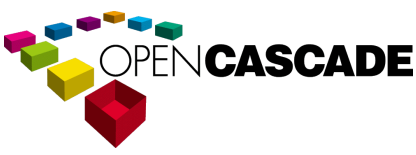
Application Framework

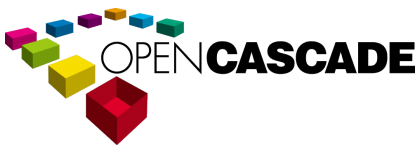
White Paper

Version 6.3 / September 2008



NOTICE FOR USERS:

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



Tour Opus 12

77, Esplanade du Général de Gaulle

92914 PARIS LA DEFENSE

FRANCE

What is OCAF ?

Purpose of OCAF

The Open CASCADE Application Framework (OCAF) 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
- 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

By providing architectural guidance and ready-to-use solutions to these issues, OCAF helps you to develop your application significantly faster: you concentrate on the application's functionality.

Overview of the Architecture

OCAF provides you with an object-oriented Application-Document-Attribute model. This consists of C++ class libraries. The main class, `Application`, is an abstract class in charge of handling documents during the working session. Services provided by this class include:

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

The document, implemented by the concrete class `Document`, is the container for the application data. Its main purpose is to centralize notifications of data editing in order to provide Undo-Redo. 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).

Application data are attributes, that is, 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 (the Data Framework is described in the next chapter, *A Look Inside OCAF*). A wide range of attributes come with OCAF, including:

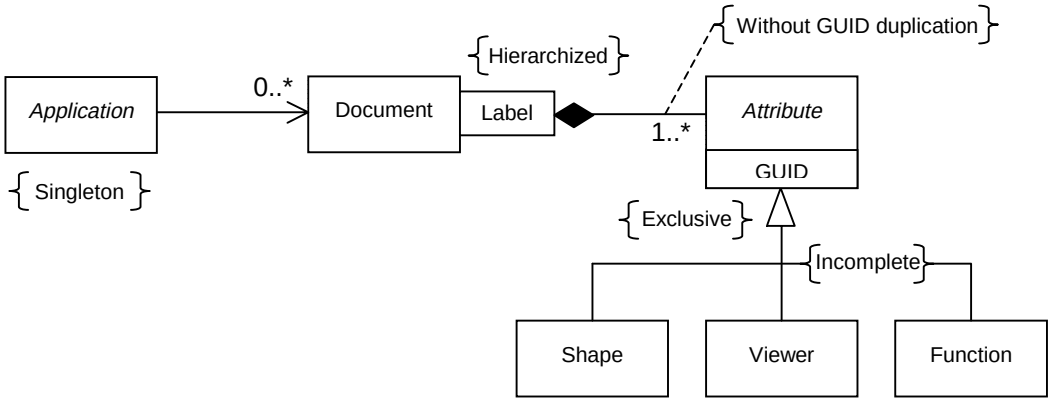
- Primitive attributes such as Integer, Real, Name and Comment
- Shape attribute containing the geometry of the whole model or elements of it
- Other geometric attributes such as Datums (points, axis and plane) and Constraints such as tangent-to, at-a-given-distance, from-a-given-angle, concentric, etc.
- Modeling step and Function attributes — the purpose of these attributes is to rebuild objects after they have been modified (parameterization of models)
- Visualization attributes — these attributes allow data to be visualized in a 2D or 3D viewer
- User attributes, that is, attributes typed by the application

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 to its persistent homolog (persistence is briefly presented in the paragraph *Persistent Data Storage* in the next chapter)

Figure 1 below illustrates this architecture.

Figure 1 The Application-Document-Attribute model



Note As OCAF uses other modules of Open CASCADE — see the technical overview of Open CASCADE in the appendix — the Shape attribute 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.

Getting Started

At the beginning of your development, you first define an application class by inheriting from the `Application` abstract class. You only have to create and determine the resources of the application for specifying the format of your documents (you generally use the standard one) and their file extension.

Then, you design the application data model by organizing attributes you choose among those provided with OCAF. You can specialize these attributes using the User attribute. For example, if you need a reflection coefficient, you aggregate a User attribute identified as a reflection coefficient with a Real attribute containing the value of the coefficient (as such, you don’t define a new class).

If you need application specific data not provided with OCAF, for example, to incorporate a finite element model in the data structure, you define a new attribute class containing the mesh, and you include its persistent homolog in a new file format.

Once you have implemented the commands which create and modify the data structure according to your specification, OCAF provides you, without any additional programming:

- Persistent reference to any data, including geometric elements — several documents can be linked with such reference
- Document-View association
- Ready-to-use functions such as
  - Undo-redo
  - Save and open application data

Finally, you develop the application's graphical user interface using the toolkit of your choice, for example:

- KDE Qt or GNOME GTK+ on Linux
- Microsoft Foundation Classes (MFC) on Windows
- Motif on Sun
- Other commercial products such as Ilog Views

You can also implement the user interface in the Java language using the Swing-based Java Application Desktop component (JAD) provided with OCAF.

Benefits of OCAF

As you use the architecture provided by OCAF, the design of your application is made easy: the application developer concentrates on the functionality instead of the underlying mechanisms required to support this functionality.

Also, thanks to the coupling with the other Open CASCADE modules, your application can rapidly be prototyped. In addition, the final application can be developed by industrializing the prototype — you don't need to restart the development from scratch.

Last but not least, you base your application on an Open Source component: this guarantees the perenniality of your development.

A Look Inside OCAF

The Design of OCAF

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:

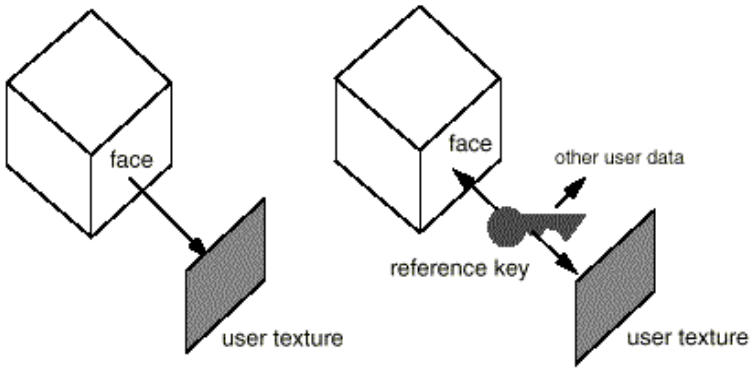
- a color
- a material
- information that a particular edge is blended

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.

Figure 2 Topology driven versus reference-key driven approaches



Topology driven **Reference-key driven**

Topological naming

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, newed and deleted faces), is used by the naming algorithm to maintain the topology attached to the reference-key. As such, on a parameterized 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 2 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, newed 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 modeling libraries.

Aggregation of attributes

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 parameterized 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 realization of the reference-key model presented in the previous paragraph. It implements the reference-key as label objects, organized in a tree structure characterized by the following features:

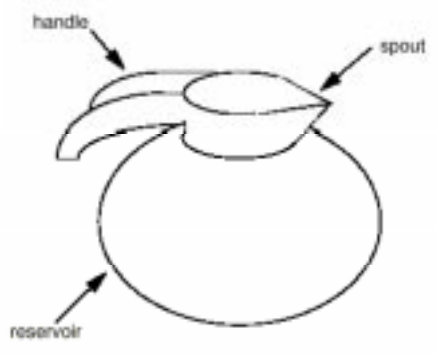
- A document contains only one tree of labels
- Each label has a tag expressed as an integer value unique at its level in the tree
- A label is identified by a string — the entry — built by concatenation of tags from the root of the tree, for example `[0:1:2:1]`
- Attributes are of a type identified by a universal unique identifier (GUID)
- Attributes are attached to labels; a label may refer to many attributes as long as each has a different GUID

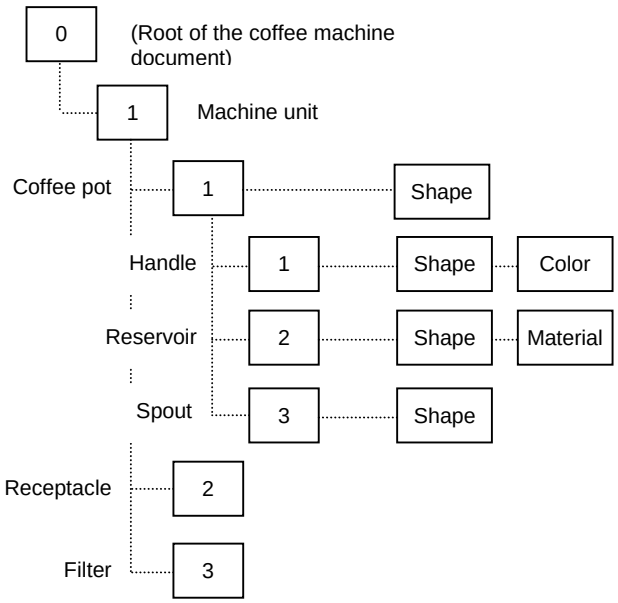
As such, each piece of data has a unique persistent address made up of the document path, its entry and the GUID of its class.

For example, an 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.

Figure 3 below illustrates this data structure.

Figure 3 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]` (as in the Figure 3 above) whereas the handle of the second pot would be `[0:1:4:1]`. This way, we avoid any confusion between coffee pot handles.

Note The purpose of the label hierarchy is to provide the data with persistent addresses. In particular, applications which show the end-user the data in a tree-list view do not display this hierarchy. For that, an attribute (`TreeNode`) which you insert in the data structure is provided.

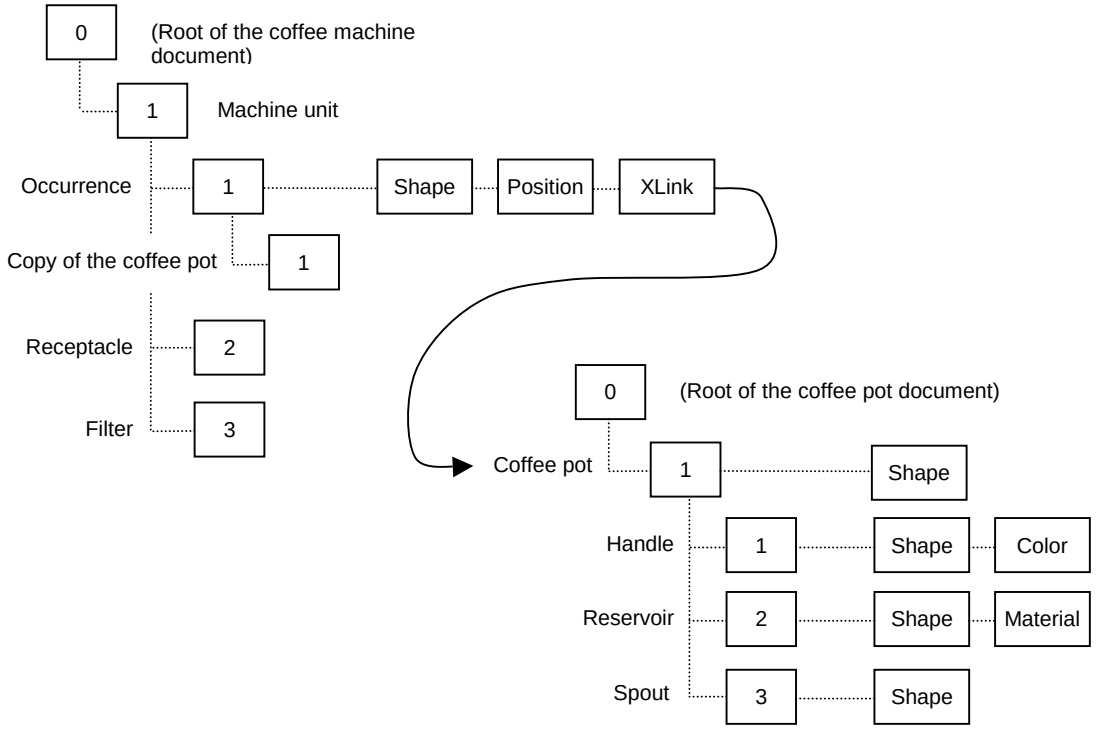
Compound documents

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]`).

Figure 4 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 consists simply of managing 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.

Persistent Data Storage

In OCAF, persistence, that is, the mechanism used to save a document in a file, is based on an explicit formal description of the data saved.

When you open a document, the application reads the corresponding file and first creates a memory representation of it. This representation is then converted to the application data model — the OCAF-based data structure the application operates on. The file's memory representation consists of objects defined by classes known as persistent. The persistent classes needed by an application to save its documents make the application's data schema. This schema defines the way the data are organized in the file — the format of the data. In other words, the file is simply an ASCII dump of the persistent data defined by the schema, the persistent data being created from the application data model during the save process.

Only canonical information is saved. As a matter of fact, the application data model usually contains additional data to optimize processing. For example, the

persistent Bézier curve is defined by its poles, whereas its data model equivalent also contains coefficients used to compute a point at a given parameter. The additional data is calculated when the document is opened.

The major advantages of this approach are the following:

- Providing that the data format is published, files created by OCAF-based applications can be read without needing a runtime of the application (openness)
- Although the persistence approach makes the data format more stable, OCAF provides a framework for managing compatibility of data between versions of the application — modification of the data format is supported through the versioning of schema.

OCAF includes a ready-to-use schema suitable for most applications. However, it can be extended if needed. For that, the only things you have to do are:

- To define the additional persistent attributes
- To implement the functions converting these persistent attribute to and from the application data model.

Note Applications using compound documents extensively (saving data in many files linked together) should implement data management services. As a matter of fact, it's out the scope of OCAF to provide functions such as:

- Version and configuration management of compound documents
- Querying a referenced document for its referencing documents

In order to ease the delegation of document management to a data management application, OCAF encapsulates the file management functions in a driver (the meta-data driver). You have to implement this driver for your application to communicate with the data management system of your choice.