Open CASCADE Technology
6.9.1

TObj Package

September 25, 2015

Contents

1 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 re-generating the schema.

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.

1.1 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 re-develop 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.

2   TObj Model

2.1   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.

Figure 1: 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:
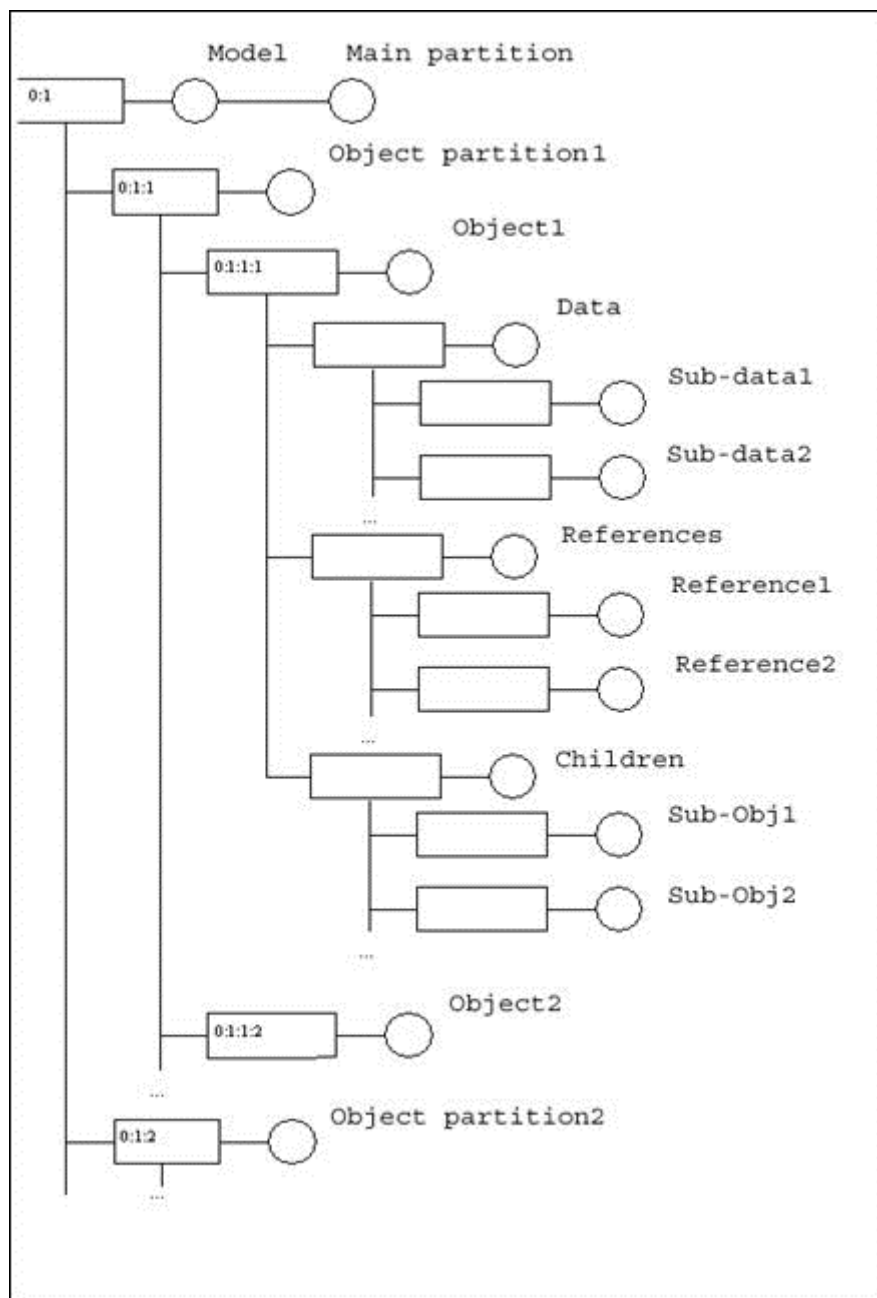

Figure 2: TObj Data Model mapped on OCAF document

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.

2.2 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*)

2.3 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.

2.4   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()->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;
```

2.5   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).

2.6   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. Ther 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.

2.7 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*.

2.8 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.

2.9 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)

2.10   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.

2.11   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*.

3   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.
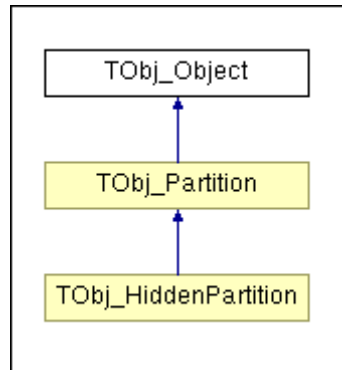

Figure 3: TObj objects hierarchy

3.1   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.
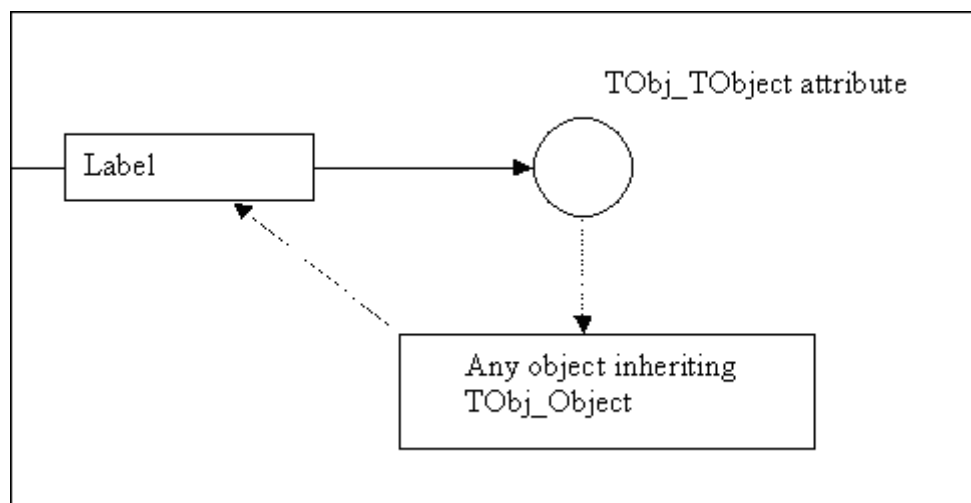

Figure 4: TObj object stored on OCAF label

3.2 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).

3.3   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.

3.4   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 *T-Obj_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*.

3.5 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.

3.6 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.
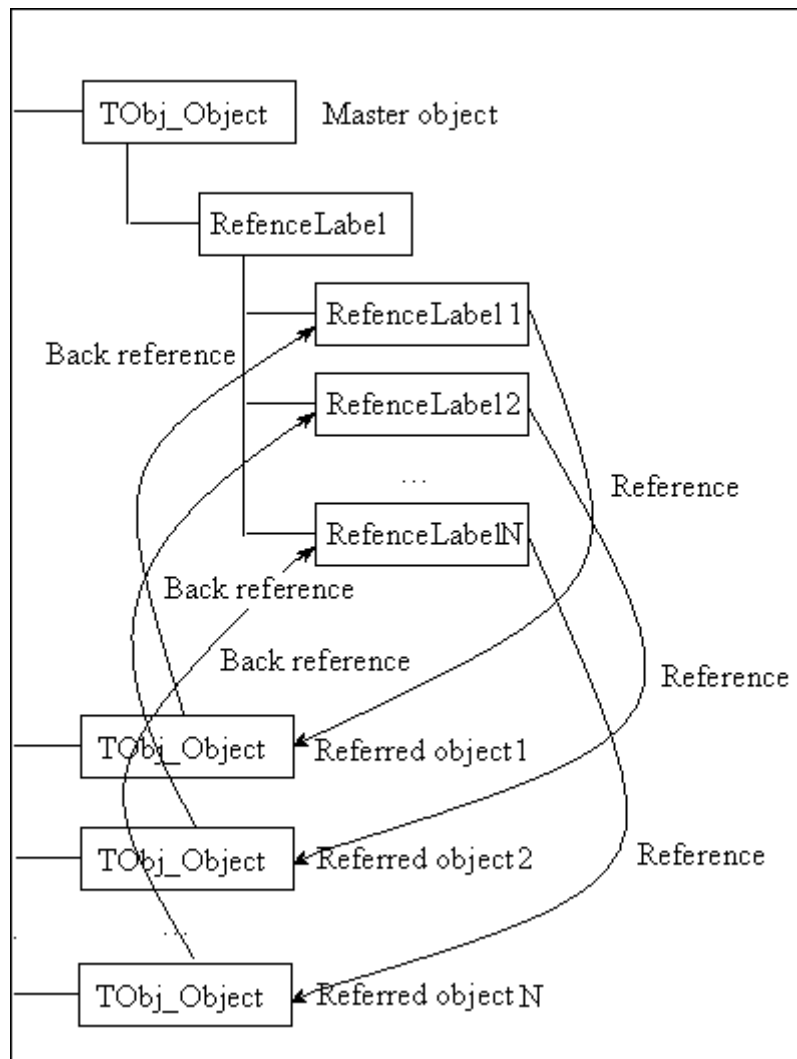
Figure 5: Objects relationship

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**)

3.7   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.

3.8   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.

3.9   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.

3.10 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.

4   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.

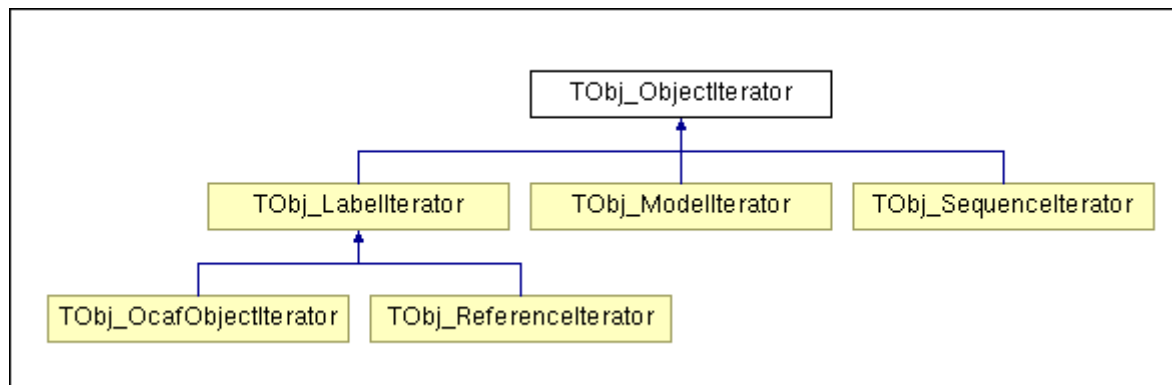The structure of *TObj* iterators hierarchy is presented below:

Figure 6: Hierarchy of iterators

5   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* – plugin 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