SpaceClaim Developers Guide

API V16 Final

March 20, 2017

Table of Contents

1 Introduction

1.1 Purpose

This document is intended to provide an overview of the SpaceClaim API and its fundamental concepts. This is the best place to start when learning about the API.

Another useful resource is the “SpaceClaim_API.chm” help file, which contains the following sections:

- **Getting Started**

This explains how to write an *add-in*, which adds new commands to SpaceClaim.

- **Examples**

This provides download links to add-in command samples written in C#, F#, C++/CLI and Visual Basic .NET. These samples show how to use the API to perform specific tasks, such as traversing assembly structure, querying the topology and geometry of a body, or creating notes.

- **API Class Library**

This is the programming documentation for the classes, structs, methods, and properties in the API.

1.2 Programming Language

Although the SpaceClaim API can be called from any .NET programming language (e.g. C#, F#, C++/CLI, Visual Basic .NET), all examples in this document are in C#.

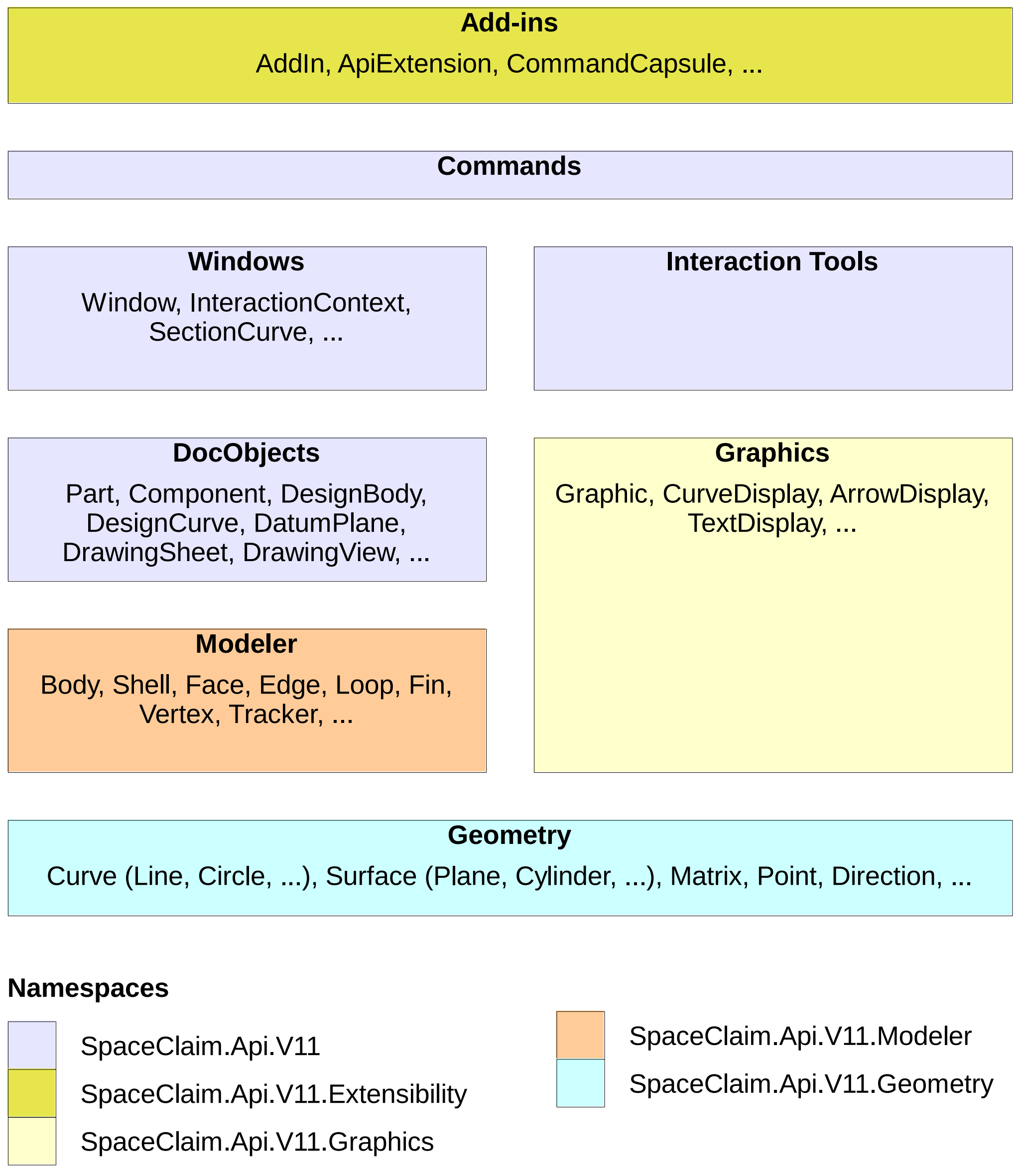
1.3 Conventions

Words in **bold** indicate types or members in the API.

2 Overview

2.1 Architecture

The diagram shows a conceptual view of the SpaceClaim architecture as presented through the API.


Add-ins
AddIn, ApiExtension, CommandCapsule, ...
Commands
Windows
Window, InteractionContext, SectionCurve, ...
Interaction Tools
DocObjects
Part, Component, DesignBody, DesignCurve, DatumPlane, DrawingSheet, DrawingView, ...
Graphics
Graphic, CurveDisplay, ArrowDisplay, TextDisplay, ...
Modeler
Body, Shell, Face, Edge, Loop, Fin, Vertex, Tracker, ...
Geometry
Curve (Line, Circle, ...), Surface (Plane, Cylinder, ...), Matrix, Point, Direction, ...
Namespaces
SpaceClaim.Api.V11
SpaceClaim.Api.V11.Extensibility
SpaceClaim.Api.V11.Graphics
SpaceClaim.Api.V11.Modeler
SpaceClaim.Api.V11.Geometry

For the purposes of this diagram, each box represents a module, where each module makes use of other modules below it in the diagram. The box sizes have no significance.

These modules are merely conceptual groupings, since the only separation apparent in the API is the separation in to namespaces, which are shown in different colors in the diagram.

Listed within many of the modules in the diagram are examples of types published by that module.

2.2 Doc Objects

Of particular importance is the distinction between *doc objects*, and the lower-level modeler and geometric objects. Doc objects, as the name suggests, belong to documents. They are first class objects, since they belong to a parent-child hierarchy, and they provide monikers (for persistent identifiers) and update states (for associative update).

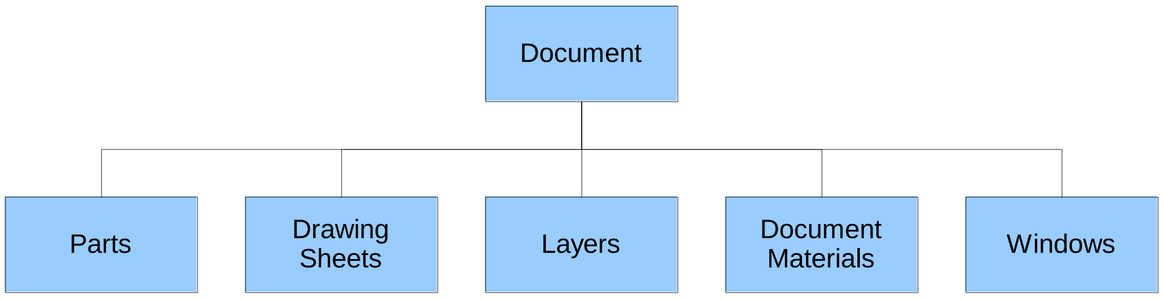
Many doc objects have references to modeler or geometry objects:

- Design bodies, design faces, and design edges are doc objects that have a reference to a corresponding modeler object: a body, face, or edge. You can create a modeler body without creating a design body, but this means no document is modified and nothing appears in the window. This can be useful if you want to perform some modeling calculation. You may or may not create a design body at the end.
- Design curves are doc objects that have a reference to a trimmed curve, which is a geometry object. Again, you can create curves and trimmed curves without ever creating a design curve, but design curves are what get displayed in windows. (You can create graphics display primitives, which reference geometry objects too, and these also get displayed in the window, although display primitives are not doc objects.)
- Datum planes are doc objects that have a reference to a plane, which is a geometry object. Again, you can create planes and other surfaces without ever creating a datum plane.

Typically the doc object will have more properties than the modeler or geometry object that it references, such as name, layer, visibility, or color.

2.3 Document Structure

A document contains the following objects:



Parts

A document always contains at least one part, known as its *main part*, and this represents the design. If the main part has internal components (instances of other parts that belong to the same document), the document will contain other parts too.

Internal components are also used for beam profiles, mid-surface parts, and sheet metal unfolded parts.

The structure of a part is described below.

Drawing Sheets

A document contains zero or more drawing sheets.

The structure of a drawing sheet is described below.

Layers

A document contains one or more layers. There is always a *default layer*, and if you delete another layer, all its objects are moved to the default layer. You cannot delete the default layer. The default layer is not the same as the *active layer*, which is the layer to which new objects are assigned. The active layer is a property of the window.

Materials

A document contains zero or more document materials, which are materials used by parts, design bodies, or beams in that document.

Windows

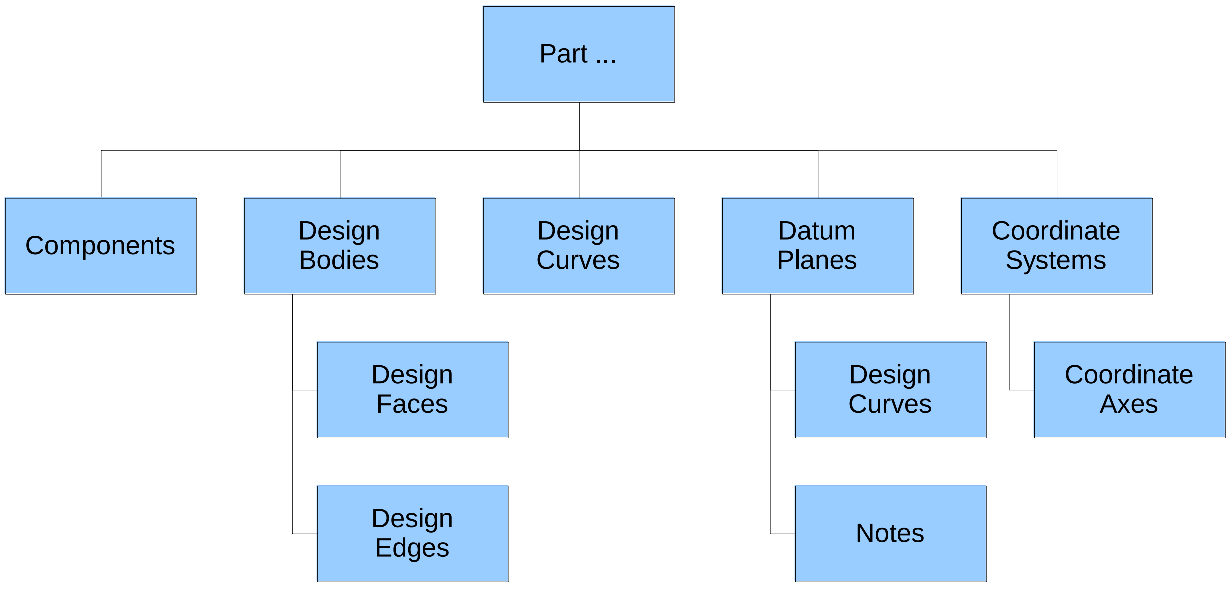
A document contains one or more windows, but it may not have any windows loaded. If the document is explicitly opened, then its windows are also loaded and opened, but if a document is loaded implicitly, for example because it is referenced from another open document, then its windows are not loaded.

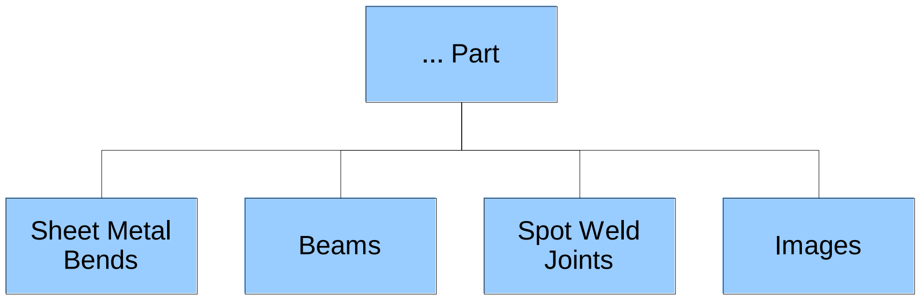
A window shows a *scene*, which is the root of the object hierarchy it displays. The window scene can be a part or a drawing sheet belonging to the same document.

The window also provides access to *interaction contexts*, which allow you to work in a specific coordinate space. The interaction context presents the current selection in that coordinate space. A useful interaction context is the *active context*, which is the context in which the user is working.

2.4 Part Structure

The SpaceClaim object model is quite flat, with the part being a bucket for many objects:





Components

A part contains zero or more components. A component is an instance of another *template* part. The template part may belong to the same document (an internal component), or it may belong to another document (an external component).

Design Bodies

A part contains zero or more design bodies. A design body can be open (a surface body) or closed (a solid body). A design body contains design faces and design edges.

Design Curves

A part contains zero or more design curves. Design curves have 3D geometry, even though they are often sketched in a plane. For example, if you copy and paste design edges, design curves are created, and these need not lie in a plane.

Design curves can also belong to datum planes and drawing sheets.

Datum Planes

A part contains zero or more datum planes. As well as serving as construction planes, as the name suggests, datum planes can also contain design curves and text notes, which lie in the plane,. When the datum plane is moved, its children are moved too.

Coordinate Systems

A part contains zero or more coordinate systems. A coordinate system contains three mutually perpendicular coordinate axes.

The world coordinate system, which can be displayed in the user interface, does not belong to any document, and is not presented through the API.

Sheet Metal Bends

If a part is a sheet metal part, then it contains zero or more sheet metal bends, which might be cylindrical or conical. If a part is a sheet metal part, then it has a sheet metal *aspect*, which is a companion object presenting sheet metal information, including bends.

Beams

A part contains zero or more beams, which have a trimmed curve path, a planar cross section, and information about the position and orientation of the cross section relative to the beam path.

Spot Weld Joints

A part contains zero or more spot weld joints. A spot weld joint has a collection of spot welds, each of which welds two or more points on design faces.

Images

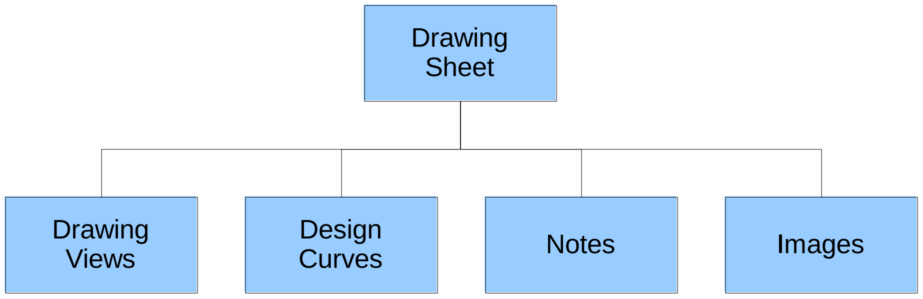
A part contains zero or more images. An image is a picture or video, either positioned in space or wrapped onto a design face. An image can also belong to a drawing sheet.

2.5 Drawing Sheet Structure

A drawing sheet contains the following doc objects:

Drawing Views

A drawing sheet contains zero or more drawing views. A drawing view is conceptually similar to a component, since it instantiates a part. In the case of a drawing view, the placement transform also contains the view scale.



Drawing views can be clipped and sectioned.

Design Curves

A drawing sheet contains zero or more design curves, which lie in the plane of the sheet. Design curves can also belong to parts and datum planes.

Notes

A drawing sheet contains zero or more text notes.

Images

A drawing sheet contains zero or more images. An image is a picture or video positioned on the drawing sheet. An image can also belong to a part.

3 Documents and Doc Objects

3.1 Overview

A **Document** is the unit of loading and saving. Assuming it has been saved, a document corresponds to a SpaceClaim scdoc file on disk.

A **DocObject** is an object that belongs to a document. Doc objects are not the only objects that get saved when the document is saved, but they are the only objects that have a **Document** property. Examples of doc objects include: **Part**, **Component**, **DesignBody**, **DesignFace**, **DatumPlane**, and **Note**.

Doc objects provide:

- A parent-child tree containment hierarchy.
- Monikers for persistent identification.
- Update states to indicate that the doc object has changed.
- Custom attributes for storing 3rd party data.

3.2 Parent-Child Hierarchy

Doc objects are part of a parent-child containment hierarchy, where the *parent* represents the container and the *children* represent the contents. If a doc object is deleted, all its *descendants* (its children, recursively) are also deleted.

For example, a **Part** contains zero or more **DesignBody** objects, each of which contains one or more **DesignFace** objects. The parent of a design face is a design body, and the parent of a design body is a part. Similarly, a design body is a child of a part, and a design face is a child of a design body.

```
Public static void Example(DesignBody desBody) {
    Part part = desBody.Parent;

    // a part is a root object, so it has no parent
    Debug.Assert(part.Parent == null);
    Debug.Assert(part.Root == part);

    // GetChildren<T> returns immediate children of type T
    foreach (DocObject child in part.GetChildren<DocObject>()) {
        // Parent returns the immediate parent
        Debug.Assert(child.Parent == part);
    }

    // DesignBody.Faces is equivalent to GetChildren<DesignFace>
    foreach (DesignFace desFace in desBody.Faces) {
        // Root returns the topmost parent
        Debug.Assert(desFace.Root == part);
    }

    // GetDescendants<T> gets all nested children of type T
    // the search starts with the object itself
    foreach (DesignFace desFace in part.GetDescendants<DesignFace>()) {
        // GetAncestor<T> crawls up the parent chain to find an object of type T
        Debug.Assert(desFace.GetAncestor<Part>() == part);

        // the search starts with the object itself
        Debug.Assert(desFace.GetAncestor<DesignFace>() == desFace);
    }
}
```

The parent chain continues up the hierarchy until the *root* object is reached. This is the topmost parent, which itself has no parent (its parent is **null**). Examples of root objects are: **Part**, **DrawingSheet**, and **Layer**.

All doc objects in the same parent-child hierarchy belong to the same document. The **Document** class provides properties to access its root objects: **MainPart**, **Parts**, **DrawingSheets**, and **Layers**.

3.3 Parts and Components

A **Part** contains zero or more **DesignBody** objects and zero or more **Component** objects. This means a part can contain both design bodies and components, in which case it is neither a pure piece part, nor a pure assembly. This is supported so that the interactive user can restructure design bodies into components, or vice versa.

A component is an *instance* of a *template* part. It has a *placement* matrix to position the component into assembly-space. The template is neither a child nor a parent of the component. If the parent-child hierarchy is visualized as an inverted tree structure with the root at the top and leaf nodes at the bottom, then the template is a sideways reference to another part, which is the root of another hierarchy.

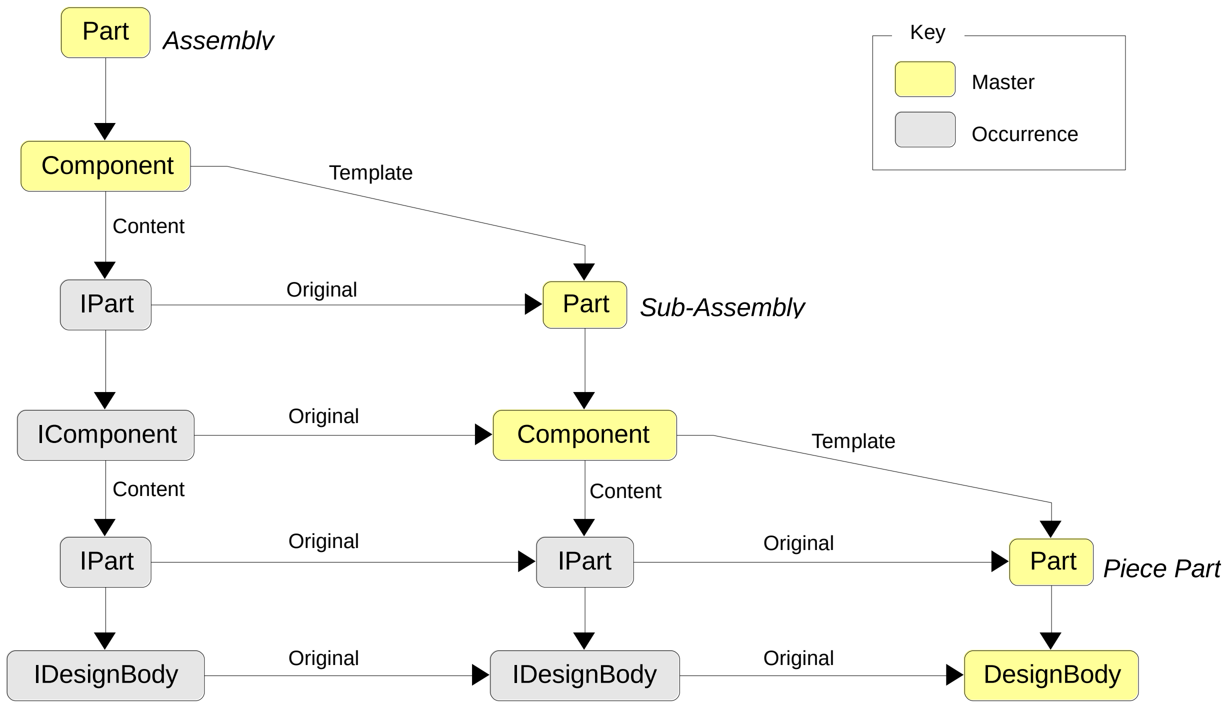
The template part may live in the same document as the component, giving rise to what the interactive user would call an *internal component*, or it may live in another document, giving rise to an *external component*. Strictly speaking, it's the template part that is either internal or external, not the component itself.

3.4 Instances and Occurrences

Drawing views are also instances of template parts, since a drawing view transforms a part into paper-space, much like a component transforms a part into assembly-space. Both **Component** and **DrawingView** inherit from **Instance**.

An **Instance** produces a transformed *occurrence* of a **Template** object, along with occurrences of its children, recursively. The template object is always a root object. The transformation is defined by the **Placement** matrix. The **Content** property provides access to the occurrence of the template object, which is is a child of the instance.

The template object is not copied. Instead, each occurrence is a lightweight wrapper, a transformed representation of its original counterpart. Geometry presented by the occurrence is in instance-space, which means assembly-space for a component, and paper-space for a drawing view. When the original object is changed, or the instance placement is changed, the occurrence changes implicitly. Any changes made to an occurrence are actually made to the original instead.



Occurrences belong to the same parent-child hierarchy as the instance that gives rise to them. The entire parent-child hierarchy is in the same coordinate-space.

3.5 Occurrence Chains

If the template object itself contains instances, occurrences of occurrences are produced. For example, this happens with a two-level assembly, since the top-level assembly contains a component that instantiates a sub-assembly part, which itself contains a component that instantiates a piece part.

3.6 General Objects and Masters

In general, a doc object is either an occurrence, or a *master*. When dealing with *general* doc objects (doc objects that might be occurrences or masters), interfaces are used, e.g. **IPart**, **IComponent**, and **IDesignBody**. These all derive from **IDocObject**.

There are some methods and properties that for theoretical or practical reasons are not presented by occurrences, and are therefore only presented by masters. When dealing with masters, classes are used, e.g. **Part**, **Component**, and **DesignBody**. These all derived from **DocObject**, which implements **IDocObject**.

Part implements **IPart**, **Component** implements **IComponent**, and so on, so a master supports everything that a general object supports, and often more besides.

Note that although **Part** is always a root object, **IPart** may or may not be a root object. If the **IPart** happens to be a master, then it is a root object, but if it happens to be an occurrence, its parent will be an **IComponent** or an **IDrawingView**.

3.7 Originals and Masters

IDocObject is the base for all doc object functionality. It provides members for traversing the parent-child hierarchy (**Children**, **GetChildren<T>**, **GetDescendants<T>**, **Parent**, **GetAncestor<T>**, **Root**), which we have already met, and it also provides members for traversing the occurrence structure:

- **Original** – Gets the original **IDocObject**, else **null** if this is a master.
- **Master** – Gets the ultimate original, i.e. the **DocObject** master at the end of the occurrence chain. If the object is a master, the object itself is returned.
- **Instance** – Gets the instance that produces this occurrence, else **null** if this is a master.

3.8 Transforming to Master-Space

The most common of these properties to use is **Master**, because the master can provide methods and properties not available on the general object. Since the master might be in a different coordinate-space to the general object, **TransformToMaster** can be used to get the transform that maps objects in general-space to objects in master-space.

```
Public static void Example(Icomponent comp, Frame placement) {
    // the Placement property is available on Component, but not Icomponent
    Component master = comp.Master;

    // map placement frame into master-space
    Matrix transToMaster = comp.TransformToMaster;
    Frame masterPlacement = transToMaster * placement;

    // apply master placement frame to master component
    master.Placement = Matrix.CreateMapping(masterPlacement);
}
```

3.9 Getting Occurrences

Having done some work in master-space, it may be necessary to obtain an object in general-space. This is common if the original object came from the window selection, and you wish to set a new selection in the window.

The window selection is always in window-space, i.e. in the same coordinate-space as the scene shown in that window (the same coordinate-system as **Window.Scene**). So if the window shows an assembly, then the selected objects are in assembly-space.

GetOccurrence can be used to obtain an object in general-space. It returns an occurrence that is similar to a companion object supplied. Note that the companion object is a general object, which may or may not be an occurrence. If it is an occurrence, then an equivalent occurrence is returned for the subject. If it is not an occurrence, then the subject itself is returned. This allows you to write code that works correctly without testing whether the object is in fact an occurrence.

```
Public static void Example() {
    Window window = Window.ActiveWindow;
    if (window == null)
        return;

    // the selected component is in window-space
    IComponent oldComp = window.SingleSelection as IComponent;
    if (oldComp == null)
        return;

    // copy the component master
    Component oldMaster = oldComp.Master;
    Component newMaster = Component.Create(oldMaster.Parent, oldMaster.Template);
    newMaster.Placement = oldMaster.Placement;

    // get an occurrence of the new master in window-space
    IComponent newComp = newMaster.GetOccurrence(oldComp);

    // select the newly created component
    window.SingleSelection = newComp;
}
```

4 Application Integration

4.1 Persistent Identifiers

Doc objects have persistent identifiers, called *monikers*, which can be recorded and later *resolved* to return the doc object again. The **Moniker** property returns a moniker for the doc object, and the **Resolve** method returns the doc object again.

Internally, master doc objects have an identifier, which is globally unique. Occurrences are identified by the instance, which is a master, along with the original object, which might itself be an occurrence. A moniker is an object that encapsulates the identifiers of the one or more master objects involved in the identity of the subject.

To record a moniker, you can record its string representation using **ToString**. The length of this string depends on the number of master objects involved. The format of this string is not documented, so you should not attempt to construct or modify such a string.

To convert the string representation back into a moniker, you can use **FromString**.

```
public static void Example(IDesignFace desFace) {
    Document doc = desFace.Document;

    // all doc objects provide a moniker
    Moniker<IDesignFace> desFaceMonikerA = desFace.Moniker;

    // resolve the moniker in the document to obtain the original object
    Debug.Assert(desFaceMonikerA.Resolve(doc) == desFace);

    // the string representation can be recorded
    string monikerText = desFaceMonikerA.ToString();

    // the moniker can be reconstructed from the string
    Moniker<IDesignFace> desFaceMonikerB = Moniker<IDesignFace>.FromString(monikerText);
    Debug.Assert(desFaceMonikerB.Resolve(doc) == desFace);
}
```

To resolve a moniker, a document must be provided as the context. SpaceClaim allows more than one version of the same scdoc file to be loaded at the same time, so the same moniker could potentially be resolved in more than one document.

Since the internal identifiers involved are globally unique, there is no danger of resolving the moniker in an unrelated document. If you attempt to do so, **null** will be returned to indicate that the object was not found.

4.2 Replacements

If the doc object has been deleted, **Resolve** returns **null** to indicate that the object was not found. Sometimes doc objects can get replaced during a command. For example, if a design face is split during a modeling operation, it will be replaced by two new design faces. Perhaps one of these new design faces gets split itself, or perhaps one of them gets deleted.

Behind the scenes, replacements are recorded, and when **Resolve** is called, if the object has been replaced, the moniker automatically returns one of the survivors, or **null** if there are no survivors.

- If it is important to know whether the object is a survivor, rather than the original object, you can compare its moniker with the moniker you already have, using the == operator. If the object is a survivor, it will have a different moniker.
- If it is important to obtain all survivors, **ResolveSurvivors** can be used instead. Note that this method only returns surviving replacements, so if the object hasn't been replaced at all, no survivors are returned.

4.3 Update States

A doc object master has an *update state*, which tells you if the object has changed.

Each time the doc object master is changed, its update state changes. Conversely, if the update state has not changed, then the object has not changed. When an object is changed due to an undo (or redo) operation, its update state is undone (or redone) too.

The update state is persistent, so you can store its string representation and use it in another SpaceClaim session.

```
public static void Example(DesignFace desFace) {
    // doc object masters provide an update state
    UpdateState beforeStateA = desFace.UpdateState;

    // the string representation can be recorded
    string stateText = beforeStateA.ToString();

    ModifyDesignBody();

    // test whether the design face was changed
    if (desFace.UpdateState != beforeStateA)
        Debug.WriteLine("Design face was changed.");

    // the update state can be reconstructed from the string
    UpdateState beforeStateB = UpdateState.FromString(stateText);
    Debug.Assert(beforeStateA == beforeStateB);
}
```

Update states are not provided for occurrences, but you can store the update states of the instances involved in the occurrence chain, along with the update state of the master. **PathToMaster** returns these instances.

The update state only represents the state of the object itself, and not the state of other objects it contains or references. For example, the update state of a part is not changed if a child design body is modified. Similarly, although the update state of a component will change if its placement is modified (since this is a property of the component itself), the update state will not change if its template part is modified.

5 Storing Custom Data

5.1 Document Properties

Documents have two types of properties:

- *Core properties* cover standard fields such as *description*, *subject*, *title*, and *creator*. The set of core properties is fixed. You cannot create new core properties.
- *Custom properties* allow 3rd party applications to store data with the document. Each custom property is a name-value pair.

So that custom property names do not clash when different applications choose a custom property name, the name should be prefixed with the application or add-in name, as in the following example:

```
public static void Example(Document doc) {
    CustomProperty.Create(doc, "BananaWorks.ApplicationVersion", 14);

    CustomProperty property;
    if (doc.CustomProperties.TryGetValue("BananaWorks.ApplicationVersion", out property))
        Debug.Assert((double) property.Value == 14);
}
```

Note that a document can contain more than one part, so if you want to store data for a part, this is best done by storing a custom attribute on the part master (see next topic).

5.2 Custom Attributes

Doc object masters provide custom attributes so that 3rd party applications can store data. Two types of attribute are provided: text attributes and number attributes. An attribute is a name-value pair. A doc object can have a text attribute and a number attribute with the same name.

So that attribute names do not clash when different applications choose an attribute name, the name should be prefixed with the application or add-in name, as in the following example:

```
public static void Example(DesignBody desBody) {
    desBody.SetTextAttribute("BananaWorks.SkinCondition", "Ripe");

    string skinType;
    if (desBody.TryGetTextAttribute("BananaWorks.SkinCondition", out skinType))
        Debug.Assert(skinType == "Ripe");
}
```

Multiple values can be stored as multiple attributes with distinct names, or they can be formatted into a single text string using **String.Format** or an XML serializer.

5.3 Attribute Propagation

Attributes applied to doc object masters are propagated if the object is replaced. For example, if a design face has a text attribute, and this face is split during a modeling operation, the replacement design face fragments will also carry the same text attribute.

6 Identifying Objects in ACIS and Parasolid Files

6.1 Identifiers During Export

When an ACIS or Parasolid file is written, either by the user or by calling **Part.Export**, *name* attributes are attached to face and edge entities in the file to indicate which design face or design edge master they came from. This is useful if the model is changed and then a new file is exported, since corresponding faces and edges will have the same *name* attributes.

- An ACIS *name* attribute is a "named attribute" (ATTRIB_GEN_NAME) with the attribute name, "ATTRIB_XACIS_NAME".
- A Parasolid *name* attribute is a system attribute with the name, "SDL/TYSA_NAME".

Design face and design edge masters have an **ExportIdentifier** property, which returns a string containing the value of the name attribute that is written when the object is exported.

6.2 Foreign Identifiers During Import and Export

There may be a requirement to import a model from another system, modify it in SpaceClaim, and then export it again, such that the other system can track the identity of faces and edges during this round trip.

When importing an ACIS or Parasolid file, if any body, face, or edge entities have *id* attributes, these are converted to SpaceClaim text attributes on the resulting design body, design face, or design edge masters. These text attributes have the reserved name, "@id".

- An ACIS *id* attribute is a "named attribute" (ATTRIB_GEN_NAME) with the attribute name, "ATTRIB_XACIS_ID".
- A Parasolid *id* attribute has the name, "ATTRIB_XPARASOLID_ID", and has an attribute definition, which is described in the documentation for the **Part.Export** method.

Attributes applied to doc object masters are propagated if the object is replaced. For example, if a design face has a text attribute, and this face is split during a modeling operation, the replacement design face fragments will also carry the same text attribute.

When exporting an ACIS or Parasolid file, if any design bodies, design faces, or design edges have text attributes with the name, "@id", these are written as *id* attributes applied to resulting ACIS or Parasolid entities.

7 Geometry and Topology

7.1 Unbounded Geometry and Trimmed Objects

In SpaceClaim, geometry is conceptually unbounded, e.g. a **Plane** extends indefinitely, a **Sphere** is complete, and a **Line** is infinite in length. On top of this there are *trimmed* objects, which are bounded and therefore provide additional properties:

- **ITrimmedCurve** – a bounded curve. It provides **Length** and parametric **Bounds**.
- **ITrimmedSurface** – a bounded surface. It provides **Area**, **Perimeter**, and a parametric **BoxUV**.
- **ITrimmedSpace** – a bounded region of 3D space. It provides **Volume** and **Surface Area**.

All of these inherit from **IBounded**, which provides **GetBoundingBox**.

A trimmed curve has a **Geometry** property that returns the **Curve**, and a trimmed surface has a **Geometry** property that returns the **Surface**. A trimmed region even has a **Geometry** property that returns a **Space** object representing all of Cartesian 3D space.

Trimmed curves and trimmed surfaces also have an **IsReversed** property, which tells you whether the sense of the object is the opposite of the sense of its geometry. The sense of a trimmed curve is its direction, and the sense of a trimmed surface is which way its normals face.

7.2 Topology

The topology of a model is made of **Body**, **Face**, and **Edge** objects, along with other objects (shells, loops, fins, and vertices) that describe in more detail how they connect up.

- **Body** inherits from **ITrimmedSpace**. It also provides **Faces** and **Edges**.
- **Face** inherits from **ITrimmedSurface**. It also provides surrounding **Edges**.
- **Edge** inherits from **ITrimmedCurve**. It also provides adjacent **Faces**.

Topology classes have more information than the trimmed object interfaces that they implement:

- Trimmed object interfaces have no concept of connectivity.
- Although they can return area and volume, respectively, **ITrimmedSurface** and **ITrimmedSpace** say nothing about the shape of their boundary. (With **ITrimmedCurve**, the boundary has no shape as such, since the curve is simply bounded by parameter values.)

7.3 Doc Objects and Geometry

Topology objects (**Body**, **Face**, **Edge**, etc.) and geometry objects (**Plane**, **Cylinder**, **Line**, etc.) are not doc objects. They are not part of a parent-child hierarchy, and they do not have monikers or update states. They are lower level objects, since they might be referenced by a doc object, but they have no knowledge of documents and doc objects themselves.

You can construct geometry, trimmed objects, and even solid bodies, in order to perform geometric calculations, without modifying a document:

```
public static void Example() {
    // create infinite line with zero parameter at the origin
    Line line = Line.Create(Point.Origin, Direction.DirX);

    // create line segment from (-0.01, 0, 0) to (0.01, 0, 0)
    ITrimmedCurve trimmedCurve = CurveSegment.Create(line, Interval.Create(-0.01, 0.01));
    Debug.Assert(Accuracy.EqualLengths(trimmedCurve.Length, 0.02));

    // find closest point to (0.05, 0.05, 0)
    CurveEvaluation eval = trimmedCurve.ProjectPoint(Point.Create(0.05, 0.05, 0));

    // closest point on line segment should be (0.01, 0, 0)
    Debug.Assert(eval.Point == Point.Create(0.01, 0, 0));
}
```

The document is modified when you create doc objects. For example, you might create a design curve from a trimmed curve.

7.4 Design Curves

Design curves are what the end user refers to as *sketch curves*. They are called *design curves* in the API for consistency with design bodies, design faces, and design edges.

A **DesignCurve** is a doc object, which has a trimmed curve **Shape**.

```
public static void Example(Part part) {
    Line line = Line.Create(Point.Origin, Direction.DirX);
    ITrimmedCurve shape = CurveSegment.Create(line, Interval.Create(-0.01, 0.01));

    // create a design curve
    DesignCurve desCurve = DesignCurve.Create(part, Plane.PlaneXY, shape);

    // the Shape property returns the trimmed curve
    Debug.Assert(Accuracy.EqualLengths(desCurve.Shape.Length, 0.02));

    // override the layer color
    desCurve.SetColor(null, Color.DarkSalmon);
}
```

A design curve has other properties that are outside the concept of a trimmed curve, such as layer, color override, and name.

7.5 Design Bodies

Just as a **DesignCurve** is a doc object, which has a **Shape** of type **ITrimmedCurve**, there is a similar pattern for bodies:

- A **DesignBody** has a **Shape** of type **Body**.
- A **DesignFace** has a **Shape** of type **Face**.
- A **DesignEdge** has a **Shape** of type **Edge**.

This is only true for these doc object masters. For the corresponding general objects, less information is available:

- **IDesignBody** has a **Shape** of type **ITrimmedSpace**.
- **IDesignFace** has a **Shape** of type **ITrimmedSurface**.
- **IDesignEdge** has a **Shape** of type **ITrimmedCurve**.

This means, you can only access detailed topology information, such as loops and fins, from masters:

```
public static void Example(IDesignFace desFace) {
    // we can get the area from the ITrimmedSurface shape
    double area = desFace.Shape.Area;

    // but to access loops, we need to use the master
    DesignFace desFaceMaster = desFace.Master;
    // if we access geometry, remember we are now in master-space
    Matrix transToMaster = desFace.TransformToMaster;

    DesignBody desBodyMaster = desFaceMaster.Parent;

    // the master Shape is a Face rather than an ITrimmedSurface
    Face face = desFaceMaster.Shape;
    foreach (Loop loop in face.Loops)
        foreach (Fin fin in loop.Fins) {
            Edge edge = fin.Edge;

            // get from shape back to doc object master
            DesignEdge desEdgeMaster = desBodyMaster.GetDesignEdge(edge);

            // from master to occurrence equivalent to desFace
            IDesignEdge desEdge = desEdgeMaster.GetOccurrence(desFace);
        }
}
```

However, the general interfaces do provide some convenient connectivity traversals at the doc object level:

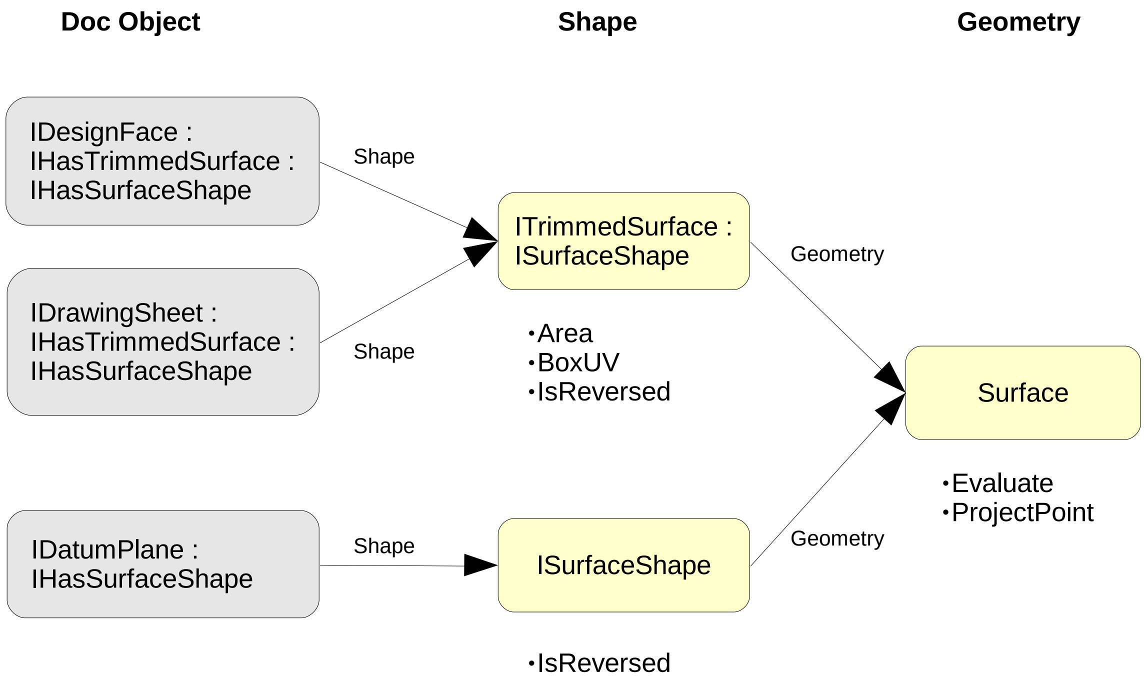
```
public static void Example(IDesignFace desFace) {
    IDesignBody desBody = desFace.Parent;

    // the Edge property returns the edges in the face boundary
    foreach (IDesignEdge desEdge in desFace.Edges) {
        Debug.Assert(desEdge.Parent == desBody);

        // the Faces property returns the faces that meet at this edge
        Debug.Assert(desEdge.Faces.Contains(desFace));
    }
}
```

7.6 Shape in General

Shape also applies to doc objects that have untrimmed geometry, e.g. a datum plane. **DatumPlane** implements **IHasSurfaceShape** and its **Shape**, not surprisingly, is a **ISurfaceShape**. Compare this to **DesignFace**, which implements **IHasTrimmedSurface** and has a **Shape** of type **ITrimmedSurface**. This parallel path exists because **IHasTrimmedSurface** is derived form **IHasSurfaceShape**, and **ITrimmedSurface** is derived from **ISurfaceShape**. Therefore, whether the geometry is untrimmed or trimmed, there is always a two step traversal, first to **Shape**, and then to **Geometry**:


Doc Object
Shape
Geometry
IDesignFace : IHasTrimmedSurface : IHasSurfaceShape
IDrawingSheet : IHasTrimmedSurface : IHasSurfaceShape
IDatumPlane : IHasSurfaceShape
Shape
Shape
Shape
ITrimmedSurface : ISurfaceShape
•Area
•BoxUV
•IsReversed
ISurfaceShape
•IsReversed
Geometry
Geometry
Surface
•Evaluate
•ProjectPoint

8 Accuracy

8.1 Linear and Angular Resolution

Internally, geometric calculations are performed to machine double precision. When comparing the results of calculations, values should always be compared to within a specific resolution. These resolutions are provided by the **Accuracy** class:

- Two lengths are considered equal if their difference is less than **LinearResolution**.
- Two angles are considered equal if their difference is less than **AngularResolution**.

For example, when **ContainsPoint** is called to determine whether a point lies in a surface, internally the distance from the surface might be calculated, and the result is **true** if this distance is less than **LinearResolution**.

8.2 Comparing Lengths and Angles

The **Accuracy** class provides methods for comparing lengths and angles.

The **CompareLengths** method takes two arguments, *lengthA* and *lengthB*, and returns an integer result:

- -1 *lengthA* is less than *lengthB*.
- 0 *lengthA* is equal to *lengthB* to within linear resolution.
- +1 *lengthA* is greater than *lengthB*.

This method provides general comparison of two lengths, but for common situations, such as comparing with zero, or testing whether two values are equal, simpler and more readable methods can be used:

```
public static void Example(double lengthA, double lengthB) {
    // same as CompareLengths(lengthA, lengthB) == 0
    bool equalLengths = Accuracy.EqualLengths(lengthA, lengthB);

    // same as CompareLengths(lengthA, 0) == 0
    bool lengthIsZero = Accuracy.LengthIsZero(lengthA);

    // same as CompareLengths(lengthA, 0) > 0
    bool lengthIsPositive = Accuracy.LengthIsPositive(lengthA);

    // same as CompareLengths(lengthA, 0) < 0
    bool lengthIsNegative = Accuracy.LengthIsNegative(lengthA);
}
```

Corresponding methods are provided for angles: **CompareAngles**, **EqualAngles**, **AngleIsZero**, **AngleIsPositive**, and **AngleIsNegative**.

8.3 Comparing XYZ Objects

The basic XYZ types, **Point**, **Vector**, **Direction**, **Box**, and **Frame**, have resolution tests build in, so you can compare objects using the == operator. For example, two points are equal if the distance between them is less than the linear resolution, and two directions are equal if the angle between them is less than then angular resolution.

```
public static void Example(Plane plane, Point point) {
    // project point onto plane
    SurfaceEvaluation eval = plane.ProjectPoint(point);
    Point pointOnPlane = eval.Point;

    // points are the same if less than linear resolution apart
    bool planeContainsPoint = point == pointOnPlane;

    // ContainsPoint is more efficient, but gives the same result
    Debug.Assert(planeContainsPoint == plane.ContainsPoint(point));
}
```

8.4 Comparing UV Objects

The same is not true for the surface parameter UV types, **PointUV**, **VectorUV**, **DirectionUV**, and **BoxUV**, or for the curve parameter type, **Interval**. These types do not know whether the parameterization they represent is linear, angular, or some other type. For example, for a plane, the U parameterization is linear, but for a cylinder, the U parameterization is angular. For a NURBS surface, the U parameterization is neither linear nor angular.

Therefore, you should not use the == operator for these types. When comparing parameters, you should use the appropriate length or angle comparison method for each of the U and V values. For NURBS parameterization, angular comparison could be used, but it is safest to evaluate points and compare these instead.

8.5 Comparing Geometry

To say that two surfaces or two curves are equal is ambiguous, since there is more than one interpretation of the concept. For example, with two planes:

- They could be coplanar, but the normals could be opposed.
- They could be coplanar, with normals having the same sense, but their frames and hence their parameterization might be different.
- They might be identical in every way.

You should not use the == operator with surface and curve types, since that will only test for reference equality.

Geometry.IsCoincident and **ITrimmedCurve.IsCoincident** are provided to make comparisons. They only test for coincidence, which means any of the above cases would pass.

9 Units

9.1 System Units and User Units

Internally SpaceClaim works in SI units: meters, kilograms, and seconds. The API also works in SI units.

The user may be working in some other units, but internally the units are still SI units. Conversions are done when values are presented to the user, or input by the user.

The **Window** class provides conversions between system units (SI units) and user units.

9.2 Outputting Values

Window.Units.Length.Format produces a length string that can be presented to the user, which includes the units symbol. As well as performing units conversion, this method also formats the output according to whether the user is working in decimals or fractions.

Window.Units.Angle.Format provides the same functionality for angles.

9.3 Inputting Values

To parse a length string entered by the user, you can use **Window.Units.Length.TryParse**. As well as converting to system units, this method also handles expressions, and values with explicit units stated:

- "(1 + 2) * 3 ^ (3/3 + 3)" = 243
- "1cm + 1 1/2 mm" = 0.0115

Window.Units.Angle.TryParse provides the same functionality for angles.

9.4 Custom Conversions

If you need more control over the formatting, you can use **Window.Units.Length.ConversionFactor** and **Window.Units.Length.Symbol** for lengths, or **Window.Units.Angle.ConversionFactor** and **Window.Units.Angle.Symbol** for angles.

10 Calling The API From Another Process

10.1 .NET Remoting

The SpaceClaim API uses .NET Remoting, which allows calls to be made from a different application domain, or from a different process. Currently, only different processes on the same computer are supported.

SpaceClaim add-ins are typically run in a separate application domain in the SpaceClaim process, although this is controlled by the **host** attribute in the add-in manifest file. When integrating SpaceClaim with another application, it may be more convenient to create a add-in for the other application instead, which calls the SpaceClaim API from the other application's process.

10.2 Template Client Code

Here is some template code for an API client in another process. In this example, the client is a simple console application.

```
static void Main(string[] args) {
    // Initialize must be called before using the API
    Api.Initialize();

    Session session = GetSpaceClaimSession();
    bool startSpaceClaim = session == null;
    if (startSpaceClaim) {
        session = Session.Start(60000); // 60 secs timeout
        if (session == null) {
            Debug.Fail("Failed to start SpaceClaim.");
            return;
        }
    }

    // attach to SpaceClaim session before making other calls
    Api.AttachToSession(session);

    // if the user exits SpaceClaim, exit this client app too
    Process process = Process.GetProcessById(session.ProcessId);
    process.EnableRaisingEvents = true;
    process.Exited += delegate {
        Debug.WriteLine("User exited SpaceClaim.");
        Environment.Exit(0);
    };

    ExecuteApiCode();

    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();

    // if we started SpaceClaim, we shall stop it
    if (startSpaceClaim)
        session.Stop();
}
```

```
static Session GetSpaceClaimSession() {
    foreach (Session session in Session.GetSessions())
        return session;
    return null;
}
```

Before the API can be used, **Api.Initialize** must be called. For add-ins, this is done automatically in the **AddInBase** constructor, but for standalone client applications in a separate process, **Api.Initialize** must be called explicitly.

Then **Api.AttachToServer** must be called. Again, for add-ins, this is done automatically in the **AddInBase** constructor. Once the API is attached to a server, it cannot be detached or re-attached. Instead, separate application domains can be used to attach to separate servers, and this even allows connection to more than one server at once.

<end>