Open CASCADE Technology
6.9.1

OCAF White-Paper

September 25, 2015

Contents

1   What is OCAF ?

1.1   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.

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

Also, thanks to the coupling with the other Open CASCADE Technology 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 long-term usefulness of your development.

1.2   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 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 (*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`)
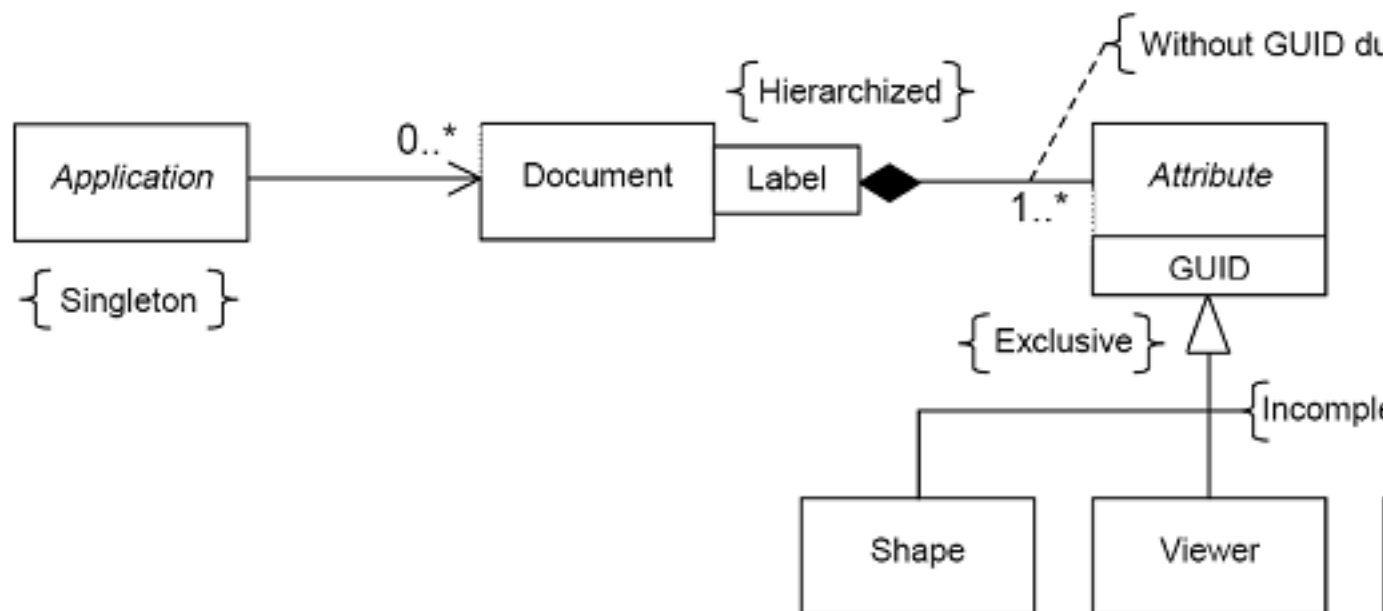

Figure 1: The Application-Document-Attribute model

OCAF uses other modules of Open CASCADE Technology — 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.

1.3 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 homologue 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.

1.4 An example of OCAF usage

To create a useful OCAF-based application, it is necessary to redefine two deferred methods: *Formats* and *ResourcesName*

In the *Formats* method, add the format of the documents, which need to be read by the application and may have been built in other applications.

For example:

```
void myApplication::Formats(TColStd_SequenceOfExtendedString& Formats)
{
  Formats.Append(TCollection_ExtendedString ("OCAF-myApplication"));
}
```

In the *ResourcesName* method, you only define the name of the resource file. This file contains several definitions for the saving and opening mechanisms associated with each format and calling of the plug-in file.

```
Standard_CString myApplication::ResourcesName()
{
  return Standard_CString ("Resources");
}
```

To obtain the saving and opening mechanisms, it is necessary to set two environment variables: *CSF_Plugin-Defaults*, which defines the path of the plug-in file, and *CSF_ResourcesDefault*, which defines the resource file:

```
SetEnvironmentVariable ( "CSF_ResourcesDefaults",myDirectory);
SetEnvironmentVariable ( "CSF_PluginDefaults",myDirectory);
```

The plugin and the resource files of the application will be located in *myDirector*. The name of the plugin file must be *Plugin*.

Resource File

The resource file describes the documents (type and extension) and the type of data that the application can manipulate by identifying the storage and retrieval drivers appropriate for this data.

Each driver is unique and identified by a GUID generated, for example, with the *uuidgen* tool in Windows.

Five drivers are required to use all standard attributes provided within OCAF:

- the schema driver (ad696002-5b34-11d1-b5ba-00a0c9064368)

- the document storage driver (ad696000-5b34-11d1-b5ba-00a0c9064368)

- the document retrieval driver (ad696001-5b34-11d1-b5ba-00a0c9064368)

- the attribute storage driver (47b0b826-d931-11d1-b5da-00a0c9064368)

- the attribute retrieval driver (47b0b827-d931-11d1-b5da-00a0c9064368)

These drivers are provided as plug-ins and are located in the *PappStdPlugin* library.

For example, this is a resource file, which declares a new model document OCAF-MyApplication:

```
formatlist:OCAF-MyApplication
OCAF-MyApplication.Description: MyApplication Document Version 1.0
OCAF-MyApplication.FileExtension: sta
OCAF-MyApplication.StoragePlugin: ad696000-5b34-11d1-b5ba-00a0c9064368
OCAF-MyApplication.RetrievalPlugin: ad696001-5b34-11d1-b5ba-00a0c9064368
OCAF-MyApplicationSchema: ad696002-5b34-11d1-b5ba-00a0c9064368
OCAF-MyApplication.AttributeStoragePlugin: 47b0b826-d931-11d1-b5da-00a0c9064368
OCAF-MyApplication.AttributeRetrievalPlugin: 47b0b827-d931-11d1-b5da-00a0c9064368
```

Plugin File

The plugin file describes the list of required plug-ins to run the application and the libraries in which plug-ins are located.

You need at least the *FWOSPlugin* and the plug-in drivers to run an OCAF application.

The syntax of each item is *Identification.Location Library_Name,* where:

- Identification is GUID.

- Location defines the location of the Identification (where its definition is found).

- Library_Name is the name (and path to) the library, where the plug-in is located.

For example, this is a Plugin file:

```
a148e300-5740-11d1-a904-080036aaa103.Location: FWOSPlugin
! base document drivers plugin
ad696000-5b34-11d1-b5ba-00a0c9064368.Location: PAppStdPlugin
ad696001-5b34-11d1-b5ba-00a0c9064368.Location: PAppStdPlugin
ad696002-5b34-11d1-b5ba-00a0c9064368.Location: PAppStdPlugin
47b0b826-d931-11d1-b5da-00a0c9064368.Location: PAppStdPlugin
47b0b827-d931-11d1-b5da-00a0c9064368.Location: PAppStdPlugin
```

2   A Look Inside OCAF

2.1   The Design of OCAF

2.1.1   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.
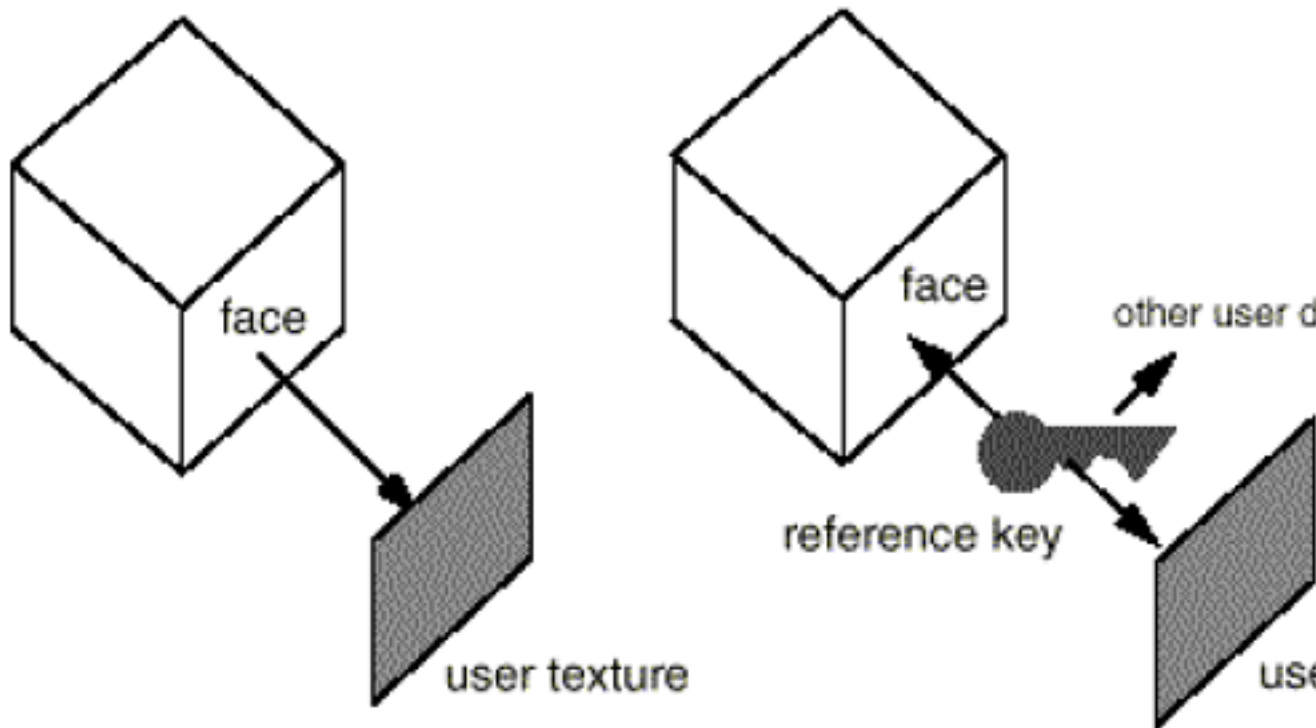

Figure 2: Figure 2. Topology driven versus reference-key driven approaches

2.1.2   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, 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 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, 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.

2.1.3   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

2.1.4   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.

2.2   The Data Framework

2.2.1   Data structure

The OCAF Data Framework is the Open CASCADE Technology 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.

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.
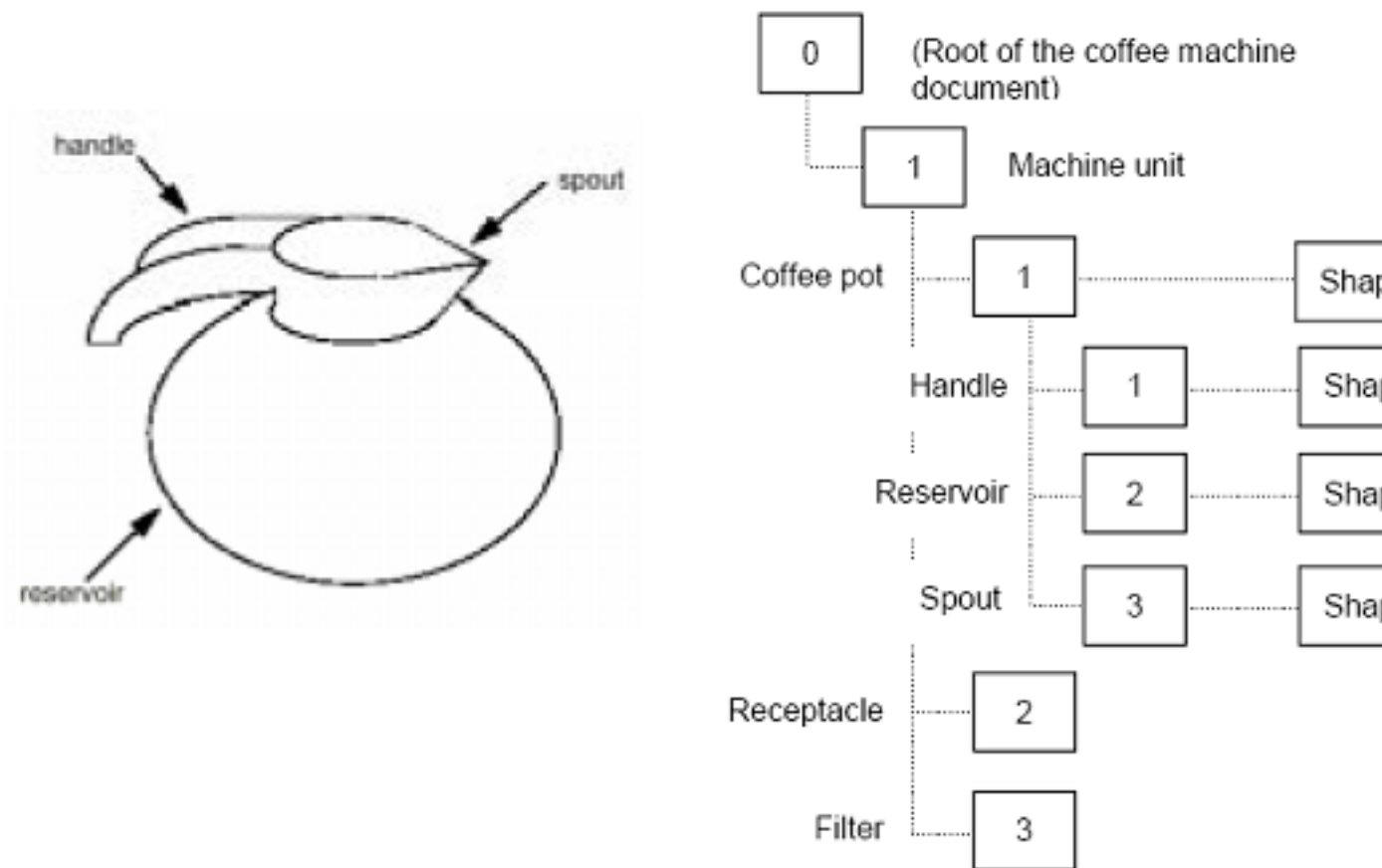
Figure 3: Figure 3.The data structure of the coffee machine

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] whereas the handle of the second pot would be [0:1:4:1]. This way, we avoid any confusion between coffee pot handles.

2.2.2   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]).
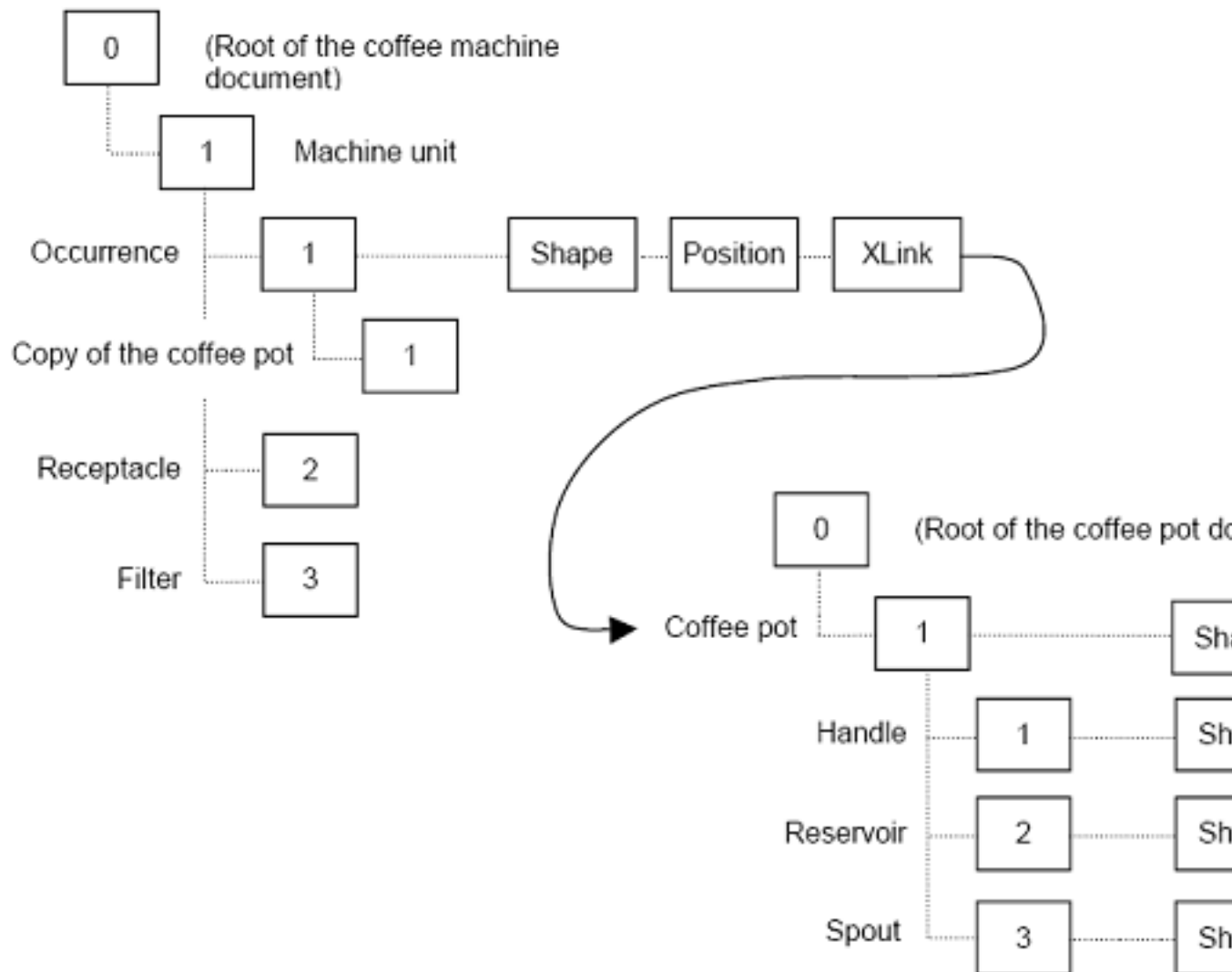
Figure 4: The coffee machine compound document

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).

2.2.3 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.

2.3   Persistent Data Storage

2.3.1   Introduction

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.

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.

2.3.2   Schemes of Persistence

There are three schemes of persistence, which you can use to store and retrieve OCAF data (documents):

   • *Standard* persistence schema, compatible with previous OCAF applications

   • *XmlOcaf* persistence, allowing the storage of all OCAF data in XML form

   • *BinOcaf* persistence, allowing the storage of all OCAF data in binary format form

All schemes are independent of each other, but they guarantee that the standard OCAF attributes stored and retrieved by one schema will be storable and retrievable by the other. Therefore in any OCAF application you can use any persistence schema or even all three of them. The choice is made depending on the *Format* string of stored OCAF documents or automatically by the file header data - on retrieval.

Persistent data storage in OCAF using the *Standard* package is presented in:

   • Basic Data Storage

   • Persistent Collections

Persistent storage of shapes is presented in the following chapters:

   • Persistent Geometry

   • Persistent Topology

Finally, information about opening and saving persistent data is presented in Standard Documents.

2.3.3   Basic Data Storage

Normally, all data structures provided by Open CASCADE Technology are run-time structures, in other words, transient data. As transient data, they exist only while an application is running and are not stored permanently. However, the Data Storage module provides resources, which enable an application to store data on disk as persistent data.

Data storage services also provide libraries of persistent classes and translation functions needed to translate data from transient to persistent state and vice-versa.

Libraries of persistent classes

Libraries of persistent classes are extensible libraries of elementary classes you use to define the database schema of your application. They include:

   • Unicode (8-bit or 16-bit character type) strings

   • Collections of any kind of persistent data such as arrays.

All persistent classes are derived from the **Persistent** base class, which defines a unique way of creating and handling persistent objects. You create new persistent classes by inheriting from this base class.

Translation Functions

Translation functions allow you to convert persistent objects to transient ones and vice-versa. These translation functions are used to build Storage and Retrieval drivers of an application.

For each class of 2D and 3D geometric types, and for the general shape class in the topological data structure library, there are corresponding persistent class libraries, which allow you to translate your data with ease.

Creation of Persistent Classes

If you use Unix platforms as well as WOK and CDL, you can create your own persistent classes. In this case, data storage is achieved by implementing *Storage* and *Retrieval* drivers.

The *Storage* package is used to write and read persistent objects. These objects are read and written by a retrieval or storage algorithm (*Storage_Schema* object) in a container (disk, memory, network ...). Drivers (*FSD_File* objects) assign a physical container for data to be stored or retrieved.

The standard procedure for an application in reading a container is as follows:

- open the driver in reading mode,
- call the Read function from the schema, setting the driver as a parameter. This function returns an instance of the *Storage_Data* class which contains the data being read,
- close the driver.

The standard procedure for an application in writing a container is as follows:

- open the driver in writing mode,
- create an instance of the *Storage_Data* class, then add the persistent data to write with the function *AddRoot*,
- call the function *Write* from the schema, setting the driver and the *Storage_Data* instance as parameters,
- close the driver.

2.3.4 Persistent Collections

Persistent collections are classes which handle dynamically sized collections of data that can be stored in the database. These collections provide three categories of service:

- persistent strings,
- generic arrays of data,
- commonly used instantiations of arrays.

Persistent strings are concrete classes that handle sequences of characters based on both ASCII (normal 8-bit) and Unicode (16-bit) character sets.

Arrays are generic classes, that is, they can hold a variety of objects not necessarily inheriting from a unique root class. These arrays can be instantiated with any kind of storable or persistent object, and then inserted into the persistent data model of a user application.

The purpose of these data collections is simply to convert transient data into its persistent equivalent so that it can be stored in the database. To this end, the collections are used to create the persistent data model and assure the link with the database. They do not provide editing or query capabilities because it is more efficient, within the operative data model of the application, to work with transient data structures (from the *TCollection* package).

For this reason:

- the persistent strings only provide constructors and functions to convert between transient and persistent strings, and

- the persistent data collections are limited to arrays. In other words, *PCollection* does not include sequences, lists, and so on (unlike *TCollection*).

Persistent string and array classes are found in the *PCollection* package. In addition, *PColStd* package provides standard, and frequently used, instantiations of persistent arrays, for very simple objects.

2.3.5   Persistent Geometry

The Persistent Geometry component describes geometric data structures which can be stored in the database. These packages provide a way to convert data from the transient "world" to the persistent "world".

Persistent Geometry consists of a set of atomic data models parallel to the geometric data structures described in the geometry packages. Geometric data models, independent of each other, can appear within the data model of any application. The system provides the means to convert each atomic transient data model into a persistent one, but it does not provide a way for these data models to share data.

Consequently, you can create a data model using these components, store data in, and retrieve it from a file or a database, using the geometric components provided in the transient and persistent "worlds". In other words, you customize the system by declaring your own objects, and the conversion of the geometric components from persistent to transient and vice versa is automatically managed for you by the system.

However, these simple objects cannot be shared within a more complex data model. To allow data to be shared, you must provide additional tools.

Persistent Geometry is provided by several packages.

The *PGeom* package describes geometric persistent objects in 3D space, such as points, vectors, positioning systems, curves and surfaces.

These objects are persistent versions of those provided by the *Geom* package: for each type of transient object provided by Geom there is a corresponding type of persistent object in the *PGeom* package. In particular the inheritance structure is parallel.

However the *PGeom* package does not provide any functions to construct, edit or access the persistent objects. Instead the objects are manipulated as follows:

- Persistent objects are constructed by converting the equivalent transient *Geom* objects. To do this you use the *MgtGeom::Translate* function.

- Persistent objects created in this way are used to build persistent data structures that are then stored in a file or database.

- When these objects are retrieved from the file or database, they are converted back into the corresponding transient objects from the Geom package. To do this, you use *MgtGeom::Translate* function.

In other words, you always edit or query transient data structures within the transient data model supplied by the session. Consequently, the documentation for the *PGeom* package consists simply of a list of available objects.

The *PGeom2d* package describes persistent geometric objects in 2D space, such as points, vectors, positioning systems and curves. This package provides the same type of services as the *PGeom* package, but for the 2-D geometric objects provided by the *Geom2d* package. Conversions are provided by the *MgtGeom::Translate* function.

```
//Create a coordinate system
Handle(Geom_Axis2Placement) aSys;


//Create a persistent coordinate PTopoDS_HShape.cdlsystem
Handle(PGeom_Axis2placement)
    aPSys = MgtGeom::Translate(aSys);

//Restore a transient coordinate system
Handle(PGeom_Axis2Placement) aPSys;

Handle(Geom_Axis2Placement)
    aSys = MgtGeom::Translate(aPSys);
```

2.3.6   Persistent Topology

The Persistent Topology component describes topological data structures which can be stored in the database. These packages provide a way to convert data from the transient "world" to the persistent "world".

Persistent Topology is based on the BRep concrete data model provided by the topology packages. Unlike the components of the Persistent Geometry package, topological components can be fully shared within a single model, as well as between several models.

Each topological component is considered to be a shape: a *TopoDS_Shape* object. The system's capacity to convert a transient shape into a persistent shape and vice-versa applies to all objects, irrespective of their complexity: vertex, edge, wire, face, shell, solid, and so on.

When a user creates a data model using BRep shapes, he uses the conversion functions that the system provides to store the data in, and retrieve it from the database. The data can also be shared.

Persistent Topology is provided by several packages.

The *PTopoDS* package describes the persistent data model associated with any BRep shape; it is the persistent version of any shape of type *TopoDS_Shape*. As is the case for persistent geometric models, this data structure is never edited or queried, it is simply stored in or retrieved from the database. It is created or converted by the *MgtBRep::Translate* function.

The *MgtBRepAbs* and *PTColStd* packages provide tools used by the conversion functions of topological objects.

```
//Create a shape
TopoDS_Shape aShape;

//Create a persistent shape
PtColStd_DoubleTransientPersistentMap aMap;

Handle(PTopoDS_HShape) aPShape =
    aMap.Bind2(MgtBRep::Translate
        aShape,aMap,MgtBRepAbs_WithTriangle));

aPShape.Nullify();

//Restore a transient shape
Handle(PTopoDS_HShape) aPShape;

Handle(TopoDS_HShape) aShape =
    aMap.Bind1(MgtBRep::Translate
        (aPShape,aMap,MgtBRepAbs_WithTriangle));

aShape.Nullify();
```

2.3.7   Standard Documents

Standard documents offer you a ready-to-use document containing a TDF-based data structure. The documents themselves are contained in a class inheriting from *TDocStd_Application* which manages 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 undo/redo functionality.