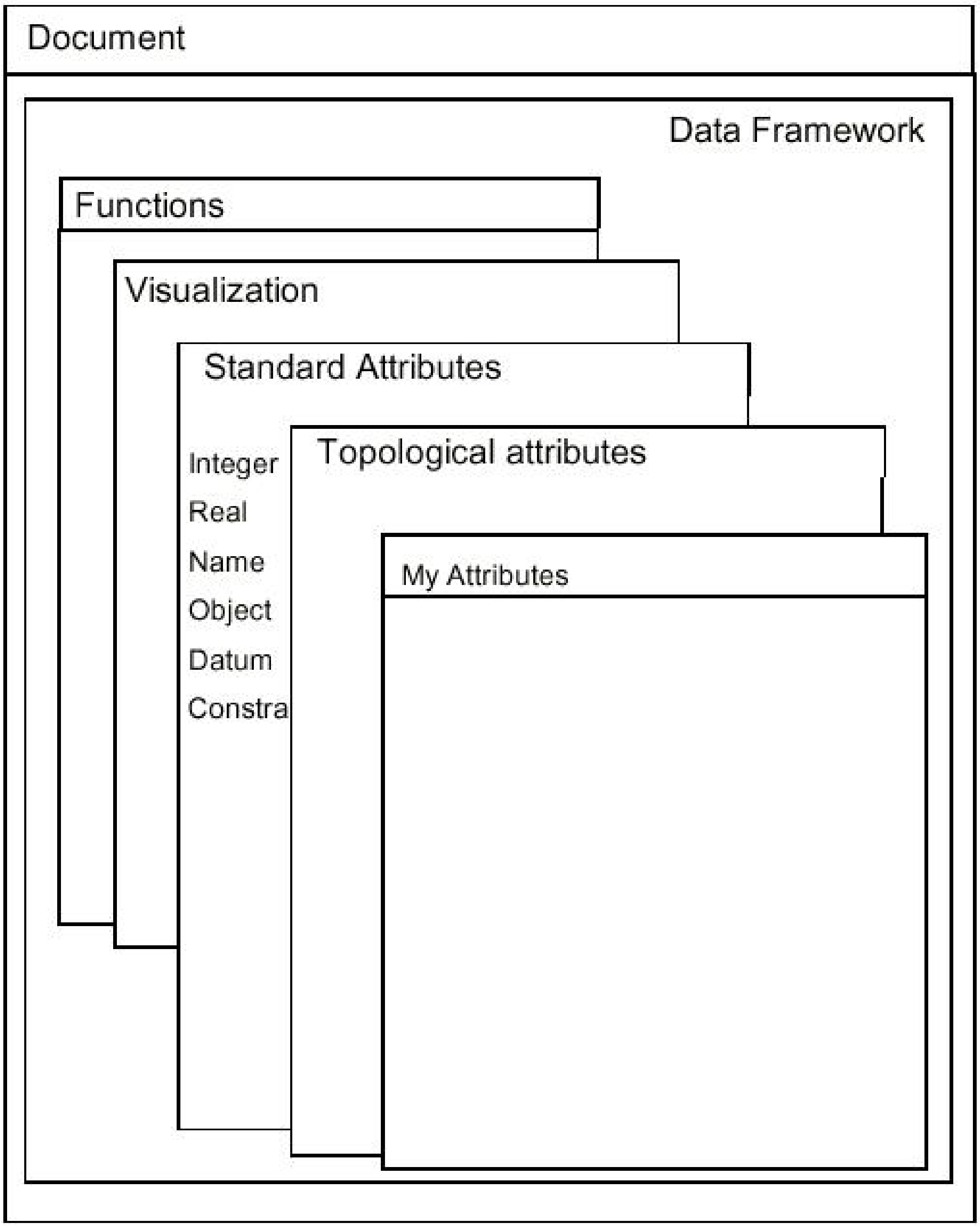
**Example**

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

# Function Services

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.



**Document structure**

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.

## 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.

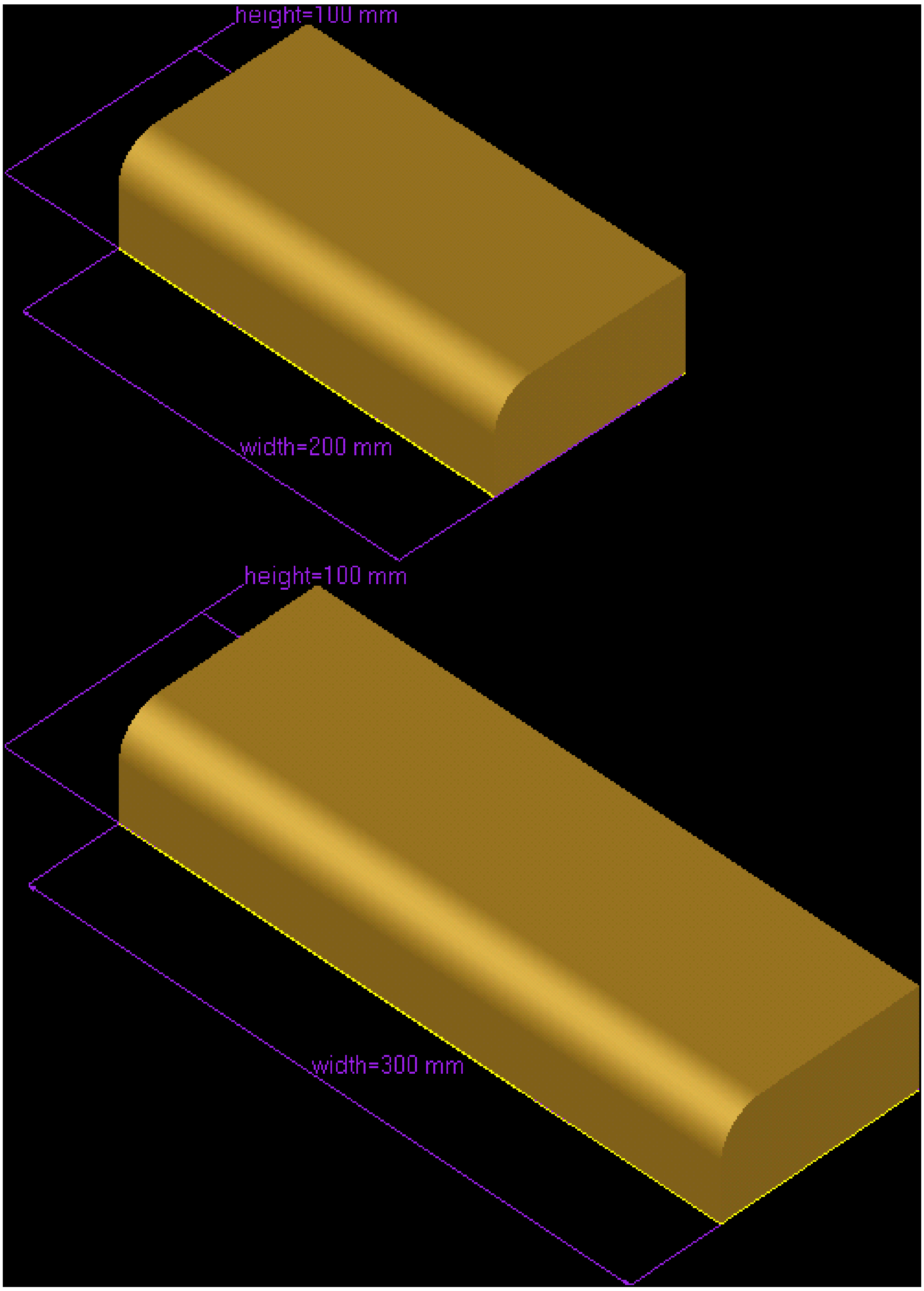
Then the solver algorithm of a data model can find the *Function* attribute on a corresponding label and call the *Execute* driver method to update the result of the function.

## 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.

## Propagating modifications

An application must implement its functions, function drivers and the common solver for parametric model creation. For example, check the following model:



The procedure of its creation is as follows:

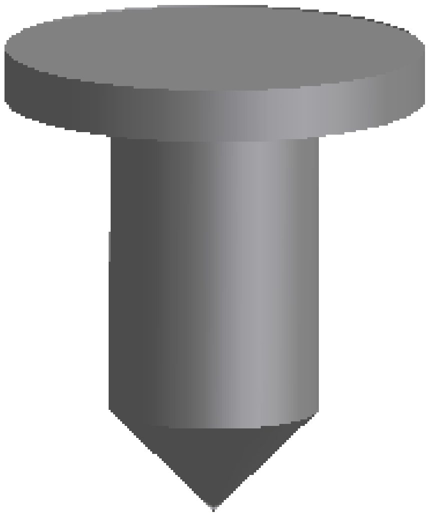
- create a rectangular planar face *F* with height 100 and width 200;
- create prism *P* using face *F* as a basis;
- create fillet *L* at the edge of the prism;
- change the width of *F* from 200 to 300;
- the solver for the function of face *F* starts;
- the solver detects that an argument of the face *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;
- the 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.

# Example of Function Mechanism Usage

## Introduction

Let us describe the usage of the Function Mechanism of Open CASCADE Application Framework on a simple example.
This example represents a "nail" composed by a cone and two cylinders of different radius and height:



**A nail**

These three objects (a cone and two cylinders) are independent, but the Function Mechanism makes them connected to each other and representing one object – a nail.
The object "nail" has the following parameters:

- The position of the nail is defined by the apex point of the cone. The cylinders are built on the cone and therefore they depend on the position of the cone. In this way we define a dependency of the cylinders on the cone.
- The height of the nail is defined by the height of the cone.
  Let's consider that the long cylinder has 3 heights of the cone and the header cylinder has a half of the height of the cone.
- The radius of the nail is defined by the radius of the cone. The radius of the long cylinder coincides with this value. Let's consider that the header cylinder has one and

a half radiuses of the cone.

So, the cylinders depend on the cone and the cone parameters define the size of the nail.

It means that re-positioning the cone (changing its apex point) moves the nail, the change of the radius of the cone produces a thinner or thicker nail, and the change of the height of the cone shortens or prolongates the nail.
It is suggested to examine the programming steps needed to create a 3D parametric model of the "nail".

## Step 1: Data Tree

The first step consists in model data allocation in the OCAF tree. In other words, it is necessary to decide where to put the data.

In this case, the data can be organized into a simple tree using references for definition of dependent parameters:

- Nail
  - Cone
    - Position (x,y,z)
    - Radius
    - Height
  - Cylinder (stem)
    - Position = "Cone" position translated for "Cone" height along Z;
    - Radius = "Cone" radius;
    - Height = "Cone" height multiplied by 3;
  - Cylinder (head)
    - Position = "Long cylinder" position translated for "Long cylinder" height along Z;
    - Radius = "Long cylinder" radius multiplied by 1.5;
    - Height = "Cone" height divided by 2.

  The "nail" object has three sub-leaves in the tree: the cone and two cylinders.

  The cone object is independent.

  The long cylinder representing a "stem" of the nail refers to the corresponding parameters of the cone to define its own data (position, radius and height). It means that the long cylinder depends on the cone.

  The parameters of the head cylinder may be expressed through the cone parameters only or through the cone and the long cylinder parameters. It is suggested to express the position and the radius of the head cylinder through the position and the radius of the long cylinder, and the height of the head cylinder through the height of the cone. It means that the head cylinder depends on the cone and the long cylinder.

## Step 2: Interfaces

The interfaces of the data model are responsible for dynamic creation of the data tree of the represented at the previous step, data modification and deletion.

The interface called *INail* should contain the methods for creation of the data tree for the nail, setting and getting of its parameters, computation, visualization and removal.

### Creation of the nail

This method of the interface creates a data tree for the nail at a given leaf of OCAF data tree.

It creates three sub-leaves for the cone and two cylinders and allocates the necessary data (references at the sub-leaves of the long and the head cylinders).

It sets the default values of position, radius and height of the nail.

The nail has the following user parameters:

- The position – coincides with the position of the cone
- The radius of the stem part of the nail – coincides with the radius of the cone
- The height of the nail – a sum of heights of the cone and both cylinders

The values of the position and the radius of the nail are defined for the cone object data. The height of the cone is recomputed as 2 * heights of nail and divided by 9.

### Computation

The Function Mechanism is responsible for re-computation of the nail. It will be described in detail later in this document.

A data leaf consists of the reference  to the location of the real data and a real value defining a coefficient of multiplication of the referenced data.

For example, the height of the long cylinder is defined as a reference to the height of the cone with coefficient 3. The data leaf of the height of the long cylinder should contain two attributes: a reference to the height of cone and a real value equal to 3.

### Visualization

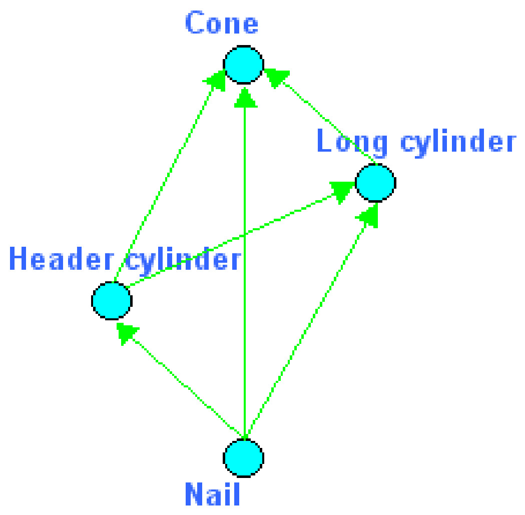
The shape resulting of the nail function can be displayed using the standard OCAF visualization mechanism.

### Removal of the nail

To automatically erase the nail from the viewer and the data tree it is enough to clean the nail leaf from attributes.

## Step 3: Functions

The nail is defined by four functions: the cone, the two cylinders and the nail function.
The function of the cone is independent. The functions of the cylinders depend on the cone function. The nail function depends on the results of all functions:



**A graph of dependencies between functions**

Computation of the model starts with the cone function, then the long cylinder, after that the header cylinder and, finally, the result is generated by the nail function at the end of function chain.

The Function Mechanism of Open CASCADE Technology creates this graph of dependencies and allows iterating it following the dependencies. The only thing the Function Mechanism requires from its user is the implementation of pure virtual methods of *TFunction_Driver*:

- *::Arguments()* – returns a list of arguments for the function
- *::Results()* – returns a list of results of the function

These methods give the Function Mechanism the information on the location of arguments and results of the function and allow building a graph of functions. The class *TFunction_Iterator* iterates the functions of the graph in the execution order.

The pure virtual method *TFunction_Driver::Execute()* calculating the function should be overridden.

The method *::MustExecute()* calls the method *::Arguments()* of the function driver and ideally this information (knowledge of modification of arguments of the function) is enough to make a decision whether the function should be executed or not. Therefore, this method usually shouldn't be overridden.

The cone and cylinder functions differ only in geometrical construction algorithms. Other parameters are the same (position, radius and height).

It means that it is possible to create a base class – function driver for the three functions, and two descendant classes producing: a cone or a cylinder.

For the base function driver the methods *::Arguments()* and *::Results()* will be overridden. Two descendant function drivers responsible for creation of a cone and a cylinder will override only the method *::Execute()*.

The method *::Arguments()* of the function driver of the nail returns the results of the functions located under it in the tree of leaves. The method *::Execute()* just collects the results of the functions and makes one shape – a nail.

This way the data model using the Function Mechanism is ready for usage. Do not forget to introduce the function drivers for a function driver table with the help of *TFunction_DriverTable* class.

## Example 1: iteration and execution of functions.

This is an example of the code for iteration and execution of functions.

```
// The scope of functions is defined.
Handle(TFunction_Scope) aScope = TFunction_Scope::Set (anyLabel);

// The information on modifications in the model is received.
TFunction_Logbook& aLog = aScope->GetLogbook();

// The iterator is iInitialized by  the scope of functions.
TFunction_Iterator anIterator (anyLabel);
anIterator.SetUsageOfExecutionOrder (true);

// The function is iterated,  its dependency is checked on the modified
      data and  executed if necessary.
for (; anIterator.more(); anIterator.Next())
{
  // The function iterator may return a list of  current functions for
      execution.
  // It might be useful for multi-threaded execution  of functions.
  const TDF_LabelList& aCurrentFunctions = anIterator.Current();

  // The list of current functions is iterated.
  for (TDF_ListIteratorOfLabelList aCurrentIterator (aCurrentFunctions);
       aCurrentIterator.More(); aCurrentIterator.Next())
  {
    //  An interface for the function is created.
    TFunction_IFunction anInterface (aCurrentIterator.Value());

    //  The function driver is retrieved.
    Handle(TFunction_Driver) aDriver = anInterface.GetDriver();

    //  The dependency of the function on the  modified data is checked.
    if (aDriver->MustExecute (aLog))
    {
      // The function is executed.
      int aRes = aDriver->Execute (aLog);
      if (aRes)
      {
        return false;
      }
    }
  }
}
```

## Example 2: Cylinder function driver

This is an example of the code for a cylinder function driver. To make the things clearer, the methods *::Arguments()* and *::Results()* from the base class are also mentioned.

```
// A virtual method  ::Arguments() returns a list of arguments of the
       function.
CylinderDriver::Arguments( TDF_LabelList&amp; args )
{
  // The direct arguments, located at sub-leaves of the function, are
       collected (see picture 2)
  TDF_ChildIterator  cIterator( Label(), false );
  for (;  cIterator.More(); cIterator.Next() )
  {
    // Direct argument.
    TDF_Label  sublabel = cIterator.Value();
    Args.Append(  sublabel );

    // The references to the external data are  checked.
    Handle(TDF_Reference)  ref;
    If (  sublabel.FindAttribute( TDF_Reference::GetID(), ref ) )
    {
      args.Append(  ref-Get() );
    }
}

// A virtual method ::Results()  returns a list of result leaves.
CylinderDriver::Results( TDF_LabelList&amp; res )
{
  // The result is kept at the function  label.
  Res.Append(  Label() );
}

// Execution of the function  driver.
Int CylinderDriver::Execute( TFunction_Logbook&amp; log )
{
  // Position of the cylinder - position of the first  function (cone)
  //is  elevated along Z for height values of all  previous functions.
  gp_Ax2 axes = …. // out of the scope of this guide.
  // The radius value is retrieved.
  // It is located at second child sub-leaf (see the  picture 2).
  TDF_Label radiusLabel  = Label().FindChild( 2 );

  // The multiplicator of the radius ()is retrieved.
  Handle(TDataStd_Real)  radiusValue;
  radiusLabel.FindAttribute(  TDataStd_Real::GetID(), radiusValue);

  // The reference to the radius is retrieved.
  Handle(TDF_Reference)  refRadius;
  RadiusLabel.FindAttribute(  TDF_Reference::GetID(), refRadius );

  // The radius value is calculated.
  double radius = 0.0;

  if (  refRadius.IsNull() )
  {
    radius  = radiusValue-Get();
  }
  else
  {
    // The referenced radius value is  retrieved.
    Handle(TDataStd_Real)  referencedRadiusValue;
    RefRadius-Get().FindAttribute(TDataStd_Real::GetID()
       ,referencedRadiusValue );
    radius  = referencedRadiusValue-Get() * radiusValue-Get();
  }

  // The height value is retrieved.
  double height = … // similar code to taking the radius value.

  // The cylinder is created.
  TopoDS_Shape cylinder  = BRepPrimAPI_MakeCylinder(axes, radius,
       height);

  // The result (cylinder) is set
  TNaming_Builder  builder( Label() );
  Builder.Generated(  cylinder );

  // The modification of the result leaf is saved in  the log.
  log.SetImpacted(  Label() );

  return 0;
}
```

# XML Support

Writing and reading XML files in OCCT is provided by LDOM package, which constitutes an integral part of XML OCAF persistence, which is the optional component provided on top of Open CASCADE Technology.

The Light DOM (LDOM) package contains classes maintaining a data structure whose main principles conform to W3C DOM Level 1 Recommendations. The purpose of these classes as required by XML OCAF persistence schema is to:

- Maintain a tree structure of objects in memory representing the XML document. The root of the structure is an object of the *LDOM_Document* type. This object contains all the data corresponding to a given XML document and contains one object of the *LDOM_Element* type named "document element". The document element contains other *LDOM_Element* objects forming a tree. Other types of nodes: *LDOM_Attr, LDOM_Text, LDOM_Comment* and *LDOM_CDATASection* – represent the corresponding XML types and serve as branches of the tree of elements.
- Provide class *LDOM_Parser* to read XML files and convert them to *LDOM_Document* objects.
- Provide class *LDOM_XmlWriter* to convert *LDOM_Document* to a character stream in XML format and store it in file.

This package covers the functionality provided by numerous products known as "DOM parsers". Unlike most of them, LDOM was specifically developed to meet the following requirements:

- To minimize the virtual memory allocated by DOM data structures. In average, the amount of memory of LDOM is the same as the XML file size (UTF-8).
- To minimize the time required for parsing and formatting XML, as well as for access to DOM data structures.

Both these requirements are important when XML files are processed by applications if these files are relatively large (occupying megabytes and even hundreds of megabytes). To meet the requirements, some limitations were imposed on the DOM Level 1 specification; these limitations are insignificant in applications like OCAF. Some of these limitations can be overridden in the course of future developments. The main limitations are:

- No Unicode support as well as various other encodings; only ASCII strings are used in DOM/XML. Note: There is a data type *TCollection_ExtendedString* for wide character data. This type is supported by *LDOM_String* as a sequence of numbers.
- Some superfluous methods are deleted: *getPreviousSibling, getParentNode,* etc.
- No resolution of XML Entities of any kind
- No support for DTD: the parser just checks for observance of general XML rules and never validates documents.
- Only 5 available types of DOM nodes: *LDOM_Element, LDOM_Attr, LDOM_Text, LDOM_Comment* and *LDOM_CDATASection*.
- No support of Namespaces; prefixed names are used instead of qualified names.
- No support of the interface *DOMException* (no exception when attempting to remove a non-existing node).

LDOM is dependent on Kernel OCCT classes only. Therefore, it can be used outside OCAF persistence in various algorithms where DOM/XML support may be required.

## Document Drivers

The drivers for document storage and retrieval manage conversion between a transient OCAF Document in memory and its persistent reflection in a container (disk, memory, network). For XML Persistence, they are defined in the package XmlDrivers.

The main methods (entry points) of these drivers are:

- *Write()* – for a storage driver;
- *Read()* – for a retrieval driver.

The most common case (which is implemented in XML Persistence) is writing/reading document to/from a regular OS file. Such conversion is performed in two steps:

First it is necessary to convert the transient document into another form (called persistent), suitable for writing into a file, and vice versa. In XML Persistence LDOM_Document is used as the persistent form of an OCAF Document and the DOM_Nodes are the persistent objects. An OCAF Document is a tree of labels with attributes. Its transformation into a persistent form can be functionally divided into two parts:

- Conversion of the labels structure, which is performed by the method XmlMDF::FromTo()
- Conversion of the attributes and their underlying objects, which is performed by the corresponding attribute drivers (one driver per attribute type).

The driver for each attribute is selected from a table of drivers, either by attribute type (on storage) or by the name of the corresponding DOM_Element (on retrieval). The table of drivers is created by by methods *XmlDrivers_DocumentStorageDriver::AttributeDrivers()* and *XmlDrivers_DocumentRetrievalDriver::AttributeDrivers()*.

Then the persistent document is written into a file (or read from a file). In standard persistence Storage and FSD packages contain classes for writing/reading the persistent document into a file. In XML persistence *LDOMParser* and *LDOM_XmlWriter* are used instead.

Usually, the library containing document storage and retrieval drivers is loaded at run time by a plugin mechanism. To support this in XML Persistence, there is a plugin *XmlPlugin* and a *Factory()* method in the *XmlDrivers* package. This method compares passed GUIDs with known GUIDs and returns the corresponding driver or generates an exception if the GUID is unknown.

The application defines which GUID is needed for document storage or retrieval and in which library it should be found. This depends on document format and application resources. Resources for XML Persistence and also for standard persistence are found in the StdResource unit. They are written for the XmlOcaf document format.

## Attribute Drivers

There is one attribute driver for XML persistence for each transient attribute from a set of standard OCAF attributes, with the exception of attribute types, which are never stored (pure transient). Standard OCAF attributes are collected in six packages, and their drivers also follow this distribution. Driver for attribute *T*_** is called *XmlM*_**. Conversion between transient and persistent form of attribute is performed by two methods *Paste()* of attribute driver.

*XmlMDF_ADriver* is the root class for all attribute drivers.

At the beginning of storage/retrieval process, one instance of each attribute driver is created and appended to driver table implemented as *XmlMDF_ADriverTable*. During OCAF Data storage, attribute drivers are retrieved from the driver table by the type of attribute. In the retrieval step, a data map is created linking names of *DOM_Elements* and attribute drivers, and then attribute drivers are sought in this map by *DOM_Element* qualified tag names.

Every transient attribute is saved as a *DOM_Element* (root element of OCAF attribute) with attributes and possibly sub-nodes. The name of the root element can be defined in the attribute driver as a string passed to the base class constructor. The default is the attribute type name. Similarly, namespace prefixes for each attribute can be set. There is no default value, but it is possible to pass NULL or an empty string to store attributes without namespace prefixes.

The basic class *XmlMDF_ADriver* supports errors reporting via the method *WriteMessage(const TCollection_ExtendedString&)*. It sends a message string to its message driver which is initialized in the constructor with a *Handle(CDM_MessageDriver)* passed from the application by Document Storage/Retrieval Driver.

## XML Document Structure

Every XML Document has one root element, which may have attributes and contain other nodes. In OCAF XML Documents the root element is named "document" and has attribute "format" with the name of the OCAF Schema used to generate the file. The standard XML format is "XmlOcaf". The following elements are sub-elements of <document> and should be unique entries as its sub-elements, in a specific order. The order is:

- **Element info** – contains strings identifying the format version and other parameters of the OCAF XML document. Normally, data under the element is used by persistence algorithms to correctly retrieve and initialize an OCAF document. The data also includes a copyright string.
- **Element comments** – consists of an unlimited number of <comment> sub-elements containing necessary comment strings.
- **Element label** – the root label of the document data structure, with the XML attribute "tag" equal to 0. It contains all the OCAF data (labels, attributes) as tree of XML elements. Every sub-label is identified by a tag (positive integer) defining a unique key for all sub-labels of a label. Every label can contain any number of elements representing OCAF attributes (see OCAF Attributes Representation below).
- **Element shapes** – contains geometrical and topological entities in BRep format. These entities being referenced by OCAF attributes written under the element <label>. This element is empty if there are no shapes in the document. It is only output if attribute driver *XmlMNaming_NamedShapeDriver* has been added to drivers table by the *DocumentStorageDriver*.

## OCAF Attributes Representation

In XML documents, OCAF attributes are elements whose name identifies the OCAF attribute type. These elements may have a simple (string or number) or complex (sub-elements) structure, depending on the architecture of OCAF attribute. Every XML type for OCAF attribute possesses a unique positive integer "id" XML attribute identifying the OCAF attribute throughout the document. To ensure "id" uniqueness, the attribute name "id" is reserved and is only used to indicate and identify elements which may be referenced from other parts of the OCAF XML document. For every standard OCAF attribute, its XML name matches the name of a C++ class in Transient data model. Generally, the XML name of OCAF attribute can be specified in the corresponding attribute driver. XML types for OCAF attributes are declared with XML W3C Schema in a few XSD files where OCAF attributes are grouped by the package where they are defined.

## Example of resulting XML file

The following example is a sample text from an XML file obtained by storing an OCAF document with two labels (0: and 0:2) and two attributes – *TDataStd_Name* (on label 0:) and *TNaming_NamedShape* (on label 0:2). The <shapes> section contents are replaced by an ellipsis.

```
<?xml version="1.0" encoding="UTF-8"?>
<document format="XmlOcaf" xmlns="http://www.opencascade.org/OCAF/XML"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opencascade.org/OCAF/XML
       http://www.opencascade.org/OCAF/XML/XmlOcaf.xsd">

<info date="2001-10-04" schemav="0" objnb="3">
<iitem>Copyright: Open Cascade, 2001</iitem>
<iitem>STORAGE_VERSION: PCDM_ReadWriter_1</iitem>
<iitem>REFERENCE_COUNTER: 0</iitem>
<iitem>MODIFICATION_COUNTER: 1</iitem>
</info>
<comments/>
<label tag="0">
<TDataStd_Name id="1">Document_1</TDataStd_Name>
<label tag="2">
<TNaming_NamedShape id="2" evolution="primitive">
<olds/>
<news>
<shape tshape="+34" index="1"/>
</news>
</TNaming_NamedShape>
</label>
</label>
\<shapes\>
...
</shapes>
</document>
```

## XML Schema

The XML Schema defines the class of a document.

The full structure of OCAF XML documents is described as a set of XML W3C Schema files with definitions of all XML element types. The definitions provided cannot be overridden. If any application defines new persistence schemas, it can use all the definitions from the present XSD files but if it creates new or redefines existing types, the definition must be done under other namespace(s).

There are other ways to declare XML data, different from W3C Schema, and it should be possible to use them to the extent of their capabilities of expressing the particular structure and constraints of our XML data model. However, it must be noted that the W3C Schema is the primary format for declarations and as such, it is the format supported for future improvements of Open CASCADE Technology, including the development of specific applications using OCAF XML persistence.

The Schema files (XSD) are intended for two purposes:

- documenting the data format of files generated by OCAF;
- validation of documents when they are used by external (non-OCAF) applications, e.g., to generate reports.

The Schema definitions are not used by OCAF XML Persistence algorithms when saving and restoring XML documents. There are internal checks to ensure validity when processing every type of data.

## Management of Namespaces

Both the XML format and the XML OCAF persistence code are extensible in the sense that every new development can reuse everything that has been created in previous projects. For the XML format, this extensibility is supported by assigning names of XML objects (elements) to different XML Namespaces. Hence, XML elements defined in different projects (in different persistence libraries) can easily be combined into the same XML documents. An example is the XCAF XML persistence built as an extension to the Standard OCAF XML persistence *[File XmlXcaf.xsd]*. For the correct management of Namespaces it is necessary to:

- Define *targetNamespace* in the new XSD file describing the format.
- Declare (in *XSD* files) all elements and types in the targetNamespace to appear without a namespace prefix; all other elements and types use the appropriate prefix (such as "ocaf:").
- Add (in the new *DocumentStorageDriver*) the *targetNamespace* accompanied with its prefix, using method *XmlDrivers_DocumentStorageDriver::AddNamespace*. The same is done for all namespaces objects which are used by the new persistence, with the exception of the "ocaf" namespace.
- Pass (in every OCAF attribute driver) the namespace prefix of the *targetNamespace* to the constructor of *XmlMDF_ADriver*.

# TObj Package

## Introduction

This document describes the package TObj, which is an add-on to the Open CASCADE Application Framework (OCAF).

This package provides a set of classes and auxiliary tools facilitating the creation of object-oriented data models on top of low-level OCAF data structures. This includes:

- Definition of classes representing data objects. Data objects store their data using primitive OCAF attributes, taking advantage of OCAF mechanisms for Undo/Redo and persistence. At the same time they provide a higher level abstraction over the pure OCAF document structure (labels / attributes).
- Organization of the data model as a hierarchical (tree-like) structure of objects.
- Support of cross-references between objects within one model or among different models. In case of cross-model references the models should depend hierarchically.
- Persistence mechanism for storing *TObj* objects in OCAF files, which allows storing and retrieving objects of derived types without writing additional code to support persistence.

This document describes basic principles of logical and physical organization of TObj-based data models and typical approaches to implementation of classes representing model objects.

### Applicability

The main purpose of the *TObj* data model is rapid development of the object-oriented data models for applications, using the existing functionality provided by OCAF (Undo/Redo and persistence) without the necessity to redevelop such functionality from scratch.

As opposed to using bare OCAF (at the level of labels and attributes), TObj facilitates dealing with higher level abstracts, which are closer to the application domain. It works best when the application data are naturally organized in hierarchical structures, and is especially useful for complex data models with dependencies between objects belonging to different parts of the model.

It should be noted that *TObj* is efficient for representing data structures containing a limited number of objects at each level of the data structure (typically less than 1000). A greater number of objects causes performance problems due to list-based organization of OCAF documents. Therefore, other methods of storage, such as arrays, are advisable for data models or their sub-parts containing a great number of uniform objects. However, these methods can be combined with the usage of *TObj* to represent the high-level structure of the model.

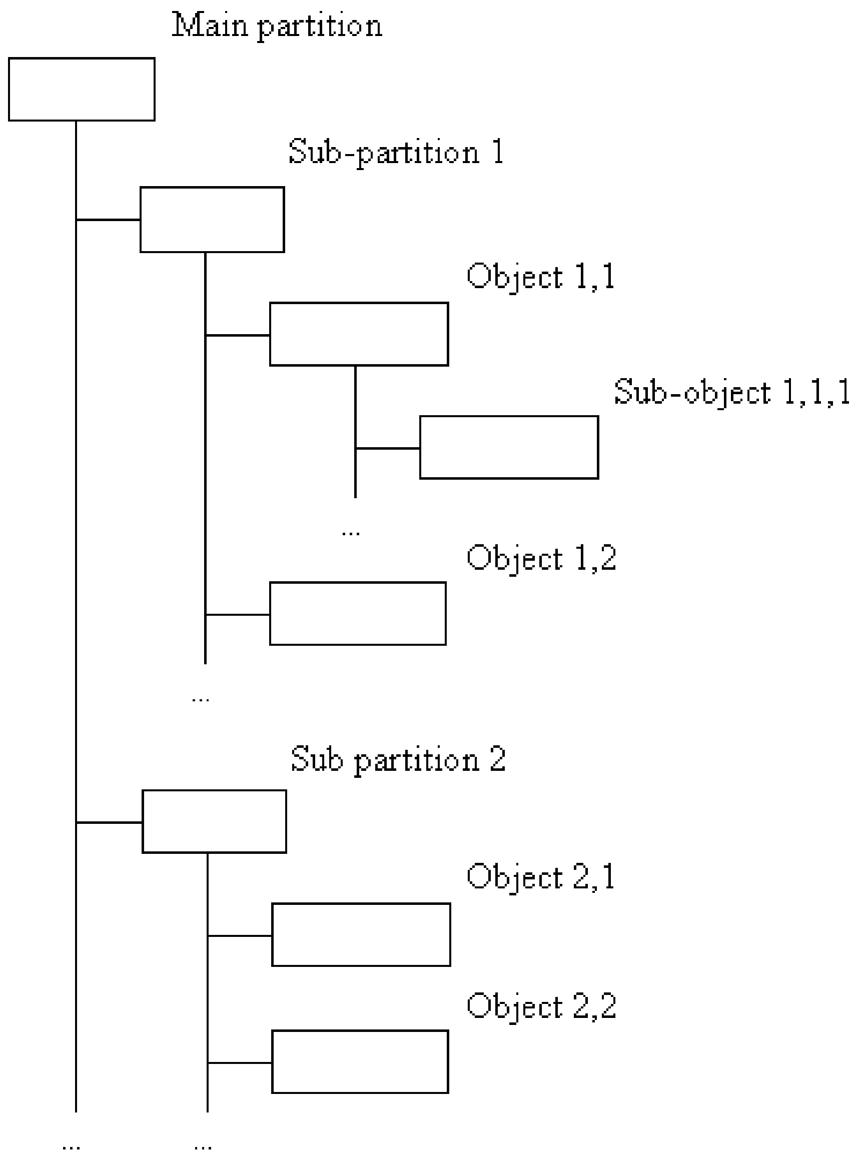
# TObj Model

## TObj Model structure

In the *TObj* data model the data are separated from the interfaces that manage them.

It should be emphasized that *TObj* package defines only the interfaces and the basic structure of the model and objects, while the actual contents and structure of the model of a particular application are defined by its specific classes inherited from *TObj* classes. The implementation can add its own features or even change the default behaviour and the data layout, though this is not recommended.

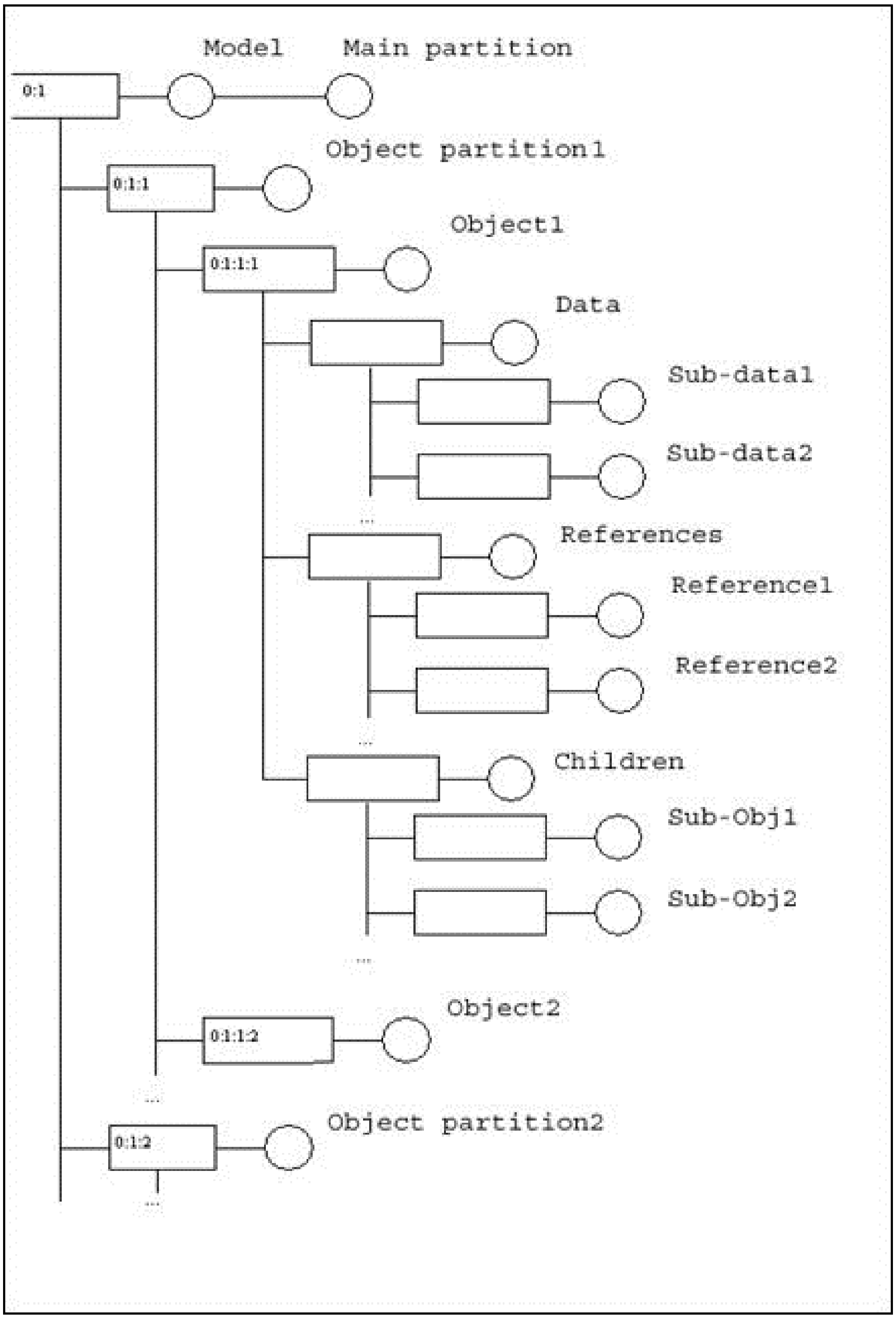
Logically the *TObj* data model is represented as a tree of model objects, with upper-level objects typically being collections of other objects (called *partitions*, represented by the class *TObj_Partition*). The root object of the model is called the *Main partition* and is maintained by the model itself. This partition contains a list of sub-objects called its *children* each sub-object may contain its own children (according to its type), etc.



**TObj Data Model**

As the *TObj* Data Model is based on OCAF (Open CASCADE Application Framework) technology, it stores its data in the underlying OCAF document. The OCAF document consists of a tree of items called *labels*. Each label has some data attached to it in the form of *attributes*, and may contain an arbitrary number of sub-labels. Each sub-label is identified by its sequential number called the *tag*. The complete sequence of tag numbers of the label and its parents starting from the document root constitutes the complete *entry* of the label, which uniquely identifies its position in the document.

Generally the structure of the OCAF tree of the *TObj* data model corresponds to the logical structure of the model and can be presented as in the following picture:



**TObj Data Model mapped on OCAF document**

All data of the model are stored in the root label (0:1) of the OCAF document. An attribute *TObj_TModel* is located in this root label. It stores the object of type *TObj_Model*. This object serves as a main interface tool to access all data and functionalities of the data model.

In simple cases all data needed by the application may be contained in a single data model. Moreover, *TObj* gives the possibility to distribute the data between several interconnected data models. This can be especially useful for the applications dealing with great amounts of data. because only the data required for the current operation is loaded in the memory at one time. It is presumed that the models have a hierarchical (tree-like) structure, where the objects of the child models can refer to the objects of the parent models, not vice-versa. Provided that the correct order of loading and closing of the models is ensured, the *TObj* classes will maintain references between the objects automatically.

## Data Model basic features

The class *TObj_Model* describing the data model provides the following functionalities:

- Loading and saving of the model from or in a file (methods *Load* and *Save*)
- Closing and removal of the model from memory (method *Close*)
- Definition of the full file name of the persistence storage for this model (method *GetFile*)
- Tools to organize data objects in partitions and iterate on objects (methods *GetObjects*, *GetMainPartition*, *GetChildren*, *getPartition*, *getElementPartition*)
- Mechanism to give unique names to model objects
- Copy (*clone*) of the model (methods *NewEmpty* and *Paste*)
- Support of earlier model formats for proper conversion of a model loaded from a file written by a previous version of the application (methods *GetFormatVersion* and *SetFormatVersion*)
- Interface to check and update the model if necessary (method *Update*)
- Support of several data models in one application. For this feature use OCAF multi-transaction manager, unique names and GUIDs of the data model (methods *GetModelName*, *GetGUID*)

## Model Persistence

The persistent representation of any OCAF model is contained in an XML or a binary file, which is defined by the format string returned by the method *GetFormat*. The default implementation works with a binary OCAF document format (*BinOcaf*). The other available format is *XmlOcaf*. The class **TObj_Model** declares and provides a default implementation of two virtual methods:

```
virtual Standard_Boolean Load (const char* theFile);
virtual Standard_Boolean SaveAs (const char* theFile);
```

which retrieve and store the model from or in the OCAF file. The descendants should define the following protected method to support Load and Save operations:

```
virtual Standard_Boolean initNewModel (const Standard_Boolean IsNew);
```

This method is called by *Load* after creation of a new model or after its loading from the file; its purpose is to perform the necessary initialization of the model (such as creation of necessary top-level partitions, model update due to version changes etc.). Note that if the specified file does not exist, method *Load* will create a new document and call *initNewModel* with the argument **True**. If the file has been normally loaded, the argument **False** is passed. Thus, a new empty *TObj* model is created by calling *Load* with an empty string or the path to a nonexistent file as argument.

The method *Load* returns **True** if the model has been retrieved successfully (or created a new), or **False** if the model could not be loaded. If no errors have been detected during initialization (model retrieval or creation), the virtual method *AfterRetrieval* is invoked for all objects of the model. This method initializes or updates the objects immediately after the model initialization. It could be useful when some object data should be imported from an OCAF attribute into transient fields which could be changed outside of the OCAF transaction mechanism. Such fields can be stored into OCAF attributes for saving into persistent storage during the save operation.

To avoid memory leaks, the *TObj_Model* class destructor invokes *Close* method which clears the OCAF document and removes all data from memory before the model is destroyed.

For XML and binary persistence of the *TObj* data model the corresponding drivers are implemented in *BinLDrivers*, *BinMObj* and *XmlLDrivers*, *XmlMObj* packages. These packages contain retrieval and storage drivers for the model, model objects and custom attributes from the *TObj* package. The schemas support persistence for the standard OCAF and *TObj* attributes. This is sufficient for the implementation of simple data models, but in some cases it can be reasonable to add specific OCAF attributes to facilitate the storage of the data specific to the application. In this case the schema should be extended using the standard OCAF mechanism.

## Access to the objects in the model

All objects in the model are stored in the main partition and accessed by iterators. To access all model objects use:

```
virtual Handle(TObj_ObjectIterator) GetObjects () const;
```

This method returns a recursive iterator on all objects stored in the model.

```
virtual Handle(TObj_ObjectIterator) GetChildren () const;
```

This method returns an iterator on child objects of the main partition. Use the following method to get the main partition:

```
Handle(TObj_Partition) GetMainPartition() const;
```

To receive the iterator on objects of a specific type *AType* use the following call:

```
GetMainPartition()-&gt;GetChildren(STANDARD_TYPE(AType) );
```

The set of protected methods is provided for descendant classes to deal with partitions:

```
virtual Handle(TObj_Partition) getPartition (const TDF_Label, const
       Standard_Boolean  theHidden) const;
```

This method returns (creating if necessary) a partition in the specified label of the document. The partition can be created as hidden (*TObj_HiddenPartition* class). A hidden partition can be useful to distinguish the data that should not be visible to the user when browsing the model in the application.

The following two methods allow getting (creating) a partition in the sub-label of the specified label in the document (the label of the main partition for the second method) and with the given name:

```
virtual Handle(TObj_Partition) getPartition (const TDF_Label, const
       Standard_Integer theIndex, const TCollection_ExtendedString&
       theName, const Standard_Boolean  theHidden) const;
virtual Handle(TObj_Partition) getPartition (const Standard_Integer
       theIndex, const TCollection_ExtendedString& theName, const
       Standard_Boolean  theHidden) const;
```

If the default object naming and the name register mechanism is turned on, the object can be found in the model by its unique name:

```
Handle(TObj_Object) FindObject (const
       Handle(TCollection_HExtendedString)& theName, const
       Handle(TObj_TNameContainer)& theDictionary) const;
```

## Own model data

The model object can store its own data in the Data label of its main partition, however, there is no standard API for setting and getting these data types. The descendants can add their own data using standard OCAF methods. The enumeration DataTag is defined in *TObj_Model* to avoid conflict of data labels used by this class and its descendants, similarly to objects (see below).

## Object naming

The basic implementation of *TObj_Model* provides the default naming mechanism: all objects must have unique names, which are registered automatically in the data model dictionary. The dictionary is a *TObj_TNameContainer* attribute whose instance is located in the model root label. If necessary, the developer can add several dictionaries into the specific partitions, providing the name registration in the correct name dictionary and restoring the name map after document is loaded from file. To ignore name registering it is necessary to redefine the methods *SetName*, *AfterRetrieval* of the *TObj_Object* class and skip the registration of the object name. Use the following methods for the naming mechanism:

```
Standard_Boolean IsRegisteredName (const
       Handle(TCollection_HExtendedString)& theName, const
       Handle(TObj_TNameContainer)& theDictionary ) const;
```

Returns **True** if the object name is already registered in the indicated (or model) dictionary.

```
void RegisterName (const Handle(TCollection_HExtendedString)& theName,
       const TDF_Label& theLabel, const Handle(TObj_TNameContainer)&
       theDictionary ) const;
```

Registers the object name with the indicated label where the object is located in the OCAF document. Note that the default implementation of the method *SetName* of the object registers the new name automatically (if the name is not yet registered for any other object)

```
void UnRegisterName (const Handle(TCollection_HExtendedString)& theName,
       const Handle(TObj_TNameContainer)& theDictionary ) const;
```

Unregisters the name from the dictionary. The names of *TObj* model objects are removed from the dictionary when the objects are deleted from the model.

```
Handle(TObj_TNameContainer) GetDictionary() const;
```

Returns a default instance of the model dictionary (located at the model root label). The default implementation works only with one dictionary. If there are a necessity to have more than one dictionary for the model objects, it is recommended to redefine the corresponding virtual method of TObj_Object that returns the dictionary where names of objects should be registered.

## API for transaction mechanism

Class *TObj_Model* provides the API for transaction mechanism (supported by OCAF):

```
Standard_Boolean HasOpenCommand() const;
```

Returns True if a Command transaction is open

```
void OpenCommand() const;
```

Opens a new command transaction.

```
void CommitCommand() const;
```

Commits the Command transaction. Does nothing If there is no open Command transaction.

```
void AbortCommand() const;
```

Aborts the Command transaction. Does nothing if there is no open Command transaction.

```
Standard_Boolean IsModified() const;
```

Returns True if the model document has a modified status (has changes after the last save)

```
void SetModified( const Standard_Boolean );
```

Changes the modified status by force. For synchronization of transactions within several *TObj_Model* documents use class *TDocStd_MultiTransactionManager*.

## Model format and version

Class *TObj_Model* provides the descendant classes with a means to control the format of the persistent file by choosing the schema used to store or retrieve operations.

```
virtual TCollection_ExtendedString GetFormat () const;
```

Returns the string *TObjBin* or *TObjXml* indicating the current persistent mechanism. The default value is *TObjBin*. Due to the evolution of functionality of the developed application, the contents and the structure of its data model vary from version to version. *TObj* package provides a basic mechanism supporting backward versions compatibility, which means that newer versions of the application will be able to read Data Model files created by previous versions (but not vice-versa) with a minimum loss of data. For each type of Data Model, all known versions of the data format should be enumerated in increasing order, incremented with every change of the model format. The current version of the model format is stored in the model file and can be checked upon retrieval.

```
Standard_Integer GetFormatVersion() const;
```

Returns the format version stored in the model file

```
void SetFormatVersion(const Standard_Integer theVersion);
```

Defines the format version used for save.

Upon loading a model, the method *initNewModel()*, called immediately after opening a model from disk (on the level of the OCAF document), provides a specific code that checks the format version stored in that model. If it is older than the current version of the application, the data update can be performed. Each model can have its own specific conversion code that performs the necessary data conversion to make them compliant with the current version.

When the conversion ends the user is advised of that by the messenger interface provided by the model (see messaging chapter for more details), and the model version is updated. If the version of data model is not supported (it is newer than the current or too old), the load operation should fail. The program updating the model after version change can be implemented as static methods directly in C++ files of the corresponding Data Model classes, not exposing it to the other parts of the application. These codes can use direct access to the model and objects data (attributes) not using objects interfaces, because the data model API and object classes could have already been changed.

Note that this mechanism has been designed to maintain version compatibility for the changes of data stored in the model, not for the changes of low-level format of data files (such as the storage format of a specific OCAF attribute). If the format of data files changes, a specific treatment on a case-by-case basis will be required.

## Model update

The following methods are used for model update to ensure its consistency with respect to the other models in case of cross-model dependencies:

```
virtual Standard_Boolean Update();
```

This method is usually called after loading of the model. The default implementation does nothing and returns **True**.

```
virtual Standard_Boolean initNewModel( const Standard_Boolean IsNew);
```

This method performs model initialization, check and updates (as described above).

```
virtual void updateBackReferences( const Handle(TObj_Object)& theObj);
```

This method is called from the previous method to update back references of the indicated object after the retrieval of the model from file (see data model - object relationship chapter for more details)

## Model copying

To copy the model between OCAF documents use the following methods:

```
virtual Standard_Boolean Paste (Handle(TObj_Model) theModel,
      Handle(TDF_RelocationTable) theRelocTable = 0 );
```

Pastes the current model to the new model. The relocation table ensures correct copying of the sub-data shared by several parts of the model. It stores a map of processed original objects of relevant types in their copies.

```
virtual Handle(TObj_Model) NewEmpty() = 0;
```

Redefines a pure virtual method to create a new empty instance of the model.

```
void CopyReferences ( const Handle(TObj_Model)& theTarget, const
      Handle(TDF_RelocationTable)& theRelocTable);
```

Copies the references from the current model to the target model.

### Messaging

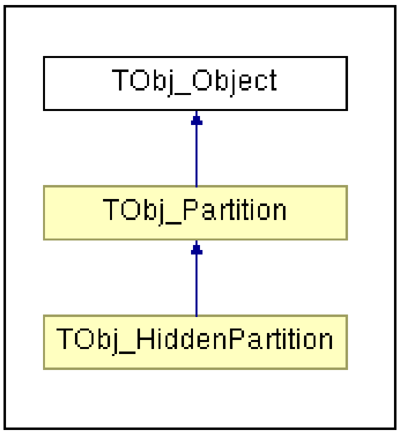
The messaging is organised using Open CASCADE Messenger from the package Message. The messenger is stored as the field of the model instance and can be set and retrieved by the following methods:

```
void SetMessenger( const Handle(Message_Messenger)& );
Handle(Message_Messenger) Messenger() const;
```

A developer should create his own instance of the Messenger bound to the application user interface, and attribute it to the model for future usage. In particular the messenger is used for reporting errors and warnings in the persistence mechanism. Each message has a unique string identifier (key). All message keys are stored in a special resource file TObj.msg. This file should be loaded at the start of the application by call to the appropriate method of the class *Message_MsgFile*.

## Model object

Class *TObj_Object* provides basic interface and default implementation of important features of *TObj* model objects. This implementation defines basic approaches that are recommended for all descendants, and provides tools to facilitate their usage.



**TObj objects hierarchy**

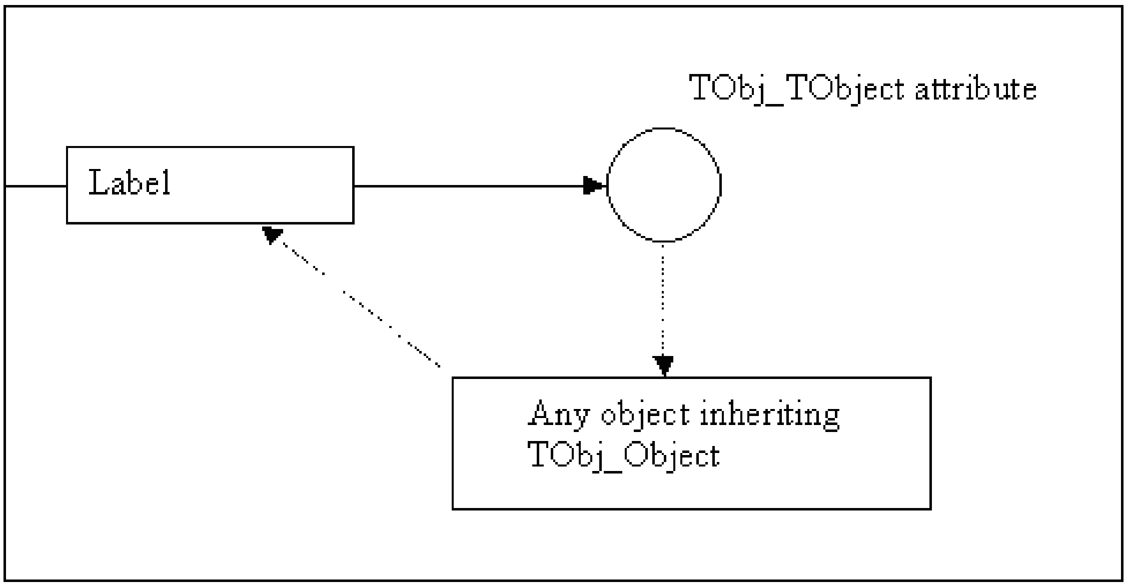
### Separation of data and interface

In the *TObj* data model, the data are separated from the interfaces that manage them. The data belonging to a model object are stored in its root label and sub-labels in the form of standard OCAF attributes. This allows using standard OCAF mechanisms for work with these data, and eases the implementation of the persistence mechanism.

The instance of the interface which serves as an API for managing object data (e.g. represents the model object) is stored in the root label of the object, and typically does not bring its own data. The interface classes are organized in a hierarchy corresponding to the natural hierarchy of the model objects according to the application.

In the text below the term 'object' is used to denote either the instance of the interface class or the object itself (both interface and data stored in OCAF).

The special type of attribute *TObj_TObject* is used for storing instances of objects interfaces in the OCAF tree. *TObj_TObject* is a simple container for the object of type *TObj_Object*. All objects (interfaces) of the data model inherit this class.



**TObj object stored on OCAF label**

## Basic features

The *TObj_Object* class provides some basic features that can be inherited (or, if necessary, redefined) by the descendants:

- Gives access to the model to which the object belongs (method *GetModel*) and to the OCAF label in which the object is stored (method *GetLabel*).
- Supports references (and back references) to other objects in the same or in another model (methods *getReference*, *setReference*, *addReference*, *GetReferences*, *GetBackReferences*, *AddBackReference*, *RemoveBackReference*, *ReplaceReference*)
- Provides the ability to contain child objects, as it is actual for partition objects (methods *GetChildren*, *GetFatherObject*)
- Organizes its data in the OCAF structure by separating the sub-labels of the main label intended for various kinds of data and providing tools to organize these data (see below). The kinds of data stored separately are:
    - Child objects stored in the label returned by the method *GetChildLabel*
    - References to other objects stored in the label returned by the method *GetReferenceLabel*
    - Other data, both common to all objects and specific for each subtype of the model object, are stored in the label returned by the method *GetDataLabel*

- Provides unique names of all objects in the model (methods *GetDictionary*, *GetName*, *SetName*)
- Provides unified means to maintain persistence (implemented in descendants with the help of macros *DECLARE_TOBJOCAF_PERSISTENCE* and *IMPLEMENT_TOBJOCAF_PERSISTENCE*)
- Allows an object to remove itself from the OCAF document and check the depending objects can be deleted according to the back references (method *Detach*)
- Implements methods for identification and versioning of objects
- Manages the object interaction with OCAF Undo/Redo mechanism (method *IsAlive*, *AfterRetrieval*, *BeforeStoring*)
- Allows make a clone (methods *Clone*, *CopyReferences*, *CopyChildren*, *copyData*)
- Contains additional word of bit flags (methods *GetFlags*, *SetFlags*, *TestFlags*, *ClearFlags*)
- Defines the interface to sort the objects by rank (methods *GetOrder*, *SetOrder*)
- Provides a number of auxiliary methods for descendants to set/get the standard attribute values, such as int, double, string, arrays etc.

An object can be received from the model by the following methods:

```
static Standard_Boolean GetObj ( const TDF_Label& theLabel,
       Handle(TObj_Object)& theResObject, const Standard_Boolean isSuper
       = Standard_False );
```

Returns *True* if the object has been found in the indicated label (or in the upper level label if *isSuper* is *True*).

```
Handle(TObj_Object) GetFatherObject ( const Handle(Standard_Type)&
       theType = NULL ) const;
```

Returns the father object of the indicated type for the current object (the direct father object if the type is NULL).

## Data layout and inheritance

As far as the data objects are separated from the interfaces and stored in the OCAF tree, the functionality to support inheritance is required. Each object has its own data and references stored in the labels in the OCAF tree. All data are stored in the sub-tree of the main object label. If it is necessary to inherit a class from the base class, the descendant class should use different labels for data and references than its ancestor.

Therefore each *TObj* class can reserve the range of tags in each of *Data*, *References*, and *Child* sub-labels. The reserved range is declared by the enumeration defined in the class scope (called DataTag, RefTag, and ChildTag, respectively). The item *First* of the enumeration of each type is defined via the *Last* item of the corresponding enumeration of the parent class, thus ensuring that the tag numbers do not overlap. The item *Last* of the enumeration defines the last tag reserved by this class. Other items of the enumeration define the tags used for storing particular data items of the object. See the declaration of the TObj_Partition class for the example.

*TObj_Object* class provides a set of auxiliary methods for descendants to access the data stored in sub-labels by their tag numbers:

```
TDF_Label getDataLabel (const Standard_Integer theRank1, const
      Standard_Integer theRank2 = 0) const;
TDF_Label getReferenceLabel (const Standard_Integer theRank1, const
      Standard_Integer theRank2 = 0) const;
```

Returns the label in *Data* or *References* sub-labels at a given tag number (theRank1). The second argument, theRank2, allows accessing the next level of hierarchy (theRank2-th sub-label of theRank1-th data label). This is useful when the data to be stored are represented by multiple OCAF attributes of the same type (e.g. sequences of homogeneous data or references).

The get/set methods allow easily accessing the data located in the specified data label for the most widely used data types (*Standard_Real*, *Standard_Integer*, *TCollection_HExtendedString*, *TColStd_HArray1OfReal*, *TColStd_HArray1OfInteger*, *TColStd_HArray1OfExtendedString*). For instance, methods provided for real numbers are:

```
Standard_Real getReal (const Standard_Integer theRank1, const
      Standard_Integer theRank2 = 0) const;
Standard_Boolean setReal (const Standard_Real theValue, const
      Standard_Integer theRank1, const Standard_Integer theRank2 = 0,
      const Standard_Real theTolerance = 0.) const;
```

Similar methods are provided to access references to other objects:

```
Handle(TObj_Object) getReference (const Standard_Integer theRank1, const
      Standard_Integer theRank2 = 0) const;
Standard_Boolean setReference (const Handle(TObj_Object) &theObject,
      const Standard_Integer theRank1, const Standard_Integer theRank2
      = 0);
```

The method *addReference* gives an easy way to store a sequence of homogeneous references in one label.

```
TDF_Label addReference (const Standard_Integer theRank1, const
      Handle(TObj_Object) &theObject);
```

Note that while references to other objects should be defined by descendant classes individually according to the type of object, *TObj_Object* provides methods to manipulate (check, remove, iterate) the existing references in the uniform way, as described below.

## Persistence

The persistence of the *TObj* Data Model is implemented with the help of standard OCAF mechanisms (a schema defining necessary plugins, drivers, etc.). This implies the possibility to store/retrieve all data that are stored as standard OCAF attributes., The corresponding handlers are added to the drivers for *TObj*-specific attributes.

The special tool is provided for classes inheriting from *TObj_Object* to add the new types of persistence without regeneration of the OCAF schema. The class *TObj_Persistence* provides basic means for that:

- automatic run-time registration of object types
- creation of a new object of the specified type (one of the registered types)

Two macros defined in the file TObj_Persistence.hxx have to be included in the definition of each model object class inheriting TObj_Object to activate the persistence mechanism:

```
DECLARE_TOBJOCAF_PERSISTENCE (classname, ancestorname)
```

Should be included in the private section of declaration of each class inheriting *TObj_Object* (hxx file). This macro adds an additional constructor to the object class, and declares an auxiliary (private) class inheriting *TObj_Persistence* that provides a tool to create a new object of the proper type.

```
IMPLEMENT_TOBJOCAF_PERSISTENCE (classname)
```

Should be included in .cxx file of each object class that should be saved and restored. This is not needed for abstract types of objects. This macro implements the functions declared by the previous macro and creates a static member that automatically registers that type for persistence.

When the attribute *TObj_TObject* that contains the interface object is saved, its persistence handler stores the runtime type of the object class. When the type is restored the handler dynamically recognizes the type and creates the corresponding object using mechanisms provided by *TObj_Persistence*.

## Names of objects

All *TObj* model objects have names by which the user can refer to the object. Upon creation, each object receives a default name, constructed from the prefix corresponding to the object type (more precisely, the prefix is defined by the partition to which the object belongs), and the index of the object in the current partition. The user has the possibility to change this name. The uniqueness of the name in the model is ensured by the naming mechanism (if the name is already used, it cannot be attributed to another object). This default implementation of *TObj* package works with a single instance of the name container (dictionary) for name registration of objects and it is enough in most simple projects. If necessary, it is easy to redefine a couple of object methods (for instance *GetDictionary*()) and to take care of construction and initialization of containers.

This functionality is provided by the following methods:

```
virtual Handle(TObj_TNameContainer) GetDictionary() const;
```

Returns the name container where the name of object should be registered. The default implementation returns the model name container.

```
Handle(TCollection_HExtendedString) GetName() const;
Standard_Boolean GetName( TCollection_ExtendedString& theName ) const;
Standard_Boolean GetName( TCollection_AsciiString& theName ) const;
```

Returns the object name. The methods with in / out argument return False if the object name is not defined.

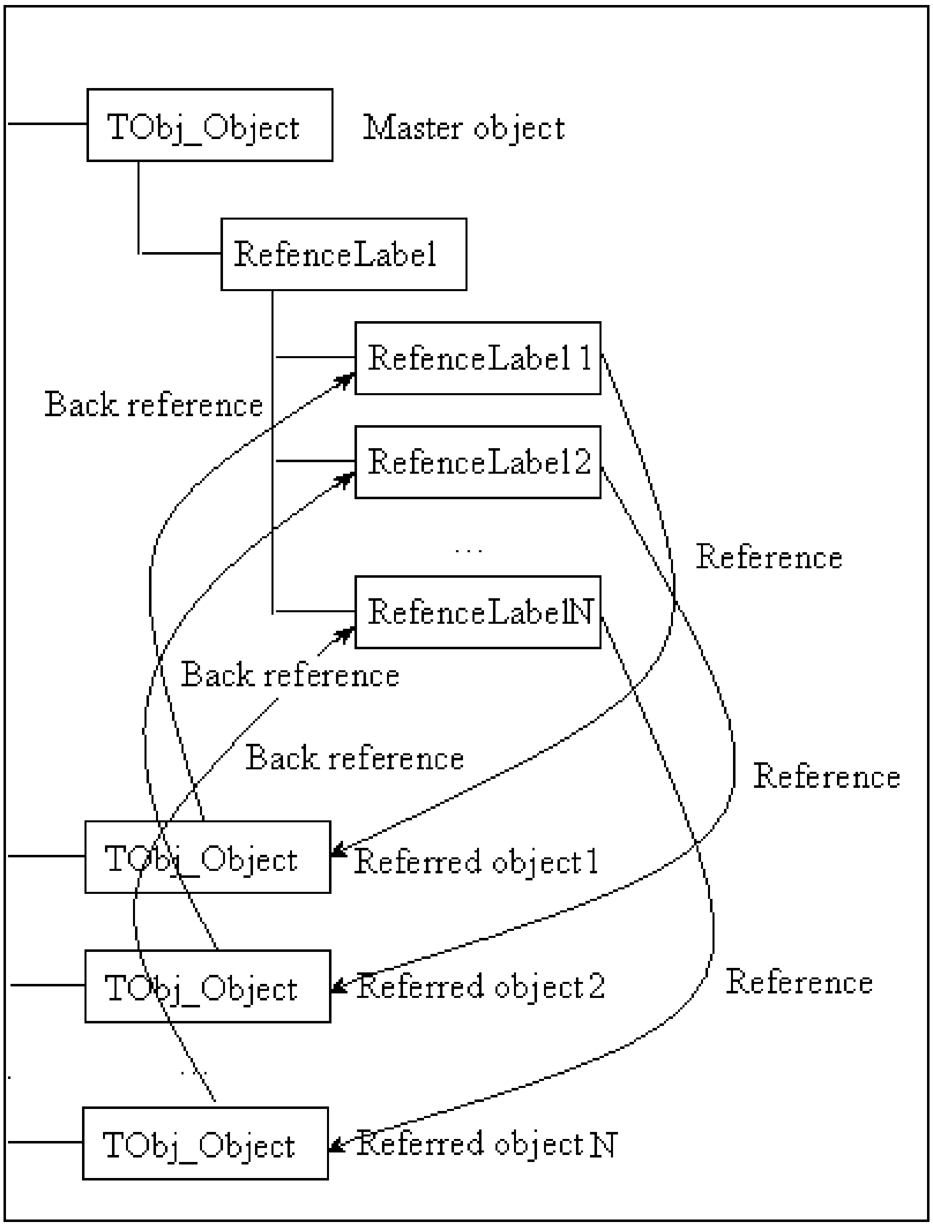
```
virtual Standard_Boolean SetName ( const
      Handle(TCollection_HExtendedString)& theName ) const;
Standard_Boolean SetName         ( const
      Handle(TCollection_HAsciiString)& theName ) const;
Standard_Boolean SetName         ( const Standard_CString theName )
      const;
```

Attributes a new name to the object and returns **True** if the name has been attributed successfully. Returns False if the name has been already attributed to another object. The last two methods are short-cuts to the first one.

### References between objects

Class *TObj_Object* allows creating references to other objects in the model. Such references describe relations among objects which are not adequately reflected by the hierarchical objects structure in the model (parent-child relationship).

The references are stored internally using the attribute TObj_TReference. This attribute is located in the sub-label of the referring object (called *master*) and keeps reference to the main label of the referred object. At the same time the referred object can maintain the back reference to the master object.



**Objects relationship**

The back references are stored not in the OCAF document but as a transient field of the object; they are created when the model is restored from file, and updated automatically when the references are manipulated. The class *TObj_TReference* allows storing references between objects from different *TObj* models, facilitating the construction of complex relations between objects.

The most used methods for work with references are:

```
virtual Standard_Boolean HasReference( const Handle(TObj_Object)&
        theObject) const;
```

Returns True if the current object refers to the indicated object.

```
virtual Handle(TObj_ObjectIterator) GetReferences ( const
        Handle(Standard_Type)& theType = NULL ) const;
```

Returns an iterator on the object references. The optional argument *theType* restricts the types of referred objects, or does not if it is NULL.

```
virtual void RemoveAllReferences();
```

Removes all references from the current object.

```
virtual void RemoveReference( const Handle(TObj_Object)& theObject );
```

Removes the reference to the indicated object.

```
virtual Handle(TObj_ObjectIterator) GetBackReferences ( const
      Handle(Standard_Type)& theType = NULL ) const;
```

Returns an iterator on the object back references. The argument theType restricts the types of master objects, or does not if it is NULL.

```
virtual void ReplaceReference  ( const Handle(TObj_Object)&
      theOldObject,  const Handle(TObj_Object)& theNewObject );
```

Replaces the reference to theOldObject by the reference to *theNewObject*. The handle theNewObject may be NULL to remove the reference.

```
virtual Standard_Boolean RelocateReferences  ( const TDF_Label&
      theFromRoot,  const TDF_Label& theToRoot, const Standard_Boolean
      theUpdateackRefs = Standard_True );
```

Replaces all references to a descendant label of *theFromRoot* by the references to an equivalent label under *theToRoot*. Returns **False** if the resulting reference does not point at a *TObj_Object*. Updates back references if theUpdateackRefs is **True**.

```
virtual Standard_Boolean CanRemoveReference ( const Handle(TObj_Object)&
      theObj) const;
```

Returns **True** if the reference can be removed and the master object will remain valid (*weak* reference). Returns **False** if the master object cannot be valid without the referred object (*strong* reference). This affects the behaviour of objects removal from the model – if the reference cannot be removed, either the referred object will not be removed, or both the referred and the master objects will be removed (depends on the deletion mode in the method **Detach**)

## Creation and deletion of objects

It is recommended that all objects inheriting from *TObj_Object* should implement the same approach to creation and deletion.

The object of the *TObj* data model cannot be created independently of the model instance, as far as it stores the object data in OCAF data structures. Therefore an object class cannot be created directly as its constructor is protected.

Instead, each object should provide a static method *Create*(), which accepts the model, with the label, which stores the object and other type-dependent parameters necessary for proper definition of the object. This method creates a new object with its data (a set of OCAF attributes) in the specified label, and returns a handle to the object's interface.

The method *Detach*() is provided for deletion of objects from OCAF model. Object data are deleted from the corresponding OCAF label; however, the handle on object remains valid. The only operation available after object deletion is the method *IsAlive*() checking whether the object has been deleted or not, which returns False if the object has been deleted.

When the object is deleted from the data model, the method checks whether there are any alive references to the object. Iterating on references the object asks each referring (master) object whether the reference can be removed. If the master object can be unlinked, the reference is removed, otherwise the master object will be removed too or the referred object will be kept alive. This check is performed by the method *Detach* , but the behavior depends on the deletion mode *TObj_DeletingMode*:

- **TObj_FreeOnly** – the object will be destroyed only if it is free, i.e. there are no references to it from other objects
- **TObj_KeepDepending** – the object will be destroyed if there are no strong references to it from master objects (all references can be unlinked)
- **TObj_Force** – the object and all depending master objects that have strong references to it will be destroyed.

The most used methods for object removing are:

```
virtual Standard_Boolean CanDetachObject (const TObj_DeletingMode
      theMode = TObj_FreeOnly );
```

Returns **True** if the object can be deleted with the indicated deletion mode.

```
virtual Standard_Boolean Detach ( const TObj_DeletingMode theMode =
      TObj_FreeOnly );
```

Removes the object from the document if possible (according to the indicated deletion mode). Unlinks references from removed objects. Returns **True** if the objects have been successfully deleted.

## Transformation and replication of object data

*TObj_Object* provides a number of special virtual methods to support replications of objects. These methods should be redefined by descendants when necessary.

```
virtual Handle(TObj_Object) Clone (const TDF_Label& theTargetLabel,
      Handle(TDF_RelocationTable) theRelocTable = 0);
```

Copies the object to theTargetLabel. The new object will have all references of its original. Returns a handle to the new object (null handle if fail). The data are copied directly, but the name is changed by adding the postfix *_copy*. To assign different names to the copies redefine the method:

```
virtual Handle(TCollection_HExtendedString) GetNameForClone ( const
      Handle(TObj_Object)& ) const;
```

Returns the name for a new object copy. It could be useful to return the same object name if the copy will be in the other model or in the other partition with its own dictionary. The method *Clone* uses the following public methods for object data replications:

```
virtual void CopyReferences (const const Handle(TObj_Object)&
      theTargetObject, const Handle(TDF_RelocationTable)
      theRelocTable);
```

Adds to the copy of the original object its references.

```
virtual void CopyChildren (TDF_Label& theTargetLabel, const
      Handle(TDF_RelocationTable) theRelocTable);
```

Copies the children of an object to the target child label.

## Object flags

Each instance of *TObj_Object* stores a set of bit flags, which facilitate the storage of auxiliary logical information assigned to the objects (object state). Several typical state flags are defined in the enumeration *ObjectState*:

- *ObjectState_Hidden* – the object is marked as hidden
- *ObjectState_Saved* – the object has (or should have) the corresponding saved file on disk
- *ObjectState_Imported* – the object is imported from somewhere
- *ObjectState_ImportedByFile* – the object has been imported from file and should be updated to have correct relations with other objects
- *ObjectState_Ordered* – the partition contains objects that can be ordered.

The user (developer) can define any new flags in descendant classes. To set/get an object, the flags use the following methods:

```
Standard_Integer GetFlags() const;
void SetFlags( const Standard_Integer theMask );
Stadnard_Boolean TestFlags( const Standard_Integer theMask ) const;
void ClearFlags( const Standard_Integer theMask = 0 );
```

In addition, the generic virtual interface stores the logical properties of the object class in the form of a set of bit flags. Type flags can be received by the method:

```
virtual Standard_Integer GetTypeFlags() const;
```

The default implementation returns the flag **Visible** defined in the enumeration *TypeFlags*. This flag is used to define visibility of the object for the user browsing the model (see class

*TObj_HiddenPartition*). Other flags can be added by the applications.

### Partitions

The special kind of objects defined by the class *TObj_Partition* (and its descendant *TObj_HiddenPartition*) is provided for partitioning the model into a hierarchical structure. This object represents the container of other objects. Each *TObj* model contains the main partition that is placed in the same OCAF label as the model object, and serves as a root of the object's tree. A hidden partition is a simple partition with a predefined hidden flag.

The main partition object methods:

```
TDF_Label NewLabel() const;
```

Allocates and returns a new label for creation of a new child object.

```
void SetNamePrefix  ( const Handle(TCollection_HExtendedString)&
       thePrefix);
```

Defines the prefix for automatic generation of names of the newly created objects.

```
Handle(TCollection_HExtendedString) GetNamePrefix() const;
```

Returns the current name prefix.

```
Handle(TCollection_HExtendedString) GetNewName ( const Standard_Boolean
       theIsToChangeCount) const;
```

Generates the new name and increases the internal counter of child objects if theIsToChangeCount is **True**.

```
Standard_Integer GetLastIndex() const;
```

Returns the last reserved child index.

```
void SetLastIndex( const Standard_Integer theIndex );
```

Sets the last reserved index.

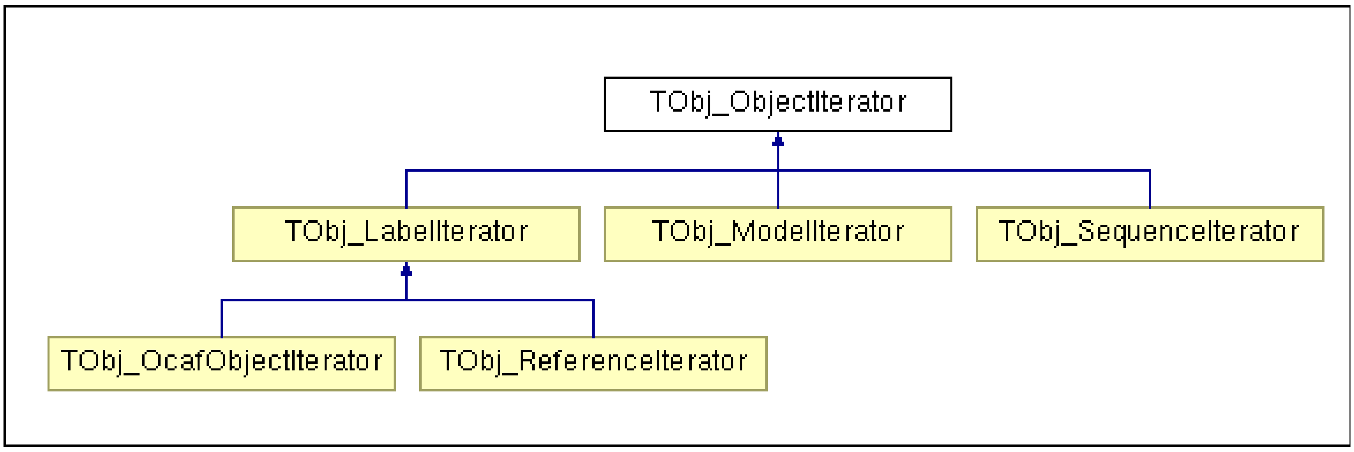
## Auxiliary classes

Apart from the model and the object, package *TObj* provides a set of auxiliary classes:

- *TObj_Application* – defines OCAF application supporting existence and operation with *TObj* documents.
- *TObj_Assistant* – class provides an interface to the static data to be used during save and load operations on models. In particular, in case of cross-model dependencies it allows passing information on the parent model to the OCAF loader to correctly resolve the references when loading a dependent model.

- *TObj_TReference* – OCAF attribute describes the references between objects in the *TObj* model(s). This attribute stores the label of the referred model object, and provides transparent cross-model references. At runtime, these references are simple Handles; in persistence mode, the cross-model references are automatically detected and processed by the persistence mechanism of *TObj_TReference* attribute.
- Other classes starting with *TObj_T...* – define OCAF attributes used to store TObj-specific classes and some types of data on OCAF labels.
- Iterators – a set of classes implementing *TObj_ObjectIterator* interface, used for iterations on *TObj* objects:
    - *TObj_ObjectIterator* – a basic abstract class for other *TObj* iterators. Iterates on *TObj_Object* instances.
    - *TObj_LabelIterator* – iterates on object labels in the *TObj* model document
    - *TObj_ModelIterator* – iterates on all objects in the model. Works with sequences of other iterators.
    - *TObj_OcafObjectIterator* – Iterates on *TObj* data model objects. Can iterate on objects of a specific type.
    - *TObj_ReferenceIterator* – iterates on object references.
    - *TObj_SequenceIterator* – iterates on a sequence of *TObj* objects.
    - *TObj_CheckModel* – a tool that checks the internal consistency of the model. The basic implementation checks only the consistency of references between objects.

The structure of *TObj* iterators hierarchy is presented below:



**Hierarchy of iterators**

## Packaging

The *TObj* sources are distributed in the following packages:

- *TObj* – defines basic classes that implement *TObj* interfaces for OCAF-based modelers.
- *BinLDrivers, XmlLDrivers* – binary and XML driver of *TObj* package
- *BinLPlugin, XmlLPlugin* – plug-in for binary and XML persistence

* *BinMObj, XmlMObj* – binary and XML drivers to store and retrieve specific *TObj* data to or from OCAF document
* *TKBinL, TKXmlL* – toolkits of binary and XML persistence

# GLOSSARY

* **Application** – a document container holding all documents containing all application data.
* **Application data** – the data produced by an application, as opposed to data referring to it.
* **Associativity of data** – the 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.
* **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. For 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.
* **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++, the application behavior is implemented in virtual functions redefined in these derived classes. This is known as overriding.

- **GUID** – Global Universal ID. A string of 37 characters intended to uniquely identify an object. For 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. It has a name in the form of an entry, which identifies its place in the data framework.
- **Modified label** – containing attributes whose data has been modified.
- **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.
- **Tag list** – a list of integers, which identify 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 topology of a model changes during the modeling, 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 label** – in a data framework, this is a label, which is already recomputed in the scope of regeneration sequence and includes 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.