Object Architecture Overview

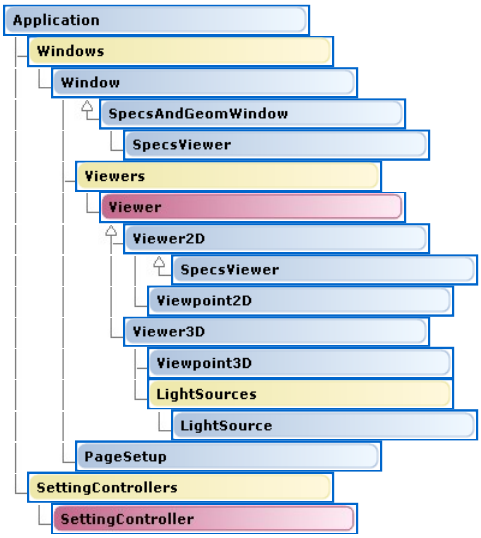
You will find in this section how the Automation objects are described, and get information about:

- Object Diagrams, Object Descriptions and Object References shows you where and how objects are described
- About Objects, Collections, Properties, and Methods helps you understand the basics of the object model required by scripting
- About Inheritance and Aggregation describes the two object model mechanisms you need to know to write macros.

Object Diagrams, Object Descriptions and Object References

To describe objects, their properties and methods, and the relationships between objects, we use object diagrams and object descriptions.

Object diagrams describe the overall structure and how objects are linked together:



Objects are depicted using colored boxes, linked together using symbols. You can learn from the diagram above that the **Application** object aggregates a **Windows** collection object because the boxes that represent them in the diagram are linked using the aggregation symbol:

Objects are depicted using a blue background color, collections objects using a yellow one, and abstract objects using a purple one.

The **Windows** collection object aggregates any number of **Window** objects. The **Window** object aggregates in turn a **Viewers** collection object and a **PageSetup** object.

The **Viewers** collection aggregates **Viewer** objects. The **Viewer** objects is an abstract object, depicted using a purple background color, and only its derived objects can actually be created, that is:

- The **Viewer2D** object
- The **Viewer3D** object
- The **SpecsViewer** object.

Abstract objects and their derived objects are linked using the inheritance symbol:

The ▶ symbol indicates that the object to which it refers, that is the object located just left of the symbol, is the root of an object structure expanded in another object diagram. The symbol ▲ shows that the root object near it can be found in one or several object diagrams as an object to expand.

For main objects, the object descriptions describe the object and how to use it. Here is the *Application Object* description [1].

For all objects, the object references contain complete reference of inheritance, properties and methods, thus encompassing all other link types between objects. Here is the *Application (Object)* reference [2].

About Objects, Collections, Properties, and Methods

Scripting languages such as Visual Basic rely on objects. Most pieces of data you can access are objects. Editors, windows, viewers, parts, sketches, pads, even lines and curves, are represented as objects in Visual Basic.

An *object* is depicted using a blue box in the object diagrams, such as:

A *collection* is an object that contains other objects. Like with stamps collecting, where all objects in the collection are stamps, a collection contains usually objects of the same type. For example a window collection contains windows. A collection is always denoted as a plural name to easily help recognize a collection among other objects. The window collection is thus named **Windows**. A collection is depicted using a yellow box in the object diagrams, such as

The collection index begins at 1, and not 0. Usually, an object in the collection is reached using its index, but it can also be reached using the name you assign to it. This ability of referring to an object using its name in the collection is stated in the documentation of each collection object, such as **Windows**, when the index type is Variant, for example in the **Item** method.

A *property* is part of an object and helps to characterize it. For example, the **Window** object has the **ActiveViewer** property, that returns the viewer that is currently used in the window, and the **PageSetup** property that contains all the parameters required to print the window. Retrieving property values of an object makes it possible to distinguish it among other objects of the same type. Modifying property values of an object changes the object characteristics and implies that the object itself is thus modified. To prevent from non-desired property modifications, an object can have read-only properties from which you can retrieve their values, but never set them. This is the case of the **ActiveViewer** property of the **Window** object. On the opposite, the **PageSetup** property is a read/write property.

To retrieve or set the value of a property of a given object, write the object reference followed by a period and the property name. For example, you can retrieve in the **oPgSetup** variable the value of the **PageSetup** value of the active window as follows:

VB VBA C# VB.NET Python

```
Dim oPgSetup
oPgSetup = CATIA.ActiveWindow.ActiveViewer.PageSetup

Dim oPgSetup As PageSetup
oPgSetup = CATIA.ActiveWindow.ActiveViewer.PageSetup

private INFITF.PageSetup oPgSetup
...
private void UsePageSetup() {
  oPgSetup = CATIA.ActiveWindow.ActiveViewer.PageSetup

Dim oPgSetup As INFITF.PageSetup
oPgSetup = CATIA.ActiveWindow.ActiveViewer.PageSetup

oPgSetup = CATIA.ActiveWindow.ActiveViewer.PageSetup
```

Or you can set this property as follows:

VB VBA VB.NET Python

```
Dim opgSetup
oPgSetup.Banner = "This is my print banner"
oPgSetup.BannerPosition = catBannerPositionTop
... ' go on setting you own printing parameters
CATIA.ActiveWindow.ActiveViewer.PageSetup = oPgSetup

Dim opgSetup As PageSetup
oPgSetup.Banner = "This is my print banner"
oPgSetup.BannerPosition = catBannerPositionTop
... ' go on setting you own printing parameters
CATIA.ActiveWindow.ActiveViewer.PageSetup = oPgSetup

Dim opgSetup As INFITF.PageSetup
opgSetup.Banner = "This is my print banner"
opgSetup.BannerPosition = catBannerPositionTop
... ' go on setting you own printing parameters
CATIA.ActiveWindow.ActiveViewer.PageSetup = opgSetup

oPgSetup.Banner = "This is my print banner"
oPgSetup.BannerPosition = catBannerPositionTop
...' go on setting you own printing parameters
CATIA.ActiveWindow.ActiveViewer.PageSetup = oPgSetup
```

Note that the keywords, such as `Dim` and `As`, are colored in blue, and that the comment, starting with the single quote character, is colored in green, as in the VBA or VSTA editors.

The root object for all macros is the **Application** object. This corresponds to the entire application including its frame window. The application is always named **CATIA** for in-process access, and you should only refer to it since it already exists when you run an in-process macro.

An object reference always starts from the root object, that is the **Application** object which is always set to **CATIA** with in-process access. Then you use the **Application** object's properties to access the objects. In this case, the application object has the **ActiveWindow** property that holds the active window. You simply need to write **CATIA.ActiveWindow** to refer to the active window. Then the **Window** object has the **ActiveViewer** property to hold its active viewer. Simply add a period followed by **ActiveViewer**, and you get this active viewer.

In the same way, you can request to maximize the active window by setting the value of its **WindowState** property to **catWindowStateMaximized**, as follows:

```
CATIA.ActiveWindow.WindowState = catWindowStateMaximized
```

A *method* is an action that you can request an object to do. For example, you can request the active window to close. To do this, the **Window** object includes the **Close** method. Simply request the window to close itself as follows:

```
CATIA.ActiveWindow.Close
```

Methods have often arguments requested by the action. For example, the **PutDirection** method of the **LightSource** object carries out the action of setting the lighting direction of a light source. This lighting direction must be provided to perform this action, otherwise the **PutDirection** method cannot work. This direction is passed as an argument of the **PutDirection** method. To set the lighting direction of the first light source in the collection of light sources attached to the active viewer parallel to the X vector of the main coordinate system, proceed as follows:

VB VBA C# VB.NET Python

```
Dim oLiSource
Set oLiSource = CATIA.ActiveViewer.LightSources.Item(1)
oLiSource.PutDirection Array(1, 0, 0)

Dim oLiSource As LightSource
Set oLiSource = CATIA.ActiveViewer.LightSources.Item(1)
oLiSource.PutDirection Array(1, 0, 0)

private INFITF.PageSetup oLiSource
...
private void SetLightSource() {
  oLiSource = CATIA.ActiveViewer.LightSources.Item(1)
  oLiSource.PutDirection Array(1, 0, 0)

Dim oLiSource As INFITF.LightSource
```

```
Set oLiSource = CATIA.ActiveViewer.LightSources.Item(1)
oLiSource.PutDirection Array(1, 0, 0)

oLiSource = CATIA.ActiveViewer.LightSources.Item(1)
oLiSource.PutDirection Array(1, 0, 0)
```

The first item of the light source collection is returned using the **Item** method of the **LightSources** collection object. Then the **PutDirection** method of the **LightSource** object is called. The argument passed to this method is an array of three integers (1, 0, 0) to designate the X vector of the main coordinate system.

You will find in the **object reference** that each method is qualified as either a **Function** or a **Sub**, and with parentheses, even if they have no arguments, as follows:

```
Function NewWindow() As Window

Sub Update()
```

This is a Visual Basic notation to distinguish a method that returns a value, or **Function**, from one that does not, or **Sub**. Note that the returned value of a **Function** is indicated using the **As** keyword. In the example above, the **NewWindow** method is a **Function** because it returns a **Window** object. See also *Some Tips about Subs and Functions* to know more about **Sub**s and **Function**s [3].

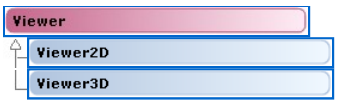
About Inheritance, Aggregation, and Object Model

Inheritance and aggregation are the two main kinds of relationships between objects.

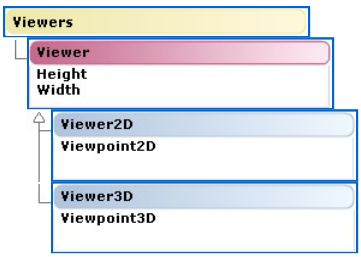
Inheritance is the ability of an object to be a specialization of another object. An object that inherits from another object, named the *base* object, inherits the properties and the methods of this base object and adds them to its own properties and methods. Inheritance helps to specialize objects while gathering common properties and methods in the base object. Inheritance is a iterative process, since an object can inherit from an object which itself inherits from another object which itself inherits, and so forth, and the lowest object in the inheritance tree inherits the properties and methods of all the objects above it. Inheritance is depicted using the following symbol  in the object diagrams.

Some objects are depicted with a purple box, such as:

These are abstract objects. An *abstract object* owns properties and methods like any other object, but it cannot be created. Only objects which inherits from it and which are not abstract objects can be created.



For example, the **Viewer** object is an abstract object which owns properties and methods shared by all viewers, whatever their type 2D or 3D, but cannot be created as such. Only objects of types **Viewer2D** and **Viewer3D** which inherit from it can actually be created.



In the above diagram, the **Viewer2D** object is a specialized viewer that is dedicated to 2D, and which inherits the properties and methods of the **Viewer** object. In addition, the **Viewer2D** object has its own properties and methods. For example, the **Viewer2D** object inherits the **Height** and **Width** properties from the **Viewer** object, like the **Viewer3D** object, and owns the **Viewpoint2D** property, which is a specific property for a 2D viewer, but which does not make sense for the **Viewer** base object the properties and methods of which must also match the **Viewer3D** object needs.

Assume that **My2DViewer** is a **Viewer2D** object and is the active viewer or the active window. The example below assigns a new view point, **My2DViewpoint**, to this viewer.

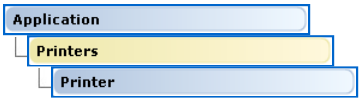
VB VBA C# VB.NET Python

```
Dim o2DViewpoint
o2DViewpoint.PutOrigin Array(0, 0)
o2DViewpoint.Zoom = 1

Dim o2DViewer
o2DViewer = CATIA.ActiveWindow.ActiveViewer
o2DViewer.Viewpoint2D = My2DViewpoint
o2DViewer.Height = 300
o2DViewer.Width = 500

Dim o2DViewpoint As Viewpoint2D
o2DViewpoint.PutOrigin Array(0, 0)
o2DViewpoint.Zoom = 1

Dim o2DViewer As Viewer2D
o2DViewer = CATIA.ActiveWindow.ActiveViewer
o2DViewer.Viewpoint2D = My2DViewpoint
o2DViewer.Height = 300
o2DViewer.Width = 500

private INFITF.Viewpoint2D o2DViewpoint
...
private void SetViewPoint2D() {
  o2DViewpoint.PutOrigin Array(0, 0)
  o2DViewpoint.Zoom = 1
...
```

```
private INFITF.Viewer2D o2DViewer
o2DViewer = CATIA.ActiveWindow.ActiveViewer
o2DViewer.Viewpoint2D = My2DViewpoint
o2DViewer.Height = 300
o2DViewer.Width = 500

Dim o2DViewpoint As INFITF.Viewpoint2D
o2DViewpoint.PutOrigin Array(0, 0)
o2DViewpoint.Zoom = 1

Dim o2DViewer As INFITF.Viewer2D
o2DViewer = CATIA.ActiveWindow.ActiveViewer
o2DViewer.Viewpoint2D = My2DViewpoint
o2DViewer.Height = 300
o2DViewer.Width = 500

o2DViewpoint.PutOrigin Array(0, 0)
o2DViewpoint.Zoom = 1

o2DViewer = CATIA.ActiveWindow.ActiveViewer
o2DViewer.Viewpoint2D = My2DViewpoint
o2DViewer.Height = 300
o2DViewer.Width = 500
```

In this example, a **Viewpoint2D** object is created, initialized with an origin and a zoom factor, and set as the **Viewer2D** object view point thanks to the **Viewpoint2D** property dedicated to **Viewer2D** objects. Then the height and width, expressed in pixels, of the **Viewer2D** object are set thanks to the **Height** and **Width** properties inherited from the **Viewer** base object.

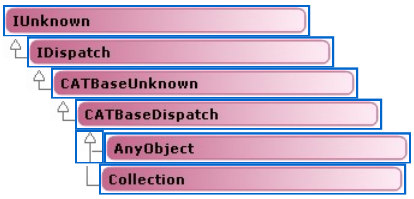
Aggregation is the ability of an object to contain another one. For example, the **Application** object contains (aggregates) a **Printers** collection object. Within CAA Automation objects models, the aggregation of a series of objects of the same type is usually reserved to collection objects. For example, the **Printers** collection contains (aggregates) a series of **Printer** objects. Aggregation is depicted in object diagrams using the following symbol └ between the aggregating object and the aggregated object.



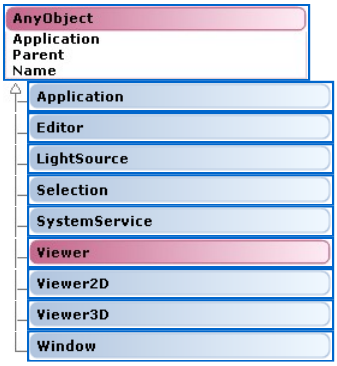
You may find a *role* description on this symbol. For example, the **Prism** object aggregates two **Limit** objects: its first limit and its second limit.

This role is usually set as the property that returns and possibly set s the object. For example, the **FirstLimit** property of the **Prism** object returns the prism first limit. The aggregation link is usually bi-directional because the **Parent** property of each object allows you to retrieve its aggregating object. For clarification purpose, mono-directional relationships may be visible on a diagram. They are depicted using the following symbol ↳. Here, for example, the **Sketch** object is aggregated by a **Sketches** collection object that is not visible but is referred to by the **SketchBasedShape** object.

In addition to this functional view, we can superimpose another view to help understand some basic mechanisms:



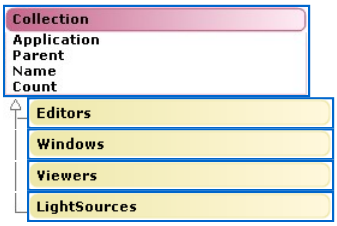
Any objects derive from the **AnyObject** abstract object, and any collections derive from the **Collection** abstract object. Both of them derive from the **CATBaseDispatch** abstract object, which derives from the **CATBaseUnknown** abstract object, which in turn derive from the **IDispatch** abstract object that derives from the root **IUnknown** abstract object. You will never deal with these last four abstract objects that enable the Automation object model to a COM object model. But the **AnyObject** and the **Collection** abstract object bring important properties to their child objects.



Any objects derive from the **AnyObject** abstract object which supplies the **Application**, **Parent**, and **Name** properties. You can then use these properties against any object, since any object features them:

- The **Application** property returns the application to which an object belongs. When running in-process macros, there is only one application named **CATIA**

- The **Name** property allows any object to be assigned a name
- The **Parent** property allows the parent object to be retrieved, the parent object being the object which aggregates the current one.



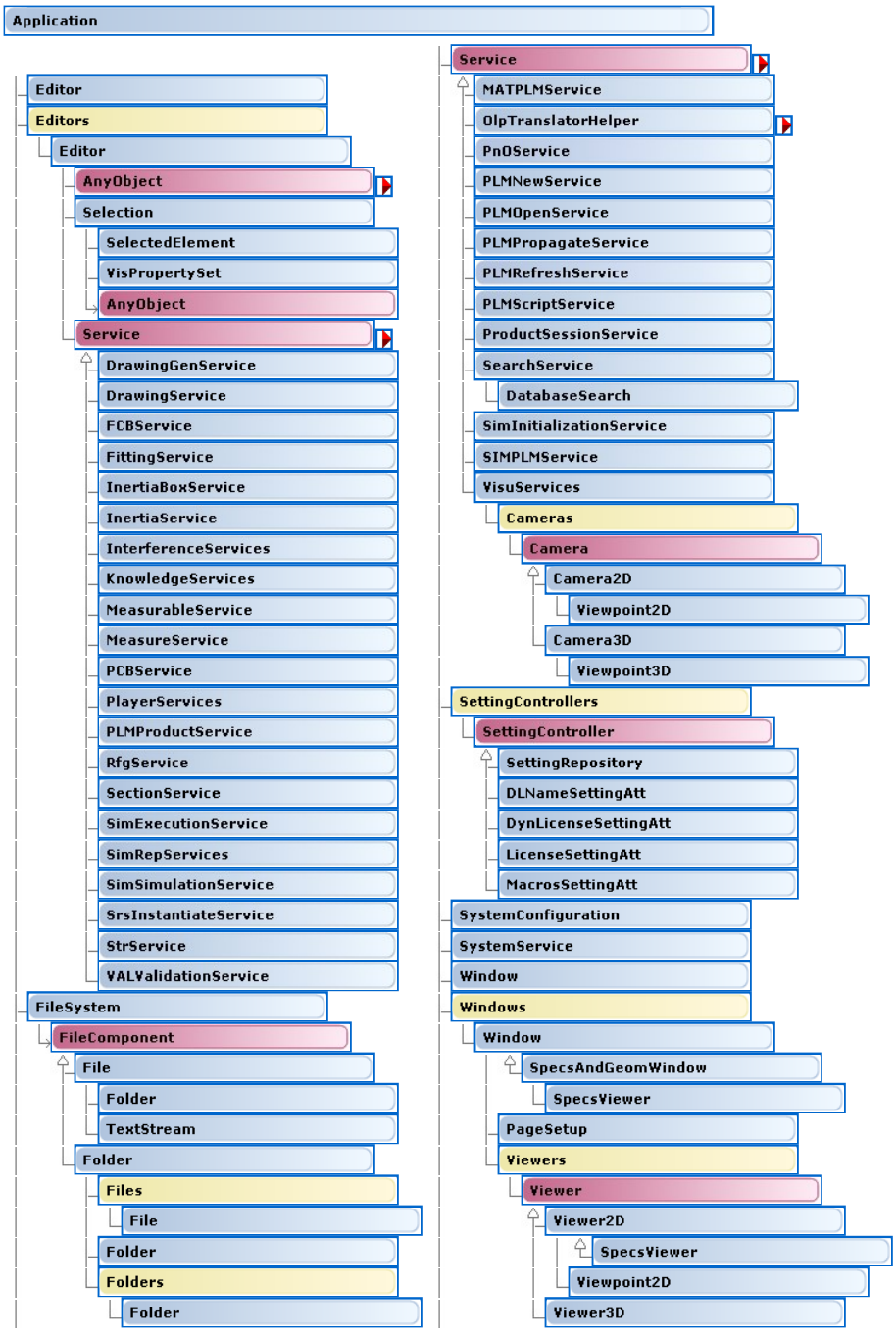
Any collections derive from the **Collection** abstract object which supplies the **Count** property in addition to the **Application**, **Parent**, and **Name** properties. The **Count** property of the **Collection** object returns the number of items in the collection. In addition, a **Collection** object can supply an **Item**, an **Add**, and a **Remove** methods to return an object from, add an object in, or remove an object from the collection.

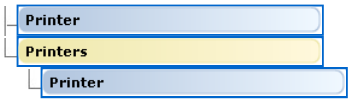
References

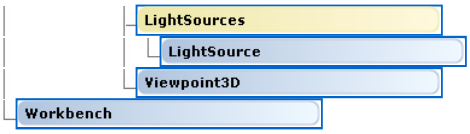
[1] Application Object Description
[2] Application (Object) Reference
[3] About Subs and Functions

Foundation Object Model Map

See Also Legend

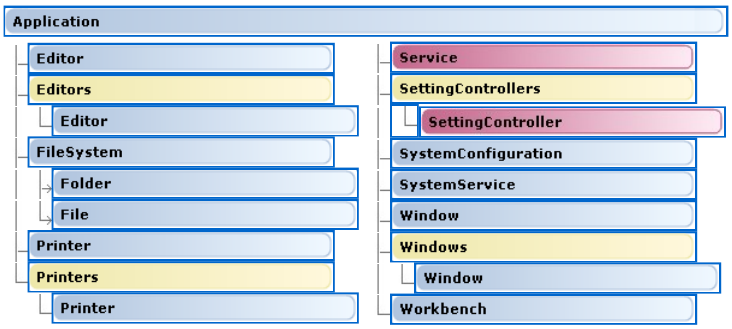






Application Object

See Also Legend Use Cases Properties Methods



Represents the entire current application and its frame window.

The **Application** object is the root object for all the other objects you can use and access from scripts. It corresponds to both the application itself and to its frame window. It directly aggregates four collections:

1. The editor collection represented by the **Editors** collection object.
   This collection contains all the editors currently managed by the application. Use the **Editors** property to return the **Editors** collection object.
2. The printer collection represented by the **Printers** collection object.
   This collection contains all the printers accessible from the application. Use the **Printers** property to return the **Printers** collection object.
3. The setting controller collection represented by the **SettingControllers** collection object.
   This collection contains a generic setting controller for each setting repository managed using an XML file, and some specific setting controllers. Use the **SettingControllers** property to return the **SettingControllers** collection object.
4. The window collection represented by the **Windows** collection object.
   This collection contains all the windows currently opened by the application, each window displaying objects managed by an editor. Use the **Windows** property to return the **Windows** collection object.

In addition to these four collections, the **Application** object also aggregates:

- An **Editor** object: at any time, an **Editor** object among those managed by the application is active and associated with the objects displayed in the active window, that is, the window the end-user is currently working in. This active editor gives access to the objects visible in the active window and to the set of operations that can be applied to those objects. This editor sets the available menus and toolbars that make it possible to edit the objects it controls, according to the root object type. Use the **ActiveEditor** property to return the active editor.
- A **FileSystem** object that enables you to manipulate folders and files. Use the **FileSystem** property to return the file system object
- A **Printer** object: the active printer. Use the **ActivePrinter** property to return the active printer
- A **Service** object: object-independent session-level **Service** objects can be retrieved from the **Application** object thanks to the **GetSessionService** method. A **Service** object has methods that can execute whatever the objects managed by the active editor. For example, creating and running a search in the database does not depend on the objects managed by the active editor in the active window. It is made possible using a method of the session-level **SearchService** object.
- A **SystemConfiguration** object, providing access to system or configuration dependent resources, such as the operating system, the current version, release, and service pack, and the available licensed products. Use the **SystemConfiguration** property to return the **SystemConfiguration** object
- A **SystemService** object, providing information about the system environment, and services to run a script or a process. For example, the **Environ** method retrieves the value of a given environment variable. Note that the **SystemService** object is not a **Service** object, that is, is retrieved thanks to the **SystemService** property of the **Application** object, while a **Service** object is retrieved using the **GetSessionService** method.
- A **Window** object: the active window. Use the **ActiveWindow** property to return the active window.
- A **Workbench** object that you can retrieve thanks to its workbench identifier using the **GetWorkbenchId** method.

The **Application** object has other properties, such as **LocalCache** to return the local cache path, and **CacheSize** to return the local cache size. As the application represents the frame window, you can retrieve or set the frame dimensions and location using the properties **Width**, **Height**, **Left**, and **Top** respectively, with values expressed in screen pixels.

Remarks

When you create or use macros for in-process access, always refer to the **Application** object as **CATIA**. For example, to retrieve the **Windows** collection from the **Application** object, refer to the **Application** object as follows:

```
Dim cWins As Windows
Set cWins = CATIA.Windows
```

Using the Application Object

Use the **ActiveEditor** property to retrieve the active editor

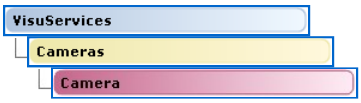
```
Dim oActEditor As Editor
Set oActEditor = CATIA.ActiveEditor
```

Use the **GetSessionService** method to return a **Service** object. This example returns the **VisuServices** object.

```
Dim oService As Service
Set oService = CATIA.GetSessionService("VisuServices")
```

Cameras Collection Object

See Also Legend Use Cases Properties Methods



Represents a collection of cameras.

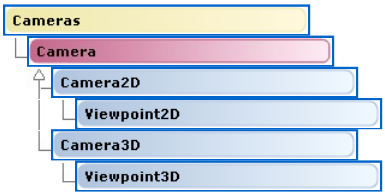
Returning the Cameras Collection Object

Use the **Cameras** property of the **VisuServices** object to return the **Cameras** collection object.

```
Dim cCameras As Cameras
Set cCameras = oVisuServices.Cameras
```

Camera Object

See Also Legend Use Cases Properties Methods



Represents a camera.

A camera is the persistent form of a viewpoint.

You can create a camera from the current viewpoint using the **NewCamera** method of the **Viewer** object.

The camera name is assigned by CATIA: **Camera00**, **Camera01**, and so forth, and the created camera is placed at the end of the camera collection and is displayed in the Named Views dialog box of the View menu.

You can rename the cameras you create with your own names using the **Name** property.

A created camera can then be assigned to a viewer to make its own viewpoint change to this of the camera.

Two kinds of cameras exist: the **Camera2D** object for 2D viewpoints, that is for Drawing Representation objects, and the **Camera3D** object for 3D viewpoints representing the real world, that is for 3D Shape Representation and Product objects.

A **Camera2D** object stores a **Viewpoint2D** object and a **Camera3D** object stores a **Viewpoint3D** object.

Creating a Camera

Use the **NewCamera** method of the **Viewer** object to create a **Camera** object.

```
Dim oViewer As Viewer
Set oViewer = CATIA.ActiveWindows.ActiveViewer

Dim oCamera As Camera
Set oCamera = oViewer.NewCamera()
```

The created camera is added to the camera collection.

Note : Created Camera for 3DPart scenario is no more located under the 3DShape.

Retrieving an Existing Camera

Use the **VisuServices** object to retrieve a **Cameras** collection object.

```
Dim oVisuServices As VisuServices
Set oVisuServices = CATIA.GetSessionService("VisuServices")

Dim cCameras As Cameras
Set cCameras = oVisuServices.Cameras
```

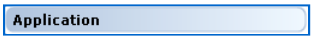
From this collection, you can either retrieve a built-in camera representing a standard view, or a camera you have created.

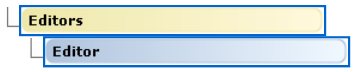
```
Dim oCamera As Camera
Set oCamera = cCameras.Item("Camera05")
```

This retrieves the front camera.

Editors Collection Object

See Also Legend Use Cases Properties Methods





A collection of the **Editor** objects that are currently held by the application.

Retrieving the Editors Collection Object

Use the **Editors** property of the **Application** object to return the **Editors** collection object.

```
Dim cEditors As Editors
Set cEditors = CATIA.Editors
```

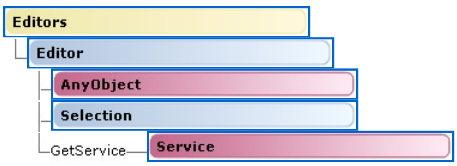
Using the Editors Collection Object

Use the **Item** method of the **Editors** collection object to return an editor among the ones hold by the **Application** object.

```
Dim oEditor As Editor
Set oEditor = CATIA.Editors.Item(2)
```

Editor Object

See Also Legend Use Cases Properties Methods



Represents an editor.

In the Model View Controller paradigm, the editor plays the Controller role. The editor federates all the objects that can be interactively edited in the same window. It holds the current workbench and thus maintains the list of all the commands that can be launched from menus and toolbars when this window is active to edit the objects it controls according to the root object PLM type.

The active editor is the editor associated with the current window, whatever the active object in that window.

Retrieving the Editor Object

Use the **ActiveEditor** property of the **Application** object to return the active editor.

```
Dim oActiveEditor As Editor
Set oActiveEditor = CATIA.ActiveEditor
```

Use the **Item** method of the **Editors** collection to return an editor among the ones hold by the **Application** object.

```
Dim oEditor As Editor
Set oEditor = CATIA.Editors.Item(2)
```

Using the Editor Object

Use the **ActiveObject** property to return the active object associated with the active **Editor** object. The active object is the root of the data being currently edited. Depending on this data, it can be a **Part** object, a **DrawingRoot** object, or a **VPMReference** object.

```
Dim oActiveObject As AnyObject
Set oActiveObject = CATIA.ActiveEditor.ActiveObject
```

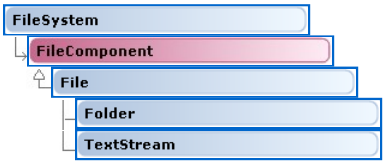
Use the **Selection** property to return the **Selection** object associated with the active **Editor** object.

```
Dim oSelection As Selection
Set oSelection = CATIA.ActiveEditor.Selection
```

Use the **GetService** method to return a **Service** object applying to the Editor. Refer to the Service Object.

File Object

See Also Legend Use Cases Properties Methods



Represents a file.

The **File** object is operating system independent. Using it instead of those available with the platform you use makes your macro portable.

The **File** object derives from the **FileComponent** object from which it inherits the **Path** property. It aggregates:

- A **Folder** object: its parent folder accessible using the **ParentFolder** property also inherited from the **FileComponent** object
- A **TextStream** object that represents the file contents as a text stream, returned thanks to the **OpenAsTextStream** method.

Retrieving the File Object

Use the **GetFile** method of the **FileSystem** object, returned thanks to the **FileSystem** property of the **Application** object, to return an existing file.

```
Dim oFile As File
Set oFile = CATIA.FileSystem.GetFile("C:\tmp\myFile.txt")
```

In the same way, use the other properties and methods of the **FileSystem** object to create, copy, delete, and check for the existence of a file.

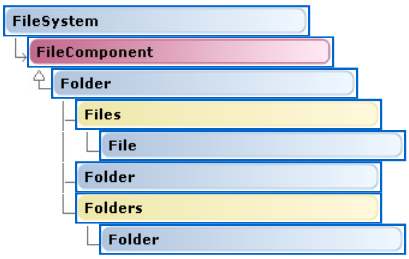
Using the File Object

Use the **Size** property to return the size of a file. The size is expressed in bytes.

```
Dim oSize As Long
Set oSize = oFile.Size
```

Folder Object

See Also Legend Use Cases Properties Methods



Represents a folder.

The **Folder** object derives from the **FileComponent** object from which it inherits the **Path** property. It aggregates:

- A **Files** collection object: it contains its files, returned thanks to the Files property.
- A **Folder** object: its parent folder accessible using the **ParentFolder** property also inherited from the **FileComponent** object.
- A **Folders** collection object: it contains its subfolders, returned thanks to the **SubFolders** property.

Retrieving the Folder Object

Use the **GetFolder** method of the **FileSystem** object, returned thanks to the **FileSystem** property of the **Application** object, to return an existing folder.

```
Dim oFolder As Folder
Set oFolder = CATIA.FileSystem.GetFolder("C:\tmp")
```

In the same way, use the other properties and methods of the **FileSystem** object to create, copy, delete, and check for the existence of a folder.

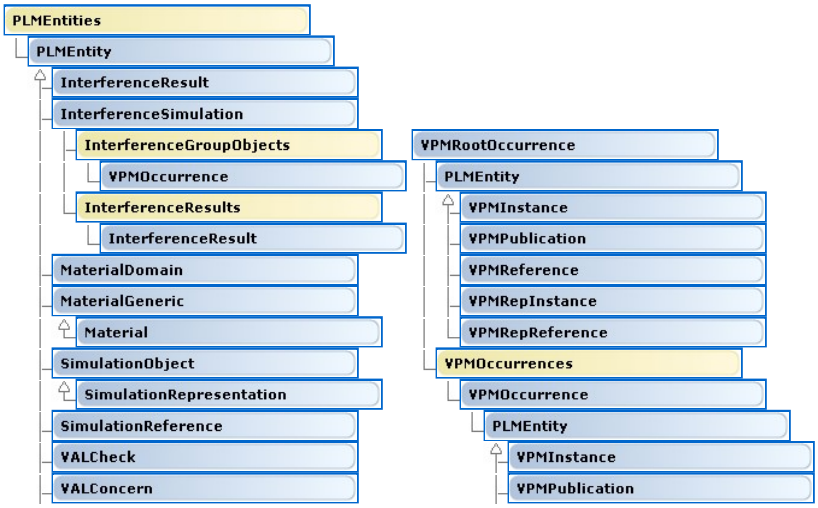
Using the Folder Object

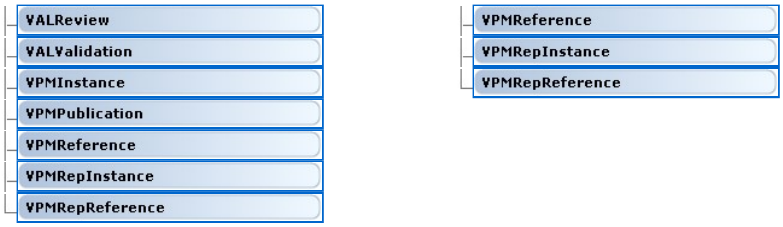
Use the **Files** property to return the **Files** collection object containing the files in the folder.

```
Dim cFiles As Files
Set cFiles = oFolder.Files
For Each oFile In cFiles
  msgbox "File path: " & oFile.Path & _
         "File type: " & oFile.Type & _
         "File size: " & oFile.Size
Next
```

PLMEntity Object

See Also Legend Use Cases Properties Methods





Represents the parent object of a PLM object either in the database or in session.

Retrieving a PLMEntity Object

A **PLMEntity** object is retrieved either:

- From the **PLMEntities** collection object that is itself retrieved from either Search Results. In this case, any kinds of **PLMEntity** objects can be retrieved, depending on what is searched. You can get the properties of these objects, but not set them as long as they are not opened in session.
- From the **PLMOccurrence** object, that is either the **VPMRootOccurrence** or a **VPMOccurrence** object that both enable you to handle the occurrence model, to which it is aggregated in the product model. In this case, the possible kinds of **PLMEntity** objects are **VPMInstance**, **VPMPublication**, **VPMReference**, **VPMRepInstance**, and **VPMRepReference** objects. These objects enable you to manipulate the instance-reference model. You can get/set the properties of these objects, since they are already opened in session. See Product Modeler Overview.

Use the **Results** property of the **DatabaseSearch** object to return a **PLMEntities** collection object as a Search Results, from which you can retrieve one or several **PLMEntity** objects.

```
Dim cPLMEntities As PLMEntities
Set cPLMEntities = oDBSearch.Results

For i = 1 To cPLMEntities.Count
  Dim oPLMEntity As PLMEntity
  Set oPLMEntity = cPLMEntities.Item(i)
  MsgBox "This PLMEntity object name is: " & oPLMEntity.GetAttributeValue("PLM_ExternalID")
Next
```

Use the **PLMEntity** property of the **VPMRootOccurrence** or a **VPMOccurrence** object, inherited from the **PLMOccurrence** object, to return the **PLMEntity** object aggregated to it in a product model.

```
Dim oVPMOccurrence As VPMOccurrence
... 'Navigate the product model to find the appropriate object
Dim oPLMEntity As PLMEntity
Set oPLMEntity = oVPMOccurrence.PLMEntity
...
```

Refer to Navigating Product Structure and to Browsing Occurrence Model.

Using a PLMEntity Object

Each **PLMEntity** derived object shares with the others the methods inherited from the **PLMEntity** object.

Use the **GetAttributeValue** and **SetAttributeValue** to get/set the value of a PLM attribute.

```
Dim oPLMEntity As PLMEntity
... 'Retrieve the PLMEntity object as described above
MsgBox "This PLMEntity object title is: " & oPLMEntity.GetAttributeValue("V_Name")
oPLMEntity.SetAttributeValue("V_Name") = "New Title"
```

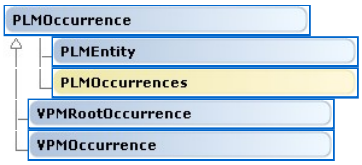
Use the **GetCustomType** to return the actual customization, type of a **PLMEntity** object.

```
Dim oPLMEntity As PLMEntity
... 'Retrieve the PLMEntity object as described above
MsgBox "This PLMEntity object customization type is: " & oPLMEntity.GetCustomType
```

Refer to each **PLMEntity** derived object to know its own properties and methods.

PLMOccurrence Object

See Also Legend Use Cases Properties Methods



Represents the parent object of the occurrence model objects.

The **PLMOccurrence** object enables you to manipulate the occurrence model. See Product Modeler Overview.

Retrieving a PLMOccurrence Object

You can retrieve a **PLMOccurrence** object through its derived objects only. See VPMRootOccurrence Object and VPMOccurrence Object.

Using a PLMOccurrence Object

Use the **PLMEntity** property to return the instance/reference model object associated with the current **PLMOccurrence** object..

```
Dim oPLMOccurrence As PLMOccurrence
... 'Retrieve either the VPMRootOccurrence or a VPMOccurrence object
Dim oPLMEntity As PLMEntity
Set oPLMEntity = oPLMOccurrence.PLMEntity
```

The **PLMOccurrences** property returns the **PLMOccurrences** collection object aggregated to the **PLMOccurrence** object. This collection object contains all the **PLMOccurrence** objects that are children of the **PLMOccurrence** object. If this collection object is empty, the **PLMOccurrence** object is a leaf node in the occurrence model tree. Prefer using the **Occurrences** properties of either the VPMRootOccurrence Object and the VPMOccurrence Object to navigate the occurrence model.

Selection Object

See Also Legend Use Cases Properties Methods

Represents the selection.

Retrieving the Selection Object

Use the **Selection** property to return the **Selection** object associated with the active **Editor** object.

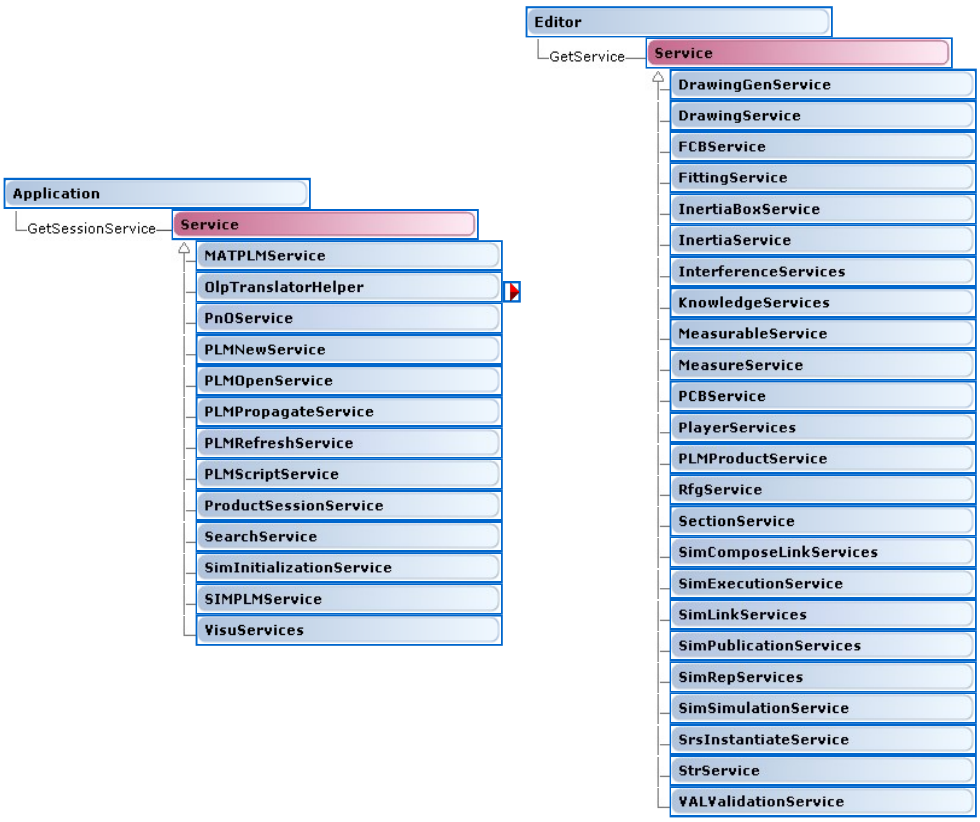
```
Dim oSelection As Selection
Set oSelection = CATIA.ActiveEditor.Selection
```

Using the Selection Object

Refer to the Selection object.

Service Object

See Also Legend Use Cases Properties Methods



Represents a service.

The **Service** object gives access to a set of object-independent operations, that either apply to the session and are editor and data independent, or globally apply to the objects controlled by the active editor and aggregated to the PLM root object. Because the **Service** object is an abstract object, you can manipulate its derived objects only.

A session-level service includes operations that either apply to any PLM type root object, or without any PLM root object active. On the opposite, an editor-level service applies only to a dedicated PLM type root object.

- Session-level services:
  - The **MATPLMService** object to create materials, set or retrieves core or covering materials, remove applied materials, and retrieves all materials in

session.
    - The **OlpTranslatorHelper** object used to access OLP data objects.
    - The **PnOService** object to access a person.
    - The **PLMNewService** object to create a PLM root object.
    - The **PLMOpenService** object to open a PLM root object.
    - The **PLMPropagateService** object to save changes held by the editor.
    - The **PLMRefreshService** object to manage the refresh operation.
    - The **PLMScriptService** object to handle scripts stored in the database.
    - The **ProductSessionService** object that aggregates a collection of **Shape3D** objects.
    - The **SearchService** object to to search for PLM objects.
    - The **SimInitializationService** object to initialize the simulation.
    - The **SIMPLMService** object that manages **SimulationReference** and other Simulation objects.
    - The **VisuServices** object dealing with layers, layer filters, windows and cameras.
- Editor-level services:
    - The **DrawingGenService** object and the **DrawingService** object that apply to a Drawing representation.
    - The **FCBService** object to create or retrieve flexible boards.
    - The **FittingService** object to create or manage tracks and Tpoints.
    - The **InertiaBoxService** object to retrieve the **InertiaBox** object representing the bounding box of a given geometric object.
    - The **InertiaService** object to retrieve the **Inertia** object for a given geometric object.
    - The **InterferenceServices** object to create an interference simulation.
    - The **KnowledgeServices** object to return the **Units** collection object or the **KnowledgeCollection** collection object.
    - The **MeasurableService** object to return objects to measure curves and surfaces.
    - The **MeasureService** object to return objects to measure curves and surfaces.
    - The **PCBServices** object to create Circuit Board Design objects.
    - The **PlayerServices** object to use the play methods.
    - The **PLMProductService** object that returns root objects from the **Editor** object.
    - The **RfgService** object to manage reference planes and surfaces.
    - The **SectionService** object that manages **Section** objects from the current review.
    - The **SimComposeLinkServices** object to create a link memory object.
    - The **SimExecutionService** object to perform simulation execution.
    - The **SimPublicationServices** object to manage publications.
    - The **SimRepServices** object to determine whether the representation reference is loaded, and to load it if it is not.
    - The **SimLinkServices** object to retrieve the occurrence, the representation instance, or the target of a link.
    - The **SimSimulationService** object to return the root occurrence of the product referred by the simulation.
    - The **SrsInstantiateService** object that manages Space Reference System objects.
    - The **StrService** object that applies to Structure objects.
    - The **VALValidationService** object that applies to validation objects.

Retrieving a Service Object

Use the **GetSessionService** method of the **Application** object to return a session-level **Service** object.

```
Dim oVisuServices As VisuServices
Set oVisuServices = CATIA.GetSessionService("VisuServices")
```

The table below lists the service identifier to pass to the **GetSessionService** method to return a session-level service.

Service	Identifier
MATPLMService	MATPLMService
OlpTranslatorHelper	OlpTranslatorHelper
PnOService	PnOService
PLMNewService	PLMNewService
PLMOpenService	PLMOpenService
PLMPropagateService	PLMPropagateService
PLMRefreshService	PLMRefreshService
PLMScriptService	PLMScriptService
ProductSessionService	ProductSessionService
SearchServices	Search
SimInitializationService	SimInitializationService
SIMPLMService	SimPLMService
VisuServices	VisuServices

Use the **GetService** method of the **Editor** object to return an editor-level **Service** object.

```
Dim oDrawingService As DrawingService
Set oDrawingService = CATIA.ActiveEditor.GetService("CATDrawingService")
```

The table below lists the service identifier to pass to the **GetService** method to return an editor-level service.

Service	Identifier
DrawingGenService	CATDrawingGenService
DrawingService	CATDrawingService
FCBService	FCBService
FittingService	FittingService
InertiaBoxService	InertiaBoxService
InertiaService	InertiaService
InterferenceServices	InterferenceServices
KnowledgeServices	KnowledgeServices
MeasurableService	MeasurableService
MeasureService	MeasureService
PCBService	PCBService
PlayerServices	PlayerServices
PLMProductService	PLMProductService
RfgService	RfgService

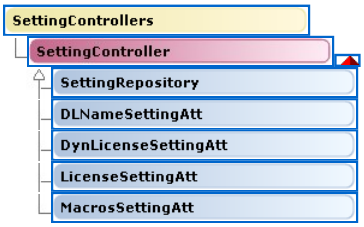
SectionService	SectionService
SimComposeLinkServices	SimComposeLinkServices
SimExecutionService	SimExecutionService
SimLinkServices	SimLinkServices
SimPublicationServices	SimPublicationServices
SimRepServices	SimRepServices
SimSimulationService	SimSimulationService
SrsInstantiateService	SrsInstantiateService
StrService	StrService
VALValidationService	ValidationService

Using a Service Object

Refer to each derived **Service** object to know the proposed operations and how to use them.

SettingController Object

See Also Legend Use Cases Properties Methods



Represents a setting controller.

The **SettingController** object enables you to access the setting attributes stored in setting repository files, and arranged in the different property pages of the Options dialog box you can display thanks to the **Options** command of the **Tools** menu. There are two kinds of setting controllers.

1. The **SettingRepository** object is the generic setting controller. It applies to any setting repository having an XML definition. Refer to Setting Repository Reference to have the list of such setting repositories
2. Specific setting controllers, each dedicated to a given setting repository that is not described using an XML file. For example, the **DynLicenseSettingAtt** object deals with the dynamic licensing setting repository that stores the list of available configurations and products for which you can request a dynamic license.

The generic setting controller and all of the specific ones share the five methods of the **SettingController** object to deal with the whole set, or a subset of the setting attributes of a setting repository:

- **Commit** to make a memory copy of the setting attribute values
- **Rollback** to restore the last memory copy of the setting attribute values
- **ResetToAdminValues** to restore the administered values of all the attributes
- **ResetToAdminValuesByName** to restore the administered values of a subset of the attributes
- **SaveRepository** to make a persistent copy of the setting attribute values in a file.

Retrieving the Generic Setting Controller

The generic setting controller can be retrieved for any of the setting repositories listed in Setting Repository Reference, provided the configuration or product it belongs to is installed on your computer. Use the **SettingControllers** property of the **Application** object to return the **SettingControllers** collection. Then use its **Item** method to return the generic setting controller applying to the GeneralGeneral setting repository.

```
Dim cSettingCtrls As SettingControllers
Set cSettingCtrls = CATIA.SettingControllers
Dim oSettingCtrl As SettingRepository
Set oSettingCtrl = cSettingCtrls.Item("GeneralGeneral")
```

Using the Generic Setting Controller

Use the **GetAttr** or **PutAttr** methods of the **SettingRepository** object to return or set the value of an attribute. The example below returns the value of the **DragAndDrop** attribute of the GeneralGeneral setting repository, and if this value is returned true, set it to false.

```
Dim oDragAndDrop As Boolean
oDragAndDrop = oSettingCtrl.GetAttr("DragAndDrop")
If oDragAndDrop Then
  oSettingCtrl.PutAttr "DragAndDrop", False
End If
```

Retrieving a Specific Setting Controller

Use the **SettingControllers** property of the **Application** object to return the **SettingControllers** collection. Then use its **Item** method with the appropriate identifier to return the specific **DynLicenseSettingAtt** object.

```
Dim oSettingCtrls As SettingControllers
Set oSettingCtrls = CATIA.SettingControllers
Dim oSettingCtrl As DLNameSettingAtt
Set oSettingCtrl = oSettingCtrls.Item("CATSysDynLicenseSettingCtrl")
```

The names to pass to the **Item** method are:

Setting Controller Objects	**Identifiers to Pass to the Item Method**
DLNameSettingAtt	CATSysDLNameSettingCtrl
DynLicenseSettingAtt	CATSysDynLicenseSettingCtrl

LicenseSettingAtt	CATSysLicenseSettingCtrl
MacrosSettingAtt	CATScriptMacrosSettingCtrl

Using a Specific Setting Controller

A specific setting controller manages the setting attributes located in a setting repository. A property and two methods enable you to manipulate each setting attribute:

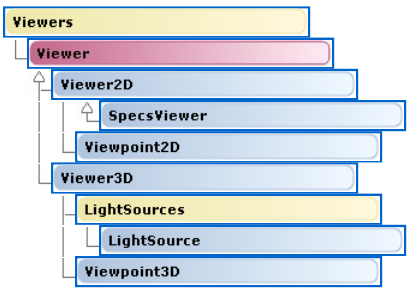
1. A read-write property to return or set the setting attribute value
2. A method to retrieve information about the setting attribute:
   - The level of administration of the setting attribute
   - The lock status of the setting attribute (Locked or Unlocked)
3. A method to set a lock onto the setting attribute

Depending on each setting attribute, when a property is not suitable, the read-write property can be replaced with a couple of methods. For example, a setting attribute managing a color must return or set three integer values. This cannot be performed by a property, and a method with three parameters must be used instead.

Refer to each derived specific **SettingController** object to know the setting attributes it manages and how to manipulate them.

Viewer Object

See Also Legend Use Cases Properties Methods



Represents a viewer.

A viewer is used to display a objects according to a given viewpoint and display options. Depending on the PLM type of the root object, the following viewers can be found in a window:

- 3D Shape Representation: a **Viewer3D** object for the part 3D objects and/or a **SpecsViewer** object for the part specification tree
- Product: a **Viewer3D** object for the assembly 3D objects and/or a **SpecsViewer** object for the assembly specification tree
- Drawing Representation: a **Viewer2D** object for the drawing sheets and/or a **SpecsViewer** object for the drawing specification tree.

When the window displays both a **Viewer3D** object and a **SpecsViewer** object, or one or several **Viewer2D** objects and a **SpecsViewer** object , it is a **SpecsAndGeomWindow** object. You can activate a given viewer in a multi-viewer window, fit all the scene in the viewer, update the display, zoom in and out, and capture the contents of the viewer as an image file.

Display options depend on the viewer type. All viewers share display options such as the background color and the display on the whole screen or in a smaller window. In addition, **Viewer3D** objects allow for different lighting modes and for modifying lighting intensity, and for depth effects, navigation styles, rendering modes, and clipping modes.

Viewpoint3D Object

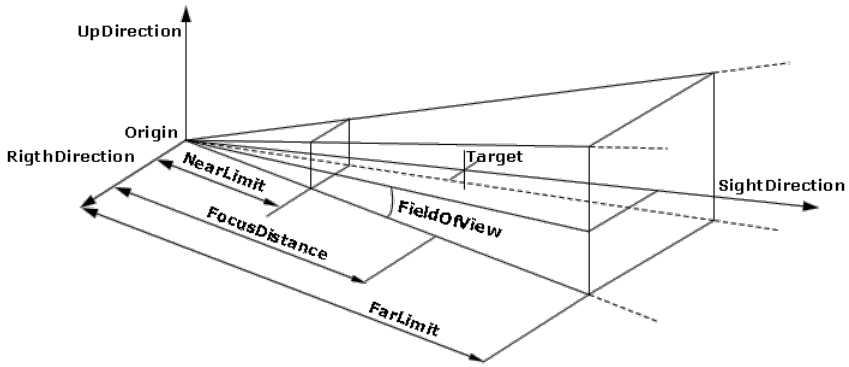
See Also Legend Use Cases Properties Methods

Represents a 3D viewpoint.

A **Viewpoint3D** object is aggregated to the **Viewer3D** object. It holds data to define how objects are seen to enable their display by a 3D viewer:

- The eye location, also named the origin of the scene to display, expressed in model units
- The distance from the eye location to the target, that is, to the looked at point in the scene.
- The sight, up, and right directions, defining a 3D coordinate system with the eye location as origin.
- The projection type: perspective (conic) or parallel (cylindrical).
- The zoom factor to apply for display.

The 3D viewpoint is the object that stores data which defines how your objects are seen to enable their display by a 3D viewer. This data includes namely the eye location, also named the origin, the distance from the eye to the target, that is to the looked at point in the scene, the sight, up, and right directions, defining a 3D axis system with the eye location as origin, the projection type chosen among perspective (conic) and parallel (cylindrical), and the zoom factor. The right direction is not exposed in a property, and is automatically computed from the sight and up directions.



VisuServices Object

See Also Legend Use Cases Properties Methods

Represents a service for visualization purposes.

The **VisuServices** object lets you manage:

- The **Cameras** collection object.
- The layers and the layer filters.
- The hidden elements visibility.
- A new window for the data displayed in the active one.

Retrieving the VisuServices Object

Refer to the Service Object.

Using the VisuServices Object

Use the **Cameras** property to return the **Cameras** collection object.

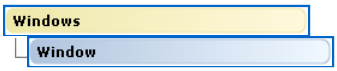
```
Dim cCameras As Cameras
Set cCameras = oVisuServices.Cameras
```

Use the **NewWindow** method to create a new **Window** object.

```
Dim oWindow As Window
Set oWindow = oVisuServices.NewWindow()
```

Windows Collection Object

See Also Legend Use Cases Properties Methods



A collection of **Window** objects that represent of all the windows currently managed by the application.

The **Windows** collection object gathers **Window** objects which make the link with the windowing system and display objects in a viewable form, mainly in 3D or 2D modes, or as a specification tree in graph or tree mode. Windows of the collection can be arranged in the frame.

Retrieving the Windows Collection

Use the **Windows** property of the **Application** object to return the **Windows** collection object.

```
Dim cWindows As Windows
Set cWindows = CATIA.Windows
```

Using the Windows Collection

Use the **Item** method to return a window from the **Windows** collection object, namely below the third one.

```
Dim oWindow As Windows
Set oWindow = CATIA.Windows.Item(3)
```

Note that you can also use the window name in place of the window range. The example below returns the search window the name of which is "Name Like Part".

```
Dim oWindow As Windows
Set oWindow = CATIA.Windows.Item("Name Like Part")
```

Use the **Arrange** method to arrange the windows. You can arrange the windows, according to the values of the CatArrangeStyle enumeration:

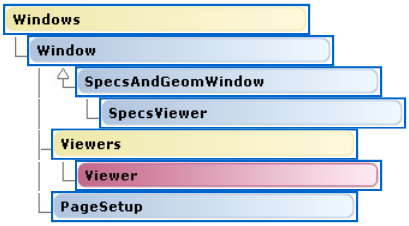
- As horizontal tiles.
- As vertical tiles.

- As a cascade.

```
CATIA.Windows.Arrange(catArrangeCascade)
```

Window Object

See Also Legend Use Cases Properties Methods



Represents a window.

A **Window** object aggregates a **Viewers** collection object that enables the window to display the application data in the appropriate modes using viewers. A **SpecsAndGeomWindow** object features altogether a 2D or a 3D viewer and a specification tree viewer. A window can be activated, that is becomes the active one, using the **Activate** method. This implies that the editor that controls the objects displayed in this window is also activated if it was not, and that subsequent interactions will affect it until another window is activated instead. The **Viewers** property returns the aggregated **Viewers** collection object, and the **ActiveViewer** property returns the active viewer in the window.

About Collections

You will find below some tips to retrieve objects in collections.

A collection is an object that gathers objects of the same type and is denoted as a plural name. For example, the **Editors** object is a collection of **Editor** objects and provides methods for managing individual editors in the collection.

A member of the collection is retrieved using the **Item** method and the index of the member in the collection. Usually, the argument representing the index in the **Item** method is a Variant. This means that it can either represent the rank of the member in the collection or the name you assigned to this member using the **Name** property. The rank in a collection begins at 1. For example, assume that the window named **CATIA** is the sixth window in the **Windows** collection. To retrieve this window in the **oWin** variable, write:

VB VBA C# VB.NET Python

```
Dim oWin
Set oWin = CATIA.Windows.Item(6)

Dim oWin As Window
Set oWin = CATIA.Windows.Item(6)

private INFITF.Window oWin
...
private void GetWindow() {
  oWin =  CATIA.Windows.Item(6)

Dim oWin As INFITF.Window
oWin = CATIA.Windows.Item(6)

oWin = CATIA.Windows.Item(6)
```

or write:

VB VBA C# VB.NET Python

```
Dim oWin
Set oWin = CATIA.Windows.Item("CATIA")

Dim oWin As Window
Set oWin = CATIA.Windows.Item("CATIA")

private INFITF.Window oWin
...
private void GetWindow() {
  oWin =  CATIA.Windows.Item("CATIA")

Dim oWin As INFITF.Window
oWin = CATIA.Windows.Item("CATIA")

oWin = CATIA.Windows.Item("CATIA")
```

Each collection has a **Count** property that holds the number of members in the collection. This is a handy property to scan the whole collection. For example, to print the name of each window of the collection of windows in a message box, write:

VB VBA C# VB.NET Python

```
For I = 1 To CATIA.Windows.Count
  msgbox CATIA.Windows.Item(I).Name
Next

For I = 1 To CATIA.Windows.Count
  msgbox CATIA.Windows.Item(I).Name
Next

for (int i = 1; i <= CATIA.Windows.Count; i++)
{
```

```
  string message = CATIA.Windows.Item(i).Name;
  string caption = "Window Name";
  MessageBoxButtons buttons = MessageBoxButtons.OK;
  DialogResult result;
  result = MessageBox.Show (message, caption, buttons);
  ...
}

For I As Integer = 1 To CATIA.Windows.Count
  MessageBox.Show(CATIA.Windows.Item(I).Name, "Window Name")
Next

import ctypes

for i in range(1, CATIA.Windows.Count):
  msgbox = ctypes.windll.user32.MessageBoxA
  ret = msgbox(None, CATIA.Windows.Item(i).Name, 'Window Name', 0)
  ...
```

or write:

VB VBA C# VB.NET Python

```
I = 0
Do
  I = I + 1
  msgbox CATIA.Windows.Item(I).Name
Loop Until I = CATIA.Windows.Count

I = 0
Do
  I = I + 1
  msgbox CATIA.Windows.Item(I).Name
Loop Until I = CATIA.Windows.Count

int i = 0;
do
{
  i = i + 1;
  string message = CATIA.Windows.Item(i).Name;
  string caption = "Window Name";
  MessageBoxButtons buttons = MessageBoxButtons.OK;
  DialogResult result;
  result = MessageBox.Show (message, caption, buttons);
  ...
} while (i <= CATIA.Windows.Count)

Dim I As Integer = 0
Do
  I = I + 1
  MessageBox.Show(CATIA.Windows.Item(I).Name, "Window Name")
Loop Until I = CATIA.Windows.Count

import ctypes

i = 0
while (i <= CATIA.Windows.Count):
  i = i + 1;
  msgbox = ctypes.windll.user32.MessageBoxA
  ret = msgbox(None, CATIA.Windows.Item(i).Name, 'Window Name', 0)
  ...
```

You can also use the **For Each** instruction to scan the collection, and get rid of the counter:

VB VBA C# VB.NET Python

```
For Each oWin In CATIA.Windows
  msgbox oWin.Name
Next

For Each oWin In CATIA.Windows
  msgbox oWin.Name
Next

private INFITF.PageSetup oPgSetup
...
foreach (oWin in CATIA.Windows)
{
  string message = oWin.Name;
  string caption = "Window Name";
  MessageBoxButtons buttons = MessageBoxButtons.OK;
  DialogResult result;
  result = MessageBox.Show (message, caption, buttons);
}

For Each oWin As INFITF.Window In CATIA.Windows
  MessageBox.Show(oWin.Name, "Window Name")
Next

import ctypes

for i in range(1, CATIA.Windows.Count):
  msgbox = ctypes.windll.user32.MessageBoxA
  ret = msgbox(None, CATIA.Windows.Item(i).Name, 'Window Name', 0)
  ...
```

In this case, the **oWin** variable is reinitialized using the current window, starting with the first one and ending with the last one.

A collection may have a **Add** method to enable you to create a new member in the collection. Otherwise, a method is provided in another object to enable you to create new objects and make them members of the collection. For example, the **Window** object has a **NewWindow** method that duplicates the window from which the method is called and adds it to the collection of windows. In the same way, a collection may have a **Remove** method to enable you delete a member of the collection. This does

not apply to objects that do require a consistent system management, such as editors, or that are managed by other software, such as printers.

About SafeArrayVariant

Many methods which return a value do not return a single value, but an array of values. For example, the **GetOrigin** property of the **Viewpoint3D** object returns a 3D point as an array of three coordinates. This array is returned as an output argument.

Such arguments are visible in the VB/VBA object browser as a **variant** but the reference documentation lists them as **CATSafeArrayVariant**.

You can retrieve in the **MyVPOrigin** variable the origin of the 3D viewpoint of the active viewer in the active window as follows:

```
ReDim MyVPOrigin(2)
CATIA.ActiveWindow.ActiveViewer.Viewpoint3D.GetOrigin MyVPOrigin
```

You should directly use the ReDim statement to assign a size to the array while declaring it as a variable size array, and then call the GetOrigin method. Assigning a size of 2 allocates an array containing three values.

To access each coordinate, you need to go deeper:

- The x coordinate is in **MyVPOrigin(0)**
- The y coordinate is in **MyVPOrigin(1)**
- The z coordinate is in **MyVPOrigin(2)**

Contrary to collections, a CATSafeArrayVariant's index begins at 0. To set a new triplet of values, you can write:

```
MyVPOrigin = Array(150, 200, 50)
CATIA.ActiveWindow.ActiveViewer.Viewpoint3D.PutOrigin MyVPOrigin
```

or

```
CATIA.ActiveWindow.ActiveViewer.Viewpoint3D.PutOrigin Array(150, 200, 50)
```

But to modify the y coordinate only, write:

```
MyVPOrigin(1) = 200
CATIA.ActiveWindow.ActiveViewer.Viewpoint3D.PutOrigin MyVPOrigin
```

When you don't know the size of an array, use the UBound function. It returns the rank of the highest element in the array. For example the following example returns 2 in Highestrank:

```
Highestrank = Ubound(MyVPOrigin)
```

About Subs and Functions

A method exposed by an objects is called by Visual Basic:

- a **Sub** if it doesn't return any value
- a **Function** if its does.

Be careful when the methods requests arguments. To pass arguments to a **Sub** with Visual Basic Script, do not use parentheses as follows:

```
Object.Sub arg1, arg2, arg3
```

But use parentheses with a **Function**:

```
Dim ReturnedObject As AnyObject
Set ReturnedObject = Object.Function (arg1, arg2, arg3)
```

You must use **Set** only if the returned value is an object, but not if it is a character string or a number. Nevertheless, character string and number defined as CATIA literals are objects and **Set** must be used if a **Function** returns a literal object.

Finally, you don't have to use **Set** if you store your return value in a **Property**:

```
myObject.aggregatedObject = Object.Function (arg1, arg2, arg3)
```

because there is no actual aggregatedObject variable, a property is a syntactical shortcut for accessor methods, here get_aggregatedObject and set_aggregatedObject, allowing to present those methods as an attribute of the object. The previous syntax is so equivalent to:

```
myObject.set_aggregatedObject( Object.Function (arg1, arg2, arg3) )
```

and no **Set** is required.

About Numbers, Literals, and Units

Except when explicitly documented, numerical values stored and internally handled for computations are expressed using the MKSA unit system except for two dimensions:

- Length are expressed in mm
- Angles are expressed in decimal degrees

This means that dimensions returned **may not be homogeneous**: surfaces are not returned in mm2 (by homogeneity with length in mm) but in m2 (MKSA).

The user interface can be set to display and get values from the end user according to another unit system which better match your needs or habits.

The parameter values you can set using macros must be expressed using the same unit system, since the user interface filter does not exist when you run macros. This also ensures your macros portability. There is one exception: the literals.

Literals are specific objects that represent a parameter with a given type. For example, the **Length** object is dedicated to store a length, but its state of object brings more

that the simple value storage. The **Length** object derives from the **Dimension** object, and thus inherits from it the **ValuateFromString** method. This method allows the value stored in the **Length** object to be valuated using a figure and a unit. For example, valuate the radius of a face fillet using the **Radius** property of the **FaceFillet** object which aggregates a **Length** object to store this radius:

```
MyFaceFillet.Radius.ValuateFromString("5.08mm")
```

The character string is interpreted as a value of 5.08 expressed in mm. You can enter a decimal value since the **Dimension** object derives from the **RealParam** object which allows for real values to be set. You may want to enter inches instead. Simply write:

```
MyHole.Diameter.ValuateFromString("2in")
```

pTo be compatible with formulas syntax, if you don't specify a Unit for the argument of **ValuateFromString**, the MKSA units are used: length are expressed in meters and angles in radians.
As a thumb rule, always specify the unit when using **ValuateFromString** or formulas.

The available unit symbols you can use are those listed in the Units tab-page of the **Tools->Options** menu. The **RealParam** and the **IntParam** objects provide to their derived objects the **Value** method which sets or returns the value expressed in the MKSA unit system, except for length expressed in millimeters and angles expressed in decimal degrees.

About Microsoft Automation Languages, Debug, and Compatibility

You will find in this section an overview of the different types of Automation languages provided by Microsoft, their main differences, and some language limitations, how to use the Script Debugger, and some tips when upgrading your installation to a new release.

Supported Scripting Languages

The supported scripting languages are:

- V6 native: CATScript and VBScript (VBS).
- Visual Basic for Applications: VBA.
- Visual Studio Tools for Applications (VSTA): VB.Net and C#.
- Python which is not addressed in this article. See About Python.

The Automation objects can be accessed using any of these languages.

CATScript is a Dassault Systèmes specific language kept for compatibility reasons, but CATScript macros are converted to VBScript by removing the typing information whenever they are run.

These languages differ from one another. For example, returning the active printer using the **ActivePrinter** property of the **Application** object is written as follows, depending on the language:

Languages	Code	Comments
CATScript	`Dim oPrinter As Printer` `Set oPrinter = CATIA.ActivePrinter`	The variable `oPrinter` is declared using the **Dim** keyword and typed using the **As** keyword. Nevertheless, this type is removed when running the macro as a VBScript one.
VBScript	`Dim oPrinter` `Set oPrinter = CATIA.ActivePrinter`	The variable `oPrinter` is declared using the **Dim** keyword, but not typed.
VBA	`Dim oPrinter As Printer` `Set oPrinter = CATIA.ActivePrinter`	The variable `oPrinter` is declared using the **Dim** keyword and typed using the **As** keyword.
VB.Net	`Dim oPrinter As INFITF.Printer` `oPrinter = CATIA.ActivePrinter`	The variable `oPrinter` is declared using the **Dim** keyword and typed using the **As** keyword by using the name of the type library containing the object as namespace. The variable value is assigned without using the **Set** keyword.

C#	`private INFITF.Printer oPrinter;`
	`...`
	`private void GetActivePrinter() {`
	`  oPrinter = CATIA.ActivePrinter;`
		The variable `oPrinter` is declared and typed outside of the function using the private keyword and by using the name of the type library containing the object as namespace. The variable value is assigned without using the **Set** keyword.

Abstract Objects

An abstract object [1] represents an object you cannot handle, but used as a parent type for objects of the same kind. When a property or a method returns an abstract object like the **ActiveViewer** property of the **Application** object that returns a **Viewer** object, the different languages behave as follows. Assume the active viewer is a 3D viewer.

Languages	Code	Comments
CATScript	`Dim oViewer As Viewer` `Set oViewer = CATIA.ActiveWindow.ActiveViewer`	The variable `oViewer` must be declared as specified in the property or method signature, that is, as **Viewer** object, but the returned `oViewer` is a **Viewer3D** object. Nevertheless, this type is removed when running the macro as a VBScript one.
VBScript	`Dim oViewer` `Set oViewer = CATIA.ActiveWindow.ActiveViewer`	The variable `oViewer` is not typed, and the returned `oViewer` is a **Viewer3D** object.
VBA	`Dim oViewer As Viewer3D` `Set oViewer= CATIA.ActiveWindow.ActiveViewer`	The variable `oViewer` should be declared as a **Viewer3D** object. In this case, the returned `oViewer` is a **Viewer3D** object. When editing macros using VBA, this sets the Intellisense with the **Viewer3D** object properties and methods.
	`Dim oViewer As Viewer` `Set oViewer = CATIA.ActiveWindow.ActiveViewer`	Nevertheless, the variable `oViewer` can be declared as a **Viewer** object. In this case, the returned `oViewer` is a **Viewer** object, and you can either convert it to, or use it as a **Viewer3D** object.
VB.Net	`Dim oViewer As INFITF.Viewer3D` `oViewer = CATIA.ActiveWindow.ActiveViewer`	The variable `oViewer` should be declared as a **Viewer3D** object. In this case, the returned `oViewer` is a **Viewer3D** object.

	`Dim oViewer As INFITF.Viewer`
	`oViewer = CATIA.ActiveWindow.ActiveViewer`
	`Dim oViewer3D As INFITF.Viewer3D`
	`oViewer3D = DirectCast(oViewer, INFITF.Viewer3D)`
		Nevertheless, the variable `oViewer` can be declared as a **Viewer** object. In this case, the returned `oViewer` is a **Viewer** object. You can convert (cast) `oViewer` to a **Viewer3D** object thanks to the `DirectCast` function.

C#	`private INFITF.Viewer3D oViewer;`
	`...`
	`private void Macro1() {`
	`  this.oViewer = CATIA.ActiveWindow.ActiveViewer;`
		The variable `oViewer` should be declared as a **Viewer3D** object. In this case, the returned `oViewer` is a **Viewer3D** object.

	`private INFITF.Viewer oViewer;`
	`private INFITF.Viewer3D oViewer3D;`
	`...`
	`private void GetActiveViewer() {`
	`  this.oViewer = CATIA.ActiveWindow.ActiveViewer;`
		Nevertheless, the variable `oViewer` can be declared as a **Viewer** object. In this case, the returned `oViewer` is a **Viewer** object. You can convert (cast) `oViewer` to a **Viewer3D** object.

```
this.oViewer3D = (INFITF.Viewer3D) oViewer;
```

If the returned object `oViewer` is a **Viewer3D** object, you can call the properties and methods of the **Viewer** object in addition to those of the **Viewer3D** object. If the returned object `oViewer` is a **Viewer** object, even if in this case the real underlying object is a 3D viewer, you can call the properties and methods of the **Viewer** object only, and not those specific to the **Viewer3D** object. With VB.Net and C#, again in this case where the real object is a 3D viewer, you can convert (cast) the returned **Viewer** object to a **Viewer3D** object. The resulting variable `oViewer3D` is a **Viewer3D** object. You can now call the properties and methods of the **Viewer3D** object in addition to those inherited from the **Viewer** object.

Case of the AnyObject Object

When an **AnyObject** object is returned, it is advised not to type the object declaration.

Languages	Code	Comments
CATScript	`Dim oRoot As Viewer` `Set oRoot = CATIA.ActiveEditor.ActiveObject`	The variable `oRoot` must be declared as specified in the property or method signature to match CATScript rules, but this type is removed when running the macro as a VBScript one.
VBScript	`Dim oRoot` `Set oRoot = CATIA.ActiveEditor.ActiveObject`	The variable `oRoot` is not typed, and the returned `oViewer` is a **Viewer3D** object.
VBA	`Dim oRoot ' As AnyObject` `Set oRoot = CATIA.ActiveEditor.ActiveObject`	The variable `oRoot` is declared, but not typed. Depending on the active object, a **Part**, **DrawingRoot**, or **VPMReference** object is returned, and can be used. In this case, when editing macros using VBA, this does not set the Intellisense with any properties and methods.

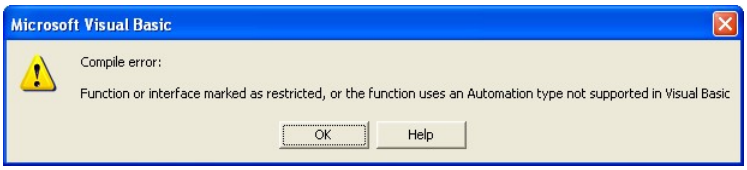
Arrays

You will also have to take care of **methods with array parameters** like in the following example where we extract and display the multiple possible values of a **StringParameter** in VBScript or CATScript:

```
Dim strParam1 As StrParam
Set strParam1 = parameters1.Item("STRING")

iSize = strParam1.GetEnumerateValuesSize()

Redim myArray(iSize-1)
strParam.GetEnumerateValues myArray

For i = 0 To iSize-1
  msgbox myArray(i)
Next
```

Copying/pasting this piece of code in a VB project may lead, depending on your VBA level to a compilation error because VB may be unable to deal with the signatures that we use for our array types and issues the following message:

```
Function or interface marked as restricted, or the function uses an Automation type not supported in Visual Basic.
```



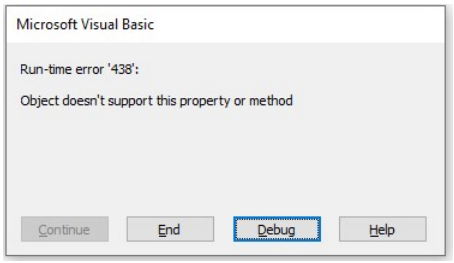
A simple workaround is to untype the variable on which the method is applied, that is, strParam1, the type of which is commented out below.

```
Dim strParam1 ' As StrParam
Set strParam1 = parameters1.Item("STRING")

iSize = strParam1.GetEnumerateValuesSize()

Redim myArray(iSize-1)
strParam1.GetEnumerateValues myArray

For i = 0 To iSize-1
    msgbox myArray(i)
Next
```

In some case, the above workround will not work as the actual type of the untyped item may not be the expected one. Following example shows a variable `ncrepman` that must be untyped as per workaround above, but the affection will retrieve an object of type `VPMReference` instead of its derived type `ManufacturingNCRepManagement`.

```
 Sub CATMain()
    Dim myEditor As Editor
    Set myEditor = CATIA.ActiveEditor
    Dim MyPrdService As PLMProductService
    Set MyPrdService = myEditor.GetService("PLMProductService")
    Dim myRootOccurrence As VPMRootOccurrence
    Set myRootOccurrence = MyPrdService.RootOccurrence
    Dim myinstance As VPMInstance
    Set myinstance = myRootOccurrence.ReferenceRootOccurrenceOf.Instances.Item(1)

    Dim refinst As VPMReference
    Set refinst = myinstance.ReferenceInstanceOf
    Dim ncrepman 'As ManufacturingNCRepManagement
    Set ncrepman = refinst
    Set newncrep = ncrepman.CreateNCRep("12345678", 2, listFiles)
  End Sub
```

`CreateNCRep` method is specific to type `ManufacturingNCRepManagement`, so calling it on a `VPMReference` will trigger this error:



To avoid such issue, the variable can be typed the same way as in the previous case using an intemediary function as below:

```
Sub CATMain()
 ...
    Dim ncrepman 'As ManufacturingNCRepManagement
    Set ncrepman = getManufacturingNCRepMgt(refinst)

        ReDim listFiles(3)
        Set newncrep = ncrepman.CreateNCRep("12345678", 2, listFiles)
End Sub

Function getManufacturingNCRepMgt(ByRef refinst As VPMReference) As ManufacturingNCRepManagement
    Dim ncrepman As ManufacturingNCRepManagement
    Set ncrepman = refinst
    Set getManufacturingNCRepMgt = ncrepman
End Function
```

If the problematic object is not declared but retrieved through a property, VBA may be able to find its type. This is for example the case for the Selection object in the following code that will not compile in VBA:

```
Redim sTypesFilter(…)
sTypesFilter(0)=…
sStatus = CATIA.ActiveEditor.Selection.SelectedElement2(sTypeFilter, …)
```

To avoid this, a non-typed intermediary object should be used:

```
Redim sTypesFilter(…)
sTypesFilter(0)=…
Dim oSelection
Set oSelection = CATIA.ActiveEditor.Selection
sStatus = oSelection.SelectedElement2(sTypeFilter, …)
```

When the array is the only argument of a method, the following syntax, using parenthesis, should be avoided:

```
Redim myArray(15)
strParam.GetEnumerateValues (myArray)
```

For methods that are not functions (meaning that they do not have a return value), this syntax requires to pass the argument by reference which may not work in some cases. The right syntax uses either no parenthesis:

```
strParam.GetEnumerateValues myArray
```

or the call keyword:

```
call strParam.GetEnumerateValues (myArray)
```

Script Debugger

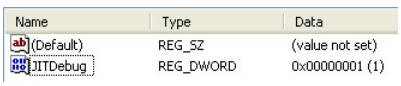
When developing in-process macros, you can use Microsoft (R) Script Debugger than you can freely download from the Microsoft (R) web site. Once installed, an error or a **Stop** order in the script will give hand to the debugger:

```
For i = 0 To Ubound(myArray)
    Stop
    msgbox myArray(i)
Next
```

To make the **Stop** order active, add to, or locate in the registry the JITDebug key in:

HKEY_CURRENT_USER\Software\Microsoft\Windows Script\Settings

and set it to 1 thanks to the contextual menu.



See the Microsoft (R) Script Debugger's documentation for more information on how to proceed.

Inter Release Compatibility

Virtual Function Table Compatibility

The Automation object model evolves with each new release. Namely, new properties and methods may be added to an existing object. Depending on the way VBA calls those methods this may have an impact on your application even if a new method has been added.

The following code for example:

```
Dim oObject
oObject.DoThis
```

`oObject` is not typed and will perform a late binding call to `DoThis`, using the `Invoke` method at build time and leaving `oObject` perform the actual call to `DoThis` at runtime inside its implementation of `Invoke`.

The following code however:

```
Dim oObject As SpecificTypeOfObject
oObject.DoThis
```

will perform an early binding call, meaning basically that a description of the virtual function table of `oObject` will be re-created from the information available in the type library describing the **SpecificTypeOfObject** type and the call will be made using a position in this table.

If, on a new release, the order of methods in this virtual function table changes, which may be the case when adding new methods, the application will call the wrong method leading to difficult-to-debug runtime problems.

Moreover, the description extracted by VBA from the type library does not seem to be always refreshed when the type library changes. Meaning that recompiling the project may not correct the problem. Here is a method that solves this problem:

- Open you VBA or VSTA project.
- Remove all references to V6/**3D**EXPERIENCE type libraries using the `Project/References` or `Tools/References` menu items.
- Save and close the project.
- Re-open the project and add again the needed references.

Obsolete Type Libraries

Some type libraries may become obsolete and disappear on a new V6/**3D**EXPERIENCE release. If an existing VBA project has references to a type library that does not exist anymore, and if your access rights allow you to modify the Windows Registry, references to those type libraries are automatically removed when opening the VBA project. Otherwise, the following message may be experienced when launching a macro:

```
CNEXT CATScriptError Message Scripting ERR_1000
Execute the script "XXXXX" |XXXXX=
The script entry point could not be found.
XXXXX
Define a "CATMain" procedure which will be the entry point of the script.
```

To avoid this, launch VBA (Alt-F11) and use the `Tools/References` menu item to launch the `References` dialog window. In this dialog window, uncheck the reference to the concerned type library and click OK.

This problem may also occur in VBA projects of non V6/**3D**EXPERIENCE applications.

Running Two Versions or Releases on the Same Machine

You may need to run two, for example, CATIA or DELMIA versions on the same machine. For example, you want to run V5R27 and V5R28, or V5R27 and **3D**EXPERIENCER2017x. If you want to run macros, note that CATIA or DELMIA use a scripting server associated with a given version/release, and so you cannot concurrently run macros in two different CATIA or DELMIA application windows running two different versions.

To switch running macros from one version to another one, say from V5R27 to **3D**EXPERIENCER2017x, do the following:

- Close all CATIA or DELMIA windows.
- Unset the V5R27 scripting server:
  1. Open a prompt window in administrator mode.
  2. Change to the folder in which you installed CATIA or DELMIA.

     By default, this folder is:

     ```
     C:\Program Files\Dassault Systemes\B27\win_b64\code\bin
     ```

     Replace `win_b64` with `intel_a` if you run a 32 bit version.

  3. Run the following command:

     ```
     catstart -run "V5RegServer -unset CATIA"
     ```

  4. This command is a silent command. Wait about for one minute to let it complete.
- Set the **3D**EXPERIENCER2017x scripting server:
  1. Open a prompt window in administrator mode.
  2. Change to the folder in which you installed CATIA or DELMIA.

     By default, this folder is:

     ```
     C:\Program Files\Dassault Systemes\B419\win_b64\code\bin
     ```

     Replace `win_b64` with `intel_a` if you run a 32 bit version.

  3. Run the following command:

     ```
     catstart -run "DSYAdmRegSrv -set CATIA" [ -env MyV6Environment -DirEnv MyV6EnvDirectory]
     ```

  4. This command is a silent command. Wait about for one minute to let it complete before starting CATIA or DELMIA V6R2011.

To unset a V6/**3D**EXPERIENCE scripting server, use the -unset option of the DSYAdmRegSrv command. In the same way, to set a V5 scripting server, use the -set option of the V5RegServer command.

In V5 many environment files can be created in any folders, so it is recommended to use the `-end` and the `-direnv` options. In V6/**3D**EXPERIENCE only one environment file is created at installation time and is located in the CATEnv folder of the installation folder, so those options can be omitted unless you are using Apps created with the CAA C++ API.

Usage of the VBA DoEvent Function

The general purpose of the VBA function `DoEvent` is to allow the system to manage input events so that the UI becomes more responsive.

Beside this function is generally dangerous because it can generate unexpected reentrancy. CATIA infrastructure is not designed to manage event treatment while the

process is blocked into a VBA macro execution. Calling `DoEvent` while a macro is executed can so lead to an unpredictable behavior.

Using `DoEvent` in VBA macros is not recommended.

References

[1] Object Architecture Overview

About Python

This article gives you general notes regarding how to consume the Automation API using Python programming language.
Note that using the Python programming language to access the Automation API is only available through the `Python` command of the `Tools` section of the action bar, available in some simulation apps.

We advise that you do not use the Python bindings unless you are familiar with the Python language. Full details on Python, including tutorials, are available at www.python.org.

A complete example of how to create and run a Python script is provided in the Getting Started with Python article.

The following notes provide an overview of the differences between the two programming languages. As the Automation API reference documentation types and most coding examples provided in this documentation are provided in Visual Basic for Applications (aka VBA) language, this allows you to take advantage of existing examples to create Python scripts.

1. Python is case sensitive. `myFeatures` is a different name from `MyFeatures`
2. Python objects are created dynamically via an assignment. A declaration, as can be done in Visual Basic using the `Dim` keyword, is not needed.
3. The Python assignment does not require a Set keyword.
4. Boolean branching syntax: in Python the end of a boolean branch statement block is determined by the indentation level.

Visual Basic for Applications Syntax

```
If condition Then
   ...
End If
```

Python Syntax

```
if condition:
    ...
```

5. Function definition: in Python, the end of a function block is determined by the indent level.

Visual Basic for Applications Syntax

```
Sub foo()
   ...
End Sub


Sub bar()
   ...
End Sub
```

Python Syntax

```
def foo():
        ...


def bar():
        ...
```

All function calls require parentheses.
The VB Sub and Function are both defined in Python using the `def` keyword in Python. The equivalent of a Visual Basic `Sub` is a function that returns None. To exit a function use:

Visual Basic for Applications Syntax

```
Exit Sub
Exit Function
```

Python Syntax

```
return
return xxx
```

6. Loops: the VBA `For Each` construction becomes a `for` loop, `Exit For` becomes `break` in Python and Python offers a `continue` keyword to continue the loop which has to be simulated in VBA using a `GoTo` and doesn't exist at all in VBScript.

Visual Basic for Applications Syntax

```
' -- declare and initialize an array
Dim myArray(3)
myArray(0) = "print"
myArray(1) = "skip"
myArray(2) = "exit"

' -- loop on array
For Each i In myArray

   If i="exit" Then
        msgbox "exit detected"
        Exit For           ' exit loop
   ElseIf i="skip" Then
        msgbox "Skip detected"
        GoTo ContinueLoop  ' loop on next item
   End If

   msgbox "print detected"
   ContinueLoop:
Next               ' loop on next item : end loop
```

Python Syntax

```
# -- initialize an array
myArray = []
myArray.append('print')
myArray.append('skip')
myArray.append('exit')

# -- loop on array
for i in myArray:

        if i=='exit':
                print ("exit detected")
                break     # exit loop

        elif i=='skip':
                print ("skip detected")
                continue  # loop on next item

        print ("print detected")

                          # enf of loop: indentation level
```

7. Type names in VB and Python are not the same, for example:

Visual Basic for Applications Syntax

```
String / CATBSTR (in reference documentation)
double
```

Python Syntax

```
unicode (Python-2.7), str (Python-3.x)
float
```

8. Working with CATSafeArrayVariant in Python

Visual Basic for Applications Syntax

```
' -- VB Method Call with input/output argument
' -- argument name usually starts with lower-case io
ReDim MyVPOrigin(2)
CATIA.ActiveWindow.ActiveViewer.Viewpoint3D.GetOrigin MyVPOrigin
' -- VB Method Call with input argument
' -- argument name usually starts with lower-case i
CATIA.ActiveWindow.ActiveViewer.Translate Array(10.0, 20.0, 30.0)
' -- VB Property to get value
Dim MyOutput as SimOutput
...
Dim MyPoints()
MyPoints = MyOutput.SectionPoints

' -- VB Property to set value
Dim MyOutput as SimOutput
...
```

Python Syntax

```
# -- Python Method Call
# -- empty argument tuple must have correct number of members
MyVPOrigin = CATIA.ActiveWindow.ActiveViewer.Viewpoint3D.GetOrig

# -- Python Method Call
CATIA.ActiveWindow.ActiveViewer.Translate((10.0, 20.0, 30.0))


# -- Python Method Call
MyPoints = MyOutput.SectionPoints


# -- Python Method Call
MyOutput.SectionPoints = (1, 2, 3)
```

```
MyOutput.SectionPoints = Array(1, 2, 3)
```

Navigating the Automation Model

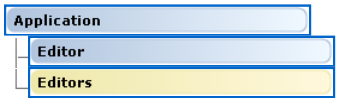
Retrieving an object from another object is the way the Automation model can be navigated. The most common navigation mechanism is jumping from an object to the object it aggregates thanks to the object properties, starting from the Application object up to the lowest objects in the aggregation tree. If this mechanism generally fulfills the navigation needs of a standalone application, this is not always convenient for a complex system that can be extended using 3rd-party or in-house developed applications. To match this need, several protocols are available, depending on the object you are starting with.

Navigating collections is not described here. Refer to the related topics.

Related Topics
About Collections
Service Object
Selection Object

Properties

The most common navigation mechanism Automation brings thanks to its aggregation model is using the properties of an object that retrieves its aggregated objects. For example, the **Application** object aggregates, among other objects, an **Editor** object that represents the active editor, that is the one that manages the root object of the object currently being edited, also named the active object, and an **Editors** collection object that contains a reference to all the **Editor** objects currently existing in the session. The partial Application object aggregation model is as follows:



To retrieve these objects, the **Application** object features two properties:

1. An **ActiveEditor** property that returns the active **Editor** object.
2. An **Editors** property that returns the active **Editors** collection object.

Use the **ActiveEditor** property to return the active **Editor** object from the **Application** object.

VB VBA C# VB.NET Python

```
Dim oEditor
Set oEditor = CATIA.ActiveEditor

Dim oEditor As Editor
Set oEditor = CATIA.ActiveEditor

oEditor = CATIA.ActiveEditor

Dim oEditor As INFITF.Editor
Set oEditor = CATIA.ActiveEditor

oEditor = CATIA.ActiveEditor
```

Use the **Editors** property to return the **Editors** collection object from the **Application** object.

VB VBA C# VB.NET Python

```
Dim cEditors
Set cEditors = CATIA.Editors

Dim cEditors As Editors
Set cEditors = CATIA.Editors

cEditors = CATIA.Editors

Dim cEditors As INFITF.Editors
Set cEditors = CATIA.Editors

cEditors = CATIA.Editors
```

Once you have retrieved such an object, you can use its own properties in turn to retrieve its own aggregated objects. For example, the **Editor** object aggregates a **Selection** object and features a **Selection** property to return it.

Use the **Selection** property to return the **Selection** object from the **Editor** object.

VB VBA C# VB.NET Python

```
Dim oSelection
Set oSelection = CATIA.Selection

Dim oSelection As Selection
Set oSelection = CATIA.Selection

oSelection = CATIA.Selection

Dim oSelection As INFITF.Selection
Set oSelection = CATIA.Selection

oSelection = CATIA.Selection
```

You can also retrieve the **Selection** object from the **Application** object in one shot.

Use the the **ActiveEditor** property to return the active **Editor** object from the **Application** object combined to the **Selection** property to return the **Selection** object from the active **Editor** object.

VB VBA C# VB.NET Python

```
Dim oSelection
Set oSelection = CATIA.ActiveEditor.Selection

Dim oSelection As Selection
Set oSelection = CATIA.ActiveEditor.Selection

oSelection = CATIA.ActiveEditor.Selection

Dim oSelection As INFITF.Selection
Set oSelection = CATIA.ActiveEditor.Selection

oSelection = CATIA.ActiveEditor.Selection
```

You can so go deeper an deeper in the aggregation model.

Conversely, any objects have a **Parent** property that enables you navigate upwards in the Automation model. For example, if you want to retrieve the **Editor** object that aggregates the **Selection** object you handle in a variable such as `oSelection`, use its **Parent** property.

VB VBA C# VB.NET Python

```
Dim oEditor
Set oEditor = oSelection.Parent

Dim oEditor As Editor
Set oEditor = oSelection.Parent

oEditor = oSelection.Parent

Dim oEditor As INFITF.Editor
Set oEditor = oSelection.Parent

oEditor = oSelection.Parent
```

Service

The **Service** abstract object can be retrieved under one its numerous flavors from the **Application** object of from the **Editor** object, depending on the chosen flavor, since some services are nor related to the active editor. For example, the **PLMOpenService** object that opens in the session the objects resulting from a query in the database can be retrieved from the **Application** object, and application related services, such as the **MeasurableService** object that can be retrieved from the **Editor** object.

Many services objects enable you:

- To retrieve objects, such as the **VisuServices** object that retrieves the **Cameras** collection object.
- To create objects, such as the **PLMNewService** object that creates a new PLM entity.
- To execute an action, such as the **DrawingGenService** object that checks the objects integrity pointed by a generative view.

Refer to the Service Object for detailed information.

GetItem

Objects than can be extended use the **GetItem** method to navigate these extensions. For example, the **Part** object can be extended by Knowledge, 2D layout, and Automated Design Process. From the **Part** object, and provided these applications are installed, use the **GetItem** method to retrieve the root object of these applications, form which you can then navigate or create their own objects.

Assume that you currently hold a **Part** object suing the `oPart` variable. You can retrieve the root objects of these applications as follows:

- For Knowledge, the root object is the **KnowledgeObjects** object:
  VB VBA C# VB.NET Python

  ```
  Dim oKnowledgeObjects
  Set oKnowledgeObjects = oPart.GetItem("KnowledgeObjects")

  Dim oKnowledgeObjects As KnowledgeObjects
  Set oKnowledgeObjects = oPart.GetItem("KnowledgeObjects")

  oKnowledgeObjects = oPart.GetItem("KnowledgeObjects")

  Dim oKnowledgeObjects As KnowledgewareTypeLib.KnowledgeObjects
  Set oKnowledgeObjects = oPart.GetItem("KnowledgeObjects")

  oKnowledgeObjects = oPart.GetItem("KnowledgeObjects")
  ```

- For 2D Layout, the root object is the **Layout2DRoot** object:
  VB VBA C# VB.NET Python

  ```
  Dim oLayRoot
  Set oLayRoot = oPart.GetItem("CATLayoutRoot")

  Dim oLayRoot As Layout2DRoot
  Set oLayRoot = oPart.GetItem("CATLayoutRoot")
  ```

```
oLayRoot = oPart.GetItem("CATLayoutRoot")

Dim oLayRoot As LAYOUT2DITF.Layout2DRoot
Set oLayRoot = oPart.GetItem("CATLayoutRoot")

oLayRoot = oPart.GetItem("CATLayoutRoot")
```

- For Automated Design Process, the root object is the **DPCOperations** collection object:
  VB VBA C# VB.NET Python

```
Dim cOperations
Set cOperations = oPart.GetItem("CATGetDPCOperations")

Dim cOperations As DPCOperations
Set cOperations = oPart.GetItem("CATGetDPCOperations")

cOperations = oPart.GetItem("CATGetDPCOperations")

Dim cOperations As DPCOperationsTypeLib.DPCOperations
Set cOperations = oPart.GetItem("CATGetDPCOperations")

cOperations = oPart.GetItem("CATGetDPCOperations")
```

The **GetItem** method is called with a parameter that depends on the application, such as **KnowledgeObjects** for Knowledge, **CATLayoutRoot** for 2D Layout, and **CATGetDPCOperations** for Automated Design Process. Refer to each application to know which objects you can retrieve and the appropriate parameter to pass to the **GetItem** call.

Once you have retrieved the application root object, the Automation object model of this application can be navigated using the object's properties and/or some of the other mechanisms described here.

Selection

The **Selection** object contains the objects currently selected. Thanks to methods of the **Selection** object, navigating this set of objects can be done in several ways:

- You can retrieve an object from its type using the **FindObject** method.
- You can retrieve an object from its rank in the selection using the **Item** method.
- You can search for an object managed by the current **Editor** object and if it is found, put it in the selection using the **Search** method.

Refer to the Selection Object for detailed information.

Navigating a Part

The **Part** object enables you to scan the object model aggregated to it using its **FindObjectByName** method.

For example, to retrieve the sketch, you can use the properties to navigate the **Part** object to the **Sketch** object.

VB VBA C# VB.NET Python

```
Dim oPart
Set oPart = CATIA.ActiveEditor.ActiveObject
Dim oBody
Set oBody = oPart.Bodies.Item(1)
Dim oSketch
Set oSketch = oBody.Sketches.Item(1)

Dim oPart As Part
Set oPart = CATIA.ActiveEditor.ActiveObject
Dim oBody As Body
Set oBody = oPart.Bodies.Item(1)
Dim oSketch As Sketch
Set oSketch = oBody.Sketches.Item(1)

oPart = CATIA.ActiveEditor.ActiveObject
oBody = oPart.Bodies.Item(1)
oSketch = oBody.Sketches.Item(1)
```

```
Dim oPart As MECMOD.Part
Set oPart = CATIA.ActiveEditor.ActiveObject
Dim oBody As MECMOD.Body
Set oBody = oPart.Bodies.Item(1)
Dim oSketch As MECMOD.Sketch
Set oSketch = oBody.Sketches.Item(1)

oPart = CATIA.ActiveEditor.ActiveObject
oBody = oPart.Bodies.Item(1)
oSketch = oBody.Sketches.Item(1)
```

This example first retrieves the **Part** object from the active editor, then **Body** object from the **Bodies** collection object aggregated to the the **Part** object, and then the **Sketch** object from the **Sketches** collection object aggregated to the **Body** object.

But you can also use the **FindObjectByName** method.

VB VBA C# VB.NET Python

```
Dim oPart
Set oPart = CATIA.ActiveEditor.ActiveObject
Dim oSketch
Set oSketch = oPart.FindObjectByName("Sketch.1")

Dim oPart As Part
Set oPart = CATIA.ActiveEditor.ActiveObject
Dim oSketch As Sketch
Set oSketch = oPart.FindObjectByName("Sketch.1")

oPart = CATIA.ActiveEditor.ActiveObject
oSketch = oPart.FindObjectByName("Sketch.1")

Dim oPart As MECMOD.Part
Set oPart = CATIA.ActiveEditor.ActiveObject
Dim oSketch As MECMOD.Sketch
Set oSketch = oPart.FindObjectByName("Sketch.1")

oPart = CATIA.ActiveEditor.ActiveObject
oSketch = oPart.FindObjectByName("Sketch.1")
```

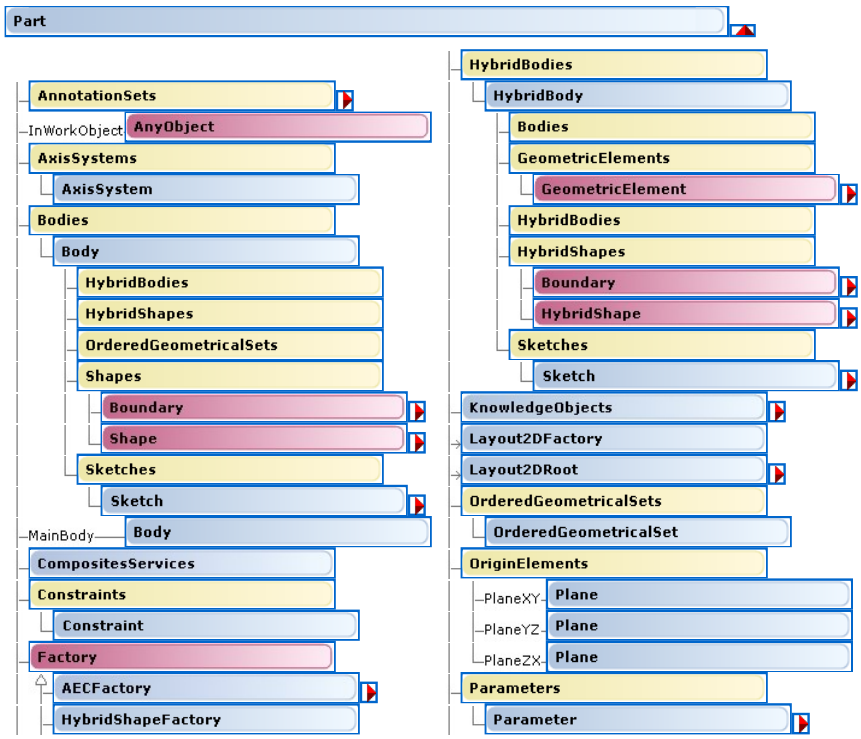
This example also first retrieves the **Part** object from the active editor, and then uses the **FindObjectByName** method to retrieve the **Sketch** object using its name Sketch.1. You can n the same way retrieve the sketch geometry object Point.- using this method.

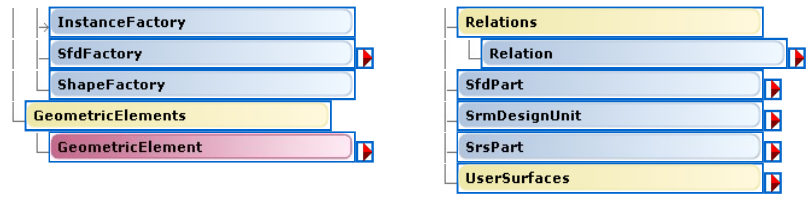
VB VBA C# VB.NET Python

```
Dim oPoint
Set oPoint = oPart.FindObjectByName("Point.6")

Dim oPoint As Point2D
Set oPoint = oPart.FindObjectByName("Point.6")

oPoint = oPart.FindObjectByName("Point.6")

Dim oPoint As MECMOD.Point2D
Set oPoint = oPart.FindObjectByName("Point.6")

oPoint = oPart.FindObjectByName("Point.6")
```

Part Object Model Map

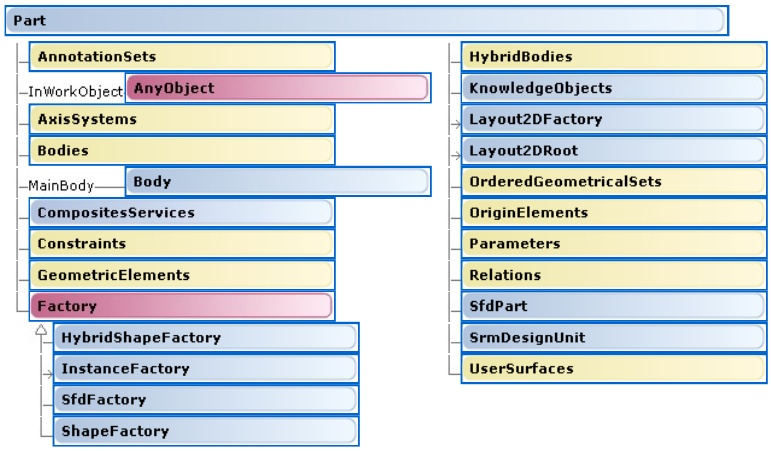
See Also Legend





Part Object

See Also Legend Use Cases Properties Methods



Represents the root object of a 3D shape representation.

The **Part** object is located at the top of the 3D shape representation specification tree. Some of the objects aggregated by the **Part** object are also can also be found in the part specification tree. These objects are:

- The three planes XY, YZ, and ZX you can retrieve from the **OriginElements** collection object using the **Part** object's **OriginElements** property.
- The **AxisSystems** collection object.
- The **OrderedGeometricalSets** collection object.
- The 3D elements you can create as references to position your 3D objects and stored in a **GeometricElements** collection object you can retrieve using the **GeometricElements** property.
- The bodies you create, starting with the main body and stored in the **Bodies** collection object you can retrieve using the **Part** object's **Bodies** property. These bodies contains the part's geometry.
- The hybrid bodies you create, stored in the **HybridBodies** collection object.

In addition, the **Part** object aggregates:

- The **CompositesServices** object you can retrieve thanks to the **GetItem** method with CATCompositesServices as argument.
- The constraints you can set to your 3D objects and stored in the **Constraints** collection object you can retrieve using the **Part** object's **Constraints** property
- The **KnowledgeObjects** object you can retrieve thanks to the **GetItem** method with KnowledgeObjects as argument.
- The parameters stored in a **Parameters** collection object you can retrieve using the **Part** object's **Parameters** property.
- The relations between parameters stored in a **Relations** collection object you can retrieve using the **Part** object's **Relations** property.
- The factories: a **ShapeFactory** object to create shapes, an **HybridShapeFactory** object to create hybrid shapes, and an **InstanceFactory** object to instantiate either User Defined Features or Power Copies. All derive from the **Factory** abstract object.
- The **Layout2DRoot** object and the associated **Layout2DFactory** object.
- The Functional Tolerancing & Annotations objects aggregated under the **AnnotationSets** and **UserSurfaces** collection objects.

The **Bodies** collection object includes **Body** objects, one being the main body returned by the **MainBody** property of the collection object. The **Part** object has in addition an in work **AnyObject** object returned or set using the **InWorkObject** property. The in work object is the object in which a new shape is added when using a factory, such as the **ShapeFactory** object. You need to make in work the appropriate object before using a factory.

The other collection objects can be classified in two categories:

1. The collection objects that only contains objects and have methods to retrieve and possibly remove them, but leave the dedicated factories for object creation, such as the **Sketches**, **GeometricElements**, and **Shapes** collection objects.
2. The collection objects that also have methods to create the objects they contain, such as **Constraints**, **Relations**, and **Parameters** collection objects.

Retrieving the Part Object

Use the **ActiveObject** property of the **Editor** object to return the **Part** object. The **Editor** object is returned from the **Application** object using the **ActiveEditor** property.

```
Dim oPart As Part
Set oPart = CATIA.ActiveEditor.ActiveObject
```

Using the Part Object

Returning a Collection Object

Use the property the name of which is the collection object name. For example, use the **Constraints** property to return the **Constraints** collection object.

```
Dim cConstraints As Constraints
Set cConstraints = oPart.Constraints
```

Returning the Factories

Use the **ShapeFactory** property to return the **ShapeFactory** object.

```
Dim oShapeFactory As ShapeFactory
Set oShapeFactory = oPart.ShapeFactory
```

Use the **HybridShapeFactory** property to return the **HybridShapeFactory** object.

```
Dim oHybriShapeFactory As HybriShapeFactory
Set oHybriShapeFactory = oPart.HybriShapeFactory
```

Use the **GetCustomerFactory** method to return the **InstanceFactory** object.

```
Dim oInstanceFactory As InstanceFactory
Set oInstanceFactory = oPart.GetCustomerFactory("InstanceFactory")
```

You can retrieve any customer factory in the same way, provided you know the factory identifying character string to pass to the **GetCustomerFactory** method.

Setting the In Work Object

Use the **InWorkObject** property to set the in work object.

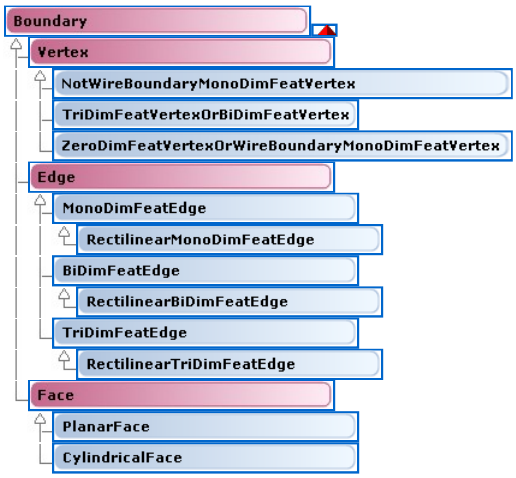
```
Dim oInWorkObject As AnyObject
Set oInWorkObject = ...
oPart.InWorkObject = oInWorkObject
```

See Also

- Properties
- Methods
- Use Cases.

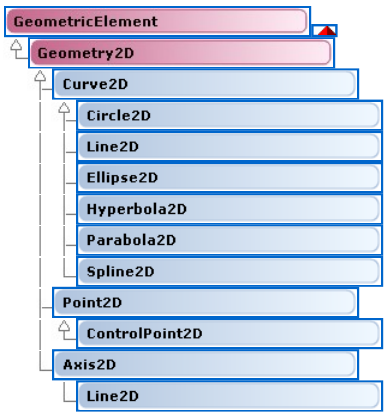
Boundary Object Model Map

See Also Legend



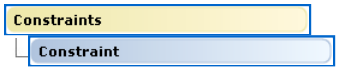
Geometric Element Object Model Map

See Also Legend



Constraints Collection Object

See Also Legend Use Cases Properties Methods



The **Constraints** collection object provides methods to create constraints. A **Constraint** object can constrain one, two, or three elements. Broken and non updated constraints can be retrieved from the collection, as well as the number of constraints that fall into these categories in the collection. A given constraint is defined using properties to set its type, mode, side, and orientation, its dimension and configuration, its reference axis, and its status. The type of a constraint defines its nature, that is how the constraint works, and how many elements can be involved in defining the constraint. The constraint status indicates if the constraint is valid, if something goes wrong with the constraint, or if the constraint is broken. Other properties make sense for certain types of constraints only. The constraint visualization location can be retrieved and set in the 3D space using the two methods **GetConstraintVisuLocation** and **SetConstraintVisuLocation** respectively.

To retrieve the **Constraints** collection object, use the `Constraints` property of the **Part** object. Assuming `oPart` is the **Part** object currently active, write:

```
Dim oPart As Part
Set oPart = CATIA.ActiveEditor.ActiveObject
Dim cConstraints As Constraints
Set cConstraints = oPart.Constraints
```

Opening 3DShape

This use case primarily focuses on the methodology to open a 3DShape.

Before you begin: Note that:

- You should first launch CATIA

Where to find the macro: CAAScdMmrUcOpening3DShapeSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

Related Topics
Launching an Automation Use Case

This use case can be divided in 4 steps

1. Retrieves the search service from CATIA session
2. Searches for a 3DShape, corresponding to the user input criteria
3. Opens a 3DShape
4. Retrieves its Part Object

1. **Retrieves the search service from CATIA session**

```
...
 Dim oSearchService As SearchService
 Set oSearchService = CATIA.GetSessionService("Search")
...
```

A call to the Application (CATIA) *GetSessionService* method returns the specified Service, SearchService, `oSearchService` in this case.

2. **Searches for a 3DShape, corresponding to the user input criteria**

```
...
Dim oDBSearch As DatabaseSearch
Set oDBSearch = Search3DShape(oSearchService)
...
```

The function **`Search3DShape`** returns **`oDBSearch`**, a DatabaseSearch object. The function details are as in the below sub steps.

i. Retrives the DatabaseSearch property from the Search Service

```
...
 Dim oDBSearch As DatabaseSearch
 Set oDBSearch = oSearchService.DatabaseSearch
...
```

ii. Sets the type of objects to search for

```
Function Search3DShape(oSearchService, oDBSearch) As DatabaseSearch

    oDBSearch.BaseType = strTheProductRepresentationReferenceType
    ...
```

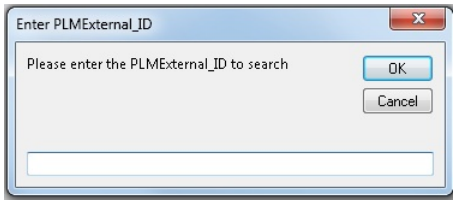
A call to the `BaseType` property sets the type of objects to search for. `strTheProductRepresentationReferenceType` is a global variable representing the Product Representation Reference type.

iii. Updates the DatabaseSearch object with the attribute criteria provided by the user as an input

```
...
    oDBSearch.AddEasyCriteria "PLM_ExternalID", strInputPLMIDName
...
```

A call to *AddEasyCriteria* method updates the DatabaseSearch object with the search criteria according to the users input as depicted in the figure below

Fig.1: User Input for Searching 3DShape from Database



In this case, we prompt the user to provide a PLM_ExternalID value

iv. Triggers the search

```
...
    oSearchService.Search
...
End Function
```

A call to *Search* method of the SearchService object actually searches for the objects which matches all the attributes of the set and matching the case of the values(i.e. search is Case Sensitive), and type.

3. **Opens a 3DShape**

As a next step, the UC opens the first PLM Entity from the collection occurring in the new tab page within Search Editor retrieved in the last step.

```
...
   Dim oEditor3DShape As Editor
   Set oEditor3DShape = Open3DShape(oDBSearch)
...
```

The function `Open3DShape` takes a DatabaseSearch, `oDBSearch`, as its input. This argument was output by the Search on the underlying database, which occurred in the previous step.

The function `Open3DShape` is detailed as in the below sub steps.

i. Retrieves the root entities from the new tab page within the Search Editor

```
Function Open3DShape(oDBSearch) As Editor
    Dim o3DShapeAsPLMEntities As PLMEntities
    Set o3DShapeAsPLMEntities = oDBSearch.Results
...
```

A call to the *Results* method of DatabaseSearch object, `oDBSearch`, returns a collection of root PLM Entities in the active (Search) editor.

ii. Retrieves a PLM Entity object from its index

```
...
    Dim o3DShapeAsPLMEntity As PLMEntity
    Set o3DShapeAsPLMEntity = o3DShapeAsPLMEntities.Item(1)
...
```

The first entity in the above collection is retrieved, thanks to the *Item* method on PLMEntities.

iii. Retrieves the Open Service from CATIA Session

```
...
    Dim oOpenService As PLMOpenService
    Set oOpenService = CATIA.GetSessionService("PLMOpenService")
...
```

A call to the Application (CATIA) *GetSessionService* method returns the specified Service, PLMOpenService, `oOpenService` in this case.

iv. Opens in the authoring session the first element occurring in the new tab page within Search Editor (a 3D Shape)

```
...
    Dim oEditor3DShape As Editor
    oOpenService.PLMOpen o3DShapeAsPLMEntity , oEditor3DShape
...
End Function
```

The PLM Entity, `o3DShapeAsPLMEntity` retrieved in the above steps is then loaded in the current session with a call to the *PLMOpen* method of the PLMOpenService. The editor associated with the loaded 3DShape in the current session, is finally returned by the *PLMOpen* call in a 3D Shape Editor, `oEditor3DShape`. The PLMEntity, `o3DShapeAsPLMEntity`, is loaded in EDIT mode

4. **Retrieves its Part Object**

A 3DShape is now loaded in session.

```
...
 Dim oPart As Part
 Set oPart = oEditor3DShape.ActiveObject
...
```

A 3DShape is now loaded in the current session. The *ActiveObject* method of 3D Shape Editor, `oEditor3DShape` returns the Part Object, `oPart`.

A Part Object, is the root element of a 3DShape representation, which aggregates the features (Part Body (seen in the figure below, Fig. 2), Hybrid Body, Ordered Geometrical Set etc.) under it. The Part object, being the root entity, it occurs at the top of the 3D Shape Representation specification tree.

Fig. 2: Opened 3D Shape

The opened 3DShape is thus seen in CATIA as above

Creating 3DShape

This use case creates a new 3DShape and retrieves its associated objects: its root object, the Part object, and its Product Representation Reference, the VPMRepReference object.

Related Topics
Part Object Model Map
Editor Object

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdPstUcCreating3DShapeSource.htm

This use case is divided in 5 steps

1. Retrieves the service object to create a new PLM object
2. Creates a new 3DShape
3. Retrieves its Part Object
4. Retrieves its Product Representation Reference
5. Displays the Product Representation Reference PLM_ExternalID value

1. **Retrieves the service object to create a new PLM object**

```
 ...
Dim oNewService As PLMNewService
Set oNewService = CATIA.GetSessionService("PLMNewService")
...
```

Returns a PLMNewService object, `oNewService` from the Application object using the *GetSessionService* method.

2. **Creates a new 3DShape**

```
...
Dim oEditor3DShape As Editor
oNewService.PLMCreate "3DShape", oEditor3DShape
...
```

A call to *PLMCreate* on `oNewService` creates and edits the 3DShape in a 3D Shape editor, `oEditor3DShape`.

3. **Retrieves its Part Object**

```
...
 Dim  oPart3D  As  Part
 Set  oPart3D = oEditor3DShape.ActiveObject
...
```

The *ActiveObject* method of 3D Shape Editor, `oEditor3DShape` returns the Part Object, `oPart3D`;

4. **Retrieves its Product Representation Reference**

```
...
 Dim  oVPMRepRef  As  VPMRepReference
 Set  oVPMRepRef = oPart3D.Parent
...
```

The `Parent` property on `oPart3D` returns `oVPMRepRef`, a VPMRepReferance type.

5. **Displays the Product Representation Reference PLM_ExternalID value**

```
...
 MsgBox oVPMRepRef.GetAttributeValue("PLM_ExternalID")
...
```

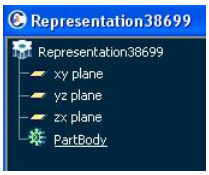
A call to *GetAttributeValue* on `oVPMRepRef` returns the PLM_ExternalID attribute value of the Representation Reference as shown in the Fig.1.

Fig.1: Product RepRef PLM_ExternalID

You can observe in Fig.2 that the PLM_ExternalID value appears at the top of the Specifiction tree. This top object is the Part object. Its name, when it is created, is valuated with the PLM_ExternalID value of the Product Representation Reference.

Fig.2: The 3DShape as

Viewed in CATIA



Retrieving the 3DShape in the Session

This use case fundamentally illustrates retriving of the Product 3DShapes from current session.

Related Topics
Service Object
Part Object Model Map

Before you begin: Note that:

- Launch CATIA
- It is recommended that at least one 3DShape loaded in session to see result

Where to find the macro: CAAScdMmrUcRetrievingThe3DShapeInTheSessionSource.htm

This use case is divided in 5 steps

1. Retrieves session service related to Product data
2. Retrieves the List of 3DShape in session
3. Displays the list of each of them in a dialog box - to lets the end user select one
4. Retrieve its Part feature for selected 3DShape
5. Creates the Pad

1. **Retrieves session service related to Product data**

```
 ...
Dim oProductSessionService As ProductSessionService
Set oProductSessionService = CATIA.GetSessionService("ProductSessionService")
...
```

Returns a ProductSessionService object, `oProductSessionService` from the Application object using the *GetSessionService* method.

2. **Retrieves the List of 3DShape in session**

```
...
Dim oShape3Ds As Shape3Ds
Set oShape3Ds = oProductSessionService.Shape3Ds
...
```

A call to *Shape3Ds* on `oProductSessionService` returns list of the 3DShape objects loaded in session, `oShape3Ds`.

3. **Displays the list of each of them in a dialog box - to lets the end user select one**

```
...
For i = 1 To oShape3Ds.Count
     Dim oShape3D  As   Shape3D
     Set oShape3D = oShape3Ds.Item(i)

    strBrowsedPLMCompIDAttr = strBrowsedPLMCompIDAttr + oShape3D.Name + " " + vbCrLf
 Next i

  Dim iInput3DshapeIndex  As  Integer
 iInput3DshapeIndex = InputBox("Please enter the index number of 3dshape to create pad", strBrowsedPLMCompIDAttr)

  Dim oShape3DSelected  As  Shape3D
  Set oShape3DSelected= oShape3Ds.Item(iInput3DshapeIndex)
...
```

The *Item* method of 3D Shape list (Shape3Ds), `oShape3Ds` returns the 3Dshape object according to item index.

4. **Retrieve its Part feature for selected 3DShape**

```
...
 Dim  part1  As  Part
 Set  part1 = oShape3DSelected.GetItem("Part")
...
```

The `GetItem` method on `oShape3DSelected` returns `part1`, a Part type.

5. **Creates the Pad**

```
...
    CreatesPad part1
...
```

A call to *CreatesPad* creates a Pad in the Part,(part1) .

Modifying Visualization Properties of Selected Mechanical Objects

This use case we change a Visualization Property (in this case, Color) of a selected Mechanical Object.

Related Topics
Editor Object
Foundation Object Model Map

Before you begin: Note that:

- Launch CATIA

- Create a New 3DShape with some Mechanical Object or Open an existing one

Where to find the macro: CAAScdMmrUcModifyVisuPropertySource.htm

This use case is divided in four steps:

1. Retrieves the Active Editor in CATIA
2. Retrieves Selection Object from Active Editor
3. Updates Selection Object with Selection Criteria
4. Changes the Visualization Property Color

1. **Retrieves the Active Editor in CATIA**

```
...
Dim  oActiveEditor  As   Editor
Set  oActiveEditor = CATIA.ActiveEditor
...
```

This step returns an Editor object, `oActiveEditor` from CATIA using the *ActiveEditor* method.

2. **Retrieves Selection Object from Active Editor**

```
...
Dim  oSelection
Set  oSelection = oActiveEditor.Selection
...
```

The `Selection` property of Editor object, `oActiveEditor` returns Selection object, `oSelection`

3. **Updates Selection Object with Selection Criteria**

In this use case we give no condition for selection. It means we provide "AnyObject" as a filter type. "AnyObject" is a base class for all Automation types of entities which could be selected in the spec tree within CATIA.

```
...
Dim  Status  As   String
Dim  InputObjectType(0)
InputObjectType(0) = "AnyObject"
Status = oSelection.SelectElement(InputObjectType, "Select a Element from spec tree", False)
...
```

A call to *SelectElement* method updates the filter criteria . Here we give any object type, as an input criteria. Since the last argument is false, the end user will always be invited to select something in the spec tree, or the 3D Viewer, though an entity (of the type set as filter in the selection criteria) has already been selected by the end-user.

4. **Changes the Visualization Property Color**

Here we first retrieve Selection object's property set. We then set their color

```
...
Dim  VisPropertySet
Set   VisPropertySet = oSelection.VisProperties
VisPropertySet.SetRealColor 0, 255, 0, 1
...
```

`VisPropertySet` represents the visualization property set of the selected object. We change this property using *SetRealColor* method.

Creating Ordered Geometrical Sets in a 3DShape

This use case fundamentally illustrates creating Ordered Geometrical Sets in a 3DShape.

Before you begin: Note that:

- Launch CATIA

Related Topics
Part Object Model Map

Where to find the macro: CAAScdGsiUcCreateOGSSource.htm

This use case can be divided in 7 steps

1. Creates a 3DShape
2. Retrieves its Part Object
3. Retrieves its Ordered Geometrical Set collection
4. Creates an Ordered Geometrical Set
5. Retrieves its own Ordered Geometrical Set Collection
6. Creates an Ordered Geometrical Set beneath the New One
7. Updates Part Object

1. **Creates a 3DShape**

As a first step, this UC creates a 3DShape thanks to the PLMNewService service in CATIA.

```
...
    Dim oPLMNewService As PLMNewService
    Set oPLMNewService = CATIA.GetSessionService("PLMNewService")

    Dim oEditor3DShape As Editor
    oPLMNewService.PLMCreate "3DShape", oEditor3DShape
...
```

The 3DShape is created and edited through the 3D Shape editor , `oEditor3DShape`. For further information on Creating 3DShape, Please refer to [1]

2. **Retrieves its Part Object**

```
...
    Dim oPart As Part
    Set oPart = oEditor3DShape.ActiveObject
...
```

The *ActiveObject* method of3D Shape Editor, `oEditor3DShape` returns the Part Object, `oPart`

3. **Retrieves its Ordered Geometrical Set collection**

   Further, we retrieve the Ordered Geometrical Sets Collection from the Part Object.

```
...
    Dim oOrderedGeometricalSets As OrderedGeometricalSets
    Set oOrderedGeometricalSets = oPart.OrderedGeometricalSets
...
```

   The *OrderedGeometricalSets* method of the Part object, returns the OrderedGeometricalSets object , which is the collection of OrderedGeometricalSet objects aggregated by the Part object, `oPart`

4. **Creates an Ordered Geometrical Set**

   In this step, we create and add the OGS to the Ordered Geometrical Sets Collection.

```
...
    Dim oOrderedGeometricalSet As OrderedGeometricalSet
    Set oOrderedGeometricalSet = oOrderedGeometricalSets.Add
...
```

   The *Add* method of the OrderedGeometricalSets object, creates a new OGS, `oOrderedGeometricalSet` and adds it to the OrderedGeometricalSets collection, `oOrderedGeometricalSets`

5. **Retrieves its own Ordered Geometrical Set Collection**

   In this step, we retrieve the Ordered Geometrical Set's OrderedGeometricalSets collection.

```
...
    Dim oOrderedGeometricalSetsBeneathOtherOGS As OrderedGeometricalSets
    Set oOrderedGeometricalSetsBeneathOtherOGS = oOrderedGeometricalSet.OrderedGeometricalSets
...
```

   The *OrderedGeometricalSets* method returns in `oOrderedGeometricalSetsBeneathOtherOGS` the collection of OrderedGeometricalSet aggregated by the OrderedGeometricalSet, `oOrderedGeometricalSet`

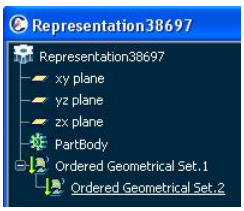
6. **Creates an Ordered Geometrical Set beneath the New One**

   In this step, we create and add the OGS to other Ordered Geometrical Sets Collection retrieved in the earlier step

```
...
    Dim oOrderedGeometricalSetBeneathOtherOGS As OrderedGeometricalSet
    Set oOrderedGeometricalSetBeneathOtherOGS = oOrderedGeometricalSetsBeneathOtherOGS.Add
...
```

   The *Add* method of the OrderedGeometricalSets object, creates a new OrderedGeometricalSet, `oOrderedGeometricalSetBeneathOtherOGS` and adds it to the OrderedGeometricalSets collection, `oOrderedGeometricalSetsBeneathOtherOGS`. As shown in below figure, Fig.1

   Fig. 1 OGS Added in the Specification Tree within CATIA

7. **Updates Part Object**

```
...
    oPart.Update
End Sub
```

   Updates the Part Object, `oPart` result with respect to its specifications.

References

[1] Creating 3DShape

Creating a Pad based on a Sketch

This use case creates a Pad based on the Sketch. In the process we learn to retrieve Part object from Body and retrieve Sketch aggregated beneath body by giving sketch name as an input.

Related Topics
Part Object
Model Map
Editor Object

Before you begin: Note that:

- You should first launch CATIA
- The macro should essentially be launched from a Part Editor (meaning input data is always a PLM Product Representation Reference)

- Product/Part editor must be active before launching the macro

Where to find the macro: CAAScdMmrUcCreatePadBasedOnSketchSource.htm

1. Prolog

   The macro begins with the **CreatePadBasedOnSketch** main subroutine. It typically begins with the following extract of code, which serves as a built-in "error handling" mechanism.

   ```
   Sub CreatePadBasedOnSketch()

   On Error GoTo ErrorSub
   ...
   ```

   The first instruction means that when a method will throw an error, the macro will skip to the line named, **ErrorSub**. See the step Epilog.

   `CreatePadBasedOnSketch` subroutine selects Body Object from specification tree or 3d viewer within CATIA. Then UC retrieves the Part object from of Body which is parent of it. Next we retrieve aggregated sketch by giving sketch name as an input. Then create Pad inside the Part using retrieved sketch and limit value given by user.

2. Here, we retrieve the current active Editor object to select PLM object

   ```
    ...
   Dim oCurrentActiveEditor As Editor
   Set oCurrentActiveEditor = CATIA.ActiveEditor
   ...
   ```

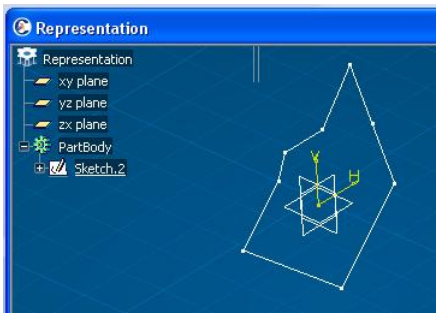
   Return a **Editor** object from the Application object (CATIA) using the **ActiveEditor** method.

3. We then retrieve Selection object from active editor

   ```
   ...
   Dim oObjSelection
   Set oObjSelection = oCurrentActiveEditor.Selection
   ...
   ```

   Return a **Selection** object from the Active Editor using **Selection** method. It is significant to note here that we haven't declared a type for `oObjSelection` (as for several other variables, further ahead in this macro). The section "Tips about VB and VBA" in the technical article "About VB, VBA, Debug, and Compatibility" [] provides an explanation for why these variables are not typed.

4. The selection object retrieved above is now updated with the selection criteria ("Body"). The following image shows sample Object from which user could select Body object (PartBody).


   ```
   ...
   Dim strStatus As String
   Dim oInputObjectType(0)

   oInputObjectType(0) = "Body"
   strStatus = oObjSelection.SelectElement(oInputObjectType, "Select a Body Element from spec tree or the 3D Viewer", False)
   ...
   ```

   The PLM objects that you can select (if the current editor is VPM Editor) is ***Body*** type. This type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

   At this stage, the application prompts an end user to select a Body from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

   The last argument is false, indicating the end user will always be invited to select something in the spec tree or the 3D Viewer, though an entity (of the type set as filter in the selection criteria) has already been selected by the end-user.

   Once the user does the selection, the application proceeds further.

5. Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Body***) from the Selection Object which contains it.

   ```
   ...
   Dim oSelectedElement As SelectedElement
   Set oSelectedElement  =  oObjSelection.Item(1)

   Dim oBody As Body
   Set oBody = oSelectedElement.Value

   MsgBox ("Selected Body Name :") + oBody.Name
   ...
   ```

   A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

   Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oBody`, a *Body* type representing the selection.

Then we display the Body object name using `Body::Name`.

6. Next retrieve the Part object aggregating selected Body. Further we use the Part to create Pad.

   first we retrieve **Bodies** object from Body object. Then from Bodies we retrieve its Parent that is **Part**.

```
...
Dim oBodies As Bodies
Set oBodies = oBody.Parent

Dim oPart As Part
Set oPart = oBodies.Parent

MsgBox ("The Part name aggrigating selected Body : ") + oPart.Name
..
```

   A call to `Body::Parent` , on `oBody` returns the Parent entity, as a *Bodies* type (`oBodies`).

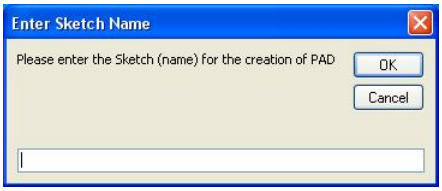
   Next a call to `Bodies::Parent` on `oBodies` (retrieved above) returns its Parent `oPart`, a *Part* type.

   Then we display the Part object name using `Part::Name`.

7. Next retrieve the Sketch from Body.

   Prompt the user to enter Sketch name for creating Pad.

```
...
Dim strInputSketchName  As String
strInputSketchName = InputBox("Please enter the Sketch (name) for the creation of PAD", "Enter Sketch Name")
...
```



   Here we retrieve the Sketches this contains the the list of all Sketches beneath selected Body object. Then we retrieve sketch by its name from list of sketches.

```
...
Dim oSketch As Sketch
Set oSketch = oBody.Sketches.Item(strInputSketchName)
...
```

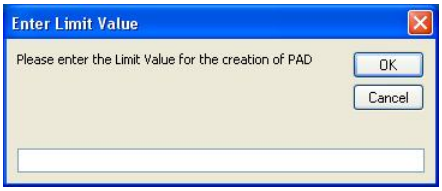
   A call to `Body::Sketches`, on `oBody` returns the list of sketches listed beneath body object, as a *Sketches* type .

   Next a call to `Sketches::Item` on `Sketches type` (retrieved above) by giving sketch name as input (`strInputSketchName`) returns `oSketch`, a *Sketch* type.

8. Next Create the Pad using Sketch retrieved above and Limit value entered by user.

   Prompt the user to enter Limit Value for creating Pad.

```
...
Dim iPadLimit  As Integer
iPadLimit = InputBox("Please enter the Limit Value for the creation of PAD", "Enter Limit Value")
...
```



   Here we create pad by giving sketch object and Limit value as an input.

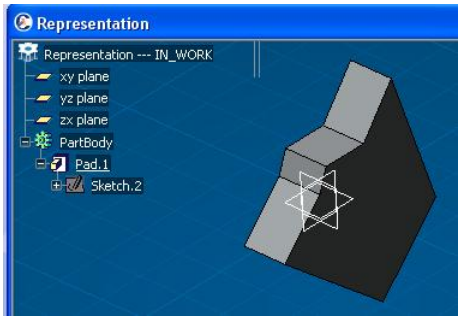
```
...
Dim oPad As Pad
Set oPad = oPart.ShapeFactory.AddNewPad(oSketch, iPadLimit)

oPart.Update
...
```

A call to `Part::ShapeFactory`, on `oPart` returns the part shape factory. It allows the creation of shapes in the part.

Next a call to `Factory::AddNewPad` on `(oPart.ShapeFactory)Factory type` (retrieved above) creates a pad and returns `oPad` , a *Pad* type.

then finally we update the Part using call to `Part::Update` on `oPart`.

Following image shows the created pad on Sketch.2 with input limit value.



9. We then save the changes.

```
...
CATIA.GetSessionService("PLMPropagateService").Save
...
```

We next save this product in the database. It is done with calls which occur in the following sequence

- A call to `Application::GetSessionService` on CATIA (an Application object, defined internally by VB) returns a *PLMPropagateService* object.
- The next call to `PLMPropagateService::Save` eventually saves this new product hierarchy in the database.

10. Epilog

The following extract of code is primarily meant for error-handling purpose. Any run time error that the macro encounters results in the execution flow reaching this part of the code, and then terminating with a normal exit from subroutine scope.

```
...
    GoTo EndSub

ErrorSub:
    MsgBox Err.Description

EndSub:
End Sub
```

Creating and Displaying New Axis System

This use case fundamentally illustrates about creating new Axis System in a 3DShape.

Before you begin: Note that:

- Launch CATIA.

Where to find the macro: CAAScdMmrUcCreateAndDisplayNewAxisSystemSource.htm

Related Topics
Part Object Model Map

This use case can be divided in eight steps:

1. Creates a 3DShape
2. Retrieves its Part Object
3. Retrieves its Axis System Collection
4. Creates a new Axis system
5. Sets the type of Axis system as per User's input
6. Retrieves and Displays Type of Axis System
7. Retrieves and Displays Origin co-ordinates of Axis System
8. Updates Part Object

1. **Creates a 3DShape**

As a first step, this UC creates a 3DShape thanks to the PLMNewService service in CATIA.

```
...
    Dim oPLMNewService As PLMNewService
    Set oPLMNewService = CATIA.GetSessionService("PLMNewService")

    Dim oEditor3DShape As Editor
    oPLMNewService.PLMCreate "3DShape", oEditor3DShape
...
```

The 3DShape is created and edited through the 3D Shape editor , `oEditor3DShape`. For further information on Creating 3DShape, Please refer to [1]

2. **Retrieves its Part Object**

```
...
    Dim oPart As Part
    Set oPart = oEditor3DShape.ActiveObject
...
```

The *ActiveObject* method of 3D Shape Editor, `oEditor3DShape` returns the Part Object, `oPart`

3. **Retrieves its Axis System Collection**

```
...
    Dim oAxisSystems As AxisSystems
    Set oAxisSystems = oPart.AxisSystems
..
```

A call to *AxisSystems* method on Part returns the Axis System Collection object ,`oAxisSystems` containing the coordinate systems aggregated under the Pa

4. **Creates a new Axis system**

```
...
    Dim oAxisSystem
    Set oAxisSystem = oAxisSystems.Add()
...
```

A call to *Add* method on the Axis System Collection creates a new Axis System and adds it to the Axis Systems Collection, `oAxisSystems`

5. **Sets the type of Axis system as per User's input**

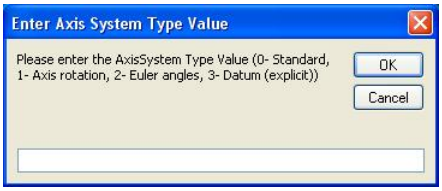
We prompt the user to give input value (0/1/2/3) for setting type of the newly created Axis System.

- Standard = 0 Specifies that the axis system is defined by an origin point and three axes.
- AxisRotation = 1 Specifies that the axis system is defined by an origin point, and a rotation around one axis
- EulerAngles = 2 Specifies that the axis system is defined by an origin point, and the three Euler angles
- Explicit = 3 Specifies that the axis system is a datum

It is interesting to note here that if the Type is not specified, the Axis System is created with the Standard(0) Type. Also the user can change the Axis Syster

```
...
    Dim iAxisSystemType As Integer
    iAxisSystemType = InputBox("Please enter the AxisSystem Type Value (0- Standard, 1- Axis rotation, 2- Euler angles, 3- Da
...
```

Fig. 1: Prompting User to fill in Axis System Type




```
...
    oAxisSystem.Type = iAxisSystemType
...
```

The `Type` property on Axis System, `oAxisSystem` sets the type of Axis system as per the user input value.

6. **Retrieves and Displays Type of Axis System**

In this step, we retrieve Axis system type value (Enumerated value). This should be the same value which we have set in earlier step to Axis system object. .

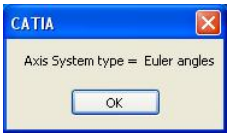
please note that `Type` property is used for both purposes.

```
...
    Dim oType
    oType = oAxisSystem.Type

    If (0 = oType) Then
        MsgBox "Axis System type =  Standard"

    ElseIf (1 = oType) Then
        MsgBox "Axis System type =  Axis rotation"

    ElseIf (2 = oType) Then
        MsgBox "Axis System type =  Euler angles"

    Else
        MsgBox "Axis System type = Datum (explicit)"

    End If
...
```

The `Type` property on Axis System, `oAxisSystem` returns the type of the Axis System.

The following fig shows the type of Axis system. This type is retrieved when user has given input value as 2 while creation.

Fig. 2: Displaying Axis System Type



7. **Retrieves and Displays Origin co-ordinates of Axis System**

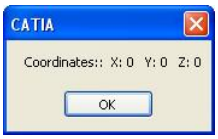
```
...
Dim originCoord(2)
oAxisSystem.GetOrigin originCoord

Dim X
```

```
  X = originCoord(0)
Dim Y
  Y = originCoord(1)
Dim Z
  Z = originCoord(0)

MsgBox " Origin X co-ordinate = " & X
MsgBox " Origin Y co-ordinate = " & Y
MsgBox " Origin Z co-ordinate = " & Z
...
```

A call to *GetOrigin* method on Axis System returns the coordinates X,Y,Z of the origin point of the Axis System, `oAxisSystem` .

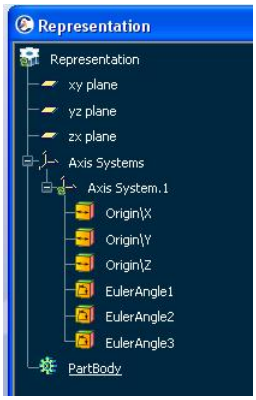
Then we display each vale of co-ordinate (X,Y,Z)

Fig. 3: Displaying X, Y and Z Coordinate values



The specification (spec) tree in CATIA after adding the Axis System looks like the figure below

Fig. 4: CATIA Spec tree after adding the Axis System to the 3DShape



8. **Updates Part Object**

```
...
    oPart.Update
End Sub
```

Updates the Part Object, `oPart` result with respect to its specifications

References

[1] Creating 3DShape

Working with Face and Edge

This use case fundamentally illustrates about working with Face and Edge. In this UC we learn how to retrieve Faces and Edge and create Fillets using it.

Related Topics
Selection Object
Part Object Model Map

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdMmrUcWorkingWithFaceAndEdgeSource.htm

This use case can be divided in 3 steps

1. Creates a 3D Shape with Faces and Edges
2. Creates face to face Fillet with constant radius
3. Creates Edge Fillet with constant radius

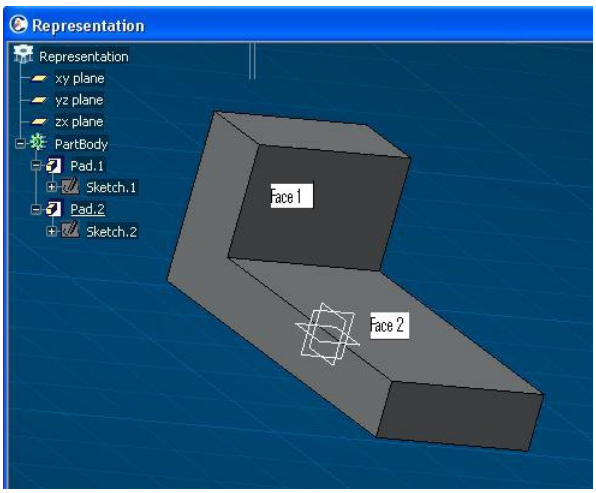
1. Creates a 3D Shape with Faces and Edges

As a first step, this UC creates the geometry with a 2D Sketch which is extruded to a Pad creating Faces and Edges

```
...
    Dim oPart As Part
    CreatingGeometry oPart
...
```

A call to *CreatingGeometry* function creates a 3DShape, retrieves its Part Object, `oPart`, creates a sketch with lines to form a bounded area to it and extrudes it to a Pad, forming Faces and Edges as depicted in the below figure, Fig1.

Fig.1 Created Geometry with Faces and Edges



We thus create the Geometry and we now proceed to selecting the faces and creating a Fillet between them.

2. Creates face to face Fillet

The Use Case prompts to user to select two faces one by one. Then from selection object Use Case retrieves Face objects. Next from two selected face object Use Case creates Face to Face fillet. For better understanding we divide this step in sub steps

I. Selects and retrieves first Face

Now we retrieve Selection object from active editor

```
...
Dim oObjSelection
Set oObjSelection = CATIA.ActiveEditor.Selection
...
```

A call to *Selection* Property of the Active Editor returns `oObjSelection`, a *Selection* object. It is significant to note here that we haven't declared a type for `oObjSelection` (as for several other variables, further ahead in this macro). The section "Virtual Function Table Compatibility" in the technical article "About Automation Languages, Debug, and Compatibility" [1] provides an explanation for why these variables are not typed.

Then prompt the user to select the first Face, a Face type object

```
...
oInputObjectType(0) = "Face"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the first face", False)
...
```

"***Face***" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

At this stage, the application prompts an end user to select a *Face* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Face***) from the Selection Object which contains it.

```
...
Dim oFirstFace As Face
Set oFirstFace = oObjSelection.Item(1).Value

MsgBox ("Selected First Face Name : ") + oFirstFace.Name
...
```

A call to *Item* method on `oObjSelection` retrieves `oFirstFace` object of the `Face`.

Then we display the `Face` object name using *Name* property of `Face` object, `oFirstFace` .

II. Selects and retrieves Second Face

Similarly next we select the second Face. Before new selection we need to clear the selection object.

```
...
oObjSelection.Clear
..
```

A *Clear* method of `Selection` object, `oObjSelection` clears the previously selected object (`Face` object) from this `Selection` object.

Then prompt the user to select the Second Face, a Face type object

```
...
Status = oObjSelection.SelectElement(oInputObjectType, "Select the second face", False)
...
```

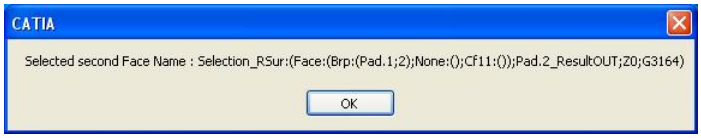
At this stage, the application prompts an end user to select a *Face* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Face***) from the Selection Object which contains it.

```
...
Dim oSecondFace As Face
Set oSecondFace = oObjSelection.Item(1).Value

MsgBox ("Selected second Face Name : ") + oSecondFace.Name
...
```

A call to *Item* method on `oObjSelection` retrieves `oSecondFace` object of the `Face`.

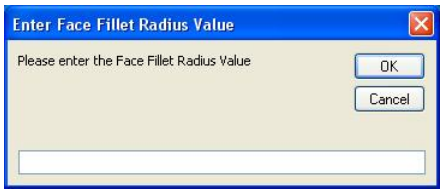
Then we display the `Face` object name using *Name* property of `Face` object, `oSecondFace` .



III. Creates face to face Fillet

Firstly we prompt the user to input the radius value for face fillet.

```
...
Dim dFaceFilletRadius As Double
dFaceFilletRadius = InputBox("Please enter the Face Fillet Radius Value  ", "Enter Face Fillet Radius Value")
...
```



Now we create the new face fillet based on two selected faces and input radius value.

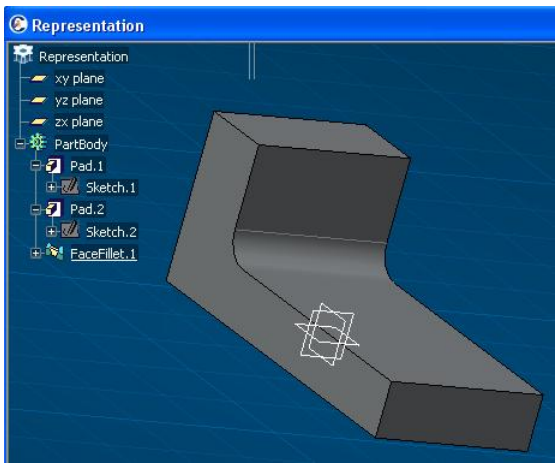
```
...
Dim oFaceFillet As FaceFillet
Set oFaceFillet = oPart.ShapeFactory.AddNewSolidFaceFillet(oFirstFace, oSecondFace, dFaceFilletRadius)

oPart.Update
...
```

A call to `ShapeFactory` property of Part on `oPart` returns the part shape factory object.

Next a call to `AddNewSolidFaceFillet` method of ShapeFactory creates a Face to Face fillet between input faces of solid with constant radius (`oFirstFace`, `oSecondFace`) with input radius (`dFaceFilletRadius`).

Then we update the Part object (`oPart`) using call *Update* method.

Fig.2 Face to Face Fillet



Above [Fig.2] shows the created face to face fillet.

3. Creates Edge Fillet

Now Use Case prompts to user for selection of Edge then creates Edge fillet with constant radius value. For better understanding we divide this step in sub steps

I. Selects and retrieves Edge

we select the Edge fillet to create Edge Fillet. Before new selection we need to clear the selection object.

```
...
oObjSelection.Clear
..
```

A *Clear* method of `Selection` object, `oObjSelection` clears the previously selected object (`Face` object) from this `Selection` object.

Then prompt the user to select the Edge type object

```
...
oInputObjectType(0) = "Edge"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the Edge", False)
...
```

"***Edge***" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

At this stage, the application prompts an end user to select a *Edge* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message).

Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Edge***) from the Selection Object which contains it.

```
...
Dim oEdge As Edge
Set oEdge = oObjSelection.Item(1).Value
...
```

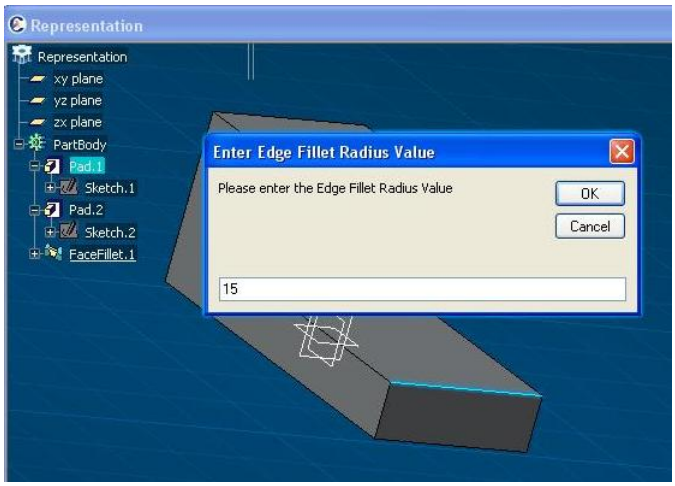
The *Item* method on `oObjSelection`, returns the iIndex-th SelectedElement Object contained by the current selection, the first Object, `oEdge` in this case.

II. Creates Edge Fillet With Constant Radius

Now we create edge fillet on selected Edge with input radius.

Firstly we prompt the user to enter Edge fillet radius value. As shown in following fig. This fig also shows selected Edge.

```
...
Dim  dEdgeFilletRadius As Double
dEdgeFilletRadius = InputBox("Please enter the Edge Fillet Radius Value ", "Enter Edge Fillet Radius Value")
...
```



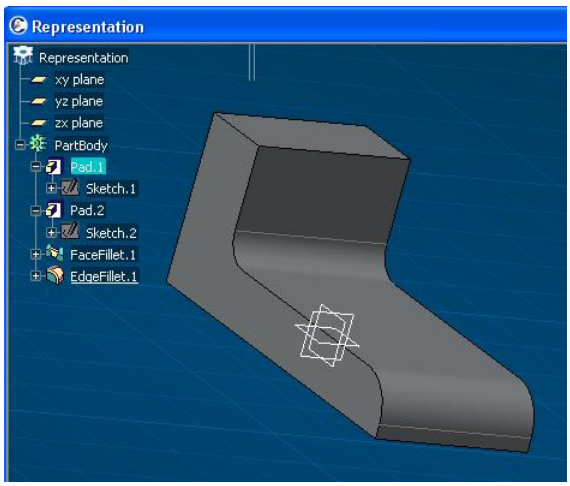

```
...
Dim oEdgeFillet As EdgeFillet
Set oEdgeFillet = oPart.ShapeFactory.AddNewEdgeFilletWithConstantRadius(oEdge, 1, dEdgeFilletRadius)

oPart.Update
...
```

A call to `ShapeFactory` property of Part on `oPart` returns the part shape factory object.

Next a call to `AddNewEdgeFilletWithConstantRadius` method of ShapeFactory creates a Edge fillet with constant radius on selected edge (`oEdge`) with given input radius (`dEdgeFilletRadius`).

Then we update the Part object (`oPart`) using call *Update* method.

Fig.3 Edge Fillet



Above [Fig.3] shows the newly created Edge Fillet

References

[1] About Microsoft Automation Languages, Debug, and Compatibility

Creating Geometrical Set in a 3DShape

This use case fundamentally illustrates creating and adding Geometrical Set to a 3DShape

---

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdGsiUcCreateGSSource.htm

Related Topics
Part Object Model Map

This use case can be divided in five steps:

1. Creates a 3DShape
2. Retrieves the Part Object
3. Retrieves its Geometrical Set collection
4. Creates a Geometrical Set
5. Updates the Part Object

1. **Creates a 3DShape**

   As a first step, this UC creates a 3DShape thanks to the PLMNewService service in CATIA.

```
...
    Dim oPLMNewService As PLMNewService
    Set oPLMNewService = CATIA.GetSessionService("PLMNewService")

    Dim oEditor3DShape As Editor
    oPLMNewService.PLMCreate "3DShape", oEditor3DShape
...
```

   The 3DShape is created and edited through the 3D Shape editor, `oEditor3DShape`. For further information on Creating 3DShape, Please refer to [1]

2. **Retrieves the Part Object**

```
...
    Dim oPart As Part
    Set oPart = oEditor3DShape.ActiveObject
...
```

   The *ActiveObject* method of 3D Shape Editor, `oEditor3DShape` returns the Part Object, `oPart`

3. **Retrieves its Geometrical Set collection**

   Further, we retrieve the HybridBodies collection from the Part Object.

```
...
    Dim oHybridBodies As HybridBodies
    Set oHybridBodies = oPart.HybridBodies
...
```

   The *HybridBodies* method of the Part object, returns the `HybridBodies` object , which is the collection of all direct HybridBody objects (i.e. GeometricalSet) aggregated by the Part object, `oPart`

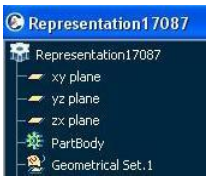
4. **Creates a Geometrical Set**

   In this step, we create and add the Geometrical Set to the HybridBodies collection.

```
...
    Dim oGeometricalSet As HybridBody
    Set oGeometricalSet = oHybridBodies.Add()
...
```

The *Add* method of the HybridBodies object, creates a new hybrid body, `oGeometricalSet` and adds it to the HybridBodies collection, `oHybridBodies`

Fig. 2 Geometrical Set Added in the Specification tree within CATIA



5. **Updates the Part Object**

```
...
    oPart.Update
End Sub
```

Updates the Part Object, `oPart` result with respect to its specifications.

References

[1] Creating 3DShape

Working with PlanarFace

This use case fundamentally illustrates about working with PlanarFace. In this UC we learn how to retrieve PlanerFace and use it.

Related Topics
Editor Object
Selection Object
Part Object Model Map

Before you begin: Note that:

- You should first launch CATIA.
- The macro should essentially be launched from a Part Editor (meaning input data is always a PLM Product Representation Reference).
- Product/Part editor must be active before launching the macro.

Where to find the macro: CAAScdMmrUcWorkingWithPlanarFaceSource.htm

1. Prolog

   The macro begins with the **WorkingWithPlanarFace** main subroutine. It typically begins with the following extract of code, which serves as a built-in "error handling" mechanism.

```
Sub WorkingWithPlanarFace()

On Error GoTo ErrorSub
...
```

   The first instruction means that when a method will throw an error, the macro will skip to the line named, **ErrorSub**. See the step Epilog.

   `WorkingWithPlanarFace` subroutine firstly retrieves Part object from selection. Then we select the Face for creation of Draft. Then we select and retrieve the neutral face it is PlanerFace type object. Then we select and retrieve parting element this is also Planer face type object. Then we create Draft using all these inputs (Face , neutral face, parting element).

2. Here, we retrieve the current active Editor object to select PLM object:

```
...
Dim oCurrentActiveEditor As Editor
Set oCurrentActiveEditor = CATIA.ActiveEditor
...
```

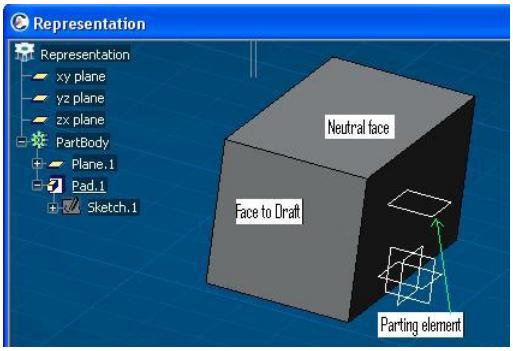
   Return a **Editor** object from the Application object (CATIA) using the **ActiveEditor** method.

3. We then retrieve Selection object from active editor:

```
...
Dim oObjSelection
Set oObjSelection = oCurrentActiveEditor.Selection
...
```

   Return a **Selection** object from the Active Editor using **Selection** method. It is significant to note here that we haven't declared a type for `oObjSelection` (as for several other variables, further ahead in this macro). The section "Tips about VB and VBA" in the technical article "About VB, VBA, Debug, and Compatibility" [] provides an explanation for why these variables are not typed.

4. The selection object retrieved above is now updated with the selection criteria ("Part"). The following image shows sample Object from which user could select Part object (Representation).




```
...
Dim strStatus As String
Dim oInputObjectType(0)

oInputObjectType(0) = "Part"
strStatus = oObjSelection.SelectElement(oInputObjectType, "Select a Part Element from spec tree or the 3D Viewer", False)
...
```

"***Part***" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

At this stage, the application prompts an end user to select a *Part* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

The last argument is false, indicating the end user will always be invited to select something in the spec tree or the 3D Viewer, though an entity (of the type set as filter in the selection criteria) has already been selected by the end-user.

Once the user does the selection, the application proceeds further.

5. Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Part***) from the Selection Object which contains it.

```
...
Dim oSelectedElement As SelectedElement
Set oSelectedElement  =  oObjSelection.Item(1)

Dim oPart As Part
Set oPart = oSelectedElement.Value

MsgBox ("Selected Part Name :") + oPart.Name
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oPart`, a *Part* type representing the selection.

Then we display the *Part* object name using `Part::Name`.

6. We select and retrieve *Face* object to create draft.

Before new selection we need to clear the selection object.

```
...
oObjSelection.Clear
..
```

A call to `Selection::Clear` , on `oObjSelection` clears the previously selected object (Part) from this Selection object.

Then prompt the user to select the Face, a *Face* type object:

```
...
oInputObjectType(0) = "Face"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the face to draft", False)
...
```

"Face" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

At this stage, the application waits for an end user to select a face to draft from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a face) from the Selection Object which contains it.

```
...
Dim oFaceToDraft As Face
Set oFaceToDraft = oObjSelection.Item(1).Value

MsgBox ("Selected Face Name :") + oFaceToDraft.Name
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oFaceToDraft`, a *Face* type representing the selection.

Then we display the *face* object name using `Face::Name`.

7. Next we select the neutral face.

   Before new selection we need to clear the selection object.

```
...
oObjSelection.Clear
..
```

   A call to `Selection::Clear` , on `oObjSelection` clears the previously selected object (Face object) from this Selection object.

   Then prompt the user to select the neutral face, a ***PlanarFace*** type object

```
...
oInputObjectType(0) = "PlanarFace"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the neutral face", False)
...
```

   "PlanarFace" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

   At this stage, the application prompts an end user to select a *neutral face* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

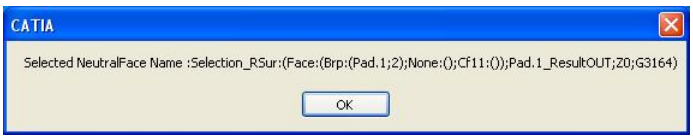
   Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a neutral face) from the Selection Object which contains it.

```
...
Dim oNeutralFace As PlanarFace
Set oNeutralFace = oObjSelection.Item(1).Value

MsgBox ("Selected PartingElement Name :") + oPartingElement.Name
...
```

   A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

   Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oNeutralFace`, a *PlanarFace* type representing the selection.

   Then we display the *PlanarFace* object name using `PlanarFace::Name`.



8. Next we select the parting element.

   Before new selection we need to clear the selection object.

```
...
oObjSelection.Clear
..
```

   A call to `Selection::Clear`, on `oObjSelection` clears the previously selected object (neutral face) from this Selection object.

   Then prompt the user to select the parting element, a ***PlanarFace*** type object

```
...
Status = oObjSelection.SelectElement(oInputObjectType, "Select the parting element", False)
...
```

   At this stage, the application prompts an end user to select a *parting element* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

   Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a parting element) from the Selection Object which contains it.

```
...
Dim oPartingElement As PlanarFace
Set oPartingElement = oObjSelection.Item(1).Value
...
```

   A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

   Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oPartingElement`, a *PlanarFace* type representing the selection.

   Then we display the *PlanarFace* object name using `PlanarFace::Name`.

9. Then we create Draft with selected elements:

```
...
Dim oDraft As Draft
Set oDraft = oPart.ShapeFactory.AddNewDraft(oFaceToDraft, oNeutralFace, 0, oPartingElement, 0#, 0#, 1#, 0, 5#, 0)
...
```

A call to `Part::ShapeFactory` , on `oPart` returns the Factory object.

Next a call to `Factory::AddNewDraft` create a draft as per input (`oFaceToDraft`, `oNeutralFace`, `oPartingElement`).

Next we retrieve DraftDomains from the newly created draft. Then retrieve first DraftDomain and set the pulling direction.

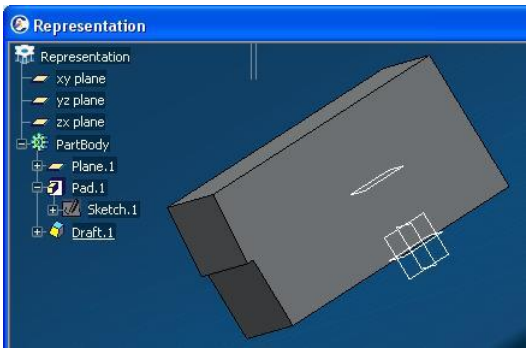
```
...
Dim oDraftDomains As DraftDomains
Set oDraftDomains = oDraft.DraftDomains

Dim oDraftDomain As DraftDomain
Set oDraftDomain = oDraftDomains.Item(1)

oDraftDomain.SetPullingDirection 0#, 0#, 1#

oPart.Update
...
```

A call to `Draft::DraftDomains` returns `oDraftDomains` (a *DraftDomains* type object) from newly created draft `oDraft` (a Draft type).

then a call to `DraftDomains::Item` on `oDraftDomains` with index 1 as input returns first DraftDomain object (`oDraftDomain`).

then call to `DraftDomain::SetPullingDirection` sets the pulling direction.

Then we update the Part object (`oPart`) using call `Part::Update`. After this call we can see the created draft as shown in below image.



10. We then save the changes.

```
...
CATIA.GetSessionService("PLMPropagateService").Save
...
```

We next save this product/ Representation in the database. It is done with calls which occur in the following sequence

- A call to `Application::GetSessionService` on CATIA (an Application object, defined internally by VB) returns a *PLMPropagateService* object.
- The next call to `PLMPropagateService::Save` eventually saves this representation in the database.

11. Epilog

The following extract of code is primarily meant for error-handling purpose. Any run time error that the macro encounters results in the execution flow reaching this part of the code, and then terminating with a normal exit from subroutine scope.

```
...
    GoTo EndSub

ErrorSub:
    MsgBox Err.Description

EndSub:
End Sub
```

Working with TriDimFeatEdge And BiDimFeatEdge

This use case fundamentally illustrates about working with TriDimFeatEdge And BiDimFeatEdge. In this use case we learn how retrieve and use the TriDimFeatEdge And BiDimFeatEdge.

Related Topics
Editor Object
Selection Object
Part Object
Model Map

Before you begin: Note that:

- You should first launch CATIA.
- The macro should essentially be launched from a Part Editor (meaning input data is always a PLM Product Representation Reference).
- Product/Part editor must be active before launching the macro.

Where to find the macro: CAAScdMmrUcWorkingWithTriDimFeatEdgeAndBiDimFeatEdgeSource.htm

1. Prolog

   The macro begins with the **WorkingWithTriDimFeatEdgeAndBiDimFeatEdge** main subroutine. It typically begins with the following extract of code, which serves as a built-in "error handling" mechanism.

   ```
   Sub WorkingWithTriDimFeatEdgeAndBiDimFeatEdge()

   On Error GoTo ErrorSub
   ...
   ```

   The first instruction means that when a method will throw an error, the macro will skip to the line named, **ErrorSub**. See the step Epilog.

   `WorkingWithTriDimFeatEdgeAndBiDimFeatEdge` subroutine firstly retrieves Part object from selection. Then we retrieve HybridBodies. Then from these HybridBodies we retrieve first HybridBody object then we prompt the user for selection of TriDimFeatEdge or BiDimFeatEdge. According to selection we create point on that edge according to input distance.

2. Here, we retrieve the current active Editor object to select PLM object

   ```
   ...
   Dim oCurrentActiveEditor As Editor
   Set oCurrentActiveEditor = CATIA.ActiveEditor
   ...
   ```

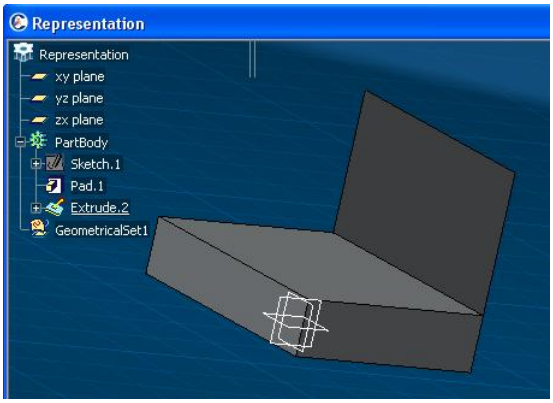
   Return a **Editor** object from the Application object (CATIA) using the **ActiveEditor** method.

3. We then retrieve Selection object from active editor

   ```
   ...
   Dim oObjSelection
   Set oObjSelection = oCurrentActiveEditor.Selection
   ...
   ```

   Return a **Selection** object from the Active Editor using **Selection** method. It is significant to note here that we haven't declared a type for `oObjSelection` (as for several other variables, further ahead in this macro). The section "Tips about VB and VBA" in the technical article "About VB, VBA, Debug, and Compatibility" [] provides an explanation for why these variables are not typed.

4. The selection object retrieved above is now updated with the selection criteria ("Part"). The following image shows sample Object from which user could select Part object (Representation).


   ```
   ...
   Dim strStatus As String
   Dim oInputObjectType(0)

   oInputObjectType(0) = "Part"
   strStatus = oObjSelection.SelectElement(oInputObjectType, "Select a Part Element from spec tree or the 3D Viewer", False)
   ...
   ```

   "***Part***" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

   At this stage, the application prompts an end user to select a *Part* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

   The last argument is false, indicating the end user will always be invited to select something in the spec tree or the 3D Viewer, though an entity (of the type set as filter in the selection criteria) has already been selected by the end-user.

   Once the user does the selection, the application proceeds further.

5. Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Part***) from the Selection Object which contains it.

   ```
   ...
   Dim oSelectedElement As SelectedElement
   Set oSelectedElement  =  oObjSelection.Item(1)

   Dim oPart As Part
   Set oPart = oSelectedElement.Value

   MsgBox ("Selected Part Name :") + oPart.Name
   ...
   ```

   A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oPart`, a *Part* type representing the selection.

Then we display the *Part* object name using `Part::Name`.

6. Next we retrieve Hybrid Body.

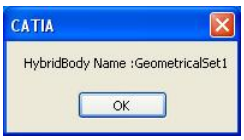
   Firstly we retrieve HybridBodies from Part and then we retrieve first HybridBody from the list of Hybrid Bodies.

```
...
Dim oHybridBodies As HybridBodies
Set oHybridBodies =  oPart.HybridBodies

Dim oHybridBody As HybridBody
Set oHybridBody =  oHybridBodies.Item(1)
...
```

   A call to Part::HybridBodies on oPart retrieves a list of Hybrid Body oHybridBodies, a HybridBodies type

   Next a call to HybridBodies::Item on oHybridBodies retrieves Hybrid Body(oHybridBody), in this case we retrieve first object from the list.

   Then we display the *HybridBody* object name using `HybridBody::Name`.



7. Next select and retrieves TriDimFeatEdge or BiDimFeatEdge.

   Prompt the user to select the TriDimFeatEdge or BiDimFeatEdge.

```
...
Dim oInputObjectType1(1)
oInputObjectType1(0) = "TriDimFeatEdge"
oInputObjectType1(1) = "BiDimFeatEdge"
Status = oObjSelection.SelectElement(oInputObjectType1, "Select the  TriDimFeatEdge or BiDimFeatEdge", False)
...
```

   "TriDimFeatEdge" and "BiDimFeatEdge" types are appended to an array (`oInputObjectType1`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

   At this stage, the application waits for an end user to select TriDimFeatEdge or BiDimFeatEdge from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

   Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (TriDimFeatEdge or BiDimFeatEdge) from the Selection Object which contains it.

```
...
Dim oCurve
Set oCurve = oObjSelection.Item(1).Value
...
```

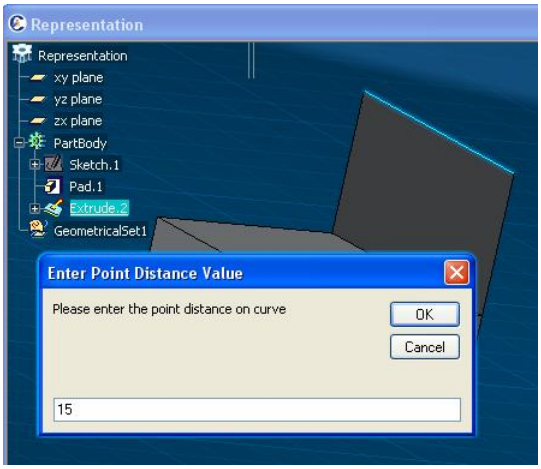
   A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

   Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oCurve`, a veriant representing the TriDimFeatEdge or BiDimFeatEdge type according to selection.

8. Then we create point on selected curve (HybridShapePointOnCurve)

   Firstly we prompt the user to input the radius value for face fillet.

```
...
Dim dPointDistance As Double
dPointDistance = InputBox("Please enter the point distance on curve  ", "Enter Point Distance Value")
...
```



Now we create point on selected curve (TriDimFeatEdge or BiDimFeatEdge) with input distance.

```
...
Dim oHybridShapePointOnCurve As HybridShapePointOnCurve
Set oHybridShapePointOnCurve = oPart.HybridShapeFactory.AddNewPointOnCurveFromDistance(oCurve, dPointDistance, False)
...
```

A call to `Part::HybridShapeFactory` , on `oPart` returns the Factory object.

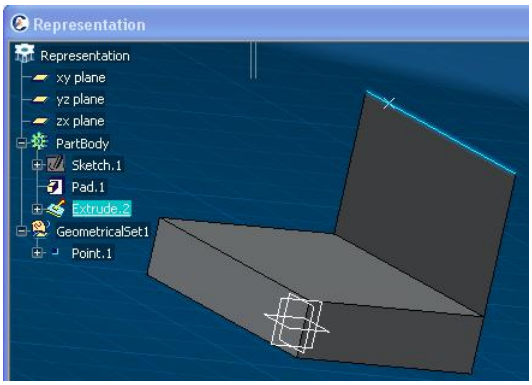
Next a call to `Factory::AddNewPointOnCurveFromDistance` create a point on the curve (`oCurve`, a *TriDimFeatEdge* or *BiDimFeatEdge* type) with input distance (`dPointDistance`).

Next we append newly created point in the HybridBody.

```
...
oHybridBody.AppendHybridShape oHybridShapePointOnCurve

oPart.Update
...
```

A call to `HybridBody::AppendHybridShape` appends newly created line `oHybridShapePointOnCurve` (a *HybridShapePointOnCurve* type).

Then we update the Part object (`oPart`) using call `Part::Update`. After this call we can see the created line as shown in below image.



9. We then save the changes.

```
...
CATIA.GetSessionService("PLMPropagateService").Save
...
```

We next save this product/ Representation in the database. It is done with calls which occur in the following sequence

- A call to `Application::GetSessionService` on CATIA (an Application object, defined internally by VB) returns a *PLMPropagateService* object.
- The next call to `PLMPropagateService::Save` eventually saves this representation n the database.

10. Epilog

The following extract of code is primarily meant for error-handling purpose. Any run time error that the macro encounters results in the execution flow reaching this part of the code, and then terminating with a normal exit from subroutine scope.

```
...
    GoTo EndSub

ErrorSub:
    MsgBox Err.Description

EndSub:
End Sub
```

Working with Vertices

This use case fundamentally illustrates about working with Vertices. It further selects two vertices and creates a line between them.

Related Topics
Editor Object
Selection Object
Part Object Model Map

Before you begin: Note that:

- Launch CATIA.

Where to find the macro: CAAScdMmrUcWorkingWithVerticesSource.htm

This use case can be divided in 3 steps

1. Creates a 3D Shape with vertices (a 2D Sketch extruded to a Pad)
2. Creates a Point to Point Line with two vertices
3. Appends newly created Line to the Geometrical Set

1. **Creates a 3D Shape with vertices (a 2D Sketch extruded to a Pad)**

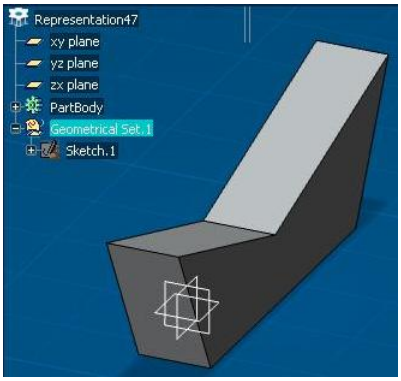
   As a first step, this UC creates the geometry with a 2D Sketch which is extruded to a Pad creating vertices

```
...
    Dim oPart As Part
    CreatingGeometry oPart
...
```

   A call to *CreatingGeometry* function creates a 3DShape, retrieves its Part Object, `oPart`, creates a Geometrical Set and adds a sketch with lines to form a bounded area to it and extrudes it to a Pad, forming vertices as depicted in the below figure, Fig1.

   Please Note here that the created 3DShape essentially has a Geometrical Set, to enable the line creation with vertices in the steps ahead.

   Fig. 1: Created Geometry with Sketch Extruded to a Pad

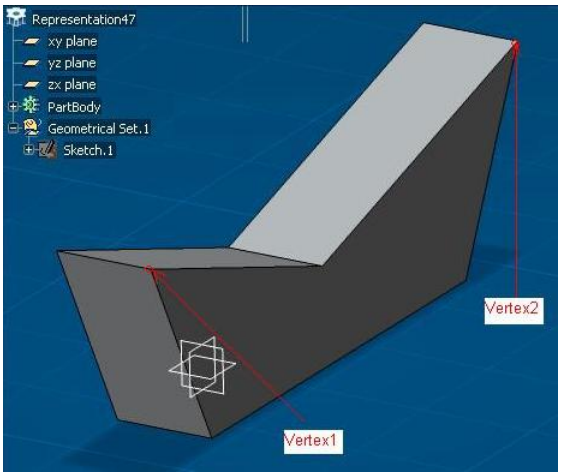


   We thus create the Geometry and we now proceed to selecting the Vertices and drawing a line between them.

2. **Creates a Point to Point Line with two vertices**

   In this step, we prompt the user to select the two vertices from the Geometry created in the above step. We then create a Point to Point line between the two selected vertices, as depicted in Fig2. For better understanding we divide this step in sub steps

   Fig. 2: Selecting two Vertices



   i. Retrieves Active Editor from CATIA

      As a first sub step, we retrieve the Selection Object from CATIA, to enable us to select the vertices in the steps ahead.

```
...
    Dim oCurrentActiveEditor As Editor
    Set oCurrentActiveEditor = CATIA.ActiveEditor
...
```

      The *ActiveEditor* property of the Application Object, CATIA returns the Editor, oCurrentActiveEditor which is currently active.

ii. Retrieves Selection object from Active Editor:

```
...
    Dim oObjSelection
    Set oObjSelection = oCurrentActiveEditor.Selection
...
```

A call to *Selection* Property of the Active Editor returns `oObjSelection`, a `Selection` object. It is significant to note here that we have not declared a type for `oObjSelection` . The section "Virtual Function Table Compatibility" in the Technical Article "About Automation Languages, Debug, and Compatibility" [1] provides an explanation for why these variables are not typed.

iii. Updates Selection Object with selection criteria(Vertex only allowed)

```
...
    Dim oInputObjectType(0)
    oInputObjectType(0) = "Vertex"
...
```

"Vertex" type is appended to an array `oInputObjectType`, which now represents the select criteria.

iv. Retrieves First Vertex through Selection

We now retrieve the First Vertex, Vertex1

```
...
    strStatus = oObjSelection.SelectElement(oInputObjectType, "Select the first vertex", True)
...
```

The *SelectElement* method on the `Selection` Object, `oObjSelection` asks the end user to select a feature (in the geometry or in the specification tree). During the selection, when the end user moves the mouse above a feature which fits in the given filter, Vertex in this case, the mouse pointer turns into the "hand" cursor; otherwise into the "no entry" cursor. Refer Fig2 for selecting the Vertex1.

```
...
    Dim oFirstVertex As Vertex
    Set oFirstVertex = oObjSelection.Item(1).Value
...
```

The *Item* method on `oObjSelection`, returns the iIndex-th SelectedElement Object contained by the current selection, the first Object, `oFirstVertex` in this case.

We further clear the above selection, to enable us to select the Second Vertex

```
...
    oObjSelection.Clear
...
```

The *Clear* method on `Selection` Object, `oObjSelection`, clears the selection.

v. Retrieves Second Vertex through Selection

We now retrieve the Second Vertex, Vertex2

```
...
    strStatus = oObjSelection.SelectElement(oInputObjectType, "Select the second vertex", True)
    Dim oSecondVertex As Vertex
    Set oSecondVertex = oObjSelection.Item(1).Value
...
```

Following the above sub step, we similarly select the second Vertex, oSecondVertex. Refer Fig2 for selecting the Vertex2.

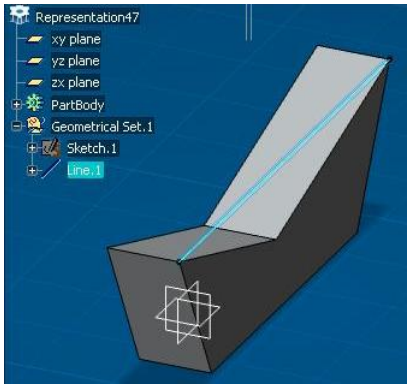
vi. Creates a Point-Point Line with two vertices

In this sub step we finally create a Point to Point line between the two selected vertices from the above sub steps

```
...
    Dim oHybridShapeLinePtPt As HybridShapeLinePtPt
    Set oHybridShapeLinePtPt = oPart.HybridShapeFactory.AddNewLinePtPt(oFirstVertex, oSecondVertex)
...
```

The *AddNewLinePtPt* method on `HybridShapeFactory` Object aggregated beneath the Part Object, creates a new point-point line where the two points are the two vertices oFirstVertex and oSecondVertex (Vertex1, Vertex2) selected by the user. Refer Fig2 for selection example.

Fig. 3: Created Line Between the two Vertices



The created Line is as depicted in the above figure, Fig3

3. **Appends newly created Line to the Geometrical Set**

   This step essentially retrieves the Geometrical Set Object from the Part Object and Append the Created line to it. We divide this step in sub steps for better understanding.

   It is interesting to note here that the Geometrical Set is internally referred to as a Hybrid Body.

   i. Retrieves Geometrical Sets Collection Object

   As a first sub step, we retrieve the Geometrical Sets Collection Object from the Part Object

```
...
    Dim oHybridBodies As HybridBodies
    Set oHybridBodies =  oPart.HybridBodies
...
```

   The *HybridBodies* method of the `Part` object, returns the `HybridBodies` Object , which is the Collection of all direct `HybridBody` objects (i.e. GeometricalSet) aggregated by the Part object, `oPart`.

   ii. Retrieves First Geometrical Set from the Geometrical Sets Collection Object.

   We now retrieve the First Geometrical Set from the Geometrical Sets Collection Object retrieved in the above sub step:

```
...
    Dim oHybridBody As HybridBody
    Set oHybridBody =  oHybridBodies.Item(1)
...
```

   Next, the *Item* method on the `HybridBodies` Object, `oHybridBodies`, retrieves the Hybrid Body by index, the first Hybrid Body from the Collection in this case.

   iii. Appends newly created Line to the Geometrical Set

   In this final sub step, we append the line created in the earlier step to the Geometrical Set retrieved in the above sub step

```
...
    oHybridBody.AppendHybridShape oHybridShapeLinePtPt
    oPart.Update
...
```

   The *AppendHybridShape* method on the `HybridBody` Object, `oHybridBody` accepts a HyBridShape Object, `oHybridShapeLinePtPt` in this case as an input and appends it to the Hybrid Body.

   And lastly, we Update the `Part` Object with a call to *Update* method on `oPart`. Refer to Fig3 to view the result in CATIA specification tree.

References

[1] About Microsoft Automation Languages, Debug, and Compatibility

Working with Cylindrical Face

This use case fundamentally illustrates working with Cylindrical Face and a Sketch Based Shape. The context is creating a circular pattern of a Sketch Based Shape along a cylindrical face.

Related Topics
Selection Object
Part Object
Model Map

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdMmrUcWorkingWithCylindricalFaceSource.htm

This use case can be divided in 4 steps

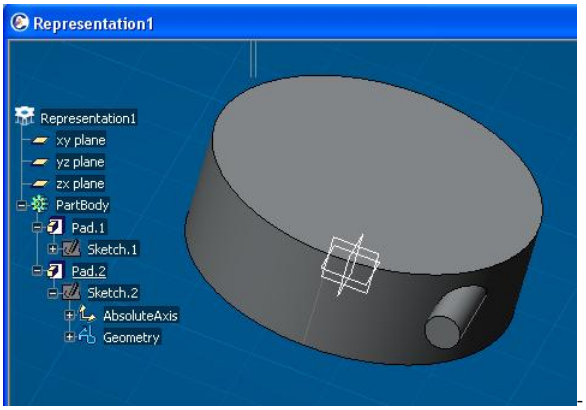
1. Creates 3DShape with Geometries
2. Selects and Retrieves Sketch Based Shape
3. Selects and Retrieves Cylindrical Face
4. Creates Circular Pattern

1. **Creates 3DShape with Geometries**

```
...
   Dim oPart As Part
   CreatePadsWithCircularShape oPart
...
```

The *CreatePadsWithCircularShape* sub routine creates a new 3DShape with geometry as shown in the following [Fig.1].

Fig.1: 3DShape with Two Circular Pads



`oPart` is the Part Object of the 3D Shape, the object aggregating all the objects making up the 3D shape, whose `Pad.1` and `Pad.2` are used to create the circu

2. **Selects and Retrieves Sketch Based Shape**

   We then retrieve Selection object from active editor

```
...
Dim oObjSelection
Set oObjSelection = CATIA.ActiveEditor.Selection
...
```

   A call to *Selection* Property of the Active Editor returns `oObjSelection`, a *Selection* object. It is significant to note here that we haven't declared a type for other variables, further ahead in this macro). The section "Virtual Function Table Compatibility" in the technical article "About Automation Languages, Del an explanation for why these variables are not typed.

   Now we prompts the user to select the Sketch Based Shape, a *SketchBasedShape* type object to create circular pattern on cylindrical face .

```
...
oInputObjectType(0) = "SketchBasedShape"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the shape to pattern", False)
...
```

   "SketchBasedShape" is appended to an array (`oInputObjectType`), which now represents the select criteria.

   A call to *SelectElement* method of the *Selection* object, `oObjSelection` runs an interactive Selection command and waits for the selection of the Sketch Bas

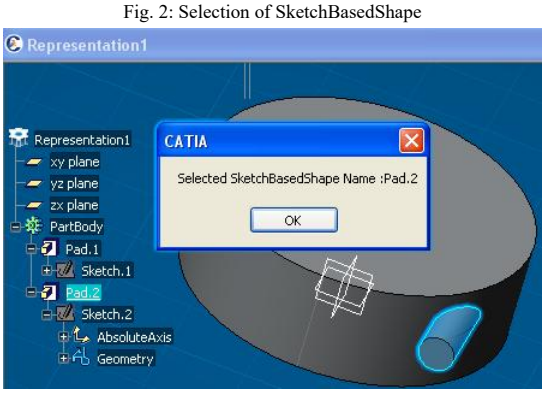
   Next step retrieves a `Shape` from the Selection Object.

```
...
Dim oSelectedShape As Shape
Set oSelectedShape = oObjSelection.Item(1).Value

MsgBox ("Selected SketchBasedShape Name :") + oSelectedShape.Name
...
```

   A call to Item on `oObjSelection` retrieves `oSelectedShape` object of the Shape.

   Then we display the `Shape` object name using *Name* property of `Shape` object, `oSelectedShape`.

   Fig. 2: Selection of SketchBasedShape



   In this step we select the Sketch Based Shape of the pad2 this will be used as pattern shape. Above [Fig.2] shows the section of SketchBasedShape (highligh

3. **Select and Retrieves Cylindrical Face**.

   Before we make a new selection we need to clear the selection object.

```
...
oObjSelection.Clear
..
```

   A *Clear* method of `Selection` object, `oObjSelection` clears the previously selected object (`Shape` object) from this `Selection` object.

   Next we prompts the user to select the Cylindrical face, a ***CylindricalFace*** type object

```
...
oInputObjectType(0) = "CylindricalFace"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the neutral face", False)
...
```

"CylindricalFace" type is appended to an array (`oInputObjectType`), which now represents the select criteria.

A call to *SelectElement* method of the *Selection* object, `oObjSelection` runs an interactive Selection command and waits for the selection of the Cylindrica

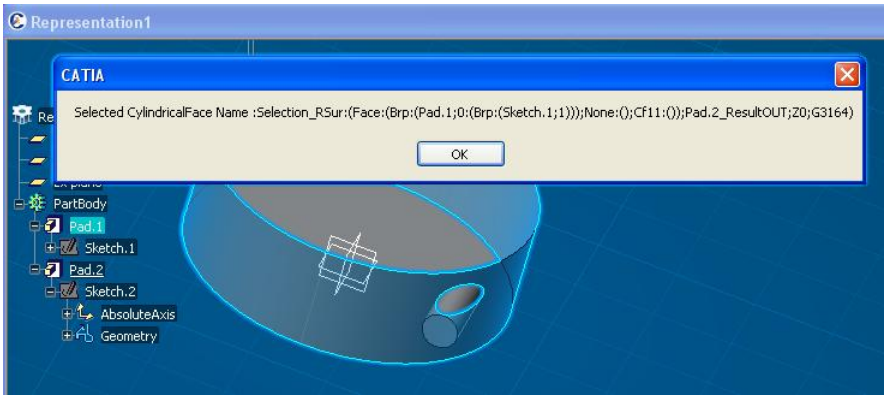
Next step retrieves a CylindricalFace from the Selection Object.

```
...
Dim oSelectedCylindricalFace As CylindricalFace
Set oSelectedCylindricalFace= oObjSelection.Item(1).Value

MsgBox ("Selected CylindricalFace Name :") + oSelectedCylindricalFace.Name
...
```

A call to Item on `oObjSelection` retrieves `oSelectedCylindricalFace` object of the *CylindricalFace* .

Then we display the *CylindricalFace* object name using *Name* property of `Shape` object, `oSelectedCylindricalFace`.

Fig. 3: Selection of Cylindrical Face



In this step we have selected *CylindricalFace* of the Pad1 on which pattern will be created. above [Fig.3] shows selected object *CylindricalFace* (highlighte

4. **Creates Circular Pattern**

For creation of circular pattern we need Rotation Center so we create first reference from Generic Naming label

Rotation Center is the point or vertex that specifies the pattern center of rotation

```
...
Dim oRotationCenter As Reference
Set oRotationCenter = oPart.CreateReferenceFromName("")
...
```

A *CreateReferenceFromName* method of the Part object creates Reference from the generic naming label. here we provide an empty string ("") since we do

Now we create actual circular pattern with selected Shape object, `oSelectedShape` as pattern, on selected CylindricalFace, `oSelectedCylindricalFace`.

```
...
Set CircPattern = oPart.ShapeFactory.AddNewCircPattern(oSelectedShape, 1, 4, 20#, 45#, 1, 4, oRotationCenter, oSelectedCylind

oPart.Update
...
```

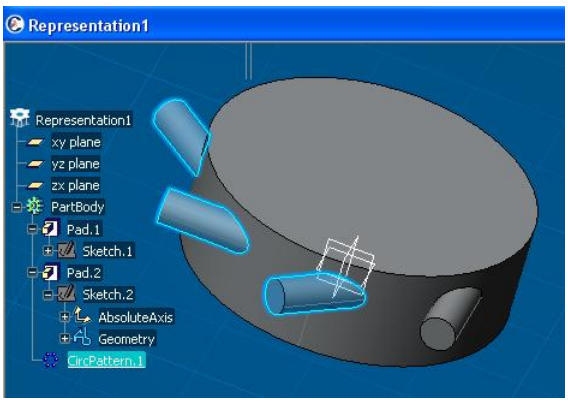
A `AddNewCircPattern` method of the `ShapeFactory` creates a circular pattern on selected *CylindricalFace* (`oSelectedCylindricalFace`) with selected *Sh* pattern object.

The arguments for the creation of circular pattern are as follows

- `oSelectedShape`, a shape to be copied by the circular pattern, in this case we have selected *SketchBasedShape* (`oSelectedShape`).
- 1, number of times `oSelectedShape` will be copied along pattern radial direction (in this it will not add in radial direction since number is 1).
- 4, number of times `oSelectedShape` will be copied along pattern angular direction (as we can see in the following Fig.4).
- 20,distance that will separate two consecutive copies in the pattern along its radial direction (in this case it will not be applicable).
- 45,angle that will separate two consecutive copies in the pattern along its angular direction.
- 1, Specifies the position of the original shape `oSelectedShape` among its copies along the radial direction.
- 4, Specifies the position of the original shape `oSelectedShape` among its copies along the angular direction.
- `oRotationCenter`, the point or vertex that specifies the pattern center of rotation.
- `oSelectedCylindricalFace`, the line or linear edge that specifies the axis around which instances will be rotated relative to each other The followin
- True, boolean type, True indicates that Item to Duplicate are copied in the direction of the natural orientation of Rotation Axis.
- 0,The angle applied to the direction iRotationAxis prior to applying the pattern. The original shape iShapeToCopy is used as the rotation center. Neve themselves are not rotated. This allows the definition of a circular pattern relatively to existing geometry, but not necessarily parallel to it.
- True, a boolean type, the instances of Item to duplicate copied by the pattern should be kept parallel to each other (True).

Then we update the Part object (`oPart`) using call *Update* method.

Fig. 4: Circular Pattern



The above [Fig.4] shows the circular pattern of SketchBasedShape generated along cylindrical face. The newly created Pattern highlighted in blue.

References

[1] About Microsoft Automation Languages, Debug, and Compatibility

Working with Rectilinear Edge

This use case fundamentally illustrates about working with Rectilinear Edge. In this use case we learn how retrieve and use the RectilinearTriDimFeatEdge, RectilinearBiDimFeatEdge and RectilinearMonoDimFeatEdge.

Related Topics
Editor Object
Selection Object
Part Object
Model Map

Before you begin: Note that:

- You should first launch CATIA.
- The macro should essentially be launched from a Part Editor (meaning input data is always a PLM Product Representation Reference).
- Product/Part editor must be active before launching the macro.

Where to find the macro: CAAScdMmrUcWorkingWithRectilinearDimFeatEdgeSource.htm

1. Prolog

   The macro begins with the **WorkingWithRectilinearDimFeatEdge** main subroutine. It typically begins with the following extract of code, which serves as a built-in "error handling" mechanism.

   ```
   Sub WorkingWithRectilinearDimFeatEdge()

   On Error GoTo ErrorSub
   ...
   ```

   The first instruction means that when a method will throw an error, the macro will skip to the line named, **ErrorSub**. See the step Epilog.

   `WorkingWithRectilinearDimFeatEdge` subroutine firstly retrieves Part object from selection. Then we select and retrieve face on which user want to create hole. Next we select the rectilinear edge (RectilinearTriDimFeatEdge or RectilinearBiDimFeatEdge or RectilinearMonoDimFeatEdge). Then we create hole and set the hole properties and finally we set the direction of hole which is same as the selected rectilinear edge.

2. Here, we retrieve the current active Editor object to select PLM object

   ```
   ...
   Dim oCurrentActiveEditor As Editor
   Set oCurrentActiveEditor = CATIA.ActiveEditor
   ...
   ```

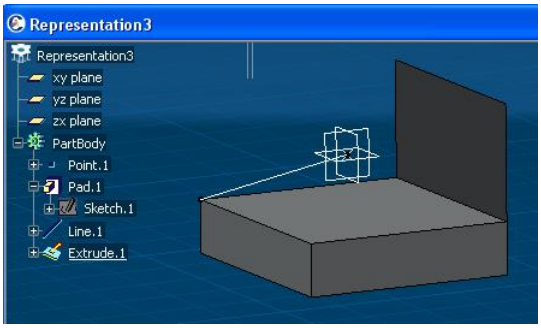
   Return a **Editor** object from the Application object (CATIA) using the **ActiveEditor** method.

3. We then retrieve Selection object from active editor

   ```
   ...
   Dim oObjSelection
   Set oObjSelection = oCurrentActiveEditor.Selection
   ...
   ```

   Return a **Selection** object from the Active Editor using **Selection** method. It is significant to note here that we haven't declared a type for `oObjSelection` (as for several other variables, further ahead in this macro). The section "Tips about VB and VBA" in the technical article "About VB, VBA, Debug, and Compatibility" [] provides an explanation for why these variables are not typed.

4. The selection object retrieved above is now updated with the selection criteria ("Part"). The following image shows sample Object from which user could select Part object (Representation).




```
...
Dim strStatus As String
Dim oInputObjectType(0)

oInputObjectType(0) = "Part"
strStatus = oObjSelection.SelectElement(oInputObjectType, "Select a Part Element from spec tree or the 3D Viewer", False)
...
```

"***Part***" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

At this stage, the application prompts an end user to select a *Part* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

The last argument is false, indicating the end user will always be invited to select something in the spec tree or the 3D Viewer, though an entity (of the type set as filter in the selection criteria) has already been selected by the end-user.

Once the user does the selection, the application proceeds further.

5. Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Part***) from the Selection Object which contains it.

```
...
Dim oSelectedElement As SelectedElement
Set oSelectedElement  =  oObjSelection.Item(1)

Dim oPart As Part
Set oPart = oSelectedElement.Value

MsgBox ("Selected Part Name :") + oPart.Name
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oPart`, a *Part* type representing the selection.

Then we display the *Part* object name using `Part::Name`.

6. Next we select and retrieve face on which user want to create hole.

Before new selection we need to clear the selection object.

```
...
oObjSelection.Clear
..
```

A call to `Selection::Clear` , on `oObjSelection` clears the previously selected object (Part) from this Selection object.

Then prompt the user to select the Face, a *Face* type object

```
...
oInputObjectType(0) = "Face"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the face to draft", False)
...
```

"Face" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

At this stage, the application waits for an end user to select a face to draft from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

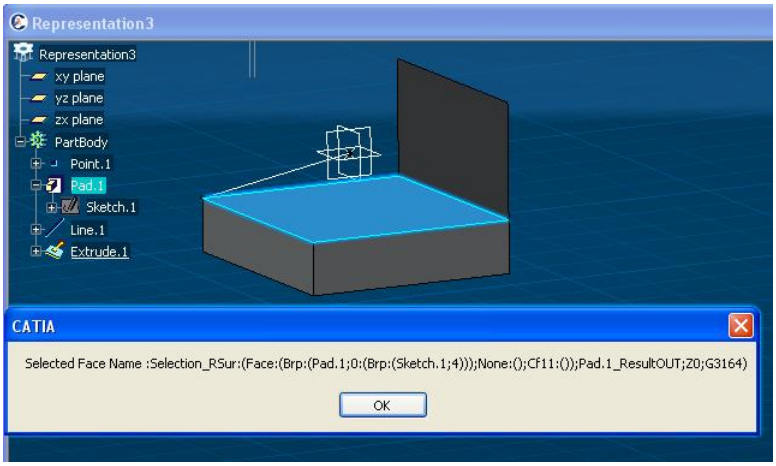
Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a face) from the Selection Object which contains it.

```
...
Dim oFace As Face
Set oFace = oObjSelection.Item(1).Value

MsgBox ("Selected Face Name :") + oFace.Name
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oFace`, a *Face* type representing the selection.

Then we display the *face* object name using `Face::Name`.



7. Create hole on selected face

Next select Rectilinear Edge for hole direction.

Prompt the user to select the RectilinearTriDimFeatEdge or RectilinearBiDimFeatEdge or RectilinearMonoDimFeatEdge.

```
...
Dim oEnabledObjectSelection(2)
oEnabledObjectSelection(0) = "RectilinearTriDimFeatEdge"
oEnabledObjectSelection(1) = "RectilinearBiDimFeatEdge"
oEnabledObjectSelection(2) = "RectilinearMonoDimFeatEdge"
Status = oObjSelection.SelectElement(oEnabledObjectSelection, "Select the hole direction", False)
...
```

" RectilinearTriDimFeatEdge", "RectilinearBiDimFeatEdge" and "RectilinearMonoDimFeatEdge" types are appended to an array (`oEnabledObjectSelection`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated with this select criteria, with a call to `SelectElement` method.

At this stage, the application waits for an end user to select RectilinearTriDimFeatEdge or RectilinearBiDimFeatEdge or RectilinearMonoDimFeatEdge from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, represents the prompt message)

Now create hole

```
...
Dim oHole As Hole
Set oHole = oPart.ShapeFactory.AddNewHoleFromPoint(20#, -5.5, 1.07, oFace, 10#)
...
```

A call to `Part::ShapeFactory` , on `oPart` returns the Factory object.

Next a call to `Factory::AddNewHoleFromPoint` create a hole `oHole`, a *Hole* type) on selected face.

Then we set the ThreadingMode and ThreadSide for the hole

```
...
oHole.ThreadingMode = 1

oHole.ThreadSide = 0
...
```

A call to `Hole::ThreadingMode` sets the threading mode. In this case we are setting threading mode by giving 1 as enum value as an input.

Next as call to `Hole::ThreadSide` sets the thread side. In this case we are setting threading side by giving 0 as enum value as an input.

The user has made alrady selection in the spec tree or the 3D Viewer, the current step retrieves it (RectilinearTriDimFeatEdge or RectilinearBiDimFeatEdge or RectilinearMonoDimFeatEdge) from the Selection Object which contains it. Then set the hole direction.

```
...
oHole.SetDirection oObjSelection.Item(1).Value
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

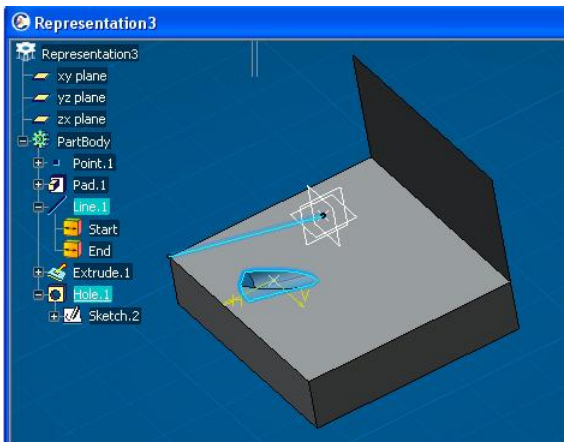
Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns a variant representing the RectilinearTriDimFeatEdge or RectilinearBiDimFeatEdge or RectilinearMonoDimFeatEdge type according to selection.

Next a call `Hole::SetDirection` sets the direction of hole with retrieved variant in above.

Then we update Part object

```
...
oPart.Update
...
```

Then we update the Part object (`oPart`) using call `Part::Update`. After this call we can see the created line as shown in below image.



8. We then save the changes.

```
...
CATIA.GetSessionService("PLMPropagateService").Save
...
```

We next save this product/ Representation in the database. It is done with calls which occur in the following sequence

- A call to `Application::GetSessionService` on CATIA (an Application object, defined internally by VB) returns a *PLMPropagateService* object
- The next call to `PLMPropagateService::Save` eventually saves this representation n the database.

9. Epilog

The following extract of code is primarily meant for error-handling purpose. Any run time error that the macro encounters results in the execution flow reaching this part of the code, and then terminating with a normal exit from subroutine scope.

```
...
    GoTo EndSub

ErrorSub:
    MsgBox Err.Description

EndSub:
End Sub
```

Adding New Constraint

This use case fundamentally illustrates about adding new constraint. This use case demonstrates how to create constraints according to its type.

Related Topics
Part Object Model Map
Constraints Collection Object

Before you begin: Note that:

- You should first launch CATIA.
- The macro should essentially be launched from a Part Editor (meaning input data is always a PLM Product Representation Reference).
- Product/Part editor must be active before launching the macro.

Where to find the macro: CAAScdMmrUcAddingNewConstraintSource.htm

1. Prolog

The macro begins with the **AddingNewConstraint** main subroutine. It typically begins with the following extract of code, which serves as a built-in "error

```
Sub AddingNewConstraint()

On Error GoTo ErrorSub
...
```

The first instruction means that when a method will throw an error, the macro will skip to the line named, **ErrorSub**. See the step Epilog.

`AddingNewConstraint` subroutine firstly retrieves Part object from selection. Then retrieve *Constraints* object. then prompt the user for selection of geomet input prompt the user to select the number of elements. Finally based on that create the constraint. In addition to this display broken constraint count and un

2. Here, we retrieve the current active Editor object to select PLM object

```
...
Dim oCurrentActiveEditor As Editor
Set oCurrentActiveEditor = CATIA.ActiveEditor
...
```

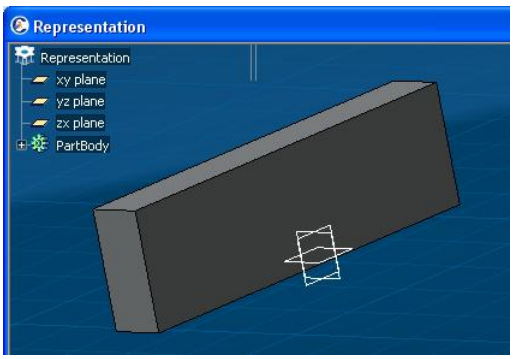
Return a **Editor** object from the Application object (CATIA) using the **ActiveEditor** method.

3. We then retrieve Selection object from active editor

```
...
Dim oObjSelection
Set oObjSelection = oCurrentActiveEditor.Selection
...
```

Return a **Selection** object from the Active Editor using **Selection** method. It is significant to note here that we haven't declared a type for `oObjSelection` (a VB and VBA" in the technical article "About VB, VBA, Debug, and Compatibility" [] provides an explanation for why these variables are not typed.

4. The selection object retrieved above is now updated with the selection criteria ("Part"). The following image shows sample Object from which user could se




```
...
Dim strStatus As String
Dim oInputObjectType(0)

oInputObjectType(0) = "Part"
strStatus = oObjSelection.SelectElement(oInputObjectType, "Select a Part Element from spec tree or the 3D Viewer", False)
...
```

"***Part***" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then updated w

At this stage, the application prompts an end user to select a *Part* from the spec tree or the 3D Viewer (second argument of the `SelectElement` method, rep

The last argument is false, indicating the end user will always be invited to select something in the spec tree or the 3D Viewer, though an entity (of the type

Once the user does the selection, the application proceeds further.

5. Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a ***Part***) from the Selection Object which contains it.

```
...
Dim oSelectedElement As SelectedElement
Set oSelectedElement  =  oObjSelection.Item(1)

Dim oPart As Part
Set oPart = oSelectedElement.Value

MsgBox ("Selected Part Name :") + oPart.Name
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oPart`, a *Part* type representing the selection.

Then we display the *Part* object name using `Part::Name`.

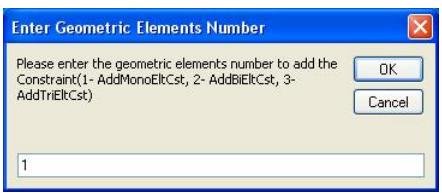
6. Next we retrieve Constraints from Part.

```
...
Dim oConstraints As Constraints
Set oConstraints =  oPart.Constraints
...
```

A call to `Part::Constraints` on `oPart` retrieves a list of constraint `oConstraints`, a *Constraints* type

7. Next we prompt the user to define the number geometrical elements for creation of constraint, according to that we select the method (AddMonoEltCst, Adc

```
...
Dim iGeometricElementsNumber As Integer
Set iGeometricElementsNumber = InputBox("Please enter the geometric elements number to add the Constraint(1- AddMonoEltCst, 2
...
```



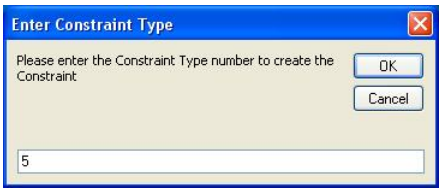
8. Now we prompt the user to select the constraint type. Following are the enum values for type.

catCstTypeReference
catCstTypeDistance
catCstTypeOn
catCstTypeConcentricity
catCstTypeTangency
catCstTypeLength

catCstTypeAngle
catCstTypePlanarAngle
catCstTypeParallelism
catCstTypeAxisParallelism
catCstTypeHorizontality
catCstTypePerpendicularity
catCstTypeAxisPerpendicularity
catCstTypeVerticality
catCstTypeRadius
catCstTypeSymmetry
catCstTypeMidPoint
catCstTypeEquidistance
catCstTypeMajorRadius
catCstTypeMinorRadius
catCstTypeSurfContact
catCstTypeLinContact
catCstTypePoncContact
catCstTypeChamfer
catCstTypeChamferPerpend
catCstTypeAnnulContact
catCstTypeCylinderRadius
catCstTypeStContinuity
catCstTypeStDistance
catCstTypeSdContinuity
catCstTypeSdShape

Here user need to input enum value of constraint type.

```
...
Dim iConstraintType As Integer
Set iConstraintType = InputBox("Please enter the Constraint Type number to create the Constraint", "Enter Constraint Type")
...
```



9. Now create constraint according to its type.

According to geometric element constraint type (Mono-element or Bi-element or tri-element) UC waits for selection of elements.

For Mono-element constraint UC waits for one element selection

```
...
oObjSelection.Clear

oInputObjectType(0) = "Reference"
Status = oObjSelection.SelectElement(oInputObjectType, "Select the first Reference", False)

Dim oFirstReference As Reference
Set oFirstReference = oObjSelection.Item(1).Value
...
```

A call to `Selection::Clear` , on `oObjSelection` clears the previously selected object (Part) from this Selection object.

"Reference" type is appended to an array (`oInputObjectType`), which now represents the select criteria. The selection object (`oObjSelection`) is then upda

At this stage, the application prompts an end user to select a *Reference* from the spec tree or the 3D Viewer (second argument of the `SelectElement` metho

Now that the user has made a selection in the spec tree or the 3D Viewer, the current step retrieves it (a *Reference*) from the Selection Object which contains

Next a call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

A call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oFirstReference`, a *Reference* type representing t

Then according to entered constraint type UC creates constraint. If geometric element type is Mono-element then there is no need of further element selectic

```
...
Dim oConstraint As Constraint

If(1 = iGeometricElementsNumber) Then
        Set oConstraint = oConstraints.AddMonoEltCst(iConstraintType, oFirstReference)
...
```

A call to `Constraints::AddMonoEltCst` on `oConstraints` creates a constraint according to input (`iConstraintType`, `oFirstReference`) and returns new

If geometric element type is Bi-element then user need to select one more element then UC creates Constraint according to selection.

```
...
ElseIf (2 = iGeometricElementsNumber) Then

        strStatus = oObjSelection.SelectElement(oInputObjectType, "Select the second Reference", False)

        Dim oSecondReference As Reference
        Set oSecondReference = oObjSelection.Item(1).Value

        Set oConstraint = oConstraints.AddBiEltCst(iConstraintType, oFirstReference, oSecondReference)
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oSecondReference` , a *Reference* type repres

Now a call to `Constraints::AddBiEltCst` on `oConstraints` creates a constraint according to input (`iConstraintType`, `oFirstReference`, `oSecondRefe`

If geometric element type is Tri-element then user has to select total three elements and one element user selected already so user need to select two more el

```
...
ElseIf (3 = iGeometricElementsNumber) Then

        strStatus = oObjSelection.SelectElement(oInputObjectType, "Select the second Reference", False)

        Dim oSecondReferenceForAddTriEltCst As Reference
        Set oSecondReferenceForAddTriEltCst = oObjSelection.Item(1).Value

        strStatus = oObjSelection.SelectElement(oInputObjectType, "Select the third Reference", False)

        Dim oThirdReferenceForAddTriEltCst As Reference
        Set oThirdReferenceForAddTriEltCst = oObjSelection.Item(1).Value

        Set oConstraint = oConstraints.AddTriEltCst(iConstraintType, oFirstReference, oSecondReferenceForAddTriEltCst, oThird

End If
...
```

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oSecondReferenceForAddTriEltCst \`, a *R*

A call to `SelectionObject::Item` at index 1, on `oObjSelection` returns the selected entity, as a *SelectedElement* type (`oSelectedElement`).

Next a call to `SelectedElement::Value` on the selected entity `oSelectedElement` (retrieved above) returns `oThirdReferenceForAddTriEltCst`, a *Refer*

Now a call to `Constraints::AddTriEltCst` on `oConstraints` creates a constraint according to input (`iConstraintType`, `oFirstReference`, `oSecondRef`
created constraint (`oConstraint`, a *Constraint* type)

10. Next we retrieve the broken constraint count and un-updated constraint count.

    Firstly we retrieve Broken constraint count

    ```
    ...
    Dim longBrokenConstraintsCount As Long
    longBrokenConstraintsCount =  oConstraints.BrokenConstraintsCount

    MsgBox " Broken Constraints Count:" & longBrokenConstraintsCount
    ...
    ```

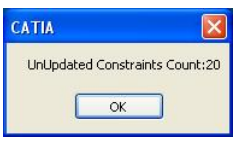
    A call to `Constraints::BrokenConstraintsCount` on `oConstraints` retrieves a broken constraint count `longBrokenConstraintsCount` a *Long* type

    Next we retrieve an un-updated constraint count

    ```
    ...
    Dim longUnUpdatedConstraintsCount As Long
    longUnUpdatedConstraintsCount = oConstraints.UnUpdatedConstraintsCount

    MsgBox "UnUpdated Constraints Count:" & longUnUpdatedConstraintsCount
    ...
    ```

    A call to `Constraints::BrokenConstraintsCount` on `oConstraints` retrieves a unUpdated constraint count `longUnUpdatedConstraintsCount` a *Long*

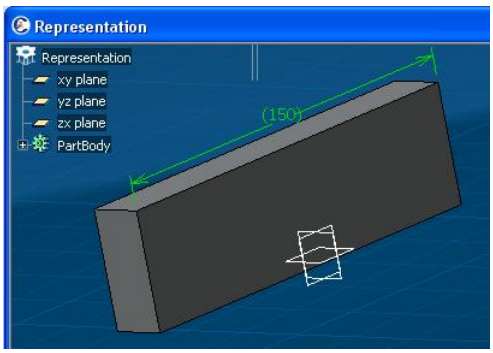
    

11. Next we update Part object.

    ```
    ...
    oPart.Update
    ...
    ```

    Then we update the Part object (`oPart`) using call `Part::Update`. .

    We can see the created constraint as shown in below image. for this we have selected 1 as geometrical element number and then 5 as type means catCstTyp
    constraint



12. We then save the changes.

```
.....
CATIA.GetSessionService("PLMPropagateService").Save
...
```

We next save this product/ Representation in the database. It is done with calls which occur in the following sequence

- A call to `Application::GetSessionService` on CATIA (an Application object, defined internally by VB) returns a *PLMPropagateService* object.
- The next call to `PLMPropagateService::Save` eventually saves this representation n the database.

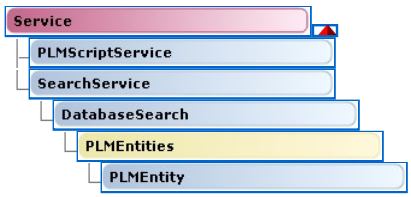
13. Epilog

The following extract of code is primarily meant for error-handling purpose. Any run time error that the macro encounters results in the execution flow reac scope.

```
...
    GoTo EndSub

ErrorSub:
    MsgBox Err.Description

EndSub:
End Subspan>
```

PLM Server Access Object Model Map

See Also Legend



SearchService Object

See Also Legend Use Cases Properties Methods

Represents a service to search for PLM objects.

The **SearchService** object lets you:

- Create a database search for PLM objects using the **DatabaseSearch** object.
- Launch the search and display the result.
- Set a title to the search browser.

Retrieving the SearchService Object

Use the **GetSessionService** of the **Application** object to return a **SearchService** object.

```
Dim oSearchService As SearchService
Set oSearchService = CATIA.GetSessionService("Search")
```

Refer to the Service Object.

Using the SearchService Object

Use the **DatabaseSearch** property to return the **DatabaseSearch** object.

```
Dim oDBSearch As DatabaseSearch
Set oDBSearch = oSearchService.DatabaseSearch
```

Use the **Search** method to launch the search when the **DatabaseSearch** object is filled in with your query criteria.

```
...
oSearchService.Search
....
```

DatabaseSearch Object

See Also Legend Use Cases Properties Methods

Represents a search query for PLM objects in the database and holds the query results.

The **DatabaseSearch** object lets you:

- Create a database search query for PLM objects.
- Retrieve the search query results as a **PLMEntities** collection object as they are displayed in the Search Results dialog box, and manage the way these results are arranged in the collection.

The **DatabaseSearch** object enables you to use different search modes:

- Predefined Queries.
- Advanced Search, divided into:
  - Easy. This is the default search mode.
  - Extended
  - Expert.

Retrieving the DatabaseSearch Object

Use the **DatabaseSearch** property of the **SearchService** object to return the **DatabaseSearch** object.

```
Dim oSearchService As SearchService
Set oSearchService = CATIA.GetSessionService("Search")
Dim oDBSearch As DatabaseSearch
Set oDBSearch = oSearchService.DatabaseSearch
```

Refer to the SearchService Object.

Using the DatabaseSearch Object

This example uses the default Advanced Search - Easy search mode. The search mode are defined thanks to the SearchMode enumeration.

```
oDBSearch.BaseType = "VPMReference"
oDBSearch.AddEasyCriteria = "PLM_ExternalID", "Ship*"
```

The **BaseType** property sets the type of the PLM objects to search for as `VPMReference`, that is, Physical Product. The **AddEasyCriteria** sub sets an additional criterion to search for **VPMReference** objects for which the value of the **PLM_ExternalID** attribute, which stores the value of the `Name` property, starts with the character string Ship, since the "*" wild character is used.

Then launch the search using the **Search** sub of the **SearchService** object.

```
...
oSearchService.Search
...
```

Finally use the **Results** property of the **DatabaseSearch** object to retrieve the search results in a **PLMEntities** collection object.

```
...
Dim cPLMEntities As PLMEntities
Set cPLMEntities = oDBSearch.Results
...
```

Predefined Queries

First set the Predefined Queries mode thanks to the **Mode** property of the **DatabaseSearch** object.

```
...
oDBSearch.Mode = SearchMode_Predefined
...
```

Then create the search criteria.

```
...
oDBSearch.AddPredefinedCriteria "prd:Ship"
...
```

The **AddPredefinedCriteria** method sets two criteria separated by a column ":". The first character string is a shortcut of the type of the objects to search, and the second one is a character string contained in the value of one of the attributes `Title`, `Name`, `Description`, or `Responsible`. In this example, `prd` stands for Physical Product.

To see a list of all predefined types for the application in which you are working, in the `Search` options list, select `What to Search for > More`. You can see the predefined query shortcuts associated with all predefined types.

Advanced Search - Easy

This is the default mode. You can set it explicitly using the **Mode** property of the **DatabaseSearch** object.

```
...
oDBSearch.Mode = SearchMode_Easy
...
```

Then create the search criteria. First set the type of objects to search using the **BaseType** property of the **DatabaseSearch** object.

```
...
oDBSearch.BaseType = "VPMReference"
...
```

The types you can use are those available in the `Object` list of the `SEARCH` dialog box that you open from the `Search` options list by selecting `How to Search > Advanced Search`. The character string to use for a type is the attribute value taken from the Unified Typing Reference Dictionary. In this example, `VPMReference` stands for Physical Product.

Finally create a criterion relying on one of the attributes of the object to search, with a character string to be found in the value of this attribute using the **AddEasyCriteria** sub of the **DatabaseSearch** object.

```
...
oDBSearch.AddEasyCriteria "V_Name", "engine"
...
```

The attributes are declared using their name in the Unified Typing Reference Dictionary. For example, `V_Name` stand for the `Title` property. The second parameter contains the character string to be searched for in the V_Name attribute values. Note that you can use the **AddEasyCriteria** sub several times with a different couple attribute/value each time. They are combined using an AND operator.

Advanced Search - Extended

First set the Advanced Search - Extended mode thanks to the **Mode** property of the **DatabaseSearch** object.

```
...
oDBSearch.Mode = SearchMode_Extended
...
```

Then create the search criteria. First set the type of objects to search using the **BaseType** property of the **DatabaseSearch** object, and possibly the extension type using the **Extension** Property.

```
...
oDBSearch.BaseType  = "Drawing"
oDBSearch.Extension = "CATDftGenDraftingExt"
...
```

The types you can use are those available in the `Object` list of the `SEARCH` dialog box that you open from the `Search` options list by selecting `How to Search > Advanced Search`, then checking `Advanced`, finally clicking `Extended`. The character string to use for a type is the attribute value taken from the Unified Typing Reference Dictionary. In this example, `Drawing` stands fortunately for `Drawing`.

The extension you can use are those available in the `Extension` list of the `SEARCH` dialog box. In this example, `CATDftGenDraftingExt` stands for `Drawing additional info`.

Create a criterion relying on one of the attributes of the object to search, for example with a character string to be found in the value of this attribute using the **AddExtendedCriteria** sub of the **DatabaseSearch** object.

```
...
oDBSearch.AddExtendedCriteria "V_Name", "engine", SearchOperator_LIKE
...
```

The attributes are declared using their name in the Unified Typing Reference Dictionary. For example, `V_Name` stands for the `Title` property. The second parameter contains the character string to be searched for in the V_Name attribute values. The third parameter is the operator to be used, here Like. The available operators are declared thanks to the SearchOperator enumeration.

You can create several criteria, such as the following:

```
...
oDBSearch.AddExtendedCriteria "originated", "01/06/2014", SearchOperator_EQ
...
```

In this example, `originated` stands for the `Creation date` property. The second parameter contains the date to search for, and the third parameter is the operator to be used, here equal to. This searches for drawings created on this date. The available operators are declared thanks to the SearchOperator enumeration.

You can also create a criterion using a range of values:

```
...
oDBSearch.AddExtendedRangeCriteria "modified", "01/07/2014", "01/10/2014", SearchOperator_BETWEEN
...
```

In this example, `modified` stands for the `Last modification` date property. The second and third parameters contain the date range to search for, and the third parameter is the operator to be used, here between to. This searches for drawings modifieded between these two dates. The available operators are declared thanks to the SearchOperator enumeration.

You can the apply an operator between these criteria.

```
...
oDBSearch.Condition = SearchCondition_AND
...
```

In this example, the AND operator is applied. This searches for the drawings with the extension `Drawing additional info`. that include the character string "engine" in their `Title` property and that were created on the 01/06/2014 and that were modified between the 01/07/2014 and the 01/10/2014.

The available operators are AND and OR, and are declared thanks to the SearchCondition enumeration.

Advanced Search - Expert

First set the Advanced Search - Expert mode thanks to the **Mode** property of the **DatabaseSearch** object.

```
...
oDBSearch.Mode = SearchMode_Expert
...
```

Then create the search criteria.

```
...
```

```
oDBSearch.BaseType  = "Drawing"
oDBSearch.Extension = "CATDftGenDraftingExt"
...
```

Then create the search criteria.

```
...
oDBSearch.SetExpertExpression "(`Drawing`.""Title"" Like ""engine"" AND `Drawing`.""Name"" NotLike ""body"")" _
                              & "OR `Drawing`.""Creation date"" == Date(""01/06/14"")"
...
```

You can copy such a query from the `Expert` tab of the `SEARCH` dialog box. Pay attention that this query is localized to your region and language in the `SEARCH` dialog box, and that the macro must be run using the same localization. In addition, replace each double quotes coming from the `SEARCH` dialog box with two double quotes, and enclose it with double quotes.

If you want to display a long query on several lines as shown above, use the _ character to make two lines a single statement, and use the & operator to concatenate the two character strings in a single one.

Launching the Search

Once your criteria are created, you can launch the search thanks to the **Search** sub of the **SearchService** object.

```
...
oSearchService.Search
...
```

Retrieving the Search Results

When the search completes and returns results, they are displayed in the Search Results. You can retrieve these results as **PLMEntity** objects in a **PLMEntities** collection object using the **Results** properties of the **DatabaseSearch** object.

```
...
Dim cPLMEntities As PLMEntities
Set cPLMEntities = oDBSearch.Results
...
```

Note that you can get properties from these **PLMEntity** objects, but not set them. To do this, open these object using the **PLMOpen** sub of the **PLMOpenService** object.

Search Mode Dedicated Properties and Subs

The following properties and subs of the **DatabaseSearch** object are dedicated to one or several search modes. The others are mode independent.

Property or Sub	Predefined Queries	Advanced Search		
		Easy	Extended	Expert
BaseType	No	Yes	Yes	Yes
Condition	No	No	Yes	No
Extension	No	Yes	Yes	Yes
AddEasyCriteria	No	Yes	No	No
AddExtendedCriteria	No	No	Yes	No
AddExtendedRangedCriteria	No	No	Yes	No
AddPredefinedCriteria	Yes	No	No	No
SetExpertExpression	No	No	No	Yes

Searching PLM Objects

This use case searches for PLM objects from the underlying database, and illustrates the failure to set a PLM attribute to a PLM object if it is not loaded in the current session.

Related Topics
Launching an Automation Use Case
SearchService Object
DatabaseSearch Object

Before you begin: Note that:

- You should first launch 3DEXPERIENCE

Where to find the macro: CAAScdSrvUcSearchPLMObjectsSource.htm

1. **Prolog**

   The macro is made up of the **SearchPLMObjects** sub. It typically begins with the following extract of code, which serves as a built-in "error handling" mechanism.

   ```
   Sub SearchPLMObjects()

     'Error handling
     On Error GoTo ErrorSub
   ...
   ```

   This instruction means that when a method will throw an error, the macro will skip to the line named, **ErrorSub**. See the step Epilog.

2. **Retrieving the SearchService object**

   The first step is to retrieve a **SearchService** object from the session using the **Application** object.

   ```
   ...
     Dim oSearchService As SearchService
     Set oSearchService = CATIA.GetSessionService("Search")
   ...
   ```

   The **SearchService** object is retrieved from the **Application** object (CATIA) using the `GetSessionService` function.

3. **Retrieving the DatabaseSearch object**

   The **DatabaseSearch** object is aggregated to the **SearchService** object to build and hold the search and the search results.

```
...
 Dim oDBSearch As DatabaseSearch
 Set oDBSearch = oSearchService.DatabaseSearch
...
```

   The **DatabaseSearch** object is retrieved from the **SearchService** object thanks to the **DatabaseSearch** property.

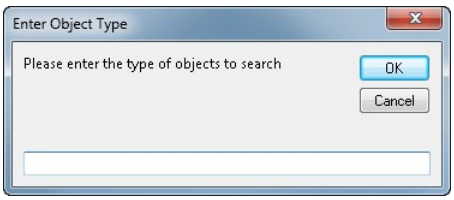
4. **Building the search**

   Several search modes exist. The Advanced Search Easy mode is used here. This search mode is the default one, so it does not need to be explicitly set. The search built here searches for objects of a given type, and the names of which include a given character string. Both type and name are asked from the end user who will type each of them in an InputBox.

   First ask the end user to enter an object type.

```
...
 Dim strObjectType As String
 strObjectType = InputBox("Please enter the type of objects to search", "Enter Object Type")
...
```

   Fig. 1 Dialog to Input Object Search Type



   The input object type is retrieved using the `strObjectType` variable. You can use any customization type available in your environment. To search for physical products, for example, type VPMReference.

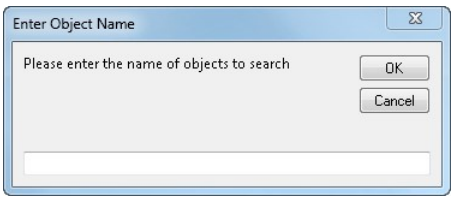
```
...
 oDBSearch.BaseType = strObjectType
...
```

   This type is assigned to the search using the **BaseType** property of the **DatabaseSearch** object.

   Then ask the end user to enter a value for the Name property, or a substring of this value.

```
...
 Dim strObjectName As String
 strObjectName = InputBox("Please enter the name of objects to search", "Enter Object Name")
...
```

   Fig. 2 Dialog to Input Object Name



   The input object name is retrieved using the `strObjectName` variable. The Name property is internally represented using the PLM_ExternalID attribute.

```
...
 oDBSearch.AddEasyCriteria "PLM_ExternalID", strObjectName
...
```

   It is assigned to the search using the **AddEasyCriteria** sub of the **DatabaseSearch** object.

5. **Launching the search**

```
...
 oSearchService.Search
...
```

   The **Search** sub of the **SearchService** object launches the search. The list of found objects is displayed in the Search results.

6. **Retrieving the search results**

```
...
 Dim cObjects As PLMEntities
 Set cObjects = oDBSearch.Results
...
```

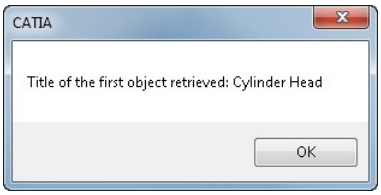
   The **Results** property of the **DatabaseSearch** object returns the PLM objects of the database matching the criteria in a **PLMEntities** collection object. This collection contains references to all the PLM objects found and displayed in the Search results.

7. **Displaying the title of the first object in the collection**

```
...
 MsgBox "Title of the first object retrieved: " & cObjects.Item(1).GetAttributeValue("V_Name")
...
```

The **GetAttributeValue** function of the first **PLMEntity** object of the collection (`cObjects.Item(1)`) retrieves its title stored using the value of the **V_Name** attribute and displays it in the message box as for example in the image below.

Fig. 3 Title of the First Object Retrieved



8. **Attempting to set PLM Attribute V_Name through a PLM entity listed in the Search Results**

```
...
  cObjects.Item(1).SetAttributeValue "V_Name", "New Name"
...
```

A call to the **SetAttributeValue** sub on a **PLMEntity** object (`cObjects.Item(1)`), attempts to set the the value "New Name" to the attribute "V_Name". But this call fails. This is because the objects retrieved from a search and handled as **PLMEntity** objects are not PLM objects, but references to the PLM objects in the database. See Epilog.

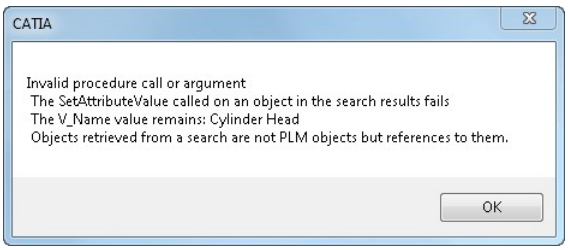
9. **Epilog**

The following extract of code typically occurs at the end of the subroutine. It is primarily meant for error-handling purpose. Any run time error that the macro encounters, results in the execution flow reaching this part of the code, and then terminating with a normal exit from the sub scope.

```
...
'Skip to the end of the macro
  GoTo EndSub

'Line reached when an error is thrown by a method
ErrorSub:
  MsgBox Err.Description _
  + vbCrLf + " The SetAttributeValue called on an object in the search results fails" _
  + vbCrLf + " The V_Name value remains: " + PLMComps.Item(1).GetAttributeValue("V_Name") _
  + vbCrLf + " Objects retrieved from a search are not PLM objects but references to them."

'Last line of the macro
EndSub:
End Sub
```

Usually the statement `MsgBox Err.Description` is used alone. In this case, the **SetAttributeValue** sub called against a **PLMEntity** object will always fail, because it is not the PLM object itself, but a reference to it, and attribute values cannot be set from such references.

Fig. 4 Error Issued when Attempting to Set a New Name to a PLM Object from the Search Results



To set attributes to, and work with PLM objects, use the **PLMOpen** sub of the **PLMOpenService** object to open the **PLMEntity** object in the session.

Adopting 3DEXPERIENCE PLM Search Objects

This task shows you how to adopt the search objects new in **3D**EXPERIENCE R2014x in place of the **PLMSearchService**, **PLMSearches**, and **PLMSearch** objects.

Related Topics
SearchService Object
DatabaseSearch Object

Before you begin:

- You should first locate in the database the macros with which you use PLM search objects.

1. Retrieving the search service object.

Replace the **PLMSearchService** object with the **SearchService** object, and **PLMSearch** argument with **Search**.

Up to V6R2014	Starting with 3DEXPERIENCE R2014x
`Dim oSearchService As PLMSearchService`	`Dim oSearchService As SearchService`
`Set oSearchService = CATIA.GetSessionService("PLMSearch")`	`Set oSearchService = CATIA.GetSessionService("Search")`

The **PLMSearchService** object is replaced with the **SearchService** object, still retrieved from the **Application** object using its **GetSessionService** method with **Search** as argument instead of **PLMSearch**.

2. Retrieving the object holding the query.

Replace the **PLMSearches** and **PLMSearch** objects with the single object **DatabaseSearch**.

Up to V6R2014	Starting with 3DEXPERIENCE R2014x
`Dim oSearches As PLMSearches`	
`Set oSearches = oSearchService.Searches`	`Dim oDBSearch As DatabaseSearch`

```
                                                Set oDBSearch = oSearchService.DatabaseSearch
Dim oSearch As PLMSearch
Set oSearch = oSearches.Add
```

Rather than retrieving a **PLMSearches** collection object from the **PLMSearchService** object and adding a **PLMSearch** object to the collection, a single **DatabaseSearch** object is aggregated to the **SearchServices** object, and returned thanks to the **DatabaseSearch** property.

3. Building the query.

   Replace the **PLMSearch** object used by the query construction with the **DatabaseSearch** object, as well as the property and method names.

Up to V6R2014	Starting with 3DEXPERIENCE R2014x (for example using R2017x equivalent type)
`oSearch.Type = "PLMProductDS"`	`oDBSearch.BaseType = "VPMReference"`
`oSearch.AddAttributeCriteria "PLM_ExternalID", "Ship*"`	`oDBSearch.AddEasyCriteria "PLM_ExternalID", "Ship*"`

   The PLM Object type is declared using the **BaseType** property instead of Type, and a simple criteria to query for PLM object having a given value for a PLM attribute is set using the **AddEasyCriteria** method instead of **AddAttributeCriteria**. Note that other methods exist to create more complex criteria that with the **PLMSearch** object, such as **AddExtendedCriteria** or **AddExtendedRangeCriteria**.

4. Launching the query.

   Replace the **PLMSearch** object launching the search with the **SearchService** object

Up to V6R2014	Starting with 3DEXPERIENCE R2014x
`oSearch.Search`	`oSearchService.Search`

   The search query is still launched using the **Search** method.

5. Retrieving the results.

   Replace the **PLMSearch** object used to retrieve the search result with the **DatabaseSearch** object, as well as the property name.

Up to V6R2014	Starting with 3DEXPERIENCE R2014x
`Dim cPLMEntities As PLMEntities`	`Dim cPLMEntities As PLMEntities`
`Set cPLMEntities = oSearch.EditedContent`	`Set cPLMEntities = oDBSearch.Results`

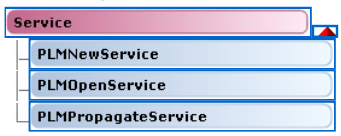
   The results are retrieved from the **DatabaseSearch** object using the **Results** properties instead of the service object, still as a **PLMEntities** collection object.

In Short

This task shows you how to replace in your macros the pre-**3D**EXPERIENCE R2014x **PLMSearchService**, **PLMSearches**, and **PLMSearch** objects with the **SearchService** and **DatabaseSearch** objects.

PLM Session Builder Object Model Map

See Also Legend



PLMNewService Object

See Also Legend Use Cases Properties Methods

Represents a service to create PLM objects.

The **PLMNewService** object lets you:

- Create a new PLM object in session.
- Retrieve the **Editor** object managing this PLM object.

Retrieving the PLMNewService Object

Use the **GetSessionService** of the **Application** object to return a **PLMNewService** object.

```
Dim oPLMNewService As PLMNewService
Set oPLMNewService = CATIA.GetSessionService("PLMNewService")
```

Refer to the Service Object.

Using the PLMNewService Object

Use the **PLMCreate** sub to create a PLM object.

```
...
Dim oEditor As Editor
oPLMNewService.PLMCreate "3DShape", oEditor
...
```

This example create a 3DShape representation by passing the 3DShape type.

Use the **ActiveObject** property of the **Editor** object to retrieve the **Part** object of the created **3DShape** object.

```
...
Dim oPart As Part
Set oPart = oEditor.ActiveObject
...
```

PLMOpenService Object

See Also Legend Use Cases Properties Methods

Represents a service to open PLM objects.

The **PLMOpenService** object lets you:

- Open a PLM object in session.
- Retrieve the **Editor** object managing this PLM object.

Retrieving the PLMOpenService Object

Use the **GetSessionService** of the **Application** object to return a **PLMOpenService** object.

```
Dim oPLMOpenService As PLMOpenService
Set oPLMOpenService = CATIA.GetSessionService("PLMOpenService")
```

Refer to the Service Object.

Using the PLMOpenService Object

Use the **PLMOPen** sub to open a PLM object, for example, using a **PLMEntity** object obtained from the Search Results using the Results property of the **DatabaseSearch** object. See the DatabaseSearch Object.

```
...
Dim cPLMEntities As PLMEntities
Set cPLMEntities = oDBSearch.Results
...
Dim oPLMEntity As PLMEntity
Set oPLMEntity = cPLMEntities.Item(1)

Dim oEditor As Editor
oPLMOpenService.PLMOpen oPLMEntity, oEditor

Dim oRootOcc As VPMRootOccurrence
Set oRootOcc = oEditor.ActiveObject

Dim oRootProduct As VPMReference
Set oRootProduct = oRootOcc.PLMEntity
...
```

This example retrieves the Search Results, supposed to be the results of a search for **VPMReference** objects, in a **PLMEntities** collection object.

The first item of the collection is opened using the **PLMOpen** sub of the **PLMOpenService** object and retrieved as a **VPMRootOccurrence** object of the occurrence model thanks to the **ActiveObject** property of the **Editor** object.

To handle it as a **VPMReference** object of the instance reference model, use the **PLMEntity** property of the **VPMRootOccurrence** object.

PLMPropagateService Object

See Also Legend Use Cases Properties Methods

Represents a service to save PLM objects.

The **PLMPropagateService** object lets you save in the database changes to PLM objects opened in the active editor.

Retrieving the PLMPropagateService Object

Use the **GetSessionService** method of the **Application** object to return a **PLMPropagateService** object.

```
Dim oPLMPropagateService As PLMPropagateService
Set oPLMPropagateService = CATIA.GetSessionService("PLMPropagateService")
```

Refer to the Service Object.

Using the PLMPropagateService Object

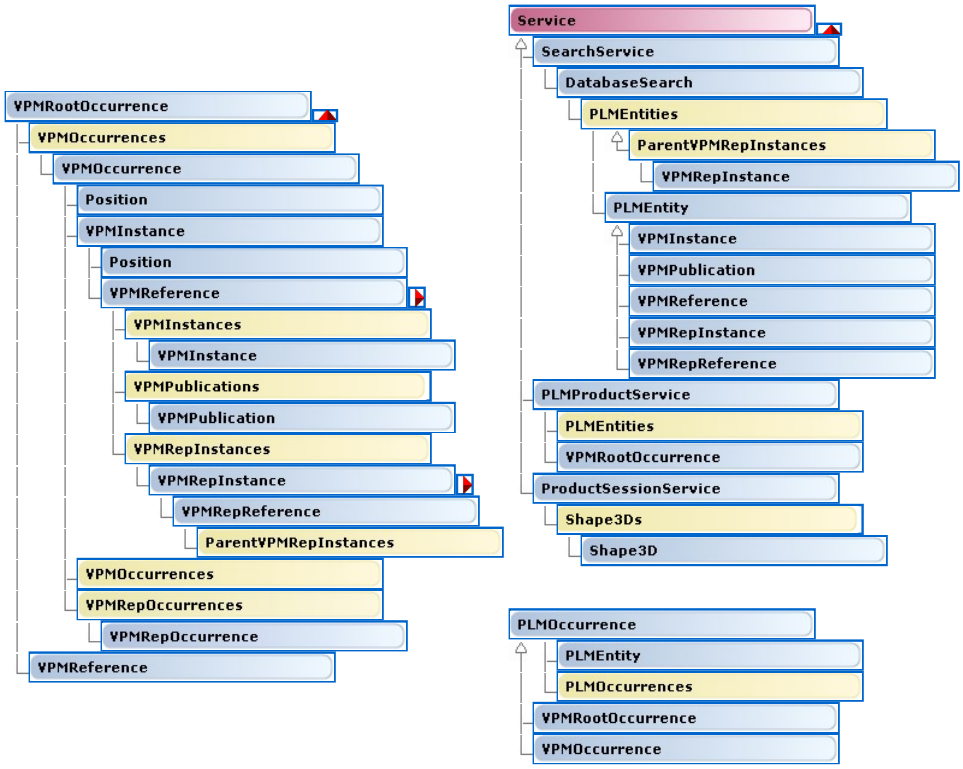
Use the **PLMPropagate** method of the **PLMPropagateService** object to save changes to objects opened in the active editor.

```
...
Dim oPLMPropagateService As PLMPropagateService
Set oPLMPropagateService = CATIA.GetSessionService("PLMPropagateService")
oPLMPropagateService.PLMPropagate
...
```

This example retrieves the **PLMPropagateService** object from the **Application** object thanks to its **GetSessionService** method, and the changed objects opened in the active editor are saved in the database using the **PLMPropagate** method.

Product Object Model Map

See Also Legend



Product Modeler Overview

Technical Article

Abstract

The Product modeler is one of the main V6 PLM modelers. This article gradually introduces the concepts onto which the Product modeler relies, along with its objects and their inheritances from the PLM Core modeler (root of any PLM modeler), using a simple assembly example as red wire. After a first chapter showing what the industrial world requires from a Product modeler, the two others describe the two models making up the V6 Product modeler: the instance/reference model used to store product data in the database, and the occurrence model used to display and edit it in session.

The Industrial Requirements

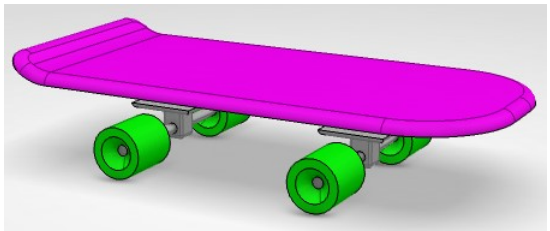
The industrial world main requirements against a modeler intended to handle industrial products are to enable:

- The assembled physical model, that is, the parts, their relative locations in the assembly, and the constraints that can be necessary between some objects.
- Versioning: the different products that can be manufactured, assembled, and sold from a single physical model.
- Configuration: the changes brought to the physical model during its life time and these changes effectivities.
- Part manufacturing.
- Simulations.
- Bill of materials.
- Assembly process.

This article deals with the first of these requirements. Also, if a modeler includes objects and their behaviors, this article focuses on the objects only.

The Skateboard below is used throughout this article to illustrate the models.

Fig. 1: The Skateboard



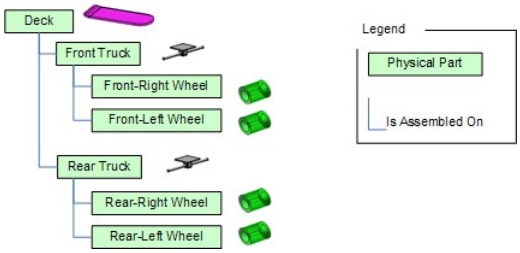
This simplified skateboard chosen here is made up of seven assembled **physical parts** belonging to three types:

1. A deck (pink).
2. Two trucks (gray).
3. Four wheels (green).

From the Industrial Actual Model to the Persistent Model

These seven physical parts can be schematically represented as a tree shown below.

Fig. 2: The Skateboard Made up of Physical Parts



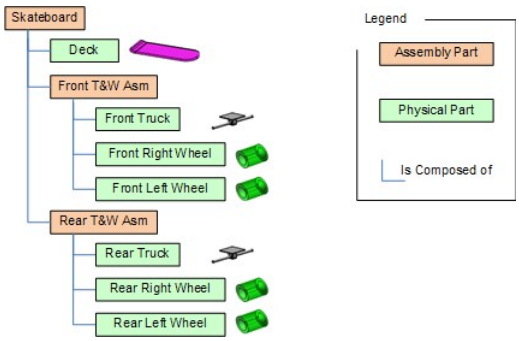
Each part is linked with those that are assembled on it. For example, the front truck is linked with the two front wheels assembled on it. This chapter shows how the persistent data structure is built from this assembly. It first describes how to model the assembly, then how to link the assembly to the parts, how to set up constraints between parts, and finally how a part can publish one of its internal elements for some others to rely on.

Modeling an Assembly

The skateboard is created by assembling seven parts. This assembly itself is an object. In the same way, to handle a truck and wheel assembly as a whole, an object is required in addition to the truck and the two wheels. Such objects called **assembly parts** are made up of named sets of physical parts. Assembly parts have many qualities, such as holding a part number to order it as a whole assembled subset rather than ordering each part individually and assembling them.

In the skateboard, the assembly part named Truck & Wheel assembly (`T&W Asm`) is made up of a truck, a left wheel, and a right wheel. In the same way, the assembly part `Skateboard` is made up a deck, a front `T&W Asm`, and a rear `T&W Asm`.

Fig. 3: The Skateboard with Assembly Parts



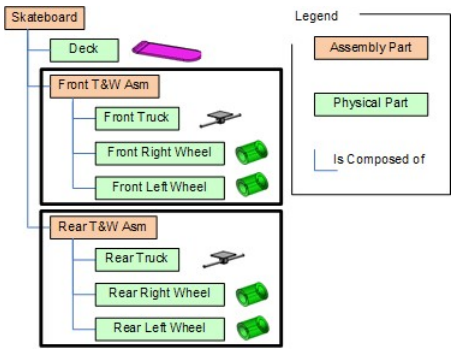
There is now a clear distinction between the deck, which is a physical part, and the skateboard as an assembly of seven physical parts. This distinction did not appear on the [Fig. 2]. Note that the relation between the parts has changed, moving from "is assembled on" to "is composed of".

The seven initial parts and the three added assembly parts make up now a ten part model.

Note also that on the [Fig. 3], assembly parts have no associated image compared with physical parts. They could be represented only using a combination of the images of the physical part they aggregate.

When looking closer at the model, you can see that the two Truck & Wheel Assemblies are identical. Only their respective locations differ.

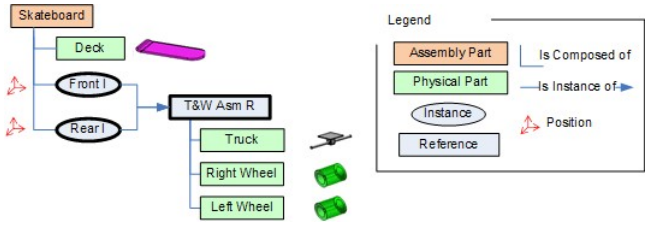
Fig. 4: The Skateboard with Common Parts



The model above highlights these two identical sets enclosed by thick boxes. (Such symbols with thick outlines will show the model evolutions throughout this article.)

These two identical sets enable a model simplification, avoiding object duplication.

Fig. 5: The Skateboard First Simplification

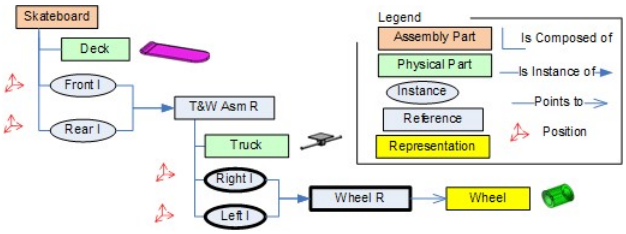


The two `Front T&W Asm` and `Rear T&W Asm` of the [Fig. 4] become two **instances**, `Front I` and `Rear I` of the same `T&W Asm R` **reference**, this reference being the assembly part. The two instances differ only by their locations symbolized using the coordinate systems the origins and director vectors of which are expressed in the absolute coordinate system associated with the `Skateboard`. These instance and reference objects build an instance/reference model graphically shown as a graph that will progressively replace the initial tree.

Note that the instances `I` are represented with an oval, and the references `R` with a rectangle.

Another simplification is possible.

Fig. 6: The Skateboard Second Simplification



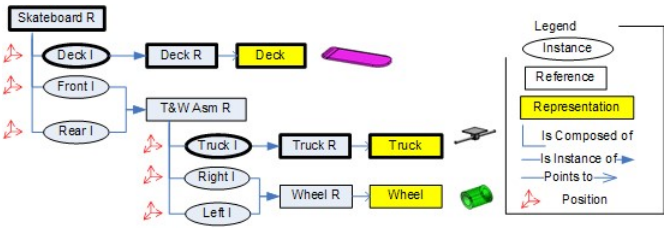
The two `Right Wheel` and `Left Wheel` parts become two instances, `Right I` and `Left I` of the same wheel reference `Wheel R`. The two instances have their own locations in the `T&W Asm R` context, another way to say "expressed in the `T&W Asm R` coordinate system".

This model clearly shows up a single wheel reference describing the wheel whatever its instance locations in the assembly. This is not obvious in the first model on [Fig. 2] where nothing shows that the four wheels are identical. The wheel reference holds reference wheel data, such as its part number, and points to the wheel **representation** (yellow box) that gives shape and geometry.

The reference objects, common to the assembly and physical parts, enable the model consistence. The reference of an assembly part that aggregates instances, or the reference of a physical part that points to a representation are of the same nature. If the first one will still be called an assembly part, the second one will simply be called part.

Generalizing the reference addition to the other parts replaces all the physical parts with instances, references, and representations.

Fig. 7: The Skateboard Instance/Reference/Representation Model

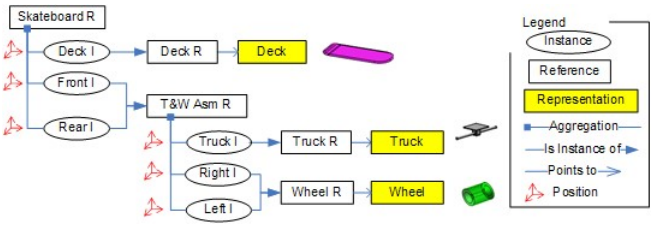


The physical part `Deck` is replaced with a set of objects: a `Deck` instance, aggregated by the `Skateboard` reference, and instance of a `Deck` reference that in turn points to a `Deck` representation. The same replacement happens with the `Truck` physical parts. In addition, the `Skateboard` part is replaced with a `Skateboard` reference, root of the assembly.

Starting from a physical model containing only geometric and duplicated data and applying the different changes described above lead to the resulting current model describing the assembly, and pointing to the part geometry, with a minimum of non redundant objects.

The data model representing the real model begins to take shape. It uses the conventions below in the following:

Fig. 8: The Skateboard Product Instance/Reference/Representation Model



To summarize, this model features:

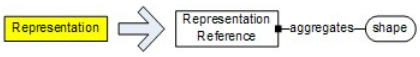
- **References** to model either a part or an assembly part.
- **Instances** to model the presence of a reference at a given location with respect to the reference it is assembled with.
- **Representations** to model the shape and geometry of a part.
- **Links** between instances and references:
  - **Aggregation**: an instance is always aggregated by a reference. (where previously we said "is Composed of"
  - **Is Instance of**: an instance is always related to a reference.
- **Pointers** from references to their representations.

This last link shall now be detailed.

Modeling Representations

The main mission of representations is to embody the parts by describing their shapes and geometries. Since these shapes and geometries do not belong to the Product modeler, but to the Part modeler, the representation object is split into a **representation reference** in the Product modeler and a **shape** object in the Part modeler.

Fig. 9: Representation Reference and Shape

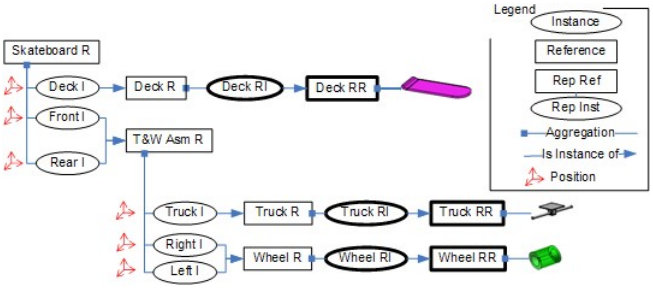


These two objects are not logically distinct. Even if the representation reference is an access handle to the shape, that is, a pointer, it actually encapsulates the shape with an aggregation behavior. If the representation reference is deleted, the shape is also deleted.

The representation reference is a leaf object, and thus is not semantically identical to the reference which is a structuring object.

To integrate the representation reference in the instance/reference model, it is naturally instantiated thanks to a dedicated instance: the **representation instance**.

Fig. 10: The Skateboard Full Instance/Reference Model



The representations in [Fig. 7] are replaced with a representation instance, aggregated by a reference, and the instantiated representation reference that aggregates the shape. The word representation is used to designate these three object as a whole.

Representation reference objects enable references:

1. To only aggregate instances, that is, to be either node objects aggregating instances from references or leaf objects aggregating representation instances.
2. To have a dedicated life cycle. They may be versioned, changed, or deleted independently of the representation references. For example, deleting a reference must not delete the representation reference and its aggregated shape.
3. To be multi-represented. A reference can aggregate several representations, such as an exact representation of the actual part, and a light representation for display and interference checks. As an example of the multi-representation, the wheel can have an exact representation, a light representation such as a cylinder representing its bounding block, graphics data for an exact but light display, the results of a simulation, or a drawing. Note that all the representations are equivalent for the Product modeler. The difference between them is set by the applications.

To summarize, the seven physical objects in [Fig. 2] are now 17 Product modeler objects of four types:

1. Two **references** for the assembly parts: `Skateboard R` and `T&W Asm R`, and three **references** for the parts: `Deck R`, `Truck R`, and `Wheel R`.
2. Six **instances**: `Deck I` for the deck, `Front I` and `Rear I` for the Truck & Wheel assemblies, `Truck I` for the truck, `Right I` and `Left I` for the wheels.
3. Three **representation instances**: `Deck RI`, `Truck RI`, and `Wheel RI`.
4. Three **representation references**: `Deck RR`, `Truck RR`, and `Wheel RR`.

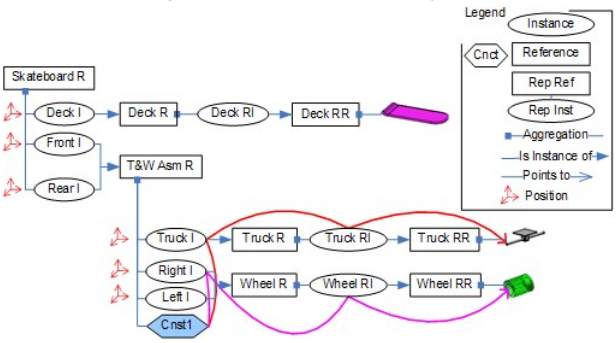
And the three shapes of the Part modeler that are not handled by the Product modeler.

Now if you want to figure out how this model really is, you need to think a bit about it. For example, its wheel count is not immediately deduced form the wheel instance count: two wheel instances, but four real wheels. Because it is a sparing model, you cannot address each real object directly. You must interpret it by expanding the different objects, running along the links from the root to the leaf objects to count actual objects. For example, starting from `Skateboard R` through `Front I`, `T&W Asm R`, up to `Right I` and its representation shows the front right wheel, and so on for the others. This model must thus be interpreted to understand the real model, for example, to display it. This is discussed in From the Persistent Model to the Session Model, but two other assembly design requirements should be fulfilled before.

Modeling Constraints and Publications

The assembly is built by positioning parts the ones with respect to the others. This is necessary, but sometimes not enough. For example, wheel and truck axle rotate sharing the same rotation axis and thus must be coaxial. If you move the wheel, it must remain coaxial with the truck shaft. To ensure this, you can set a constraint using the Assembly Design workbench between a wheel and the truck, assumed here for simplification to be identical with its axle.

Fig. 11: Constraints between two Objects

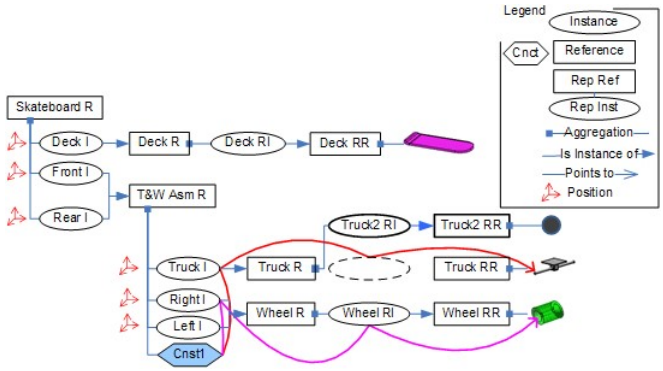


The constraint is naturally added in the Product structure graph at the same level than the objects it constrains. In the example, the constraint (`Cnst1`) points to the truck and wheel rotation axes in their respective shapes using paths passing by the instance, jumping to the representation instance, the representation reference, and finally to the shape. To complete the design, a second constraint should be set to constraint the left wheel and the truck to be coaxial.

From the PLM Core modeler standpoint, the constraint is a **connection**, a more generic object to model link patterns between objects within a PLM modeler, or from different PLM modelers. Another classical connection usage is associating a material to a reference.

Now assume that the truck part owner changes the shape geometry, for example by replacing this shape with another one.

Fig. 12: Change a Shape

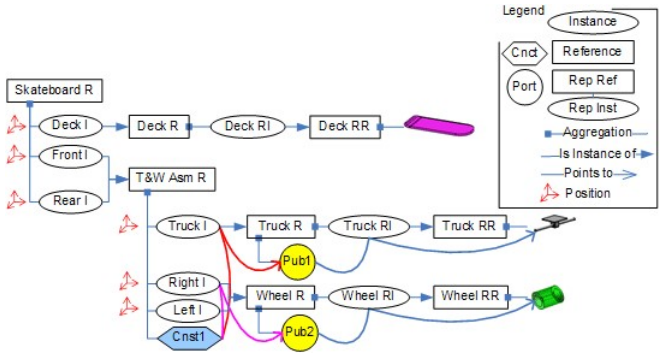


The representation instance aggregated by the `Truck R` reference is changed to `Truck2 RI` of the newly created `Truck2 RR` representation reference aggregating the new Shape. The previous instance is deleted and the constraint `Cnst1` points now a disappeared instance: the constraint is broken.

The assembly owner can of course manually reroute the constraint to the new representation instance and new Shape. This implies the truck part owner and the assembly owner, if they are not the same person, to communicate about the design changes. In addition, if the truck part is pointed by several assemblies belonging to different owners, the communication process will shortly turn to a nightmare. It is thus much simpler and more efficient to use an intermediate object named **publication** to make the pointing mechanism endure.

Using a publication, the `Truck R` reference owner ensures that the part will always offer to assembly designers the truck rotation axis, whatever the changes he/she could bring to the shape itself. The publication is a kind of interface the part exhibits assorted with a perennial contract assembly designers can rely on to set their constraints against that axis. The same applies to the wheel.

Fig. 13: The Publications to Make the Model Easily Usable



Instead of pointing the shape through the instances only, the constraint now points the two publication objects which make up a stable contract to access a geometric element inside a shape.

From the PLM Core modeler standpoint, the publication is a **port**, a more generic object that can be used to model other data patterns. Another classical usage of ports is to interface two systems in the logical or functional modelers.

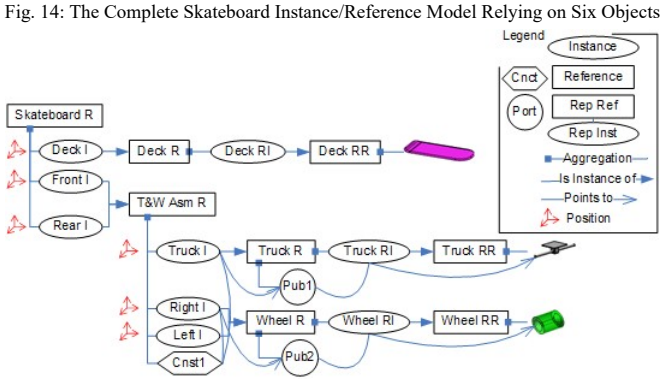
The Instance/Reference Model

Thanks to this simple skateboard model, but without excessive simplification, the resulting data model, except for the shapes, relies on six object types making up the PLM Core modeler:

1. Reference
2. Instance
3. Representation reference
4. Representation instance
5. Connection
6. Port.

Any modeler built from this six objects is an instance/reference model, such as the Product modeler, but also the Process, Logical, and Functional modelers.

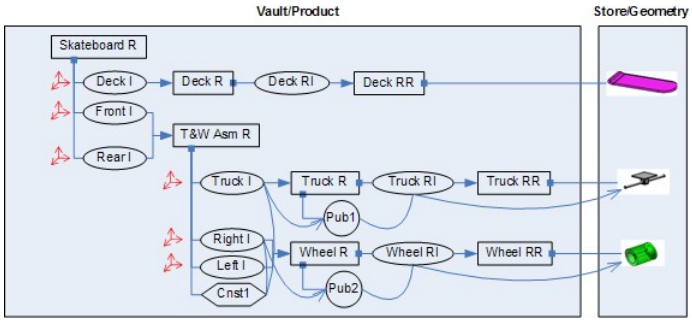
Fig. 14: The Complete Skateboard Instance/Reference Model Relying on Six Objects



Even if this example is simple compared to an industrial assembly, featuring three instance/reference levels only, it represents the objects you will handle and the mechanisms you will deal with when designing, browsing, or scanning your assemblies.

Such an assembly is created and designed during a **client session**, and saved in the **database**. The assembly is saved as an instance/reference model in the **Vault server**, while the shapes and their geometries are streamed in the **Store server**.

Fig. 15: The Vault and Store Servers

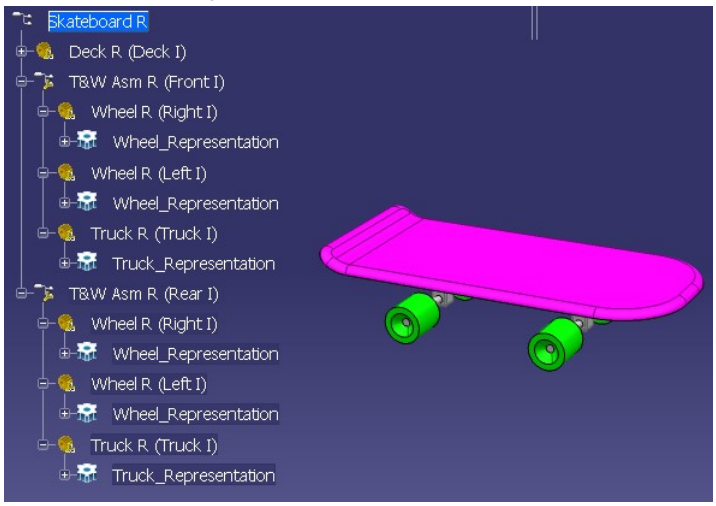


To know more about this model, see Product Instance/Reference Model [1].

From the Persistent Model to the Session Model

If the instance/reference model synthetically represents an assembly and efficiently saves it in a database, this model does not match the session requirements. The picture below shows the skateboard in the CATIA 3D editor window.

Fig. 16: The Skateboard in a Client Session



The skateboard is shown as a 3D model and as a tree structure, usually named the specification tree. These are two views of the same model in session.

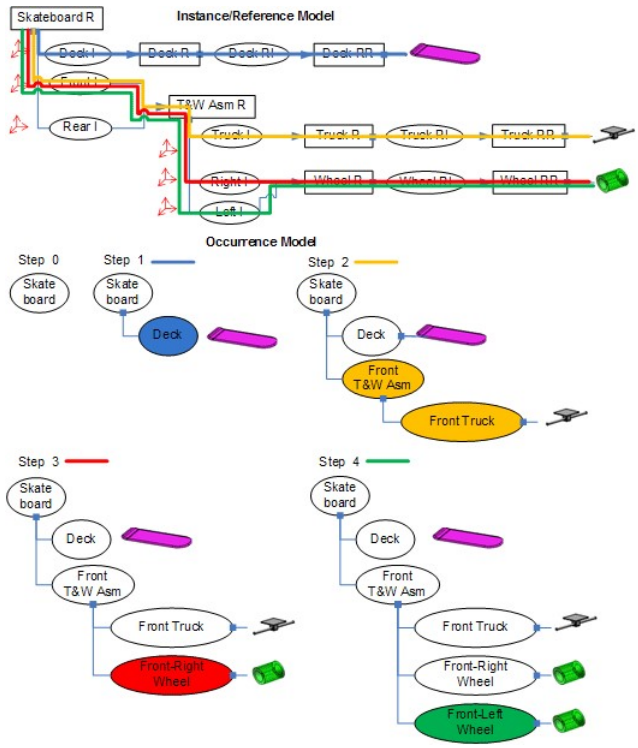
On the 3D view, you can see that the skateboard has four wheels, and not two if you just look at the instance/reference model without interpreting it. This 3D view is thus not a rough view of the instance/reference model. It is built on a model named the **occurrence model**, created from the instance/reference model.

The occurrence model is not persistent, that is, not saved in the database. It is rebuilt whenever the instance/reference model is read from the database and loaded into the session. Both models reside in the session memory and can be accessed, either interactively or programmatically.

The Occurrence Model

It is created from the instance/reference model loaded from the database to the session. The following image shows how.

Fig. 17: The Occurrence Model Build



To create the occurrence model, the instance/reference model is expanded or unfolded by running along all the possible paths from the root object to the leaf objects. The colored lines shows these paths.

- **Step 0**: Create an occurrence for the root reference.
- **Step 1**: Running along the blue path from the root through `Deck I`, `Deck R`, `Deck RI`, up to `Deck RR` and its pointed part to create the `Deck` occurrence.
- **Step 2**: Running along the orange path:
  - From the root through`Front I` to `T&W Asm R` to create the Front `T&W Asm` occurrence.
  - Then from `T&W Asm R` through `Truck I`, `Truck R`, `Truck RI`, up to `Truck RR` and its pointed part to create the `Front Truck` occurrence.
- **Step 3**: Running along the red path:
  - From the root through `Front I` to `T&W Asm R` , leaving the Front `T&W Asm` occurrence created in step 2.
  - Then from `T&W Asm R` to `Right I`, `Wheel R`, `Wheel RI`, up to `Wheel RR` and its pointed part to create the `Front Right Wheel` occurrence.
- **Step 4**: Running along the green path:
  - From the root through `Front I` to `T&W Asm R`, leaving the Front `T&W Asm` occurrence created in step 2.
  - Then from `T&W Asm R` to `Left I`, `Wheel R`, `Wheel RI`, up to `Wheel RR` and its pointed part to create the `Front Left Wheel` occurrence.

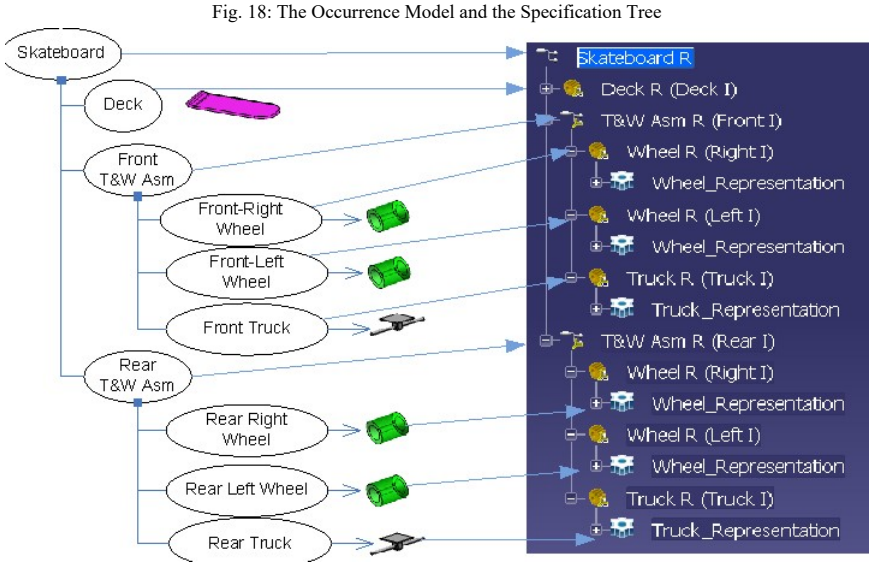
And so on until all the possible paths passing by all the instances of leaf references are run. An occurrence is created for each instance met along these paths. This is why an occurrence is also called an instance path, starting from the root and jumping from instance to instance. For example, the **front left** wheel occurrence is built by jumping from the root to the `Truck & Wheel Assembly` **front** instance and to the wheel **left** instance.

This occurrence model does not make use of the representation reference, representation instance, connection, and port objects. It does not include the representations and their parts. Nevertheless, to make things clear, the [Fig. 17] shows the seven physical objects using images, like the starting physical model.

The Specification Tree and the 3D Model

So if this occurrence model does not include neither the representations nor their parts, which objects make up the 3D model? Going back to the specification tree will unveil this mystery.

Fig. 18: The Occurrence Model and the Specification Tree



Putting the occurrence model and the specification tree side by side, you can see that the latter is well built using the occurrence model objects, but not only:

- The objects with the icons and are occurrences.
- The objects with the icons are representation instances that link with the shape and their geometries.

In the same way, the connections and ports are other instance/reference model objects seen in the specification tree, but not shown here.

To know more about this model, see the Product Occurrence Model [2].

In Short

The Product data model is persisted in database through 6 PLM Core objects: PLM Product Reference, PLM Product Instance, PLM Product Representation Reference, PLM Product Representation Instance, PLM Port, PLM Connection. This instance/reference model represents the exact physical model, but in a sparing way. So, in order to handle it, like in the real word, you need to "expend" it. It is the role of the occurrence data model. This model, not directly persisted in database, is the unfolded view of the instance/reference data model. The occurrence data model represents the different paths to get each leaf of the instance/reference data model. Be careful, this model does not contain Representation, connection and port. What you see interactively in the specification tree is a mix of the two models.

References

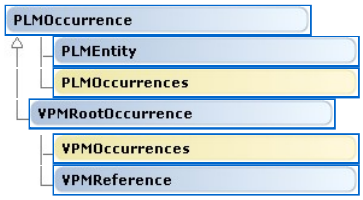
[1]

History

Version: **1** [Sep 2010] Document created

VPMRootOccurrence Object

See Also Legend Use Cases Properties Methods



Represents the root object of the product structure occurrence model object.

The **VPMRootOccurrence** object enables you to manipulate the product structure occurrence model. See Product Modeler Overview.

Retrieving a VPMRootOccurrence Object

Use the **RootOccurrence** property of the **PLMProductService** object, itself retrieved from the **Editor** object (see Service Object), to return the root occurrence object of a product model already opened in session.

```
Dim oPLMProductService As PLMProductService
Set oPLMProductService = oEditor.GetService("PLMProductService")

Dim oRootOcc As VPMRootOccurrence
Set oRootOcc = oPLMProductService.RootOccurrence
```

Using a VPMRootOccurrence Object

Use the **Occurrences** property of the **VPMRootOccurrence** object to return the **VPMOccurrences** collection object containing the **VPMOccurrence** objects children of the root occurrence object.

```
Dim cOccs As VPMOccurrences
Set cOccs = oRootOcc.Occurrences

For i = 1 To cOccs.Count
  Dim oVPMOcc As VPMOccurrence
  Set oVPMOcc = cOccs.Item(i)
  MsgBox "This occurrence object is an occurrence of the reference object: " _
         & oVPMOcc.InstanceOccurrenceOf.ReferenceInstanceOf.GetAttributeValue("PLM_ExternalID")
Next
```

This example shows you how to navigate from the occurrence to the instance, a **VPMInstance** object, it is related to thanks to the **InstanceOccurrenceOf** property of the **VPMOccurrence** object, and then to the reference, a **VPMReference** object, this object is an instance of thanks to the **ReferenceInstanceOf** property of the **VPMInstance** object. The Name attribute of this reference is displayed using the **GetAttributeValue** func of the **VPMReference** object inherited from its parent **PLMEntity** object.

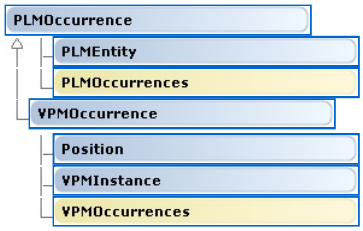
Use the **ReferenceRootOccurrenceOf** property of the **VPMRootOccurrence** object to return the **VPMReference** object associated with the root occurrence object.

```
Dim oRootRef As VPMReference
Set oRootRef = oRootOcc.ReferenceRootOccurrenceOf

MsgBox "The root occurrence object is associated with the reference object: " _
       & oRootRef.GetAttributeValue("PLM_ExternalID")
```

This example shows you how to retrieve the root product reference and display its Name attribute.

VPMOccurrence Object

See Also Legend Use Cases Properties Methods



Represents any object of the product structure occurrence model objects, except its root object.

The **VPMOccurrence** object enables you to manipulate the product structure occurrence model. See Product Modeler Overview.

Retrieving a VPMOccurrence Object

To retrieve a given **VPMOccurrence** object, start from the root occurrence and navigate the occurrence tree using the **VPMOccurrences** collection object of each **VPMOccurrence** object node until the one searched for is found.

```
Dim oPLMProductService As PLMProductService
Set oPLMProductService = oEditor.GetService("PLMProductService")

Dim oRootOcc As VPMRootOccurrence
Set oRootOcc = oPLMProductService.RootOccurrence

Dim oFoundOcc As VPMOccurrence

Dim isOccFound = False
Set isOccFound = navigateOccurrences oRootOcc, oFoundOcc

If isOccFound = True Then
    MsgBox "The occurrence object with Name set to SearchedOccName is found: " _
           & oFoundOcc.InstanceOccurrenceOf.GetAttributeValue("PLM_ExternalID")
Else
    MsgBox "The occurrence object with Name set to SearchedOccName is not found"
End If

...

Function navigateOccurrences (oOcc As VPMOccurrence, oFoundOcc As VPMOccurrence) As Boolean
  On Error GoTo ErrorSub
  Dim cOccs As VPMOccurrences
  Set cOccs = oOcc.Occurrences

  For i = 1 To cOccs.Count
    Dim oChildOcc As VPMOccurrence
    Set oChildOcc = cOccs.Item(i)

    If oChildOcc.InstanceOccurrenceOf.GetAttributeValue("PLM_ExternalID") = "SearchedOccName" Then
      oFoundOcc = oChildOcc
      Return True
    End If
    If navigateOccurrences (oChildOcc, oFoundOcc ) Then Return True
  Next

  GoTo EndSub
  ErrorSub:
    MsgBox Err.Description
  EndSub:
End Function
```

The navigateOccurrences func retrieves the **VPMOccurrences** collection object of the current VPMOccurrence object thanks to its **Occurrences** property, and loops onto the collection to scan each element until the appropriate one is found, here if the appropriate name of the associated **VPMInstance** object is found, or recursively scans the node below.

Using a VPMOccurrence Object

Use the **PLMEntity** property inherited from the **PLMOccurrence** object to retrieve to return the instance/reference model object associated with the current **VPMOccurrence** object.

```
Dim oVPMOcc As VPMOccurrence
... 'Retrieve either the VPMOccurrence object
Dim oPLMEntity As PLMEntity
Set oPLMEntity = oVPMOcc.PLMEntity
```

The **PLMOccurrences** property returns the **VPMOccurrence** object aggregated to the **PLMOccurrence** object. This collection object contains all the **PLMOccurrence** objects that are just below children of the **PLMOccurrence** object. lf this collection object is empty, the **PLMOccurrence** object is a leaf object in the occurrence model tree. Otherwise, it is a node. Prefer using the **Occurrences** properties of either the VPMRootOccurrence Object and the VPMOccurrence Object to navigate the occurrence model.

Use the **Position** property of the **VPMOccurrence** object to return the **Position** object storing the occurrence object location in the 3D space.

```
Dim oPosition As Position
Set oPosition = oVPMOcc.Position

Dim oPosArray (11)
oPosition.GetComponents oPosArray

MsgBox "Relative position of the occurrence of " & _
        oVPMOcc.VPMInstance.GetAttributeValue("PLM_ExternalID") & " in the 3D space" & _

vbCrLf & " Local coordinate system X vector: " & _
oPosArray (0) & ", " & oPosArray (1) & ", " & oPosArray (2) & _

vbCrLf & " Local coordinate system X vector: " & _
oPosArray (3) & ", " & oPosArray (4) & ", " & oPosArray (5) & _
```

```
vbCrLf & " Local coordinate system Z vector: " & _
oPosArray (6) & ", " & oPosArray (7) & ", " & oPosArray (8) & _

vbCrLf & " Local coordinate system origin coordinates: " & _
oPosArray (9) & ", " & oPosArray (10) & ", " & oPosArray (11)
```

The **Position** property of the **VPMOccurrence** object returns a **Position** object from which the coordinate system local to the occurrence can be retrieved. This coordinate system can expressed either with respect to the parent object coordinate system using the **GetComponents** sub as shown here, or to the absolute coordinate system thanks to the **GetAsbComponents** sub.

Navigating Product Model

Technical Article

Abstract

This article gives brief introduction to navigating Product model. Article introduces object structure and APIs to navigate through product model, this navigation includes Occurrence model as well Reference-Instance Model navigation. In addition this article gives way to switching objects occurrence model to Ref-Inst model.

Introduction

In order to navigate or edit a data model there are some API's exposed by the Automation Interfaces. This article explains how to navigate on a Reference-Instance Model and on an Occurrence structure. In addition it explains how to jump from the Occurrence Model to the Reference-Instance model. For understanding of this article user need to read article "Product Modeler Overview" [1] which explains Occurrence and Reference-Instance model in detail.

Navigating a VPM Occurrence Structure

This section explains how to navigate on a *VPMOccurrence* structure, starting from the Root Occurrence to a terminal Occurrence.

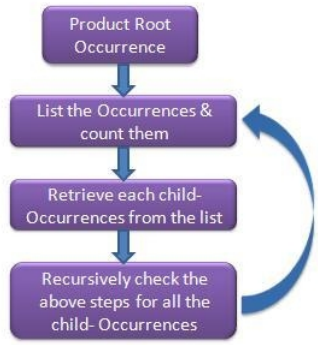
From an occurrence it is possible to retrieve the Reference-Instance Model corresponding object (*VPMInstance*/*VPMReference*) and continue the navigation through the product model.

Navigating Occurrence Model basically means navigating through or retrieving every occurrence which is part of the hierarchy or the tree structure of the Occurrence model.

This navigation includes logical steps
i) To find or retrieve Product Root Occurrence from VPM editor (logically speaking: to work on some object first you need to retrieve it).

ii) Once we find the Root-Occurrence, we list all the child-occurrences under the Root Occurrence using the property Occurrences exposed by the *VPMRootOccurrence* Interface.

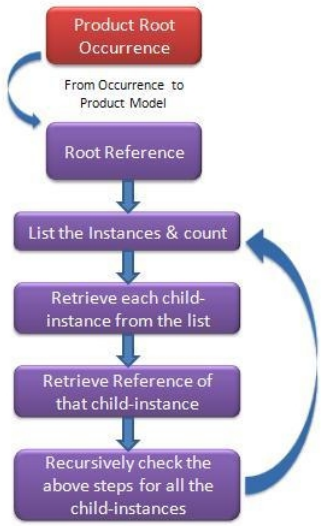
iii) Now all the child-occurrences found may still have further children. In order to list these children, we run a loop where all the above found list of child-occurrences are each navigated recursively. The list of children found is stored in the object of *VPMOccurrence* [2].



Navigating the Reference-Instance model or VPM Product Structure

This section exposes how to navigate on a Reference-Instance model, starting from the root product to a terminal representation by following the Instance–Reference model. From a Representation it is possible to jump to the applicative automation model included inside the stream associated to this representation and then continue the navigation through this data. To implement this operation, each entities of the product structure modeler (Reference, instance , Representation reference, Representation Instance, Port) expose an Automation Interface.

Navigating through the Reference-Instance basically means navigation thorough or retrieving the Reference-Instance modeler entities (Reference, Instance , Representation Reference, Representation Instance).



In order to navigate Reference-Instance Model:

i) We need to retrieve the Root-Reference of the product. But there's no API/Index available as such which will directly retrieve the Root-Reference of the Reference-Instance structure. This is because in case of UserInterface we have to retrieve the Root from the editor, and from editor we can retrieve only Occurrence Root and not Root-Reference.

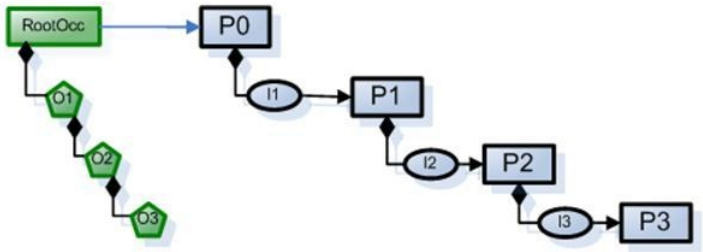
ii) Hence we first retrieve the Root Product Occurrence (as in case of while navigating the occurrence model). Then the Root-Reference is retrieved from the Root Product Occurrence using the property `ReferenceRootOccurrenceOf` exposed by the *VPMReference* interface.

iii) Once we get the Root-Reference, we list all the Children-Instances under the Product-reference using the property `Instances` which is exposed by the *VPMReference* Interface. The instances may be Reference-Instances or Representation-Instances. The instances are then counted.

iv) Reference-Instances point to References and Representation-Instances point to Representation-References.

v) Now these References (found at second level) may again have Reference-Instances or Representation-Instances and this chain goes on. So in order to list down all these entities we run a loop which goes on listing all the instances at every level and we can retrieve their corresponding References [3].

The following picture presents the Occurrence model (green) and the corresponding Reference-Instance model (blue).



Using automation APIs, it is possible to navigate:

- From RootOccurrence (RootOcc) to get the Product Root Reference (P0), then from P0. continue navigation through Reference-Instance model.
- From Occurrence O1 to get the Product Instance I1, then from I1 to continue navigation through Reference-Instance model.

In Short

It is possible to navigate through the Reference-Instance Model as well as Occurrence Model. We can directly navigate through the Occurrence Model but to navigate through the Reference-Instance Model we have to jump from the Occurrence to the Reference-Instance side.

References

[1] Product Modeler Overview
[2] Browsing Occurrence Model
[3] Browsing Product Contents

History

Version: **1** [Dec 2011] Document created

Opening Product Reference

This Use case retrieves a Root Product Reference from database according to end user criteria. In the process Use Case demonstrates about searching the Product Reference from the database and opening it into VPM Editor. Further Use Case demonstrates about retrieving handle to Root object though various methods.

Related Topics
Launching an Automation Use Case

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdPstUcOpenProductReferenceSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

This use case can be divided in 6 steps

1. Searches for a Product in database
2. Opens the Product
3. Retrieves Handle of Root object as an occurrence from editor
4. Retrieves Handle of Root object as a reference from editor
5. Retrieves Handle of Root object as a reference from root occurrence
6. Retrieves Handle of UI Active Object

1. Searches for a Product in database

   As a first step, the UC retrieves a Product from database

   It begins with a call to **SearchProduct** function. This function searches for a list of PLM Components from the underlying database based on an input search criteria. This list is output in the PLM Search Result window in CATIA.

```
...
    Dim oDBSearch  As  DatabaseSearch
    Set oDBSearch = SearchProduct(oSearchService)
...
```

   The function *SearchProduct* returns the `oDBSearch` object, a `DatabaseSearch` .

   We build up the search criteria, with a Product Reference as PLM Entity type , along with PLM Attributes PLM_ExternalID and revision as an input.

   The function **`SearchProduct(oSearchService)`** details are as in the below sub steps.

   i. Retrieves the DatabaseSearch property from the Search Service

```
Function SearchProduct(oSearchService) As DatabaseSearch
    Dim oDBSearch  As DatabaseSearch
    Set oDBSearch = oSearchService.DatabaseSearch
...
```

   ii. Sets the type of objects to search for

```
...
    oDBSearch.BaseType  = strTheProductReferenceType
...
```

   A call to the `BaseType` property sets the type of objects to search for. `strTheProductReferenceType` is a global variable defining a Product Reference type.

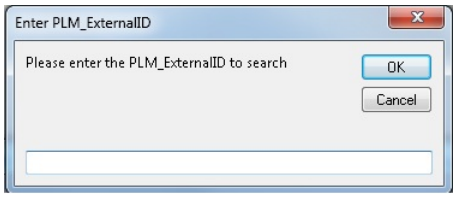
   iii. Updates the PLM Search object created in the above steps with the attribute criteria provided by the user as an input

```
...
    oDBSearch.AddEasyCriteria  "PLM_ExternalID", strInputPLMIDName
    oDBSearch.AddEasyCriteria  "revision", strInputRevision
...
```

   A call to *AddEasyCriteria* method, updates the created DatabaseSearch object with the search criteria according to the users input as depicted in the figures below

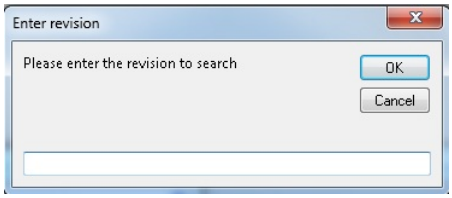
   Prompt the user to input the PLM_ExternalID for search purpose. A dialog box appears:

   Fig. 1 Dialog to input PLM_ExternalID for search



   Next, we prompt the user to input the revision for search purpose. A dialog box appears:

   Fig. 2 Dialog to input revision for search



   iv. Triggers the search

```
...
    oSearchService.search
```

```
...
End Function
```

A call to *search* method of the SearchService object actually searches for the objects which matches all the attributes of the set and matching the case of the values(i.e. search is Case Sensitive), and type.

2. Opens the Product and retrieves its Editor

As a next step, the UC essentially loads in session first output by search collection occurring in the new tab page retrieved in the last step.

```
...
    Dim oProductEditor As Editor
    Set oProductEditor = OpenProductAndGetEditor(oDBSearch)
...
```

The function *OpenProductAndGetEditor* takes a `DatabaseSearch`, `oDBSearch`, as its input and returns an Editor which contains opened Product. This `oDBSearch` input argument object to this method was updated output by the Search on the underlying database, which occurred in the previous step

The function `OpenProductAndGetEditor` is detailed as in the below sub steps.

i. Retrieves the root entities from the new search tab page

```
Function OpenProductAndGetEditor(oDBSearch) As Editor
    Dim oPLMProdRefAsEntities As PLMEntities
    Set oPLMProdRefAsEntities = oDBSearch.Results
...
```

A call to the *Results* method of DatabaseSearch object, `oDBSearch` returns a collection of root PLM Entities in the Search tab.

ii. Retrieves a PLM Entity object from its index

```
...
    Dim oPLMProdRefAsPLMEntity As PLMEntity
    Set oPLMProdRefAsPLMEntity = oPLMProdRefAsEntities.Item(1)
...
```

The first entity in the above collection is retrieved, thanks to the *Item* method on PLMEntities.

iii. Retrieves the Open Service from CATIA Session

```
...
    Dim oOpenService As PLMOpenService
    Set oOpenService = CATIA.GetSessionService("PLMOpenService")
...
```

A call to the Application (CATIA) *GetSessionService* method returns the specified Service, PLMOpenService, `oOpenService` in this case.

iv. Opens in the authoring session the first element occurring in the new search tab page (a Product)

```
...
    Dim oProductEditor As Editor
    oOpenService.PLMOpen oPLMProdRefAsPLMEntity , oProductEditor
...
End Function
```

The PLM Entity, `oPLMProdRefAsPLMEntity` retrieved in the above steps is then loaded in the current session with a call to the *PLMOpen* method of the PLMOpenService. The editor associated with the loaded Product in the current session, is finally returned by the *PLMOpen* call in a Product Editor, `oProductEditor`. The PLMEntity, `oPLMProdRefAsPLMEntity` , is loaded in Authering mode

3. Retrieves Handle of Root object as an occurrence from editor

Retrieve the 'PLMProductService' service from the editor. The editor object being a VPMEditor, we can use a service to retrieve information specific to the VPM editor.

```
...
    Dim  oProductService  As   PLMProductService
    Set  oProductService = oProductEditor.GetService("PLMProductService")
...
```

The `PLMProductService`, `oProductService` is retrieved from the `Editor`, `oProductEditor`, thanks to the *GetService* method with the string "*PLMProductService*" as an input

Retrieve the root occurrence of the product

```
...
    Dim  oVPMRootOccOnRoot  As   VPMRootOccurrence
    Set  oVPMRootOccOnRoot = oProductService.RootOccurrence

    MsgBox "Success in retrieving Handle of Root object as an occurrence : " + oVPMRootOccOnRoot.Name
...
```

A call to *RootOccurrence* property of the `PLMProductService`, `oProductService`, returns the Product Root as an occurrence `VPMRootOccurrence`, `oVPMRootOccOnRoot`.

4. Retrieves Handle of Root object as a reference from editor

```
...
    Dim  oPLMEntities  As   PLMEntities
    Set  oPLMEntities = oProductService.EditedContent
...
```

A call to *EditedContent* property of `PLMAppContext` (the object where derived PLMProductService) called on `oProductService` returns the root Product as a VPMReference. The list of PLMEntiries returned by EditedContent only contains one element in the VPMEditor case.

Next retrieve first object from PLMEntities collection object. The number of elements in the list are always 1 for VPM Editor since root object is product

reference Root object. In this case Root object is of type VPMReference since the Product Root object is of Reference-Instance Model root.

```
...
    Dim  oPLMEntityAsVPMRefOnRoot  As  VPMReference
    Set  oPLMEntityAsVPMRefOnRoot = oPLMEntities.Item(1)
...
```

A call to *Item* method of PLMEntities returns the indexed (1) `PLMEntity` in this case we are retrieving first object as `VPMReference` object.

5. Retrieves Handle of Root object as a reference from root occurrence

   Retrieve the root object of instance -reference model `VPMReference` type from the occurrence model. It is the same Root object retrieved in previous step [#].

```
...
    Dim  oVPMRefOnRoot  As   VPMReference
    Set  oVPMRefOnRoot = oVPMRootOccOnRoot.ReferenceRootOccurrenceOf

    MsgBox "Success in retrieving Root object as a reference from root occurrence : " + oVPMRefOnRoot.Name

...
```

   A call to *ReferenceRootOccurrenceOf* property of `VPMRootOccurrence`, `oVPMRootOccOnRoot` returns the Product Reference `oVPMRefOnRoot` corresponding object.

6. Retrieves Handle of UI Active Object

   By default in VPM Editor of opened Product the UI active object is Root occurrence

```
...
    Dim  oUIActiveObject  As   VPMRootOccurrence
    Set  oUIActiveObject = oProductEditor.ActiveObject

    MsgBox "Success in retrieving UI Active Object : " + oUIActiveObject.Name
...
```

   A call to *ActiveObject* property of Editor on editor returns UI active object, in this case it returns Root occurrence object `oUIActiveObject`

Browsing Occurrence Model

This use case fundamentally illustrates navigation of the Product Occurrence Model. Use case navigates down the hierarchy of an Occurrence Model associated with a Product Root Reference loaded in session. It retrieves the Occurrences aggregated under the Root Occurrence, and displays the entire hierarchy in a tree format, as viewed in the Specification Tree within CATIA, but without the Representation Instances/References.

Related Topics
Launching an Automation Use Case

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdPstUcBrowseOccurrenceModelSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

This use case can be divided in 3 steps

1. Searches for a Product in database
2. Opens the Product and retrieves its Editor
3. Navigates the Product Occurrence Recursively

1. **Searches for a Product in database**

   As a first step, the UC retrieves a Product from database

   It begins with a call to **SearchProduct** function. This function searches for a list of PLM Components from the underlying database based on an input search criteria. This list is output in the PLM Search Result window in CATIA.

```
 ...
    Dim oDBSearch  As  DatabaseSearch
    Set oDBSearch = SearchProduct(oSearchService)
...
```

   The function *SearchProduct* returns the `oDBSearch` object, a `DatabaseSearch`.

   We build up the search criteria, with a Product Reference type , and PLM_ExternalID and revision as an input [1].

2. **Opens the Product and retrieves its Editor**

   As a next step, the UC essentially loads in session first PLM Entity output by search collection occurring in the new tab page within Search Editor retrieved in the last step [1].

```
...
    Dim oProductEditor As Editor
    Set oProductEditor = OpenProductAndGetEditor(oDBSearch)
...
```

   The function *OpenProductAndGetEditor* takes a `DatabaseSearch`, `oDBSearch`, as its input and returns an Editor which contains opened Product. This argument was output by the Search on the underlying database, which occurred in the previous step

3. **Navigates the Product Occurrence Recursively**

   Now Use case retrieves the Product Occurrence (Root) associated with its Occurrence Model and then navigates down its product hierarchy to display its

contents, precisely in the same way, as seen in the specification tree within CATIA, except that the Representation Instances/ References are not displayed as shown in the Fig.3 below.

```
...
    NavigateProductOccurrence oProductEditor
...
```

The function *NavigateProductOccurrence* takes an `Editor`, `oProductEditor`, as an input.

Fig.3: Occurrence Product model sample output

The function *NavigateProductOccurrence* details are as in the below sub steps.

I. Retrieves the Root Product Occurrence from the current VPM Editor to navigate

```
Sub NavigateProductOccurrence(oProductEditor)

    Dim  oProductService  As   PLMProductService
    Set  oProductService= oProductEditor.GetService("PLMProductService")

    Dim  oVPMRootOccOnRoot  As   VPMRootOccurrence
    Set  oVPMRootOccOnRoot = oProductService.RootOccurrence
...
```

The `PLMProductService`, `oProductService` is retrieved from the `Editor`, `oProductEditor`, thanks to the *GetService* method with the string "*PLMProductService*" as input

Then a call to *RootOccurrence* Property of the `PLMProductService`, `oProductService`, returns a `VPMRootOccurrence`, `oVPMRootOccOnRoot` .

II. Navigates the Product Occurrence

Here we navigate through occurrence model recursively. We will start navigation from root occurrence which is retrieved in last step.

```
...
    NavigateProdOccurrences oVPMRootOccOnRoot, 0
...
```

The function *NavigateProdOccurrences* takes a `VPMRootOccurrence`, `oVPMRootOccOnRoot`, as an input. last argument 0 is depth which will further used for display purpose.

The function *NavigateProdOccurrences* details are as in the below sub steps

i. Retrieves the list of Occurrences within the input Occurrence

```
Sub NavigateProdOccurrences(oOccurrence, depth)

   Dim  oListChildrenOccurrences  As   VPMOccurrences
   Set  oListChildrenOccurrences = oOccurrence.Occurrences

   strBrowsedPLMCompIDAttr = strBrowsedPLMCompIDAttr + oOccurrence.Name + vbCrLf
...
```

A call to Occurrences Property of the `VPMOccurrence`, retrieves the `VPMOccurrences` a collection object, `oListChildrenOccurrences`

Then a call to Name Property returns the name of Occurrence further we append this name into string `strBrowsedPLMCompIDAttr` for display with tab value `vbCrLf`.

ii. Navigates through each child Occurrence recursively

```
...
 For i = 1 To oListChildrenOccurrences.Count

     Dim  oChildOcc As   VPMOccurrence
     Set  oChildOcc =  oListChildrenOccurrences.Item(i)

     NavigateProdOccurrences oChildOcc, depth + 1
 Next
...
```

We parse through the *VPMOccurrences* collect object, a `oListChildrenOccurrences` and retrieves PLM Product Occurrence `oChildOcc` from its index, thanks *Item* method of *VPMOccurrences*

A recursive call to *NavigateProdOccurrences* with the `VPMOccurrence` object, `a oChildOcc` as one of its arguments. The second argument (depth + 1) represents the offset at which the child occurrence will be displayed in the output, implying it is output at "+1 tab" offset relative to its parent occurrence.

References

[1] Opening Product Reference

Browsing Product Contents

This use case fundamentally illustrates navigation of the Product Structure Model. Use case navigates down the hierarchy of an Product Reference-Instance Model associated with a Product Root Reference loaded in session. And displays the entire hierarchy in a tree format, as viewed in the Specification Tree within CATIA, but with the Representation Instances/References.

Related Topics
Launching an Automation Use Case

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdPstUcBrowsingProductContentsSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

This use case can be divided in 3 steps

1. Searches for a Product in database
2. Opens the Product and retrieves its Editor
3. Navigates the Product Reference Recursively

1. **Searches for a Product in database**

   As a first step, the UC retrieves a Product from database

   It begins with a call to **SearchProduct** function. This function searches for a list of PLM Components from the underlying database based on an input search criteria[1]. This list is output in the PLM Search Result window in CATIA.

```
...
    Dim oDBSearch  As  DatabaseSearch
    Set oDBSearch = SearchProduct(oSearchService)
...
```

   The function *SearchProduct* returns the `oDBSearch` object, a `DatabaseSearch`.

   We build up the search criteria, with a Product Reference type , and PLM_ExternalID and revision as an input.

2. **Opens the Product and retrieves its Editor**

   As a next step, the UC essentially loads in session first PLM Entity output by search collection occurring in the new tab page within Search Editor retrieved in the last step[1].

```
...
    Dim oProductEditor As Editor
    Set oProductEditor = OpenProductAndGetEditor(oDBSearch)
...
```

   The function *OpenProductAndGetEditor* takes a `DatabaseSearch`, `oDBSearch`, as its input and returns an Editor which contains opened Product. This argument was output by the Search on the underlying database, which occurred in the previous step

3. **Navigates the Product Reference Recursively**

   Now Use case retrieves the Product Reference (Root) associated with its reference-Instance Model and then navigates down its product hierarchy to display its contents, precisely in the same way, as seen in the specification tree within CATIA, with the Representation Instances/ References as shown in the Fig.1 below.

```
...
    NavigateProductReference  oProductEditor
...
```

   The function *NavigateProductReference* takes an `Editor`, `oProductEditor`, as an input.

   Fig.1: Product model sample output

   The function *NavigateProductReference* details are as in the below sub steps.

   I. Retrieves the Root Product Occurrence from the current VPM Editor to navigate

```
Sub NavigateProductReference (oProductEditor)

    Dim  oProductService  As   PLMProductService
    Set  oProductService = oProductEditor.GetService("PLMProductService")

    Dim  oVPMRootOccOnRoot  As   VPMRootOccurrence
    Set  oVPMRootOccOnRoot = oProductService.RootOccurrence
...
```

   The `PLMProductService`, `oProductService` is retrieved from the `Editor`, `oProductEditor`, thanks to the *GetService* method with the string "*PLMProductService*" as input

   Then a call to *RootOccurrence* Property of the `PLMProductService`, `oProductService`, returns a `VPMRootOccurrence`, `oVPMRootOccOnRoot` .

II. Retrieves the Root Reference from the occurrence model

```
...
    Dim  oVPMRefOnRoot As   VPMReference
    Set  oVPMRefOnRoot = oVPMRootOccOnRoot.ReferenceRootOccurrenceOf
...
```

The VPMReference, oVPMRefOnRoot is retrieve corresponding Root Reference from the Root Occurrence VPMRootOccurrence, oVPMRootOccOnRoot

This step is important, user needs to switch from occurrence model to Reference-instance model.

III. Navigates the Product Reference

Here we navigate through model recursively. We will start navigation from Root Reference which is retrieved in last step.

```
...
    NavigateProdReference oVPMRefOnRoot, 1
...
```

The function *NavigateProdReference* takes a `VPMReference`, `oVPMRefOnRoot`, as an input. last argument 1 is depth which will further used for display purpose.

The function *NavigateProdReference* details are as in the below sub steps

i. Retrieves the Rep Instances under the Product Reference

```
Sub NavigateProdReference(oProdRef, depth)

    NavigateProdReferenceForRepInst oProdRef, depth
...
```

The function *NavigateProdReferenceForRepInst* takes a `VPMReference`, `oProdRef`, as an input. last argument 1 is depth which will further used for display purpose.

The function *NavigateProdReferenceForRepInst* details are as in the below sub steps

a. Retrieves the list of Rep Instances within the input reference

```
Sub NavigateProdReferenceForRepInst(oProdRef, depth)

   Dim  oVPMRepInsts  As   VPMRepInstances
   Set  oVPMRepInsts = oProdRef.RepInstances
...
```

A call to *RepInstances* Property of the `VPMReference`, retrieves the `VPMRepInstances` a collection object, `oVPMRepInsts`

b. Navigates through each child Rep instances

```
...
    For k = 1 To oVPMRepInsts.Count

       Dim oVPMRepInst As VPMRepInstance
       Set oVPMRepInst = oVPMRepInsts.Item(k)

       Dim oVPMRepRef As VPMRepReference
       Set oVPMRepRef = oVPMRepInst.ReferenceInstanceOf

    Next k
...
```

Next UC parse through the *VPMRepInstances* collection object, a `oVPMRepInsts` and retrieves Product Representation Reference Instance `oVPMRepInst` from its index, thanks *Item* method of *VPMRepInstances*

Further we retrieve VPM Rep reference of rep instance, a *VPMRepReference* `oVPMRepRef` using *ReferenceInstanceOf* property of *VPMRepReference,* `oVPMRepRef`

ii. Retrieves the list of Instances within the input *Reference*

```
...
   Dim  oListChildrenInstances As   VPMInstances
   Set  oListChildrenInstances = oProdRef.Instances
...
```

A call to Instances Property of the `VPMReference`, retrieves the `VPMInstances` a collection object, `oListChildrenInstances`

iii. Navigates through each child Instances recursively

```
...
    For i = 1 To oListChildrenInstances.Count

        Dim  oVPMInst As   VPMInstance
        Set  oVPMInst = oListChildrenInstances.Item(i)

        Dim  oVPMRef As   VPMReference
        Set  oVPMRef = oVPMInst.ReferenceInstanceOf
        ...
        NavigateProdReference oVPMRef, depth + 1

    Next i
...
```

We parse through the *VPMInstances* collect object, a `oListChildrenInstances` and retrieves PLM Product Instance `oVPMInst` from its index, thanks *Item* method of *VPMInstances*

From each instance a call to *ReferenceInstanceOf* of VPMInstance returns a Product reference `VPMReference, oVPMRef`

A recursive call to *NavigateProdReference* with the `VPMReference` object, a `oVPMRef` as one of its arguments. The second argument (depth + 1) represents the offset at which the child Reference will be displayed in the output, implying it is output at "+1 tab" offset relative to its parent Reference.

References

[1] Opening Product Reference

Product Modeler Attributes

Technical Article

Abstract

This article shows you what are the attributes you can handle, and how you can dot it.

Product Modeler Attributes

For the four following entities of the Product Modeler (*VPMReference, VPMInstance, VPMRepReference, VPMRepInstance*) [1] you can handle their attributes by VB. To distinguish them from other attributes, we also called them, **PLM Attributes**.

The accessible PLM Attributes are only the public attributes, and only the read/write attributes are modifiable.

The API to access them (see the next chapter for more details), does not work with UI mask: you have a full access to the public attributes, but still according to read/write right defined by the metadata.

You can access the modeler's attributes (DS defined), like yours.

Internal versus External Name

In the Edit Properties dialog box, the PLM Attributes are displayed with their NLS names, while in the API the internal name is required. The following documents, one for each Product Entity type, list their PLM attributes. Inside, you will retrieve a conversion from the NLS name to the internal name. But be carefull, this conversion has some limitations:

- NLS version is only english
- NLS version can be customized on your site - In other words, the NLS name of an attribute can be different between the name given by DS and your customization.

To reach the appropriate document for the below mentioned Types, you will have to go through the Product Documentation i.e. End-User Documentation
The path to reach the desired article is as follows

End-User Doc --> Enovia Development and Configuration --> Unified Live collaboration --> V6 Data reference Dictionary --> Modelers --> PRODUCT TYPES

- VPMReference Attributes
- VPMInstance Attributes
- VPMRepReference Attributes
- VPMRepInstance Attributes

When you handle a simple occurrence, VPMOccurence, retrieve its instance (VPMInstance), and when you handle the root occurence, a VPMRootOccurence, retrieve its reference. The referenced use case [2] explains to you how to do.

Accessing PLM Attributes

The API to handle a PLM attributes, is defined by the *PLMEntity* object through the `GetAttributeValue` and `SetAttributeValue` methods [2].

An example for Getting:

```
...
Dim  strValue As String
Dim oVPMEntityOnMyObject as PLMEntity
...
strValue = oVPMEntityOnMyObject.GetAttributeValue("V_description")
...
```

An example for Setting:

```
...
Dim strValuePID As String
Dim oVPMEntityOnMyObject as PLMEntity
...
oVPMEntityOnMyObject.SetAttributeValue "V_description", "My new description"
...
```

If the type of the attribute is a list, values must be provided as a string with semi colon separators, like in the following:

```
...
oVPMEntityOnMyObject.SetAttributeValue "V_myListString", "value1;value2;"
oVPMEntityOnMyObject.SetAttributeValue "V_myListInteger", "1;2;"
...
```

In Short

References

[1] Product Modeler Presentation
[2] Modifying PLM Attributes of a Selected Product Object

History

Version: **1** [May 2010] Document created

Modifying PLM Attributes of a Selected Product Object

This use case selects and retrieves the PLM object from the CATIA spec tree and retrieves and modifies its PLM Attribute (V_description).
In the process it also illustrates that a PLM Attribute is associated with an Instance-Reference model.

Related Topics
Editor Object
Selection Object
Product Object
Model Map

Before you begin: Note that:

- Launch CATIA
- Load a Product from database or create one before launching the macro
- Keep Product editor active before launching the macro

Where to find the macro: CAAScdPstUcDisplayAndModifyProductAttrFromSelectionSource.htm

This use case can be divided in 4 steps

1. Retrieve active editor
2. Retrieve Selection object from active editor
3. Display and Modify PLM Attributer Value

1. Retrieves Active Editor from CATIA

   We retrieve the Selection Object from CATIA, to enable us to select the vertices in the steps ahead.

```
...
    Dim oCurrentActiveEditor As Editor
    Set oCurrentActiveEditor = CATIA.ActiveEditor
...
```

   The *ActiveEditor* property of the Application Object, CATIA returns the Editor, `oCurrentActiveEditor` which is currently active.

   Retrieves Selection object from Active Editor

```
...
    Dim oObjSelection
    Set oObjSelection = oCurrentActiveEditor.Selection
...
```

   A call to *Selection* Property of the Active Editor returns `oObjSelection`, a `Selection` object. It is significant to note here that we have not declared a type for `oObjSelection` . The section "Virtual Function Table Compatibility" in the Technical Article "About Automation Languages, Debug, and Compatibility" [1] provides an explanation for why these variables are not typed.

2. Retrieve Selection object from active editor

   1. Update selection object with the search criteria and prompt for selection

```
...
Dim oInputObjectType(2)
oInputObjectType(2) = "VPMRootOccurrence"
oInputObjectType(1) = "VPMOccurrence"
oInputObjectType(0) = "VPMRepInstance"

Dim strStatus As String
strStatus = oObjSelection.SelectElement(oInputObjectType, "Select a Element from spec tree", False)
...
```

      The PLM objects that you can select (if the current editor is VPM Editor) are **VPMRootOccurrence**, **VPMOccurrence** and **VPMRepInstance** types. These types are appended to an array `oInputObjectType`, which now represents the select criteria. The selection object `oObjSelection` is then updated with this select criteria, with a call to *SelectElement* method. Since the last argument is false, the end user will always be invited to select something in the spec tree, or the 3D Viewer, though an entity (of the type set as filter in the selection criteria) has already been selected by the end-user.

   2. Retrieve Selected element and display its type

```
...
Dim oSelectedElement As SelectedElement
Set oSelectedElement  =  oObjSelection.Item(1)

MsgBox "Type is " + oSelectedElement.Type
...
```

      The *Item* method on `oObjSelection`, returns the iIndex-th `SelectedElement` Object contained by the current selection.

      Next, *Type* property on the `oSelectedElement`, returns selected object type

      Fig.1 Type of The Selected Product Object

This will show the type of object which has been selected by user interactively from spec tree. This type could be a VPMOccurrence or VPMRootOccurrence (type defined by an Occurrence Model) or the type could be a VPMRepInstance (the type defined by an Instance-Reference model). The type will be seen as shown in [Fig.1].

3. Retrieve the PLM Entities underlying the user selection in CATIA spec tree , or the 3D Viewer

   It is significant to note that the PLM Attributes are actually associated NOT with the entities of an Occurrence model but with those of an Instance/Reference model. Since the user in this case has selected an entity from the spec tree in CATIA, he has the liberty to either select an Occurrence or a Rep Instance.Hence if the selected entity is an Occurrence, we seek the underlying PLM entity (Instance / Reference). While if the selected entity is a Rep Instance, we can very well go ahead directly with setting its PLM Attributes.

   We begin with the case when a Product Rep Inst is selected

```
...
Dim oVPMEntityOnSelectedObject As PLMEntity
If (oSelectedElement.Type = "VPMRepInstance") Then

    Set oVPMEntityOnSelectedObject = oSelectedElement.Value
...
```

   The *Type* property on the `oSelectedElement`, returns the Automation type (in this case VPMRepInstance) of the Product Component underlying the selected entity.

   Then a *Value* property on the `oSelectedElement`, returns the `PLMEntity` type object `oVPMEntityOnSelectedObject` underlying the entity selected in the spec tree, or the 3D Viewer. A Rep Instance in this case, as a feature is selected, since for a feature in the spec tree the nearest aggregating PLM Entity which conforms to the selection criteria is a Rep Instance.

   On the other hand, if the automation type of the underlying PLM entity is VPMRootOccurrence or VPMOccurrence, it is treated as the code below depicts.

```
...
ElseIf (oSelectedElement.Type = "VPMRootOccurrence" Or oSelectedElement.Type = "VPMOccurrence") Then

    Dim oVPMOccOnSelectedObject As PLMOccurrence
    Set oVPMOccOnSelectedObject = oSelectedElement.Value

    Set oVPMEntityOnSelectedObject = oVPMOccOnSelectedObject.PLMEntity

End If
...
```

   The *Type* property on the `oSelectedElement`, returns the Automation type in this case returns a `VPMRootOccurrence` or `VPMOccurrence` based on the selection.

   Then a *Value* property on the `oSelectedElement`, returns the PLM Occurrence type, oVPMOccOnSelectedObject.

   Next a *PLMEntity* property of `oVPMOccOnSelectedObject` returns `oVPMEntityOnSelectedObject` (a `PLMEntity` type on the underlying First Instance or Reference)

3. Display and Modify PLM Attributer Value

   i. Retrieve and display the value of PLM_ExternalID of selected Object

   Now that we have retrieved the PLM Entity from the user selection in above step [#], we can retrieve and display the value of its PLM Attributes (in this case, PLM_ExternalID)

```
...
Dim strValuePID As String
strValuePID = oVPMEntityOnSelectedObject.GetAttributeValue("PLM_ExternalID")

MsgBox (strValuePID)
...
```

   Return a **PLM_ExternalID** attribute value from the `PLMEntity` object using the *GetAttributeValue* method.

   Fig.2
   PLM_ExternalID Attribute Value of Selected Object

   The above [Fig.3] shows the PLM_ExternalID attribute value.

   ii. Modify the V_description Attribute value of PLM entity selected

   Retrieve and display the value of V_description before re-valuation

```
...
Dim strAttrValueV_description As String
strAttrValueV_description = oVPMEntityOnSelectedObject.GetAttributeValue("V_description")

MsgBox (strAttrValueV_description)
...
```

   Return a **V_description** attribute value from the `PLMEntity` object using the *GetAttributeValue* method.

   Fig.3

V_description
Attribute Value
Before Change

The above [Fig.3] shows V_description attribute value before revaluation. (before re-valuation value = blank)

Modify the V_description Attribute value

```
...
oVPMEntityOnSelectedObject.SetAttributeValue "V_description", "NewDescription"
...
```

Valuate a **V_description** attribute value with new value of `PLMEntity` object using the *SetAttributeValue* method.

Retrieve and display the value of V_description attribute after re-valuation

```
...
strAttrValueV_description= oVPMEntityOnSelectedObject.GetAttributeValue("V_description")

MsgBox (strAttrValueV_description)
...
```

Return a **V_description** attribute value from the `PLMEntity` object using the *GetAttributeValue* method.

Fig.4
V_description
Attribute Value
After Change

The above [Fig.4] shows V_description attribute value after revaluation.(after re-valuation value = "NewDescription")

The PLM Attributes associated with an Instance-Reference model are indeed persistent and hence are SAVED in the underlying database.

References

[1] About Microsoft Automation Languages, Debug, and Compatibility

Modifying Visualization Attributes of a Selected Product Object

This use case selects and retrieves the PLM object from the CATIA spec tree and modifies its Visualization Property (Color). In the process it also illustrates that a Visualization Properties are associated with an Occurrence model.

Related Topics
Editor Object
Selection Object
Foundation Object
Model Map

Before you begin: Note that:

- Launch CATIA
- Load a Product from database or create one before launching the macro
- Keep Product editor active before launching the macro

Where to find the macro: CAAScdPstUcModifyVisuProductAttrFromSelectionSource.htm

This use case can be divided in 4 steps

1. Retrieve active editor
2. Retrieve Selection object from active editor
3. Change the visualization property Color of the selection

1. Retrieves Active Editor from CATIA

   We retrieve the Selection Object from CATIA, to enable us to select the vertices in the steps ahead.

   ```
   ...
       Dim oCurrentActiveEditor As Editor
       Set oCurrentActiveEditor = CATIA.ActiveEditor
   ...
   ```

   The *ActiveEditor* property of the Application Object, CATIA returns the Editor, `oCurrentActiveEditor` which is currently active.

   Retrieves Selection object from Active Editor

   ```
   ...
       Dim oObjSelection
       Set oObjSelection = oCurrentActiveEditor.Selection
   ...
   ```

   A call to *Selection* Property of the Active Editor returns `oObjSelection`, a `Selection` object. It is significant to note here that we have not declared a type for `oObjSelection` . The section "Virtual Function Table Compatibility" in the Technical Article "About Automation Languages, Debug, and Compatibility" [1] provides an explanation for why these variables are not typed.

2. Retrieve Selection object from active editor

    1. Update selection object with the search criteria and prompt for selection

```
...
Dim oInputObjectType(1)
oInputObjectType(1) = "VPMRootOccurrence"
oInputObjectType(0) = "VPMOccurrence"

Dim strStatus As String
strStatus = oObjSelection.SelectElement(oInputObjectType, "Select a Element from spec tree", False)
...
```

    The PLM objects that you can select (if the current editor is VPM Editor) are **VPMRootOccurrence and VPMOccurrence** types. These types are appended to an array `oInputObjectType`, which now represents the select criteria. The selection object `oObjSelection` is then updated with this select criteria, with a call to *SelectElement* method. Since the last argument is false, the end user will always be invited to select something in the spec tree, or the 3D Viewer, though an entity (of the type set as filter in the selection criteria) has already been selected by the end-user.

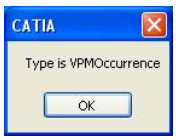
    2. Retrieve Selected element and display its type

```
...
Dim oSelectedElement As SelectedElement
Set oSelectedElement  =  oObjSelection.Item(1)

MsgBox "Type is " + oSelectedElement.Type
...
```

    The *Item* method on `oObjSelection`, returns the iIndex-th `SelectedElement` Object contained by the current selection.

    Next, *Type* property on the `oSelectedElement`, returns selected object type

    Fig.1 Type of The Selected Product Object



    This will show the type of object which has been selected by user interactively from spec tree. This type could be a VPMOccurrence or VPMRootOccurrence (type defined by an Occurrence Model). The type will be seen as shown in [Fig.1].

3. Change the visualization property Color of the selection

    At this stage, the user has finalized an entity selection from the CATIA spec tree. We proceed to change its visualization property "Color" (set to Green here) . Please note that Visualization property is associated with occurrence model only. So change in VPMRepInstance object will not be seen.

```
...
Dim oVisPropertySet
Set oVisPropertySet = oObjSelection.VisProperties

oVisPropertySet.SetRealColor 0, 255, 0, 1
...
```

    Return a `VisPropertySet` object from Selection object using *VisProperties* property. Then set a color using *SetRealColor* method of `VisPropertySet`.

    It is extremely important to note that this Save is not applicable for any changes in the Visualization Properties associated with an Occurrence model.

References

[1] About Microsoft Automation Languages, Debug, and Compatibility

Retrieving Product Reference from Mono-instantiable Representation Reference

The Use Case fundamentally retrieves the parent of a MonoInstantiable Representation Reference. A MonoInstantiable Representation Reference can be instantiated only once across all Product contexts. As a result, the Product Reference which aggregates its only Instance, is considered the parent of the Mono Representation Reference too.

In contrast, the Use Case illustrates the failure to retrieve the parent of a Multi Instantiable Representation Reference, since it has several Representation Instances each aggregated by its own parent. So we cannot associate a unique parent with a Multi Instantiable Representation Reference.

Related Topics
Product Object Model Map

Before you begin: Note that:

- Launch CATIA
- Load a Product (which contains at least one mono-instantiable Representation Instance beneath it) from database or create one before launching the macro
    - Input Model scenario
        - Create Product Reference (Root)
        - Insert 3Dshape object beneath Root (By default it will be Mono-Instanciable). Similarly create 2 instances
        - • From contextual menu of 3DShape instance click on " Share this representation" command, this will make 3DShape Multi-Instantiable Representation Reference from Mono-Instantiable Representation Reference. Switch one of the Mono-Instantiable Representation Reference to Multi-Instantiable Representation Reference so that we will have both cases.
- Keep Product editor active before launching the macro

Where to find the macro: CAAScdPstUcRetrieveProductReferenceFromMonoInstantiableRepRefSource.htm

This use case can be divided in three steps:

1. Retrieve Selected Representation Instance As VPMRepInstance from from spec tree or 3D Viewer
2. Retrieve Product Representation Reference from selected Representation Instance
3. Retrieve Product Reference aggregating the Product Representation Reference

1. Select and Retrieve Product Representation Instance As VPMRepInstance from spec tree or 3D Viewer

   The PLM object that you can select in this Use Case (if the current editor is VPM Editor) is **VPMRepInstance** type (Since this is selection criteria). The mc "Modifying Attributes (PLM and Visualization) of a Selected Product Object" Use Case[1].

   Use Case initially prompts an end user to select the mono-instantiable Representation Instance from spec tree or the 3D Viewer within an active editor of C

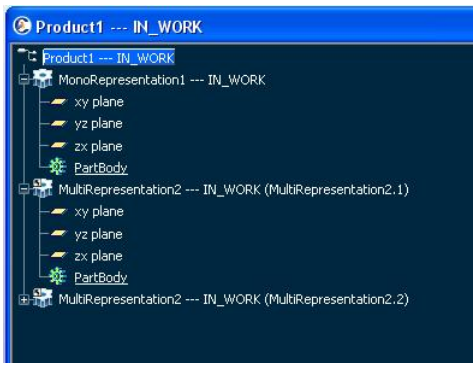
```
...
Dim oVPMRepInstance As VPMRepInstance
Set oVPMRepInstance = SelectAndRetrieveRepInst
...
```

   The function *SelectAndRetrieveRepInst*, returns `oVPMRepInstance`, a `VPMRepInstance` type

   It is important to note here that inspite of that restriction to select only a *VPMRepInstance* type from the spec tree or the 3D Viewer, user still has the liberty beneath a Representation Instance. In this case, the focus shifts to the aggregating Representation Instance, which is infact considered to be the selected enti

   Following is the sample example to demonstrate this use case.

   Fig.1 Sample Model for Selection



   For a model as depicted in [Fig. 1] we have a Product1 as Root Reference. It has an instance of a Mono-Instantiable Representation Reference (MonoRepres and two instances of a MultiInstantiable Representation Reference (MultiRepresentation2.1--- and MultiRepresentation2.2---, represented by icon in the for the code extract in the steps ahead are based on the execution of this macro on this input model.

2. Retrieve Product Representation Reference from selected Representation Instance

```
...
Dim oVPMRepRef As VPMRepReference
Set oVPMRepRef = oVPMRepInstance.ReferenceInstanceOf
...
```

   The call to *ReferenceInstanceOf* property of `VPMRepInstance , oVPMRepInstance` returns the Product Representation Reference `oVPMRepRef`, `VPMRepRef`

   Fig.2 Message box showing success in retrieving Representation Reference

   his output is depicted in [Fig. 2] above, which corresponds to an end-user selecting the Mono Representation Instance (MonoRepresentation1) under the Prc Message box displays the PLM_ExternalID of the Mono Instantiable Representation Reference.

3. Retrieve Product Reference aggregating the Product Representation Reference

   Now that we have retrieved the Representation Reference, we proceed to seek its parent, the Product Reference that aggregates it. For reasons mentioned ea MonoInstantiable Representation Reference exists, for the simple reason, that it has a single Instance, whose parent could be treated as the parent of the Mo

   The concept of a parent, is non-existent for a Multi-Instantiable Representation Reference, for reasons mentioned later.

```
...
Dim oProdReference As VPMReference
Set oProdReference = oVPMRepRef.Father

MsgBox ("Success in retrieving aggregating Product Reference from Representation Reference. Product Reference Id = ") + oProd
...
```

   The call to *Father* property of `VPMRepReference, oVPMRepRef` returns a aggregating Product reference of Product Representation Reference, as a `VPMRef`

   Note: The call to *Father* is likely to fail under two circumstances

   - If the Representation Reference is MultiInstantiable, each Representation Instance would have its own parent. The concept of parent is irrelevant for r

- If the aggregating Product Reference is not loaded in session.

Fig.3 Message box which showing success in retrieving Product Reference

The output is depicted in [Fig. 3] above. It represents a Message Box, which displays the PLM_ExternalID of the Root Reference (Product1), a Product Ref Representation Instance (MonoRepresentation1), as depicted in the Product hierarchy in [Fig. 1] above.

Fig.4 Message box showing failure

The error, depicted in [Fig. 4] above, is typically generated if the user selects a MultiInstantiable Representation Instance (e.g. if we select MultiRepresentat hierarchy represented by Fig.1) or if the parent is not loaded in session, though the latter scenario is unlikely to occur (beyond the scope of the current UC, t

References

[1] Modifying PLM Attributes of a Selected Product Object

Retrieving Part Object From Product Root Reference

This use case primarily focuses on the methodology to retrieve the "Part" object, associated with a 3DShape representation in Product context. The UC actually navigates down a Product Reference which aggregates beneath it, this 3DShape Representation.

Related Topics
Launching an Automation Use Case

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdPstUcRetrievePartObjectFromProductReferenceSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

This use case can be divided in 2 steps

1. Search and Retrieve the Product Reference (Root) from underlying database
2. Retrieve Part Object associated with the 3DShape aggregated beneath the Root

1. Search and Retrieve the Product Reference (Root) from underlying database

```
...
 Dim oVPMRootReference As VPMReference
 Set oVPMRootReference = SearchAndRetrieveProdReference()
...
```

The function *SearchAndRetrieveProdReference*, returns `oVPMRootReference`, a `VPMReference` type [1]. The PLM_ExternalID and revision values are sought as input from an end-user. A search criteria is built with those inputs and the PLM type as a Product Reference type. When this query is run on an underlying database, it typically returns a list of entities listed within a search editor. We retrieve the first element in this list, load it in the current session and retrieve the associated Product Reference.

Please note that the Product Reference object for which we are searching it should necessarily have 3DShape instance beneath it.

2. Retrieve Part Object associated with the 3DShape aggregated beneath the Root

```
...
Dim oPartAs Part
Set oPart = RetrievePart(oVPMRootReference)
...
```

This function *RetrievePart* returns `oPart`, a `Part` type, a root object associated with a 3DShape.

This function navigates down the product hierarchy, to seek the first 3DShape Representation Instance beneath Root. It further retrieves the "Part" Object associated with the 3DShape (Product Representation Reference).

The function *RetrievePart* details are as in the below sub steps.

I. Retrieve list of Product Representation Instance beneath Product Reference

The subroutine begins with retrieving Product Representation Instance collection under Root.

```
Function  RetrievePart(oProdRef) As Part
...
 Dim oVPMRepInsts As VPMRepInstances
 Set oVPMRepInsts = oProdRef.RepInstances
...
```

A call to the *RepInstances* method on the Root Reference, `oProdRef`, a `VPMReference` type returns a collection of Product Representation Instances, `oVPMRepInsts`, a `VPMRepInstances` type under Root.

II. Retrieve the Product Representation Instance(First element among the list of instances retrieved earlier)

```
...
 Dim oVPMRepInst As VPMRepInstance
 Set oVPMRepInst = oVPMRepInsts.Item(1)
...
```

The call to *Item* method on `VPMRepInstances, oVPMRepInsts` returns the first Product Representation Instance `VPMRepInstance,oVPMRepInst` aggregated beneath it by giving 1 as input index.

If there is no Product Representation Instance beneath a Root Reference, it results in an error, which is handled exclusively at the end of this routine. Please note that here we consider that the first element in the list of Product Representation Instance is 3DShape, else macro will fail in further steps.

III. Retrieve the Representation Reference of the Representation instance (we suppose later it is a 3D Shape).

Next, we retrieve the Representation Reference associated with each Product Rep Instance, just retrieved.

```
...
 Dim oVPMRepRef As VPMRepReference
 Set oVPMRepRef = oVPMRepInst.ReferenceInstanceOf
...
```

The call to *ReferenceInstanceOf* property of `VPMRepInstance , oVPMRepInst` returns the Product Representation Reference `oVPMRepRef` associated with it.

IV. Retrieve Part object associated with the 3DShape

```
...
 Dim oPart As Part
 Set oPart = oVPMRepRef.GetItem("Part")
...
```

A call to *GetItem* method of `AnyObject` on `oVPMRepRef`, returns the associated Part Object, `oPart` (a Part type) for this we have given "Part" string as input.

Please note that Part is not an Object name but an Object Type. In addition, this call ensures that retrieved element is 3DShape since associated element is Part.

References

[1] Opening Product Reference

Generating Bill Of Materials (BOM)

This use case fundamentally illustrates generation of the Bill of materials. The use case navigate through the instance/reference Product structure and calculates the count of each leaf node's occurrences. Then it displays status of each leaf node along with Product Structure and finally displays the Bill of material in tabular format.

Related Topics
Launching an Automation Use Case

Before you begin: Note that:

- Launch CATIA

Where to find the macro: CAAScdPstUcBOMSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

This use case can be divided in 4 steps

1. Retrieves Root Reference from the database
2. Navigates the Product Reference Recursively
3. Gets Leaf Node Status (New or existing) And Add Count
4. Displays Bill of Materials

1. Retrieves the root reference from the database

   As a first step, the Use case retrieves a Product Reference (Root) from database.

   It begins with a call to **SearchProduct** function. This function searches for a list of Product References from the underlying database based on an input sear
   Search Result window in CATIA. We build up the search criteria, with the PLM Entity type of a Product Reference type, PLM_ExternalID and revision as

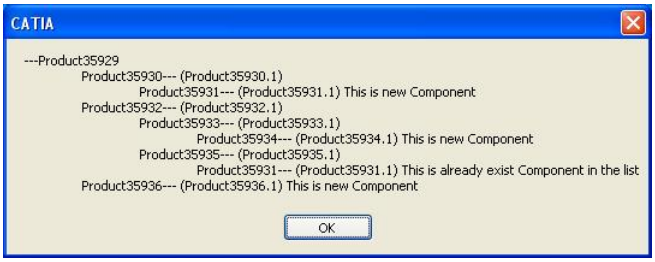
   As a next step, the UC essentially loads in session first Product Reference (Root) output by search collection occurring in the new tab page within Search Ec

   The steps of retrieving Root in session from database explained in detail in [1] article.

2. Navigates the Product Reference Recursively

   Now Use case retrieves the Product Reference (Root) associated with its reference-Instance Model and then navigates down its product hierarchy to display
   seen in the specification tree within CATIA, with the Representation Instances/ References as shown in the Fig.1 below. Please Note that in this case we are
   reference elements in addition to that we are showing the status of leaf node References.

   Fig. 1 The sample out of Product structure with updated status




```
...
    NavigateProductReference  oProductEditor
...
```

The function *NavigateProductReference* takes an `Editor`, `oProductEditor`, as an input.

The function *NavigateProductReference* details are given in the [2] use case. The only additional part in this method is explained here below rest is same as sections are basically added to generate Bill of Material.

I. Retrieves the Leaf nodes

In this section our main aim is to retrieve leaf node Product Reference while navigating through instance/reference Product structure. To identify leaf Instances (children) aggregated under Product Reference, The zero count of aggregated Reference instances proves that the current Reference is Leaf

```
Sub NavigateProdReference(oProdRef, depth)
    ...
    Dim oListChildrenInstances As VPMInstances
    Set oListChildrenInstances = oProdRef.Instances

    For i = 1 To oListChildrenInstances.Count

        Dim oVPMInst As VPMInstance
        Set oVPMInst = oListChildrenInstances.Item(i)
        ...

        Dim oVPMRef As VPMReference
        Set oVPMRef = oVPMInst.ReferenceInstanceOf

        ...
        Dim  oListChildrenInstToIdentifyLeafNode As VPMInstances
        Set  oListChildrenInstToIdentifyLeafNode= oVPMRef.Instances

        Dim  StrNewObjRef As   String
        StrNewObjRef = ""

        If 0 = oListChildrenInstances2.Count Then
...
```

A call to *Instances* Property of the `VPMReference`, retrieves the `VPMInstances` a collection object, `oListChildrenInstToIdentifyLeafNode`

Further we check for count of Instances in the collection object `oListChildrenInstances2`, if that is zero then we consider that current reference ob Reference.

II. Count and display leaf node occurrences

After confirmation of leaf node Product Reference use case makes an entry of this Product Reference inside the array. This array will be further used node reference or already existing leaf node Reference). In addition to this Use case maintains the count of each leaf node occurred in the instance ref

```
...
           StrNewObjRef = GetLeafNodeStatusAndAddCount(oVPMRef)
        End If

        strBrowsedPLMCompIDAttr = strBrowsedPLMCompIDAttr + oVPMRef.GetAttributeValue("PLM_ExternalID") + oVPMRef.GetAtt
        strBrowsedPLMCompIDAttr = strBrowsedPLMCompIDAttr + oVPMInst.GetAttributeValue("PLM_ExternalID") + ")" + StrNewO

        NavigateProdReference oVPMRef, depth + 1

    Next i
...
```

A call to *GetLeafNodeStatusAndAddCount* function returns the status (New/Existing Product Reference) of the input leaf node `oVPMRef` .

Finally we show the Product Structure with updated status [Fig.1].

The function *GetLeafNodeStatusAndAddCount* described below in step 3

The elements those are not leaf node we browse though them recursively to get leaf nodes by calling *NavigateProdReference* method recursively.

3. Gets Leaf Node Status (New or existing) And Add Count

To create BOM it is very important to identify the Leaf node (this step we already done). Next important thing is to identify the count of each leaf node's oc structure.

For calculating occurrence count first we need to find out new Reference Component. Then if component is new then mark there entry in the array list and i the entry in the array list in that case we increase its count. These occurrence counts also we are maintaining in the array.

In this way we get the total count of each leaf node entry as well status of each leaf node. steps elaborated further below.

please note that due to this array we have some limitation we can only count that many new Components as much the size of array is declared, in this case w

i. We Parse through the component list array, this array contains the PLM_ExternalID attribute values. We compare current leaf node's PLM_ExternalI flag as true if entry found in the array list.

Further if object is already exist in the list then we increase the count, this maintains the count of each occurrence of Leaf node Reference

```
Function GetLeafNodeStatusAndAddCount(oVPMLeafRef) As String

    Dim  bObjFound As   Boolean
    bObjFound = False

    For g = 0 To 10
       If lstCompList(g) = oVPMLeafRef.GetAttributeValue("PLM_ExternalID") Then
          bObjFound = True
          lstCompCount(g) = lstCompCount(g) + 1
       End If
    Next g
...
```

A call to *GetAttributeValue* method returns PLM_ExternalID attribute value. which we compare with each element of array, and if values are same th

Next if we found entry already exist then we add its count.

ii. Update the status (New or already exist in list) and for new element make an entry in the array.

```
...
    Dim  strIsNewObject As String
    strIsNewObject = ""

    If False = bObjFound Then
       lstCompList(iNewObjectIndex) = oVPMLeafRef.GetAttributeValue("PLM_ExternalID")
       lstCompCount(iNewObjectIndex) = 1

       iNewObjectIndex = iNewObjectIndex + 1
       strIsNewObject = " This is new Component"

    Else
       strIsNewObject = " This is already exist Component in the list"
    End If
...
```

After identifying new leaf element (flag check) for each new element we make its entry in the array and initialize count to 1 (one).

Further we update the status as Component is new if component is new otherwise update status as already exist in the list.

4. Displays Bill of Materials

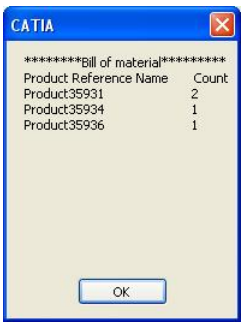
Display action of BOM happens at the end of use case.

Actual calculation of the each leaf node Product Reference and their count of occurrence is already done. In this step we are only displaying aggregated bill

```
...
    Dim  strBOM As   String
    strBOM = "********Bill of material*********" + vbCrLf
    strBOM = strBOM + "Product Reference Name    " + "    Count" + vbCrLf
    For k = 0 To 10
        Dim  strCount As   String
        strCount = lstCompCount(k)
        strBOM = strBOM + lstCompList(k) + "                          " + strCount + vbCrLf
    Next k

    MsgBox strBOM
...
```

Here we parse though the array list and Build the String `strBOM` with output of PLM Reference name and their count . And finally we display the message b

Fig.2 Sample Bill of Materials



The above image [Fig.2] shows the sample output of the Bill of Materials.

References

[1] Opening Product Reference
[1] Browsing Product Contents

Positioning Product Instances

Technical Article

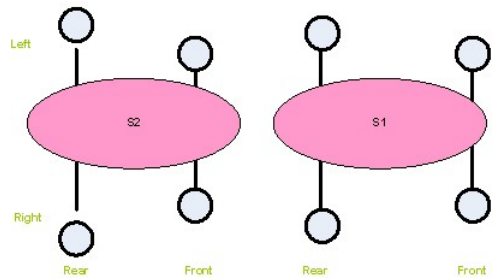
Abstract

This technical article primarily deals with the fundamentals of positioning the PLM entities inside an assembly model. The first section explains the basic positioning of a PLM entity followed by the concept of flexibility, an ability to overload the positions in various context. The second section details the APIs involved in the process.

Instance Positioning Principles

Our article illustrates the concepts of positioning a product instance using the model depicted below. Its what we call as the "Double Skate" model, since it consists of two elementary skate assemblies.

Fig. 1 The Double Skate Model



The model (rather fictional) consists of:

- Two elementary Skates, each made up of two Axles (named AxleAssy) and a Board. The AxleAssy is made up of two wheels and an Axle (name Axle).
- The right Skate (named S1) has its right (or Front) AxleAssy closer to the board's border with respect to the left Skate (named S2).
- On Skate S2, the wheels of the rear AxleAssy are more on the periphery with respect to S1.

Put on your helmet and Knee protection, we start.

Positioning Without Overloading

We begin by creating the Skate.

The first step involves creating an AxleAssy, comprising of an Axle and two Wheel instances at either end of the Axle.

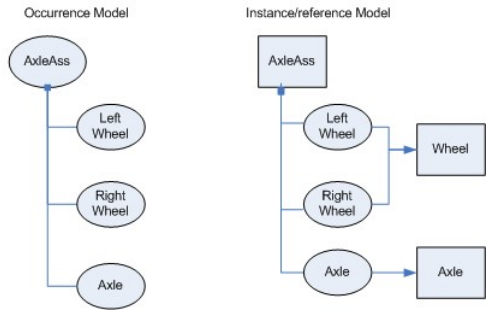
Fig. 2 The AxleAssy

On instantiation of the two wheels, they are seen as overlapped one over the other in the AxleAssy. The two Wheels and the Axle are located at the position absolute (0,0,0) in the global coordinate system of the AxleAssy.

Fig. 3 AxleAssy On Initial Creation

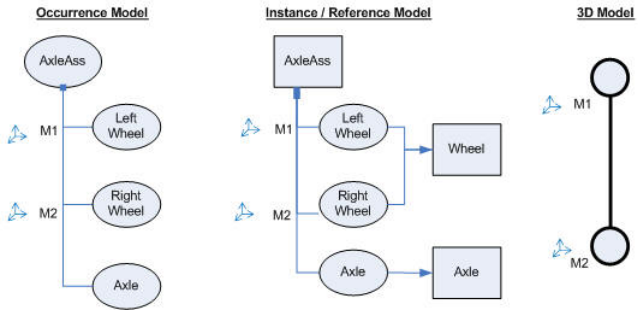
Now, it is known we have two distinct data models associated with a Product namely. the Occurrence and the Ref-Inst model. Both these models for the AxleAssy are depicted in the schema that follows.

Schema 1 AxleAssy Data Model



We proceed to relocate the two Wheels at either end of the Axle. To accomplish this, two possibilities exist. Either we position the wheel Occurrence or its related Instance. In either case, the resulting model is as depicted in the Schema 2 that follows.

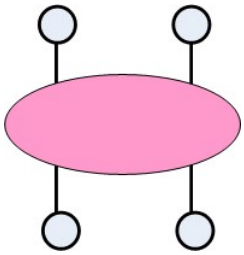
Schema 2 Wheel Positioned at Ends of Axle



- Position the left wheel at M1 and right wheel at M2, absolute positions relative to the global co-ordinate system of the AxleAssy. The associated occurrences and their related instances are seen positioned at M1 and M2.

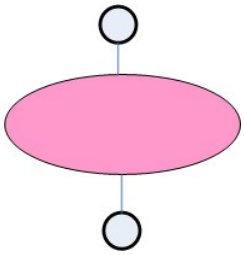
We next proceed to create the Skate. The Skate comprises of a board with two AxleAssy positioned at either end of the board as depicted in the picture that follows.

Fig. 4 The Complete Skate



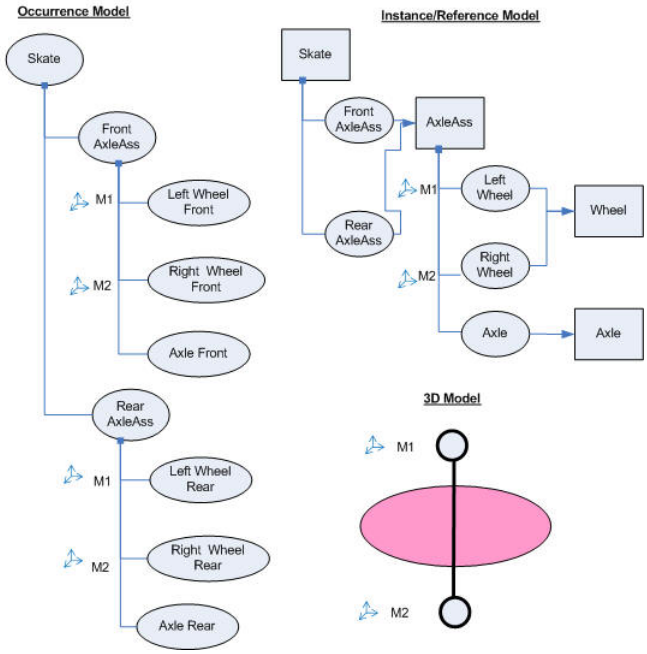
To begin with, we will have the two AxleAssy overlapping one another as seen below. The AxleAssy and the Board are positioned at absolute (0,0,0) in the global co-ordinate system of the Skate.

Fig. 5 Skate on Creation



The schema that follows, represents the data model of the Skate on creation.

Schema 3. The Data Model of Skate on Creation

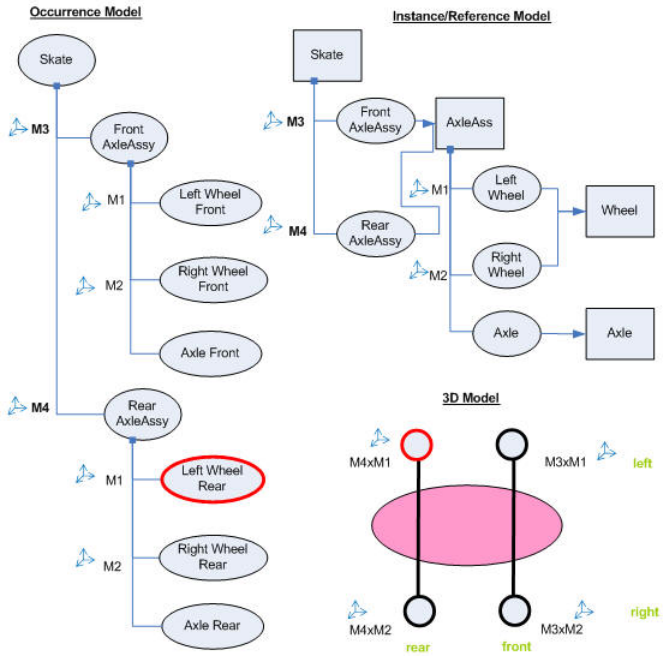


- The Front and the Rear AxleAssy are positioned at the origin (the absence of the position matrix in the schema, indicates placement at Origin).
- The Left and Right wheels were positioned at M1 and M2 respectively, in the earlier step for both models.

We proceed further to reposition the two AxleAssy at either end of the Board. That will create a complete Skate, as depicted in [Fig. 4] above. The schema

representing the data model of the complete skate follows.

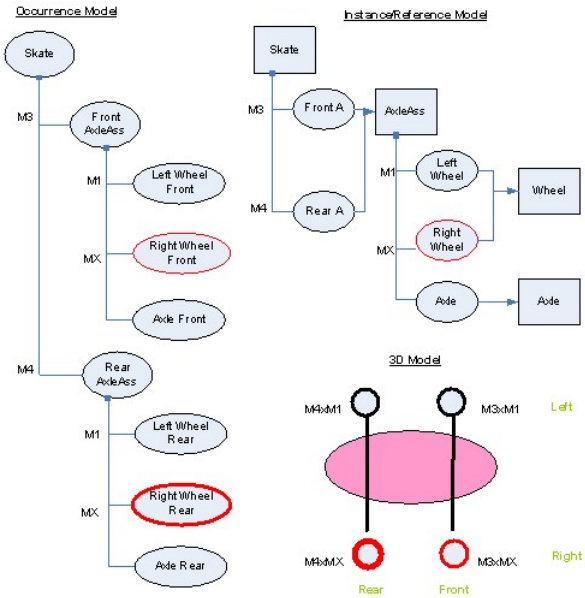
Schema 4 The Data Model of the Complete Skate



- M3/M4 is the absolute position matrix of the front and the rear AxleAssy in the global coordinate system of the Skate.
- M1/M2 is the absolute position matrix of the left and the right wheel occurrences/instances.
- The position of each wheel (in 3D Data model) is the product of the position matrix of each occurrence beginning with that under Root to the Wheel occurrence. Thus the position matrix for the rear left wheel in 3D (in red) is M4 (rear AxleAssy occurrence) X M1 (LeftWheel occurrence).

Now let us proceed to reposition one of the wheels in the Skate assembly. We will relocate the rear right wheel to **M4xMX** as depicted in the 3D Model within the figure that follows.

Schema 5. The Data Model of a Skate on Repositioning its Wheel Occurrence



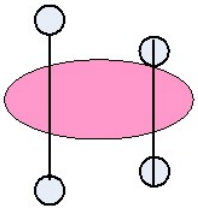
- The rear right wheel was repositioned in 3D at **M4xMX** (in bold red in the 3D model).
- It results in the right wheel occurrence and its related instance getting repositioned to MX.
- This results in the front right wheel getting repositioned to **M3xMX** in 3D.

To sum up, if an entity inside a 3D Model is repositioned, its associated Occurrence (and related Instance) is relocated. The impact is then propagated to all associated Occurrences in the 3D model.

Introduction to Overloaded Position, Flexibility & Repositionability Concept

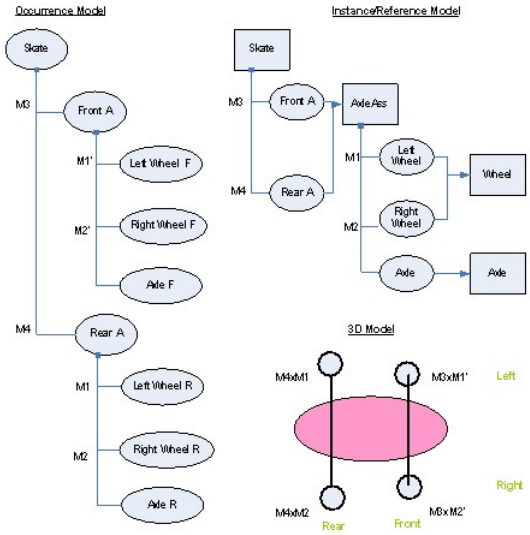
Did you notice in the previous case, repositioning the rear right wheel resulted in repositioning the front right wheel as well ? But, this may not be our expectation. We may want for example, only the two front wheels to be drawn closer, without affecting the rear wheels, as the picture that follows indicates.

Fig. 6 Skate with the Closer Front Wheels



In other words, the position of the front two wheel occurrences is specific to the Skate context. Meaning, it is overloaded in the skate context. The schema that follows represents the data model for the skate with its front wheels drawn closer to one another.

Schema 6. Skate Data Model with Closer Front Wheels



- The front left wheel is repositioned at M3x**M1'**, while the front right wheel is repositioned to M3X**M2'**, as depicted in the 3D model.
- **M1'** is the position matrix of the left wheel occurrence, locally in the Skate context (while that of its related instance is M1).
- **M2'** is the position matrix of the right wheel occurrence, locally in the Skate context (while that of its related instance is M2).

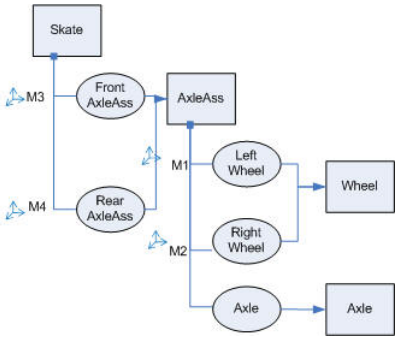
When the matrix of an Occurrence is not the same as its related Instance, the position of the Occurrence is said to be overloaded. Two obvious questions that arise here:

- Who stores this overloaded information? The answer is it is the Reference-Instance model which stores this information (M1', M2'), which is retrieved each time we recreate the Skate occurrence model, and other occurrences model which are built over the Skate occurrence model (like the "Double Skate" in the next section). Moreover please note that the Ref-Inst model is persistent (stored in database), while the Occurrence model is transient (not stored).
- How is this overloading accomplished?

  This is possible if the AxleAssy Reference is made "**flexible**", and the two wheel instances in it, are "**repositionable**".

  You may either open AxleAssy reference standalone or in the Skate context, the AxleAssy be UI-Activated. The using the "**Properties**" command, the "**Mechanical Behavior**" page, you check the "**Flexible at Instantiation**" button, and you check the two instances of Wheel as "**Repositionable**".

Fig. 7 Setting Flexibility to Reference Dialog

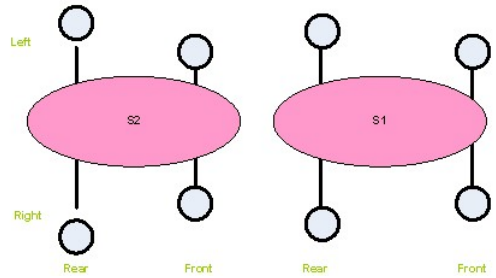


We proceed to a more generic explanation on this subject of "**Position Overloading**".

Overloaded Position in detail

In order to introduce a genericity in our discussion, we'll have to introduce another level in our Occurrence data model. It is why we propose a "Double Skate" model, which consists of two elementary skate assemblies, as depicted in the Fig. that follows.

Fig. 8 The Double Skate Model

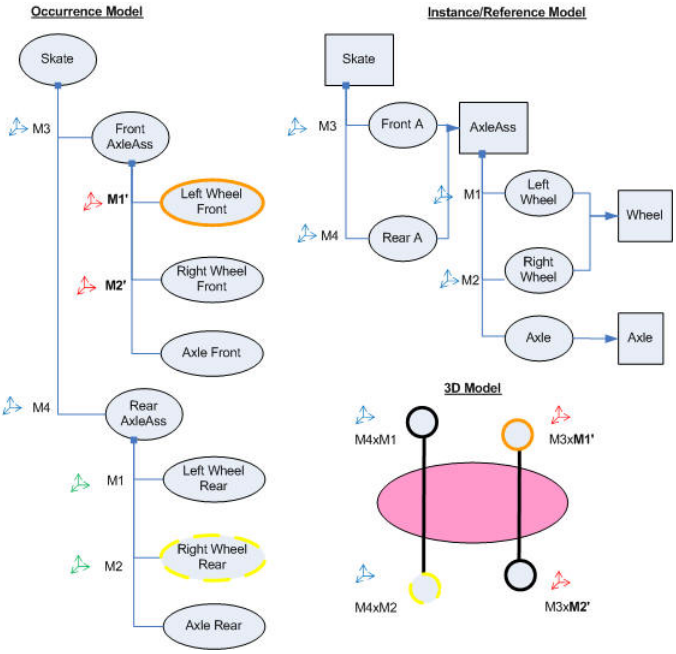


It simply consists of two instances of the Skate reference namely. S1 and S2.

A significant point to note here is that in S1 (right skate assembly), the front AxleAssy is closer to the skate board's border. Also in S2 (left skate assembly) you'll notice the the wheels of the rear AxleAssy are more towards the periphery. These are the two cases of "overlaoded occurrences" in the "Double Skate" context.

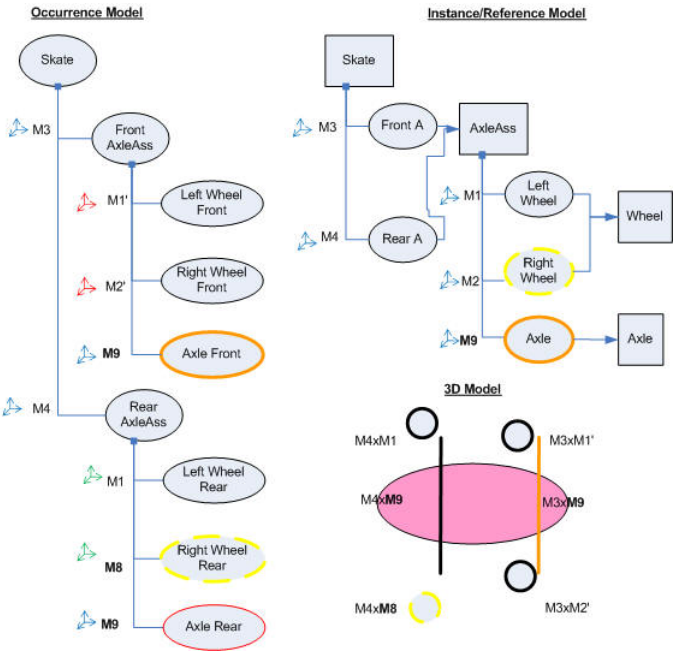
To begin with, we create the "Double Skate" assembly, simply by positioning S1 at M5, and S2 at M6. The model is depicted in the schema that follows. Ofcourse, you see the two skates distinctly (not overlapped) since we have located them at distinct locations (M5, M6).

Schema 7 Data Model of a Double Skate on Creation



We next proceed to overload the occurrences in the "Double Skate" context, so that we have the final model as depicted in the [Fig. 8] above.

Schema 8 Data Model of a Double Skate on Overloading



The overloaded occurrences in the "Double Skate" context, are circled in red above, in the Occurrences model. You will find that the two S2-rear AxleAssy

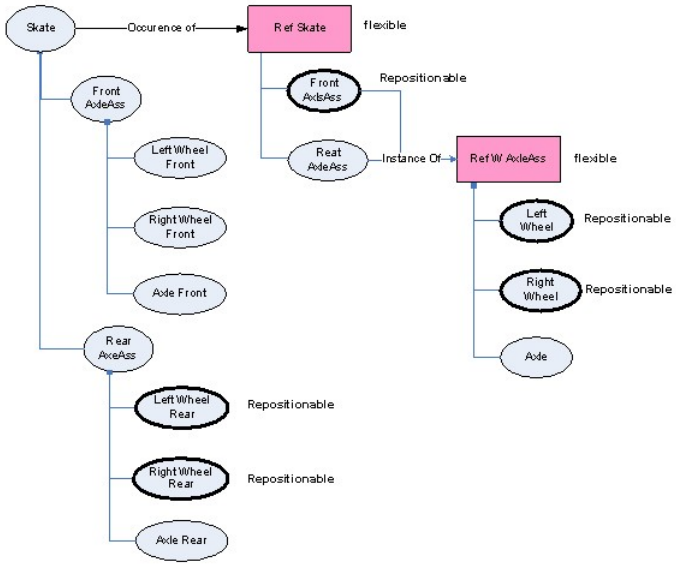
Wheels are overloaded (more towards the periphery, **M1''**, **M2''**) and the S1-Front AxleAssy is closer to the board (**M3'**).

To realize this model you will have to do the steps listed below:

- To move the S1-Front AxleAssy independently (in the "Double Skate" context), the Skate Reference will have to rendered flexible, and within it, the Front AxleAssy instance will have to be rendered repositionable.
- To move the two S2-rear AxleAssy Wheels independently, we will have to:
  - the AxleAssy Reference will have to be rendered flexible, and within it, the two Wheel instances will have to be further rendered repositionable.
  - The Skate Reference will have to be flexible and in the Skate context the Occurrence "Wheel" must be repositionable.

The above is effectively summed up by the schema that follows

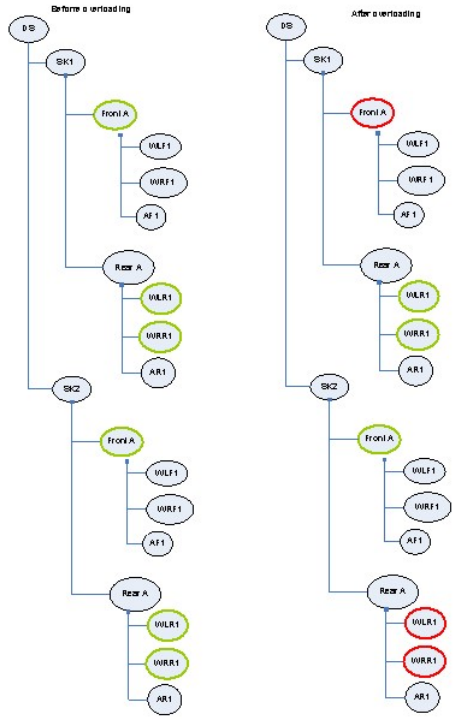
Schema 9 Reference Flexibility and Instance/Occurrence Repositionability



A significant point to note in the schema above is that when an occurrence at first level (say the Front AxleAssy) is to be set repositionable, it is the related instance which is set repositionable. This is because the instances are tightly bound to the occurrences at the first level.

Now with this configuration, the schema that follows represents the occurrences which can be potentially overloaded and those which have been actually overloaded resulting in our model depicted in [Fig.] above.

Schema 10 Potentially Overloadable Occurrences and Overloaded Model



- On the left, we have the occurrence model which represents in green, those occurrences which can be potentially overloaded
- On the right, we have the occurrence model after we have the occurrences overloaded, those represented in red. You'll find we have overloaded the Front AxleAssy of Skate1 and the two wheels of the rear AxleAssy of Skate2.

To conclude, to overload an occurrence, the rule is for all the occurrences from the one just below the root to the occurrence aggregating the one for overload:

- The references of all these occurrences must be flexible.
- In the occurrence model computed from these references, the occurrence/instance (for first occurrence) corresponding to the one to be repositioned, must also be repositionable.

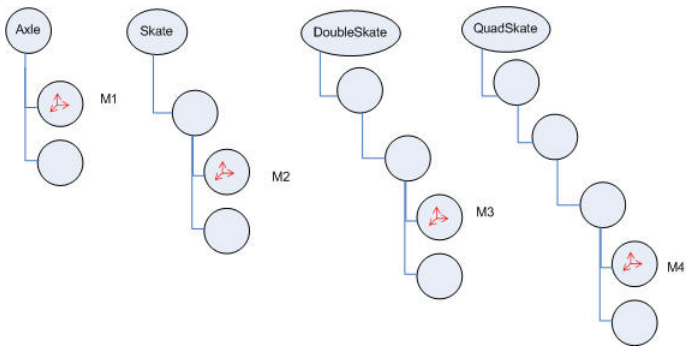
Attention: While reading this TA, you may have an impression that the person who intends to overload an occurrence will render flexibility to references and repositionability to instances/occurrences. It is possible, but it is usually not the case. In fact, it is for the person responsible for a reference to decide the flexibility status (on/off) for the reference, and further, which instances/occurrences beneath it will be be rendered repositionable.

Flexibility and Repositionability Removal

To finish this section, we have to explain the consequences of rendering a reference inflexible or one of its instances/occurrences no longer repositionable, on an occurrence which is overloaded. This section is too technical, and we recommend beginners skip it, for the moment.

When a reference is no more flexible, all its instances/occurrences which were so far repositionable, cease to be so. We explain that with the schema 11 that follows.

Schema 11. Consequences of Disabling Flexibility



On schema 11. we consider an instance (e.g., a wheel) used across several occurrences model. As we progress from left to right, we add a level to the occurrences data model (AxleAssy, Skate, DoubleSkate, QuadSkate)

In the first model, the position matrix for the wheel occurrence is represented by M1. As we progress to the right M2, M3 and M4 are the overloaded matrices in the respective contexts.

Now we consider possible configurations on removing the flexibility of the references:

- AxleAssy is not more flexible. In this case, M2 becomes M1 and so do, M3 and M4.
- Skate is no more flexible. In this case M2 remains M2, while M3 and M4 become M2.

Instance Position by APIs

Since this article is independent of a programming language, you'll not find any reference to an API in it. The APIs are specifically dealt in various Use Cases we have created for each. The focus here is on the common principles which follow:

- Handling the position of an Occurrence or Instance.
- Handling the position in an absolute coordinate system.
- Handling the position in a local coordinate system.

Occurrence and Instance Position

In the instance/ reference model, only instances are movable, while in the occurrence model, except for the root occurrence, all other occurrences are movable.

Position Matrix

A Position Matrix is an entity defined by 12 floats as depicted below:

[0] [3] [6] [9]
[1] [4] [7] [10]
[2] [5] [8] [11]

where:

- coefficients from 0 to 8 define the (3x3) transformation matrix.
- Coefficients from 9 to 11 define the (x,y,z) vector position.

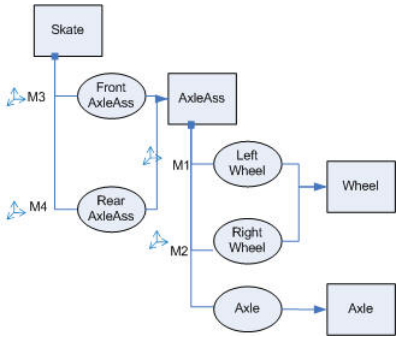
The matrix must be reversible and isometric.

Instance Position

When you work with Instances in a Ref-Inst model, irrespective of the used method (Absolute or not).

When you work on instances (instance/reference model) whatever the used method, absolute or not, you directly work on the instance. Have a look to this schema.

Schema 12 Handling Instance Matrix



- When you modify or retrieve the matrix of the Right Wheel, you modify or retrieve the Matrix of the instance (in this case, M2).
- When you modify or retrieve the matrix of the Front AxleAssy, you modify or retrieve the matrix of the instance, yet again (in this case, M3).

Working with an Ref-Inst model (handling positions, specifically) is relevant while we build an assembly, but otherwise we recommend working with an occurrence model, which is more accurate and safe, since the values of the overloaded matrices are stored inside an occurrence model.

Example of a Dangerous Scenario: If the position of the front left wheel (Skate Schema 6) is computed, using the Ref-Inst model, it will work out to M1xM3, where M1 is the position matrix for the left wheel instance, while M3 is that for the front AxleAssy instance. This result is incorrect, since in the Skate context the front wheels were drawn closer to the centre and its position is M1'xM3.

Occurrence Position

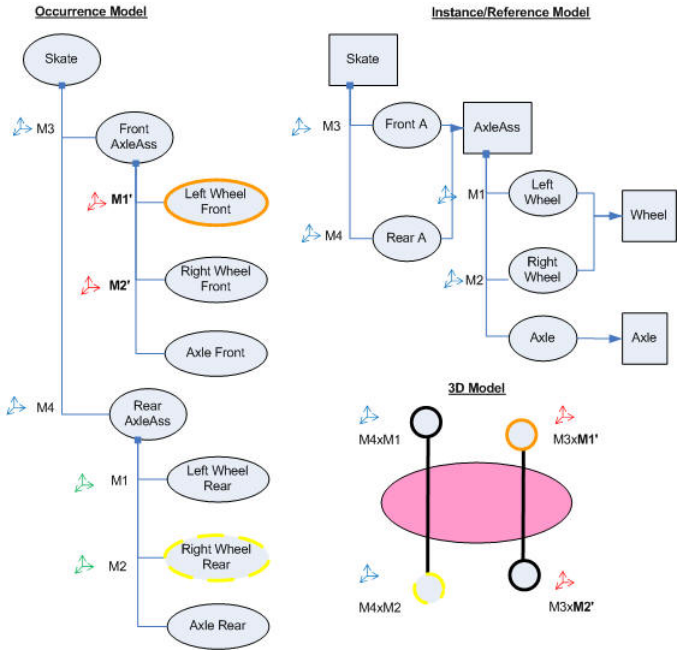
The section deals with reading and writing the position of an Occurrence.

Read an Occurrence Position

The position of an Occurrence is a function of the context in which it is sought. We have two possible cases:

- Absolute Context: The position retrieved is in the root context.
- Intrinsic Context: The position retrieved is in the context of its aggregating parent.

Schema 13 Reading an Occurrence Position



In the schema above,

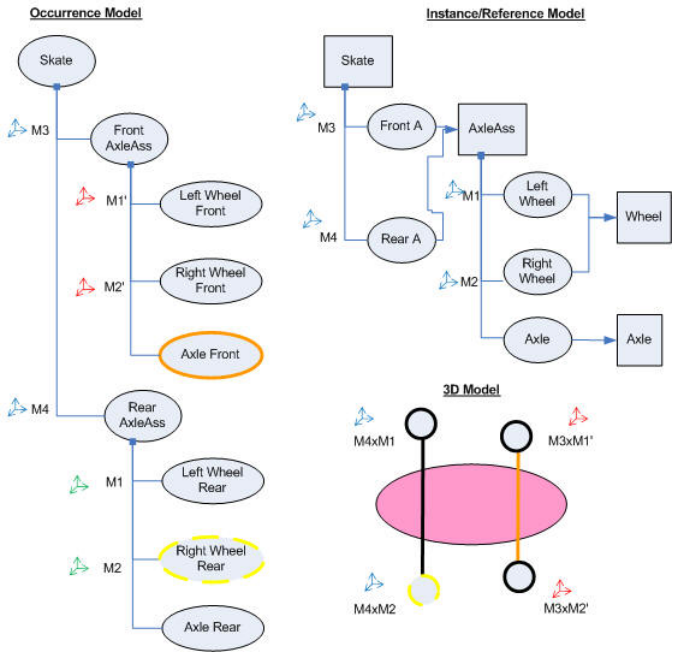
- You retrieve the position of the Right Wheel Rear Occurrence (bold yellow-dashed):
  - In the absolute context it returns M4xM2.
  - In the intrinsic context it returns M2.
- You retrieve the position of the Left Wheel Front Occurrence (bold orange):
  - In the absolute context it returns M3xM1' (M1' is the overloaded matrix in the Skate context).
  - In the intrinsic context it returns M1'.

Write an Occurrence Position

An occurrence can be positioned in two contexts:

- Absolute context: The new value of the Occurrence is computed such that it satisfies the equation: InputMatrix = NewMatrix of the Occurrence X Matrix[i] relative to Root.
- Intrinsic context: The new value of the occurrence is the matrix input.

Schema 14 Writing an Occurrence Position



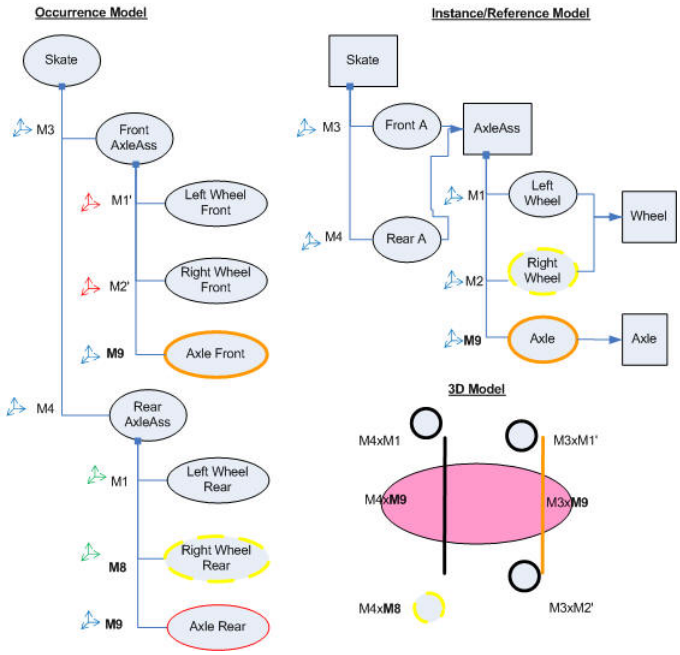
- We intend to move the Right Wheel Rear Occurrence (yellow-dashed) to M8:
  - In the absolute context, M2 is replaced by Mx, such that M8 = M4xMx.
  - In the intrinsic context, M2 is replaced by M8.
- We intend to move the Front Axle Occurrence (orange) to M9:
  - In the absolute context, the identity matrix is replaced by Mx, such that M9 = M3 x Mx.
  - In the intrinsic context, the identity matrix is replaced by M9.

This discussion will be incomplete without a specific mention of the fact that:

- When an Occurrence is repositionable, the write operation only modifies the Occurrence.
- When an Occurrence is not repositionable, the write operation modifies the Occurrence as well as its related Instance.

We describe specific cases based on the schema that follows which will help you understand that better.

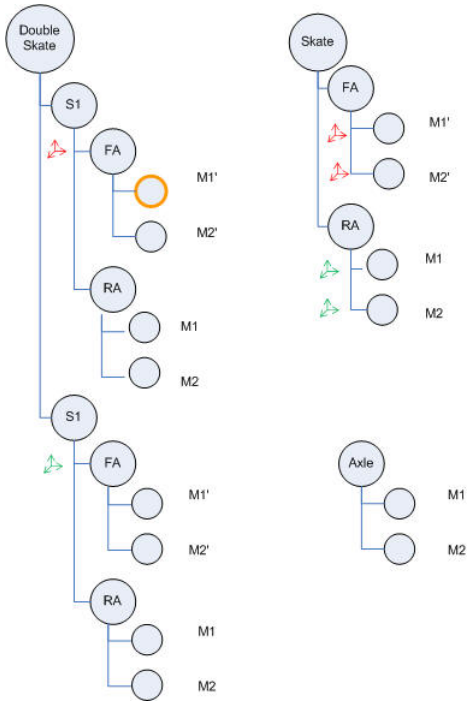
Schema 15 Impact on Repositioning an Occurrence



- Irrespective of the context (Absolute or Intrinsic) when we move the rear right wheel occurrence (bold yellow dashed), to the position M8, since this occurrence is repositionable, the position of only the occurrence is changed. The Ref-Inst model remains unchanged.
- Irrespective of the context (Absolute or Intrinsic) when we move the front Axle Occurrence (orange bold), to the position M9, since this occurrence is not repositionable, its related instance is repositioned to M9 and further all occurrences associated with this instance are also impacted (orange in the occurrence model). The two axles move on the Skate.

To complete this discussion of "Writing an Occurrence Position" we must introduce another level to this Occurrence Model. We now use the "Double Skate" model with a slight difference, which will let us be more exhaustive in our examples that follow. Please take a look at the schema that follows.

Schema 16 Impact on Modifying a Non-Repositionable Occurrence



In the schema above, we have three occurrence models. The color conventions we have followed is:

- If the coordinate system is not shown, it implies the occurrence cannot be overloaded.
- A green coordinate system indicates the occurrence can be overloaded (potentially yes).
- A red coordinate system indicates, the occurrence is overloaded.

By consequence, you will observe that:

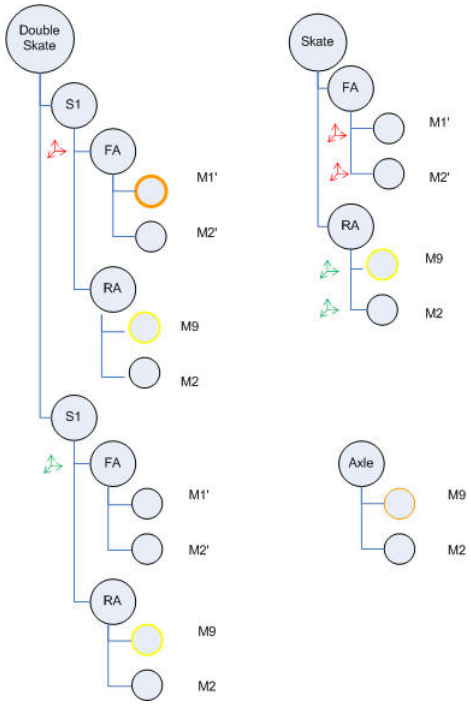
- In the double skate, only the front AxleAssy can be overloaded, and S1-Front is overloaded.
- On the simple skate all the wheels can be overloaded, and the ones on the front AxleAssy are overloaded.

We will now present several configurations wherein we modify the position of an occurrence within the "Double Skate" occurrence model. The first two are based on the same concept as previously:

- Irrespective of the context (Absolute or Intrinsic), when we move the front AxleAssy occurrence since these are repositionable, the position of only the modified occurrence is changed. (You'll notice the front AxleAssy of S1 is overloaded while not that of S2).
- Irrespective of the context (Absolute or Intrinsic) when we move the rear Axle occurrence since these are not repositionable, the position of its related instance is changed and further all occurrences associated with this instance are impacted. The two rear AxleAssy move in the DoubleSkate context.

Let us now consider a concrete example. We move the S1-Front LeftWheel, marked in orange bold to the position M9, in Schema 17 that follows. The result is also depicted in the following schema.

Schema 17



- Since the S1-Front LeftWheel occurrence cannot be overloaded, the position of its related instance updates to M9 (orange in the Axle model).
- In the skate model, the non-overloaded left wheel repositions to M9 too (bold yellow in the skate occurrence).
- In the DoubleSkate model, we have two occurrences at M9 (bold yellow in double skate), the instantiation of the non overloaded occurrences of the Skate model.

In Short

This TA primarily dealt with the fundamentals of positioning entities within a product model and its corelation with the associated entities in the Occurrence and the Ref-Inst model.

References

History

Version: **1** [Mar 2010] Document created

Positioning Product Model

The use case mainly deals with Positioning of the occurrence of Skate model objects.

Before you begin: Note that:

- Launch CATIA
- Create Skate model as described in the [1] article

Related Topics
Launching an Automation Use Case

Where to find the macro: CAAScdPstUcPositionProductModelSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

This use case can be divided in 3 steps

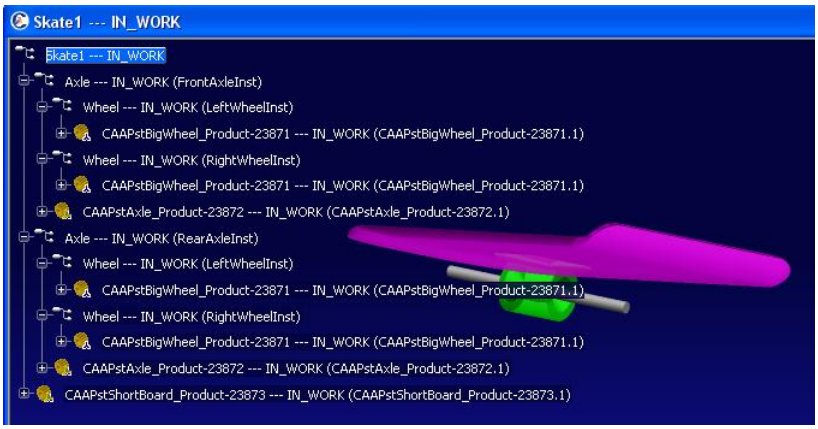
1. Retrieves the Product Reference (Skate) from underlying database
2. Positions the skate model
3. Displays the Occurrence Product Model Contents after positioning of the skate model

1. Retrieves the Product Reference (Skate) from underlying database

```
...
 Dim oVPMRootOccOnSkateRef As VPMRootOccurrence
 Set oVPMRootOccOnSkateRef = RetrieveProdRootOcc
...
```

The function *RetrieveProdRootOcc*, returns `oVPMRootOccOnSkateRef` , a *VPMRootOccurrence* type [2]. The PLM_ExternalID and V_version values are sought as input from an end-user. A search criteria is built with those inputs and a Product Reference type. When this query is run on an underlying database, it typically returns a list of entities listed within a search editor. We retrieve the first element in this list, load it in the current session and retrieve the associated Product Reference.

Fig.1 The Input Skate Model

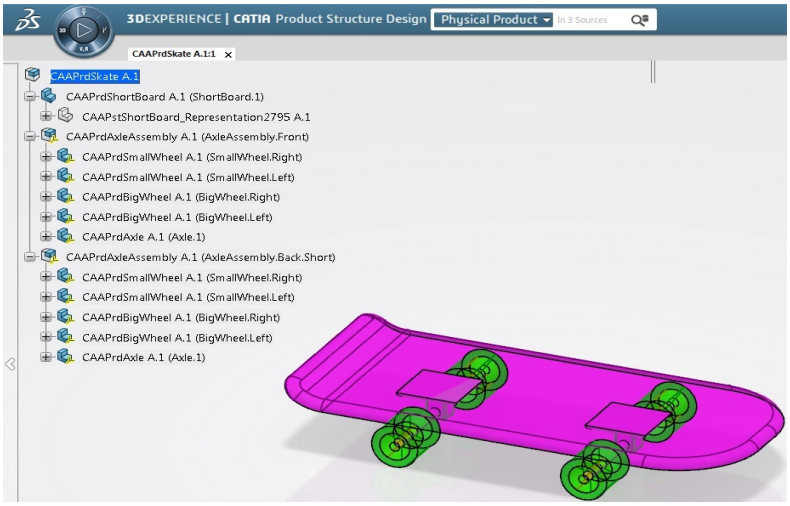


The Input model imported from database which has "Skate1" as PLM_ExternalID of root and "---" version value as shown in [Fig.1]

2. Positions the skate model

As we could see that in [Fig.1] all wheels and Axles are merged and positioned to center. Now this step will explain about positioning. The wheel and Axle instances are so positioned so that they will obtain the final Model position as shown in [Fig.2].

Fig.2 Positioned Skate model with 3DParts



This step sets the positions

```
...
 PositionSkateOccurrences oVPMRootOccOnSkateRef, 0
...
```

A call to *PositionSkateOccurrences* sets the positions of Product Occurrences. The retrieval of Occurrences are not explained here those details are explained in the [3] use case.

The *PositionSkateOccurrences* method internally calls the *SetPositionOfOccurrence* method (this is crux of this use case), some of the important points of this method are detailed below.

The *SetPositionOfOccurrence* call sets the Product occurrence positions. This method has Occurrence object and X,Y,Z co-ordinate values as inputs.

The *SetPositionOfOccurrence* method logically divided in two steps as described below

i. Creates Position matrix with input X,Y,Z values.

```
Sub SetPositionOfOccurrence(oOccurrence, x, y, z)

    Dim TransformationArray(11)
    TransformationArray(0) = 1#
    TransformationArray(1) = 0
    TransformationArray(2) = 0
    TransformationArray(3) = 0
    TransformationArray(4) = 1#
    TransformationArray(5) = 0
    TransformationArray(6) = 0
    TransformationArray(7) = 0
    TransformationArray(8) = 1
    TransformationArray(9) = x
    TransformationArray(10) = y
    TransformationArray(11) = z
...
```

`TransformationArray` is Position matrix which will define the values of position of occurrence in next step, A Position Matrix is an entity defined by 12 components as depicted below, The array initialized with the components to set to the object's position.

The first nine components represent the rotation matrix.

The last three components represent the translation vector. Here use case updates input X,Y,Z values .

ii. Sets the position of input Occurrence object defined in Position matrix.

```
...
  Dim oPositionMoveOnOcc As  Position
  Set oPositionMoveOnOcc = oOccurrence.Position

    oPositionMoveOnOcc.SetComponents TransformationArray
...
```

A call to *Position* method called on the input Occurrence (`oOccurrence` input to *SetPositionOfOccurrence* method) returns the position object (derived from ***Move***) is the 3D-axis system associated with an Occurrence object. `oPositionMoveOnOcc` is *position* object which represents PLM Product Occurrence position.

Next call to *SetComponents* method of Position Sets the relative object's position. This sets the 3D-axis system associated with the object.

3. Displays the Occurrence Product Model contents after positioning of the skate model

Displays complete model with both positions (absolute as well as relative position respectively) as shown in [Fig.3]

Initially we initialize global variable with Root Occurrence name then we Navigate through the occurrence model and retrieve name and positions of all children occurrence objects.

```
...
    NavigateAndDisplayProdOccurrencesPosition oVPMRootOccOnSkateRef, 0
...
```

A call to *NavigateAndDisplayProdOccurrencesPosition* navigates through the occurrence model and retrieves all occurrence object names with their

positions and updates these values in global string variable which will finally used for displaying Structure in message box.

The function *NavigateAndDisplayProdOccurrencesPosition* details some of the important steps related to Positioning as in the below sub steps.

i. Retrieve Absolute position of occurrence object

```
Sub NavigateAndDisplayProdOccurrencesPosition(oOccurrence, depth)
...
   Dim oPositionMoveOnOccObj As  Position
   Set oPositionMoveOnOccObj = oOccurrence.Position

   Dim oAxisComponentsArrayAbsPos(11)
   oPositionMoveOnOccObj.GetAbsComponents oAxisComponentsArrayAbsPos
...
```

A call to *Position* method returns the position object (derived from ***Move***) is the 3D-axis system associated with an Occurrence object . `oPositionMoveOnOccObj` is *position* object which represents PLM Product Occurrence position.

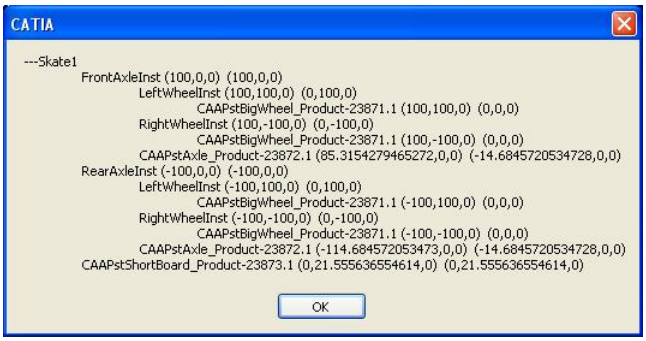
A call to *GetAbsComponents* method of Position returns the object's position in a global context. This returns the 3D-axis system associated with the object.

ii. Retrieve relative position of occurrence model

```
...
   Dim oAxisComponentsArrayRelPos(11)
   oPositionMoveOnOccObj.GetComponents oAxisComponentsArrayRelPos
...
```

A call to *GetComponents* method of Position returns relative object's position. This returns the 3D-axis system associated with the object.

Fig.3 Positioned Skate Model



The above image [Fig.3] shows the final Positioned Skate Model. This Structure contains Instance name along with their positions for eg. FrontAxleInst has one LeftWheelInst beneath it, which has

- (100,100,0) absolute value which means positions with its global 3D-axis system(in this case respective with Skate Model) (position in an absolute coordinate system)
- (0,100,0) relative value which means position with its parent object (in this case with respective to FrontAxleInst)(position in a local coordinate system).

References

[1] Creating Skate Product Model Interactively
[2] Opening Product Reference
[3] Browsing Occurrence Model

Overloading Occurrence Position

The use case mainly deals with overloading of the occurrence of 'Skate' Model objects.

Before you begin: Note that:

- Launch CATIA
- Create Positioned Skate model as described in the [1] Use Case
- Position the Skate Model using the previous macro CAAScdPstUcPositionProductModelSource.htm
- Enable the Flexibility options of the listed elements in the [4] article

Related Topics
Launching an Automation Use Case

Where to find the macro: CAAScdPstUcOverloadingOccurrenceModelSource.htm

Attention the macro can request a slight change to take into account your own Product types. First read Launching an Automation Use Case before using it.

This use case can be divided in five steps:

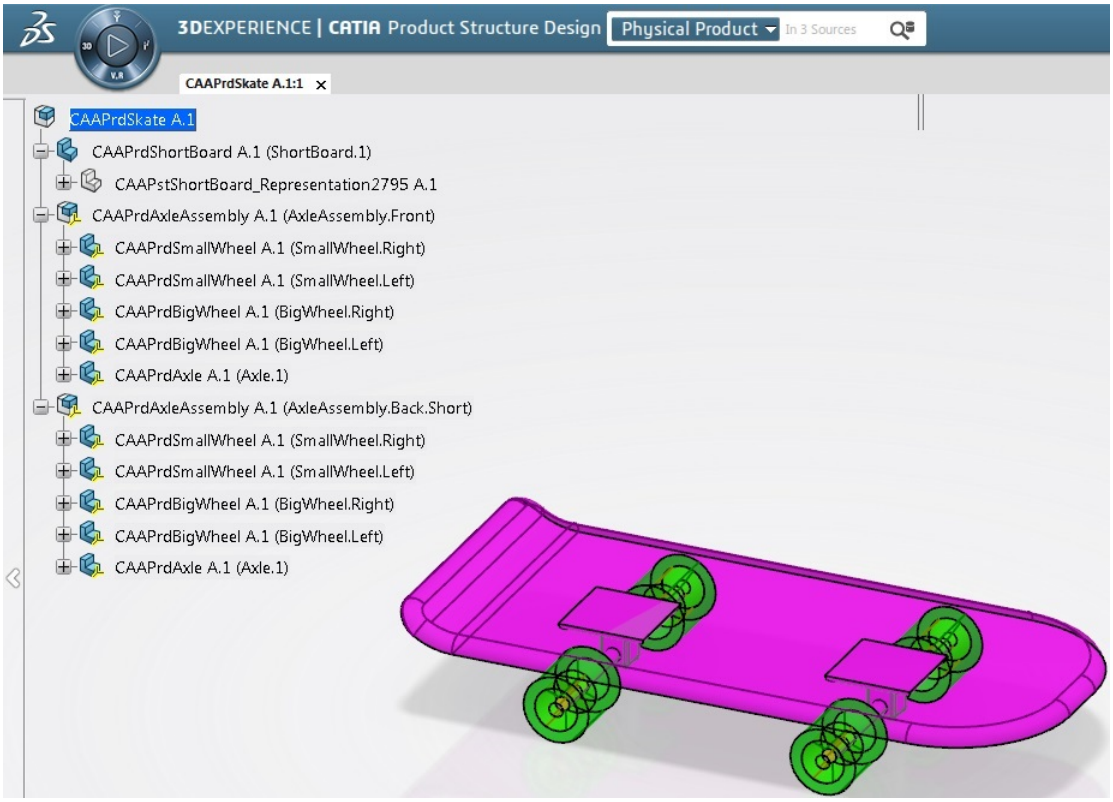
1. Retrieve the Product Reference (Skate) from underlying database
2. Overload the Skate Occurrence Model Objects

1. Retrieves the Product Reference (Skate) from underlying database

```
...
 Dim oVPMRootOccOnSkateRef As VPMRootOccurrence
 Set oVPMRootOccOnSkateRef = RetrieveProdRootOcc
...
```

The function *RetrieveProdRootOcc*, returns `oVPMRootOccOnSkateRef` , a *VPMRootOccurrence* type [2]. The PLM_ExternalID and V_version values are sought as input from an end-user. A search criteria is built with those inputs and the PLM type as a Product Reference type. When this query is run on an underlying database, it typically returns a list of entities listed within a search editor. We retrieve the first element in this list, load it in the current session ar retrieve the associated Product Reference.

Fig.1 The Positioned Skate Model



The Input model imported from database which has "CAAPrdSkate" as PLM_ExternalID of root and "A.1" version value as shown in [Fig.1]

The input data is retrieved by importing **`CAAPstBrowsingProductModel.3dxml`** file for Root product from the **`CAAProductStructure.edu`** fwk.

```
InstallRootDirectory\CAAProductStructure.edu\InputData\
```

where `InstallRootDirectory` is the directory where the CAA CD-ROM is installed.

2. Overload the skate Occurrence model objects

The objects to be overloaded from the Skate Occurrence model are as follows

- AxleAssembly.Front->BigWheel.Left (125,0,0)
- AxleAssembly.Front->BigWheel.Right (-125,0,0)

Before overloading the Occurrence model we need to enable Flexibility/Repostionable options of the objects. The process of enabling Flexibility/Repositionable detailed in the [4] article.

Next we set the relative positions of the occurrence objects mentioned above with the values mentioned in the bracket. The value are just an example to get desired object after overloading.

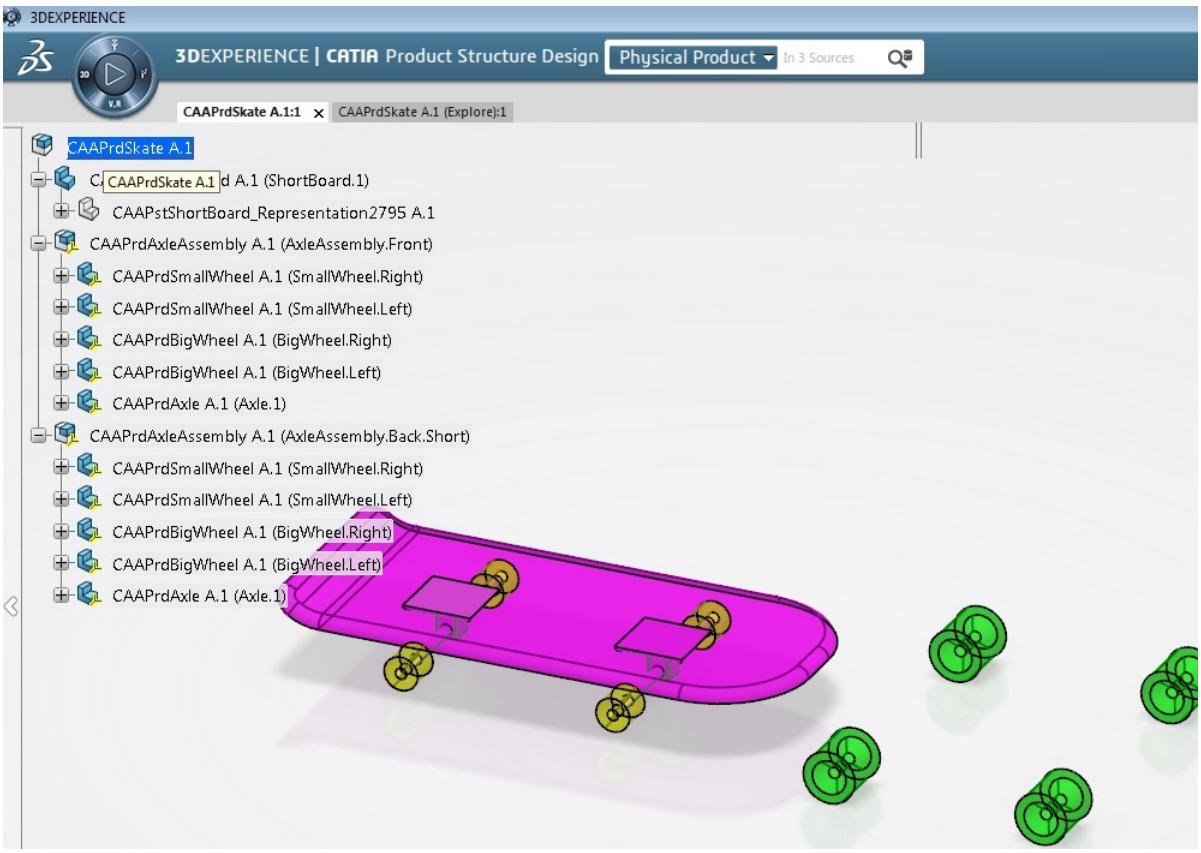
```
....
   OverloadSkateOccurrencesPosition oVPMRootOccOnSkateRef, 0
....
```

A call to *OverloadSkateOccurrencesPosition* overloads the positions of the Skate model occurrences described above. The process of setting positions explained in the [1] article.

The browsing and retrieving occurrence object are not detailed here those details are mentioned in the [3] Use Case.

After overloading the Skate model will look like as shown in [Fig.2].

Fig.2 Overloaded Skate Model



In the [Fig.2] we can observe that the two Wheel instances are moved outward from the center of the Skate model compared with front wheel instances.

References

[1] Positioning Product Model
[2] Opening Product Reference
[3] Browsing Occurrence Model
[4] Setting Flexibility/Repositionable for Skate Product Model Interactively

Creating Skate Product Model Interactively

The use case mainly deals with describing steps for creating Skate model interactively.

Before you begin: Note that:

- Launch CATIA

This scenario of creation of the Skate structure model can be divided in eight steps:

1. Create or open a Wheel reference.
2. Create or open an Axle reference.
3. Instantiate the Wheel under the Axle reference.

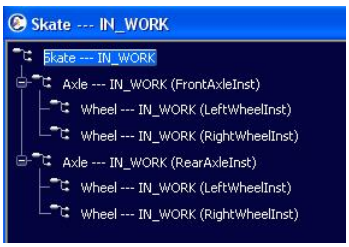
   Rename it as "LeftWheelInst".
4. Create one more instance of Wheel below the Axle reference.

   Rename it as "RightWheelInst".
5. Create or open a Skate reference.
6. Instantiate the Axle below the Skate reference

   Rename ir as "FrontAxleInst".
7. Create one more instance of Axle below the Skate reference.

   Rename it as "RearAxleInst".
8. Save into database.

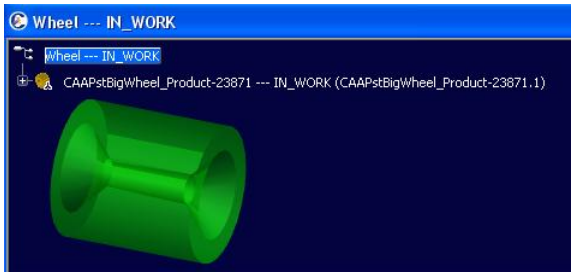
Fig.1 The Skate Model



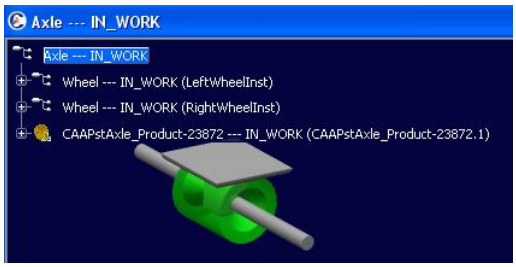
The Skate model "Skate" as PLM_ExternalID of root and "---" version value is as shown in [Fig.1].

9. Before positioning the Skate model, the next main steps to see the object with the actual 3DParts, execute the following steps. This is not mandatory but this enables you to display the positioned assembly.

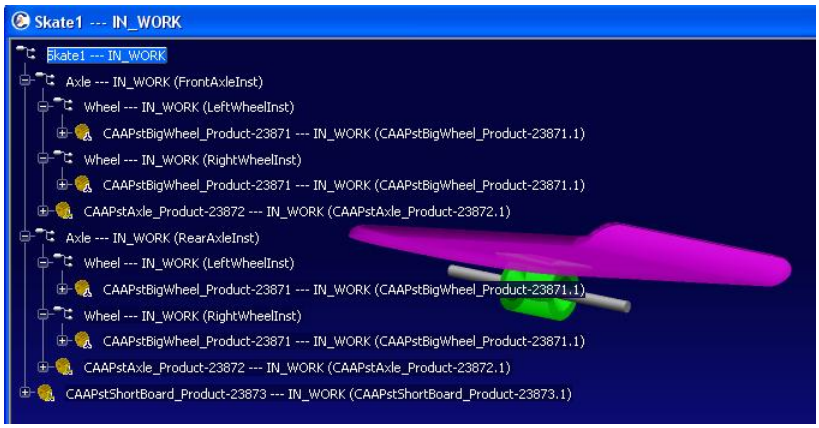
   I. Open the Wheel reference (created above), right click on it, and add the existing Wheel 3DPart.



   II. Next open the Axle reference and add the existing Skate 3DPart below it.



   III. Finally open the Skate reference and add/instantiate the existing SkateBoard 3DPart below it.



Setting Flexibility/Repositionable for Skate Product Model Interactively

The use case mainly deals with describing steps for Setting Flexibility/Repositionable for Skate Product Model Interactively.

Before you begin: Note that:

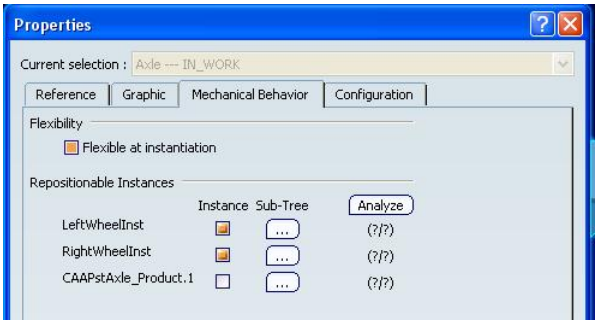
- Launch CATIA

This scenario of creation of Skate Structure Model can be divided in five steps:

1. Open the Axle assembly reference.
2. Set the flexibility/repositioning option for the following objects as shown in [Fig.1]
   - Axle Reference -> Flexibility
     - Left Wheel Inst -> Repositionable

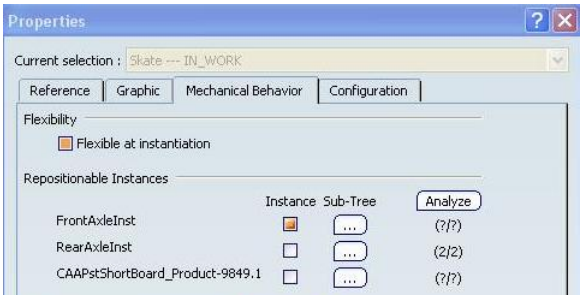
- Right Wheel Inst -> Repositionable

Fig.1 Axle Assembly Flexibility



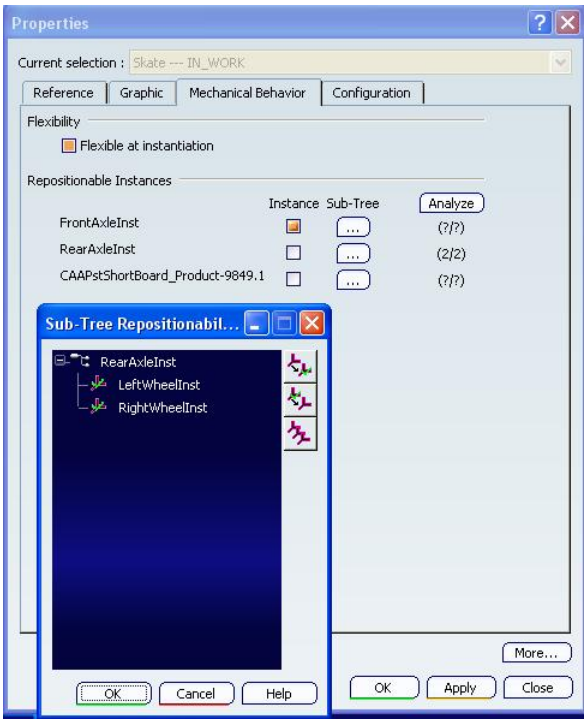
3. Open the Skate reference.
4. Set the flexibility/repositioning option for following objects as shown in [Fig.2].
   - Skate reference -> Flexibility
     - Front Axle instance -> Repositionable

Fig.2 Flexibility



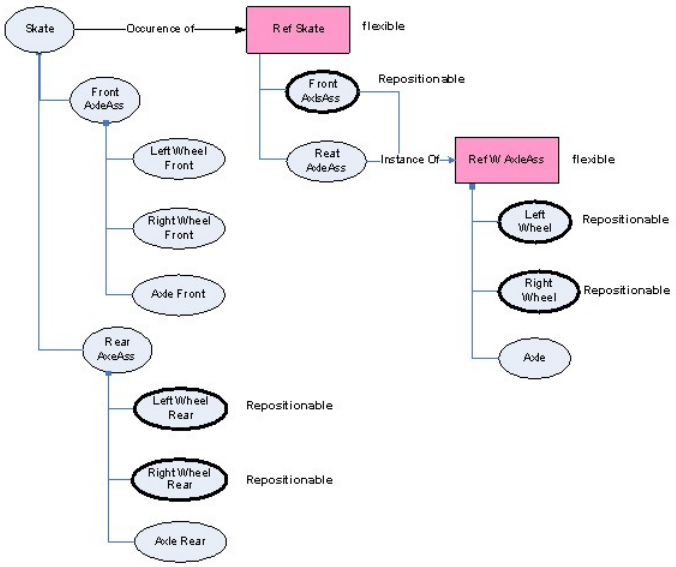
5. Next set the repositionable for the following occurrence objects.
   - RearAxleInst-LeftWheelInst -> Repositionable
   - RearAxleInst-RightWheelInst-> Repositionable

Fig.3 Setting Repositionable Occurrence Objects



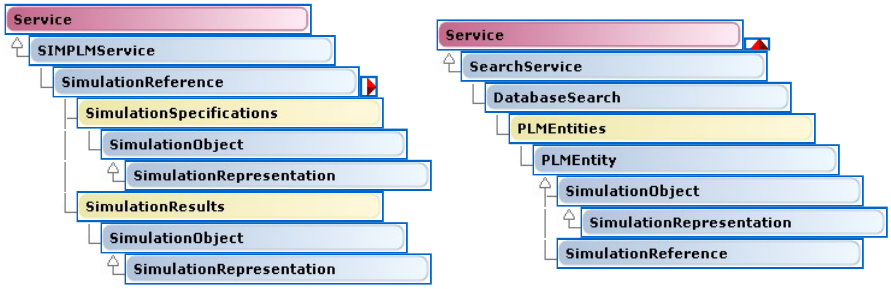
The objects to which the flexible or repositionable option as shown in [Fig.4].

Fig.4 All Elements Need to Be Set Flexibility /Repositionable



MSR Object Model Map

See Also Legend



Opening Simulation Reference

This Use case retrieves a Root Simulation Reference from database according to end user criteria. In the process Use Case demonstrates about searching the Simulation Reference from the database . Further Use Case demonstrates about retrieving handle to Root object though various methods.

Before you begin:

- You should first launch 3DEXPERIENCE and import the file `InstallRootFolder\CAADoc\CAAScdSimulation\samples\ghaSimulationMultyLab A.1.3dxml` where `InstallRootFolder` is the folder where the CAA CD-ROM is installed.
- Then, select one of the Physics Simulation apps.

Where to find the macro: CAAScdSimSearchOpenSource.htm

Attention the macro can request a slight change to take into account your own Simulation types. First read Launching an Automation Use Case before using it.

This use case can be divided in 1 step

1. Searches for a Simulation in database

Related Topics
Simulation Platform Object Model Map
Launching an Automation Use Case

1. Searches for a Simulation in database

   As a first step, the UC retrieves a Simulation from database

   It begins with a call to **SearchSimulation** function. This function searches for a list of PLM Components from the underlying database based on an input search criteria. This list is output in the PLM Search Result window in CATIA.

```
...
    Dim oDatabaseSearch As  DatabaseSearch
    Set oDatabaseSearch = SearchSimulation
...
```

   The function *SearchSimulation* returns the DatabaseSearch object, a **`oDatabaseSearch`** , which represents a tab in the PLM Search Result window in CATIA. It is interesting to note that each tab in this search corresponds to each input search criteria which user would launch on the underlying database.

   We build up the search criteria, with a Simulation Reference as PLM Entity type , along with PLM Attributes PLM_ExternalID as an input.

   The function **`SearchSimulation`** details are as in the below sub steps.

   i. Retrieves the Search service from CATIA session

```
Function SearchSimulation() As DatabaseSearch
    Dim oSearchService As SearchService
```

```
    Set oSearchService = CATIA.GetSessionService("Search")
...
```

The `PLMSearchService`, `oSearchService` is retrieved by calling *GetSessionService* method on the Application (CATIA) with "Search" as an input.

ii. Retrieving the the Databasesearch Object

```
...
   Dim oDBSearch As DatabaseSearch
   Set oDBSearch = oSearchService.DatabaseSearch
...
```

iii. Sets the type of objects to search for SIMObjSimulationObjectGeneric in this case)

```
...
    oDBSearch.BaseType = "SIMObjSimulationObjectGeneric"
...
```

A call to the `Type` property sets the type of objects to search for : "SIMObjSimulationObjectGeneric"

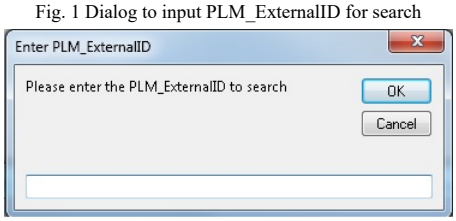
iv. Updates the PLM Search object created in the above steps with the attribute criteria provided by the user as an input

```
...
    oDBSearch.AddEasyCriteria "PLM_ExternalID", strInputPLMIDName
...
```

A call to *AddeasyCriteria* method, updates the created PLMSearch object with the search criteria according to the users input as depicted in the figures below

Prompt the user to input the PLM_ExternalID for search purpose. A dialog box appears:

Fig. 1 Dialog to input PLM_ExternalID for search



Triggers the search

```
...
    oSearchService.Search
...
End Function
```

A call to S*earch* method of the SearchService object actually searches for the objects which matches all the attributes of the set and matching the case of the values(i.e. search is Case Sensitive), and type. A Search object is equivalent to the new tab page within PLM Search window interactively.

Once the Simulation Object has been found right click on it and select the Open command in order to open it.

Navigation on a Simulation

Before you begin:

- You should first launch 3DEXPERIENCE and import the file `InstallRootFolder\CAADoc\CAAScdSimulation\samples\ghaSimulationMultyLab A.1.3dxml`

  where `InstallRootFolder` is the folder where the CAA CD-ROM is installed.
- Then, select one of the Physics Simulation apps.

Related Topics
Simulation Platform Object Model Map
Launching an Automation Use Case

Where to find the macro: CAAScdSimulationNavSource.htm

This use case can be divided in seven steps:

1. Retrieves the current Editor
2. Retrieves the Simulation Service
3. Gets the Simulation Specs Representation manager from the Editor
4. Retrieves the Simulation Reference from the Simulation Specs Representation manager
5. Retrieves the Simulated Model
6. Retrieving the Specifications Collection
7. Retrieving the Scenario Representation under the Scenarios Category
8. Retrieving the Root Feature of the Rep Scenario
9. Retrieving the Results Collection
10. Retrieving the Result Representation under the Results Category
11. Retrieving the Root Feature of the Rep Result

1. **Retrieves the current Editor**

```
...
          Dim oSimulationEditor As Editor
          Set oSimulationEditor = CATIA.ActiveEditor
...
```

2. **Retrieves the Simulation Service**

```
...
          Dim oSimulationService As SimSimulationService
```

```
            Set oSimulationService = oSimulationEditor.GetService("SimSimulationService")
...
```

3. **Gets the Simulation Specs Representation manager from the Editor**

```
...
            Dim oSimulationSpecsRepManager As SimSpecsRepManager
            Set oSimulationSpecsRepManager = oSimulationEditor.ActiveObject
...
```

4. **Retrieves the Simulation Reference from the Simulation Specs Representation manager**

```
...
             Dim oSimulationRoot As SimulationReference
             Set oSimulationRoot = oSimulationService.GetSimulationReferenceFromObject(oSimulationSpecsRepManager)
...
```

5. **Retrieves the Simulated Model**

   In this step, we retrieve the Simulated Model.

```
...
            Dim oPLMModel  As AnyObject
            Set oPLMModel = oSimulationRoot.Model
...
```

6. **Retrieving the Specifications Collection**

   In this step, we retrieve the Simulation Category Scenario.

```
...
            Dim oSpecifications  As  SimulationSpecifications
            Set oSpecifications = oSimulationRoot.Specifications
...
```

7. **Retrieving the Scenario Representation under the Scenarios Category**

```
...
             Dim oSimulationRepSpec  As SimulationRepresentation
             Set oSimulationRepSpec = oSpecifications.Item(1)
...
```

8. **Retrieving the Root Feature of the Rep Scenario**

```
...
             Dim oRootRepSpec  As AnyObject
             Set oRootRepSpec = oSimulationRepSpec.Root
...
```

9. **Retrieving the Results Collection**

```
...
            Dim oResults   As  SimulationResults
            Set oResults  = oSimulationRoot.Results
...
```

10. **Retrieving the Result Representation under the Results Category**

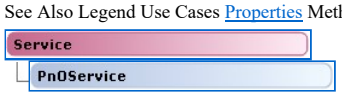
```
...
            Dim oSimulationRepRes  As SimulationRepresentation
            Set oSimulationRepRes = oResults.Item(1)
...
```

11. **Retrieving the Root Feature of the Rep Result**

```
...
             Dim oRooRepRest  As AnyObject
             Set oRootRepRes = oSimulationRepRes.Root
...
```

PnOService Object

See Also Legend Use Cases Properties Methods



Represents a service to retrieve the connected user as a Person object.

Retrieving the PnOService Object

Use the **GetSessionService** of the **Application** object to return a **PnOService** object.

```
Dim oPnOService As PnOService
Set oPnOService = CATIA.GetSessionService("PnOService")
```

Refer to the Service Object.

Using the PnOService Object

Use the **Person** property to retrieve the connected user as a Person object.

```
...
...
Dim oConnectedPerson As Person
```

```
Set oConnectedPerson = oPnOService.Person
...
```

Person Object

See Also Legend Use Cases Properties Methods

Represents the connected user.

The **Person** object lets you retrieve :

- The connected person login (**3D**EXPERIENCE login username),
- The credentials information entered in the Select Credentials panel.

Retrieving the Person Object

See PnOService Object page.

Using the Person Object

There are three information making up the 3DSpace credentials: a role, an organization and a collaborative space. You retrieve them as follows:

```
...
Dim sCollabSpace As CATBSTR
Set sCollabSpace = oPerson.CollaborativeSpaceID
...
Dim sRole As CATBSTR
Set sRole = oPerson.RoleID
...
Dim sOrganization As CATBSTR
Set sOrganization = oPerson.OrganizationID
...
```

From these three strings you can build the current security context as follow:

```
Dim sSC As CATBSTR
Set sSC =  sRole + "." + sOrganization + "." + sCollabSpace
```

The username of the person is retrieved as follows:

```
...
Dim sUserId As CATBSTR
Set sUserId = oPerson.PersonID
...
```