Creo Object TOOLKIT C++
User's Guide
9.0.8.0

PTC Inc., 121 Seaport Blvd, Boston, MA 02210 USA

Contents

About This Guide

This section contains information about the contents of this user's guide and the conventions used.

Purpose

This manual describes how to use Creo Object TOOLKIT C++, an object-based C ++ toolkit API for Creo Parametric. Creo Object TOOLKIT C++ makes possible the development of C++ programs that access the internal components of a Creo session, to customize Creo models.

Note

Creo Object TOOLKIT C++ is supported with Creo Parametric. It is not supported with the other Creo applications.

Audience

This manual is intended for experienced Creo Parametric users who are already familiar with object-oriented language.

Prerequisites

This manual assumes you have the following knowledge:

- Creo Parametric
- The syntax and language structure of C++.

Documentation

The documentation for Creo Object TOOLKIT C++ includes the following:

- *Creo Object TOOLKIT C++ User's Guide*
- An online browser that describes the syntax of the Creo Object TOOLKIT C++ methods and provides a link to the online version of this manual. The online version of the documentation is updated more frequently than the printed version. If there are any discrepancies, the online version is the correct one.

Conventions

The following table lists conventions and terms used throughout this book.

Convention	Description
UPPERCASE	Creo—type menu name (for example, PART).
Boldface	Windows-type menu name or menu or dialog box option (for example, **View**), or utility. **Boldface** font is also used for keywords,Creo Object TOOLKIT C++ methods, names of dialog box buttons, and Creo commands.
`Monospace(Courier)`	Code samples appear `in courier font like this`. C++ aspects (methods, classes, data types, object names, and so on) also appear in Courier font.
Emphasis	Important information appears *in italics like this*. Italic font is also used for file names and uniform resource locators (URLs).
Choose	Highlight a menu option by placing the arrow cursor on the option and pressing the left mouse button.
Select	A synonym for "choose" as above, Select also describes the actions of selecting elements on a model and checking boxes.
Element	An element describes redefinable characteristics of a feature in a model.
Mode	An environment in Creo in which you can perform a group of closely related functions (Drawing, for example).
Model	An assembly, part, drawing, format, notebook, case study, sketch, and so on.
Option	An item in a menu or an entry in a configuration file or a setup file.
Solid	A part or an assembly.
`<creo_loadpoint>`	The location where the Creo applications are installed, for example, `C:\Program Files\PTC\Creo 1.0`.
`<creo_otk_loadpoint_app>`	The location where the Creo Object TOOLKIT C++ application files are installed, that is, `<creo_loadpoint>\<datecode>\Common Files\otk\otk_cpp`.

Convention	Description
<creo_otk_loadpoint_doc>	The location where the Creo Object TOOLKIT C++ documentation files are installed, that is, <creo_loadpoint>\<datecode>\Common Files\otk_cpp_doc.
<creo_toolkit_loadpoint>	The location where the Creo TOOLKIT application is installed, that is, <creo_loadpoint>\<datecode>\Common Files\protoolkit.

Note

- Important information that should not be overlooked appears in notes like this.
- All references to mouse clicks assume use of a right-handed mouse.

Software Product Concerns and Documentation Comments

For resources and services to help you with PTC software products, see the *PTC Customer Service Guide*. It includes instructions for using the World Wide Web or fax transmissions for customer support.

In regard to documentation, PTC welcomes your suggestions and comments. You can send feedback in the following ways:

- Send comments electronically to *MCAD-documentation@ptc.com*.
- Fill out and mail the PTC Documentation Survey in the customer service guide.

1

Overview of Creo Object TOOLKIT C++

This chapter provides an overview of Creo Object TOOLKIT C++.

Setting Up Creo Object TOOLKIT C++

Creo Object TOOLKIT C++ is a Creo object-based C++ toolkit API. It can be used with Creo TOOLKIT inside the same C++ based application.

Installing Creo Object TOOLKIT C++

Creo Object TOOLKIT C++ is a part of the Creo installation, and is installed automatically, if selected.

The following directories are specific to the Creo Object TOOLKIT C++ installation:

- `<creo_otk_loadpoint_app>\include`—Contains all the header files specific to Creo Object TOOLKIT C++.
- `<creo_otk_loadpoint_app>\<platform>\obj`—Contains platform-specific libraries of Creo Object TOOLKIT C++ which must be used with the Creo TOOLKIT libraries.

Building a Sample Application

The Creo Object TOOLKIT C++ loadpoint contains sample applications and makefiles. The makefiles show how to build the application on supported platforms.

The sample applications in `otk_install.zip` demonstrate how to build an application in the Microsoft Visual Studio 2019 environment without using makefiles. `otk_install.zip` is located in `<creo_otk_loadpoint_app>/<machine_type>/obj` directory.

Note

otk

In Creo ParametricCreo Parametric 7.0.1.0 and later, Creo Object TOOLKIT C++ supports Visual Studio 2019. The compiler flags and libraries are available for Visual Studio 2019. Creo Object TOOLKIT C++ no longer supports Visual Studio 2015.

All Creo Object TOOLKIT C++ applications on 64-bit Windows platforms built using the Microsoft Visual Studio 2019 compiler must set the configuration property Platform Toolset as Visual Studio 2019 (v142).

The steps required to build and run the test applications are described in the subsequent sections.

Creo Object TOOLKIT C++ based applications use the same format of `creotk.dat` file as the Creo TOOLKIT based applications. Refer to the chapter Creo Object TOOLKIT C++ Registry File on page 652, for more information.

The `creotk.dat` file should contain the following lines:

```
NAME          YourApplicationName
TOOLKIT             OBJECT [optional; omitted means protoolkit]
CREO_TYPE           [optional; omitted means PARAMETRIC]
EXEC_FILE           $LOADDIR/$MACHINE_TYPE/obj/filename.dll
TEXT_DIR            $LOADDIR
STARTUP             dll [dll/spawn/java]
END
```

Licensing Options for Creo Object TOOLKIT C++

From Creo Object TOOLKIT C++ 4.0 F000, advanced licensing is supported. To develop and test Creo Object TOOLKIT C++ applications, you require the following licenses:

- `ObjectToolkitCpp`—Specifies the basic Creo Object TOOLKIT C++ development license.
- `TOOLKIT-for-3D_Drawings`—Specifies the advanced license. Certain methods are available under the 222 license option. Methods that require license 222 have the comment “LICENSE: 222” added in the APIWizard.

From Creo Object TOOLKIT C++ 4.0 F000 onward, the following new libraries have been added. It is mandatory that you include one of the libraries in your application.

- `otk_222.lib`—If you have the advanced license option 222, that is, the `TOOLKIT-for-3D_Drawings` license, include this library in your application.
- `otk_no222.lib`—If you do not have the advanced license option, include this library in your application.

Note

From Creo Object TOOLKIT C++ 4.0 F000 onward, all your existing and new applications must include the library `otk_222.lib` or `otk_no222.lib`.

To check if you have the Creo Object TOOLKIT C++ license option, run the `Ptcstatus` utility from the Creo load point. For example, check for the license option `ObjectToolkitCpp`.

You can check if you have the Creo Object TOOLKIT C++ license option in the Creo user interface. Click **File ▸ Help ▸ System Information**. In the **INFORMATION WINDOW**, under **Configured Option Modules** check for the license option `ObjectToolkitCpp`. In case the license is not available please contact your system administrator.

The Creo Object TOOLKIT C++ applications must be unlocked before distributing it to the end users. To unlock the application, both the Creo Object TOOLKIT C++ and Creo TOOLKIT licenses are required.

Unlocking the Creo Object TOOLKIT C++ Application

Before you distribute your application executable to the end user, you must unlock it. This enables the end user to run your applications without having Creo Object TOOLKIT C++ as an option.

To unlock a Creo Object TOOLKIT C++ application, you must have both the Creo Object TOOLKIT C++ license and the Creo TOOLKIT license.

Note

The Creo Object TOOLKIT C++ license is now a part of the Creo TOOLKIT license pack. If you are unable to unlock the Creo Object TOOLKIT C++ applications, request for the updated license pack from the Technical Support page.

In Creo 6.0.0.0 and later you can digitally sign your Creo Object TOOLKIT C++ application.

To unlock your application, enter the following command:

```
     <creo_loadpoint>/<app_name>/bin/protk_unlock.bat [-cxx]
[-cxxd] <path
 to executables or DLLs to unlock>
```

where, `<app_name>`—`Parametric` for Creo Parametric.

Note

- The Creo Object TOOLKIT C++ is unlocked even if you do not specify the `-cxx` option.
- To unlock and digitally sign your application, specify the `-cxxd` option. Note that it is mandatory to sign your application if you use the `-cxxd` option. See the section Digitally Signing the Application on page , for more information on digitally signing your application.

You can provide more than one Creo TOOLKIT binary file on the command line.

Note

Once you have unlocked the executable, you can distribute your application program to Creo Object TOOLKIT C++ users in accordance with the license agreement.

Using `protk_unlock.bat` requires a valid Creo TOOLKIT license to be present and unused on your license server. If the Creo Parametric license server is configured to add a Creo TOOLKIT license as a startup option, `protk_unlock.bat` will cause the license server to hold only the Creo TOOLKIT option for 15 minutes. The license will not be available for any other development activity or unlocking during this period.

If you use `-cd` option to unlock your application, a message appears asking you to digitally sign your application before using it in Creo.

If the only available Creo TOOLKIT license is locked to a Creo Parametric license, the entire Creo Parametric license including the Creo TOOLKIT option will be held for 15 minutes. PTC recommends you configure your Creo TOOLKIT license option as a startup option to avoid tying up your Creo Parametric licenses.

Note

Only one license will be held for the specified time period, even if multiple applications were successfully unlocked.

Unlocking the application may also require one or more advanced licensing options. Your Creo Object TOOLKIT C++ application may have calls to Creo Object TOOLKIT C++ methods or Creo TOOLKIT functions, which require one or more advanced licensing options. The `protk_unlock` application will detect

whether any functions using advanced licenses are in use in the application, and if so, will make a check for the availability of the advanced license option. If that option is not present, unlocking will not be permitted. If they are present, the unlock will proceed. Advanced options are not held on the license server for any length of time. Refer to the chapter Advanced Licensing Options on page 659 for more information.

If the required licenses are available, the `protk_unlock.bat` application will unlock the Creo Object TOOLKIT C++ application immediately. An unlocked application does not require any of the Creo TOOLKIT license options to run. Depending on the functionality invoked by the application, it may still require certain Creo Parametric options to work correctly.

Note

Once an application binary has been unlocked, it should not be modified in any way (which includes statically linking the unlocked binary with other libraries after the unlock). The unlocked binary must not be changed or else Creo Parametric will again consider it "locked".

Digitally Signing the Application

In Creo 6.0.0.0 and later, you can digitally sign your application. Use the standard Microsoft utility SignTool to digitally sign your application. See the Microsoft documentation for more information on this utility and to create the digital certificate.

Unlock Messages

The following table lists the messages that can be returned when you unlock a Creo Object TOOLKIT C++ application.

Message	Cause
<application name>:Successfully unlocked application.	The application is unlocked successfully.
Usage: `protk_unlock.bat` <one or more Creo TOOLKIT executables or DLLs>	No arguments supplied.
<application name>:ERROR: No READ access <application name>:ERROR: No WRITE access	You do not have READ/WRITE access for the executable.
<application name>:Executable is not a Creo Object TOOLKIT C++ application.	The executable is not linked with the Creo Object TOOLKIT C++ libraries, or does not use any methods from those libraries.
<application name>:Executable is already unlocked.	The executable is already unlocked.
Error: Licenses do not contain Creo TOOLKIT License Code.	A requirement for unlocking a Creo Object TOOLKIT C++ application.

Message	Cause
ERROR: No Creo Parametric licenses are available for the startup command specified	Could not contact the license server.
<application name>:Unlocking this application requires option `TOOLKIT-for-3D_Drawings`.	The application uses methods that require an advanced option; and this option is not available. The license option 222, that is, the `TOOLKIT-for-3D_Drawings` license is not available.

How Creo Object TOOLKIT C++ Works

The standard method by which Creo Object TOOLKIT C++ application code is integrated into Creo application is through the use of dynamically linked libraries (DLLs). When you compile your Creo Object TOOLKIT C++ application code and link it with the Creo Object TOOLKIT C++ libraries, you create an object library file designed to be linked into the Creo executable when the Creo application starts up. This method is referred to as DLL mode.

Creo Object TOOLKIT C++ also supports a second method of integration: the "multiprocess," or spawned mode. In this mode, the Creo Object TOOLKIT C++ application code is compiled and linked to form a separate executable. This executable is designed to be spawned byCreo application and runs as a child process of the Creo session. In DLL mode, the exchanges between the Creo Object TOOLKIT C++ application and Creo application are made through direct function calls. In multiprocess mode, the same effect is created by an inter-process messaging system that simulates direct function calls by passing the information necessary to identify the function and its argument values between the two processes.

Multiprocess mode involves more communications overhead than DLL mode, especially when the Creo Object TOOLKIT C++ application makes frequent calls to Creo Object TOOLKIT C++ library functions, because of the more complex method of implementing those calls. However, it offers the following advantage: it enables you to run the Creo Object TOOLKIT C++ application with a source-code debugger without also loading the whole Creo executable into the debugger.

You can use a Creo Object TOOLKIT C++ application in either DLL mode or multiprocess mode without changing any of the source code in the application. It is also possible to use more than one Creo Object TOOLKIT C++ application within a single session of Creo application, and these can use any combination of modes.

If you use multiprocess mode during development of your application to debug more easily, you should switch to DLL mode when you install the application for your end users because the performance is better in that mode. However, take care to test your application thoroughly in DLL mode before you deliver it. Any programming errors in your application that cause corruption to memory used by

the Creo application or Creo Object TOOLKIT C++ are likely to show quite different symptoms in each mode, so new bugs may emerge when you switch to DLL mode.

Although multiprocess mode involves two processes running in parallel, these processes do not provide genuine parallel processing. There is, however, another mode of integrating asz Creo Object TOOLKIT C++ application that provides this ability, called “asynchronous mode”. For more information on asynchronous mode, see the chapter Asynchronous Mode on page 587. The DLL and multiprocess modes are given the general name “synchronous mode”. An asynchronous Creo Object TOOLKIT C++ application is fundamentally different in its architecture from a synchronous mode application, so you should choose between these methods before writing any application code. As a general rule, synchronous mode should be the default choice unless there is some unavoidable reason to use asynchronous mode, because the latter mode is more complex to use.

Note

All Creo Object TOOLKIT C++ calls running in either synchronous (DLL or multiprocess) mode or asynchronous mode always clear the Undo/Redo stack in the Creo session. The Creo user interface reflects this by making the **Undo** and **Redo** menu options unavailable.

Domains of Creo Object TOOLKIT C++

Creo Object TOOLKIT C++ consists of the following domains:

- `pfc`—This domain provides classes that support basic functionality.
- `wfc`—This domain provides classes that support advanced functionality.
- `uifc`—This domain provides classes that allow you to create user interface components. Refer to the *PTC Creo UI Editor User’s Guide*, for more information on how to create and customize the user interface components.

The splitting of classes under `pfc` and `wfc` domains is relevant in other bindings, for example Java. However, since Creo Object TOOLKIT is uniform through all its bindings, it is also visible in its C++ binding.

Casting of Creo Object TOOLKIT C++ pfc Classes to wfc Classes

The actual type of pfc objects are specified by the equivalent wfc objects, for example, `pfcSolid` object is `wfcWSolid` and `pfcFeature` object is `wfcWFeature`. Therefore, the methods from `wfcWFeature` and `wfcWSolid`

become available to these objects only after applying `wfcWFeature::cast` or `wfcWSolid::cast`. You must check that this cast does not return a null pointer.

To make the methods of wfc classes available in pfc classes, cast the pfc objects to the equivalent wfc objects as described in the table below:

Pfc class	Equivalent WFC class
pfcAssembly	wfcWAssembly
pfcComponentPath	wfcWComponentPath
pfcComponentPaths	wfcWComponentPaths
pfcExplodedState	wfcWExplodedState
pfcSimpRep	wfcWSimpRep
pfcSimpReps	wfcWSimpReps
pfcComponentFeat	wfcWComponentFeat
pfcDetailEntityItem	wfcWDetailEntityItem
pfcDetailGroupItem	wfcWDetailGroupItem
pfcDetailNoteItem	wfcWDetailNoteItem
pfcDetailOLEObject	wfcWDetailOLEObject
pfcDetailSymbolDefItem	wfcWDetailSymbolDefItem
pfcDetailSymbolInstItem	wfcWDetailSymbolInstItem
pfcDimension	wfcWDimension
pfcDimension2D	wfcWDimension2D
pfcRefDimension	wfcWRefDimension
pfcUDFDimension	wfcWUDFDimension
pfcDisplay	wfcWDisplay
pfcWindow	wfcWWindow
pfcDrawing	wfcWDrawing
pfcDrawingFormat	wfcWDrawingFormat
pfcFamilyMember	wfcWFamilyMember
pfcFamilyTableRow	wfcWFamilyTableRow
pfcFamilyTableRows	wfcWFamilyTableRows
pfcCoordSysFeat	wfcWCoordSysFeat
pfcCurveFeat	wfcWCurveFeat
pfcDatumAxisFeat	wfcWDatumAxisFeat
pfcDatumPlaneFeat	wfcWDatumPlaneFeat
pfcDatumPointFeat	wfcWDatumPointFeat
pfcFeature	wfcWFeature

Pfc class	Equivalent WFC class
pfcFeatures	wfcWFeatures
pfcRoundFeat	wfcWRoundFeat
pfcArc	wfcWArc
pfcArrow	wfcWArrow
pfcAxis	wfcWAxis
pfcBSpline	wfcWBSpline
pfcCircle	wfcWCircle
pfcCompositeCurve	wfcWCompositeCurve
pfcCompositeCurveDescriptor	wfcWCompositeCurveDescriptor
pfcCone	wfcWCone
pfcConeDescriptor	wfcWConeDescriptor
pfcContour	wfcWContour
pfcContours	wfcWContours
pfcCoonsPatch	wfcWCoonsPatch
pfcCoonsPatchDescriptor	wfcWCoonsPatchDescriptor
pfcCurve	wfcWCurve
pfcCurves	wfcWCurves
pfcCylinder	wfcWCylinder
pfcCylinderDescriptor	wfcWCylinderDescriptor
pfcCylindricalSplineSurface	wfcWCylindricalSplineSurface
pfcCylindricalSplineSurfaceDescriptor	wfcWCylindricalSplineSurfaceDescriptor
pfcEdge	wfcWEdge
pfcEdges	wfcWEdges
pfcEllipse	wfcWEllipse
pfcFilletSurface	wfcWFilletSurface
pfcFilletSurfaceDescriptor	wfcWFilletSurfaceDescriptor
pfcForeignSurface	wfcWForeignSurface
pfcForeignSurfaceDescriptor	wfcWForeignSurfaceDescriptor
pfcLine	wfcWLine
pfcNURBSSurface	wfcWNURBSSurface
pfcNURBSSurfaceDescriptor	wfcWNURBSSurfaceDescriptor
pfcPlane	wfcWPlane
pfcPlaneDescriptor	wfcWPlaneDescriptor
pfcPoint	wfcWPoint

Pfc class	Equivalent WFC class
pfcPolygon	wfcWPolygon
pfcQuilt	wfcWQuilt
pfcRevolvedSurface	wfcWRevolvedSurface
pfcRevolvedSurfaceDescriptor	wfcWRevolvedSurfaceDescriptor
pfcRuledSurface	wfcWRuledSurface
pfcRuledSurfaceDescriptor	wfcWRuledSurfaceDescriptor
pfcSurface	wfcWSurface
pfcSurfaceDescriptor	wfcWSurfaceDescriptor
pfcSurfaceDescriptors	wfcWSurfaceDescriptors
pfcSurfaces	wfcWSurfaces
pfcText	wfcWText
pfcTorus	wfcWTorus
pfcTorusDescriptor	wfcWTorusDescriptor
pfcDiagram	wfcWDiagram
pfcIntfNeutral	wfcWIntfNeutral
pfcLayout	wfcWLayout
pfcMarkup	wfcWMarkup
pfcModel	wfcWModel
pfcReport	wfcWReport
pfcUnit	wfcWUnit
pfcModelItem	wfcWModelItem
pfcModelItems	wfcWModelItems
pfcParameter	wfcWParameter
pfcParameterOwner	wfcWParameterOwner
pfcRelationOwner	wfcWRelationOwner
pfcNote	wfcWNote
pfcPart	wfcWPart
pfcSelection	wfcWSelection
pfcSelectionOptions	wfcWSelectionOptions
pfcSelections	wfcWSelections
pfcBaseSession	wfcWBaseSession
pfcSession	wfcWSession
pfcSolid	wfcWSolid
pfcRegenInstructions	wfcWRegenInstructions

Pfc class	Equivalent WFC class
pfcTable	wfcWTable
pfcXSection	wfcWXSection

Categories of Creo Object TOOLKIT C++ Classes

Creo Object TOOLKIT C++ is made up of a number of classes. Creo Object TOOLKIT C++ supersedes the older PTC object-oriented toolkits without duplicating them. It incorporates Creo Elements/Pro `pfc` interfaces and adds new interfaces under `wfc` prefix, where "`w`" indicates "Writable", because in addition to `pfc`, the `wfc` classes provide more capabilities to change a model. For example, you can use the `wfc` classes to create features from feature element trees.

The following are the main classes:

- `Creo-Related Interfaces`—Contains unique methods and attributes that are directly related to the functions in the Creo applications. See the section Creo-Related Interfaces on page 26 for additional information.
- `Compact Data Classes`—Classes containing data needed as arguments to some Creo Object TOOLKIT C++ methods. See the section Compact Data Classes on page 28 for additional information.
- `Union Classes`—Classes with a potential for multiple types of values. See the section Union Classes on page 28 for additional information.
- `Sequence Classes`—Expandable arrays of objects or primitive data types. See the section Sequence Classes on page 29 for more information.
- `Array Classes`—Arrays that are limited to a certain size. See the section Array Classes on page 31 for more information.
- `Enumeration Classes`—Classes that define the enumerated types. See the section Enumeration Classes on page 32 for more information.
- `ActionListener Classes`—Classes that enable you to specify callbacks that will run only if certain events in Creo application take place. See the section Action Listener Classes on page 33 for more information.
- `Utility Classes`—Classes that contain the static methods used to initialize certain Creo Object TOOLKIT C++ objects. See the section Utilities on page 35 for more information.

Each class shares specific rules regarding initialization, attributes, methods, inheritance, or exceptions. The following sections describe these classes in detail.

List of Classes and Methods

PTC provides a list of all the available classes and methods in Creo Object TOOLKIT C++. The information is available in the file `otk_methods.txt` located at `creo_otk_loadpoint_doc`. The file is a tab delimited text file, which can be read in Microsoft Excel.

The file lists all the methods in the `pfc` and `wfc` domains along with their description. The file has the following fields:

- `C++ Header`, `C++ Class`, and `C++ Method`—Lists the Creo Object TOOLKIT C++ header files, classes and methods.
- `Exposure`—Specifies `PMA` if the method is supported in Creo Parametric. The field specifies `PMA` when the method is supported in Creo Parametric.
- `Description`—Describes the class or method.

Creo-Related Interfaces

The Creo-related interfaces contain methods that directly manipulate objects in the Creo application. Examples of these objects include models, features, and parameters.

Initialization, Smart Pointers, and Memory Management

You cannot construct one of these objects using the C++ constructors, because their types are represented by the pure virtual C++ classes. Such objects must be returned by `Get` or `Create` method of the special top-level interfaces. The actual type of these objects is represented by a smart pointer. Smart pointer is a class whose instances are kept alive until other objects which refer to them exist in runtime memory . Smart pointers are based on PTC Component Interface Protocol (CIP), and they are defined through the `xrchandle` template, applied to the pure virtual class whose interface they are about to provide.

For example, `pfcSession::GetCurrentModel` returns a `pfcModel_ptr` object set to the current model and `pfcParameterOwner::CreateParam` returns a newly created `pfcParameter_ptr` object for manipulation.

Attributes

Attributes within Creo-related objects are not directly accessible, but can be accessed through `Get` and `Set` methods. These methods are of the following types:

```
Attribute name: int XYZ
Methods:    int GetXYZ();
            void SetXYZ (xint i);
```

Some of the attributes that have been designated as read-only, can be accessed only by the `Get` method.

Methods

All non-static methods must be used on smart-pointer objects, that follow the C++ syntax for pointers. You must first initialize that object.

For example, the following calls are legal:

```
pfcSession_ptr session = pfcGetProESession();
pfcWindow_ptr window = session->GetCurrentWindow();
       // You have initialized the window object.
window->Activate();
window->Repaint();
```

The following calls are illegal:

```
pfcWindow window; // pfcWindow is a pure virtual class, C++
constructor
 cannot be used

window.Activate();

Repaint();        // There is no invoking object.
```

Inheritance

All Creo-related objects are defined as interfaces so that they can inherit methods from other interfaces. To use these methods, call them directly (no casting is needed). For example, for `pfcFeature`, which inherits `pfcModelItem`, you can do this:

```
pfcFeature_ptr myfeature;           // Previously initialized

String name = myfeature->GetName(); // GetName is in the
                                    // class pfcModelItem.
```

However, if you have a reverse situation, you need to explicitly cast the object. For example:

```
pfcModelItem_ptr item;  // You know this is a pfcFeature --
perhaps
                        // you previously checked its type.
int number = pfcFeature::cast(item)->GetNumber();
                        // GetNumber() is a pfcFeature method.
```

Exceptions

Almost every Creo Object TOOLKIT C++ method can throw an exception of type `jxthrowable`. Surround each method you use with a `xcatchbegin-try-xcatch-xcatchend` block to handle any exceptions that are generated. See the Exceptions section for more information.

Compact Data Classes

Compact data classes are data-only classes. They are used, as needed, for arguments and return values for some Creo Object TOOLKIT C++ methods. They do not represent the actual objects in the Creo application.

Initialization

You can create instances of compact classes using the static create methods. For example:

```
pfcBOMExportInstructions::Create()
```

Such static methods usually belong to the same class whose instances they create.

Attributes

Attributes within the compact data related classes are not directly accessible, but can be accessed using the `Get` and `Set` methods. These methods are of the following types:

```
Attribute name: int XYZ
Methods:    int GetXYZ();
            void SetXYZ (xint i);
```

Methods

You must obtain a smart pointer on an object before calling its methods. For example, the following calls are illegal:

```
pfcSelectionOptions options;

options.SetMaxNumSels(); // The object has not been initialized.
SetOptionsKeywords();    // There is no invoking object
```

Inheritance

Compact objects can inherit methods from other compact interfaces. To use these methods, call them directly (no casting needed).

Exceptions

Almost every Creo Object TOOLKIT C++ method can throw an exception of type `xthrowable`. Surround each method you use with a `try-catch-finally` block to handle any exceptions that are generated.

Union Classes

Unions are interface-like objects. Every union has a discriminator method with the pre-defined name `Getdiscr()`. This method returns a value identifying the type of data that the union objects holds. For each union member, a pair of `Get`/`Set`

methods are used to access the different data types. It is illegal to call any `Get` method except the one that matches the value returned from `Getdiscr()`. However, any `Set` method can be called. This switches the discriminator to the new value.

The following is an example of the Creo Object TOOLKIT C++ union:

```
class pfcParamValue : public xobject
{
xsdeclare (pfcParamValue)
public:
    pfcParamValueType            Getdiscr ();
    xstring                                  GetStringValue ();
    void                                     SetStringValue
(xrstring value);
    xint                                     GetIntValue ();
    void                                     SetIntValue (xint
value);
    xbool                                    GetBoolValue ();
    void                                     SetBoolValue (xbool
value);
    xreal                                    GetDoubleValue ();
    void                                     SetDoubleValue (xreal
value);
    xint                 GetNoteId ();
    void                                        SetNoteId (xint
value);

private:
    pfcParamvalueValue              mProParamvalue;

public:
        pfcParamValue (const pfcParamvalueValue &inProParamvalue);
        pfcParamValue ();

public:
        const pfcParamvalueValue    &GetProParamvalue();
};
```

Sequence Classes

Sequences are expandable arrays of primitive data types or objects in Creo Object TOOLKIT C++. All sequence classes have the same methods for adding and accessing the array. Sequence classes are identified by a plural name, or the suffix `seq`.

Initialization

You cannot construct sequences of objects using the C++ constructors. Static create methods for each sequence type are available. For example, `pfcModels::create()` returns an empty `Models` sequence object for you to fill in.

Attributes

The attributes within the sequence objects must be accessed using methods.

Methods

Sequence objects always contain the same methods: `get`, `set`, `getarraysize`, `insert`, `insertseq`, `removerange`, and `create`. Methods must be invoked from an initialized object of the correct type, except for the static `create` method, which is invoked from the sequence class.

Inheritance

Sequence classes do not inherit from any other Creo Object TOOLKIT C++ classes. Therefore, you cannot cast sequence objects to any other type of Creo Object TOOLKIT C++ object, including other sequences. For example, if you have a sequence of model items that happen to be features, you cannot make the following call:

```
pfcFeatures_ptr features = pfcFeatures::cast(modelitems);
```

To construct the sequence of features, you must insert each member of the sequence separately while casting it to a `pfcFeature`.

Exceptions

If you try to get or remove an object outside of the range of the sequence, the exceptions `cipXInvalidSeqIndex` or `cipXNegativeIndex` are thrown.

Example Code: Sequence Class

The following shows the declaration of the sequence class `pfcModels` (only public methods are quoted). Please note that the methods of this class come from different typedef, macros and parent classes. There is no specific header file which contains all methods of `pfcModels` in one place. This is typical for sequences in C++ Creo Object TOOLKIT C++.

```
// in pfcModel::h:
xclssequence (optional pfcModel_ptr, pfcModels);

// in cipxseq.h:
# define xclssequence(TYPE, NAME)        \
    class NAME : public xtcsequence \
```

```
 {         \
        xsdeclare (NAME)       \
        NAME () {}           \
        NAME (xint capacity) : xtcsequence<TYPE> (capacity)
  NAME (const NAME *src) : xtcsequence<TYPE> (src)    \
static NAME *create ();   \
static NAME *createCapacity (xint capacity);     \
}

template <class ElemType>
class xtcsequence : public xtbsequence  <class ElemType>
{
    public:
                xtcsequence ();
                xtcsequence (xint capacity);
                xtcsequence (const xtcsequence<ElemType> *src);
       ~xtcsequence ();
};

template <class ElemType>
class xtbsequence : public xobject, public xbasesequence
{
 public:
     virtual xint                    getarraysize ();
     inline ElemType                                &operator  []
(xint idx);
     virtual ElemType                                 get (xint
idx);
     virtual void                             set (xint idx,
ElemType value);
     virtual void                             append (ElemType
value);
     virtual void                           operator+= (ElemType
value);
     virtual void                      insert (xint atidx, ElemType
value);
     virtual void                             insertseq (xint
atidx,
       xtbsequence <ElemType> *arr);
virtual void                               removerange (xint
frominc, xint toexcl);

};
```

Array Classes

Arrays are groups of primitive types or objects of a specified size. An array can be one or two dimensional. The following array classes are available in `pfcBase.h`: `pfcMatrix3D`, `pfcPoint2D`, `pfcPoint3D`,

pfcOutline2D, pfcOutline3D, pfcUVVector, pfcUVParams, pfcVector2D, and pfcVector3D. See the online reference documentation to determine the exact size of these arrays.

Initialization

You cannot construct one of these objects using the C++ constructors. Static creation methods are available for each array type. For example, the method pfcPoint2D::create returns an pfcPoint2D array object for you to fill in. If the arrays are not initialized, the element values of such arrays must be considered as undefined.

Attributes

The attributes within array objects must be accessed using array class methods.

Methods

Array objects always contain the same methods: get, set, and create. Methods must be invoked from an initialized object of the correct type, except for the create method, which is invoked from the name of the array class.

Inheritance

Array classes do not inherit from any other Creo Object TOOLKIT C++ classes.

Exceptions

If you try to get or remove an object outside of the range of the array, the exceptions cipXInvalidSeqIndex or cipXNegativeIndex are thrown.

Enumeration Classes

In Creo Object TOOLKIT C++ the enumeration classes are used in the same way as an enum is used in C or C++.

Sharing Enumerations and Constants with Creo TOOLKIT

Since Creo Object TOOLKIT C++ and Creo TOOLKIT calls can be combined in the same code, there is no need in introducing special enumerations and constants for Creo Object TOOLKIT C++ (except for pfc enumerations and constants which come from pre-Creo Parametric 1.0 pfc interfaces).

Action Listener Classes

Use ActionListeners in Creo Object TOOLKIT C++ to assign programmed reactions to events that occur within the Creo application. Creo Object TOOLKIT C++ defines a set of action listener interfaces that can be implemented enabling Creo to call your Creo Object TOOLKIT C++ application when specific events occur. These interfaces are designed to respond to events from action sources in Creo application. Examples of action sources include the session, user-interface commands, models, solids, parameters, and features.

Initialization

Creo Object TOOLKIT C++ provides a few Action Listener classes, whose actions map most of notification types in Creo TOOLKITProNotifyType. For example, the actions of pfcSolidActionListener correspond to PRO_SOLID_REGEN_PRE, PRO_SOLID_REGEN_POST, PRO_FEATURE.Create_PRE, PRO_FEATURE.Create_POST, PRO_FEATURE_DELETE_POST, PRO_SOLID_UNIT_CONVERT_PRE, or PRO_SOLID_UNIT_CONVERT_POST. All action listeners are derived from top-level pfcActionListener class. Each Action Listener class contains only pure virtual methods. After choosing an appropriate action listener, you have to subclass it and provide the implementation of all its actions, even if you need only few. To make this more convenient, PTC provides default implementations of each ActionListener class, with an empty implementation of each action. These default classes are not part of Creo Object TOOLKIT C++, but they are skeletons, which you can insert into your application after appropriate modifications. Construct the instance of your Action Listener class using the C++ keyword new. Thereafter, assign your action listener to an pfcActionSource using the AddActionListener() or pfcActionSource::AddActionListenerWithType method of the action source. When you use the method pfcActionSource::AddActionListenerWithType, you can choose to register only specific actions. These actions are specified in the ActionTypes array.

It is recommended to use smart pointer to create an instance of pfc Action Listener class. The smart pointer can be destroyed by calling the pfcActionSource::RemoveActionListener method. The keyword delete is not required to delete the smart pointer.

If an instance of the uifc Action Listener class is assigned to a C-type (*) pointer, then use the C++ keyword delete to delete the pointer. However, if the instance of Action Listener class is assigned to a smart pointer, then the keyword delete is not required to delete the pointer. It is managed and deleted internally.

Attributes

Action listeners do not have any accessible attributes.

Methods

You must override the methods you need in the default class to create an `ActionListener` object correctly. The methods you create can call other methods in the `ActionListener` class or in other classes.

Inheritance

All Creo Object TOOLKIT C++ `ActionListener` objects inherit from the interface `pfcActionListener`.

Exceptions

Action listeners cause methods to be called outside of your application. Therefore, you must include exception-handling code inside the `ActionListener` implementation if you want to respond to exceptions. In some methods, when called before an event, propagating an exception out of your method will cancel the impending event.

Example Code: Listener Class

The following example code shows part of the `pfcSolidActionListener` interface.

```
class pfcSolidActionListener : public virtual pfcActionListener
{
xaideclare (pfcSolidActionListener)
public:
    virtual void                    OnBeforeRegen (
        pfcSolid_ptr                    Sld,
        optional pfcFeature_ptr         StartFeature
    ) = 0;

    virtual void                    OnAfterRegen (
        pfcSolid_ptr                    Sld,
        optional pfcFeature_ptr         StartFeature,
        xbool                           WasSuccessful
    ) = 0;

    virtual void                    OnBeforeUnitConvert (
        pfcSolid_ptr                    Sld,
        xbool                           ConvertNumbers
    ) = 0;

    virtual void                    OnAfterUnitConvert (
        pfcSolid_ptr                    Sld,
        xbool                           ConvertNumbers
    ) = 0;

    virtual void                    OnBeforeFeatureCreate (
```

```
                pfcSolid_ptr            Sld,
                xint                    FeatId
    ) = 0;

    virtual void                    OnAfterFeatureCreate (
                pfcSolid_ptr            Sld,
                pfcFeature_ptr          Feat
    ) = 0;

    virtual void                    OnAfterFeatureDelete (
                pfcSolid_ptr            Sld,
                xint                    FeatId
    ) = 0;
};
```

Utilities

Each package in Creo Object TOOLKIT C++ has one class that contains special static methods used to create and access some of the other classes in the package. These utility classes have the same name as the package, such as `pfcModel::pfcModel`.

Initialization

Because the utility packages have only static methods, you do not need to initialize them. Simply access the methods through the name of the class, as follows:

```
ParamValue pv = pfcModelItem.CreateStringParamValue ("my_param");
```

Attributes

Utilities do not have any accessible attributes.

Methods

Utilities contain only static methods used for initializing certain Creo Object TOOLKIT C++ objects.

Inheritance

Utilities do not inherit from any other Creo Object TOOLKIT C++ classes.

Exceptions

Methods in utilities can throw `jxthrowable` type exceptions.

Sample Utility Class

The following code example shows the utility class `pfcGlobal`.

```
public class pfcGlobal
{
public static pfcSession::Session GetProESession()
    throws jxthrowable

public static stringseq GetProEArguments()
    throws jxthrowable

public static string GetProEVersion()
    throws jxthrowable

public static string GetProEBuildCode()
    throws jxthrowable
}
```

Creo Object TOOLKIT C++ Support for Creo

Methods Introduced:

- **wfcWSession::RunAsCreoType**
- **wfcWSession::GetCreoType**

Creo Object TOOLKIT C++ supports applications in synchronous and asynchronous modes for Creo applications.

Note

Creo Object TOOLKIT C++ does not support other Creo applications, such as, Creo Layout, Creo Simulate and so on.

In future, the methods of Creo Object TOOLKIT C++ will be enhanced to extend support to all Creo applications.

If you call a method that is not supported in the Creo application, the `pfcXMethodForbidden` exception is thrown.

You can simulate the Creo type at runtime and check for `pfcXMethodForbidden` exceptions.

Use the method `wfcWSession::RunAsCreoType` to run the Creo Object TOOLKIT C++ application in the specified Creo application. Specify the Creo application using the enumerated data type `wfcCreoType`. The valid value is:

- `wfcCREO_PARAMETRIC`

Note

The method `wfcWSession::RunAsCreoType` must be used only with Creo Parametric. If used with other Creo applications, the method returns the `pfcXMethodForbidden` exception. Therefore, you must remove the call to this method before deploying the final application.

The method `wfcWSession::GetCreoType` returns the Creo application to which the Creo Object TOOLKIT C++ application is connected.

The keywords `toolkit` and `creo_type` defined in the registry file are used to specify the information required to run a Creo Object TOOLKIT C++ application in a non-Creo Parametric application. The non-Creo Parametric applications read the information from the registry file.

The keyword `toolkit` has two values:

- `protoolkit` for applications based on Creo TOOLKIT
- `object` for applications based on Creo Object TOOLKIT C++

If you run an application of type `protoolkit` with any Creo application other than Creo Parametric, an error message appears. Similarly, if you try to run the Creo Object TOOLKIT C++ application on a Creo application other than the one mentioned in the registry file, an error message appears.

The **Auxiliary Applications** dialog box is available within the non-Creo Parametric applications.

Support for Multi-CAD Models Using Creo Unite

Creo Unite enables you to open non-Creo parts and assemblies in Creo Parametric and other Creo applications such as Creo Simulate without creating separate Creo models. You can then assemble the part and assembly models that you opened as components of Creo assemblies to create multi-CAD assemblies of mixed content.

You can open the part and assembly models of the following non-Creo file formats in Creo applications:

- CATIA V5 (`.CATPart`, `.CATProduct`)
- CATIA V5 CGR

- CATIA V4 (`.Model`)
- SolidWorks (`.sldasm`, `.sldprt`)
- NX (`.prt`)

Most of the Creo Object TOOLKIT C++ methods support multi-CAD assemblies. The methods which do not support assemblies of mixed content throw the exception `pfcXToolkitUnsupported`, when a non-native part or assembly is passed as the input model.

Using Creo Object TOOLKIT C++ with Creo TOOLKIT

Since C++ and C functions can be used together in the same application, you can use the Creo TOOLKIT applications where the Creo Object TOOLKIT C++ interfaces are not yet available. To do this seamlessly, the program should be able to obtain Creo TOOLKIT handles out of Creo Object TOOLKIT C++ objects and vice versa. For example, to obtain a `ProMdl` out of `pfcModel_ptr` and `pfcModel_ptr` out of `ProMdl`.

Two static functions allow you to achieve this:

- `xobject_ptr wfcGetObjectFromHandle (wfcHandleType type, void *handle);`
- `const void *wfcGetHandleFromObject (pfcObject_ptr object);`

The static function `wfcGetObjectFromHandle` returns a Creo Object TOOLKIT C++ object from a Creo TOOLKIT handle. Use the function `wfcGetHandleFromObject` to obtain a Creo TOOLKIT handle from a Creo Object TOOLKIT C++ object.

Note

The calls to Creo TOOLKIT functions from a Creo Object TOOLKIT C++ based application is supported only for Creo Parametric. Thus, the support for functions `wfcGetHandleFromObject` and `wfcGetObjectFromHandle` is restricted to applications that run with Creo Parametric only.

For more information on all the possible values of `wfcHandleType` see `wfcGlobal.h`. The following is an example of using `wfcGetObjectFromHandle` and `wfcGetHandleFromObject`:

```
// getting pfcFeature_ptr out of ProFeature
ProFeature *feat;
```

```
pfcFeature_ptr pfcFeat = pfcFeature::cast
( wfcGetObjectFromHandle(wfcProModelitemHandle, (void*)feat) );

// getting ProFeature out of pfcFeature_ptr
pfcFeature_ptr pfcFeat;
ProFeature *feat = (ProFeature*) wfcGetHandleFromObject
( pfcObject::cast(pfcFeat) );
```

Migrating Creo TOOLKIT Applications to Creo Object TOOLKIT C++ Using Tools

The tool `mark_otkmethod.pl` helps you migrate Creo TOOLKIT applications to Creo Object TOOLKIT C++. This perl script is located at `<creo_loadpoint>\<datecode>\Common Files\protoolkit\scripts`. The script searches for Creo TOOLKIT functions, and recommends possible replacing Creo Object TOOLKIT C++ methods.

The Creo TOOLKIT APIWizard also provides links to equivalent Creo Object TOOLKIT C++ methods.

Creating Applications

The following sections describe how to create applications. The topics are as follows:

Application Hierarchy

The rules of object dependencies require a certain sequence of object creation when you start a Creo Object TOOLKIT C++ application. First, you must obtain a `wfcSessionm :`

```
wfcWSession_ptr ses = wfcWSession::cast(pfcGetProESession()
```

which returns a handle to the current session of Creo application.

The application must iterate down to the level of object you want to access. For example, to list all the datum axes contained in the hole features in all models in session, do the following:

1. Get a handle to the session.
2. Get the models that are active in the session.
3. Get the feature model items in each model.

4. Filter out the features of type hole.
5. Get the subitems in each feature that are axes.

Exception Handling

Many Creo Object TOOLKIT C++ methods can throw an exception. Exceptions match errors returned by Creo TOOLKIT APIs. The following sections describe the exceptions in detail.

Note

Exceptions thrown by Creo Object TOOLKIT C++ methods are not documented in the Creo Object TOOLKIT C++ User's Guide. Refer to the APIWizard for more information on method specific exceptions.

Cip Exceptions

The `Cip` exceptions are thrown by `Cip` classes. For more information, `cipxx.h` and its includes. With the exception of `intseq`, `realseq`, `boolseq`, and `stringseq` classes, these classes are used only internally.

The following table describes these exceptions.

Exception	Purpose
`cipXInvalidArrayIndex`, `cipXInvalidDictIndex`, `cipXNegativeIndex`	Illegal index value used when accessing a `Cip` array, sequence, or dictionary. (The Creo Object TOOLKIT C++ interface does not currently include any dictionary classes.)
Other, internal errors	Internal assertions that should not append and which need not be caught individually.

PFC/WFC Exceptions

The PFC/WFC exceptions are thrown by the classes that make up Creo Object TOOLKIT C++'s public interface. The following table describes these exceptions.

Exception	Purpose
`pfcXBadExternalData`	Indicates an attempt to read contents of an external data object which has been terminated.
`pfcXBadGetArgValue`	Indicates an attempt to read the wrong type of data from the `ArgValue` union.
`pfcXBadGetExternalData`	Indicates an attempt to read the wrong type of data from the `ExternalData` union.
`pfcXBadGetParamValue`	Indicates an attempt to read the wrong type of data from the `ParamValue` union.
`pfcXBadOutlineExcludeType`	Indicates that an invalid type of item was passed to the outline calculation method.

Exception	Purpose
pfcXCancelProEAction	This exception type will not be thrown by Creo Object TOOLKIT C++ methods, but you may instantiate and throw this from certain ActionListener methods to cancel the corresponding action in the Creo application.
pfcXCannotAccess	Indicates that the contents of a Creo Object TOOLKIT C++ object cannot be accessed in this situation.
pfcXEmptyString	Indicates an empty string was passed to a method that does not accept this type of input.
pfcXInvalidEnumValue	Indicates an invalid value for a specified enumeration class.
pfcXInvalidFileName	Indicates that a file name passed to a method was incorrectly structured.
pfcXInvalidFileType	Indicates a model descriptor contained an invalid file type for a requested operation.
pfcXInvalidModelItem	Indicates that the item requested to be used is no longer usable (for example, it may have been deleted).
pfcXInvalidSelection	Indicates that the selection passed is invalid or is missing a needed piece of information. For example, its component path, drawing view, or parameters.
pfcXModelNotInSession	Indicates that the model is no longer in session; it may have been erased or deleted.
pfcXNegativeNumber	Numeric argument was negative.
pfcXNumberTooLarge	Numeric argument was too large.
pfcXProEWasNotConnected	The Creo session is not available so the operation failed.
pfcXSequenceTooLong	Indicates that the sequence argument was too long.
pfcXStringTooLong	Indicates that the string argument was too long.
pfcXUnimplemented	Indicates unimplemented method.
pfcXUnknownModelExtension	Indicates that a file extension does not match a known Creo model type.

Creo TOOLKIT Errors

The pfcXToolkitError exception provides access to error codes from Creo TOOLKIT functions that Creo Object TOOLKIT C++ uses internally and to the names of the functions returning such errors. pfcXToolkitError is the exception you are most likely to encounter because Creo Object TOOLKIT C++ is built on top of Creo TOOLKIT. The following table lists the integer values that can be returned by the pfcXToolkitError::GetErrorCode() method and shows the corresponding Creo TOOLKIT constant that indicates the cause of the error. Each specific pfcXToolkitError exception is represented by an appropriately named child class, allowing you to catch specific exceptions you need to handle separately.

pfcXToolkitError Child Class	Creo TOOLKIT Error	#
pfcXToolkitGeneralError	PRO_TK_GENERAL_ERROR	-1
pfcXToolkitBadInputs	PRO_TK_BAD_INPUTS	-2
pfcXToolkitUserAbort	PRO_TK_USER_ABORT	-3
pfcXToolkitNotFound	PRO_TK_E_NOT_FOUND	-4
pfcXToolkitFound	PRO_TK_E_FOUND	-5
pfcXToolkitLineTooLong	PRO_TK_LINE_TOO_LONG	-6
pfcXToolkitContinue	PRO_TK_CONTINUE	-7
pfcXToolkitBadContext	PRO_TK_BAD_CONTEXT	-8
pfcXToolkitNotImplemented	PRO_TK_NOT_IMPLEMENTED	-9
pfcXToolkitOutOfMemory	PRO_TK_OUT_OF_MEMORY	-10
pfcXToolkitCommError	PRO_TK_COMM_ERROR	-11
pfcXToolkitNoChange	PRO_TK_NO_CHANGE	-12
pfcXToolkitSuppressedParents	PRO_TK_SUPP_PARENTS	-13
pfcXToolkitPickAbove	PRO_TK_PICK_ABOVE	-14
pfcXToolkitInvalidDir	PRO_TK_INVALID_DIR	-15
pfcXToolkitInvalidFile	PRO_TK_INVALID_FILE	-16
pfcXToolkitCantWrite	PRO_TK_CANT_WRITE	-17
pfcXToolkitInvalidType	PRO_TK_INVALID_TYPE	-18
pfcXToolkitInvalidPtr	PRO_TK_INVALID_PTR	-19
pfcXToolkitUnavailableSection	PRO_TK_UNAV_SEC	-20
pfcXToolkitInvalidMatrix	PRO_TK_INVALID_MATRIX	-21
pfcXToolkitInvalidName	PRO_TK_INVALID_NAME	-22
pfcXToolkitNotExist	PRO_TK_NOT_EXIST	-23
pfcXToolkitCantOpen	PRO_TK_CANT_OPEN	-24
pfcXToolkitAbort	PRO_TK_ABORT	-25
pfcXToolkitNotValid	PRO_TK_NOT_VALID	-26
pfcXToolkitInvalidItem	PRO_TK_INVALID_ITEM	-27
pfcXToolkitMsgNotFound	PRO_TK_MSG_NOT_FOUND	-28
pfcXToolkitMsgNoTrans	PRO_TK_MSG_NO_TRANS	-29
pfcXToolkitMsgFmtError	PRO_TK_MSG_FMT_ERROR	-30
pfcXToolkitMsgUserQuit	PRO_TK_MSG_USER_QUIT	-31
pfcXToolkitMsgTooLong	PRO_TK_MSG_TOO_LONG	-32
pfcXToolkitCantAccess	PRO_TK_CANT_ACCESS	-33
pfcXToolkitObsoleteFunc	PRO_TK_OBSOLETE_FUNC	-34
pfcXToolkitNoCoordSystem	PRO_TK_NO_COORD_SYSTEM	-35
pfcXToolkitAmbiguous	PRO_TK_E_AMBIGUOUS	-36
pfcXToolkitDeadLock	PRO_TK_E_DEADLOCK	-37
pfcXToolkitBusy	PRO_TK_E_BUSY	-38
pfcXToolkitInUse	PRO_TK_E_IN_USE	-39
pfcXToolkitNoLicense	PRO_TK_NO_LICENSE	-40
pfcXToolkitBsplUnsuitableDegree	PRO_TK_BSPL_UNSUITABLE_DEGREE	-41
pfcXToolkitBsplNonStdEndKnots	PRO_TK_BSPL_NON_STD_END_KNOTS	-42
pfcXToolkitBsplMultiInnerKnots	PRO_TK_BSPL_MULTI_INNER_KNOTS	-43

pfcXToolkitError Child Class	Creo TOOLKIT Error	#
pfcXToolkitBadSrfCrv	PRO_TK_BAD_SRF_CRV	-44
pfcXToolkitEmpty	PRO_TK_EMPTY	-45
pfcXToolkitBadDimAttach	PRO_TK_BAD_DIM_ATTACH	-46
pfcXToolkitNotDisplayed	PRO_TK_NOT_DISPLAYED	-47
pfcXToolkitCantModify	PRO_TK_CANT_MODIFY	-48
pfcXToolkitCheckoutConflict	PRO_TK_CHECKOUT_CONFLICT	-49
pfcXToolkitCreateViewBadSheet	PRO_TK_CRE_VIEW_BAD_SHEET	-50
pfcXToolkitCreateViewBadModel	PRO_TK_CRE_VIEW_BAD_MODEL	-51
pfcXToolkitCreateViewBadParent	PRO_TK_CRE_VIEW_BAD_PARENT	-52
pfcXToolkitCreateViewBadType	PRO_TK_CRE_VIEW_BAD_TYPE	-53
pfcXToolkitCreateViewBadExplode	PRO_TK_CRE_VIEW_BAD_EXPLODE	-54
pfcXToolkitUnattachedFeats	PRO_TK_UNATTACHED_FEATS	-55
pfcXToolkitRegenerateAgain	PRO_TK_REGEN_AGAIN	-56
pfcXToolkitDrawingCreateErrors	PRO_TK_DWGCREATE_ERRORS	-57
pfcXToolkitUnsupported	PRO_TK_UNSUPPORTED	-58
pfcXToolkitNoPermission	PRO_TK_NO_PERMISSION	-59
pfcXToolkitAuthenticationFailure	PRO_TK_AUTHENTICATION_FAILURE	-60
pfcXToolkitMultibodyUnsupported	PRO_TK_MULTIBODY_UNSUPPORTED	-69
pfcXToolkitAppNoLicense	PRO_TK_APP_NO_LICENSE	-92
pfcXToolkitAppExcessCallbacks	PRO_TK_APP_XS_CALLBACKS	-93
pfcXToolkitAppStartupFailed	PRO_TK_APP_STARTUP_FAIL	-94
pfcXToolkitAppInitializationFailed	PRO_TK_APP_INIT_FAIL	-95
pfcXToolkitAppVersionMismatch	PRO_TK_APP_VERSION_MISMATCH	-96
pfcXToolkitAppCommunicationFailure	PRO_TK_APP_COMM_FAILURE	-97
pfcXToolkitAppNewVersion	PRO_TK_APP_NEW_VERSION	-98

The exception `XProdevError` represents a general error that occurred when executing a Pro/DEVELOP function and is equivalent to a `Xtoolkit` general error. (PTC does not recommend the use of Pro/DEVELOP functions.)

The exception `XExternalDataError` and its children are thrown from `External Data` methods. See the chapter on External Data for more information.

Approaches to Creo Object TOOLKIT C++ Exception Handling

To deal with the exceptions generated by Creo Object TOOLKIT C++ methods surround each method with a `try-xcatchbegin-xcatch-xcatchend` block. For example:

```
try {
```

```
    OtkObject->DoSomething()
}
xcatchbegin
xcatch (xthrowable, x) {
    // Respond to the exception.
}
xcatchend
```

Rather than catching the generic exception, you can set up your code to respond to specific exception types, using multiple `catch` blocks to respond to different situations, as follows:

```
try
{
    OtkObject->DoSomething()
}
xcatchbegin
xcatch (pfcXToolkitError, x)
    {
        // Respond based on the error code.
        x->GetErrorCode();
    }
xcatch (cipXConnectionClosed, x)
    {
        // Respond to the exception.
    }
catch (xthrowable, x)  // Do not forget to check for
                        // an unexpected error!
{
        // Respond to the exception.
}
xcatchend
catch(...) {
        // non-OTK exceptions go here
}
```

Version Compatibility: Creo Parametric and Creo Object TOOLKIT C++

In many situations it will be inconvenient or impossible to ensure that the users of your Creo Object TOOLKIT C++ application use the same build of Creo Parametric used to compile and link the Creo Object TOOLKIT C++ application. This section summarizes the rules for mixing Creo Object TOOLKIT C++ and Creo Parametric versions. The Creo Object TOOLKIT C++ version is the Creo Parametric CD version from which the user installed the Creo Object TOOLKIT C++ version used to compile and link the application.

Method Introduced:

- **wfcWSession::GetReleaseNumericVersion**

This method returns the version number of the Creo Parametric executable to which the Creo Object TOOLKIT C++ application is connected. This number is an absolute number and represents the major release of the product. The version number of Creo Parametric 3.0 is 32.

The following points summarize the rules for mixing Creo Object TOOLKIT C++ and Creo Parametric versions:

- Creo Parametric release newer than a Creo Object TOOLKIT C++ release:

  This works in many, but not all, cases. The communication method used to link Creo Object TOOLKIT C++ to Creo Parametric provides full compatibility between releases. However, there are occasional cases where changes internal to Creo Parametric may require changes to the source code of a Creo Object TOOLKIT C++ application in order that it continues to work correctly. Whether you need to convert Creo Object TOOLKIT C++ applications depends on what functionality it uses and what functionality changed in Creo Parametric and Creo Object TOOLKIT C++. PTC makes every effort to keep these effects to a minimum. The Release Notes for Creo Object TOOLKIT C++ detail any conversion work that could be necessary for that release.
- Creo Parametric build newer than Creo Object TOOLKIT C++ build

  This is always supported.

Retrieving Creo Datecode

Method Introduced:

- **wfcWSession::GetDisplayDateCode**

The method `wfcWSession::GetDisplayDateCode` returns the user-visible datecode string from the Creo application. The applications that present a datecode string to users in messages and information should use the Creo datecode format.

Compatibility of Deprecated Methods

In a release cycle, some methods are deprecated, and new methods are added. The deprecated methods are supported in the current release, and then obsoleted in a future release. You can either choose to let the deprecated methods work in the current release, or allow only new methods to work. Use the methods explained in this section along with the compatibility value to work with either deprecated or new methods for the current release.

Methods Introduced:

- **pfcAppInfo::GetCompatibility**

- **pfcAppInfo::SetCompatibility**
- **pfcAppInfo::Create**
- **pfcSession::GetAppInfo**
- **pfcSession::SetAppInfo**

The methods `pfcAppInfo::GetCompatibility` and `pfcAppInfo::SetCompatibility` get and set the compatibility value for the specified application using the enumerated data type `pfcCreoCompatibility`. The valid values are:

- `pfcCompatibilityUndefined`—Specifies that compatibility value is not set. The default compatibility value is used.
- `pfcC3Compatible`—Specifies that the methods deprecated in Creo Object TOOLKIT C++ 4.0 are compatible and continue working in Creo Object TOOLKIT C++ 4.0. By default the compatibility is set to `pfcC3Compatible`.
- `pfcC4Compatible`—Specifies that the methods deprecated in Creo Object TOOLKIT C++ 4.0 will not work in Creo Object TOOLKIT C++ 4.0. If your application uses the deprecated methods, you must replace these methods with new methods and rebuild you applications.

Note

If you rebuild or run Creo Object TOOLKIT C++ applications from previous releases in the current release, the compatibility is set `pfcC3Compatible` to current release. For example, if you rebuild a Creo Object TOOLKIT C++ 3.0 application in release 4.0, the compatibility is set to `pfcC3Compatible`. To set the compatibility to the current release, use the method `pfcAppInfo::SetCompatibility`.

The method `pfcAppInfo::Create` creates a new instance of the `pfcAppInfo` object.

The methods `pfcSession::GetAppInfo` and `pfcSession::SetAppInfo` get and set the information for an application in terms of their compatibility value as a `pfcAppInfo` object.

Visit Methods

Methods Introduced:

- **wfcVisitingClient::ApplyAction**
- **wfcVisitingClient::ApplyFilter**

- **wfcWAssembly::VisitComponents**
- **wfcWFeatureGroup::VisitDimensions**
- **wfcWModel::VisitItems**
- **wfcWFeature::VisitItems**
- **wfcWModel::VisitDetailItems**

In a Creo Object TOOLKIT C++ application, you may often want to perform an operation on all the objects that belong to another object, such as all the features in a part, or all the surfaces in a feature. For such cases, Creo Object TOOLKIT C++ provides an appropriate visit method. The visit method is an alternative to passing back an array of data.

The method `wfcWAssembly::VisitComponents` visits all the components in an assembly. The input argument *visitingClient* specifies an object of type `wfcVisitingClient` that contains the visit action and filter methods for visiting items.

The method `wfcVisitingClient::ApplyAction` is the method that you want to be called for each item and pass its pointer to the Creo Object TOOLKIT C++ visit method. This method is referred to as the visit action method. The Creo Object TOOLKIT C++ visit method calls the visit action method once for every visited item.

The method `wfcVisitingClient::ApplyFilter` is referred to as the filter method. The filter method is called for each visited item before the action method. The return value of the filter method controls whether the action method must be called for that item. You can use the filter method as a way of visiting only a particular subset of the items in the list.

The filter method must return one of the following values defined in enumerated data type `wfcStatus`:

- `wfcTK_CONTINUE`—Specifies that the visit action method must not visit this object, but continue to visit the subsequent objects.
- Any other value—Specifies that the visit action method must be called for this object. The return value must be passed as the input argument to the visit action method.

The visit action method must return one of the following values defined in enumerated data type `wfcStatus`:

- `wfcTK_NO_ERROR`—Specifies that the visit action method must continue visiting the other objects in the list.
- `wfcTK_E_NOT_FOUND`—Specifies that no items of the specified type were found and therefore no objects could be visited.

- Any other value including `wfcTK_CONTINUE`—Terminates the visit. Typically this status is returned from the visit action method on termination, so that the calling method knows the reason for the abnormal termination of the visit.

For the method `wfcWSolid::VisitItems`, the actual type of item passed to the methods `wfcVisitingClient::ApplyAction` and `wfcVisitingClient::ApplyFilter` is of type `wfcWModelItem`. To make the methods of `wfcWModelItem` available in `pfcModelItem`, cast `pfcModelItem` to `wfcWModelItem`.

Similarly, for the method `wfcWAssembly::VisitComponents`, to make the methods of `wfcWComponentFeat` available in `pfcObject`, cast `pfcObject` to `wfcWComponentFeat`.

If you are using Creo Object TOOLKIT C++ together with Creo TOOLKIT, use the utility method `wfcStatusFromPro` to convert the `ProError` to equivalent Creo Object TOOLKIT C++ `wfcStatus`. Use the utility method `wfcStatusToPro` to convert a Creo Object TOOLKIT C++ `wfcStatus` to an equivalent `ProError`.

The method `wfcWFeatureGroup::VisitDimensions` traverses the members of the feature group.

The method `wfcWModel::VisitItems` visits the `pfcModelItemType` objects in the model for the specified type of item only if the model has a single body, else returns the error `wfcTK_MULTIBODY_UNSUPPORTED`.

The method `wfcWFeature::VisitItems` visits the annotation elements in the specified feature.

The method `wfcWModel::VisitDetailItems` visits the `pfcDetailType` objects in the model for the specified drawing and sheet of a detail item.

2

The Creo Object TOOLKIT C++ Online Browser

This chapter describes how to use the online browser provided with Creo Object TOOLKIT C++.

Online Documentation for Creo Object TOOLKIT C++ APIWizard

Creo Object TOOLKIT C++ provides an online browser called the Creo Object TOOLKIT C++ APIWizard that displays detailed documentation. This browser displays information from the *Creo Object TOOLKIT C++ User's Guide* and API specifications derived from Creo Object TOOLKIT C++ header file data.

The Creo Object TOOLKIT C++ APIWizard contains the following items:

- Definitions of Creo Object TOOLKIT C++ libraries
- Definitions of Creo Object TOOLKIT C++ classes and their hierarchical relationships
- Descriptions of Creo Object TOOLKIT C++ methods
- Declarations of data types used by Creo Object TOOLKIT C++ methods
- The *Creo Object TOOLKIT C++ User's Guide*, which you can browse by topic or by class
- Code examples for Creo Object TOOLKIT C++ methods (taken from the sample applications provided as part of the Creo Object TOOLKIT C++ installation)

Read the Release Notes and README file for the most up-to-date information on documentation changes.

Note

- The *Creo Object TOOLKIT C++ User's Guide* is also available in PDF format. This file is located at:
  `<creo_otk_loadpoint_doc>\otkug.pdf`
- From Creo 4.0 F000, the applet based APIWizard is no longer supported. Use the non-applet based APIWizard instead.

Installing the APIWizard

The Creo Object TOOLKIT C++ installation procedure automatically installs the Creo Object TOOLKIT C++ APIWizard. The files reside in a directory under the Creo Object TOOLKIT C++ load point. The location for the Creo Object TOOLKIT C++ APIWizard files is:
`<creo_otk_loadpoint_doc>\objecttoolkit_Creo`

APIWizard Overview

The APIWizard supports Internet Explorer, Firefox, and Chromium browsers.

Start the Creo Object TOOLKIT C++ APIWizard by pointing your browser to:
`<creo_otk_loadpoint_doc>\objecttoolkit_Creo\index.html`

A page containing links to the Creo Object TOOLKIT C++ APIWizard and User's Guide will open in the web browser.

Non-Applet APIWizard Top Page

The top page of non-applet based APIWizard has links to the Creo Object TOOLKIT C++ APIWizard and User's Guide. The APIWizard opens an HTML page that contains links to Creo Object TOOLKIT C++ classes and related methods. The User's Guide opens an HTML page that displays the Table of Contents of the User's Guide, with links to the chapters, and sections under the chapters.

Click APIWizard to open the list of Creo Object TOOLKIT C++ classes and related methods. Click a class or method name to read more about it.

You can search for specified information in the APIWizard. Use the search field at the top left pane to search for methods. You can search for information using the following criteria:

- Search by method names
- Search using wildcard character *, where * (asterisk) matches zero or more nonwhite space characters

The resulting method names are displayed in a drop down list with links to html pages.

You can also hover the mouse over  after you enter a string in the search field. The following search options are displayed:

- **Class/Methods**—Searches for classes and methods.
- **Global Methods**—Searches only for global methods.
- **Compact Class/Methods**—Reserved for future use.
- **Licensed Methods**—Searches for methods which are available under license 222. In the search field, specify "*". It displays all the methods under this license.
- **Exceptions**—Searches only for exceptions.
- **Enumeration**—Searches only for enumerations.

Select an option and the search results are displayed based on this criteria.

User's Guide

Click User's Guide to access the *Creo Object TOOLKIT C++ User's Guide*.

3

Session Objects

This chapter describes how to program on the session level using Creo Object TOOLKIT C++.

Overview of Session Objects

The `Session` object, contained in the class `pfcSession`, is the highest level object in Creo Object TOOLKIT C++. Any program that accesses data from Creo must first get a handle to the `Session` object before accessing more specific data.

The `Session` object contains methods to perform the following operations:

- Accessing models and windows (described in the Models and Windows chapters).
- Working with the Creo user interface.
- Allowing interactive selection of items within the session.
- Accessing global settings such as line styles, colors, and configuration options.

The following sections describe these operations in detail.

Getting the Session Object

Method Introduced:

- **pfcGetCurrentSession**
- **pfcGetCurrentSessionWithCompatibility**
- **pfcGetProESession**

For every application, Creo assigns a unique session. The session contains license information, the compatibility information as a `pfcCreoCompatibility` object, and other additional data of the application. When a session is assigned to an application, Creo sets the compatibility to `pfcCompatibilityUndefined` in the associated `pfcAppInfo` object. You must set the compatibility of the application before working with sessions. To set the compatibility, call the method `pfcGetCurrentSessionWithCompatibility`. Use the values defined in the enumerated data type `pfcCreoCompatibility` to set the compatibility of the application. See Compatibility of Deprecated Methods on page 45 for more information on compatibility and `pfcAppInfo` object. Use the method `pfcGetCurrentSession` to get the current `pfcSession` object in synchronous mode. If the compatibility is not set, the method throws the exception `pfcXCompatibilityNotSet`.

The method `pfcGetProESession` also gets the `Session` object. This method will be deprecated in a future release of Creo. If you call this method without setting the compatibility, the method sets the compatibility to `C3Compatible`. This setting ensures forward compatibility of the Creo applications. If you set a specific compatibility using the method

pfcGetCurrentSessionWithCompatibility, and call the method pfcGetProESession() then all calls to pfcGetProESession return the session with the set compatibility.

Note

You can make multiple calls to this method, but each call gives you a handle to the same object.

Getting Session Information

Methods Introduced:

- **pfcGetProEArguments**
- **pfcGetProEVersion**
- **pfcGetProEBuildCode**

The method pfcGetProEArguments returns an array containing the command line arguments passed to Creo if these arguments follow one of two formats:

- Any argument starting with a plus sign (+) followed by a letter character.
- Any argument starting with a minus (-) followed by a capitalized letter.

The first argument passed in the array is the full path to the Creo executable.

The method pfcGetProEVersion returns a string that represents the Creo version.

The method pfcGetProEBuildCode returns a string that represents the build code of the Creo session.

Note

The preceding methods can only access information in synchronous mode.

Creo License Data

Method Introduced:

- **wfcWSession::IsOptionOrdered**

The method wfcWSession::IsOptionOrdered returns a boolean value indicating if the specified Creo license option is currently available in the Creo session. For example, Pro/MESH option.

Directories

Methods Introduced:

- **pfcBaseSession::GetCurrentDirectory**
- **pfcBaseSession::ChangeDirectory**

The method `pfcBaseSession::GetCurrentDirectory` returns the absolute path name for the current working directory of Creo application.

The method `pfcBaseSession::ChangeDirectory` changes Creo to another working directory.

File Handling

Methods Introduced:

- **pfcBaseSession::ListFiles**
- **pfcBaseSession::ListSubdirectories**
- **wfcWSession::UIEditFile**
- **wfcWSession::ParseFileName**
- **wfcParsedFileNameData::GetDirectoryPath**
- **wfcParsedFileNameData::GetName**
- **wfcParsedFileNameData::GetExtension**
- **wfcParsedFileNameData::GetVersion**
- **wfcWSession::DisplayInformationWindow**

The method `pfcBaseSession::ListFiles` returns a list of files in a directory, given the directory path. You can filter the list to include only files of a particular type, as specified by the file extension. Use the `FILE_LIST_ALL` option to include all versions of a file in the list; use `FILE_LIST_LATEST` to include only the latest version.

The method `pfcBaseSession::ListFiles` lists the instance objects when accessing Windchill workspaces or folders. A PDM location (for workspace or commonspace) must be passed as the directory path. The following options have been added in the `pfcFileListOpt` enumerated type:

- `pfcFILE_LIST_ALL`—Lists all the files. It may also include multiple versions of the same file.
- `pfcFILE_LIST_LATEST`—Lists only the latest version of each file.
- `pfcFILE_LIST_ALL_INST`—Same as the `pfcFILE_LIST_ALL` option. It returns instances only for PDM locations.
- `pfcFILE_LIST_LATEST_INST`—Same as the `pfcFILE_LIST_LATEST` option. It returns instances only for PDM locations.

The method `pfcBaseSession::ListSubdirectories` returns the subdirectories in a given directory location.

The method `wfcWSession::UIEditFile` opens an edit window for the specified text file. The editor used is the default editor for Creo application. The method returns a boolean value to indicate if the file was edited.

The file utility methods refer to files using a single wide character string, which composes of the directory path, file name, extension, and version. The method `wfcWSession::ParseFileName` takes such a string as input, and returns the four segments as a `wfcParsedFileNameData` object.

The method `wfcParsedFileNameData::GetDirectoryPath` returns the directory path for the file.

The method `wfcParsedFileNameData::GetName` returns the name of the file.

The method `wfcParsedFileNameData::GetExtension` returns the extension of the file.

The method `wfcParsedFileNameData::GetVersion` returns the version of the file.

The method `wfcWSession::DisplayInformationWindow` creates a window and displays the content of the specified file. The input arguments are:

- *FilePath*—Specifies the name of the file.
- *XOffest*—Specifies the location of the window along the X-direction in reference to the Creo main window. The valid range is from 0.0 to 1.0.
- *YOffest*—Specifies the location of the window along the Y-direction in reference to the Creo main window. The valid range is from 0.0 to 1.0.
- *Rows*—Specifies the size of the window in terms of rows. The valid range is from 6 to 33.
- *Columns*—Specifies the size of the window in terms of columns. The valid range is from 8 to 80.

Configuration Options

Methods Introduced:

- **pfcBaseSession::GetConfigOptionValues**
- **pfcBaseSession::SetConfigOption**
- **pfcBaseSession::LoadConfigFile**

You can access configuration options programmatically using the methods described in this section.

Use the method `pfcBaseSession::GetConfigOptionValues` to retrieve the value of a specified configuration file option. Pass the *Name* of the configuration file option as the input to this method. The method returns an array of values that the configuration file option is set to. It returns a single value if the configuration file option is not a multi-valued option. The method returns a null if the specified configuration file option does not exist.

The method `pfcBaseSession::SetConfigOption` is used to set the value of a specified configuration file option. If the option is a multi-value option, it adds a new value to the array of values that already exist.

The method `pfcBaseSession::LoadConfigFile` loads an entire configuration file into Creo application.

Registry File Data

Functions Introduced:

- **wfcWSession::GetApplicationPath**
- **wfcWSession::GetApplicationTextPath**

The method `wfcWSession::GetApplicationPath` returns the path to the Creo Object TOOLKIT C++ executable file, `exec_file`, from the registry file.

The method `wfcWSession::GetApplicationTextPath` returns the path to the directory containing the `text` folder for the application from the registry file.

Macros

Method Introduced:

- **pfcBaseSession::RunMacro**
- **wfcWSession::ExecuteMacro**

The method `pfcBaseSession::RunMacro` runs a macro string. A Creo Object TOOLKIT C++ macro string is equivalent to a Creo Parametric mapkey minus the key sequence and the mapkey name. To generate a macro string, create a mapkey in Creo Parametric. Refer to the Creo Parametric online help for more information about creating a mapkey.

Copy the `Value` of the generated mapkey Option from the **Tools ▸ Options** dialog box. An example `Value` is as follows:

```
$F2 @MAPKEY_LABELtest;
~ Activate `main_dlg_cur` `ProCmdModelNew.file`;
~ Activate `new` `OK`;
```

The key sequence is `$F2`. The mapkey name is `@MAPKEY_LABELtest`. The remainder of the string following the first semicolon is the macro string that should be passed to the method `pfcBaseSession::RunMacro`.

In this case, it is as follows:

```
~ Activate `main_dlg_cur` `ProCmdModelNew.file`;
~ Activate `new` `OK`;
```

Note

Creating or editing the macro string manually is not supported as the mapkeys are not a supported scripting language. The syntax is not defined for users and is not guaranteed to remain constant across different datecodes of Creo Parametric.

The method `wfcWSession::ExecuteMacro` executes the macros previously loaded using the method `pfcBaseSession::RunMacro`.

Execution Rules

Consider the following rules about the execution of macros:

- Macros are executed in asynchronous mode as soon as they are registered. Macros in asynchronous mode are run in the same order in which they were saved.
- In synchronous mode, the mapkey or the macro strings are pushed onto a stack and are popped off and executed only when control returns to Creo Parametric from the Creo Object TOOLKIT C++ program. Macros in synchronous mode are stored in reverse order, last in, first out. Due to the last in, first out nature of the stack, macros that cannot be passed entirely in one `pfcBaseSession::RunMacro` call should have the strings loaded in reverse order of required execution.
- To execute a macro from within Creo Object TOOLKIT C++, call the function `wfcWSession::ExecuteMacro`. The method runs the Creo Parametric macro and returns the control to the Creo Object TOOLKIT C++ application. The function works only in the synchronous mode.
- Do not call the function `wfcWSession::ExecuteMacro` during the following operations:
  - Activating dialog boxes or setting the current model
  - Erasing the current model
  - Replaying a trail file
- While executing macros, if you click **OK** in the dialog box to complete the command operation, the dialog box may be displayed momentarily without completing the command operation.

Note

- You can execute only the dialog boxes with built-in exit confirmation as macros, by canceling the exit action.
- It is possible that a macro may not be executed because a command specified in the macro is currently inaccessible in the menus. The functional success of `wfcWSession::ExecuteMacro` depends on the priority of the executed command against the current context.

- If some of the commands require input to be specified from the keyboard (such as a part name), the macro continues execution after you type the input and press `ENTER`. However, if you must select something with the mouse (such as selecting a sketching plane), the macro is interrupted and ignores the remaining commands in the string.

  This allows the application to create object-independent macros for long sequences of repeating choices. Note that during execution of the macro the user does not have to select any geometry.

Colors and Line Styles

Methods Introduced:

- **pfcBaseSession::SetStdColorFromRGB**
- **pfcBaseSession::GetRGBFromStdColor**
- **pfcBaseSession::SetTextColor**
- **pfcBaseSession::SetLineStyle**

These methods control the general display of a Creo session.

Use the method `pfcBaseSession::SetStdColorFromRGB` to customize any of the Creo standard colors.

To change the color of any text in the window, use the method `pfcBaseSession::SetTextColor`.

To change the appearance of nonsolid lines (for example, datums) use the method `pfcBaseSession::SetLineStyle`.

Initializing Objects

The helper methods described in this section allows you to initialize session objects.

Methods Introduced:

- **wfcCreateMatrix3D**
- **wfcCreatePoint2D**
- **wfcCreatePoint3D**
- **wfcCreateOutline2D**
- **wfcCreateOutline3D**
- **wfcCreateVector2D**
- **wfcCreateVector3D**

The method `wfcCreateMatrix3D` initializes a three-dimensional matrix with the specified values.

Use the methods `wfcCreatePoint2D` and `wfcCreatePoint3D` to initialize a two-dimensional and three-dimensional point respectively with the specified values.

The methods `wfcCreateOutline2D` and `wfcCreateOutline3D` initialize a two-dimensional and three-dimensional line respectively with the specified values.

Use the methods `wfcCreateVector2D` and `wfcCreateVector3D` to initialize a two-dimensional and three-dimensional vector respectively with the specified values.

Accessing the Creo User Interface

The `Session` object has methods that work with the Creo user interface. These methods provide access to the menu bar and message window. For more information on accessing menus, refer to the chapter Menus, Commands, and Pop-up Menus on page 90.

The Text Message File

A text message file is where you define strings that are displayed in the Creo user interface. This includes the strings on the command buttons that you add to the Creo number, the help string that is displayed when the cursor is positioned over such a command button, and text strings that you display in the Message Window. You have the option of including a translation for each string in the text message file.

Restrictions on the Text Message File

You must observe the following restrictions when you name your message file:

- The name of the file must be 30 characters or less, including the extension.

- The name of the file must contain lower case characters only.
- The file extension must be three characters.
- The version number must be in the range 1 to 9999.
- All message file names must be unique, and all message key strings must be unique across all applications that run with Creo application. Duplicate message file names or message key strings can cause Creo application to exhibit unexpected behavior. To avoid conflicts with the names of Creo or foreign application message files or message key strings, PTC recommends that you choose a prefix unique to your application, and prepend that prefix to each message file name and each message key string corresponding to that application

Note

Message files are loaded into Creo application only once during a session. If you make a change to the message file while Creo is running you must exit and restart Creo application before the change will take effect.

Contents of the Message File

The message file consists of groups of four lines, one group for each message you want to write. The four lines are as follows:

1. A string that acts as the identifier for the message. This keyword must be unique for all Creo messages.
2. The string that will be substituted for the identifier.

   This string can include placeholders for run-time information stored in a `stringseq` object (shown in Writing Messages to the Message Window).
3. The translation of the message into another language (can be blank).
4. An intentionally blank line reserved for future extensions.

Writing a Message Using a Message Pop-up Dialog Box

Method Introduced:

- **pfcSession::UIShowMessageDialog**

The method `pfcSession::UIShowMessageDialog` displays the UI message dialog. The input arguments to the method are:

- *Message*—The message text to be displayed in the dialog.

- *Options*—An instance of the `pfcMessageDialogOptions` containing other options for the resulting displayed message. If this is not supplied, the dialog will show a default message dialog with an **Info** classification and an **OK** button. If this is not to be null, create an instance of this options type with `pfcMessageDialogOptions::Create()`. You can set the following options:
  - `Buttons`—Specifies an array of buttons to include in the dialog. If not supplied, the dialog will include only the **OK** button. Use the method `pfcMessageDialogOptions::SetButtons` to set this option.
  - `DefaultButton`—Specifies the identifier of the default button for the dialog box. This must match one of the available buttons. Use the method `pfcMessageDialogOptions::SetDefaultButton` to set this option.
  - `DialogLabel`—The text to display as the title of the dialog box. If not supplied, the label will be the english string `Info`. Use the method `pfcMessageDialogOptions::SetDialogLabel` to set this option.
  - `MessageDialogType`—The type of icon to be displayed with the dialog box (**Info**, **Prompt**, **Warning**, or **Error**). If not supplied, an **Info** icon is used. Use the method `pfcMessageDialogOptions::SetMessageDialogType` to set this option.

Accessing the Message Window

The following sections describe how to access the message window using Creo Object TOOLKIT C++. The topics are as follows:

- Writing Messages to the Message Window on page 63
- Writing Messages to an Internal Buffer on page 64

Writing Messages to the Message Window

Methods Introduced:

- **pfcSession::UIDisplayMessage**
- **pfcSession::UIDisplayLocalizedMessage**
- **pfcSession::UIClearMessage**

These methods enable you to display program information on the screen.

The input arguments to the methods `pfcSession::UIDisplayMessage` and `pfcSession::UIDisplayLocalizedMessage` include the names of the message file, a message identifier, and (optionally) a `stringseq` object that

contains upto 10 pieces of run-time information. For `pfcSession::UIDisplayMessage`, the strings in the `stringseq` are identified as`%0s`, `%1s`, ... `%9s` based on their location in the sequence. For `pfcSession::UIDisplayLocalizedMessage`, the strings in the `stringseq` are identified as`%0w`, `%1w`, ... `%9w` based on their location in the sequence. To include other types of run-time data (such as integers or reals) you must first convert the data to strings and store it in the string sequence.

Writing Messages to an Internal Buffer

Methods Introduced:

- **pfcBaseSession::GetMessageContents**
- **pfcBaseSession::GetLocalizedMessageContents**

The methods `pfcBaseSession::GetMessageContents` and `pfcBaseSession::GetLocalizedMessageContents` enable you to write a message to an internal buffer instead of the Creo message area.

These methods take the same input arguments and perform exactly the same argument substitution and translation as the `pfcSession::UIDisplayMessage` and `pfcSession::UIDisplayLocalizedMessage` methods described in the previous section.

Message Classification

Messages displayed in Creo Object TOOLKIT C++ include a symbol that identifies the message type. Every message type is identified by a classification that begins with the characters `%C`. A message classification requires that the message key line (line one in the message file) must be preceded by the classification code.

Note

Any message key string used in the code should not contain the classification.

Creo Object TOOLKIT C++ applications can now display any or all of the following message symbols:

- `Prompt`—This Creo Object TOOLKIT C++ message is preceded by a green arrow. The user must respond to this message type. Responding includes, specifying input information, accepting the default value offered, or canceling the application. If no action is taken, the progress of the application is halted. A response may either be textual or a selection. The classification for Prompt messages is `%CP`

- `Info`—This Creo Object TOOLKIT C++ message is preceded by a blue dot. Info message types contain information such as user requests or feedback from Creo Object TOOLKIT C++ or Creo applications. The classification for Info messages is `%CI`

> **Note**
>
> Do not classify messages that display information regarding problems with an operation or process as `Info`. These types of messages must be classified as `Warnings`.

- `Warning`—This Creo Object TOOLKIT C++ message is preceded by a triangle containing an exclamation point. Warning message types contain information to alert users to situations that could potentially lead to an error during a later stage of the process. Examples of warnings could be a process restriction or a suspected data problem. A Warning will not prevent or interrupt a process. Also, a Warning should not be used to indicate a failed operation. Warnings must only caution a user that the completed operation may not have been performed in a completely desirable way. The classification for Warning messages is `%CW`
- `Error`—This Creo Object TOOLKIT C++ message is preceded by a broken square. An Error message informs the user that a required task was not completed successfully. Depending on the application, a failed task may or may not require intervention or correction before work can continue. Whenever possible redress this situation by providing a path. The classification for Error messages is `%CE`
- `Critical`—This Creo Object TOOLKIT C++ message is preceded by a red X. A Critical message type informs the user of an extremely serious situation that is usually preceded by loss of user data. Options redressing this situation, if available, should be provided within the message. The classification for a Critical message is `%CC`.

Reading Data from the Message Window

Methods Introduced:

- **pfcSession::UIReadIntMessage**
- **pfcSession::UIReadRealMessage**
- **pfcSession::UIReadStringMessage**

These methods enable a program to get data from the user. The methods obtain keyboard input from a text box in the Creo Parametric user interface.

Note

When the user presses Esc or clicks **Cancel** in the Creo Parametric user interface, these methods throw the exception `pfcXToolkitMsgUserQuit`.

The `pfcSession::UIReadIntMessage` and `pfcSession::Session.UIReadRealMessage` methods contain optional arguments that can be used to limit the value of the data to a certain range.

The method `pfcSession::UIReadStringMessage` includes an optional Boolean argument that specifies whether to echo characters entered onto the screen. You would use this argument when prompting a user to enter a password.

Note

A default value is displayed in the text box as input. When user presses the Enter key as input in the user interface, the default value is not passed to the Creo Object TOOLKIT C++ method; instead, the method returns a constant string `use_default_string`. When this string is returned, the application must interpret that the user wants to use the default value.

Displaying Feature Parameters

Method Introduced:

- **pfcSession::UIDisplayFeatureParams**

The method `pfcSession::UIDisplayFeatureParams` allows Creo application to show dimensions or other parameters stored on a specific feature. The displayed dimensions may then be interactively selected by the user.

File Dialogs

Methods Introduced:

- **pfcSession::UIOpenFile**
- **pfcFileOpenOptions::Create**
- **pfcFileOpenOptions::SetFilterString**
- **pfcFileOpenOptions::SetPreselectedItem**
- **pfcFileUIOptions::SetDefaultPath**

- **pfcFileUIOptions::SetDialogLabel**
- **pfcFileUIOptions::SetShortcuts**
- **pfcFileOpenShortcut::Create**
- **pfcFileOpenShortcut::SetShortcutName**
- **pfcFileOpenShortcut::SetShortcutPath**
- **wfcWSession::UIOpenFileType**
- **wfcFiletypeOpenOptions::Create**
- **wfcFiletypeOpenOptions::GetModelFiletypes**
- **wfcFiletypeOpenOptions::SetModelFiletypes**
- **wfcFiletypeOpenOptions::GetPreselectedItem**
- **wfcFiletypeOpenOptions::SetPreselectedItem**
- **pfcSession::UISaveFile**
- **pfcFileSaveOptions::Create**
- **wfcWSession::UISaveFileType**
- **wfcFiletypeSaveOptions::Create**
- **wfcFiletypeSaveOptions::GetModelFiletypes**
- **wfcFiletypeSaveOptions::SetModelFiletypes**
- **wfcFiletypeSaveOptions::GetPreselectedItem**
- **wfcFiletypeSaveOptions::SetPreselectedItem**
- **pfcSession::UISelectDirectory**
- **pfcDirectorySelectionOptions::Create**
- **pfcBaseSession::UIRegisterFileOpen**
- **pfcFileOpenRegisterOptions::Create**
- **pfcFileOpenRegisterOptions::SetFileDescription**
- **pfcFileOpenRegisterOptions::SetFileType**
- **pfcFileOpenRegisterListener::FileOpenAccess**
- **pfcFileOpenRegisterListener::OnFileOpenRegister**
- **pfcBaseSession::UIRegisterFileSave**
- **pfcFileSaveRegisterOptions::Create**
- **pfcFileSaveRegisterOptions::SetFileDescription**
- **pfcFileSaveRegisterOptions::SetFileType**
- **pfcFileSaveRegisterListener::FileSaveAccess**
- **pfcFileSaveRegisterListener::OnFileSaveRegister**

The method pfcSession::UIOpenFile opens the dialog box to browse directories and open files. The method lets you specify options for the file open dialog box using the objects pfcFileOpenOptions and pfcFileUIOptions.

Use the method pfcFileOpenOptions::Create to create a new instance of the pfcFileOpenOptions object. You can specify a filter string to include only files of a particular type. The filter string is specified using file extension. In the input argument FilterString you can specify all types of files extensions with wildcards separated by commas, for example, *.prt, *.asm, *.txt, *.avi, and so on. Use the methods pfcFileOpenOptions::GetFilterString and pfcFileOpenOptions::SetFilterString to get and set the types of file extensions.

Use the methods pfcFileOpenOptions::GetPreselectedItem and pfcFileOpenOptions::SetPreselectedItem to get and set the name of an item that must be preselected in the dialog box.

The pfcUI::FileUIOptions object contains the following options:

- DefaultPath—Specifies the name of the path to be opened by default in the dialog box. Use the method pfcFileUIOptions::SetDefaultPath to set this option.
- DialogLabel—Specifies the title of the dialog box. Use the method pfcFileUIOptions::SetDialogLabel to set this option.
- Shortcuts—Specifies an array of file shortcuts of the type pfcFileOpenShortcut. Create this object using the method pfcFileOpenShortcut::Create. This object contains the following attributes:
  - ShortcutName—Specifies the name of shortcut path to be made available in the dialog box.
  - ShortcutPath—Specifies the string for the shortcut path.

  Use the method pfcFileUIOptions::SetShortcuts to set the array of file shortcuts.

The method wfcWSession::UIOpenFileType opens the dialog box to browse directories and open files. The method lets you specify options for the file open dialog box using the objects wfcFiletypeOpenOptions and pfcFileUIOptions.

Use the method wfcFiletypeOpenOptions::Create to create a new instance of wfcFiletypeOpenOptions object. You can specify a filter string to include only files of a particular type in the dialog box. In the input argument ModelFiletypes, you can specify an array of file types using the enumerated data type wfcMdlfileType. Use the methods

wfcFiletypeOpenOptions::GetModelFiletypes and wfcFiletypeOpenOptions::SetModelFiletypes to read and set the file types. Use the methods wfcFiletypeOpenOptions::GetPreselectedItem and wfcFiletypeOpenOptions::GetPreselectedItem to get and set the name of an item that must be preselected in the dialog box.

Note

The methods pfcSession::UIOpenFile and wfcWSession::UIOpenFileType, do not actually open the file, but return the file path of the selected file.

The method pfcSession::UISaveFile opens the save dialog box. The method accepts options similar to pfcSession::UIOpenFile through the pfcFileSaveOptions and pfcFileUIOptions objects. Use the method pfcFileSaveOptions::Create to create a new instance of the pfcFileSaveOptions object. When using the **Save** dialog box, you can set the name to a non-existent file.

The method wfcWSession::UISaveFileType opens the save dialog box. The method accepts options similar to wfcWSession::UIOpenFileType. The method lets you specify options for the save dialog box using the objects wfcFiletypeSaveOptions and pfcFileUIOptions. Use the method wfcFiletypeSaveOptions::Create to create a new instance of the wfcFiletypeSaveOptions object. You can specify a filter string to include only files of a particular type. In the input argument ModelFiletypes, you can use the specify an array of file types using the enumerated data type wfcMdlfileType. Use the methods wfcFiletypeSaveOptions::GetModelFiletypes and wfcFiletypeSaveOptions::SetModelFiletypes to read and set the file types. Use the methods wfcFiletypeSaveOptions::GetPreselectedItem and wfcFiletypeSaveOptions::GetPreselectedItem to get and set the name of an item that must be preselected in the dialog box.

Note

- The methods `pfcSession::UISaveFile` and `wfcWSession::UISaveFileType`, do not actually save the file, but return the file path of the selected file.
- For multi-CAD models, in a linked session of Creo Parametric with Windchill, the methods `pfcSession::UISaveFile` and `wfcWSession::UISaveFileType` do not support a file path location on local disk.

The method `pfcSession::UISelectDirectory` prompts the user to select a directory using the Creo dialog box for browsing directories. The method accepts options through the `pfcDirectorySelectionOptions` object which is similar to the `pfcFileUIOptions` object (described for the method `pfcSession::UIOpenFile`). Specify the default directory path, the title of the dialog box, and a set of shortcuts to other directories to start browsing. If the default path is specified as `NULL`, the current directory is used. Use the method `pfcDirectorySelectionOptions::Create` to create a new instance of the `pfcDirectorySelectionOptions` object. The method `pfcSession::UISelectDirectory` returns the selected directory path; the application must use other methods or techniques to perform other relevant tasks with this selected path.

The method `pfcBaseSession::UIRegisterFileOpen` registers a new file type in the **File ▸ Open** dialog box in Creo application. This method takes the `pfcFileOpenRegisterOptions` and `pfcFileOpenRegisterListener` objects as its input arguments. These objects are as follows:

- `pfcFileOpenRegisterOptions`—This object contains the options for registering an open operation. Use the method `pfcFileOpenRegisterOptions::Create` to create a new instance of the object. It contains the following options:
  - `FileDescription`—Specifies the short description of the file type to be opened. This description appears for the file type in the **File ▸ Open** dialog box. Use the method `pfcFileOpenRegisterOptions::SetFileDescription` to modify this option.
  - `FileType`—Specifies the file type to be opened. The file type appears as the file extension in the **File ▸ Open** dialog box. Use the method `pfcFileOpenRegisterOptions::SetFileType` to modify this option.

- `pfcFileOpenRegisterListener`—This object provides the action listener methods for the new file type to be registered. The method `pfcFileOpenRegisterListener::FileOpenAccess` is called to determine whether the new file type can be opened using the **File ▸ Open** dialog box. The method `pfcFileOpenRegisterListener::OnFileOpenRegister` is called on clicking **Open** for the newly registered file type.

The method `pfcBaseSession::UIRegisterFileSave` registers a new file type in the **Save a Copy** dialog box in Creo application. This method takes the `FileSaveRegisterOptions` and `pfcFileSaveRegisterListener` objects as its input arguments. These objects are described as follows:

- `pfcFileSaveRegisterOptions`—This object contains the options for registering a save operation. Use the method `pfcFileSaveRegisterOptions::Create` to create a new instance of the object. It contains the following options:
  - `FileDescription`—Specifies the short description of the file type to be saved. This description appears for the file type in the **Save a Copy** dialog box. Use the method `pfcFileSaveRegisterOptions::SetFileDescription` to modify this option.
  - `FileType`—Specifies the file type to be saved. The file type appears as the file extension in the **Save a Copy** dialog box. Use the method `pfcFileSaveRegisterOptions::SetFileType` to modify this option.
- `pfcFileSaveRegisterListener`—This object provides the action listener methods for the new file type to be registered. The method `pfcFileSaveRegisterListener::FileSaveAccess` is called to determine whether the new file type can be saved using the **Save a Copy** dialog box. The method `pfcFileSaveRegisterListener::OnFileSaveRegister` is called on clicking **OK** for the newly registered file type.

Customizing the Creo Navigation Area

The Creo navigation area includes the Model and Layer Tree pane, Folder browser pane, and Favorites pane. The methods described in this section enable Creo Object TOOLKIT C++ applications to add custom panes that contain Web pages to the Creo navigation area.

Adding Custom Web Pages

To add custom Web pages to the navigation area, the Creo Object TOOLKIT C++ application must:

1. Add a new pane to the navigation area.
2. Set an icon for this pane.
3. Set the URL of the location that will be displayed in the pane.

Methods Introduced:

- **pfcSession::NavigatorPaneBrowserAdd**
- **pfcSession::NavigatorPaneBrowserIconSet**
- **pfcSession::NavigatorPaneBrowserURLSet**

The method `pfcSession::NavigatorPaneBrowserAdd` adds a new pane that can display a Web page to the navigation area. The input parameters are:

- *PaneName*—Specify a unique name for the pane. Use this name in susbsequent calls to `pfcSession::NavigatorPaneBrowserIconSet` and `pfcSession::NavigatorPaneBrowserURLSet`.
- *IconFileName*—Specify the name of the icon file, including the extension. A valid format for the icon file is the PTC-proprietary format used by Creo `.BIF`, `.GIF`, `.JPG`, or `.PNG`. The new pane is displayed with the icon image. If you specify the value as `NULL`, the default Creo icon is used.

  The default search paths for finding the icons are:

  - `<creo_loadpoint>\<datecode>\Common Files\text\resource`
  - `<Application text dir>\resource`
  - `<Application text dir>\<language>\resource`

  The location of the application text directory is specified in the registry file.
- *URL*—Specify the URL of the location to be accessed from the pane.

Use the method `pfcSession::NavigatorPaneBrowserIconSet` to set or change the icon of a specified browser pane in the navigation area.

Use the method `pfcSession::NavigatorPaneBrowserURLSet` to change the URL of the page displayed in the browser pane in the navigation area.

4

Selection

This chapter describes how to use Interactive Selection in Creo Object TOOLKIT C++.

Interactive Selection

Methods Introduced:

- **pfcBaseSession::Select**
- **pfcSelectionOptions::Create**
- **pfcSelectionOptions::SetMaxNumSels**
- **pfcSelectionOptions::SetOptionKeywords**
- **wfcWSelectionOptions::Create**
- **wfcWSelectionOptions::GetSelEnvOptions**
- **wfcWSelectionOptions::SetSelEnvOptions**
- **wfcSelectionEnvironmentOption::Create**
- **wfcSelectionEnvironmentOption::GetAttribute**
- **wfcSelectionEnvironmentOption::SetAttribute**
- **wfcSelectionEnvironmentOption::GetAttributeValue**
- **wfcSelectionEnvironmentOption::SetAttributeValue**
- **wfcWSelection::Verify**
- **wfcWSelection::GetWindow**

When you call the method `pfcBaseSession::Select`, the **Select** dialog box appears in Creo Parametric user interface for selecting objects and returns a `pfcSelections` sequence that contains the objects the user selected. Using the *Options* argument, you can control the type of object that can be selected and the maximum number of selections.

In addition, you can pass in a `pfcSelections` sequence to the method. The returned `pfcSelections` sequence will contain the input sequence and any new objects.

The methods `pfcSelectionOptions::Create` and `pfcSelectionOptions::SetOptionKeywords` take a *String* argument made up of one or more of the identifiers listed in the table below, separated by commas.

For example, to allow the selection of features and axes, the arguments would be *feature*, *axis*.

Creo Parametric Database Item	String Identifier	ModelItemType
Datum point	*point*	`pfcITEM_POINT`
Datum axis	*axis*	`pfcITEM_AXIS`
Datum plane	*datum*	`pfcITEM_SURFACE`
Coordinate system datum	*csys*	`pfcITEM_COORD_SYS`
Feature	*feature*	`pfcITEM_FEATURE`

Creo Parametric Database Item	String Identifier	ModelItemType
Edge (solid or datum surface)	*edge*	`pfcITEM_EDGE`
Edge (solid only)	*sldedge*	`pfcITEM_EDGE`
Edge (datum surface only)	*qltedge*	`pfcITEM_EDGE`
Datum curve	*curve*	`pfcITEM_CURVE`
Composite curve	*comp_crv*	`pfcITEM_CURVE`
Surface (solid or quilt)	*surface*	`pfcITEM_SURFACE`
Surface (solid)	*sldface*	`pfcITEM_SURFACE`
Surface (datum surface)	*qltface*	`pfcITEM_SURFACE`
Quilt	*dtmqlt*	`pfcITEM_QUILT`
Dimension	*dimension*	`pfcITEM_DIMENSION`
Reference dimension	*ref_dim*	`pfcITEM_REF_DIMENSION`
Integer parameter	*ipar*	`pfcITEM_DIMENSION`
Part	*part*	N/A
Part or subassembly	*prt_or_asm*	N/A
Assembly component model	*component*	N/A
Component or feature	*membfeat*	`pfcITEM_FEATURE`
Detail symbol	*dtl_symbol*	`pfcITEM_DTL_SYM_INSTANCE`
Note	*any_note*	`pfcITEM_NOTE`, `pfcITEM_DTL_NOTE`
Draft entity	*draft_ent*	`pfcITEM_DTL_ENTITY`
Table	*dwg_table*	`pfcITEM_TABLE`
Table cell	*table_cell*	`pfcITEM_TABLE`
Drawing view	*dwg_view*	N/A
Solid body	*3d_body*	`pfcITEM_BODY`
Datum Curve End	*curve_end*	`pfcITEM_CRV_START` or `pfcITEM_CRV_END`

When you specify the maximum number of selections, the argument to `pfcSelectionOptions::SetMaxNumSels` must be an Integer. The code will be as follows:

```
sel_options->setMaxNumSels (10);
```

The default value assigned when creating a `SelectionOptions` object is –1, which allows any number of selections by the user.

The method `wfcWSelectionOptions::Create` allows you to set options for selecting objects by specifying the selection attribute. The input arguments are:

- *OptionKeywords*—Specifies the selection filter.
- *MaxNumSels*—Specifies the maximum number of selections.
- *SelEnvOpts*—Specifies the selection attribute set using the method `wfcWSelectionOptions::SetSelEnvOptions`.

The method `wfcWSelectionOptions::GetSelEnvOptions` retrieves the selection attribute.

The method `wfcSelectionEnvironmentOption::Create` creates a data object of type `wfcSelectionEnvironmentOption` that contains information about the attributes for the interactive selection in Creo Parametric user interface. Use the method `wfcSelectionEnvironmentOption::SetAttribute` to set the selection attribute. The following attribute types are available:

- `wfcSELECT_DONE_REQUIRED`—Specifies that user has to click **OK** in the **Select** dialog box to get the selected items.
- `wfcSELECT_BY_MENU_ALLOWED`—Specifies that search tool is available in the method `pfcBaseSession::Select` when the attribute value is set to `1`, which is the default value.
- `wfcSELECT_BY_BOX_ALLOWED`—Specifies that user must draw a bounding box to get the items selected within the box.
- `wfcSELECT_ACTIVE_COMPONENT_IGNORE`—Specifies that user can select items external to the activate component.
- `wfcSELECT_HIDE_SEL_DLG`—Specifies that the **Select** dialog box must be hidden.

The method `wfcSelectionEnvironmentOption::GetAttribute` retrieves the selection attribute.

Use the method `wfcSelectionEnvironmentOption::GetAttributeValue` to get the integer value of the attributes set in the selection object.

The method `wfcWSelection::Verify` verifies if the content of the selection object is valid.

The method `wfcWSelection::GetWindow` retrieves the window where the selection was made.

Accessing Selection Data

Methods Introduced:

- **pfcSelection::GetSelModel**
- **pfcSelection::GetSelItem**
- **pfcSelection::GetPath**
- **pfcSelection::GetParams**
- **pfcSelection::GetTParam**
- **pfcSelection::GetPoint**
- **pfcSelection::GetDepth**
- **pfcSelection::GetSelView2D**

- **pfcSelection::GetSelTableCell**
- **pfcSelection::GetSelTableSegment**
- **wfcWSelection::GetDrawingTable**
- **wfcWSelection::GetPipelineFeature**

These methods return objects and data that make up the selection object. Using the appropriate methods, you can access the following data:

- For a selected model or model item use `pfcSelection::GetSelModel` or `pfcSelection::GetSelItem`.
- For an assembly component use `pfcSelection::GetPath`.
- For UV parameters of the selection point on a surface use `pfcSelection::GetParams`.
- For the T parameter of the selection point on an edge or curve use `pfcSelection::GetTParam`.
- For a three-dimensional point object that contains the selected point use `pfcSelection::GetPoint`.
- For selection depth, in screen coordinates use `pfcSelection::GetDepth`.
- For the selected drawing view, if the selection was from a drawing, use `pfcSelection::GetSelView2D`.
- For the selected table cell, if the selection was from a table, use `pfcSelection::GetSelTableCell`.
- For the selected table segment, if the selection was from a table, use `pfcSelection::GetSelTableSegment`.
- For the selected drawing table cell, if the selection was a drawing table, use `wfcWSelection::GetDrawingTable`.
- For the selected pipeline to get the pipeline feature, use `wfcWSelection::GetPipelineFeature`.

Controlling Selection Display

Methods Introduced:

- **pfcSelection::Highlight**
- **pfcSelection::UnHighlight**
- **pfcSelection::Display**

These methods cause a specific selection to be highlighted or dimmed on the screen using the color specified as an argument.

The method `pfcSelection::Highlight` highlights the selection in the current window. This highlight is the same as the one used by Creo application when selecting an item—it just repaints the wire-frame display in the new color. The highlight is removed if you use the **Repaint** command or `pfcWindow::Repaint`; it is not removed if you use `pfcWindow::Refresh`.

The method `pfcSelection::UnHighlight` removes the highlight.

The method `pfcSelection::Display` causes a selected object to be displayed on the screen, even if it is suppressed or hidden.

Note

This is a one-time action and the next repaint will erase this display.

Programmatic Selection

Creo Object TOOLKIT C++ provides methods whereby you can make your own Selection objects, without prompting the user. These Selections are required as inputs to some methods and can also be used to highlight certain objects on the screen.

Methods Introduced:

- **pfcCreateModelItemSelection**
- **pfcCreateComponentSelection**
- **pfcCreateModelSelection**
- **pfcSelection::SetSelItem**
- **pfcSelection::SetPath**
- **pfcSelection::SetParams**
- **pfcSelection::SetTParam**
- **pfcSelection::SetPoint**
- **pfcSelection::SetSelTableCell**
- **pfcSelection::SetSelView2D**

The method `pfcCreateModelItemSelection` creates a selection out of any model item object. It takes a `pfcModelItem` and optionally a `pfcComponentPath` object to identify which component in an assembly the Selection Object belongs to.

The method `pfcCreateComponentSelection` creates a selection out of any component in an assembly. It takes a `pfcComponentPath` object. For more information about `pfcComponentPath` objects, see the section Getting a Solid Object on page 203 in the Solid on page 202 chapter.

Use the method `pfcCreateModelSelection` to create a `pfcSelection` object, based on a `pfcModel` object.

Some Creo Object TOOLKIT C++ methods require more information to be set in the selection object. The methods allow you to set the following:

The selected item using the method `pfcSelection::SetSelItem`.

The selected component path using the method `pfcSelection::SetPath`.

The selected UV parameters using the method `pfcSelection::SetParams`.

The selected T parameter (for a curve or edge), using the method `pfcSelection::SetTParam`.

The selected XYZ point using the method `pfcSelection::SetPoint`.

The selected table cell using the method `pfcSelection::SetSelTableCell`.

The selected drawing view using the method `pfcSelection::SetSelView2D`.

Selection Buffer

Introduction to Selection Buffers

Selection is the process of choosing items on which you want to perform an operation. In Creo application, before a feature tool is invoked, the user can select items to be used in a given tool's collectors. Collectors are like storage bins of the references of selected items. The location where preselected items are stored is called the selection buffer.

Depending on the situation, different selection buffers may be active at any one time. In Part and Assembly mode, Creo offers the default selection buffer, the Edit selection buffer, and other more specialized buffers. Other Creo modes offer different selection buffers.

In the default Part and Assembly buffer there are two levels at which selection is done:

- First Level Selection

  Provides access to higher-level objects such as features or components. You can make a second level selection only after you select the higher-level object.
- Second Level Selection

Provides access to geometric objects such as edges and faces.

Note

First-level and second-level objects are usually incompatible in the selection buffer.

Creo Object TOOLKIT C++ allows access to the contents of the currently active selection buffer. The available functions allow your application to:

- Get the contents of the active selection buffer.
- Remove the contents of the active selection buffer.
- Add to the contents of the active selection buffer.

Reading the Contents of the Selection Buffer

Methods Introduced:

- **pfcSession::GetCurrentSelectionBuffer**
- **pfcSelectionBuffer::GetContents**

The method `pfcSession::GetCurrentSelectionBuffer` returns the selection buffer object for the current active model in session. The selection buffer contains the items preselected by the user to be used by the selection tool and popup menus.

Use the method `pfcSelectionBuffer::GetContents` to access the contents of the current selection buffer. The method returns independent copies of the selections in the selection buffer (if the buffer is cleared, this array is still valid).

Removing the Items of the Selection Buffer

Methods Introduced:

- **pfcSelectionBuffer::RemoveSelection**
- **pfcSelectionBuffer::Clear**

Use the method `pfcSelectionBuffer::RemoveSelection` to remove a specific selection from the selection buffer. The input argument is the *IndexToRemove* specifies the index where the item was found in the call to the method `pfcSelectionBuffer::GetContents`.

Use the method `pfcSelectionBuffer::Clear` to clear the currently active selection buffer of all contents. After the buffer is cleared, all contents are lost.

Adding Items to the Selection Buffer

Method Introduced:

- **pfcSelectionBuffer::AddSelection**
- **wfcWSession::AddCollectionToSelectionBuffer**
- **wfcWSelection::GetCollection**

Use the method `pfcSelectionBuffer::AddSelection` to add an item to the current selection buffer.

Note

The selected item must refer to an item that is in the current model such as its owner, component path or drawing view.

This method may fail due to any of the following reasons:

- There is no current selection buffer active.
- The selection does not refer to the current model.
- The item is not currently displayed and so cannot be added to the buffer.
- The selection cannot be added to the buffer in combination with one or more objects that are already in the buffer. For example: geometry and features cannot be selected in the default buffer at the same time.

Use the method `wfcWSession::AddCollectionToSelectionBuffer` to add collection to the current selection buffer.

The method `wfcWSelection::GetCollection` to get collection object that contains the items selected by the user.

5

Ribbon Tabs, Groups, and Menu Items

This chapter describes the Creo Object TOOLKIT C++ support for the Ribbon User Interface (UI). It also describes the impact of the ribbon user interface on legacy Pro/TOOLKIT applications and the procedure to place the commands, buttons, and menu items created by the legacy applications in the Creo Parametric ribbon user interface. Refer to the Creo Parametric Help for more information on the ribbon user interface and the procedure to customize the ribbon.

Creating Ribbon Tabs, Groups, and Menu Items

Customizations to the ribbon user interface using the Creo Object TOOLKIT C++ applications are supported through the **Customize Ribbon** tab in the **Creo Parametric Options** dialog box. You can specify the user interface layout for a Creo Object TOOLKIT C++ application and save the layout definition in a ribbon definition file, `toolkitribbonui.rbn`. Set the configuration option `tk_enable_ribbon_custom_save` to `yes` before customizing the ribbon user interface using the Creo Object TOOLKIT C++ application. When you run Creo Parametric, the `toolkitribbonui.rbn` file is loaded along with the Creo Object TOOLKIT C++ application and the commands created by the Creo Object TOOLKIT C++ application appear in the ribbon user interface. Refer to the section About the Ribbon Definition File on page 85 for more information on the `toolkitribbonui.rbn` file.

You can customize the ribbon user interface only for a particular mode in Creo Parametric. For example, if you customize the ribbon user interface and save it to the `toolkitribbonui.rbn` file in the Part mode, then on loading Creo Parametric the customized user interface will be visible only in the Part mode. To view a particular tab or group in all the modes, you must customize the ribbon user interface and save the `toolkitribbonui.rbn` file in each mode. Refer to the Fundamentals Help for more information on customizing the ribbon.

Note

You can add a new group to an existing tab or create a new tab using the **Customize Ribbon** tab in the**Creo Parametric Options** dialog box. You will not be able to modify the tabs or groups that are defined by Creo Parametric.

Workflow to Add Menu Items to the Ribbon User Interface

Note

The instructions explained below are applicable only if the application is implemented in full asynchronous mode. This is because applications in simple asynchronous mode cannot handle requests, that is, command callbacks, from Creo Parametric. Refer to the chapter Asynchronous Mode on page 587, for more information.

Set the configuration option `tk_enable_ribbon_custom_save` to `yes` before customizing the ribbon user interface. The steps to add commands to the Creo Parametric ribbon user interface are as follows:

1. Create a Creo Object TOOLKIT C++ application with complete command definition, which includes specifying command label, help text, large icon name, and small icon name. Designate the command using the `pfcUICommand::Designate`.
2. Start the Creo Object TOOLKIT C++ application and ensure that it has started or connected to Creo Parametric. The commands created by the Creo Object TOOLKIT C++ application will be loaded in Creo Parametric.
3. Click **File ▸ Options**. The **Creo Parametric Options** dialog box opens.
4. Click **Customize Ribbon**.
5. In the **Customize the Ribbon** list, select a tab and create a new group in it or create a new tab and a group in it.
6. In the **Choose commands from** list, select **TOOLKIT Commands**. The commands created by the Creo Object TOOLKIT C++ application are displayed.
7. Click **Add** to add the commands to the new tab or group.
8. Click **Import/Export ▸ Save the Auxilliary Application User Interface**. The changes are saved to the `toolkitribbonui.rbn` file. The `toolkitribbonui.rbn` file is saved in the text folder specified in the Creo Object TOOLKIT C++ application registry file. For more information refer to the section on About the Ribbon Definition File on page 85.

> **Note**
>
> The **Save the Auxilliary Application User Interface** button is enabled only if you set the configuration option `tk_enable_ribbon_custom_save` to `yes`.

9. Click **Apply**. The custom settings are saved to the `toolkitribbonui.rbn` file.
10. Reload the Creo Object TOOLKIT C++ application or restart Creo Parametric. The `toolkitribbonui.rbn` file will be loaded along with the Creo Object TOOLKIT C++ application.

If translated messages are available for the newly added tabs or groups, then Creo Parametric displays the translated strings by searching for the same string from the list of string based messages that are loaded. For more information refer to the section on Localizing the Ribbon User Interface Created by the Object TOOLKIT Applications on page 88.

About the Ribbon Definition File

A ribbon definition file is a file that is created through the **Customize Ribbon** interface in Creo Parametric. This file defines the containers, that is, Tabs, Group, or Cascade menus that are created by a particular Creo Object TOOLKIT C++ application. It contains information on whether to show an icon or label. It also contains the size of the icon to be used, that is, a large icon (32X32) or a small icon (16x16).

The ribbon user interface displays the commands referenced in the ribbon definition file only if the commands are loaded and are visible in that particular Creo Parametric mode. If translated messages are available for the newly added tabs or groups, then Creo Parametric displays the translated strings by searching for the same string from the list of string based messages that are loaded. For more information refer to the section on Localizing the Ribbon User Interface Created by the Object TOOLKIT Applications on page 88.

Set the configuration option `tk_enable_ribbon_custom_save` to `yes` before customizing the ribbon user interface. To save the ribbon user interface layout definition to the `toolkitribbonui.rbn` file:

1. Click **File ▸ Options**. The **Creo Parametric Options** dialog box opens.
2. Click **Customize Ribbon**.
3. In the **Customize the Ribbon** list, select a tab and create a new group in it or create a new tab and a group in it.
4. In the **Choose commands from** list, select **TOOLKIT Commands**. The commands created by the Creo Object TOOLKIT C++ application are displayed.
5. Click **Add** to add the commands to the new tab or group.
6. Click **Import/Export ▸ Save the Auxilliary Application User Interface**. The modified layout is saved to the `toolkitribbonui.rbn` file located in the `text` folder within the Creo Object TOOLKIT C++ application directory, that is, `<application_dir>\text`

> **Note**
>
> The **Save the Auxilliary Application User Interface** button is enabled only if you set the configuration option `tk_enable_ribbon_custom_save` to `yes`.

7. Click **OK**.

Note

You cannot edit the `toolkitribbonui.rbn` file manually.

To Specify the Path for the Ribbon Definition File

You can rename the `toolkitribbonui.rbn` to another filename with the `.rbn` extension. To enable the Creo Object TOOLKIT C++ application to read the ribbon definition file having a name other than `toolkitribbonui.rbn`, it must be available at the location `<application_dir>\text\ribbon`. The function introduced in this section enables you to load the ribbon definition file from within a Creo Object TOOLKIT C++ application.

Function Introduced:

- **pfcSession::RibbonDefinitionfileLoad**

The function `pfcSession::RibbonDefinitionfileLoad` loads a specified ribbon definition file from a default path into the Creo Parametric application. The input argument is as follows:

- *file_name* - Specify the name of the ribbon definition file including its extension. The default search path for this file is:
  - The working directory from where Creo Parametric is loaded.
  - `<application_text_dir>\ribbon`
  - `<application_text_dir>\language\ribbon`

Note

- The location of the application text directory is specified in the Creo Object TOOLKIT C++ registry file.
- A Creo Object TOOLKIT C++ application can load a ribbon definition file only once. After the application has loaded the ribbon, calls made to the method `pfcSession::RibbonDefinitionfileLoad` to load other ribbon definition files are ignored.

Loading Multiple Applications Using the Ribbon Definition File

Creo Parametric supports loading of multiple `.rbn` files in the same session. You can develop multiple Creo Object TOOLKIT C++ applications that share the same tabs or groups and each application will have its own ribbon definition file. As each application is loaded, its `.rbn` file will be read and applied. When an application is unloaded, the containers and command created by its `.rbn` file will be removed.

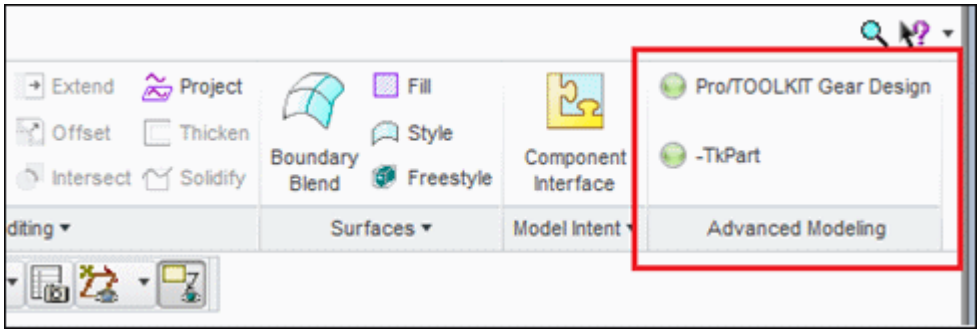
For example, consider two Creo Object TOOLKIT C++ applications, namely, `pt_geardesign` and `pt_examples` that add commands to the same group on a tab on the Ribbon user interface. The application `pt_geardesign` adds a command **Pro/TOOLKIT Gear Design** to the **Advanced Modeling** group on the **Modeling** tab and the application `pt_examples` adds a command **TKPart** to the **Advanced Modeling** group on the **Model** tab. The ribbon definition file for each application will contain an instruction to create the **Advanced Modeling** group and if both the ribbon files are loaded, the group will be created only once and the two ribbon customizations will be merged into the same group.

That is, if both the applications are running in the same Creo Parametric session, then the commands, **Pro/TOOLKIT Gear Design** and **TKPart** will be available under the **Advanced Modeling** group on the **Model** tab.

Note

The order in which the commands will be displayed within the group will depend on the order of loading of the `.rbn` file for each application.

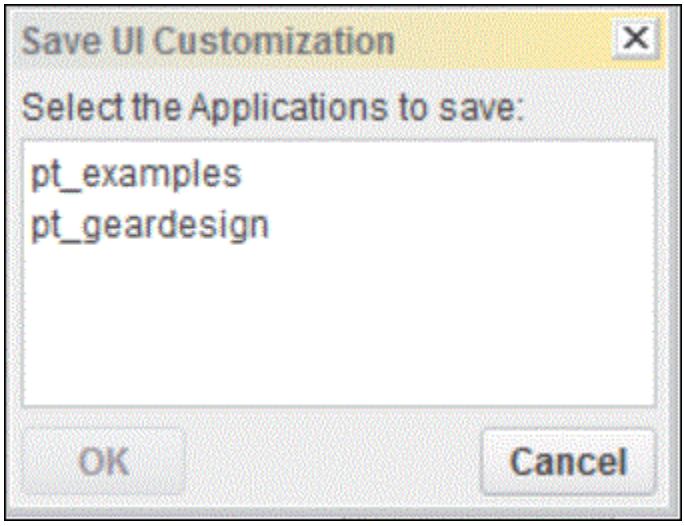
The following image displays commands added by two Creo Object TOOLKIT C++ applications to the same group.



To save the customization when multiple applications are loaded:

1. Click **File** ▸ **Options**. The **Creo Parametric Options** dialog box opens.
2. Click **Customize Ribbon**.

3. In the **Customize the Ribbon** list, select a tab and create a new group in it or create a new tab and a group in it.
4. In the **Choose commands from list**, select **TOOLKIT Commands**. The commands created by the Creo Object TOOLKIT C++ application are displayed.
5. Click **Add** to add the commands to the new tab or group.
6. Click **Import/Export** ▸ **Save the Auxilliary Application User Interface**. The **Save UI Customization** dialog box opens.
7. Select a Creo Object TOOLKIT C++ application and Click **Save**. The modified layout is saved to the `.rbn` file of the specified Creo Object TOOLKIT C++ application.
   The **Save UI Customization** dialog box is shown in the following image:



Localizing the Ribbon User Interface Created by the Creo Object TOOLKIT C++ Applications

The labels for the custom tabs, groups, and cascade menus belonging to the Creo Object TOOLKIT C++ application can be translated in the languages supported by Creo Parametric. To display localized labels, specify the translated labels in the `ribbonui.txt` file and save this file at the location `<application_text_dir>\<language>`. For example, the text file for German locale must be saved at the location `<application_text_dir>\german\ribbonui.txt`.

Create a file containing translations for each of the languages in which the Creo Object TOOLKIT C++ application is localized. The Localized translation files must use the UTF-8 encoding with BOM character for the translated text to be displayed correctly in the user interface.

The format of the `ribbonui.txt` file is as shown below. Specify the following lines for each label entry in the file:

1. A hash sign (#) followed by the label, as specified in the ribbon definition file.
2. The label as specified in the ribbon definition file and as displayed in the ribbon user interface.
3. The translated label.
4. Add an empty line at the end of each label entry in the file.

For example, if the Creo Object TOOLKIT C++ application creates a tab with the name TK_TAB having a group with the name TK_GROUP, then the translated file will contain the following:

```
#TK_TAB
TK_TAB
<translation for TK_TAB>
<Empty_line>
#TK_GROUP
TK_GROUP
<translation for TK_GROUP>
<Empty_line>
```

6

Menus, Commands, and Pop-up Menus

This chapter describes the methods provided by Creo Object TOOLKIT C++ to create and modify menus, buttons, and pop-up menus in the Creo user interface.

Refer to the chapter Ribbon Tabs, Groups, and Menu Items on page 82 for more information. Also, refer to the Creo Parametric Help for more information on customizing the ribbon user interface.

Introduction

The Creo Object TOOLKIT C++ classes enable you to supplement the Creo ribbon user interface.

Once the Creo Object TOOLKIT C++ application is loaded, you can add a new group to an existing tab or create a new tab using the **Customize Ribbon** tab in the **Creo Parametric Options** dialog box in Creo Parametric. You will not be able to modify the groups that are defined by Creo Parametric. If the translated messages are available for the newly added tabs or groups, then Creo Parametric will use them by searching for the same string in the list of sting based messages loaded.

You can customize the ribbon user interface only for a particular mode in Creo Parametric. For example, if you customize the ribbon user interface and save it to the `toolkitribbonui.rbn` file in the Part mode, then on loading Creo Parametric the customized user interface will be visible only in the Part mode. To view a particular tab or group in all the modes, you must customize the ribbon user interface and save the `toolkitribbonui.rbn` file in each mode. Refer to the Creo Parametric Fundamentals Help for more information on customizing the ribbon.

Menu Bar Definitions

- Menu—A menu, such as the **File** menu, or a sub-menu, such as the **Manage File** menu under the **File** menu.
- Menu button—A named item in a group or menu that is used to launch a set of instructions.
- Pop-up menu—A menu invoked by selection of an item in the Creo Parametric graphics window.
- Command—A procedure in Creo Parametric that may be activated from a button.

Menu Buttons and Menus

The following methods enable you to add new menu buttons in any location on the Ribbon user interface.

Methods Introduced:

- **pfcSession::UICreateCommand**
- **pfcSession::UICreateMaxPriorityCommand**
- **pfcUICommandActionListener::OnCommand**

The method pfcSession::UICreateCommand creates a pfcUICommand object that contains a pfcUICommandActionListener. You should override the pfcUICommandActionListener::OnCommand method with the code that you want to execute when the user clicks a button.

The method pfcSession::UICreateMaxPriorityCommand creates a pfcUICommand object having maximum command priority. The priority of the action refers to the level of precedence the added action takes over other Creo Parametric actions. Maximum priority actions dismiss all other actions except asynchronous actions.

Maximum command priority should be used only in commands that open and activate a new model in a window. Create all other commands using the method pfcSession::UICreateCommand.

The listener method pfcUICommandListener::OnCommand is called when the command is activated in Creo Parametric by pressing a button.

Designate the command using the function pfcUICommand::Designate and add a button to the ribbon user interface using the using the **Customize Ribbon** tab in the **Creo Parametric Options** dialog box. This operation binds the command to the button.

Finding Creo Parametric Commands

This method enables you to find existing Creo Parametric commands in order to modify their behavior.

Method Introduced:

- **pfcSession::UIGetCommand**

The method pfcSession::UIGetCommand returns a pfcUICommand object representing an existing Creo Parametric command. The method allows you to find the command ID for an existing command so that you can add an access function or bracket function to the command. You must know the name of the command in order to find its ID.

To find the name of an action command, click the corresponding icon on the ribbon user interface and then check the last entry in the trail file. For example, for the Save icon, the trail file will have the corresponding entry:

```
~ Command `ProCmdModelSave`
```

The action name for the Save icon is ProCmdModelSave. This string can be used as input to pfcSession::UIGetCommand to get the command ID.

You can determine a command ID string for an option without an icon by searching through the resource files located in the <creo_loadpoint>\<datecode>\Common Files\text\resources directory. If you search for the menu button name, the line will contain the corresponding action command for the button.

Access Listeners for Commands

These methods allow you to apply an access listener to a command. The access listener determines whether or not the command is visible at the current time in the current session.

Methods Introduced:

- **pfcActionSource::AddActionListener**
- **pfcUICommandAccessListener::OnCommandAccess**

Use the method `pfcActionSource::AddActionListener` to register a new `pfcUICommandAccessListener` on any command (created either by an application or Creo Parametric). This listener will be called when buttons based on the command might be shown.

The listener method `pfcUICommandAccessListener::OnCommandAccess` allows you to impose an access function on a particular command. The method determines if the action or command should be available, unavailable, or hidden.

The potential return values are listed in the enumerated type `pfcCommandAccess` and are as follows:

- `pfcACCESS_REMOVE`—The button is not visible and if all of the menu buttons in the containing menu possess an access function returning `pfcACCESS_REMOVE`, the containing menu will also be removed from the Creo Parametric user interface.
- `pfcACCESS_INVISIBLE`—The button is not visible.
- `pfcACCESS_UNAVAILABLE`—The button is visible, but gray and cannot be selected.
- `pfcACCESS_DISALLOW`—The button shows as available, but the command will not be executed when it is chosen.
- `pfcACCESS_AVAILABLE`—The button is not gray and can be selected by the user. This is the default value.

Bracket Listeners for Commands

These methods allow you to apply a bracket listener to a command. The bracket listener is called before and after the command runs, which allows your application to provide custom logic to execute whenever the command is selected.

Methods Introduced:

- **pfcActionSource::AddActionListener**
- **pfcUICommandBracketListener::OnBeforeCommand**
- **pfcUICommandBracketListener::OnAfterCommand**

Use the method `pfcActionSource::AddActionListener` to register a new `pfcCommand::UICommandBracketListener` on any command (created either by an application or Creo Parametric). This listener will be called when the command is selected by the user.

The listener methods `pfcUICommandBracketListener::OnBeforeCommand` and `pfcUICommandBracketListener::OnAfterCommand` allow the creation of functions that will be called immediately before and after execution of a given command. These methods are used to add the business logic to the start or end (or both) of an existing Creo Parametric command.

The method `pfcUICommandBracketListener::OnBeforeCommand` could also be used to cancel an upcoming command. To do this, throw a `pfcXCancelProEAction` exception from the body of the listener method using `pfcXCancelProEAction::Throw`.

Designating Commands

Using Creo Object TOOLKIT C++ you can designate Creo Parametric commands. These commands can later appear in the Creo Parametric ribbon user interface.

To add a command you must:

1. Define or add the command to be initiated on clicking the icon in the Creo Object TOOLKIT C++ application.
2. Optionally designate an icon button to be used with the command.
3. Designate the command to appear in the **Customize Ribbon** tab in the **Creo Parametric Options** dialog box.

> **Note**
>
> Refer to the chapter on Ribbon Tabs, Groups, and Menu Items on page 82 for more information. Also, refer to the Creo Parametric Help for more information on customizing the Ribbon User Interface.

4. Save the configuration in Creo Parametric so that changes to the ribbon user interface appear when a new session of Creo Parametric is started.

Command Icons

Method Introduced:

- **pfcUICommand::SetIcon**

The method `pfcUICommand::SetIcon` allows you to designate an icon to be used with the command you created. The method adds the icon to the Creo Parametric command. Specify the name of the icon file, including the extension as the input argument for this method. A valid format for the icon file is a standard `.GIF`, `.JPG`, or `.PNG`. PTC recommends using `.PNG` format. All icons in the Creo Parametric ribbon are either 16x16 (small) or 32x32 (large) size. The naming convention for the icons is as follows:

- Small icon—`<iconname.ext>`
- Large icon—`<iconname_large.ext>`

Note

- The legacy naming convention for icons `<icon_name_icon_size.ext>` will not be supported in future releases of Creo Parametric. The icon size was added as a suffix to the name of the icon. For example, the legacy naming convention for small icons was `iconname16X16.ext`. It is recommended to use the standard naming conventions for icons, that is, `iconname.ext` or `iconname_large.ext`.
- While specifying the name of the icon file, do not specify the full path to the icon names.

The application searches for the icon files in the following locations:

- `<creo_loadpoint>\<datecode>\Common Files\text\resource`
- `<Application text dir>\resource`
- `<Application text dir>\<language>\resource`

  The location of the application text directory is specified in the registry file.

Commands that do not have an icon assigned to them display the button label.

You may also use this method to assign a small icon to a button. The icon appears to the left of the button label.

Designating the Command

Method Introduced:

- **pfcUICommand::Designate**

This method allows you designate the command as available in the **Customize Ribbon** tab in the **Creo Parametric Options** dialog of Creo Parametric. After a Creo Object TOOLKIT C++ application has used the method `pfcUICommand::Designate` on a command, you can add the button associated with this command into the Creo Parametric ribbon user interface.

If this method is not called, the button will not be visible in the **Toolkit Commands** list in the **Customize Ribbon** tab in the **Creo Parametric Options** dialog of Creo Parametric.

The arguments to this method are:

- *Label*—The message string that refers to the icon label. This label (stored in the message file) identifies the text seen when the button is displayed. If the command is not assigned an icon, the button label string appears on the toolbar button by default.
- *Help*—The one-line Help for the icon. This label (stored in the message file) identifies the help line seen when the mouse moves over the icon.
- *Description*—The message appears in the **Customize Ribbon** tab in the **Creo Parametric Options** dialog box and also when **Description** is clicked in Creo Parametric.
- *MessageFile*—The message file name. All the labels including the one-line Help labels must be present in the message file.

> **Note**
>
> This file must be in the directory `<text_path>\text` or `<text_path>\text\<language>`.

Placing the Button

Once the button has been created using the methods discussed, place the button on the Creo Parametric ribbon user interface. Refer to the chapter on Ribbon Tabs, Groups, and Menu Items on page 82 for more information. Also, refer to the Creo Parametric Help for more information on customizing the Ribbon User Interface.

Pop-up Menus

Creo Parametric provides shortcut menus that contain frequently used commands appropriate to the currently selected items. You can access a shortcut menu by right-clicking a selected item. Shortcut menus are accessible in:

- Graphics window

- Model Tree
- Some dialog boxes
- Any area where you can perform an object-action operation by selecting an item and choosing a command to perform on the selected item.

The methods described in this section allow you to add menus to a graphics window pop-up menu.

Adding a Pop-up Menu to the Graphics Window

You can activate different pop-up menus during a given session of Creo Parametric. Every time the Creo Parametric context changes when you open a different model type, enter different tools or special modes such as **Edit**, a different pop-up menu is created. When Creo Parametric moves to the next context, the pop-up menu may be destroyed.

As a result of this, Creo Object TOOLKIT C++ applications must attach a button to the pop-up menu during initialization of the pop-up menu. The Creo Object TOOLKIT C++ application is notified each time a particular pop-up menu is created, which then allows the user to add to the pop-up menu.

Use the following procedure to add items to pop-up menus in the graphics window:

1. Obtain the name of the existing pop-up menus to which you want to add a new menu using the trail file.
2. Create commands for the new pop-up menu items.
3. Implement access listeners to provide visibility information for the items.
4. Add an action listener to the session to listen for pop-up menu initialization.
5. In the listener method, if the pop-up menu is the correct menu to which you wish to add the button, then add it.

The following sections describe each of these steps in detail. You can add push buttons and cascade menus to the pop-up menus. You can add pop-up menu items only in the active window. You cannot use this procedure to remove items from existing menus.

Using the Trail File to Determine Existing Pop-up Menu Names

The trail file in Creo Parametric contains a comment that identifies the name of the pop-up menu if the configuration option, `auxapp_popup_menu_info` is set to `yes`.

For example, the pop-up menu, **Edit Properties**, has the following comment in the trail file:

Listening for Pop-up Menu Initialization

Methods Introduced:

- **pfcActionSource::AddActionListener**
- **pfcActionSource::AddActionListenerWithType**
- **pfcPopupmenuListener::OnPopupmenuCreate**

Use the method `pfcActionSource::AddActionListener` or `pfcActionSource::AddActionListenerWithType` to register a new `pfcPopupmenuListener` to the session. This listener will be called when pop-up menus are initialized.

The method `pfcPopupmenuListener::OnPopupmenuCreate` is called after the pop-up menu is created internally in Creo Parametric and may be used to assign application-owned buttons to the pop-up menu.

Accessing the Pop-up Menus

The method described in this section provides the name of the pop-up menus used to access these menus while using other methods.

Method Introduced:

- **pfcPopupmenu::GetName**

The method `pfcPopupmenu::GetName` returns the name of the pop-up menu.

Adding Content to the Pop-up Menus

Methods Introduced:

- **pfcPopupmenu::AddButton**
- **pfcPopupmenu::AddMenu**

Use `pfcPopupmenu::AddButton` to add a new item to a pop-up menu. The input arguments are:

- *Command*—Specifies the command associated with the pop-up menu.
- *Options* – A `pfcPopupmenuOptions` object containing other options for the method. The options that may be included are:
  - `PositionIndex`—Specifies the position in the pop-up menu at which to add the menu button. Pass null to add the button to the bottom of the menu. Use the method `pfcPopupmenuOptions::SetPositionIndex` to set this option.

- `Name`—Specifies the name of the added button. The button name is placed in the trail file when the user selects the menu button. Use the method `pfcPopupmenuOptions::SetName` to set this option.
- `SetLabel`—Specifies the button label. This label identifies the text displayed when the button is displayed. Use the method to set this option.
- `Helptext`—Specifies the help message associated with the button. Use the method `pfcPopupmenuOptions::SetHelptext` to set this option.

Use the method `pfcPopupmenu::AddMenu` to add a new cascade menu to an existing pop-up menu.

The argument for this method is a `pfcPopupmenuOptions` object, whose members have the same purpose as described for newly added buttons. This method returns a new `pfcPopupmenu` object to which you may add new buttons.

7

User Interface Foundation Classes for Dialogs

This chapter describes the classes provided by Creo Object TOOLKIT C++ to create and modify dialog boxes, menus, buttons, pop-up menus and so on.

Introduction

From Creo 3.0 onward, the Creo UI Editor allows you to interactively create and edit dialog boxes for Creo Object TOOLKIT C++ customizations. The Creo UI Editor can be installed from the Creo Installer. The editor provides a library of graphical user interface components such as buttons, lists, and so on. The UIFC classes provide enhanced attributes and actions for the user interface components. The UIFC framework is available in Creo Object TOOLKIT C++. You can generate callbacks in Creo Object TOOLKIT C++. Refer to the *Creo UI Editor C ++ User's Guide*, for more information.

- `uifcButtonBase`
- `uifcCheckButton`
- `uifcComponent`
- `uifcCore`
- `uifcDialog`
- `uifcDrawingArea`
- `uifcExternal_Containers`
- `uifcGlobals`
- `uifcGridContainer`
- `uifcHTMLWindow`
- `uifcInputPanel`
- `uifcLabel`
- `uifcLayout`
- `uifcList`
- `uifcMenuBar`
- `uifcMenuPane`
- `uifcNakedWindow`
- `uifcOptionMenu`
- `uifcPGLWindow`
- `uifcPHolder`
- `uifcProgressBar`
- `uifcPushButton`
- `uifcRadioGroup`
- `uifcRange_Controls`
- `uifcSash`
- `uifcScrollBar`
- `uifcScrolledLayout`
- `uifcSelection`

- uifcSeparator
- uifcSlider
- uifcSpinBox
- uifcTab
- uifcTable
- uifcText
- uifcTextArea
- uifcThumbWheel
- uifcToolBar
- uifcTree

8

Models

This chapter describes how to program on the model level using Creo Object TOOLKIT C++.

Overview of Model Objects

Models can be any Creo file type, including parts, assemblies, drawings, sections, and notebook. The classes and methods in `pfcModel` provide generic access to models, regardless of their type. The available methods enable you to do the following:

- Access information about a model.
- Open, copy, rename, and save a model.

Getting a Model Object

Methods Introduced:

- **pfcFamilyTableRow::CreateInstance**
- **pfcSelection::GetSelModel**
- **pfcBaseSession::GetModel**
- **pfcBaseSession::GetCurrentModel**
- **pfcBaseSession::GetActiveModel**
- **pfcBaseSession::ListModels**
- **pfcBaseSession::GetByRelationId**
- **pfcWindow::GetModel**

These methods get a model object that is already in session.

The method `pfcSelection::GetSelModel` returns the model that was interactively selected.

The method `pfcBaseSession::GetModel` returns a model based on its name and type, whereas `pfcBaseSession::GetByRelationId` returns a model in an assembly that has the specified integer identifier.

The method `pfcBaseSession::GetCurrentModel` returns the current active model.

The method `pfcBaseSession::GetActiveModel` returns the active Creo model.

Use the method `pfcBaseSession::ListModels` to return a sequence of all the models in session.

For more methods that return solid models, refer to the chapter Solid on page 202.

Model Descriptors

Methods Introduced:

- **pfcModelDescriptor::Create**
- **pfcModelDescriptor::CreateFromFileName**
- **pfcModelDescriptor::SetGenericName**
- **pfcModelDescriptor::SetInstanceName**
- **pfcModelDescriptor::SetType**
- **pfcModelDescriptor::SetDevice**
- **pfcModelDescriptor::SetPath**
- **pfcModelDescriptor::SetFileVersion**
- **pfcModelDescriptor::GetFullName**
- **pfcModel::GetFullName**

Model descriptors are data objects used to describe a model file and its location in the system. The methods in the model descriptor enable you to set specific information that enables the Creo application to find the specific model you want.

The static utility method `pfcModelDescriptor::Create` allows you to specify as data to be entered a model type, an instance name, and a generic name. The model descriptor constructs the full name of the model as a string, as follows:

```
xstring FullName = InstanceName+"<"+GenericName+">";
                                    // As long as the
                                    // generic name is
                                    // not an empty
                                    // string ("")
```

If you want to load a model that is not a family table instance, pass an empty string as the generic name argument so that the full name of the model is constructed correctly. If the model is a family table interface, you should specify both the instance and generic names.

Note

You are allowed to set other fields in the model descriptor object, but they may be ignored by some methods.

The static utility method `pfcModelDescriptor::CreateFromFileName` allows you to create a new model descriptor from a given a file name. The file name is a string in the form `<name>.<extension>`.

Retrieving Models

Methods Introduced:

- **pfcBaseSession::RetrieveModel**

- **pfcBaseSession::RetrieveModelWithOpts**
- **pfcBaseSession::OpenFile**
- **pfcSolid::HasRetrievalErrors**

These methods enable Creo to retrieve the model that corresponds to the `pfcModelDescriptor` argument.

The method `pfcBaseSession::RetrieveModel` retrieves the specified model into the Creo session given its model descriptor from a standard directory. This method ignores the path argument specified in the model descriptor. But this function does not create a window for it, nor does it display the model anywhere.

The method `pfcBaseSession::RetrieveModelWithOpts` retrieves the specified model into the Creo session based on the path specified by the model descriptor. The path can be a disk path, a workspace path, or a commonspace path. The *Opts* argument (given by the `pfcRetrieveModelOptions` object) provides the user with the option to specify simplified representations.

The method `pfcBaseSession::OpenFile` brings the model into memory, opens a new window for it (or uses the base window, if it is empty), and displays the model.

Note

`pfcBaseSession::OpenFile` actually returns a handle to the window it has created.

The file version set by the `pfcModelDescriptor::SetFileVersion` method is ignored by the `pfcBaseSession::RetrieveModelWithOpts` and `pfcBaseSession::OpenFile` methods. Instead, the latest file version of the model is used by these two methods.

To get a handle to the model you need, use the method `pfcWindow::GetModel`.

The method `pfcSolid::HasRetrievalErrors` returns a true value if the features in the solid model were suppressed during the RetrieveModel or OpenFile operations. This method must be called immediately after the `pfcBaseSession::RetrieveModel` method or an equivalent retrieval method.

Model Information

Methods Introduced:

- **pfcModel::IsNativeModel**
- **pfcModel::GetFileName**

- **pfcModel::GetCommonName**
- **pfcModel::IsCommonNameModifiable**
- **pfcModel::GetFullName**
- **pfcModel::GetGenericName**
- **pfcModel::GetInstanceName**
- **wfcWModel::GetDefaultName**
- **pfcModel::GetOrigin**
- **pfcModel::GetRelationId**
- **pfcModel::GetDescr**
- **pfcModel::GetType**
- **pfcModel::GetIsModified**
- **pfcModel::GetVersion**
- **pfcModel::GetRevision**
- **pfcModel::GetBranch**
- **pfcModel::GetReleaseLevel**
- **pfcModel::GetVersionStamp**
- **pfcModel::ListDependencies**
- **pfcModel::CleanupDependencies**
- **pfcModel::ListDeclaredModels**
- **pfcModel::CheckIsModifiable**
- **pfcModel::CheckIsSaveAllowed**
- **wfcWModel::GetSubType**

The method `pfcModel::IsNativeModel` returns `true`, if the specified model is a native Creo model. It returns `false`, when the model is a non-native part or assembly created in other CAD applications.

The method `pfcModel::GetFileName` retrieves the model file name in the "name"."type" format.

The method `pfcModel::GetCommonName` retrieves the common name for the model. This name is displayed for the model in Windchill PDMLink.

Use the method `pfcModel::GetIsCommonNameModifiable` to identify if the common name of the model can be modified. You can modify the name for models that are not yet owned by Windchill PDMLink, or in certain situations if the configuration option `let_proe_rename_pdm_objects` is set to `yes`.

The method `pfcModel::GetFullName` retrieves the full name of the model in the instance <generic> format.

The method pfcModel::GetGenericName retrieves the name of the generic model. If the model is not an instance, this name must be NULL or an empty string.

The method pfcModel::GetInstanceName retrieves the name of the model. If the model is an instance, this method retrieves the instance name.

The method wfcWModel::GetDefaultName gets the default name assigned to the model by Creo application at the time of creation.

The method pfcModel::GetOrigin returns the complete path to the file from which the model was opened. This path can be a location on disk from a Windchill workspace, or from a downloaded URL.

The method pfcModel::GetRelationId retrieves the relation identifier of the specified model. It can be NULL.

The method pfcModel::GetDescr returns the descriptor for the specified model. Model descriptors can be used to represent models not currently in session.

Note

The methods pfcModel::GetFullName, pfcModel::GetGenericName, and pfcModel::GetDescr throw an exception pfcXtoolkitCantOpen if called on a model instance whose immediate generic is not in session. Handle this exception and typecast the model as pfcSolid, which in turn can be typecast as pfcFamilyMember, and use the method pfcFamilyMember::GetImmediateGenericInfo to get the model descriptor of the immediate generic model. The model descriptor can be used to derive the full name or generic name of the model.

If you wish to switch this behavior to the pre-Wildfire 4.0 mode, set the configuration option retrieve_instance_dependencies to instance_and_generic_deps.

The method pfcModel::GetType returns the type of model in the form of the pfcModelType object. The types of models are as follows:

- pfcMDL_ASSEMBLY—Specifies an assembly.
- pfcMDL_PART—Specifies a part.
- pfcMDL_DRAWING—Specifies a drawing.
- pfcMDL_2D_SECTION—Specifies a 2D section.
- pfcMDL_LAYOUT—Specifies a notebook.
- pfcMDL_DWG_FORMAT—Specifies a drawing format.

- `pfcMDL_MFG`—Specifies a manufacturing model.
- `pfcMDL_REPORT`—Specifies a report.
- `pfcMDL_MARKUP`—Specifies a drawing markup.
- `pfcMDL_DIAGRAM`—Specifies a diagram
- `pfcMDL_CE_SOLID`—Specifies a Layout model.

> **Note**
>
> Creo Object TOOLKIT C++ methods will only be able to read models of type Layout, but will not be able to pass Layout models as input to other methods. PTC recommends that you review all Creo Object TOOLKIT C++ applications that use the enumerated type `pfcModelType` and modify the code as appropriate to ensure that the applications work correctly.

The method `pfcModel::GetIsModified` identifies whether the model has been modified since it was last saved.

The method `pfcModel::GetVersion` returns the version of the specified model from the PDM system. It can be `NULL`, if not set.

The method `pfcModel::GetRevision` returns the revision number of the specified model from the PDM system. It can be `NULL`, if not set.

The method `pfcModel::GetBranch` returns the branch of the specified model from the PDM system. It can be `NULL`, if not set.

The method `pfcModel::GetReleaseLevel` returns the release level of the specified model from the PDM system. It can be `NULL`, if not set.

The method `pfcModel::GetVersionStamp` returns the version stamp of the specified model. The version stamp is a Creo specific identifier that changes with each change made to the model.

The method `pfcModel::ListDependencies` returns a list of the first-level dependencies for the specified model in the Creo workspace in the form of the `pfcDependencies` object.

Use the method `pfcModel::CleanupDependencies` to clean the dependencies for an object in the Creo workspace.

Note

Do not call the method `pfcModel::CleanupDependencies` during operations that alter the dependencies, such as, restructuring components and creating or redefining features.

The method `pfcModel::ListDeclaredModels` returns a list of all the first-level objects declared for the specified model.

The method `pfcModel::CheckIsModifiable` identifies if a given model can be modified without checking for any subordinate models. This method takes a boolean argument *ShowUI* that determines whether the Creo conflict resolution dialog box should be displayed to resolve conflicts, if detected. If this argument is false, then the conflict resolution dialog box is not displayed, and the model can be modified only if there are no conflicts that cannot be overridden, or are resolved by default resolution actions. For a generic model, if *ShowUI* is true, then all instances of the model are also checked.

The method `pfcModel::CheckIsSaveAllowed` identifies if a given model can be saved along with all of its subordinate models. The subordinate models can be saved based on their modification status and the value of the configuration option `save_objects`. This method also checks the current user interface context to identify if it is currently safe to save the model. Thus, calling this method at different times might return different results. This method takes a boolean argument *ShowUI*. Refer to the previous method for more information on this argument.

The method `wfcWModel::GetSubType` returns the subtype of the specified model using the enumerated data type `wfcModelSubType`. This is similar to selecting types and sub-types of models available in the Creo user interface using the command **File ▸ New**.

Model Operations

Methods Introduced:

- **pfcModel::Backup**
- **pfcModel::Copy**
- **pfcModel::CopyAndRetrieve**
- **pfcModel::Rename**
- **pfcModel::Save**

- **pfcModel::Erase**
- **pfcModel::EraseWithDependencies**
- **pfcModel::Delete**
- **pfcModel::Display**
- **pfcModel::SetCommonName**
- **wfcWModel::IsStandardLocation**

These model operations duplicate most of the commands available in the Creo **File** menu.

The method `pfcModel::Backup` makes a backup of an object in memory to a disk in a specified directory.

The method `pfcModel::Copy` copies the specified model to another file.

The method `pfcModel::CopyAndRetrieve` copies the model to another name, and retrieves that new model into session.

The method `pfcModel::Rename` renames a specified model.

The method `pfcModel::Save` stores the specified model to a disk.

The method `pfcModel::Erase` erases the specified model from the session. Models used by other models cannot be erased until the models dependent upon them are erased.

The method `pfcModel::EraseWithDependencies` erases the specified model from the session and all the models on which the specified model depends from disk, if the dependencies are not needed by other items in session.

Note

However, while erasing an active model, `pfcModel::Erase` and `pfcModel::EraseWithDependencies` only clear the graphic display immediately, they do not clear the data in the memory until the control returns to Creo application from the Creo Object TOOLKIT C++ application. Therefore, after calling them the control must be returned to Creo before calling any other function, otherwise the behavior of Creo may be unpredictable.

The method `pfcModel::Delete` removes the specified model from memory and disk.

The method `pfcModel::Display` displays the specified model. You must call this method if you create a new window for a model because the model will not be displayed in the window until you call `pfcDisplay`.

The method `pfcModel::SetCommonName` modifies the common name of the specified model. You can modify this name for models that are not yet owned by Windchill PDMLink, or in certain situations if the configuration option `let_proe_rename_pdm_objects` is set to `yes`.

The method `wfcWModel::IsStandardLocation` checks if the specified model was opened from a standard location. A standard file location can be the local disk or a mapped drive on a remote computer. The Universal Naming Convention (UNC) path for network drives is also considered as a standard path if the value for **DisableUNCCheck** is set to True for the key `HKEY_CURRENT_USER\Software\Microsoft\Command Processor`, in the registry file. The method returns:

- True when the file is loaded from a standard file location.
- False when the file is loaded from a nonstandard file location, such as, http, ftp, Design Exploration mode, and so on.

Running Creo Modelcheck

Creo ModelCHECK is an integrated application that runs transparently within Creo Parametric. It uses a configurable list of company design standards and best modeling practices. You can configure Creo ModelCHECK to run interactively or automatically when you regenerate or save a model.

Methods Introduced:

- **pfcBaseSession::ExecuteModelCheck**
- **pfcModelCheckInstructions::Create**
- **pfcModelCheckInstructions::SetConfigDir**
- **pfcModelCheckInstructions::SetMode**
- **pfcModelCheckInstructions::SetOutputDir**
- **pfcModelCheckInstructions::SetShowInBrowser**
- **pfcModelCheckResults::GetNumberOfErrors**
- **pfcModelCheckResults::GetNumberOfWarnings**
- **pfcModelCheckResults::GetWasModelSaved**

You can run Creo ModelCHECK from an external application using the method `pfcBaseSession::ExecuteModelCheck`. This method takes the model *Model* on which you want to run Creo ModelCHECK and instructions in the form of the object `ModelCheckInstructions` as its input parameters. This object contains the following parameters:

- `ConfigDir`—Specifies the location of the configuration files. If this parameter is set to `NULL`, the default Creo ModelCHECK configuration files are used.

- `Mode`—Specifies the mode in which you want to run Creo ModelCHECK. The modes are:
  - `MODELCHECK_GRAPHICS`—Interactive mode
  - `MODELCHECK_NO_GRAPHICS`—Batch mode
- `OutputDir`—Specifies the location for the reports. If you set this parameter to `NULL`, the default Creo ModelCHECK directory, as per `config_init.mc`, will be used.
- `ShowInBrowser`—Specifies if the results report should be displayed in the Web browser.

The method `pfcModelCheckInstructions::Create` creates the `pfcModelCheckInstructions` object containing the Creo ModelCHECK instructions described above.

Use the methods `pfcModelCheckInstructions::SetConfigDir`, `pfcModelCheckInstructions::SetMode`, `pfcModelCheckInstructions::SetOutputDir`, and `pfcModelCheckInstructions::SetShowInBrowser` to modify the Creo ModelCHECK instructions.

The method `pfcBaseSession::ExecuteModelCheck` returns the results of the Creo ModelCHECK run in the form of the `pfcModelCheckResults` object. This object contains the following parameters:

- `NumberOfErrors`—Specifies the number of errors detected.
- `NumberOfWarnings`—Specifies the number of warnings found.
- `WasModelSaved`—Specifies whether the model is saved with updates.

Use the methods `pfcModelCheckResults::GetNumberOfErrors`, `pfcModelCheckResults::GetNumberOfWarning`, and `pfcModelCheckResults::GetWasModelSaved` to access the results obtained.

Custom Checks

This section describes how to define custom checks in ModelCHECK that users can run using the standard Creo ModelCHECK interface in Creo Parametric.

To define and register a custom check:

1. Set the `CUSTMTK_CHECKS_FILE` configuration option in the start configuration file to a text file that stores the check definition. For example: `CUSTMTK_CHECKS_FILE text\custmtk_checks.txt`.
2. Set the contents of the `CUSTMTK_CHECKS_FILE` file to define the checks. Each check should list the following items:

- `DEF_<checkname>`—Specifies the name of the check. The format must be `CHKTK_<checkname>_<mode>`, where mode is `PRT`, `ASM`, or `DRW`.
- `TAB_<checkname>`—Specifies the tab category in the Creo ModelCHECK report under which the check is classified. Valid tab values are:
  - `INFO`
  - `PARAMETER`
  - `LAYER`
  - `FEATURE`
  - `RELATION`
  - `DATUM`
  - `MISC`
  - `VDA`
  - `VIEWS`
- `MSG_<checkname>`—Specifies the description of the check that appears in the lower part of the Creo ModelCHECK report when you select the name.
- `DSC_<checkname>`—Specifies the name of the check as it appears in the Creo ModelCHECK report table.
- `ERM_<checkname>`—If set to `INFO`, the check is considered an `INFO` check and the report table displays the text from the first item returned by the check, instead of a count of the items. Otherwise, this value must be included, but is ignored by Creo Parametric.

3. Add the check and its values to the Creo ModelCHECK configuration file.
4. Register the Creo ModelCHECK check from the Creo Object TOOLKIT C++ application.

Note

Other than the requirements listed above, Creo Object TOOLKIT C++ custom checks do not have access to the rest of the values in the Creo ModelCHECK configuration files. All the custom settings specific to the check, such as start parameters, constants, and so on, must be supported by the user application and not Creo ModelCHECK.

Registering Custom Checks

Methods Introduced:

- **pfcBaseSession::RegisterCustomModelCheck**
- **pfcCustomCheckInstructions::Create**
- **pfcCustomCheckInstructions::SetCheckName**
- **pfcCustomCheckInstructions::SetCheckLabel**
- **pfcCustomCheckInstructions::SetListener**
- **pfcCustomCheckInstructions::SetActionButtonLabel**
- **pfcCustomCheckInstructions::SetUpdateButtonLabel**

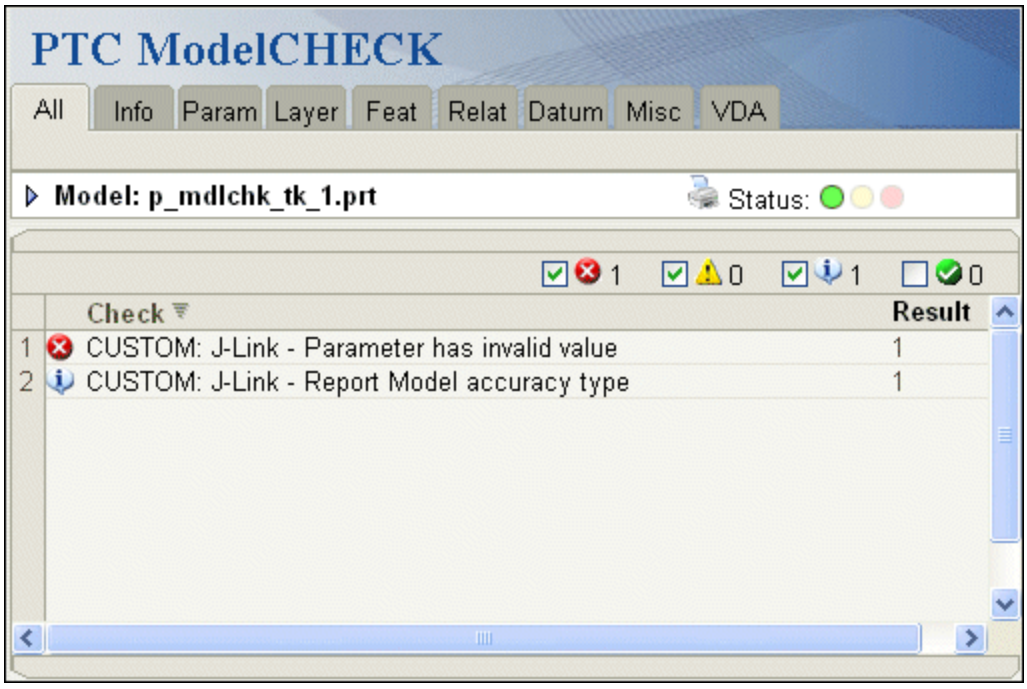
The method `pfcBaseSession::RegisterCustomModelCheck` registers a custom check that can be included in any Creo ModelCHECK run. This method takes the instructions in the form of the `pfcCustomCheckInstructions` object as its input argument. This object contains the following parameters:

- *CheckName*—Specifies the name of the custom check.
- *CheckLabel*—Specifies the label of the custom check.
- *Listener*—Specifies the listener object containing the custom check methods. Refer to the section Custom Check Listeners on page 116 for more information.
- *ActionButtonLabel*—Specifies the label for the action button. If you specify `NULL` for this parameter, this button is not shown.
- *UpdateButtonLabel*—Specifies the label for the update button. If you specify `NULL` for this parameter, this button is not shown.

The method `pfcCustomCheckInstructions::Create` creates the `pfcCustomCheckInstructions` object containing the custom check instructions described above.

Use the methods `pfcCustomCheckInstructions::SetCheckName`, `pfcCustomCheckInstructions::SetCheckLabel`, `pfcCustomCheckInstructions::SetListener`, `pfcCustomCheckInstructions::SetActionButtonLabel`, and `pfcCustomCheckInstructions::SetUpdateButtonLabel` to modify the instructions.

The following figure illustrates how the results of some custom checks might be displayed in the Creo ModelCHECK report.



Custom Check Listeners

Methods Introduced:

- **pfcModelCheckCustomCheckListener::OnCustomCheck**
- **pfcModelCheckCustomCheckListener::OnCustomCheckAction**
- **pfcModelCheckCustomCheckListener::OnCustomCheckUpdate**
- **pfcCustomCheckResults::Create**
- **pfcCustomCheckResults::SetResultsCount**
- **pfcCustomCheckResults::SetResultsTable**
- **pfcCustomCheckResults::SetResultsUrl**

The interface `pfcModelCheckCustomCheckListener` provides the method signatures to implement a custom Creo ModelCHECK check.

Each listener method takes the following input arguments:

- *CheckName*—The name of the custom check as defined in the original call to the method `pfcBaseSession::RegisterCustomModelCheck`
- *Mdl*—The model being checked.

The application method that overrides `pfcModelCheckCustomCheckListener::OnCustomCheck` is used to evaluate a custom defined check. The user application runs the check on the specified model and returns the results in the form of the `CustomCheckResults` object.

- *ResultsCount*—Specifies an integer indicating the number of errors found by the check. This value is displayed in the Creo ModelCHECK report generated.
- *ResultsTable*—Specifies a list of text descriptions of the problem encountered for each error or warning.
- *ResultsUrl*—Specifies the link to an application-owned page that provides information on the results of the custom check.

The method `pfcCustomCheckResults::Create` creates the `pfcCustomCheckResults` object containing the custom check results described above.

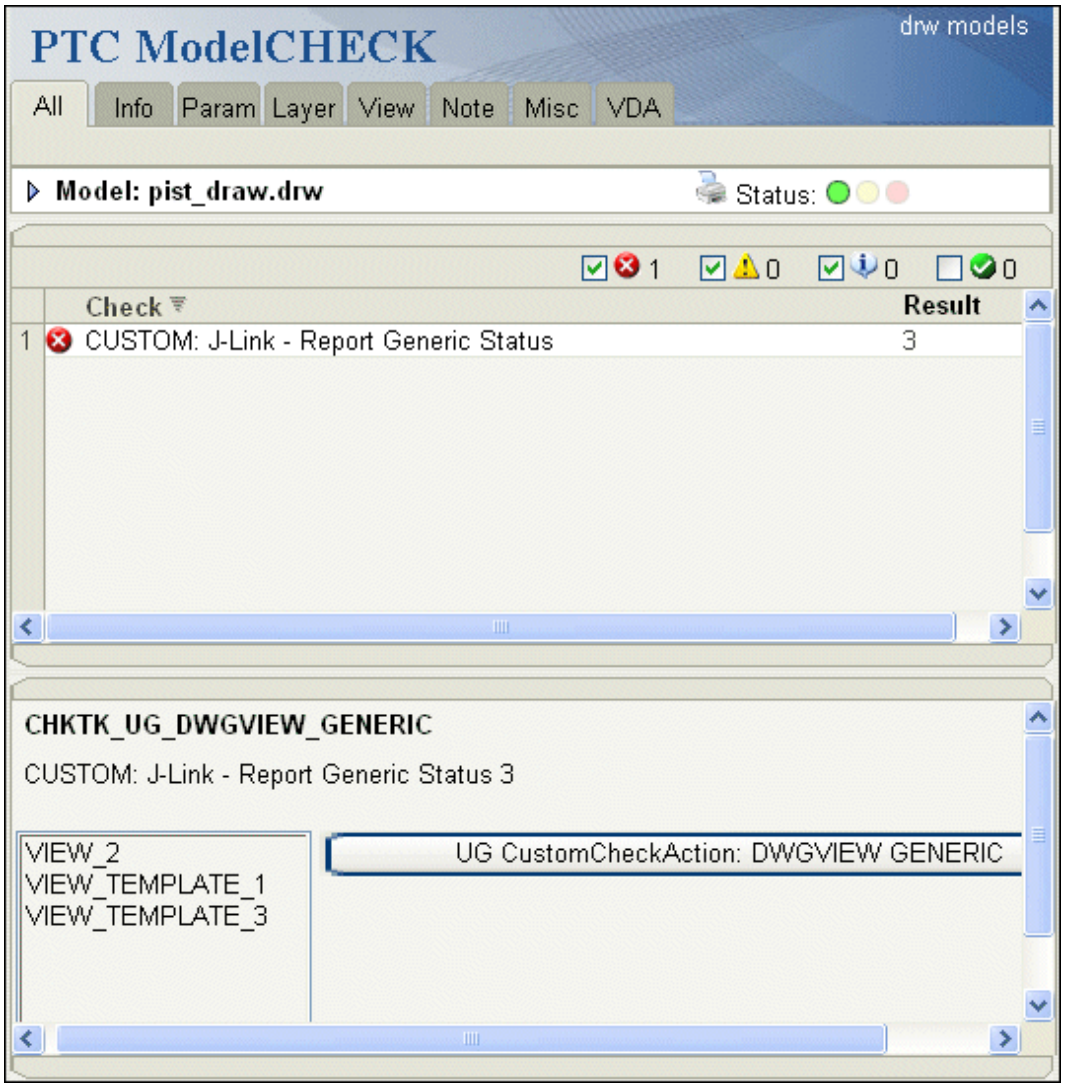
Use the methods `pfcCustomCheckResults.SetResultsCount`, `pfcCustomCheckResults::SetResultsTable`, and `pfcCustomCheckResults::SetResultsUrl` listed above to modify the custom checks results obtained.

The method that overrides `pfcModelCheckCustomCheckListener::OnCustomCheckAction` is called when the custom check's `Action` button is pressed. The input supplied includes the text selected by the user from the custom check results.

The function that overrides `pfcModelCheckCustomCheckListener::OnCustomCheckUpdate` is called when the custom check's `Update` button is pressed. The input supplied includes the text selected by the user from the custom check results.

Custom Creo ModelCHECK checks can have an `Action` button to highlight the problem, and possibly an `Update` button to fix it automatically.

The following figure displays the Creo ModelCHECK report with an `Action` button that invokes the `pfcModelCheckCustomCheckListener::OnCustomCheckAction`



Notebook

The methods described in this section work with notebook (.lay) files.

Methods introduced:

- **wfcWLayout::Declare**
- **wfcWLayout::Undeclare**

The method `wfcWLayout::Declare` declares a notebook name to the specified Creo Parametric model or notebook. You can resolve conflicts using the enumerated type `wfcLayoutDeclareOption`. It has the following values:

- *wfcLAYOUT_DECLARE_INTERACTIVE*— Resolves the conflict in interactive mode.

- *wfcLAYOUT_DECLARE_OBJECT_SYMBOLS*— Keep the symbols in the specified Creo Parametric model or notebook object.
- *wfcLAYOUT_DECLARE_LAYOUT_SYMBOLS* — Keep the symbols specified in the notebook.
- *wfcLAYOUT_DECLARE_ABORT*— abort the notebook declaration process and return an error.

Use the function `wfcWLayout::Undeclare` to undeclare the notebook name to the specified Creo Parametric model or notebook. You can resolve conflicts using the enumerated type `wfcLayoutUndeclareOption`. It has the following values:

- *wfcLAYOUT_UNDECLARE_FORCE*— Continues to undeclare the notebook even if references exist.
- *wfcLAYOUT_UNDECLARE_CANCEL*— Does not undeclare the notebook if references exist.

9

Drawings

This chapter describes how to program drawing functions using Creo Object TOOLKIT C++.

Overview of Drawings in Creo Object TOOLKIT C++

This section describes the functions that deal with drawings. You can create drawings of all Creo models using the functions in Creo Object TOOLKIT C++. You can annotate the drawing, manipulate dimensions, and use layers to manage the display of different items.

Unless otherwise specified, Creo Object TOOLKIT C++ functions that operate on drawings use world units.

Creating Drawings from Templates

Drawing templates simplify the process of creating a drawing using Creo Object TOOLKIT C++. Creo can create views, set the view display, create snap lines, and show the model dimensions based on the template. Use templates to:

- Define layout views
- Set view display
- Place notes
- Place symbols
- Define tables
- Show dimensions

Method Introduced:

- **pfcBaseSession::CreateDrawingFromTemplate**

Use the method `pfcBaseSession::CreateDrawingFromTemplate` to create a drawing from the drawing template and to return the created drawing. The attributes are:

- New drawing name
- Name of an existing template
- Name and type of the solid model to use while populating template views
- Sequence of options to create the drawing. The options are as follows:
  - `pfcDRAWINGCREATE_DISPLAY_DRAWING`—display the new drawing.
  - `pfcDRAWINGCREATE_SHOW_ERROR_DIALOG`—display the error dialog box.
  - `pfcDRAWINGCREATE_WRITE_ERROR_FILE`—write the errors to a file.
  - `pfcDRAWINGCREATE_PROMPT_UNKNOWN_PARAMS`—prompt the user on encountering unknown parameters

Drawing Creation Errors

Methods Introduced:

- **pfcXToolkitDrawingCreateErrors::GetErrors**
- **pfcDrawingCreateError::GetType**
- **pfcDrawingCreateError::GetViewName**
- **pfcDrawingCreateError::GetObjectName**
- **pfcDrawingCreateError::GetSheetNumber**
- **pfcDrawingCreateError::GetView**

The exception `pfcXToolkitDrawingCreateErrors` is thrown if an error is encountered when creating a drawing from a template. This exception contains a list of errors which occurred during drawing creation.

Note

When this exception type is encountered, the drawing is actually created, but some of the contents failed to generate correctly.

The error structure contains an array of drawing creation errors. Each error message may have the following elements:

- *Type*—The type of error as follows:
  - `pfcDWGCREATE_ERR_SAVED_VIEW_DOESNT_EXIST`—Saved view does not exist.
  - `pfcDWGCREATE_ERR_X_SEC_DOESNT_EXIST`—Specified cross section does not exist.
  - `pfcDWGCREATE_ERR_EXPLODE_DOESNT_EXIST`—Exploded state did not exist.
  - `pfcDWGCREATE_ERR_MODEL_NOT_EXPLODABLE`—Model cannot be exploded.
  - `pfcDWGCREATE_ERR_SEC_NOT_PERP`—Cross section view not perpendicular to the given view.
  - `pfcDWGCREATE_ERR_NO_RPT_REGIONS`—Repeat regions not available.
  - `pfcDWGCREATE_ERR_FIRST_REGION_USED`—Repeat region was unable to use the region specified.
  - `pfcDWGCREATE_ERR_NOT_PROCESS_ASSEM`— Model is not a process assembly view.
  - `pfcDWGCREATE_ERR_NO_STEP_NUM`—The process step number does not exist.

- pfcDWGCREATE_ERR_TEMPLATE_USED—The template does not exist.
- pfcDWGCREATE_ERR_NO_PARENT_VIEW_FOR_PROJ—There is no possible parent view for this projected view.
- pfcDWGCREATE_ERR_CANT_GET_PROJ_PARENT—Could not get the projected parent for a drawing view.
- pfcDWGCREATE_ERR_SEC_NOT_PARALLEL—The designated cross section was not parallel to the created view.
- pfcDWGCREATE_ERR_SIMP_REP_DOESNT_EXIST—The designated simplified representation does not exist.

- *ViewName*—Name of the view where the error occurred.
- *SheetNumber*—Sheet number where the error occurred.
- *ObjectName*—Name of the invalid or missing object.
- *View*—2D view in which the error occurred.

Use the method pfcXToolkitDrawingCreateErrors::GetErrors to obtain the preceding array elements from the error object.

Obtaining Drawing Models

This section describes how to obtain drawing models.

Methods Introduced:

- **pfcBaseSession::RetrieveModel**
- **pfcBaseSession::GetModel**
- **pfcBaseSession::GetModelFromDescr**
- **pfcBaseSession::ListModels**
- **pfcBaseSession::ListModelsByType**

The method pfcBaseSession::RetrieveModel retrieves the drawing specified by the model descriptor. Model descriptors are data objects used to describe a model file and its location in the system. The method returns the retrieved drawing.

The method pfcBaseSession::GetModel returns a drawing based on its name and type, whereas pfcBaseSession::GetModelFromDescr returns a drawing specified by the model descriptor. The model must be in session.

Use the method pfcBaseSession::ListModels to return a sequence of all the drawings in session.

Drawing Information

Methods Introduced:

- **pfcModel2D::ListModels**
- **pfcModel2D::GetCurrentSolid**
- **pfcModel2D::ListSimplifiedReps**
- **pfcModel2D::GetTextHeight**

The method `pfcModel2D::ListModels` returns a list of all the solid models used in the drawing.

The method `pfcModel2D::GetCurrentSolid` returns the current solid model of the drawing.

The method `pfcModel2D::ListSimplifiedReps` returns the simplified representations of a solid model that are assigned to the drawing.

The method `pfcModel2D::GetTextHeight` returns the text height of the drawing.

Access Drawing Location in Grid

Method introduced:

- **wfcWModel2D::GetLocationGridColumnFromPosition**
- **wfcWModel2D::GetLocationGridRowFromPosition**

Use the methods `wfcWModel2D::GetLocationGridColumnFromPosition` and `wfcWModel2D::GetLocationGridRowFromPosition` to find the grid coordinates of a location in the drawing. This method returns strings representing the row and column containing the point. The method specifies the position of a point, expressed in screen coordinates. This method returns strings representing the row and column containing the point.

Drawing Tree

Methods introduced:

- **wfcWModel2D::CollapseTree**
- **wfcWModel2D::ExpandTree**
- **wfcWModel2D::RefreshTree**

Use the method `wfcWModel2D::CollapseTree` to collapse all nodes of the drawing tree in the Creo Parametric window and make its child nodes invisible. The input argument *WindowId* returns the Id of the window which contains the drawing. Use -1 to expand the drawing tree in the active window.

Use the method `wfcWModel2D::ExpandTree` to expand the drawing tree in the Creo Parametric window and make all drawing sheets and drawing items in the active drawing sheet visible.

Use the method `wfcWModel2D::RefreshTree` to rebuild the drawing tree in the Creo Parametric window that contains the drawing. You can use this function after adding a single drawing item or multiple drawing items to a drawing.

Note

Specify the input argument *WindowId* as -1 to refresh, expand, or collapse the drawing tree in the active window.

Drawing Operations

Methods Introduced:

- **pfcModel2D::AddModel**
- **pfcModel2D::DeleteModel**
- **pfcModel2D::ReplaceModel**
- **pfcModel2D::SetCurrentSolid**
- **pfcModel2D::AddSimplifiedRep**
- **pfcModel2D::DeleteSimplifiedRep**
- **pfcModel2D::Regenerate**
- **pfcModel2D::SetTextHeight**
- **pfcModel2D::CreateDrawingDimension**
- **pfcModel2D::CreateView**

The method `pfcModel2D::AddModel` adds a new solid model to the drawing.

The method `pfcModel2D::DeleteModel` removes a model from the drawing. The model to be deleted should not appear in any of the drawing views.

The method `pfcModel2D::ReplaceModel` replaces a model in the drawing with a related model (the relationship should be by family table or interchange assembly). It allows you to replace models that are shown in drawing views and regenerates the view.

The method `pfcModel2D::SetCurrentSolid` assigns the current solid model for the drawing. Before calling this method, the solid model must be assigned to the drawing using the method `pfcModel2D::AddModel`. To see the changes to parameters and fields reflecting the change of the current solid model, regenerate the drawing using the method `pfcSheetOwner::RegenerateSheet`.

The method `pfcModel2D::AddSimplifiedRep` associates the drawing with the simplified representation of an assembly.

The method `pfcModel2D::DeleteSimplifiedRep` removes the association of the drawing with an assembly simplified representation. The simplified representation to be deleted should not appear in any of the drawing views.

Use the method `pfcModel2D::Regenerate` to regenerate the drawing draft entities and appearance.

The method `pfcModel2D::SetTextHeight` ets the value of the text height of the drawing.

The method `pfcModel2D::CreateDrawingDimension` creates a new drawing dimension based on the data object that contains information about the location of the dimension. This method returns the created dimension. Refer to the section Drawing Dimensions on page 140.

The method `pfcModel2D::CreateView` creates a new drawing view based on the data object that contains information about how to create the view. The method returns the created drawing view. Refer to the section Creating Drawing Views on page 133.

Merge Drawings

You can merge two drawings to enhance the performance and management of large drawings. Individual Creo Parametric users can work in parallel and then merge their separate drawings into a single drawing file.

Methods Introduced:

- **wfcWDrawing::Merge**

Use the method `wfcWDrawing::Merge` to merge two drawings.

Drawing Sheets

A drawing sheet is represented by its number. Drawing sheets in Creo Object TOOLKIT C++ are identified by the same sheet numbers seen by a Creo user.

Note

These identifiers may change if the sheets are moved as a consequence of adding, removing or reordering sheets.

Drawing Sheet Information

Methods Introduced:

- **wfcWDrawing::GetSheetName**
- **pfcSheetOwner::GetSheetTransform**
- **pfcSheetOwner::GetSheetInfo**
- **pfcSheetOwner::GetSheetScale**
- **pfcSheetOwner::GetSheetFormat**
- **pfcSheetOwner::GetSheetFormatDescr**
- **wfcWDrawing::GetFormatSheet**
- **pfcSheetOwner::GetSheetBackgroundView**
- **pfcSheetOwner::GetNumberOfSheets**
- **pfcSheetOwner::GetCurrentSheetNumber**
- **pfcSheetOwner::GetSheetUnits**

Superseded Method:

- **pfcSheetOwner::GetSheetData**

The method `wfcWDrawing::GetSheetName` retrieves the name of the specified drawing sheet.

The method `pfcSheetOwner::GetSheetTransform` returns the transformation matrix for the sheet specified by the sheet number. This transformation matrix includes the scaling needed to convert screen coordinates to drawing coordinates (which use the designated drawing units).

The method `pfcSheetOwner::GetSheetData` and the interface `pfcSheetData` have been deprecated. Use the method `pfcSheetOwner::GetSheetInfo` and the interface `pfcSheetInfo` instead.

The method `pfcSheetOwner::GetSheetInfo` returns sheet data including the size, orientation, and units of the sheet specified by the sheet number.

The method `pfcSheetOwner::GetSheetScale` returns the scale of the drawing on a particular sheet based on the drawing model used to measure the scale. If no models are used in the drawing then the default scale value is 1.0.

The method `pfcSheetOwner::GetSheetFormat` returns the drawing format used for the sheet specified by the sheet number. It returns a null value if no format is assigned to the sheet.

The method `pfcSheetOwner::GetSheetFormatDescr` returns the model descriptor of the drawing format used for the specified drawing sheet.

drawing format for a specified drawing and sheet number within the drawing . A drawing format file can have two sheets defining two formats. You can create drawings using either of the sheets as drawing format. If such a drawing format file is used to create the drawing, the method `wfcWDrawing::GetFormatSheet` retrieves the sheet number of the drawing format which was used to create the specified drawing.

The method `pfcSheetOwner::GetSheetBackgroundView` returns the view object representing the background view of the sheet specified by the sheet number.

The method `pfcSheetOwner::GetNumberOfSheets` returns the number of sheets in the model.

The method `pfcSheetOwner::GetCurrentSheetNumber` returns the current sheet number in the model.

Note

The sheet numbers range from 1 to n, where n is the number of sheets.

The method `pfcSheetOwner::GetSheetUnits` returns the units used by the sheet specified by the sheet number.

Drawing Sheet Operations

Methods Introduced:

- **pfcSheetOwner::AddSheet**
- **pfcSheetOwner::DeleteSheet**
- **pfcSheetOwner::ReorderSheet**
- **pfcSheetOwner::RegenerateSheet**
- **pfcSheetOwner::SetSheetScale**
- **pfcSheetOwner::SetSheetFormat**
- **pfcSheetOwner::SetCurrentSheetNumber**
- **wfcWModel2D::CopyDrawingSheet**
- **wfcWModel2D::ShowSheetFormat**
- **wfcWModel2D::IsSheetFormatBlanked**
- **wfcWModel2D::IsSheetFormatShown**
- **wfcWModel2D::GetToleranceStandard**
- **wfcWModel2D::SetToleranceStandard**
- **wfcWModel2D::CreateLeaderWithArrowTypeNote**

The method `pfcSheetOwner::AddSheet` adds a new sheet to the model and returns the number of the new sheet.

The method `pfcSheetOwner::DeleteSheet` removes the sheet specified by the sheet number from the model.

Use the method `pfcSheetOwner::ReorderSheet` to reorder the sheet from a specified sheet number to a new sheet number.

Note

The sheet number of other affected sheets also changes due to reordering or deletion.

The method `pfcSheetOwner::RegenerateSheet` regenerates the sheet specified by the sheet number.

Note

You can regenerate a sheet only if it is displayed.

Use the method `pfcSheetOwner::SetSheetScale` to set the scale of a model on the sheet based on the drawing model to scale and the scale to be used. Pass the value of the *DrawingModel* parameter as null to select the current drawing model.

Use the method `pfcSheetOwner::SetSheetFormat` to apply the specified format to a drawing sheet based on the drawing format, sheet number of the format, and the drawing model.

The sheet number of the format is specified by the *FormatSheetNumber* parameter. This number ranges from 1 to the number of sheets in the format. Pass the value of this parameter as null to use the first format sheet.

The drawing model is specified by the *DrawingModel* parameter. Pass the value of this parameter as null to select the current drawing model.

The method `pfcSheetOwner::SetCurrentSheetNumber` sets the current sheet to the sheet number specified.

The method `wfcWModel2D::CopyDrawingSheet` creates a copy of a specified drawing sheet. Specify the sheet number of the sheet to be copied in the input argument *sheet*. Set it to a value less than 1 to create a copy of the currently selected sheet.

Use the method `wfcWModel2D::ShowSheetFormat` to display or hide the drawing format for a specified drawing sheet. The input arguments are:

- *show*—Specifies if the drawing format must be displayed. Pass true to display the drawing format.
- *sheet*—Specifies the sheet number of the sheet. Set it to a value less than 1 to display or hide the drawing format of the currently selected sheet.

The method `wfcWModel2D::IsSheetFormatBlanked` checks if the drawing format of a specified drawing sheet is blank. In case of the current sheet, set the input argument *sheet* to a value less than 1.

The method `wfcWModel2D::IsSheetFormatShown` checks if the drawing format of a specified drawing sheet is shown. The output argument *sheet* returns the value as `True`, if the drawing format is shown and returns the value as `False`, if the drawing format is not shown.

The method `wfcWModel2D::GetToleranceStandard` returns the tolerance standard that is assigned to the specified drawing.

Use the method `wfcWModel2D::SetToleranceStandard` to set the tolerance standard for a drawing.

The method `wfcWModel2D::CreateLeaderWithArrowTypeNote` enables you to create a note with a leader and the specified arrow type in a specified drawing. The input arguments are:

- *TextLines*—Specifies the text lines using the `pfcDetailTextLines` object.
- *NoteAttach*—Specifies the details of the attachment of the note using the `pfcAttachment` object.
- *LeaderAttachs*—Specifies the leaders attached to the note using the `pfcDetailLeaderAttachments` object.
- *Types*—Specifies the types of arrowheads used for leaders attached to the note using the `wfcLeaderArrowTypes` object.

Drawing Format Files

The format of a drawing refers to the boundary lines, referencing marks and graphic elements that appear on every sheet before any drawing elements are shown or added. These usually include items such as tables for the company name, detailers name, revision number and date. In a Creo Parametric drawing, you can associate a format file (.frm) with the drawing. This file carries all the format graphical information, and it can also carry some optional default attributes like text size and draft scale. The functions described in this section allow you to get and set the size of the drawing format.

Methods Introduced:

- **wfcWModel2D::GetFormatSize**

- **wfcWModel2D::SetFormatSize**
- **wfcFormatSizeData::Create**
- **wfcFormatSizeData::GetPaperSize**
- **wfcFormatSizeData::SetPaperSize**
- **wfcFormatSizeData::GetWidth**
- **wfcFormatSizeData::SetWidth**
- **wfcFormatSizeData::GetHeight**
- **wfcFormatSizeData::SetHeight**

The methods `wfcWModel2D::GetFormatSize` and `wfcWModel2D::SetFormatSize` retrieve and set the size of the drawing format in the specified drawing as a`wfcModel.FormatSizeData` object. You can add a standard or customize size format in the drawing.

Use the method `wfcFormatSizeData::Create` to create a drawing format in a specified drawing. The input arguments are:

- *PaperSize*—Specifies the size of the drawing using the enumerated data type `pfcPlotPaperSize`.
- *Width*—Specifies the width of the drawing in inches, when *PaperSize* is set to `pfcVARIABLESIZEPLOT`.

  It specifies the width of the drawing in millimeters, when size is set to `pfcVARIABLESIZE_IN_MM_PLOT`.

  Note

  This argument is ignored for all the other sizes of the drawing except `pfcVARIABLESIZEPLOT` and `pfcVARIABLESIZE_IN_MM_PLOT`.

- *Height*—Specifies the height of the drawing in inches, when the size is set to `pfcVARIABLESIZEPLOT`.

The methods `wfcFormatSizeData::GetPaperSize` and `wfcFormatSizeData::SetPaperSize` get and set the size of the drawing using the enumerated data type `pfcModel.PlotPaperSize`.

The methods `wfcFormatSizeData::GetWidth` and `wfcFormatSizeData::SetWidth` get and set the width of the drawing.

The methods `wfcFormatSizeData::GetHeight` and `wfcFormatSizeData::SetHeight` get and set the height of the drawing.

Drawing Views

A drawing view is represented by the interface `pfcView2D`. All model views in the drawing are associative, that is, if you change a dimensional value in one view, the system updates other drawing views accordingly. The model automatically reflects any dimensional changes that you make to a drawing. In addition, corresponding drawings also reflect any changes that you make to a model such as the addition or deletion of features and dimensional changes.

Creating Drawing Views

Method Introduced:

- **pfcModel2D::CreateView**

The method `pfcModel2D::CreateView` reates a new view in the drawing. Before calling this method, the drawing must be displayed in a window.

The interface `pfcView2DCreateInstructions` contains details on how to create the view. The types of drawing views supported for creation are:

- `pfcDRAWVIEW_GENERAL`—General drawing views
- `pfcDRAWVIEW_PROJECTION`—Projected drawing views

General Drawing Views

The interface `pfcGeneralViewCreateInstructions` contains details on how to create general drawing views.

Methods Introduced:

- **pfcGeneralViewCreateInstructions::Create**
- **pfcGeneralViewCreateInstructions::SetViewModel**
- **pfcGeneralViewCreateInstructions::SetLocation**
- **pfcGeneralViewCreateInstructions::SetSheetNumber**
- **pfcGeneralViewCreateInstructions::SetOrientation**
- **pfcGeneralViewCreateInstructions::SetExploded**
- **pfcGeneralViewCreateInstructions::SetScale**
- **pfcGeneralViewCreateInstructions::GetViewScale**
- **pfcGeneralViewCreateInstructions::SetViewScale**

The method `pfcGeneralViewCreateInstructions::Create` creates the `pfcGeneralViewCreateInstructions` object used for creating general drawing views.

Use the method `pfcGeneralViewCreateInstructions::SetViewModel` to assign the solid model to display in the created general drawing view.

Use the method `pfcGeneralViewCreateInstructions::SetLocation` to assign the location in a drawing sheet to place the created general drawing view.

Use the method `pfcGeneralViewCreateInstructions::SetSheetNumber` to set the number of the drawing sheet in which the general drawing view is created.

The method `pfcGeneralViewCreateInstructions::SetOrientation` assigns the orientation of the model in the general drawing view in the form of the `pfcTransform3D` data object. The transformation matrix must only consist of the rotation to be applied to the model. It must not consist of any displacement or scale components. If necessary, set the displacement to {0, 0, 0} using the method `pfcTransform3D::SetOrigin`, and remove any scaling factor by normalizing the matrix.

Use the method `pfcGeneralViewCreateInstructions::SetExploded` to set the created general drawing view to be an exploded view.

Use the method `pfcGeneralViewCreateInstructions::SetScale` to assign a scale to the created general drawing view. This value is optional, if not assigned, the default drawing scale is used.

The methods `pfcGeneralViewCreateInstructions::GetViewScale` and `pfcGeneralViewCreateInstructions::SetViewScale` get and set a scale to the created general drawing view. This value is optional, if not assigned, the default drawing scale is used.

Projected Drawing Views

The interface `pfcProjectionViewCreateInstructions` contains details on how to create general drawing views.

Methods Introduced:

- **pfcProjectionViewCreateInstructions::Create**
- **pfcProjectionViewCreateInstructions::SetParentView**
- **pfcProjectionViewCreateInstructions::SetLocation**
- **pfcProjectionViewCreateInstructions::SetExploded**

The method `pfcProjectionViewCreateInstructions::Create` creates the `pfcProjectionViewCreateInstructions` data object used for creating projected drawing views.

Use the method `pfcProjectionViewCreateInstructions::SetParentView` to assign the parent view for the projected drawing view.

Use the method `pfcProjectionViewCreateInstructions::SetLocation` to assign the location of the projected drawing view. This location determines how the drawing view will be oriented.

Use the method `pfcProjectionViewCreateInstructions::SetExploded` to set the created projected drawing view to be an exploded view.

Obtaining Drawing Views

Methods Introduced:

- **pfcSelection::GetSelView2D**
- **pfcModel2D::List2DViews**
- **pfcModel2D::GetViewByName**
- **pfcModel2D::GetViewDisplaying**
- **pfcSheetOwner::GetSheetBackgroundView**

The method `pfcSelection::GetSelView2D` returns the selected drawing view (if the user selected an item from a drawing view). It returns a null value if the selection does not contain a drawing view.

The method `pfcModel2D::List2DViews` lists and returns the drawing views found. This method does not include the drawing sheet background views returned by the method `pfcSheetOwner::GetSheetBackgroundView`.

The method `pfcModel2D::GetViewByName` returns the drawing view based on the name. This method returns a null value if the specified view does not exist.

The method `pfcModel2D::GetViewDisplaying` returns the drawing view that displays a dimension. This method returns a null value if the dimension is not displayed in the drawing.

Note

This method works for solid and drawing dimensions.

The method `pfcSheetOwner::GetSheetBackgroundView` returns the drawing sheet background views.

Drawing View Information

Methods Introduced:

- **pfcChild::GetDBParent**

- **pfcView2D::GetSheetNumber**
- **pfcView2D::GetIsBackground**
- **pfcView2D::GetModel**
- **pfcView2D::GetScale**
- **pfcView2D::GetIsScaleUserdefined**
- **pfcView2D::GetOutline**
- **pfcView2D::GetLayerDisplayStatus**
- **pfcView2D::GetIsViewdisplayLayerDependent**
- **pfcView2D::GetDisplay**
- **pfcView2D::GetTransform**
- **pfcView2D::GetName**
- **pfcView2D::GetSimpRep**

The inherited method `pfcChild::GetDBParent`, when called on a `pfcView2D` object, provides the drawing model which owns the specified drawing view. The return value of the method can be downcast to a `pfcModel2D` object.

Note

The method `pfcChild::GetOId` is reserved for internal use.

The method `pfcView2D::GetSheetNumber` returns the sheet number of the sheet that contains the drawing view.

The method `pfcView2D::GetIsBackground` returns a value that indicates whether the view is a background view or a model view.

The method `pfcView2D::GetModel` returns the solid model displayed in the drawing view.

The method `pfcView2D::GetScale` returns the scale of the drawing view.

The method `pfcView2D::GetIsScaleUserdefined` specifies if the drawing has a user-defined scale.

The method `pfcView2D::GetOutline` returns the position of the view in the sheet in world units.

The method `pfcView2D::GetLayerDisplayStatus` returns the display status of the specified layer in the drawing view.

The method `pfcView2D::GetDisplay` returns an output structure that describes the display settings of the drawing view. The fields in the structure are as follows:

- *Style*—Whether to display as wireframe, hidden lines, no hidden lines, or shaded
- *TangentStyle*—Linestyle used for tangent edges
- *CableStyle*—Linestyle used to display cables
- *RemoveQuiltHiddenLines*—Whether or not to apply hidden-line-removal to quilts
- *ShowConceptModel*—Whether or not to display the skeleton
- *ShowWeldXSection*—Whether or not to include welds in the cross-section

The method `pfcView2D::GetTransform` returns a matrix that describes the transform between 3D solid coordinates and 2D world units for that drawing view. The transformation matrix is a combination of the following factors:

- The location of the view origin with respect to the drawing origin.
- The scale of the view units with respect to the drawing units
- The rotation of the model with respect to the drawing coordinate system.

The method `pfcView2D::GetName` returns the name of the specified view in the drawing.

The simplified representations of assembly and part can be used as drawing models to create general views. Use the method `pfcView2D::GetSimpRep` to retrieve the simplified representation for the specified view in the drawing.

Drawing View Display Information

Methods Introduced:

- **wfcDrawingViewDisplay::Create**
- **wfcDrawingViewDisplay::GetCableDisp**
- **wfcDrawingViewDisplay::SetCableDisp**
- **wfcDrawingViewDisplay::GetConceptModel**
- **wfcDrawingViewDisplay::SetConceptModel**
- **wfcDrawingViewDisplay::GetDispStyle**
- **wfcDrawingViewDisplay::SetDispStyle**
- **wfcDrawingViewDisplay::GetQuiltHLR**
- **wfcDrawingViewDisplay::SetQuiltHLR**
- **wfcDrawingViewDisplay::GetTanEdgeDisplay**
- **wfcDrawingViewDisplay::SetTanEdgeDisplay**
- **wfcDrawingViewDisplay::GetWeldXSec**
- **wfcDrawingViewDisplay::SetWeldXSec**

The method wfcDrawingViewDisplay::Create creates a new instance of the wfcDrawingViewDisplay object that contains information about the display styles being used in a view.

The methods wfcDrawingViewDisplay::GetCableDisp and wfcDrawingViewDisplay::SetCableDisp get and set the style used to display the cables. You can set the cable style using the object wfcCableDisplay:

- wfcCABLEDISP_DEFAULT—Uses the display setting from **Creo Parametric Options** dialog box, under **Entity Display**, **Cable display settings**.
- wfcCABLEDISP_CENTERLINE—Displays centerlines of cables and wires.
- wfcCABLEDISP_THICK—Displays cables and wires as a thick lines.
- wfcCableDisplay_nil—NULL value.

In a view, you can define whether to show or hide the skeleton model. The methods wfcDrawingViewDisplay::GetConceptModel and wfcDrawingViewDisplay::SetConceptModel get and set the status of skeleton models. If you set the value as True, then the skeleton model is displayed.

The methods wfcDrawingViewDisplay::GetDispStyle and wfcDrawingViewDisplay::SetDispStyle get and set the display style of the model geometry. You can set the display style using the object pfcDisplayStyle:

- pfcDISPSTYLE_DEFAULT—When you import drawings from Pro/ENGINEER Wildfire 2.0 or earlier releases that were saved with the **Default** option, this option is retained for these drawings. Once you update these drawings in Pro/ENGINEER Wildfire 3.0 and later releases, the wfcDISPSTYLE_DEFAULT option changes to wfcDISPSTYLE_FOLLOW_ENVIRONMENT.
- pfcDISPSTYLE_WIREFRAME—Shows all edges in wireframe style.
- pfcDISPSTYLE_HIDDEN_LINE—Shows all edges in hidden line style.
- pfcDISPSTYLE_NO_HIDDEN—Removes all hidden edge from view display.
- pfcDISPSTYLE_SHADED—Shows the view in shaded display mode.
- pfcDISPSTYLE_FOLLOW_ENVIRONMENT—Uses the current configuration settings for display.
- pfcDISPSTYLE_SHADED_WITH_EDGES—Shows the model as a shaded solid along with its edges.

You can remove the hidden lines in a quilt. The methods wfcDrawingViewDisplay::GetQuiltHLR and wfcDrawingViewDisplay::SetQuiltHLR get and set the hidden line removal property in quilts.

The methods `wfcDrawingViewDisplay::GetTanEdgeDisplay` and `wfcDrawingViewDisplay::SetTanEdgeDisplay` get and set the display style for tangent edges. You can set the tangent edge display style using the object `wfcTangentEdgeDisplay`:

- `wfcTANEDGE_DEFAULT`—Uses the default settings.
- `wfcTANEDGE_NONE`—Turns off the display of tangent edges
- `wfcTANEDGE_CENTERLINE`—Displays tangent edges in centerline font.
- `wfcTANEDGE_PHANTOM`—Displays tangent edges in phantom font.
- `wfcTANEDGE_DIMMED`—Displays tangent edges in dimmed system color.
- `wfcTANEDGE_SOLID`—Displays tangent edges as solid lines.

You can show and hide the weld cross-sections in a drawing. The methods `wfcDrawingViewDisplay::GetWeldXSec` and `wfcDrawingViewDisplay::SetWeldXSec` get and set the display of weld cross-sections in a drawing.

Drawing Views Operations

Methods Introduced:

- **pfcView2D::SetScale**
- **pfcView2D::Translate**
- **pfcView2D::Delete**
- **pfcView2D::Regenerate**
- **pfcView2D::SetLayerDisplayStatus**
- **pfcView2D::SetDisplay**

The method `pfcView2D::SetScale` sets the scale of the drawing view.

The method `pfcView2D::Translate` moves the drawing view by the specified transformation vector.

The method `pfcView2D::Delete` deletes a specified drawing view. Set the *DeleteChildren* parameter to `true` to delete the children of the view. Set this parameter to false or null to prevent deletion of the view if it has children.

The method `pfcView2D::Regenerate` erases the displayed view of the current object, regenerates the view from the current drawing, and redisplays the view.

The method `pfcView2D::SetLayerDisplayStatus` sets the display status for the layer in the drawing view.

The method `pfcView2D::SetDisplay` sets the value of the display settings for the drawing view.

Drawing Dimensions

This section describes the methods that give access to the types of dimensions that can be created in the drawing mode. They do not apply to dimensions created in the solid mode, either those created automatically as a result of feature creation, or reference dimension created in a solid. A drawing dimension or a reference dimension shown in a drawing is represented by the interface `pfcDimension2D`.

Obtaining Drawing Dimensions

Methods Introduced:

- **pfcModelItemOwner::ListItems**
- **pfcModelItemOwner::GetItemById**
- **pfcSelection::GetSelItem**

The method `pfcModelItemOwner::ListItems` returns a list of drawing dimensions specified by the parameter *Type* or returns null if no drawing dimensions of the specified type are found. This method lists only those dimensions created in the drawing.

The values of the parameter *Type* for the drawing dimensions are:

- `pfcITEM_DIMENSION`—Dimension
- `pfcITEM_REF_DIMENSION`—Reference dimension

Set the parameter *Type* to the type of drawing dimension to retrieve. If this parameter is set to null, then all the dimensions in the drawing are listed.

The method `pfcModelItemOwner::GetItemById` returns a drawing dimension based on the type and the integer identifier. The method returns only those dimensions created in the drawing. It returns a null if a drawing dimension with the specified attributes is not found.

The method `pfcSelection::GetSelItem` returns the value of the selected drawing dimension.

Creating Drawing Dimensions

Methods Introduced:

- **pfcDrawingDimCreateInstructions::Create**
- **pfcModel2D::CreateDrawingDimension**
- **pfcEmptyDimensionSense::Create**
- **pfcPointDimensionSense::Create**
- **pfcSplinePointDimensionSense::Create**
- **pfcTangentIndexDimensionSense::Create**

- **pfcLinAOCTangentDimensionSense::Create**
- **pfcAngleDimensionSense::Create**
- **pfcPointToAngleDimensionSense::Create**

The method `pfcDrawingDimCreateInstructions::Create` creates an instructions object that describes how to create a drawing dimension using the method `pfcModel2D::CreateDrawingDimension`.

The parameters of the instruction object are:

- *Attachments*—The entities that the dimension is attached to. The selections should include the drawing model view.
- *IsRefDimension*—True if the dimension is a reference dimension, otherwise null or false.
- *OrientationHint*—Describes the orientation of the dimensions in cases where this cannot be deduced from the attachments themselves.
- *Senses*—Gives more information about how the dimension attaches to the entity, i.e., to what part of the entity and in what direction the dimension runs. The types of dimension senses are as follows:
  - `pfcDIMSENSE_NONE`
  - `pfcDIMSENSE_POINT`
  - `pfcDIMSENSE_SPLINE_PT`
  - `pfcDIMSENSE_TANGENT_INDEX`
  - `pfcDIMSENSE_LINEAR_TO_ARC_OR_CIRCLE_TANGENT`
  - `pfcDIMSENSE_ANGLE`
  - `pfcDIMSENSE_POINT_TO_ANGLE`
- *TextLocation*—The location of the dimension text, in world units.

The method `pfcModel2D::CreateDrawingDimension` creates a dimension in the drawing based on the instructions data object that contains information needed to place the dimension. It takes as input an array of `pfcSelection` objects and an array of `pfcDimensionSense` structures that describe the required attachments. The method returns the created drawing dimension.

The method `pfcEmptyDimensionSense::Create` creates a new dimension sense associated with the type `pfcDIMSENSE NONE`. The `sense` field is set to *Type* In this case no information such as location or direction is needed to describe the attachment points. For example, if there is a single attachment which is a straight line, the dimension is the length of the straight line. If the attachments are two parallel lines, the dimension is the distance between them.

The method `pfcPointDimensionSense::Create` creates a new dimension sense associated with the type `pfcDIMSENSE POINT` which specifies the part of the entity to which the dimension is attached. The `sense` field is set to the value of the parameter *PointType*.

The possible values of *PointType* are:

- `pfcDIMPOINT_END1`— The first end of the entity
- `pfcDIMPOINT_END2`—The second end of the entity
- `pfcDIMPOINT_CENTER`—The center of an arc or circle
- `pfcDIMPOINT_NONE`—No information such as location or direction of the attachment is specified. This is similar to setting the *PointType* to `pfcDIMSENSE NONE`.
- `pfcDIMPOINT_MIDPOINT`—The mid point of the entity

The method `pfcSplinePointDimensionSense::Create` creates a dimension sense associated with the type `pfcDIMSENSE_SPLINE_PT`. This means that the attachment is to a point on a spline. The `sense` field is set to *SplinePointIndex* i.e., the index of the spline point.

The method `pfcTangentIndexDimensionSense::Create` creates a new dimension sense associated with the type `pfcDIMSENSE_TANGENT_INDEX`. The attachment is to a tangent of the entity, which is an arc or a circle. The `sense` field is set to *TangentIndex*, i.e., the index of the tangent of the entity.

The method `pfcLinAOCTangentDimensionSense::Create` creates a new dimension sense associated with the type `pfcDIMSENSE_LINEAR_TO_ARC_OR_CIRCLE_TANGENT`. The dimension is the perpendicular distance between the a line and a tangent to an arc or a circle that is parallel to the line. The `sense` field is set to the value of the parameter *TangentType*.

The possible values of *TangentType* are:

- `pfcDIMLINAOCTANGENT_LEFT0`—The tangent is to the left of the line, and is on the same side, of the center of the arc or circle, as the line.
- `pfcDIMLINAOCTANGENT_RIGHT0`—The tangent is to the right of the line, and is on the same side, of the center of the arc or circle, as the line.
- `pfcDIMLINAOCTANGENT_LEFT1`—The tangent is to the left of the line, and is on the opposite side of the line.
- `pfcDIMLINAOCTANGENT_RIGHT1`— The tangent is to the right of the line, and is on the opposite side of the line.

The method `pfcAngleDimensionSense::Create` creates a new dimension sense associated with the type `pfcDIMSENSE_ANGLE`. The dimension is the angle between two straight entities. The `sense` field is set to the value of the parameter *AngleOptions*.

The possible values of *AngleOptions* are:

- `IsFirst`—Is set to `TRUE` if the angle dimension starts from the specified entity in a counterclockwise direction. Is set to `FALSE` if the dimension ends at the specified entity. The value is `TRUE` for one entity and `FALSE` for the other entity forming the angle.
- `ShouldFlip`—If the value of `ShouldFlip` is `FALSE`, and the direction of the specified entity is away from the vertex of the angle, then the dimension attaches directly to the entity. If the direction of the entity is away from the vertex of the angle, then the dimension is attached to the a witness line. The witness line is in line with the entity but in the direction opposite to the vertex of the angle. If the value of `ShouldFlip` is `TRUE` then the above cases are reversed.

The method `pfcPointToAngleDimensionSense::Create` creates a new dimension sense associated with the type `pfcDIMSENSE_POINT_TO_ANGLE`. The dimension is the angle between a line entity and the tangent to a curved entity. The curve attachment is of the type `pfcDIMSENSE_POINT_TO_ANGLE` and the line attachment is of the type `DIMSENSE POINT`. In this case both the `angle` and the `angle_sense` fields must be set. The field `sense` shows which end of the curve the dimension is attached to and the field `angle_sense` shows the direction in which the dimension rotates and to which side of the tangent it attaches.

Drawing Dimensions Information

Methods Introduced:

- **pfcDrawing::IsDimensionAssociative**
- **pfcDimension2D::GetIsReference**
- **pfcDrawing::IsDimensionDisplayed**
- **pfcDrawing::GetDimensionAttachPoints**
- **pfcDrawing::GetDimensionSenses**
- **pfcDrawing::GetDimensionOrientHint**
- **pfcDrawing::GetBaselineDimension**
- **pfcDrawing::GetDimensionLocation**
- **pfcDrawing::GetDimensionView**
- **pfcDimension2D::GetTolerance**
- **wfcWDrawing::GetDimensionPath**
- **wfcWDrawing::GetDualDimensionOptions**
- **wfcDualDimensionGlobalOptions::GetDualDimensionType**
- **wfcDualDimensionGlobalOptions::GetSecondaryUnit**
- **wfcDualDimensionGlobalOptions::GetDigitsDifference**
- **wfcDualDimensionGlobalOptions::AreBracketsAllowed**

The method `pfcDrawing::IsDimensionAssociative` checks if the dimension or reference dimension is associative.

The method `pfcDimension2D::GetIsReference` determines whether the dimension is a reference dimension.

The method `pfcDrawing::IsDimensionDisplayed` checks if the dimension is displayed in the drawing or soild.

The method `pfcDrawing::GetDimensionAttachPoints` returns a sequence of attachment points for the dimension. The array of dimension senses returned by the method `pfcDimension2D::GetDimensionSenses` gives more information on how these attachments are interpreted. It gives information about how the dimension is attached to each attachment point of the model.

The method `pfcDrawing::GetDimensionSenses` returns a sequence of dimension senses, describing how the dimension is attached to each attachment returned by the method `pfcDrawing::GetDimensionAttachPoints`.

The method `pfcDrawing::GetDimensionOrientHint` returns the orientation of the dimensions in case where this cannot be deduced from the attachment themselves. The orientation of the dimensions is given by the enumerated type `pfcDimOrientationHint`. The orientation determines how Creo will orient the dimension with respect to the attachment points.

Note

This methods described above are applicable only for dimensions created in the drawing or the solid mode. It does not support dimensions created at intersection points of entities.

The method `pfcDrawing::GetBaselineDimension` returns an ordinate baseline dimension. It returns a null value if the dimension is not an ordinate dimension.

Note

The method updates the display of the dimension only if it is currently displayed.

The method `pfcDrawing::GetDimensionLocation` returns the placement location of the dimension.

The method `pfcDrawing::GetDimensionView` returns the drawing view in which the dimension is displayed. For dimensions created in drawing mode and owned by a solid, which can be displayed only in the context of that drawing, the method returns the drawing view. The method returns `NULL` if the dimension is not attached to a drawing view.

The method `pfcDimension2D::GetTolerance` retrieves the upper and lower tolerance limits of the drawing dimension in the form of the `pfcDimTolerance` object. A null value indicates a nominal tolerance.

The method `pfcDrawing::IsDimensionToleranceDisplayed` checks if the dimension tolerance is displayed in the drawing or solid.

The method `wfcWDrawing::GetDimensionPath` extracts the component path for a dimension displayed in a drawing.

The method `wfcWDrawing::GetDualDimensionOptions` retrieves information about the various options of dual dimensioning in a drawing as a `wfcDualDimensionGlobalOptions` object.

Use the method `wfcDualDimensionGlobalOptions::GetDualDimensionType` to get the display style of dual dimensions. The display type is given by the enumerated type `wfcDualDimensionDisplayType`. The valid values are:

- `wfcSECONDARY_DIM_DISPLAY_OFF`— Turns off display of dual dimension.
- `wfcSECONDARY_DIM_DISPLAY_TOP`—Displays the secondary dimension on top of the primary dimension. The primary dimension is displayed in brackets.
- `wfcSECONDARY_DIM_DISPLAY_BOTTOM`— Displays the secondary dimension below the primary dimension. The secondary dimension is displayed in brackets.
- `wfcSECONDARY_DIM_DISPLAY_LEFT`— Displays the secondary dimension to the left of the primary dimension. The primary dimension is displayed in brackets.
- `wfcSECONDARY_DIM_DISPLAY_RIGHT`— Displays the secondary dimension to the right of the primary dimension. The secondary dimension is displayed in brackets.
- `wfcSECONDARY_DIM_DISPLAY_ONLY`— Displays only the secondary dimension.

Use the method `wfcDualDimensionGlobalOptions::GetSecondaryUnit` to retrieve the type of units used for the secondary dimension.

The method `wfcDualDimensionGlobalOptions::GetDigitsDifference` retrieves the number of decimal places that a secondary dimension contains as compared to the primary dimension in a dual dimension.

The method `wfcDualDimensionGlobalOptions::AreBracketsAllowed` checks if one of the dimensions of a dual dimension is displayed in brackets.

Drawing Dimensions Operations

Methods Introduced:

- **pfcDrawing::ConvertOrdinateDimensionToLinear**
- **pfcDrawing::ConvertLinearDimensionToOrdinate**
- **pfcDimension2D::SetLocation**
- **pfcDrawing::SwitchDimensionView**
- **pfcDrawing::EraseDimension**
- **pfcModel2D::SetViewDisplaying**

The method `pfcDrawing::ConvertOrdinateDimensionToLinear` converts an ordinate dimension to a linear dimension. The drawing or solid containing the dimension must be displayed.

The method `pfcDrawing::ConvertLinearDimensionToOrdinate` converts a linear dimension to an ordinate baseline dimension.

The method `pfcDimension2D::SetLocation` sets the placement location of a dimension or reference dimension in a drawing or solid.

The method `pfcDrawing::SwitchDimensionView` changes the view where a dimension created in the drawing or solid is displayed.

The method `pfcDrawing::EraseDimension` permanently erases the dimension from the drawing or solid.

The method `pfcModel2D::SetViewDisplaying` changes the view where a dimension created in a solid model is displayed.

Ordinate Dimensions

Methods Introduced:

- **wfcWSolid::CreateOrdinateDimension**

The method `wfcWSolid::CreateOrdinateDimension` creates a new model ordinate driven dimension or a model ordinate reference dimension in a solid model. It requires the input of a reference baseline annotation as well as a

geometry reference. The annotation plane for the new dimension will be inherited from the baseline. Once the reference dimension is created, use the method `wfcAnnotation::ShowInDrawing` to display it.

Drawing Tables

A drawing table in Creo Object TOOLKIT C++ is represented by the interface `pfcTable`. It is a child of the `pfcModelItem` interface.

Some drawing table methods operate on specific rows or columns. The row and column numbers in Creo Object TOOLKIT C++ begin with 1 and range up to the total number of rows or columns in the table. Some drawing table methods operate on specific table cells. The interface `pfcTableCell` is used to represent a drawing table cell.

Creating Drawing Cells

Method Introduced:

- **pfcTableCell::Create**

The method `pfcTableCell::Create` creates the `pfcTableCell` object representing a cell in the drawing table.

Some drawing table methods operate on specific drawing segment. A multisegmented drawing table contains 2 or more areas in the drawing. Inserting or deleting rows in one segment of the table can affect the contents of other segments. Table segments are numbered beginning with 0. If the table has only a single segment, use 0 as the segment id in the relevant methods.

Selecting Drawing Tables and Cells

Methods Introduced:

- **pfcBaseSession::Select**
- **pfcSelection::GetSelItem**
- **pfcSelection::GetSelTableCell**
- **pfcSelection::GetSelTableSegment**

Tables may be selected using the method `pfcBaseSession::Select`. Pass the filter `dwg_table` to select an entire table and the filter `table_cell` to prompt the user to select a particular table cell.

The method `pfcSelection::GetSelItem` returns the selected table handle. It is a model item that can be cast to a `pfcTable` object.

The method `pfcSelection::GetSelTableCell` returns the row and column indices of the selected table cell.

The method `pfcSelection::GetSelTableSegment` returns the table segment identifier for the selected table cell. If the table consists of a single segment, this method returns the identifier 0.

Creating Drawing Tables

Methods Introduced:

- **pfcTableCreateInstructions::Create**
- **pfcTableOwner::CreateTable**

The method `pfcTableCreateInstructions::Create` creates the `pfcTableCreateInstructions` data object that describes how to construct a new table using the method `pfcTableOwner::CreateTable`.

The parameters of the instructions data object are:

- *Origin*—This parameter stores a three dimensional point specifying the location of the table origin. The origin is the position of the top left corner of the table.
- *RowHeights*—Specifies the height of each row of the table.
- *ColumnData*—Specifies the width of each column of the table and its justification.
- *SizeTypes*—Indicates the scale used to measure the column width and row height of the table.

The method `pfcTableOwner::CreateTable` creates a table in the drawing specified by the `pfcTableCreateInstructions` data object.

Retrieving Drawing Tables

Methods Introduced

- **pfcTableRetrieveInstructions::Create**
- **pfcTableRetrieveInstructions::SetFileName**
- **pfcTableRetrieveInstructions::SetPath**
- **pfcTableRetrieveInstructions::SetVersion**
- **pfcTableRetrieveInstructions::SetPosition**
- **pfcTableRetrieveInstructions::SetReferenceSolid**
- **pfcTableRetrieveInstructions::SetReferenceRep**
- **pfcTableOwner::RetrieveTable**
- **pfcTableOwner::RetrieveTableByOrigin**

The method `pfcTableRetrieveInstructions::Create` creates the `pfcTableRetrieveInstructions` data object that describes how to retrieve a drawing table using the methods

pfcTableOwner::RetrieveTable and pfcTableOwner::RetrieveTableByOrigin. The method returns the created instructions data object.

The parameters of the instruction object are:

- *FileName*—Name of the file containing the drawing table.
- *Position*—Coordinates of the point on the drawing sheet, where the retrieved table must be placed. You must specify the value in screen coordinates.

You can also set the parameters for pfcTableRetrieveInstructions data object using the following methods:

- pfcTableRetrieveInstructions::SetFileName—Sets the name of the drawing table. You must not specify the extension.
- pfcTableRetrieveInstructions::SetPath—Sets the path to the drawing table file. The path must be specified relative to the working directory.
- pfcTableRetrieveInstructions::SetVersion—Sets the version of the drawing table that must be retrieved. If you specify NULL rthe latest version of the drawing table is retrieved.
- pfcTableRetrieveInstructions::SetPosition—Sets the coordinates of the point on the drawing sheet, where the table must be placed. You must specify the value in screen coordinates.
- pfcTableRetrieveInstructions::SetReferenceSolid—Sets the model from which data must be copied into the drawing table. If this argument is passed as NULL, an empty table is created.
- pfcTableRetrieveInstructions::SetReferenceRep—Sets the handle to the simplified representation in a solid, from which data must be copied into the drawing table. If this argument is passed as NULL, and the argument solid is not NULL, then data from the solid model is copied into the drawing table

The method pfcTableOwner::RetrieveTable retrieves a table specified by the pfcTableRetrieveInstructions data object from a file on the disk. It returns the retrieved table. The upper-left corner of the table is placed on the drawing sheet at the position specified by the pfcTableRetrieveInstructions data object.

The method pfcTableOwner::RetrieveTableByOrigin also retrieves a table specified by the pfcTableRetrieveInstructions data object from a file on the disk. The origin of the table is placed on the drawing sheet at the position specified by the pfcTableRetrieveInstructions data object. Tables can be created with different origins by specifying the option **Direction**, in the **Insert Table** dialog box.

Drawing Tables Information

Methods Introduced:

- **pfcTableOwner::ListTables**
- **pfcTableOwner::GetTable**
- **pfcTable::GetRowCount**
- **pfcTable::GetColumnCount**
- **pfcTable::CheckIfIsFromFormat**
- **pfcTable::GetRowSize**
- **pfcTable::GetColumnSize**
- **pfcTable::GetText**
- **pfcTable::GetCellNote**

The method `pfcTableOwner::ListTables` returns a sequence of tables found in the model.

The method `pfcTableOwner::GetTable` returns a table specified by the table identifier in the model. It returns a null value if the table is not found.

The method `pfcTable::GetRowCount` returns the number of rows in the table.

The method `pfcTable::GetColumnCount` returns the number of columns in the table.

The method `pfcTable::CheckIfIsFromFormat` checks if the drawing table was created using the format. The method returns a true value if the table was created by applying the drawing format.

The method `pfcTable::GetRowSize` returns the height of the drawing table row specified by the segment identifier and the row number.

The method `pfcTable::GetColumnSize` returns the width of the drawing table column specified by the segment identifier and the column number.

The method `pfcTable::GetText` returns the sequence of text in a drawing table cell. Set the value of the parameter *Mode* to `pfcDWGTABLE_NORMAL` to get the text as displayed on the screen. Set it to `pfcDWGTABLE_FULL` to get symbolic text, which includes the names of parameter references in the table text.

The method `pfcTable::GetCellNote` returns the detail note item contained in the table cell.

Drawing Tables Operations

Methods Introduced:

- **pfcTable::Erase**

- **pfcTable::Display**
- **pfcTable::RotateClockwise**
- **pfcTable::InsertRow**
- **pfcTable::InsertColumn**
- **pfcTable::MergeRegion**
- **pfcTable::SubdivideRegion**
- **pfcTable::DeleteRow**
- **pfcTable::DeleteColumn**
- **pfcTable::SetText**
- **pfcTableOwner::DeleteTable**
- **wfcWTable::WrapCelltext**
- **wfcWTable::GetGrowthDirection**
- **wfcWTable::SetGrowthDirection**
- **wfcWTable::Save**
- **wfcWTable::SetColumnWidth**
- **wfcWTable::SetRowHeight**
- **wfcWTable::GetRowHeightAutoAdjustType**
- **wfcWTable::SetRowHeightAutoAdjustType**

The method `pfcTable::Erase` erases the specified table temporarily from the display. It still exists in the drawing. The erased table can be displayed again using the method `pfcTable::Display`. The table will also be redisplayed by a window repaint or a regeneration of the drawing. Use these methods to hide a table from the display while you are making multiple changes to the table.

The method `pfcTable::RotateClockwise` rotates a table clockwise by the specified amount of rotation.

The method `pfcTable::InsertRow` inserts a new row in the drawing table. Set the value of the parameter *RowHeight* to specify the height of the row. Set the value of the parameter *InsertAfterRow* to specify the row number after which the new row has to be inserted. Specify 0 to insert a new first row.

The method `pfcTable::InsertColumn` inserts a new column in the drawing table. Set the value of the parameter *ColumnWidth* to specify the width of the column. Set the value of the parameter *InsertAfterColumn* to specify the column number after which the new column has to be inserted. Specify 0 to insert a new first column.

The method `pfcTable::MergeRegion` merges table cells within a specified range of rows and columns to form a single cell. The range is a rectangular region specified by the table cell on the upper left of the region and the table cell on the lower right of the region.

The method `pfcTable::SubdivideRegion` removes merges from a region of table cells that were previously merged. The region to remove merges is specified by the table cell on the upper left of the region and the table cell on the lower right of the region.

The methods `pfcTable::DeleteRow` and `pfcTable::DeleteColumn` delete any specified row or column from the table. The methods also remove the text from the affected cells.

The method `pfcTable::SetText` sets text in the table cell.

Use the method `pfcTableOwner::DeleteTable` to delete a specified drawing table from the model permanently. The deleted table cannot be displayed again.

Note

Many of the above methods provide a parameter *Repaint* If this is set to true the table will be repainted after the change. If set to false or null Creo will delay the repaint, allowing you to perform several operations before showing changes on the screen.

The method `wfcWTable::WrapCelltext` wraps the text in a cell.

The method `wfcWTable::GetGrowthDirection` and `wfcWTable::SetGrowthDirection` get and set the growth direction of the table using the enumerated type `wfcTableGrowthDirType`.

The valid values for growth direction are defined in the enumerated data type `wfcTableGrowthDirType`:

- *wfcTABLEGROWTHDIR_DOWNRIGHT*
- *wfcTABLEGROWTHDIR_DOWNLEFT*
- *wfcTABLEGROWTHDIR_UPRIGHT*
- *wfcTABLEGROWTHDIR_UPRIGHT*

The method `wfcWTable::Save` saves a drawing table in different formats. The formats given by the enumerated type `wfcTableFormatType` can be of the following types:

- *TABLEFORMAT_TBL*—Specifies the tabular format.
- *TABLEFORMAT_TBL*—Specifies the tabular format.
- *TABLEFORMAT_CSV*—Specifies the CSV format.

The methods `wfcWTable::SetColumnWidth` and `wfcWTable::SetRowHeight` assign the width of a specified column and the height of a specified row depending upon the size of the drawing table. The drawing table size given by the enumerated data type `pfcTableSizeType` and the valid values are:

- *pfcTABLESIZE_BY_NUM_CHARS*—Specifies the size in characters. If the specified value for width of a column or height of a row is a fraction, *pfcTABLESIZE_BY_NUM_CHARS* rounds down the fractional value to the nearest whole number.
- *pfcTABLESIZE_BY_LENGTH*—Specifies the size in screen coordinates.

The methods `wfcWTable::GetRowHeightAutoAdjustType` and `wfcWTable::SetRowHeightAutoAdjustType` get and set the automatic row height adjustment property for a row of a drawing table. The type of height adjustment property is defined in the enumerated type `wfcTableRowheightAutoadjustType`:

- *wfcTBLROWHEIGHT_AUTOADJUST_FALSE*— Specifies that the automatic row height adjustment property is not set.
- *wfcTBLROWHEIGHT_AUTOADJUST_TRUE*— Specifies that the automatic row height adjustment property is set.
- *wfcTBLROWHEIGHT_AUTOADJUST_TRUE_LEGACY*— Specifies a pre-Creo Parametric 1.0 release behavior. In this behavior, sometimes the row height may be automatically adjusting and sometimes may not be automatically adjusting. To set an explicit row adjustment status use the method `wfcWTable::SetRowHeightAutoAdjustType`.

Drawing Table Segments

Drawing tables can be constructed with one or more segments. Each segment can be independently placed. The segments are specified by an integer identifier starting with 0.

Methods Introduced:

- **pfcSelection::GetSelTableSegment**
- **pfcTable::GetSegmentCount**
- **pfcTable::GetSegmentSheet**
- **pfcTable::MoveSegment**
- **pfcTable::GetInfo**
- **wfcWTable::SetSegmentOrigin**
- **wfcWTable::GetSegmentExtents**
- **wfcSegmentExtents::GetFirstColumn**
- **wfcSegmentExtents::GetFirstRow**

- **wfcSegmentExtents::GetLastColumn**
- **wfcSegmentExtents::GetLastRow**

The method `pfcSelection::GetSelTableSegment` returns the value of the segment identifier of the selected table segment. It returns a null value if the selection does not contain a segment identifier.

The method `pfcTable::GetSegmentCount` returns the number of segments in the table.

The method `pfcTable::GetSegmentSheet` determines the sheet number that contains a specified drawing table segment.

The method `pfcTable::MoveSegment` moves a drawing table segment to a new location. Pass the co-ordinates of the target position in the format x, y, z=0.

Note

Set the value of the parameter *Repaint* to true to repaint the drawing with the changes. Set it to false or null to delay the repaint.

To get information about a drawing table pass the value of the segment identifier as input to the method `pfcTable::GetInfo`. The method returns the table information including the rotation, row and column information, and the 3D outline.

Use the method `wfcWTable::SetSegmentOrigin` to assign the origin for a specified drawing table segment.

The method `wfcWTable::GetSegmentExtents` returns the start and end rows and columns of a specified table segment. The input argument *SegmentId* is the table segment ID. Pass the value as `-1` if you are referring to the only segment of a one-segment table.

The methods `wfcSegmentExtents::GetFirstColumn` and `wfcSegmentExtents::GetFirstRow` returns the first row and first column of a table segment.

The methods `wfcSegmentExtents::GetLastColumn` and `wfcSegmentExtents::GetLastRow` returns the last row and last column of a table segment.

Repeat Regions

Methods Introduced:

- **pfcTable::IsCommentCell**
- **pfcTable::GetCellComponentModel**

- **pfcTable::GetCellReferenceModel**
- **pfcTable::GetCellTopModel**
- **pfcTableOwner::UpdateTables**

The methods `pfcTable::IsCommentCell`, `pfcTable::GetCellComponentModel`, `pfcTable::GetCellReferenceModel`, `pfcTable::GetCellTopModel`, and `pfcTableOwner::UpdateTables` apply to repeat regions in drawing tables.

The method `pfcTable::IsCommentCell` tells you whether a cell in a repeat region contains a comment.

The method `pfcTable::GetCellComponentModel` returns the path to the assembly component model that is being referenced by a cell in a repeat region of a drawing table. It does not return a valid path if the cell attribute is set to `NO DUPLICATE` or `NO DUPLICATE/LEVEL`.

The method `pfcTable::GetCellReferenceModel` returns the reference component that is being referred to by a cell in a repeat region of a drawing table, even if cell attribute is set to `NO DUPLICATE` or `NO DUPLICATE/LEVEL`.

The method `pfcTable::GetCellTopModel` returns the top model that is being referred to by a cell in a repeat region of a drawing table, even if cell attribute is set to `NO DUPLICATE` or `NO DUPLICATE/LEVEL`.

Use the method `pfcTableOwner::UpdateTables` to update the repeat regions in all the tables to account for changes to the model. It is equivalent to the command **Table**, **Repeat Region**, **Update**.

Drawing Views And Models

Listing Drawing Views

Methods Introduced:

- **wfcWView2D::GetParentView**
- **wfcWView2D::GetChildren**
- **wfcWView2D::GetProjectionArrow**
- **wfcWView2D::GetPerspectiveScaleEyePointDistance**
- **wfcWView2D::GetPerspectiveScaleViewDiameter**
- **wfcWView2D::GetColorSource**
- **wfcWView2D::GetZClippingReference**
- **wfcWView2D::GetViewType**
- **wfcWView2D::GetViewId**

- **wfcWView2D::IsErased**
- **wfcWView2D::GetAlignmentInstructions**
- **wfcWView2D::SetAlignmentInstructions**
- **wfcViewAlignmentInstructions::Create**
- **wfcViewAlignmentInstructions::GetReferenceView**
- **wfcViewAlignmentInstructions::SetReferenceView**
- **wfcViewAlignmentInstructions::GetAlignmentStyle**
- **wfcViewAlignmentInstructions::SetAlignmentStyle**
- **wfcViewAlignmentInstructions::GetViewReference**
- **wfcViewAlignmentInstructions::SetViewReference**
- **wfcViewAlignmentInstructions::GetAlignedViewReference**
- **wfcViewAlignmentInstructions::SetAlignedViewReference**
- **wfcWView2D::GetOriginSelectionRef**
- **wfcWView2D::GetOrigin**
- **wfcWView2D::GetDatumDisplayStatus**
- **wfcWView2D::GetPipingDisplay**
- **wfcWView2D::GetErasedViewSheet**
- **wfcWDrawing::NeedsRegen**
- **wfcWDrawing::GetDrawingView**
- **wfcWSession::OpenDrawingAsReadOnly**

The method `wfcWView2D::GetParentView` retrieves the parent view of a specified drawing view.

The method `wfcWView2D::GetChildren` retrieves the child views of a drawing view.

The method `wfcWView2D::GetProjectionArrow` checks if the projection arrow flag has been set for a projected or detailed drawing view.

The method `wfcWView2D::GetPerspectiveScaleEyePointDistance` retrieves the eye-point distance from model space for the perspective scale applied to a drawing view. This scale option is available only for general views.

The method `wfcWView2D::GetPerspectiveScaleViewDiameter` specifies the view diameter in paper units such as mm for the perspective scale applied to a drawing view. This scale option is available only for general views.

The method `wfcWView2D::GetColorSource` retrieves information about color designation of the drawing view using the enumerated type `wfcDrawingViewColorSource`. The valid values are:

- wfcVIEW_MODEL_COLOR—Specifies that the drawing colors are determined by the model settings.
- wfcVIEW_DRAWING_COLOR—Specifies that the drawing colors are determined by the drawing settings.

The method wfcWView2D::GetZClippingReference retrieves the reference of the Z-clipping on the drawing view. The reference can be an edge, datum, or point on the surface that is parallel to the view. Geometry contained in the Z-clipping plane and in front of the plane appears, but geometry behind the plane does not appear. The system clips geometry that intersects the plane.

The method wfcWView2D::GetViewType retrieves the type of a specified drawing view using the enumerated data type wfcViewType. A drawing view can be of the following types:

- wfcVIEW_GENERAL—Specifies a general drawing view.
- wfcVIEW_PROJECTION—Specifies a projected drawing view.
- wfcVIEW_AUXILIARY—Specifies an auxiliary drawing view.
- wfcVIEW_DETAIL—Specifies a detailed drawing view.
- wfcVIEW_REVOLVE—Specifies a revolved drawing view.
- wfcVIEW_COPY_AND_ALIGN—Specifies a copy and align drawing view.
- wfcVIEW_OF_FLAT_TYPE—Specifies a flat type drawing view.

The method wfcWView2D::GetViewId retrieves the ID of the drawing view.

The method wfcWView2D::IsErased checks if the drawing view is erased or not. When you erase a view, it is removed from display in the drawing. The view is not deleted from the drawing.

The methods wfcWView2D::GetAlignmentInstructions and wfcWView2D::SetAlignmentInstructions retrieve and set the alignment of a drawing view with respect to a reference view as a wfcViewAlignmentInstructions object.

The method wfcViewAlignmentInstructions::Create creates a data object that contains information about view alignment.

The methods wfcViewAlignmentInstructions::GetReferenceView and wfcViewAlignmentInstructions::SetReferenceView retrieve and set the reference view to which the drawing view is aligned.

The methods wfcViewAlignmentInstructions::GetAlignmentStyle and wfcViewAlignmentInstructions::SetAlignmentStyle get and set the alignment style using the enumerated type wfcDrawingViewAlignStyle. The valid values are:

- wfcVIEW_ALIGN_HORIZONTAL—Specifies a horizontal alignment for the view. The drawing view and the reference view lie on the same horizontal line.

- wfcVIEW_ALIGN_VERTICAL—Specifies a vertical alignment for the view. In case of vertical alignment, the drawing view and the view are aligned to lie on the same vertical line.

The methods wfcViewAlignmentInstructions::GetViewReference and wfcViewAlignmentInstructions::SetViewReference get and set the geometry, such as an edge, on the reference view. The view is aligned along this geometry. If no geometry is specified, the reference view is aligned according to its view origin.

The methods wfcViewAlignmentInstructions::GetAlignedViewReference and wfcViewAlignmentInstructions::SetAlignedViewReference get and set the geometry, such as an edge, on the drawing view. If no geometry is specified, the drawing view is aligned according to its view origin.

The method wfcWView2D::GetOriginSelectionRef retrieves the selection reference as a pfcSelection object for the drawing view.

The method wfcWView2D::GetOrigin retrieves the location of the origin as a pfcPoint3D object for the drawing view.

The method wfcWView2D::GetDatumDisplayStatus checks if a solid model datum has been explicitly shown in a particular drawing view using the enumerated type wfcViewItemdisplayStatus. The valid values are:

- wfcVIEWDISP_NOT_SHOWN—Specifies that the solid model has never been shown in a particular drawing.
- wfcVIEWDISP_SHOWN—Specifies that the solid model has been shown in a particular drawing.
- wfcVIEWDISP_ERASED—Specifies that the solid model has been erased in a particular drawing.

The method wfcWView2D::GetPipingDisplay retrieves the piping display option for a drawing view using the enumerated type wfcPipingDisplay. The valid values are:

- wfcPIPINGDISP_DEFAULT—Displays the default appearance of pipes for the piping assembly.
- wfcPIPINGDISP_CENTERLINE—Displays pipes as centerlines without insulation.
- wfcPIPINGDISP_THICK_PIPES—Displays thick pipes without insulation.
- wfcPIPINGDISP_THICK_PIPES_AND_INSULATION—Displays thick pipes and insulation.

The method `wfcWView2D::GetErasedViewSheet` retrieves the sheet number which contained the view that was erased. If the sheet that contained the erased view is deleted, an exception is thrown.

The method `wfcWDrawing::NeedsRegen` checks whether the drawing or the specified drawing view needs to be regenerated.

The method `wfcWDrawing::GetDrawingView` retrieves the drawing view handle for the specified view ID.

The method `wfcWSession::OpenDrawingAsReadOnly` opens a drawing in the view only mode.

Modifying Views

Methods Introduced:

- **wfcWView2D::SetViewAsProjection**
- **wfcWView2D::SetProjectionArrow**
- **wfcWView2D::SetZClippingReference**
- **wfcWView2D::Erase**
- **wfcWView2D::Resume**
- **wfcWView2D::SetOrigin**
- **wfcWView2D::SetPipingDisplay**

The method `wfcWView2D::SetViewAsProjection` assigns the specified drawing view as a projection.

The method `wfcWView2D::SetProjectionArrow` sets the projection arrow flag to `true` for a projected or detailed drawing view.

The method `wfcWView2D::SetZClippingReference` sets the Z-clipping on the drawing view to reference a given edge, datum, or point on the surface that is parallel to the view. Geometry contained in the Z-clipping plane and in front of the plane appears, but geometry behind the plane does not appear. The system clips geometry that intersects the plane.

The method `wfcWView2D::Erase` removes the specified drawing view from display. To display the view back in the drawing, use the method `wfcWView2D::Resume`.

The method `wfcWView2D::SetOrigin` assigns the location of the origin as a `pfcPoint3D` object and the selection reference for a specified drawing view as a `pfcSelection` object. Both the input arguments *OriginLocation* and *Reference* are optional. However, to set the origin successfully, you must pass a valid value to either of the input arguments. If you pass both the input arguments as `Null`, the method throws an exception `pfcXToolkitBadInputs`.

The method `wfcWView2D::SetPipingDisplay` assigns the piping display option for a drawing view using the enumerated data type `wfcPipingDisplay`.

Detailed Views

Methods Introduced:

- **wfcWDrawing::CreateDetailView**
- **wfcWView2D::GetDetailViewInstructions**
- **wfcWView2D::SetDetailViewInstructions**
- **wfcDetailViewInstructions::Create**
- **wfcDetailViewInstructions::GetReference**
- **wfcDetailViewInstructions::SetReference**
- **wfcDetailViewInstructions::GetCurveData**
- **wfcDetailViewInstructions::SetCurveData**
- **wfcDetailViewInstructions::GetBoundaryType**
- **wfcDetailViewInstructions::SetBoundaryType**
- **wfcDetailViewInstructions::ShowBoundary**
- **wfcDetailViewInstructions::IsBoundaryShown**

A detailed view is a small portion of a model shown enlarged in another view.

The method `wfcWDrawing::CreateDetailView` creates a detailed view given the reference point on the parent view, the spline curve data, and location of the new view. A note with the detailed view name and the spline curve border are included in the parent view for the created detailed view. The input arguments are:

- *Instructions*— Specifies a `wfcDetailViewInstructions` object which contains all the information needed to create a detailed view.
- *Location*— Specifies the centerpoint of the view as a `pfcPoint3D` object.

Use the methods `wfcWView2D::GetDetailViewInstructions` and `wfcWView2D::SetDetailViewInstructions` to get and set the information related to detailed views as a `wfcDetailViewInstructions` object.

The method `wfcDetailViewInstructions::Create` creates a data object that contains information about the detailed view.

Use the methods `wfcDetailViewInstructions::GetReference` and `wfcDetailViewInstructions::SetReference` to get and set the reference point on the parent view for a specified detailed view.

Use the methods `wfcDetailViewInstructions::GetCurveData` and `wfcDetailViewInstructions::SetCurveData` to get and set the spline curve data as a `pfcCurveDescriptor` object for the specified detailed view. The information about the curve geometry is returned using the enumerated data type `pfcCurveType`. The curve data specifies the following:

- The X and Y coordinate directions match the screen space.
- The coordinate point (0,0) maps to the reference point.
- The scaling unit is of one inch relative to the top model of the view. If two points in the spline are at a distance of '1' from each other, then in the actual view, the points will be one inch distant from each other, if measured in the scale of the top model. For example, if one of the points in the spline definition has coordinates (0.5, 0.0, 0.0), then the position of that point is not half an inch to the right of the reference point on the paper. Instead, when projected as a point in the space of the top model of the view, it is half an inch to the right of the reference point when measured in the space of that model.

The methods `wfcDetailViewInstructions::GetBoundaryType` and `wfcDetailViewInstructions::SetBoundaryType` get and set the boundary type for a detailed view in terms of the enumerated type `wfcViewDetailBoundaryType`. The types of boundaries are:

- `wfcDETAIL_BOUNDARY_CIRCLE`—Draws a circle in the parent view.
- `wfcDETAIL_BOUNDARY_ELLIPSE`—Draws an ellipse in the parent view.
- `wfcDETAIL_BOUNDARY_HORZ_VER_ELLIPSE`—Draws an ellipse with horizontal or vertical major axis.
- `wfcDETAIL_BOUNDARY_SPLINE`—Displays the spline boundary drawn by the user in the parent view.
- `wfcDETAIL_BOUNDARY_ASME_CIRCLE`—Displays an ASME standard compliant circle as an arc with arrows and the detailed view name.

The method `wfcDetailViewInstructions::ShowBoundary` displays the boundary of the detailed view in the parent view.

The method `wfcDetailViewInstructions::IsBoundaryShown` checks if the boundary of the detailed view is displayed in the parent view.

Auxiliary Views

Methods Introduced:

- **wfcWDrawing::CreateAuxiliaryView**
- **wfcWView2D::GetAuxiliaryViewInstructions**
- **wfcAuxiliaryViewInstructions::Create**
- **wfcAuxiliaryViewInstructions::GetReference**
- **wfcAuxiliaryViewInstructions::SetReference**

- **wfcAuxiliaryViewInstructions::GetLocation**
- **wfcAuxiliaryViewInstructions::SetLocation**
- **wfcWView2D::SetViewAsAuxiliary**

An auxiliary view is a type of projected view that projects at right angles to a selected surface or axis. The selected surface or axis in the parent view must be perpendicular to the plane of the screen.

The method `wfcWDrawing::CreateAuxiliaryView` creates an auxiliary view. Pass the information required to create the auxiliary view as a `wfcAuxiliaryViewInstructions` object.

The method `wfcWView2D::GetAuxiliaryViewInstructions` retrieves information about the auxiliary view as a `wfcAuxiliaryViewInstructions` object.

The method `wfcAuxiliaryViewInstructions::Create` creates a data object that contains information about the auxiliary view.

The methods `wfcAuxiliaryViewInstructions::GetReference` and `wfcAuxiliaryViewInstructions::SetReference` retrieve and set the selection reference for the auxiliary view as a `pfcSelection` object.

The methods `wfcAuxiliaryViewInstructions::GetLocation` and `wfcAuxiliaryViewInstructions::SetLocation` retrieve and set the centerpoint of the auxiliary view as a `pfcPoint3D` object. By default, the origin of a drawing view is in the center of its outline. You can reset the origin of a drawing view by parametrically referencing model geometry or defining a location on the drawing sheet.

The method `wfcWView2D::SetViewAsAuxiliary` sets a specified drawing view as the auxiliary view.

Revolved Views

Methods Introduced:

- **wfcWDrawing::CreateRevolveView**
- **wfcWView2D::GetRevolveViewInstructions**
- **wfcRevolveViewInstructions::Create**
- **wfcRevolveViewInstructions::GetReference**
- **wfcRevolveViewInstructions::SetReference**
- **wfcRevolveViewInstructions::GetXSec**
- **wfcRevolveViewInstructions::SetXSec**
- **wfcRevolveViewInstructions::GetLocation**
- **wfcRevolveViewInstructions::SetLocation**

A revolved view is a cross section of an existing view, revolved 90 degrees around a cutting plane projection.

The method `wfcWDrawing::CreateRevolveView` creates a revolved view given a cross section, the selection reference, and the point location. Pass the information required to create the auxiliary view as a `wfcRevolveViewInstructions` object.

The method `wfcWView2D::GetRevolveViewInstructions` retrieves information about the revolved view as a `wfcRevolveViewInstructions` object.

The method `wfcRevolveViewInstructions::Create` creates a data object that contains information about the revolved view.

The methods `wfcRevolveViewInstructions::GetReference` and `wfcRevolveViewInstructions::SetReference` retrieve and set the selection reference for the revolved view as a `pfcSelection` object.

The methods `wfcRevolveViewInstructions::GetXSec` and `wfcRevolveViewInstructions::SetXSec` retrieve and set the cross section of the revolved view as a `pfcXSection` object.

The methods `wfcRevolveViewInstructions::GetLocation` and `wfcRevolveViewInstructions::SetLocation` retrieve and set the centerpoint of the revolved view as a `pfcPoint3D` object. By default, the origin of a drawing view is in the center of its outline. You can reset the origin of a drawing view by parametrically referencing model geometry or defining a location on the drawing sheet.

View Orientation

Methods Introduced:

- **wfcWView2D::SetOrientation**
- **wfcWView2D::SetOrientationFromReference**
- **wfcWView2D::SetOrientationFromAngle**

Note

The drawing view must be displayed before applying any orientation to it.

The method `wfcWView2D::SetOrientation` orients the specified drawing view using saved views from the model. The input arguments are:

- *MdlView*—Specifies the name of the saved view in the model.
- *Orientation*—Specifies the orientation of the view using the enumerated data type `wfcDrawingViewOrientationType`. The view types are:

- wfcVIEW_ORIENTATION_ISOMETRIC—Sets the view orientation to isometric.
- wfcVIEW_ORIENTATION_TRIMETRIC—Sets the view orientation to trimetric.
- wfcVIEW_ORIENTATION_USERDEFINED—Sets the view orientation to user-defined.

- *XAngle*—This input argument is required when the *Orientation* type is specified as wfcVIEW_ORIENTATION_USERDEFINED. For all other orientation types, this argument is ignored.
- *YAngle*—This input argument is required when the *Orientation* type is specified as wfcVIEW_ORIENTATION_USERDEFINED. For all other orientation types, this argument is ignored.

The method wfcWView2D::SetOrientationFromReference orients the view using geometric references.

- *MdlOrient1*—Specifies the orientation of the first geometric reference using the enumerated data type wfcDrawingViewOrientationRefType. The orientation types are:
  - wfcVIEW_ORIENTATION_REF_FRONT—Orients the view by using the front surface as a reference.
  - wfcVIEW_ORIENTATION_REF_BACK—Orients the view by using the back surface as a reference.
  - wfcVIEW_ORIENTATION_REF_LEFT—Orients the view by using the left surface as a reference.
  - wfcVIEW_ORIENTATION_REF_RIGHT—Orients the view by using the right surface as a reference.
  - wfcVIEW_ORIENTATION_REF_TOP—Orients the view by using the top surface as a reference.
  - wfcVIEW_ORIENTATION_REF_BOTTOM—Orients the view by using the bottom surface as a reference.
  - wfcVIEW_ORIENTATION_REF_VERTAXIS—Orients the view by using the vertical axis as a reference.
  - wfcVIEW_ORIENTATION_REF_HORIZAXIS—Orients the view by using the horizontal axis as a reference.
- *OrientRef1*—Specifies the first reference selection on the model.
- *MdlOrient2*—Specifies the orientation of the second geometric reference using the enumerated data type wfcDrawingViewOrientationRefType.
- *OrientRef2*—Specifies the second reference selection on the model.

The method `wfcWView2D::SetOrientationFromAngle` orients the specified drawing view using angles of selected references or custom angles. The input arguments are:

- *RotationAngleRef*—Specifies the angle reference using the enumerated data type `wfcDrawingViewOrientationAngleType`. The angle types are:
  - `wfcVIEW_ORIENTATION_ANGLE_NORMAL`—Orients the model around an axis through the view origin and normal to the drawing sheet.
  - `wfcVIEW_ORIENTATION_ANGLE_HORIZONTAL`—Orients the model around an axis through the view origin and horizontal to the drawing sheet.
  - `wfcVIEW_ORIENTATION_ANGLE_VERTICAL`—Orients the model around an axis through the view origin and vertical to the drawing sheet.
  - `wfcVIEW_ORIENTATION_ANGLE_EDGE_AXIS`—Orients the model around an axis through the view origin and according to the designated angle to the drawing sheet.
- *RefAngle*—Specifies the angle in degrees with the selected reference.
- *AxisSel*—Specifies the reference selection. It can be an axis or `NULL` for other type.

Sections of a View

Methods Introduced:

- **wfcWView2D::GetDrawingViewSectionType**
- **wfcWView2D::GetSection2DInstructions**
- **wfcWView2D::SetSection2DInstructions**
- **wfcSection2DInstructions::Create**
- **wfcSection2DInstructions::GetSectionAreaType**
- **wfcSection2DInstructions::SetSectionAreaType**
- **wfcSection2DInstructions::GetSectionName**
- **wfcSection2DInstructions::SetSectionName**
- **wfcSection2DInstructions::GetReference**
- **wfcSection2DInstructions::SetReference**
- **wfcSection2DInstructions::GetCurveData**
- **wfcSection2DInstructions::SetCurveData**
- **wfcSection2DInstructions::GetArrowDisplayView**
- **wfcSection2DInstructions::SetArrowDisplayView**
- **wfcWView2D::GetSinglePartSection**
- **wfcWView2D::SetSinglePartSection**
- **wfcWView2D::Get3DSectionName**

- **wfcWView2D::Is3DSectionXHatchingShown**
- **wfcWView2D::Set3DSection**

The method `wfcWView2D::GetDrawingViewSectionType` retrieves the section type for a specified drawing view using the enumerated data type `wfcDrawingViewSectionType`. A section can be of the following types:

- `wfcVIEW_NO_SECTION`—Specifies no section.
- `wfcVIEW_TOTAL_SECTION`—Specifies the complete drawing view.
- `wfcVIEW_AREA_SECTION`—Specifies a 2D cross section.
- `wfcVIEW_PART_SURF_SECTION`—Specifies a 3D cross section.
- `wfcVIEW_3D_SECTION`—Specifies a section created out of a solid surface or a datum quilt in the model.

The methods `wfcWView2D::GetSection2DInstructions` and `wfcWView2D::SetSection2DInstructions` retrieve and set the 2D cross section for a specified drawing view as a `wfcSection2DInstructions` object.

The method `wfcSection2DInstructions::Create` creates a data object that contains information about the 2D cross section.

The methods `wfcSection2DInstructions::GetSectionAreaType` and `wfcSection2DInstructions::SetSectionAreaType` retrieve and set the type of section area. The section area is given by the enumerated type `wfcDrawingViewSectionAreaType` and can be of the following types:

- `wfcVIEW_SECTION_AREA_FULL`—Sectioning is applied to the full drawing view.
- `wfcVIEW_SECTION_AREA_HALF`—Sectioning is applied to half drawing view depending upon the inputs for half side.
- `wfcVIEW_SECTION_AREA_LOCAL`—Specifies local sectioning.
- `wfcVIEW_SECTION_AREA_UNFOLD`—Unfold the drawing view and section it.
- `wfcVIEW_SECTION_AREA_ALIGNED`—Sectioning is as per the aligned views.

The methods `wfcSection2DInstructions::GetSectionName` and `wfcSection2DInstructions::SetSectionName` retrieve and set the name of the 2D cross section.

The methods `wfcSection2DInstructions::GetReference` and `wfcSection2DInstructions::SetReference` retrieve and set the selection reference as a `pfcSelection` object.

The methods `wfcSection2DInstructions::GetCurveData` and `wfcSection2DInstructions::SetCurveData` retrieve and set the spline curve data as a `pfcCurveDescriptor` object. The information about curve geometry is given by the enumerated data type `pfcCurveType`.

The methods `wfcSection2DInstructions::GetArrowDisplayView` and `wfcSection2DInstructions::SetArrowDisplayView` retrieve and set the drawing view, that is, either the parent or child view, where the section arrow is to be displayed.

The method `wfcWView2D::GetSinglePartSection` retrieves the section created out of a solid surface or a datum quilt in the model for a specified drawing view.

The method `wfcWView2D::SetSinglePartSection` assigns the reference selection as a `pfcSelection` object for the solid surface or datum quilt that is used to create the section in the view.

The method `wfcWView2D::Get3DSectionName` retrieves the name of the 3D cross section for the drawing view.

The method `wfcWView2D::Is3DSectionXHatchingShown` checks if Xhatching is displayed in the 3D cross-sectional view.

The method `wfcWView2D::Set3DSection` assigns the following arguments to define a 3D cross section for a view:

- `SectionName`—Specifies the name of the 3D cross section.
- `ShowXHatch`—Specifies a boolean value that determines whether the cross section hatching must be displayed in the 3D cross-sectional view. Set this argument to `true` to display the X-hatching

Visible Areas of Views

Methods Introduced:

- **wfcWView2D::SetVisibleArea**
- **wfcWView2D::GetVisibleAreaInstructions**
- **wfcVisibleAreaInstruction::GetType**
- **wfcBreakAreaInstruction::Create**
- **wfcBreakAreaInstruction::GetCurveData**
- **wfcBreakAreaInstruction::SetCurveData**
- **wfcBreakAreaInstruction::GetFirstBreakLine**
- **wfcBreakAreaInstruction::SetFirstBreakLine**
- **wfcBreakAreaInstruction::GetSecondBreakLine**
- **wfcBreakAreaInstruction::SetSecondBreakLine**
- **wfcBreakAreaInstruction::GetViewBrokenDir**

- **wfcBreakAreaInstruction::SetViewBrokenDir**
- **wfcBreakAreaInstruction::GetViewBrokenLineStyle**
- **wfcBreakAreaInstruction::SetViewBrokenLineStyle**
- **wfcBrokenAreaVisibility::Create**
- **wfcBrokenAreaVisibility::GetInstructions**
- **wfcBrokenAreaVisibility::SetInstructions**
- **wfcFullAreaVisibility::Create**
- **wfcHalfAreaVisibility::Create**
- **wfcHalfAreaVisibility::DisplayLeftSide**
- **wfcHalfAreaVisibility::IsLeftSideDisplayed**
- **wfcHalfAreaVisibility::GetLineType**
- **wfcHalfAreaVisibility::SetLineType**
- **wfcHalfAreaVisibility::GetReference**
- **wfcHalfAreaVisibility::SetReference**
- **wfcPartialAreaVisibility::Create**
- **wfcPartialAreaVisibility::GetCurveData**
- **wfcPartialAreaVisibility::SetCurveData**
- **wfcPartialAreaVisibility::GetReferencePoint**
- **wfcPartialAreaVisibility::SetReferencePoint**
- **wfcPartialAreaVisibility::IsBoundaryShown**
- **wfcPartialAreaVisibility::ShowBoundary**

As you detail your model, certain portions of the model may be more relevant than others or may be clearer if displayed from a different view point. You can define the visible area of the view to determine which portion or portions to show or hide.

The method `wfcWView2D::SetVisibleArea` assigns the type of visible area for a specified drawing view using the enumerated data type `wfcVisibleAreaInstruction` object.

The method `wfcWView2D::GetVisibleAreaInstructions` retrieves the type of visible area in terms of `wfcVisibleAreaInstruction` object for a specified drawing view.

The method `wfcVisibleAreaInstruction::GetType` retrieves the type of visible area for a specified drawing view using the enumerated type `wfcDrawingViewVisibleareaType`. The visible area can be of the following types:

- `wfcVIEW_FULL_AREA`—The complete drawing view is retained as the visible area.

- `wfcVIEW_HALF_AREA`—A portion of the model from the view on one side of a cutting plane is removed.
- `wfcVIEW_PARTIAL_AREA`—A portion of the model in a view within a closed boundary is displayed.
- `wfcVIEW_BROKEN_AREA`—A portion of the model view from between two or more selected points is removed, and the gap between the remaining two portions is closed within a specified distance.

Note

A broken visible area can be created only for general and projected view types.

The method `wfcBreakAreaInstruction::Create` enables you to create the instructions for a break area in a drawing view. The input arguments are:

- *ViewBrokenDir*—Specify the direction of the broken lines that define the broken area to be removed in terms of the enumerated type `wfcViewBrokenDir`.
- *FirstBreakLine*—Specify the selection point in terms of the `pfcSelection` object for the first break line.
- *SecondBreakLine*—Specify the selection point in terms of the `pfcSelection` object for the second break line.
- *ViewBrokenLineStyle*—Specifies the line style for the broken lines in terms of the enumerated type `wfcViewBrokenLineStyle`.

The methods `wfcBreakAreaInstruction::GetCurveData` and `wfcBreakAreaInstruction::SetCurveData` retrieve and set the spline curve data in terms of the `pfcCurveDescriptor` object. These methods are applicable only for `wfcVIEW_BROKEN_LINE_S_CURVE_GEOMETRY` line style.

The methods `wfcBreakAreaInstruction::GetFirstBreakLine` and `wfcBreakAreaInstruction::SetFirstBreakLine` retrieve and set the selection point in terms of the `pfcSelection` object for the first break line.

The methods `wfcBreakAreaInstruction::GetSecondBreakLine` and `wfcBreakAreaInstruction::SetSecondBreakLine` retrieve and set the selection point in terms of the `pfcSelection` object for the second break line.

The methods `wfcBreakAreaInstruction::GetViewBrokenDir` and `wfcBreakAreaInstruction::SetViewBrokenDir` retrieve and set the direction of the broken lines that define the broken area to be removed. The direction is given by the enumerated type `wfcViewBrokenDir` and it takes the following values:

- `wfcVIEW_BROKEN_DIR_HORIZONTAL`—Specifies the horizontal direction.
- `wfcVIEW_BROKEN_DIR_VERTICAL`—Specifies the vertical direction.

The methods `wfcBreakAreaInstruction::GetViewBrokenLineStyle` and `wfcBreakAreaInstruction::SetViewBrokenLineStyle` retrieve and set the line style for the broken lines in terms of the enumerated type `wfcViewBrokenLineStyle`. It can be one of the following types:

- `wfcVIEW_BROKEN_LINE_STRAIGHT`—Specifies a straight broken line.
- `wfcVIEW_BROKEN_LINE_SKETCH`—Specifies a random sketch drawn by the user that defines the broken line.
- `wfcVIEW_BROKEN_LINE_S_CURVE_OUTLINE`—Specifies a S-curve on the view outline.
- `wfcVIEW_BROKEN_LINE_S_CURVE_GEOMETRY`—Specifies a S-curve on the geometry.
- `wfcVIEW_BROKEN_LINE_HEART_BEAT_OUTLINE`—Specifies a heartbeat type of curve on the view outline.
- `wfcVIEW_BROKEN_LINE_HEART_BEAT_GEOMETRY`—Specifies a heartbeat type of curve on the geometry.

The method `wfcBrokenAreaVisibility::Create` enables you to create a broken visible area in a drawing view. The input argument is *instructions*, which is a collection of objects to create the break area.

The methods `wfcBrokenAreaVisibility::GetInstructions` and `wfcBrokenAreaVisibility::SetInstructions` retrieve and set the instructions for a visible broken area in a drawing view.

The method `wfcFullAreaVisibility::Create` enables you to create a full visible area in a drawing view.

The method `wfcHalfAreaVisibility::Create` enables you to create a half visible area in a drawing view.

The method `wfcHalfAreaVisibility::DisplayLeftSide` displays the left side of the half visible area.

The method `wfcHalfAreaVisibility::IsLeftSideDisplayed` identifies if the left side of a half visible area is displayed.

The methods `wfcHalfAreaVisibility::GetLineType` and `wfcHalfAreaVisibility::SetLineType` retrieve and set the line type in a half visible area in terms of the enumerated type `wfcDrawingLineStandardType` and it takes the following values:

- `wfcHVL_NONE`—Specifies no line.
- `wfcHVL_SOLID`—Specifies a solid line.
- `wfcHVL_SYMMETRY`—Specifies a symmetrical line.
- `wfcHVL_SYMMETRY_ISO`—Specifies an ISO-standard symmetrical line.
- `wfcHVL_SYMMETRY_ASME`—Specifies an ASME-standard symmetrical line.

The methods `wfcHalfAreaVisibility::GetReference` and `wfcHalfAreaVisibility::SetReference` retrieve and set the selection reference in terms of the `pfcSelection`object that divides the drawing view. The cutting plane can be a planar surface or a datum, but it must be perpendicular to the screen in the new view.

The method `wfcPartialAreaVisibility::Create` enables you to create a partial visible area in a drawing view.

The methods `wfcPartialAreaVisibility::GetCurveData` and `wfcPartialAreaVisibility::SetCurveData` retrieve and set the spline curve data in a partial visible area in terms of the `pfcCurveDescriptor` object.

The methods `wfcPartialAreaVisibility::GetReferencePoint` and `wfcPartialAreaVisibility::SetReferencePoint` retrieve and set the reference point in terms of the `pfcSelection` object.

The method `wfcPartialAreaVisibility::IsBoundaryShown` identifies if the boundary of the partial visible area contained within the spline is shown.

The method `wfcPartialAreaVisibility::ShowBoundary` displays or shows the boundary of the partial visible area contained within the spline.

Refer to the Detailed Drawings module from the Creo Parametric Help for more information on Visible Area of the View.

View States

Methods Introduced:

- **wfcWView2D::SetExplodedState**
- **wfcWView2D::IsExplodedState**
- **wfcWView2D::SetSimpRep**

In the method `wfcWView2D::SetExplodedState` set the input argument *ExplodedState* as `true`, to display the specified drawing view in the exploded state.

The method `wfcWView2D::IsExplodedState` checks if the specified view is set to be displayed in the exploded state.

The method `wfcWView2D::SetSimpRep` retrieves the simplified representation for a specified drawing view.

Drawing Models

Methods Introduced:

- **wfcWDrawing::VisitDrawingModels**

The method `wfcWDrawing::VisitDrawingModels` visits the solids in the specified drawing.

Drawing Edges

The methods described in this section provide access to the display properties such as color, line font, and thickness of model edges in drawing views. Model edges can be regular edges, silhouette edges, or non-analytical silhouette edges.

Note

You can select model edges from detailed views for modification, but no change will be applied. To modify the display of a model edge in a detailed view, you must select the edge in the parent view.

Methods Introduced:

- **wfcWDrawing::GetEdgeDisplay**
- **wfcWDrawing::SetEdgeDisplay**
- **wfcEdgeDisplay::Create**
- **wfcEdgeDisplay::GetColor**
- **wfcEdgeDisplay::SetColor**
- **wfcEdgeDisplay::GetFont**
- **wfcEdgeDisplay::SetFont**
- **wfcEdgeDisplay::GetWidth**
- **wfcEdgeDisplay::SetWidth**
- **wfcWDrawing::IsEdgeDisplayGlobal**

- **wfcWDrawing::SetEdgeDisplayGlobal**

The methods `wfcWDrawing::GetEdgeDisplay` and `wfcWDrawing::SetEdgeDisplay` retrieve and set the display properties of a specified model edge in a drawing view as a `wfcEdgeDisplay` object. After assigning the properties, you must repaint the drawing view to update the display.

The method `wfcEdgeDisplay::Create` creates a data object that contains information about the display properties of an edge in a drawing view.

The methods `wfcEdgeDisplay::GetColor` and `wfcEdgeDisplay::SetColor` retrieve and set the color to be used for the display of a specified model edge.

The methods `wfcEdgeDisplay::GetFont` and `wfcEdgeDisplay::SetFont` retrieve and set the line font to be used for the display of a specified model edge.

The methods `wfcEdgeDisplay::GetWidth` and `wfcEdgeDisplay::SetWidth` retrieve and set the width to be used for the display of a specified model edge. You must pass a value less than zero to use the default width.

The method `wfcWDrawing::IsEdgeDisplayGlobal` checks if the model edge display properties such as color, line font, and width have been applied globally to all the drawing views in the drawing sheet.

The method `wfcWDrawing::SetEdgeDisplayGlobal` sets the flag that assigns the model edge display properties such as color, line font, and width globally to all the drawing views in the drawing sheet.

Detail Items

The methods described in this section operate on detail items.

In Creo Object TOOLKIT C++ you can create, delete and modify detail items, control their display, and query what detail items are present in the drawing. The types of detail items available are:

- Draft Entities—Contain graphical items created in Creo. The items are as follows:
  - Arc
  - Ellipse
  - Line
  - Point
  - Polygon
  - Spline
- Notes—Textual annotations

- Symbol Definitions—Contained in the drawing's symbol gallery.
- Symbol Instances—Instances of a symbol placed in a drawing.
- Draft Groups—Groups of detail items that contain notes, symbol instances, and draft entities.
- OLE objects—Object Linking and Embedding (OLE) objects embedded in the Creo drawing file.

Listing Detail Items

Methods Introduced:

- **pfcModelItemOwner::ListItems**
- **pfcDetailItemOwner::ListDetailItems**
- **pfcModelItemOwner::GetItemById**
- **pfcDetailItemOwner::CreateDetailItem**

The method `pfcModelItemOwner::ListItems` returns a list of detail items specified by the parameter *Type* or returns null if no detail items of the specified type are found.

The values of the parameter *Type* for detail items are:

- `pfcITEM_DTL_ENTITY`—Detail Entity
- `pfcITEM_DTL_NOTE`—Detail Note
- `pfcITEM_DTL_GROUP`—Draft Group
- `pfcITEM_DTL_SYM_DEFINITION`—Detail Symbol Definition
- `pfcITEM_DTL_SYM_INSTANCE`—Detail Symbol Instance
- `pfcITEM_DTL_OLE_OBJECT`—Drawing embedded OLE object

If this parameter is set to null, then all the model items in the drawing are listed.

If the model has multiple bodies, the method `pfcModelItemOwner::ListItems` returns the exception `pfcXToolkitMultibodyUnsupported`.

The method `pfcDetailItemOwner::ListDetailItems` also lists the detail items in the model. Pass the type of the detail item and the sheet number that contains the specified detail items.

Set the input parameter *Type* to the type of detail item to be listed. Set it to null to return all the detail items. The input parameter *SheetNumber*determines the sheet that contains the specified detail item. Pass null to search all the sheets. This argument is ignored if the parameter *Type* is set to `pfcDETAIL_SYM_DEFINITION`.

The method returns a sequence of detail items and returns a null if no items matching the input values are found.

The method `pfcModelItemOwner::GetItemById` returns a detail item based on the type of the detail item and its integer identifier. The method returns a null if a detail item with the specified attributes is not found.

Creating, Modifying and Reading Detail Items

Methods Introduced:

- **pfcDetailItemOwner::CreateDetailItem**
- **pfcDetailGroupInstructions::Create**
- **wfcWDetailSymbolDefItem::CopyToAnotherModel**

The method `pfcDetailItemOwner::CreateDetailItem` creates a new detail item based on the instruction data object that describes the type and content of the new detail item. The instructions data object is returned by the method `pfcDetailGroupInstructions::Create`. The method returns the newly created detail item.

The method `wfcWDetailSymbolDefItem::CopyToAnotherModel` copies a specified symbol definition from one model to another. Pass the model to which the symbol definition must be copied as the input argument.

Detail Note Data

Methods Introduced:

- **wfcWDetailNoteItem::CollectSymbolInstances**
- **pfcDetailNoteItem::GetNoteTextStyle**
- **pfcDetailNoteItem::SetNoteTextStyle**

The method `wfcWDetailNoteItem::CollectSymbolInstances` retrieves a list of all the symbol instances that are declared in a detail note. The symbol instances are returned in the order in which they are seen in the note text.

The method `pfcDetailNoteItem::GetNoteTextStyle` and `pfcDetailNoteItem::SetNoteTextStyle` retrieve and set the text style for the specified note as a `pfcAnnotationTextStyle` object. Refer to the section AnnotationTextStyles on page 252, for more information on TextStyles.

Cross-referencing 3D Notes and Drawing Annotations

The methods described in this section provide access to the drawing object that represents a shown 3D note, (if the 3D note is shown in the drawing), or vice-versa.

Methods Introduced:

- **wfcWDetailNoteItem::GetAssociativeNoteInDrawing**
- **wfcWDetailNoteItem::GetAssociativeNoteInSolid**
- **wfcGTol::GetDetailNote**

The method `wfcWDetailNoteItem::GetAssociativeNoteInDrawing` retrieves a detail note in the drawing that represents the specified model note in a solid. Specify the drawing in the input argument *DrawingModel*.

Note

This method retrieves the first detail note that calls out the solid model note. Creo Parametric does not restrict users to showing only a single version of a solid model note callout.

The method `wfcWDetailNoteItem::GetAssociativeNoteInSolid` retrieves the model note in a solid that is displayed as a detail note in the drawing.

The method `wfcGTol::GetDetailNote` returns the detail note that represents a shown geometric tolerance.

Note

This method returns the first detail note that calls out the solid model note. Creo Parametric does not restrict users to showing only a single version of a solid model note callout.

Symbol Definition Attachments

Methods Introduced:

- **wfcWDetailSymbolDefItem::AddAttachment**

The method `wfcWDetailSymbolDefItem::AddAttachment` adds parametric leader attachments to the symbol definition.

Symbol Instance Data

Methods Introduced:

- **wfcWDetailSymbolInstItem::GetAttachedDimension**
- **wfcWDetailSymbolInstItem::AddLeader**
- **wfcWDetailSymbolInstItem::AddVarText**

- **wfcWDetailSymbolInstItem::GetNoteInstructions**
- **wfcWDetailSymbolInstItem::GetEntityInstructions**

The method `wfcWDetailSymbolInstItem::GetAttachedDimension` retrieves the dimension to which the specified symbol instance is attached as a `pfcDimension` object. The method throws an exception `pfcXToolkitBadContext` when the dimension to which the specified symbol instance is attached is not available. In this case, the model containing the dimension was either deleted or suppressed in the assembly.

The method `wfcWDetailSymbolInstItem::AddLeader` adds a leader to a symbol instance description.

The method `wfcWDetailSymbolInstItem::AddVarText` adds variable text to the symbol instance.

The method `wfcWDetailSymbolInstItem::GetNoteInstructions` retrieves the data of a note in the symbol instance as a `pfcDetailNoteInstructions` object. The input arguments are:

- *Item*—Specifies the note attached to the symbol instance as a `pfcDetailNoteItem` object.
- *DispMode*—Specifies the display mode for parameters and dimensions in the note using the enumerated data type `pfcDimDisplayMode`. The valid values are:
  - `pfcDIM_DISPLAY_NUMERIC`—Displays the parameters and dimensions as numeric values.
  - `pfcDIM_DISPLAY_SYMBOLIC`—Displays the parameters and dimensions as symbolic values.

Refer to the section Instructions on page 182 for more information on `pfcDetailNoteInstructions` methods.

The method `wfcWDetailSymbolInstItem::GetEntityInstructions` retrieves the data of an entity in the symbol instance as a `pfcDetailEntityInstructions` object.

Refer to the sectionInstructions on page 178 for more information on `pfcDetailEntityInstructions` methods.

Cross-referencing Weld Symbols and Drawing Annotations

The methods described in this section provide a drawing object that represents a shown weld symbol, if the weld symbol is shown in the drawing, or the weld feature that owns a shown weld symbol.

Methods Introduced:

- **wfcWDetailSymbolInstItem::GetFeature**

The method `wfcWDetailSymbolInstItem::GetFeature` retrieves the weld feature that owns the shown weld symbol.

Detail Group Data

Methods Introduced:

- **wfcWDetailGroupItem::AddElement**

The method `wfcWDetailGroupItem::AddElement` adds an item to the group contents. Items supported in the groups include entities, notes, symbol instances, and draft drawing dimensions.

Drawing Symbol Groups

This section describes Creo Object TOOLKIT C++ methods that give access to user-defined groups contained in drawing symbols. Creo Parametric allows a hierarchal relationship between the groups in a symbol definition. Thus, some groups contain groups, or are parents of other groups.

Manipulating Symbol Groups

Methods Introduced:

- **wfcWDetailSymbolGroup::AddItem**

The method `wfcWDetailSymbolGroup::AddItem` adds a single item to the symbol group, provided such an item belongs to the symbol definition. The item to be added can be a detail entity or a note.

Detail Entities

A detail entity in Creo Object TOOLKIT C++ is represented by the interface `pfcDetailEntityItem`. It is a child of the `pfcDetail` interface.

The interface `pfcDetailEntityInstructions` contains specific information used to describe a detail entity item.

Instructions

Methods Introduced:

- **pfcDetailEntityInstructions::Create**
- **pfcDetailEntityInstructions::GetGeometry**
- **pfcDetailEntityInstructions::SetGeometry**

- **pfcDetailEntityInstructions::GetIsConstruction**
- **pfcDetailEntityInstructions::SetIsConstruction**
- **pfcDetailEntityInstructions::GetColor**
- **pfcDetailEntityInstructions::SetColor**
- **pfcDetailEntityInstructions::GetFontName**
- **pfcDetailEntityInstructions::SetFontName**
- **pfcDetailEntityInstructions::GetWidth**
- **pfcDetailEntityInstructions::SetWidth**
- **pfcDetailEntityInstructions::GetView**
- **pfcDetailEntityInstructions::SetView**

The method `pfcDetailEntityInstructions::Create` creates an instructions object that describes how to construct a detail entity, for use in the methods `pfcDetailItemOwner::CreateDetailItem`, `pfcDetailSymbolDefItem::CreateDetailItem`, and `pfcDetailEntityItem::Modify`.

The instructions object is created based on the curve geometry and the drawing view associated with the entity. The curve geometry describes the trajectory of the detail entity in world units. The drawing view can be a model view returned by the method `pfcModel2D::List2DViews` or a drawing sheet background view returned by the method `pfcSheetOwner::GetSheetBackgroundView`. The background view indicates that the entity is not associated with a particular model view.

The method returns the created instructions object.

Note

Changes to the values of a `pfcDetailEntityInstructions` object do not take effect until that instructions object is used to modify the entity using `pfcDetailEntityItem::Modify`.

The method `pfcDetailEntityInstructions::GetGeometry` returns the geometry of the detail entity item.

The method `pfcDetailEntityInstructions::SetGeometry` sets the geometry of the detail entity item. For more information refer to Curve Descriptors on page 416.

The method `pfcDetailEntityInstructions::GetIsConstruction` returns a value that specifies whether the entity is a construction entity.

The method `pfcDetailEntityInstructions::SetIsConstruction` specifies if the detail entity is a construction entity.

The method `pfcDetailEntityInstructions::GetColor` returns the color of the detail entity item.

The method `pfcDetailEntityInstructions::SetColor` sets the color of the detail entity item. Pass null to use the default drawing color.

The method `pfcDetailEntityInstructions::GetFontName` returns the line style used to draw the entity. The method returns a null value if the default line style is used.

The method `pfcDetailEntityInstructions::SetFontName` sets the line style for the detail entity item. Pass null to use the default line style.

The method `pfcDetailEntityInstructions::GetWidth` returns the value of the width of the entity line. The method returns a null value if the default line width is used.

The method `pfcDetailEntityInstructions::SetWidth` specifies the width of the entity line. Pass null to use the default line width.

The method `pfcDetailEntityInstructions::GetView` returns the drawing view associated with the entity. The view can either be a model view or a drawing sheet background view.

The method `pfcDetailEntityInstructions::SetView` sets the drawing view associated with the entity. The view can either be a model view or a drawing sheet background view.

Detail Entities Information

Methods Introduced:

- **pfcDetailEntityItem::GetInstructions**
- **pfcDetailEntityItem::GetSymbolDef**

The method `pfcDetailEntityItem::GetInstructions` returns the instructions data object that is used to construct the detail entity item.

The method `pfcDetailEntityItem::GetSymbolDef` returns the symbol definition that contains the entity. This method returns a null value if the entity is not a part of a symbol definition.

Detail Entities Operations

Methods Introduced:

- **pfcDetailEntityItem::Draw**
- **pfcDetailEntityItem::Erase**
- **pfcDetailEntityItem::Modify**

The method `pfcDetailEntityItem::Draw` temporarily draws a detail entity item, so that it is removed during the next draft regeneration.

The method `pfcDetailEntityItem::Erase` undraws a detail entity item temporarily, so that it is redrawn during the next draft regeneration.

The method `pfcDetailEntityItem::Modify` modifies the definition of an entity item using the specified instructions data object.

OLE Objects

An object linking and embedding (OLE) object is an external file, such as a document, graphics file, or video file that is created using an external application and which can be inserted into another application, such as Creo. You can create and insert supported OLE objects into a two-dimensional Creo file, such as a drawing, report, format file, notebook, or diagram. The functions described in this section enable you to identify and access OLE objects embedded in drawings.

Methods Introduced:

- **pfcDetailOLEObject::GetApplicationType**
- **pfcDetailOLEObject::GetOutline**
- **pfcDetailOLEObject::GetPath**
- **pfcDetailOLEObject::GetSheet**

The method `pfcDetailOLEObject::GetApplicationType` returns the type of the OLE object as a string, for example, `Microsoft Word Document`.

The method `pfcDetailOLEObject::GetOutline` returns the extent of the OLE object embedded in the drawing.

The method `pfcDetailOLEObject::GetPath` returns the path to the external file for each OLE object, if it is linked to an external file.

The method `pfcDetailOLEObject::GetSheet` returns the sheet number for the OLE object.

Detail Notes

A detail note in Creo Object TOOLKIT C++ is represented by the interface `pfcDetailNoteItem`. It is a child of the `pfcDetail` interface.

The interface `pfcDetailNoteInstructions` contains specific information that describes a detail note.

The interface `pfcDetailNoteInstructions` and all the methods under this interface are deprecated.

Instructions

Methods Introduced:

- **pfcDetailNoteInstructions::Create**
- **pfcDetailNoteInstructions::GetTextLines**
- **pfcDetailNoteInstructions::SetTextLines**
- **pfcDetailNoteInstructions::GetIsDisplayed**
- **pfcDetailNoteInstructions::SetIsDisplayed**
- **pfcDetailNoteInstructions::GetIsReadOnly**
- **pfcDetailNoteInstructions::SetIsReadOnly**
- **pfcDetailNoteInstructions::GetIsMirrored**
- **pfcDetailNoteInstructions::SetIsMirrored**
- **pfcDetailNoteInstructions::GetHorizontal**
- **pfcDetailNoteInstructions::SetHorizontal**
- **pfcDetailNoteInstructions::GetVertical**
- **pfcDetailNoteInstructions::SetVertical**
- **pfcDetailNoteInstructions::GetColor**
- **pfcDetailNoteInstructions::SetColor**
- **pfcDetailNoteInstructions::GetLeader**
- **pfcDetailNoteInstructions::SetLeader**
- **pfcDetailNoteInstructions::GetTextAngle**
- **pfcDetailNoteInstructions::SetTextAngle**

The method `pfcDetailNoteInstructions::Create` creates a data object that describes how a detail note item should be constructed when passed to the methods `pfcDetailItemOwner::CreateDetailItem`, `pfcDetailSymbolDefItem::CreateDetailItem`, or `pfcDetailNoteItem::Modify`. The parameter *inTextLines* specifies the sequence of text line data objects that describe the contents of the note.

Note

Changes to the values of a `pfcDetailNoteInstructions` object do not take effect until that instructions object is used to modify the note using `pfcDetailNoteItem::Modify`.

For creating a detail note item, the method `pfcDetailItemOwner::CreateDetailItem` is deprecated, as it uses the deprecated interface `pfcDetailNoteInstructions`. Use the methods `pfcDetailItemOwner::CreateFreeNote`, `pfcDetailItemOwner::CreateOffsetNote`, `pfcDetailItemOwner::CreateOnItemNote`, or `pfcDetailItemOwner::CreateLeaderNote` instead.

The method `pfcDetailNoteInstructions::GetTextLines` returns the description of text line contents in the note.

The method `pfcDetailNoteInstructions::SetTextLines` sets the description of the text line contents in the note.

The method `pfcDetailNoteInstructions::GetIsDisplayed` returns a boolean indicating if the note is currently displayed.

The method `pfcDetailNoteInstructions::SetIsDisplayed` sets the display flag for the note.

The method `pfcDetailNoteInstructions::GetIsReadOnly` determines whether the note can be edited by the user, while the method `pfcDetailNoteInstructions::SetIsReadOnly` toggles the read only status of the note.

The method `pfcDetailNoteInstructions::GetIsMirrored` determines whether the note is mirrored, while the method `pfcDetailNoteInstructions::SetIsMirrored` toggles the mirrored status of the note.

The method `pfcDetailNoteInstructions::GetHorizontal` returns the value of the horizontal justification of the note, while the method `pfcDetailNoteInstructions::SetHorizontal` sets the value of the horizontal justification of the note.

The method `pfcDetailNoteInstructions::GetVertical` returns the value of the vertical justification of the note, while the method `pfcDetailNoteInstructions::SetVertical` sets the value of the vertical justification of the note.

The method `pfcDetailNoteInstructions::GetColor` returns the color of the detail note item. The method returns a null value to represent the default drawing color.

Use the method `pfcDetailNoteInstructions::SetColor` to set the color of the detail note item. Pass null to use the default drawing color.

The method `pfcDetailNoteInstructions::GetLeader` returns the locations of the detail note item and information about the leaders.

The method `pfcDetailNoteInstructions::SetLeader` sets the values of the location of the detail note item and the locations where the leaders are attached to the drawing.

The method `pfcDetailNoteInstructions::GetTextAngle` returns the value of the angle of the text used in the note. The method returns a null value if the angle is 0.0.

The method `pfcDetailNoteInstructions::SetTextAngle` sets the value of the angle of the text used in the note. Pass null to use the angle 0.0.

Detail Notes Information

Methods Introduced:

- **pfcDetailNoteItem::GetInstructions**
- **pfcDetailNoteItem::GetSymbolDef**
- **pfcDetailNoteItem::GetLineEnvelope**
- **pfcDetailNoteItem::GetModelReference**

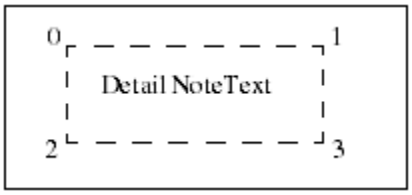
The method `pfcDetailNoteItem::GetInstructions` returns an instructions data object that describes how to construct the detail note item. This method takes a `ProBoolean` argument, *GiveParametersAsNames*, which determines whether symbolic representations of parameters and drawing properties in the note text should be displayed, or the actual text seen by the user should be displayed.

Note

Creo does not resolve and replace symbolic callouts for notes which are not displayed. Therefore, if the note is not displayed or is hidden in a layer, the text retrieved may contain symbolic callouts, even when *GiveParametersAsNames* is false.

The method `pfcDetailNoteItem::GetSymbolDef` returns the symbol definition that contains the note. The method returns a null value if the note is not a part of a symbol definition.

The method `pfcDetailNoteItem::GetLineEnvelope` determines the screen coordinates of the envelope around the detail note. This envelope is defined by four points. The following figure illustrates how the point order is determined.



The ordering of the points is maintained even if the notes are mirrored or are at an angle.

The method `pfcDetailNoteItem::GetModelReference` returns the model referenced by the parameterized text in a note. The model is referenced based on the line number and the text index where the parameterized text appears.

Details Notes Operations

Methods Introduced:

- **pfcDetailNoteItem::Draw**
- **pfcDetailNoteItem::Show**
- **pfcDetailNoteItem::Erase**
- **pfcDetailNoteItem::Remove**
- **pfcDetailNoteItem::KeepArrowTypeAsIs**
- **pfcDetailNoteItem::Modify**

The method `pfcDetailNoteItem::Draw` temporarily draws a detail note item, so that it is removed during the next draft regeneration.

The method `pfcDetailNoteItem::Show` displays the note item, such that it is repainted during the next draft regeneration.

The method `pfcDetailNoteItem::Erase` undraws a detail note item temporarily, so that it is redrawn during the next draft regeneration.

The method `pfcDetailNoteItem::Remove` undraws a detail note item permanently, so that it is not redrawn during the next draft regeneration.

The method `pfcDetailNoteItem::KeepArrowTypeAsIs` allows you to keep arrow type of the leader note as it is, after a note is modified. You must call this method before the method `pfcDetailNoteItem::Modify` is called.

The method `pfcDetailNoteItem::Modify` modifies the definition of an existing detail note item based on the instructions object that describes the new detail note item.

Detail Groups

A detail group in Creo Object TOOLKIT C++ is represented by the interface `pfcDetailGroupItem`. It is a child of the `pfcDetailItem` interface.

The interface `pfcDetailGroupInstructions` contains information used to describe a detail group item.

Instructions

Method Introduced:

- **pfcDetailGroupInstructions::Create**
- **pfcDetailGroupInstructions::GetName**
- **pfcDetailGroupInstructions::SetName**
- **pfcDetailGroupInstructions::GetElements**
- **pfcDetailGroupInstructions::SetElements**
- **pfcDetailGroupInstructions::GetIsDisplayed**
- **pfcDetailGroupInstructions::SetIsDisplayed**

The method `pfcDetailGroupInstructions::Create` creates an instruction data object that describes how to construct a detail group for use in `pfcDetailItemOwner::CreateDetailItem` and `pfcDetailGroupItem::Modify`.

Note

Changes to the values of a `pfcDetailGroupInstructions` object do not take effect until that instructions object is used to modify the group using `pfcDetailGroupItem::Modify`.

The method `pfcDetailGroupInstructions::GetName` returns the name of the detail group.

The method `pfcDetailGroupInstructions::SetName` sets the name of the detail group.

The method `pfcDetailGroupInstructions::GetElements` returns the sequence of the detail items(notes, groups and entities) contained in the group.

The method `pfcDetailGroupInstructions::SetElements` sets the sequence of the detail items contained in the group.

The method `pfcDetailGroupInstructions::GetIsDisplayed` returns whether the detail group is displayed in the drawing.

The method `pfcDetailGroupInstructions::SetIsDisplayed` toggles the display of the detail group.

Detail Groups Information

Method Introduced:

- **pfcDetailGroupItem::GetInstructions**

The method `pfcDetailGroupItem::GetInstructions` gets a data object that describes how to construct a detail group item. The method returns the data object describing the detail group item.

Detail Groups Operations

Methods Introduced:

- **pfcDetailGroupItem::Draw**
- **pfcDetailGroupItem::Erase**
- **pfcDetailGroupItem::Modify**

The method `pfcDetailGroupItem::Draw` temporarily draws a detail group item, so that it is removed during the next draft generation.

The method `pfcDetailGroupItem::Erase` temporarily undraws a detail group item, so that it is redrawn during the next draft generation.

The method `pfcDetailGroupItem::Modify` changes the definition of a detail group item based on the data object that describes how to construct a detail group item.

Detail Symbols

Detail Symbol Definitions

A detail symbol definition in Creo Object TOOLKIT C++ is represented by the interface `pfcDetailSymbolDefItem`. It is a child of the `pfcDetailItem` interface.

The interface `pfcDetailSymbolDefInstructions` contains information that describes a symbol definition. It can be used when creating symbol definition entities or while accessing existing symbol definition entities.

Instructions

Methods Introduced:

- **pfcDetailSymbolDefInstructions::Create**

- **pfcDetailSymbolDefInstructions::GetSymbolHeight**
- **pfcDetailSymbolDefInstructions::SetSymbolHeight**
- **pfcDetailSymbolDefInstructions::GetHasElbow**
- **pfcDetailSymbolDefInstructions::SetHasElbow**
- **pfcDetailSymbolDefInstructions::GetIsTextAngleFixed**
- **pfcDetailSymbolDefInstructions::SetIsTextAngleFixed**
- **pfcDetailSymbolDefInstructions::GetScaledHeight**
- **pfcDetailSymbolDefInstructions::GetAttachments**
- **pfcDetailSymbolDefInstructions::SetAttachments**
- **pfcDetailSymbolDefInstructions::GetFullPath**
- **pfcDetailSymbolDefInstructions::SetFullPath**
- **pfcDetailSymbolDefInstructions::GetReference**
- **pfcDetailSymbolDefInstructions::SetReference**

The method `pfcDetailSymbolDefInstructions::Create` creates an instruction data object that describes how to create a symbol definition based on the path and name of the symbol definition. The instructions object is passed to the methods `pfcCreateDetailItem` and `pfcModify`.

Note

Changes to the values of a `pfcDetailSymbolDefInstructions` object do not take effect until that instructions object is used to modify the definition using the method `pfcDetailSymbolDefItem::Modify`.

The method `pfcDetailSymbolDefInstructions::GetSymbolHeight` returns the value of the height type for the symbol definition. The symbol definition height options are as follows:

- `pfcSYMDEF_FIXED`—Symbol height is fixed.
- `pfcSYMDEF_VARIABLE`—Symbol height is variable.
- `pfcSYMDEF_RELATIVE_TO_TEXT`—Symbol height is determined relative to the text height.

The method `pfcDetailSymbolDefInstructions::SetSymbolHeight` sets the value of the height type for the symbol definition.

The method `pfcDetailSymbolDefInstructions::GetHasElbow` determines whether the symbol definition includes an elbow.

The method `pfcDetailSymbolDefInstructions::SetHasElbow` decides if the symbol definition should include an elbow.

The method `pfcDetailSymbolDefInstructions::GetIsTextAngleFixed` returns whether the text of the angle is fixed.

The method `pfcDetailSymbolDefInstructions::SetIsTextAngleFixed` toggles the requirement that the text angle be fixed.

The method `pfcDetailSymbolDefInstructions::GetScaledHeight` returns the height of the symbol definition in inches.

The method `pfcDetailSymbolDefInstructions::GetAttachments` returns the value of the sequence of the possible instance attachment points for the symbol definition.

The method `pfcDetailSymbolDefInstructions::SetAttachments` sets the value of the sequence of the possible instance attachment points for the symbol definition.

The method `pfcDetailSymbolDefInstructions::GetFullPath` returns the value of the complete path of the symbol definition file.

The method `pfcDetailSymbolDefInstructions::SetFullPath` sets the value of the complete path of the symbol definition path.

The method `pfcDetailSymbolDefInstructions::GetReference` returns the text reference information for the symbol definition. It returns a null value if the text reference is not used. The text reference identifies the text item used for a symbol definition which has a height type of `pfcSYMDEF_RELATIVE_TO_TEXT`.

The method `pfcDetailSymbolDefInstructions::SetReference` sets the text reference information for the symbol definition.

Detail Symbol Definitions Information

Methods Introduced:

- **pfcDetailSymbolDefItem::ListDetailItems**
- **pfcDetailSymbolDefItem::GetInstructions**

The method `pfcDetailSymbolDefItem::ListDetailItems` lists the detail items in the symbol definition based on the type of the detail item.

The method `pfcDetailSymbolDefItem::GetInstructions` returns an instruction data object that describes how to construct the symbol definition.

Detail Symbol Definitions Operations

Methods Introduced:

- **pfcDetailSymbolDefItem::CreateDetailItem**
- **pfcDetailSymbolDefItem::Modify**

The method `pfcDetailSymbolDefItem::CreateDetailItem` creates a detail item in the symbol definition based on the instructions data object. The method returns the detail item in the symbol definition.

The method `pfcDetailSymbolDefItem::Modify` modifies a symbol definition based on the instructions data object that contains information about the modifications to be made to the symbol definition.

Retrieving Symbol Definitions

Methods Introduced:

- **pfcDetailItemOwner::RetrieveSymbolDefItem**

From Creo 4.0 F000 onwards, the method `pfcDetailItemOwner::RetrieveSymbolDefinition` has been deprecated. Use the method `pfcDetailItemOwner::RetrieveSymbolDefItem` instead.

Creo Parametric symbols exist in two different areas: the user-defined area and the system symbols area.

The method `pfcDetailItemOwner::RetrieveSymbolDefItem` retrieves a symbol definition from the user-defined location designated by the configuration option `pro_symbol_dir`. The symbol definition should have been previously saved to a file using Creo Parametric.

The method `pfcDetailItemOwner::RetrieveSymbolDefItem` also retrieves a symbol definition from the system directory. The system area contains symbols provided by Creo Parametric with the Detail module (such as the Welding Symbols Library).

The input parameters of this method are:

- *FileName*—Name of the symbol definition file.
- *Source*—Source of the symbol definition file. The input parameter *Source* is defined by the enumerated type `pfcDetailSymbolDefItemSource`. The valid values which are supported are listed below:
  - `pfcDTLSYMDEF_SRC_SYSTEM`—Specifies the system symbol definition directory.
  - `pfcDTLSYMDEF_SRC_PATH`—Specifies the absolute path to a directory containing the symbol definition.

- *FilePath*—Path to the symbol definition file. It is relative to the path specified by the option `pro_symbol_dir` in the configuration file. A null value indicates that the function should search the current directory.
- *Version*—Numerical version of the symbol definition file. A null value retrieves the latest version.
- *UpdateUnconditionally*—True if Creo should update existing instances of this symbol definition, or false to quit the operation if the definition exists in the model.

The method returns the retrieved symbol definition.

Detail Symbol Instances

A detail symbol instance in Creo Object TOOLKIT C++ is represented by the interface `pfcDetailSymbolInstItem`. It is a child of the `pfcDetailItem` interface.

The interface `pfcDetailSymbolInstInstructions` contains information that describes a symbol instance. It can be used when creating symbol instances and while accessing existing groups.

Instructions

Methods Introduced:

- **pfcDetailSymbolInstInstructions::Create**
- **pfcDetailSymbolInstInstructions::GetIsDisplayed**
- **pfcDetailSymbolInstInstructions::SetIsDisplayed**
- **pfcDetailSymbolInstInstructions::GetColor**
- **pfcDetailSymbolInstInstructions::SetColor**
- **pfcDetailSymbolInstInstructions::GetSymbolDef**
- **pfcDetailSymbolInstInstructions::SetSymbolDef**
- **pfcDetailSymbolInstInstructions::GetAttachOnDefType**
- **pfcDetailSymbolInstInstructions::SetAttachOnDefType**
- **pfcDetailSymbolInstInstructions::GetDefAttachment**
- **pfcDetailSymbolInstInstructions::SetDefAttachment**
- **pfcDetailSymbolInstInstructions::GetInstAttachment**
- **pfcDetailSymbolInstInstructions::SetInstAttachment**
- **pfcDetailSymbolInstInstructions::GetAngle**
- **pfcDetailSymbolInstInstructions::SetAngle**
- **pfcDetailSymbolInstInstructions::GetScaledHeight**
- **pfcDetailSymbolInstInstructions::SetScaledHeight**

- **pfcDetailSymbolInstInstructions::GetTextValues**
- **pfcDetailSymbolInstInstructions::SetTextValues**
- **pfcDetailSymbolInstInstructions::GetCurrentTransform**
- **pfcDetailSymbolInstInstructions::SetGroups**

The method `pfcDetailSymbolInstInstructions::Create` creates a data object that contains information about the placement of a symbol instance.

Note

Changes to the values of a `pfcDetailSymbolInstInstructions` object do not take effect until that instructions object is used to modify the instance using `pfcDetailSymbolInstItem::Modify`.

The method `pfcDetailSymbolInstInstructions::GetIsDisplayed` returns a value that specifies whether the instance of the symbol is displayed.

Use the method `pfcDetailSymbolInstInstructions::SetIsDisplayed` to switch the display of the symbol instance.

The method `pfcDetailSymbolInstInstructions::GetColor` returns the color of the detail symbol instance. A null value indicates that the default drawing color is used.

The method `pfcDetailSymbolInstInstructions::SetColor` sets the color of the detail symbol instance. Pass null to use the default drawing color.

The method `pfcDetailSymbolInstInstructions::GetSymbolDef` returns the symbol definition used for the instance.

The method `pfcDetailSymbolInstInstructions::SetSymbolDef` sets the value of the symbol definition used for the instance.

The method `pfcDetailSymbolInstInstructions::GetAttachOnDefType` returns the attachment type of the instance. The method returns a null value if the attachment represents a free attachment. The attachment options are as follows:

- `pfcSYMDEFATTACH_FREE`—Attachment on a free point.
- `pfcSYMDEFATTACH_LEFT_LEADER`—Attachment via a leader on the left side of the symbol.
- `pfcSYMDEFATTACH_RIGHT_LEADER`— Attachment via a leader on the right side of the symbol.
- `pfcSYMDEFATTACH_RADIAL_LEADER`—Attachment via a leader at a radial location.

- `pfcSYMDEFATTACH_ON_ITEM`—Attachment on an item in the symbol definition.
- `pfcSYMDEFATTACH_NORMAL_TO_ITEM`—Attachment normal to an item in the symbol definition.

The method `pfcDetailSymbolInstInstructions::SetAttachOnDefType` sets the attachment type of the instance.

The method `pfcDetailSymbolInstInstructions::GetDefAttachment` returns the value that represents the way in which the instance is attached to the symbol definition.

The method `pfcDetailSymbolInstInstructions::SetDefAttachment` specifies the way in which the instance is attached to the symbol definition.

The method `pfcDetailSymbolInstInstructions::GetInstAttachment` returns the value of the attachment of the instance that includes location and leader information.

The method `pfcDetailSymbolInstInstructions::SetInstAttachment` sets value of the attachment of the instance.

The method `pfcDetailSymbolInstInstructions::GetAngle` returns the value of the angle at which the instance is placed. The method returns a null value if the value of the angle is 0 degrees.

The method `pfcDetailSymbolInstInstructions::SetAngle` sets the value of the angle at which the instance is placed.

The method `pfcDetailSymbolInstInstructions::GetScaledHeight` returns the height of the symbol instance in the owner drawing or model coordinates. This value is consistent with the height value shown for a symbol instance in the Creo user interface.

Note

The scaled height obtained using the above method is partially based on the properties of the symbol definition assigned using the method `pfcDetail.DetailSymbolInstInstructions.GetSymbolDef`. Changing the symbol definition may change the calculated value for the scaled height.

The method `pfcDetailSymbolInstInstructions::SetScaledHeight` sets the value of the height of the symbol instance in the owner drawing or model coordinates.

The method `pfcDetailSymbolInstInstructions::GetTextValues` returns the sequence of variant text values used while placing the symbol instance.

The method `pfcDetailSymbolInstInstructions::SetTextValues` sets the sequence of variant text values while placing the symbol instance.

The method `pfcDetailSymbolInstInstructions::GetCurrentTransform` returns the coordinate transformation matrix to place the symbol instance.

The method `pfcDetailSymbolInstInstructions::SetGroups` sets the option for displaying groups in the symbol instance using the enumerated type `pfcDetailSymbolGroupOption`. It has the following values:

- `pfcDETAIL_SYMBOL_GROUP_INTERACTIVE`—Symbol groups are interactively selected for display. This is the default value in the `GRAPHICS` mode.
- `pfcDETAIL_SYMBOL_GROUP_ALL`—All non-exclusive symbol groups are included for display.
- `pfcDETAIL_SYMBOL_GROUP_NONE`—None of the non-exclusive symbol groups are included for display.
- `pfcDETAIL_SYMBOL_GROUP_CUSTOM`—Symbol groups specified by the application are displayed.

Refer to the section Detail Symbol Groups on page 195 for more information on detail symbol groups.

Detail Symbol Instances Information

Method Introduced:

- **pfcDetailSymbolInstItem::GetInstructions**

The method `pfcDetailSymbolInstItem::GetInstructions` returns an instructions data object that describes how to construct a symbol instance. This method takes a `ProBoolean` argument, *GiveParametersAsNames*, which determines whether symbolic representations of parameters and drawing properties in the symbol instance should be displayed, or the actual text seen by the user should be displayed.

Detail Symbol Instances Operations

Methods Introduced:

- **pfcDetailSymbolInstItem::Draw**

- **pfcDetailSymbolInstItem::Erase**
- **pfcDetailSymbolInstItem::Show**
- **pfcDetailSymbolInstItem::Remove**
- **pfcDetailSymbolInstItem::Modify**

The method `pfcDetailSymbolInstItem::Draw` draws a symbol instance temporarily to be removed on the next draft regeneration.

The method `pfcDetailSymbolInstItem::Erase` undraws a symbol instance temporarily from the display to be redrawn on the next draft generation.

The method `pfcDetailSymbolInstItem::Show` displays a symbol instance to be repainted on the next draft regeneration.

The method `pfcDetailSymbolInstItem::Remove` deletes a symbol instance permanently.

The method `pfcDetailSymbolInstItem::Modify` modifies a symbol instance based on the instructions data object that contains information about the modifications to be made to the symbol instance.

Detail Symbol Groups

A detail symbol group in Creo Object TOOLKIT C++ is represented by the interface `pfcDetailSymbolGroup`. It is a child of the `pfcObject` interface. A detail symbol group is accessible only as a part of the contents of a detail symbol definition or instance.

The interface `pfcDetailSymbolGroupInstructions` contains information that describes a symbol group. It can be used when creating new symbol groups, or while accessing or modifying existing groups.

Instructions

Methods Introduced:

- **pfcDetailSymbolGroupInstructions::Create**
- **pfcDetailSymbolGroupInstructions::GetItems**
- **pfcDetailSymbolGroupInstructions::SetItems**
- **pfcDetailSymbolGroupInstructions::GetName**
- **pfcDetailSymbolGroupInstructions::SetName**

The method `pfcDetailSymbolGroupInstructions::Create` creates the `pfcDetailSymbolGroupInstructions` data object that stores the name of the symbol group and the list of detail items to be included in the symbol group.

Note

Changes to the values of the `pfcDetailSymbolGroupInstructions` data object do not take effect until this object is used to modify the instance using the method `pfcDetailSymbolGroup.Modify`.

The method `pfcDetailSymbolGroupInstructions::GetItems` returns the list of detail items included in the symbol group.

The method `pfcDetailSymbolGroupInstructions::SetItems` sets the list of detail items to be included in the symbol group.

The method `pfcDetailSymbolGroupInstructions::GetName` returns the name of the symbol group.

The method `pfcDetailSymbolGroupInstructions::SetName` assigns the name of the symbol group.

Detail Symbol Group Information

Methods Introduced:

- **pfcDetailSymbolGroup::GetInstructions**
- **pfcDetailSymbolGroup::GetParentGroup**
- **pfcDetailSymbolGroup::GetParentDefinition**
- **pfcDetailSymbolGroup::ListChildren**
- **pfcDetailSymbolDefItem::ListSubgroups**
- **pfcDetailSymbolDefItem::IsSubgroupLevelExclusive**
- **pfcDetailSymbolInstItem::ListGroups**

The method `pfcDetailSymbolGroup::GetInstructions` returns the `pfcDetailSymbolGroupInstructions` data object that describes how to construct a symbol group.

The method `pfcDetailSymbolGroup::GetParentGroup` returns the parent symbol group to which a given symbol group belongs.

The method `pfcDetailSymbolGroup::GetParentDefinition` returns the symbol definition of a given symbol group.

The method `pfcDetailSymbolGroup::ListChildren` lists the subgroups of a given symbol group.

The method `pfcDetailSymbolDefItem::ListSubgroups` lists the subgroups of a given symbol group stored in the symbol definition at the indicated level.

The method `pfcDetailSymbolDefItem::IsSubgroupLevelExclusive` identifies if the subgroups of a given symbol group stored in the symbol definition at the indicated level are exclusive or independent. If groups are exclusive, only one of the groups at this level can be active in the model at any time. If groups are independent, any number of groups can be active.

The method `pfcDetailSymbolInstItem::ListGroups` lists the symbol groups included in a symbol instance. The `pfcSymbolGroupFilter` argument determines the types of symbol groups that can be listed. It takes the following values:

- `pfcDTLSYMINST_ALL_GROUPS`—Retrieves all groups in the definition of the symbol instance.
- `pfcDTLSYMINST_ACTIVE_GROUPS`—Retrieves only those groups that are actively shown in the symbol instance.
- `pfcDTLSYMINST_INACTIVE_GROUPS`—Retrieves only those groups that are not shown in the symbol instance.

Detail Symbol Group Operations

Methods Introduced:

- **pfcDetailSymbolGroup::Delete**
- **pfcDetailSymbolGroup::Modify**
- **pfcDetailSymbolDefItem::CreateSubgroup**
- **pfcDetailSymbolDefItem::SetSubgroupLevelExclusive**
- **pfcDetailSymbolDefItem::SetSubgroupLevelIndependent**

The method `pfcDetailSymbolGroup::Delete` deletes the specified symbol group from the symbol definition. This method does not delete the entities contained in the group.

The method `pfcDetailSymbolGroup::Modify` modifies the specified symbol group based on the `pfcDetailSymbolGroupInstructions` data object that contains information about the modifications that can be made to the symbol group.

The method `pfcDetailSymbolDefItem::CreateSubgroup` creates a new subgroup in the symbol definition at the indicated level below the parent group.

The method `pfcDetailSymbolDefItem::SetSubgroupLevelExclusive` makes the subgroups of a symbol group exclusive at the indicated level in the symbol definition.

Note

After you set the subgroups of a symbol group as exclusive, only one of the groups at the indicated level can be active in the model at any time.

The method `pfcDetailSymbolDefItem::SetSubgroupLevelIndependent` makes the subgroups of a symbol group independent at the indicated level in the symbol definition.

Note

After you set the subgroups of a symbol group as independent, any number of groups at the indicated level can be active in the model at any time.

Detail Attachments

A detail attachment in Creo Object TOOLKIT C++ is represented by the interface `pfcAttachment`. It is used for the following tasks:

- The way in which a drawing note or a symbol instance is placed in a drawing.
- The way in which a leader on a drawing note or symbol instance is attached.

Method Introduced:

- **pfcAttachment::GetType**

The method `pfcAttachment::GetType` returns the `pfcAttachmentType` object containing the types of detail attachments. The detail attachment types are as follows:

- `pfcATTACH_FREE`—The attachment is at a free point possibly with respect to a given drawing view.
- `pfcATTACH_PARAMETRIC`—The attachment is to a point on a surface or an edge of a solid.
- `pfcATTACH_OFFSET`—The attachment is offset to another drawing view, to a model item, or to a 3D model annotation.
- `pfcATTACH_TYPE_UNSUPPORTED`—The attachment is to an item that cannot be represented in PFC at the current time. However, you can still retrieve the location of the attachment.

Free Attachment

The `pfcATTACH_FREE` detail attachment type is represented by the interface `pfcFreeAttachment`. It is a child of the `pfcAttachment` interface.

Methods Introduced:

- **pfcFreeAttachment::GetAttachmentPoint**
- **pfcFreeAttachment::SetAttachmentPoint**
- **pfcFreeAttachment::GetView**
- **pfcFreeAttachment::SetView**

The method `pfcFreeAttachment::GetAttachmentPoint` returns the attachment point. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes.

The method `pfcFreeAttachment::SetAttachmentPoint` sets the attachment point.

The method `pfcFreeAttachment::GetView` returns the drawing view to which the attachment is related. The attachment point is relative to the drawing view, that is the attachment point moves when the drawing view is moved. This method returns a `NULL` value, if the detail attachment is not related to a drawing view, but is placed at the specified location in the drawing sheet, or if the attachment is offset to a model item or to a 3D model annotation.

The method `pfcFreeAttachment::SetView` sets the drawing view.

Parametric Attachment

The `pfcATTACH_PARAMETRIC` detail attachment type is represented by the interface `pfcParametricAttachment`. It is a child of the `pfcAttachment` interface.

Methods Introduced:

- **pfcParametricAttachment::GetAttachedGeometry**
- **pfcParametricAttachment::SetAttachedGeometry**

The method `pfcParametricAttachment::GetAttachedGeometry` returns the `pfcSelection` object representing the item to which the detail attachment is attached. This includes the drawing view in which the attachment is made.

The method `pfcParametricAttachment::SetAttachedGeometry` assigns the `pfcSelection` object representing the item to which the detail attachment is attached. This object must include the target drawing view. The attachment will occur at the selected parameters.

Offset Attachment

The `pfcATTACH_OFFSET` detail attachment type is represented by the interface `pfcOffsetAttachment`. It is a child of the `pfcAttachment` interface.

Methods Introduced:

- **pfcOffsetAttachment::GetAttachedGeometry**
- **pfcOffsetAttachment::SetAttachedGeometry**
- **pfcOffsetAttachment::GetAttachmentPoint**
- **pfcOffsetAttachment::SetAttachmentPoint**

The method `pfcOffsetAttachment::GetAttachedGeometry` returns the `pfcSelection` object representing the item to which the detail attachment is attached. This includes the drawing view where the attachment is made, if the offset reference is in a model.

The method `pfcOffsetAttachment::SetAttachedGeometry` assigns the `pfcSelection` object representing the item to which the detail attachment is attached. This can include the drawing view. The attachment will occur at the selected parameters.

The method `pfcOffsetAttachment::GetAttachmentPoint` returns the attachment point. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes. The distance from the attachment point to the location of the item to which the detail attachment is attached is saved as the offset distance.

The method `pfcOffsetAttachment::SetAttachmentPoint` sets the attachment point in screen coordinates.

Unsupported Attachment

The `pfcATTACH_TYPE_UNSUPPORTED` detail attachment type is represented by the interface `pfcUnsupportedAttachment`. It is a child of the `pfcAttachment` interface.

Methods Introduced:

- **pfcUnsupportedAttachment::GetAttachmentPoint**
- **pfcUnsupportedAttachment::SetAttachmentPoint**

The method `pfcUnsupportedAttachment::GetAttachmentPoint` returns the attachment point. This location is in screen coordinates for drawing items, symbol instances and surface finishes on flat-to-screen annotation planes, and in model coordinates for symbols and surface finishes on 3D model annotation planes.

The method `pfcUnsupportedAttachment::SetAttachmentPoint` assigns the attachment point in screen coordinates.

10

Solid

Most of the objects and methods in Creo Object TOOLKIT C++ are used with solid models (parts and assemblies). Because solid objects inherit from the interface `pfcModel`, you can use any of the `pfcModel` methods on any `pfcSolid`, `pfcPart`, or `pfcAssembly` object.

Getting a Solid Object

Methods Introduced:

- **pfcBaseSession::CreatePart**
- **pfcBaseSession::CreateAssembly**
- **pfcComponentPath::GetRoot**
- **pfcComponentPath::GetLeaf**
- **pfcMFG::GetSolid**
- **wfcWSession::GetSolid**

The methods `pfcBaseSession::CreatePart` and `pfcBaseSession::CreateAssembly` create new solid models with the names you specify.

The methods `pfcComponentPath::GetRoot` and `pfcComponentPath::GetLeaf` specify the solid objects that make up the component path of an assembly component model. You can get a component path object from any component that has been interactively selected.

The method `pfcMFG::GetSolid` retrieves the storage solid in which the manufacturing model's features are placed. In order to create a UDF group in the manufacturing model, call the method `pfcSolid::CreateUDFGroup` on the storage solid.

The method `wfcWSession::GetSolid` returns the handle to the specified solid. You must specify the name of the solid and the solid type as the input arguments.

Solid Information

Methods Introduced:

- **pfcSolid::GetRelativeAccuracy**
- **pfcSolid::SetRelativeAccuracy**
- **pfcSolid::GetAbsoluteAccuracy**
- **pfcSolid::SetAbsoluteAccuracy**

You can set the relative and absolute accuracy of any solid model using these methods. Relative accuracy is relative to the size of the solid. For example, a relative accuracy of .01 specifies that the solid must be accurate to within 1/100 of its size. Absolute accuracy is measured in absolute units (inches, centimeters, and so on).

Note

For a change in accuracy to take effect, you must regenerate the model.

Displaying a Solid

Method Introduced:

- **wfcWSolid::DisplaySolid**

The method `wfcWSolid::DisplaySolid` displays the specified solid in the current Creo window. This method does not make the solid as the current object in the Creo application.

Solid Operations

Methods Introduced:

- **pfcSolid::GetGeomOutline**
- **pfcSolid::GetSurfaceSolidBody**
- **pfcSolid::GetEdgeSolidBody**
- **pfcSolid::EvalOutline**
- **pfcSolid::GetIsSkeleton**
- **pfcSolid::ListGroups**
- **wfcWSolid::GetSolidFeatureStatusFlags**
- **wfcWSolid::GetIsNoResolveMode**
- **wfcStyleState::IsStyleStateDefault**
- **wfcWSolid::GetStyleStateFromName**
- **wfcWSolid::GetStyleStateFromId**
- **wfcWSolid::GetActiveStyleState**
- **wfcWSolid::ActivateStyleState**
- **wfcWSolid::ListStyleStateItems**
- **wfcWSolid::GetDisplayOutline**
- **wfcWSolid::FindShellsAndVoids**
- **wfcShellData::GetShellOrders**
- **wfcShellOrder::GetOrientation**
- **wfcShellOrder::GetFirstFace**
- **wfcShellOrder::GetNumberOfFaces**

- **wfcShellOrder::GetAmbientShell**
- **wfcShellData::GetShellFaces**
- **wfcShellFace::GetSurfaceId**
- **wfcShellFace::GetContour**
- **wfcWSolid::CheckFamilyTable**

The method `pfcSolid::GetGeomOutline` returns the three-dimensional bounding box for the specified solid.

Note

Do not use `pfcSolid::GetGeomOutline` to calculate the outline of a solid as the dimensions of the boundary box could be slightly bigger than the outline dimensions of the geometry. Use `pfcSolid::EvalOutline` to compute an accurate outline of a solid.

The method `pfcSolid::GetSurfaceSolidBody` returns the body to which the surface belongs.

The method `pfcSolid::GetEdgeSolidBody` returns the body to which the edge belongs.

The method `pfcSolid::EvalOutline` also returns a three-dimensional bounding box, but you can specify the coordinate system used to compute the extents of the solid object.

The method `pfcSolid::GetIsSkeleton` determines whether the part model is a skeleton or a concept model. It returns a true value if the model is a skeleton, else it returns a false.

The method `pfcSolid::ListGroups` returns the list of groups including UDFs in the solid.

The method `wfcWSolid::GetSolidFeatureStatusFlags` returns a list of objects representing the status of each feature in the model.

The method `wfcWSolid::GetIsNoResolveMode` returns `True` if the model regeneration is set to no resolve mode.

The method `wfcStyleState::IsStyleStateDefault` determines if the display style of the solid component is the default style for the owner model.

The method `wfcStyleState::GetStyleStateFromName` returns the handle to the specified style state in the component with the style state name as the input argument.

The method `wfcWSolid::GetStyleStateFromId` returns the handle to the specified style state in the component with the style state ID as the input argument.

Use the method `wfcWSolid::GetActiveStyleState` to get the current active style state in the solid.

The method `wfcWSolid::ActivateStyleState` activates the specified style state as current display style. The input arguments to this method are:

- *WStyleState*—Specifies the handle to the style state.
- *RedisplayFlag*—Specifies if the assembly must be displayed to the set display style in the call to the method. If set to True, user must call the required methods to regenerate the assembly.

The method `wfcWSolid::ListStyleStateItems` returns a list of style states in the solid.

The method `wfcWSolid::GetDisplayOutline` returns the maximum and minimum values of x, y, and z coordinates for the display outline of the solid, with respect to the default coordinate system.

The method `wfcWSolid::FindShellsAndVoids` returns a list of surface-contour pairs for each shell and void in the solid as a `wfcShellData` object. When a surface is split it has more than one external contour. In this case the contours may belong to different shells.

The method `wfcShellData::GetShellOrders` returns information about the ordered list of surface-contour pairs in a solid.

The method `wfcShellOrder::GetOrientation` returns an integer value to indicate the orientation of the shell. `1` specifies that the shell is oriented outward and `-1` specifies that the shell is oriented inward, that is, it is a void.

The method `wfcShellOrder::GetFirstFace` returns the index of the first face for the specified shell or void.

The method `wfcShellOrder::GetNumberOfFaces` returns the number of faces in the shell.

A single shell or void can consist of many shells or voids within it. The method `wfcShellOrder::GetAmbientShell` returns the index of the ambient shell, that is, the smallest shell inside which the specified shell is located. The method returns `-1` if the specified shell is external and is not located inside any other shell.

The method `wfcShellData::GetShellFaces` returns information about the shell surface and contour.

The method `wfcShellFace::GetSurfaceId` returns the ID of the shell surface.

The method `wfcShellFace::GetContour` returns the information about the contour as a `pfcContour` object.

Use the method `wfcSolid::CheckFamilyTable` to check if the specified solid has a family table associated with it. The method also checks whether the associated family table is empty.

Regenerating a Solid

Methods Introduced:

- **pfcSolid::Regenerate**
- **pfcRegenInstructions::Create**
- **pfcRegenInstructions::SetAllowFixUI**
- **pfcRegenInstructions::SetForceRegen**
- **pfcRegenInstructions::SetFromFeat**
- **pfcRegenInstructions::SetRefreshModelTree**
- **pfcRegenInstructions::SetResumeExcludedComponents**
- **pfcRegenInstructions::SetUpdateAssemblyOnly**
- **pfcRegenInstructions::SetUpdateInstances**
- **pfcRegenInstructions::GetResolveModeRegen**
- **pfcRegenInstructions::SetResolveModeRegen**
- **wfcWSolid::WRegenerate**
- **wfcWRegenInstructions::Create**
- **wfcWRegenInstructions::GetNoResolveMode**
- **wfcWRegenInstructions::SetNoResolveMode**
- **wfcWRegenInstructions::GetResolveMode**
- **wfcWRegenInstructions::SetResolveMode**
- **wfcWRegenInstructions::GetAllowConfirm**
- **wfcWRegenInstructions::SetAllowConfirm**
- **wfcWRegenInstructions::GetUndoIfFail**
- **wfcWRegenInstructions::SetUndoIfFail**
- **wfcWRegenInstructions::GetTopAsmOnly**
- **wfcWRegenInstructions::SetTopAsmOnly**

The method `pfcSolid::Regenerate` causes the solid model to regenerate according to the instructions provided in the form of the `pfcRegenInstructions` object. Passing a null value for the instructions argument causes an automatic regeneration.

In the No-Resolve mode, if a model and feature regeneration fails, failed features and children of failed features are created and regeneration of other features continues. However, Creo Object TOOLKIT C++ does not support regeneration in

this mode. The method `pfcSolid::Regenerate` throws an exception `pfcXToolkitBadContext`, if Creo Parametric is running in the No-Resolve mode. To switch back to the Pro/ENGINEER Wildfire 4.0 behavior in the Resolve mode, set the configuration option `regen_failure_handling` to `resolve_mode` in the Creo Parametric session.

Note

Setting the configuration option to switch to Resolve mode ensures the old behavior as long as you do not retrieve the models saved under the No-Resolve mode. To consistently preserve the old behavior, use Resolve mode from the beginning and throughout your Creo Parametric session.

The `pfcRegenInstructions` object contains the following input parameters:

- *AllowFixUI*—Determines whether or not to activate the **Fix Model** user interface, if there is an error.

  Use the method `pfcRegenInstructions::SetAllowFixUI` to modify this parameter.
- *ForceRegen*—Forces the solid model to fully regenerate. All the features in the model are regenerated. If this parameter is false, Creo determines which features to regenerate. By default, it is false.

  Use the method `pfcRegenInstructions::SetForceRegen` to modify this parameter.
- *FromFeat*—Not currently used. This parameter is reserved for future use.

  Use the method `pfcRegenInstructions::SetFromFeat` to modify this parameter.
- *RefreshModelTree*—Refreshes the Creo model tree after regeneration. The model must be active to use this attribute. If this attribute is false, the model tree is not refreshed. By default, it is false.

  Use the method `pfcRegenInstructions::SetRefreshModelTree` to modify this parameter.
- *ResumeExcludedComponents*—Enables Creo to resume the available excluded components of the simplified representation during regeneration. This results in a more accurate update of the simplified representation.

  Use the method `pfcRegenInstructions::SetResumeExcludedComponents` to modify this parameter.

- *UpdateAssemblyOnly*—Updates the placements of an assembly and all its sub-assemblies, and regenerates the assembly features and intersected parts. If the affected assembly is retrieved as a simplified representation, then the locations of the components are updated. If this attribute is false, the component locations are not updated, even if the simplified representation is retrieved. By default, it is false.

  Use the method `pfcRegenInstructions::SetUpdateAssemblyOnly` to modify this parameter.
- *UpdateInstances*—Updates the instances of the solid model in memory. This may slow down the regeneration process. By default, this attribute is false.

  Use the method `pfcRegenInstructions::SetUpdateInstances` to modify this parameter.
- *ResolveModeRegen*—Allows regeneration of a solid in resolve mode. The regeneration behavior is controlled by temporarily overriding the default settings.

  Use the method `pfcRegenInstructions::SetResolveModeRegen` to modify this parameter. By default, it is false. If you want to set your Creo Parametric application in resolve mode, you need to set the value of this parameter to `true`.

Note

However, resolve mode will be deprecated in a future release of Creo Parametric. Hence, it is recommended to run the application without calling the method `pfcRegenInstructions::SetResolveModeRegen`.

The method `wfcWSolid::WRegenerate` regenerates the model according to the regeneration instructions provided in the form of the `wfcRegenInstructions` object. The `wfcRegenInstructions` object contains the following input parameters:

- *NoResolveMode*—Gets and sets the no resolve mode in a model using the methods `wfcWRegenInstructions::GetNoResolveMode` and `wfcWRegenInstructions::SetNoResolveMode`
- *ResolveMode*—Gets and sets the resolve mode in a model using the methods `wfcWRegenInstructions::GetResolveMode` and `wfcWRegenInstructions::SetResolveMode`

Note

The *NoResolveMode* and *ResolveMode* temporarily override the default settings, which control the regeneration behavior in a model.

AllowConfirm—This parameter has been deprecated from Creo Parametric 4.0 M030. Gets and sets the state of regeneration failure window in a model using the methods `wfcWRegenInstructions::GetAllowConfirm` and `wfcWRegenInstructions::SetAllowConfirm`

Note

The interactive dialog box which provided an option to retain failed features and children of failed features, if regeneration fails is no longer supported. Creo Parametric displays a warning message which gives details of failed features.

- *UndoIfFail*—If possible, gets and sets the undo mode if the regeneration of the model fails using the methods `wfcWRegenInstructions::GetUndoIfFail` and `wfcWRegenInstructions::SetUndoIfFail`

Note

The *AllowConfirm* and *UndoIfFail* cannot be used together and are applicable only when the input parameter is *NoResolveMode* in a model.

- *TopAsmOnly*—Forces only top level assembly to regenerate using the methods `wfcWRegenInstructions::GetTopAsmOnly` and `wfcWRegenInstructions::SetTopAsmOnly`. All the other models follow standard regeneration rules.

The method `wfcWRegenInstructions::Create` creates instructions for regenerating a model.

Combined States of a Solid

With a combined state, you can combine and apply multiple display states to a Creo model. Combined states are composed of the following two or more display states:

- Saved Views

- Layer state
- Annotations
- Cross section
- Exploded view
- Simplified representation
- Model style

Methods Introduced:

- **wfcWSolid::ListCombStates**
- **wfcCombState::GetCombStateData**
- **wfcCombStateData::Create**
- **wfcCombStateData::GetCombStateName**
- **wfcCombStateData::GetReferences**
- **wfcCombStateData::SetReferences**
- **wfcCombStateData::GetClipOption**
- **wfcCombStateData::SetClipOption**
- **wfcCombStateData::GetIsExploded**
- **wfcCombStateData::SetIsExploded**
- **wfcWSolid::CreateCombState**
- **wfcWSolid::GetActiveCombState**
- **wfcWSolid::ActivateCombState**
- **wfcWSolid::DeleteCombState**
- **wfcWSolid::GetAnnotationsOfActiveState**
- **wfcCombState::RedefineCombState**
- **wfcCombState::GetStateOfSupplGeometry**
- **wfcCombState::SetStateOfAnnotations**
- **wfcCombState::SetStateOfSupplGeometry**
- **wfcCombState::GetAnnotations**
- **wfcCombStateAnnotation::Create**
- **wfcCombStateAnnotation::GetAnnotation**
- **wfcCombStateAnnotation::SetAnnotation**
- **wfcCombStateAnnotation::GetOption**
- **wfcCombStateAnnotation::SetOption**
- **wfcCombState::AddAnnotations**
- **wfcCombState::RemoveAnnotations**
- **wfcCombState::EraseAnnotation**

- **wfcCombState::GetStateOfAnnotations**
- **wfcCombState::IsDefault**
- **wfcCombState::IsPublished**
- **wfcCombStateItem::Create**
- **wfcCombStateItem::GetItem**
- **wfcCombStateItem::SetItem**
- **wfcCombStateItem::GetOption**
- **wfcCombStateItem::SetOption**
- **wfcCombState::AddItems**
- **wfcCombState::GetItems**
- **wfcCombState::RemoveItems**

The method `wfcWSolid::ListCombStates` returns a list of combined states in the specified solid.

The method `wfcCombState::GetCombStateData` returns information for a specified combined state as a `wfcCombStateData` object.

The method `wfcCombStateData::Create` creates a the `wfcCombStateData` data object that contains information about the specified combined state.

The method `wfcCombStateData::GetCombStateName` returns the name of the combined state.

Use the methods `wfcCombStateData::GetReferences` and `wfcCombStateData::SetReferences` get and set an array of reference states of the type `pfcModelItem`.

The methods `wfcCombStateData::GetClipOption` and `wfcCombStateData::SetClipOption` get and set the cross section clip. This is applicable only in case of a valid reference of the type `pfcITEM_XSEC`. The `pfcITEM_XSEC` item represents a `wfcXSection` object or a zone feature. The values for the cross section clip are specified by the enumerated type `wfcCrossSectionClipOption`. The valid values are as follows:

- `wfcVIS_OPT_NONE`—Specifies that the cross section or zone feature is not clipped.
- `wfcVIS_OPT_FRONT`—Specifies that the cross section or zone feature is clipped by removing the material on the front side. The front side is where the positive normals of the planes of the cross section or zone feature are directed.
- `wfcVIS_OPT_BACK`—Specifies that the cross section or zone feature is clipped by removing the material on the back side.

The method `wfcCombStateData::GetIsExploded` returns a boolean value that specifies if the combined state is exploded. This value is available only if when a valid `pfcITEM_EXPLODED_STATE` reference state is retrieved. It is not available for Creo parts since an exploded state does not exist in the part mode.

The method `wfcWSolid::CreateCombState` creates a new combined state based on specified references. The input arguments of this method are as follows:

- *CombStateName*—Specifies the name of the new combined state.
- *CombStateData* —Specifies the combined state data as `wfcCombStateData` object.

In case, you do not want to redefine a reference state, pass the reference state with the same value. While redefining, you must specify reference states. If you do not pass reference states, the combined state is redefined to a `NO_STATE` state. `NO_STATE` state means the display of the reference state is not changed on activation of combined state.

The method `wfcWSolid::GetActiveCombState` retrieves the active combined state in a specified solid model. The active combined state is the default state when the model is opened.

Use the method `wfcWSolid::ActivateCombState` to activate a specified combined state.

Use the method `wfcWSolid::DeleteCombState` to delete a specified combined state. The method fails if the specified combined state is the default or active combined state.

The method `wfcWSolid::GetAnnotationsOfActiveState` gets annotations of active combined state.

Use the method `wfcCombState::RedefineCombState` to redefine a created combined state. Pass the `wfcCombStateData` object as the input argument.

Use the method `wfcCombState::GetStateOfSupplGeometry` to check if the display of supplementary geometry is controlled by the specified combined state or layers.

The methods `wfcCombState::SetStateOfAnnotations` and `wfcCombState::SetStateOfSupplGeometry` allow you to change the display of annotations and supplementary geometry by the combined state or layers.

The method `wfcCombState::GetAnnotations` retrieves annotations and their status flags from a specified combined state item as a `wfcCombStateAnnotations` object. The `wfcCombStateAnnotations` object uses the methods of the `wfcCombStateAnnotation` object to add and get annotations.

The method `wfcCombStateAnnotation::Create` creates a data object that contains information about annotations from a combined state. The input arguments are:

- *Annotation*—Specifies the annotation as a `wfcAnnotation` object.
- *Flag*—Specifies if the annotation must be displayed in the combined state in terms of the enumerated data type `wfcCombStateAnnotationOption`. The valid values are:
  - `wfcCOMBSTATE_ANNOTATION_SHOW`—Specifies that the annotation must be shown in the combined state.
  - `wfcCOMBSTATE_ANNOTATION_ERASE`—Specifies that the annotation must not be shown in the combined state.

The methods `wfcCombStateAnnotation::GetAnnotation` and `wfcCombStateAnnotation::SetAnnotation` retrieve and set the annotations for the combined state as a `wfcAnnotation` object.

Use the methods `wfcCombStateAnnotation::GetOption` and `wfcCombStateAnnotation::SetOption` to retrieve and specify if the displayed annotation is in the combined state in terms of the enumerated data type `wfcCombStateAnnotationOption`.

The method `wfcCombState::AddAnnotations` adds annotations to a specified combined state item.

Use the method `wfcCombState::RemoveAnnotations` to remove the annotations from a specified combined state item.

The method `wfcCombState::EraseAnnotation` removes an annotation from the display for the specified combined state.

The method `wfcCombState::GetStateOfAnnotations` checks if the display of annotations is controlled by the specified combined state or layers. The method returns `TRUE` when the display is controlled by combined state.

The method `wfcCombState::IsDefault` checks if the specified combined state is set as the default combined state for the model.

The method `wfcCombState::IsPublished` checks if the specified combined state has been published to Creo View.

The method `wfcCombStateItem::Create` creates a data object that contains information about items from a combined state. The input arguments are:

- *Item*—Specifies the items as a `pfcModelItem` object.
- *Flag*—Specifies if the annotation or supplementary geometry must be displayed in the combined state in terms of the enumerated data type `wfcCombStateAnnotationOption`. The valid values are:
  - `wfcCOMBSTATE_ANNOTATION_SHOW`—Specifies that the annotation or supplementary geometry must be shown in the combined state.

- wfcCOMBSTATE_ANNOTATION_ERASE—Specifies that the annotation or supplementary geometry must not be shown in the combined state.

The methods wfcCombStateItem::GetItem and wfcCombStateItem::SetItem retrieve and set the combined state items as a pfcModelItem object.

Use the methods wfcCombStateItem::GetOption and wfcCombStateItem::SetOption to retrieve and specify if the annotation or supplementary geometry must be displayed in the combined state or not. The method wfcCombStateItem::SetOption is defined by the enumerated data type wfcCombStateAnnotationOption.

The method wfcCombState::AddItems adds an array of annotations or supplementary geometry to a specified combined state item.

The method wfcCombState::GetItems retrieves an array of annotations or supplementary geometry and their status flags from a specified combined state item.

Use the method wfcCombState::RemoveItems to remove the annotations or supplementary geometry from a specified combined state item.

Solid Units

Each model has a basic system of units to ensure all material properties of that model are consistently measured and defined. All models are defined on the basis of the system of units. A part can have only one system of unit.

The following types of quantities govern the definition of units of measurement:

- Basic Quantities—The basic units and dimensions of the system of units. For example, consider the Centimeter GramSecond (CGS) system of unit. The basic quantities for this system of units are:
  - Length—cm
  - Mass—g
  - Force—dyne
  - Time—sec
  - Temperature—K
- Derived Quantities—The derived units are those that are derived from the basic quantities. For example, consider the Centimeter Gram Second (CGS) system of unit. The derived quantities for this system of unit are as follows:
  - Area—cm^2
  - Volume—cm^3
  - Velocity—cm/sec

In Creo Object TOOLKIT C++, individual units in the model are represented by the interface `pfcUnit`.

Types of Unit Systems

The types of systems of units are as follows:

- Pre-defined system of units—This system of unit is provided by default.
- Custom-defined system of units—This system of unit is defined by the user only if the model does not contain standard metric or nonmetric units, or if the material file contains units that cannot be derived from the predefined system of units or both.

In Creo application, the system of units are categorized as follows:

- Mass Length Time (MLT)—The following systems of units belong to this category:
  - `CGS—Centimeter Gram Second`
  - `MKS—Meter Kilogram Second`
  - `mmKS—millimeter Kilogram Second`
- Force Length Time (FLT)—The following systems of units belong to this category:
  - Creo Default—`Inch lbm Second`. This is the default system followed by Creo application.
  - `FPS—Foot Pound Second`
  - `IPS—Inch Pound Second`
  - `mmNS—Millimeter Newton Second`

In Creo Object TOOLKIT C++, the system of units followed by the model is represented by the interface `pfcUnitSystem`.

Accessing Individual Units

Methods Introduced:

- **pfcSolid::ListUnits**
- **pfcSolid::GetUnit**
- **pfcUnit::GetName**
- **pfcUnit::GetExpression**
- **pfcUnit::GetType**
- **pfcUnit::GetIsStandard**
- **pfcUnit::GetReferenceUnit**
- **pfcUnit::GetConversionFactor**

- **pfcUnitConversionFactor::GetOffset**
- **pfcUnitConversionFactor::GetScale**
- **wfcWModel::CreateUnitByExpression**
- **wfcWUnit::ModifyByExpression**

The method `pfcSolid::ListUnits` returns the list of units available to the specified model.

The method `pfcSolid::GetUnit` retrieves the unit, based on its name or expression for the specified model in the form of the `pfcUnit` object.

The method `pfcUnit::GetName` returns the name of the unit.

The method `pfcUnit::GetExpression` returns a user-friendly unit description in the form of the name (for example, `ksi`) for ordinary units and the expression (for example, `N/m^3`) for system-generated units.

The method `pfcUnit::GetType` returns the type of quantity represented by the unit in terms of the `pfcUnitType` object. The types of units are as follows:

- `pfcUNIT_LENGTH`—Specifies length measurement units.
- `pfcUNIT_MASS`—Specifies mass measurement units.
- `pfcUNIT_FORCE`—Specifies force measurement units.
- `pfcUNIT_TIME`—Specifies time measurement units.
- `pfcUNIT_TEMPERATURE`—Specifies temperature measurement units.
- `pfcUNIT_ANGLE`—Specifies angle measurement units.

The method `pfcUnit::GetIsStandard` identifies whether the unit is system-defined (if the property *IsStandard* is set to true) or user-defined (if the property *IsStandard* is set to false).

The method `pfcUnit::GetReferenceUnit` returns a reference unit (one of the available system units) in terms of the `pfcUnit` object.

The method `pfcUnit::GetConversionFactor` identifies the relation of the unit to its reference unit in terms of the `pfcUnitConversionFactor` object. The unit conversion factors are as follows:

- `Offset`—Specifies the offset value applied to the values in the reference unit.
- `Scale`—Specifies the scale applied to the values in the reference unit to get the value in the actual unit.

```
Example - Consider the formula to convert temperature from
Centigrade
 to Fahrenheit
F = a + (C * b)
where
F is the temperature in Fahrenheit
C is the temperature in Centigrade
```

```
a = 32 (constant signifying the offset value)
b = 9/5 (ratio signifying the scale of the unit)
```

Note

Creo application scales the length dimensions of the model using the factors listed above. If the scale is modified, the model is regenerated. When you scale the model, the model units are not changed. Imported geometry cannot be scaled.

Use the methods `pfcUnitConversionFactor::GetOffset` and `pfcUnitConversionFactor::GetScale` to retrieve the unit conversion factors listed above.

The method `wfcWModel::CreateUnitByExpression` creates a derived unit. Use the method `wfcWUnit::ModifyByExpression` to modify a derived unit.

Modifying Individual Units

Methods Introduced:

- **pfcUnit::Modify**
- **pfcUnit::Delete**
- **pfcUnit::SetName**
- **pfcUnitConversionFactor::SetOffset**
- **pfcUnitConversionFactor::SetScale**

The method `pfcUnit::Modify` modifies the definition of a unit by applying a new conversion factor specified by the `pfcUnitConversionFactor` object and a reference unit.

The method `pfcUnit::Delete` deletes the unit.

Note

You can delete only custom units and not standard units.

The method `pfcUnit::SetName` modifies the name of the unit.

Use the methods `pfcUnitConversionFactor::SetOffset` and `pfcUnitConversionFactor::SetScale` to modify the unit conversion factors.

Creating a New Unit

Methods Introduced:

- **pfcSolid::CreateCustomUnit**
- **pfcUnitConversionFactor::Create**

The method `pfcSolid::CreateCustomUnit` creates a custom unit based on the specified name, the conversion factor given by the `pfcUnitConversionFactor` object, and a reference unit.

The method `pfcUnitConversionFactor::Create` creates the `pfcUnitConversionFactor` object containing the unit conversion factors.

Accessing Systems of Units

Methods Introduced:

- **pfcSolid::ListUnitSystems**
- **pfcSolid::GetPrincipalUnits**
- **pfcUnitSystem::GetUnit**
- **pfcUnitSystem::GetName**
- **pfcUnitSystem::GetType**
- **pfcUnitSystem::GetIsStandard**

The method `pfcSolid::ListUnitSystems` returns the list of unit systems available to the specified model.

The method `pfcSolid::GetPrincipalUnits` returns the system of units assigned to the specified model in the form of the `pfcUnitSystem` object.

The method `pfcUnitSystem::GetUnit` retrieves the unit of a particular type used by the unit system.

The method `pfcUnitSystem::GetName` returns the name of the unit system.

The method `pfcUnitSystem::GetType` returns the type of the unit system in the form of the `pfcUnitSystemType` object. The types of unit systems are as follows:

- `pfcUNIT_SYSTEM_MASS_LENGTH_TIME`—Specifies the Mass Length Time (MLT) unit system.
- `pfcUNIT_SYSTEM_FORCE_LENGTH_TIME`—Specifies the Force Length Time (FLT) unit system.

For more information on these unit systems listed above, refer to the section Types of Unit Systems on page 216.

The method `pfcUnitSystem::GetIsStandard` identifies whether the unit system is system-defined (if the property *IsStandard* is set to true) or user-defined (if the property *IsStandard* is set to false).

Modifying Systems of Units

Methods Introduced:

- **pfcUnitSystem::Delete**
- **pfcUnitSystem::SetName**

The method `pfcUnitSystem::Delete` deletes a custom-defined system of units.

Note

You can delete only a custom-defined system of units and not a standard system of units.

Use the method `pfcUnitSystem::SetName` to rename a custom-defined system of units. Specify the new name for the system of units as an input parameter for this function.

Creating a New System of Units

Method Introduced:

- **pfcSolid::CreateUnitSystem**

The method `pfcSolid::CreateUnitSystem` creates a new system of units in the model based on the specified name, the type of unit system given by the `pfcUnitSystemType` object, and the types of units specified by the `pfcUnits` sequence to use for each of the base measurement types (length, force or mass, and temperature).

Conversion to a New Unit System

Methods Introduced:

- **wfcWUnit::CalculateUnitConversion**
- **pfcSolid::SetPrincipalUnits**
- **pfcUnitConversionOptions::Create**
- **pfcUnitConversionOptions::SetDimensionOption**
- **pfcUnitConversionOptions::SetIgnoreParamUnits**

The method `wfcWUnit::CalculateUnitConversion` calculate the conversion factor between two units. These units can belong to the same model or two different models.

The method `pfcSolid::SetPrincipalUnits` changes the principal system of units assigned to the solid model based on the the unit conversion options specified by the `pfcUnitConversionOptions` object. The method `pfcUnitConversionOptions::Create` creates the `pfcUnitConversionOptions` object containing the unit conversion options listed below.

The types of unit conversion options are as follows:

- `DimensionOption`—Use the option while converting the dimensions of the model.

  Use the method `pfcUnitConversionOptions::SetDimensionOption` to modify this option.

  This option can be of the following types:

  - `pfcUNITCONVERT_SAME_DIMS`—Specifies that unit conversion occurs by interpreting the unit value in the new unit system. For example, 1 inch will equal to 1 millimeter.
  - `pfcUNITCONVERT_SAME_SIZE`—Specifies that unit conversion will occur by converting the unit value in the new unit system. For example, 1 inch will equal to 25.4 millimeters.
- `IgnoreParamUnits`—This boolean attribute determines whether or not ignore the parameter units. If it is null or true, parameter values and units do not change when the unit system is changed. If it is false, parameter units are converted according to the rule.

  Use the method `pfcUnitConversionOptions::SetIgnoreParamUnits` to modify this attribute.

Mass Properties

Method Introduced:

- **pfcSolid::GetMassProperty**
- **pfcSolid::GetMassPropertyWithDensity**
- **pfcAssembly::GetMassPropertyByCompPath**

The function `pfcSolid::GetMassProperty` provides information about the distribution of mass in the part or assembly. It can provide the information relative to a coordinate system datum, which you name, or the default one if you provide `null` as the name. It returns a class called `MassProperty`.

The class contains the following fields:

- The volume.

- The surface area.
- The density. The density value is 1.0, unless a material has been assigned.
- The mass.
- The center of gravity (COG).
- The inertia matrix.
- The inertia tensor.
- The inertia about the COG.
- The principal moments of inertia (the eigen values of the COG inertia).
- The principal axes (the eigenvectors of the COG inertia).

The method `pfcSolid::GetMassPropertyWithDensity` calculates the mass properties of a part or an assembly in the specified coordinate system. The input arguments follow:

- *CoordSysName*—Name of the coordinate system. If this is `Null`, the method uses the default coordinate system.
- *DensityOpt*—Value of the density flag specified using the enumerated data type `pfcMPDensityUse` and the valid values are as follows:
  - `pfcMP_DENSITY_DEFAULT`—Calculate the mass properties using the material density.
  - `pfcMP_DENSITY_USE_ALWAYS`—Calculate the mass properties using the specified density, even if material has a defined density.
  - `pfcMP_DENSITY_USE_IF_MISSING`—Calculate mass properties using specified density, even if material does not have a defined density.
- *density*—Density used while calculating mass properties depending on the value specified for the input argument `DensityOpt`.

The method `pfcAssembly::GetMassPropertyByCompPath` calculates the mass properties of a solid that is referenced by the specified coordinate system selection, using the respective component paths. The input arguments follow:

- *CompPath*—Component path of the solid. If this is null, the top assembly is referred.
- *CsysItem*—Coordinate system model item. If this is null, default coordinate system is referred.
- *CsysPath*—Component path of the coordinate system. If this is null, default coordinate system is referred.

Part Properties

The methods described in this section get information about the part.

Methods Introduced:

- **wfcWPart::GetDensity**
- **wfcWPart::SetDensity**

The method `wfcWPart::GetDensity` returns the density of the specified part. Use the method `wfcWPart::SetDensity` to set the density of the part without assigning a material. If a material has been defined for the part, then density is set for the material.

Annotations

Methods Introduced:

- **pfcNote::GetLines**
- **pfcNote::SetLines**
- **pfcNote::GetText**
- **pfcNote::GetURL**
- **pfcNote::SetURL**
- **pfcNote::Display**
- **pfcNote::Delete**
- **pfcNote::GetOwner**
- **wfcWSolid::GetDefaultTextHeight**

3D model notes are instance of ModelItem objects. They can be located and accessed using methods that locate model items in solid models, and downcast to the Note interface to use the methods in this section.

The method `pfcNote::GetLines` returns the text contained in the 3D model note. The method `pfcNote::Note.SetLines` modifies the note text.

The method `pfcNote::GetText` returns the the text of the solid model note. If you set the parameter *GiveParametersAsNames* to `TRUE`, then the text displays the parameter callouts with ampersands (&). If you set the parameter to `FALSE`, then the text displays the parameter values with no callout information.

The method `pfcNote::GetURL` returns the URL stored in the 3D model note. The method `pfcNote::Note.SetURL` modifies the note URL.

The method `pfcNote::Display` forces the display of the model note.

The method `pfcNote::Delete` deletes a model note.

The method `pfcNote::GetOwner` returns the solid model owner of the note.

The method `wfcWSolid::GetDefaultTextHeight` returns the default height of the annotations and dimensions in the specified solid model.

Materials

Creo Object TOOLKIT C++ enables you to programmatically access the material types and properties of parts. Using the methods and properties described in the following sections, you can perform the following actions:

- Create or delete materials
- Set the current material
- Access and modify the material types and properties

Methods Introduced:

- **pfcMaterial::Save**
- **pfcMaterial::Delete**
- **pfcPart::GetCurrentMaterial**
- **pfcPart::SetCurrentMaterial**
- **pfcPart::ListMaterials**
- **pfcPart::CreateMaterial**
- **pfcPart::RetrieveMaterial**
- **wfcWSolidBody::GetMaterial**
- **wfcWSolidBody::SetMaterial**

The method `pfcMaterial::Save` writes to a material file that can be imported into any Creo part.

The method `pfcMaterial::Delete` removes material from the part.

The method `pfcPart::GetCurrentMaterial` returns the master material assigned for the part.

The method `pfcPart::SetCurrentMaterial` sets the master material assigned to the part.

Note

- By default, while assigning a master material to a sheetmetal part, the method `pfcPart::SetCurrentMaterial` modifies the values of the sheetmetal properties such as Y factor and bend table according to the material file definition. This modification triggers a regeneration and a modification of the developed length parameters of the sheetmetal part. To prevent this regeneration, set the value of the configuration option `material_update_smt_bend_table` to `never_replace`. To trigger a regeneration and a modification of the developed length parameters of the sheetmetal part, set the configuration option `material_update_smt_bend_table` to `always_replace`. The default value is `always_replace`.
- The method `pfcPart::SetCurrentMaterial` may change the model display, if the new material has a default appearance assigned to it.
- The method may also change the family table, if the parameter `PTC_MASTER_MATERIAL` is a part of the family table.

Note

You can still use the legacy parameter `PTC_MASTER_MATERIAL`, however, these legacy parameters do not appear correctly in calculations and reports when you are working with a part that uses multiple materials.

The method `pfcPart::ListMaterials` returns a list of the materials available in the part.

The method `pfcPart::CreateMaterial` creates a new empty material in the specified part.

The method `pfcPart::RetrieveMaterial` imports a material file into the part. The name of the file read can be as either:

- `<name>.mtl`—Specifies the new material file format.

If the material is not already in the part database, `pfcPart::RetrieveMaterial` adds the material to the database after reading the material file. If the material is already in the database, the function replaces the material properties in the database with those contained in the material file.

Use the method `wfcWSolidBody::GetMaterial` to retrieve the information of the material assigned to the body.

The method `wfcWSolidBody::SetMaterial` assigns the material to the specified body.

Note

Refer to the Creo Parametric online help for more information about Materials.

Accessing Material Types

Methods Introduced:

- **pfcMaterial::GetStructuralMaterialType**
- **pfcMaterial::SetStructuralMaterialType**
- **pfcMaterial::GetThermalMaterialType**
- **pfcMaterial::SetThermalMaterialType**
- **pfcMaterial::GetSubType**
- **pfcMaterial::SetSubType**
- **pfcMaterial::GetPermittedSubTypes**
- **pfcMaterial::GetFluidMaterialType**
- **pfcMaterial::SetFluidMaterialType**

The method `pfcMaterial::GetStructuralMaterialType` returns the material type for the structural properties of the material. The material types are as follows:

- `pfcMTL_ISOTROPIC`—Specifies a a material with an infinite number of planes of material symmetry, making the properties equal in all directions.
- `pfcMTL_ORTHOTROPIC`—Specifies a material with symmetry relative to three mutually perpendicular planes.
- `pfcMTL_TRANSVERSELY_ISOTROPIC`—Specifies a material with rotational symmetry about an axis. The properties are equal for all directions in the plane of isotropy.
- `pfcMTL_FLUID`—Specifies a material with fluid properties.

Use the method `pfcMaterial::SetStructuralMaterialType` to set the material type for the structural properties of the material.

The method `pfcMaterial::GetThermalMaterialType` returns the material type for the thermal properties of the material. The material types are as follows:

- `pfcMTL_ISOTROPIC`—Specifies a material with an infinite number of planes of material symmetry, making the properties equal in all directions.

- pfcMTL_ORTHOTROPIC—Specifies a material with symmetry relative to three mutually perpendicular planes.
- pfcMTL_TRANSVERSELY_ISOTROPIC—Specifies a material with rotational symmetry about an axis. The properties are equal for all directions in the plane of isotropy.
- pfcMTL_FLUID—Specifies a material with fluid properties.

Use the method pfcMaterial::SetThermalMaterialType to set the material type for the thermal properties of the material.

The method pfcMaterial::GetSubType returns the subtype for the MTL_ISOTROPIC material type.

Use the method pfcMaterial::SetSubType to set the subtype for the MTL_ISOTROPIC material type.

Use the method pfcMaterial::GetPermittedSubTypes to retrieve a list of the permitted string values for the material subtype.

The method pfcMaterial::GetFluidMaterialType returns the material type for the fluid properties of the material. The material types are as follows:

- pfcMTL_ISOTROPIC—Specifies a material with an infinite number of planes of material symmetry, making the properties equal in all directions.
- pfcMTL_ORTHOTROPIC—Specifies a material with symmetry relative to three mutually perpendicular planes.
- pfcMTL_TRANSVERSELY_ISOTROPIC—Specifies a material with rotational symmetry about an axis. The properties are equal for all directions in the plane of isotropy.
- pfcMTL_FLUID—Specifies a material with fluid properties.

Use the method pfcMaterial::SetFluidMaterialType to set the material type for the fluid properties of the material.

Accessing Material Properties

The methods listed in this section enable you to access material properties.

Methods Introduced:

- **pfcMaterialProperty::Create**
- **pfcMaterial::GetPropertyValue**
- **pfcMaterial::SetPropertyValue**
- **pfcMaterial::SetPropertyUnits**
- **pfcMaterial::RemoveProperty**
- **pfcMaterial::GetDescription**
- **pfcMaterial::SetDescription**

- **pfcMaterial::GetFatigueType**
- **pfcMaterial::SetFatigueType**
- **pfcMaterial::GetPermittedFatigueTypes**
- **pfcMaterial::GetFatigueMaterialType**
- **pfcMaterial::SetFatigueMaterialType**
- **pfcMaterial::GetPermittedFatigueMaterialTypes**
- **pfcMaterial::GetFatigueMaterialFinish**
- **pfcMaterial::SetFatigueMaterialFinish**
- **pfcMaterial::GetPermittedFatigueMaterialFinishes**
- **pfcMaterial::GetFailureCriterion**
- **pfcMaterial::SetFailureCriterion**
- **pfcMaterial::GetPermittedFailureCriteria**
- **pfcMaterial::GetHardness**
- **pfcMaterial::SetHardness**
- **pfcMaterial::GetHardnessType**
- **pfcMaterial::SetHardnessType**
- **pfcMaterial::GetCondition**
- **pfcMaterial::SetCondition**
- **pfcMaterial::GetBendTable**
- **pfcMaterial::SetBendTable**
- **pfcMaterial::GetCrossHatchFile**
- **pfcMaterial::SetCrossHatchFile**
- **pfcMaterial::GetMaterialModel**
- **pfcMaterial::SetMaterialModel**
- **pfcMaterial::GetPermittedMaterialModels**
- **pfcMaterial::GetModelDefByTests**
- **pfcMaterial::SetModelDefByTests**
- **wfcMaterialItem::GetDescription**
- **wfcMaterialItem::SetDescription**

The method `pfcMaterialProperty::Create` creates a new instance of a material property object.

All numerical material properties are accessed using the same set of APIs. You must provide a property type to indicate the property you want to read or modify.

The method `pfcMaterial::GetPropertyValue` returns the value and the units of the material property.

Use the method `pfcMaterial::SetPropertyValue` to set the value and units of the material property. If the property type does not exist for the material, then this method creates it.

Use the method `pfcMaterial::SetPropertyUnits` to set the units of the material property.

Use the method `pfcMaterial::RemoveProperty` to remove the material property.

Material properties that are non-numeric can be accessed via property-specific get and set methods.

The methods `pfcMaterial::GetDescription` and `pfcMaterial::SetDescription` return and set the description string for the material respectively.

The methods `pfcMaterial::GetFatigueType` and `pfcMaterial::SetFatigueType` return and set the valid fatigue type for the material respectively.

Use the method `pfcMaterial::GetPermittedFatigueTypes` to get a list of the permitted string values for the fatigue type.

The methods `pfcMaterial::GetFatigueMaterialType` and `pfcMaterial::SetFatigueMaterialType` return and set the class of material when determining the effect of the fatigue respectively.

Use the method `pfcMaterial::GetPermittedFatigueMaterialTypes` to retrieve a list of the permitted string values for the fatigue material type.

The methods `pfcMaterial::GetFatigueMaterialFinish` and `pfcMaterial::SetFatigueMaterialFinish` return and set the type of surface finish for the fatigue material respectively.

Use the method `pfcMaterial::GetPermittedFatigueMaterialFinishes` to retrieve a list of permitted string values for the fatigue material finish.

The method `pfcMaterial::GetFailureCriterion` returns the reduction factor for the failure strength of the material. This factor is used to reduce the endurance limit of the material to account for unmodeled stress concentrations, such as those found in welds. Use the method `pfcMaterial::SetFailureCriterion` to set the reduction factor for the failure strength of the material.

Use the method `pfcMaterial::GetPermittedFailureCriteria` to retrieve a list of permitted string values for the material failure criterion.

The methods `pfcMaterial::GetHardness` and `pfcMaterial::SetHardness` return and set the hardness for the specified material respectively.

The methods `pfcMaterial::GetHardnessType` and `pfcMaterial::SetHardnessType` return and set the hardness type for the specified material respectively.

The methods `pfcMaterial::GetCondition` and `pfcMaterial::GetCondition` return and set the condition for the specified material respectively.

The methods `pfcMaterial::GetBendTable` and `pfcMaterial::SetBendTable` return and set the bend table for the specified material respectively.

The methods `pfcMaterial::GetCrossHatchFile` and `pfcMaterial::SetCrossHatchFile` return and set the file containing the crosshatch pattern for the specified material respectively.

The methods `pfcMaterial::GetMaterialModel` and `pfcMaterial::SetMaterialModel` return and set the type of hyperelastic isotropic material model respectively.

Use the method `pfcMaterial::GetPermittedMaterialModels` to retrieve a list of the permitted string values for the material model.

The methods `pfcMaterial::GetModelDefByTests` determines whether the hyperelastic isotropic material model has been defined using experimental data for stress and strain.

Use the method `pfcMaterial::SetModelDefByTests` to define the hyperelastic isotropic material model using experimental data for stress and strain.

The methods `wfcMaterialItem::GetDescription` and `wfcMaterialItem::SetDescription` get and set the material description for the specified model item.

Accessing User-defined Material Properties

Materials permit assignment of user-defined parameters. These parameters allow you to place non-standard properties on a given material. Therefore `pfcMaterial` is a child of `pfcParameterOwner`, which provides access to user-defined parameters and properties of materials through the methods in that interface.

11

Solid Bodies

The methods in this chapter allow a Creo Object TOOLKIT C++ application to access, modify, and create geometric solid bodies in a model. We recommend that you study the Creo Parametric documentation on solid bodies, and develop experience with manipulating geometric solid bodies using the Creo Parametric commands before attempting to use these methods.

Solid Body Information

The state of the body is derived from the features and geometry in which the body is created.

Methods introduced:

- **pfcSolidBody::GetBodyState**
- **pfcSolidBody::IsConstruction**
- **pfcSolidBody::ListSurfaces**
- **pfcSolid::GetDefaultBody**
- **pfcSolidBody::GetFeatures**
- **pfcSolidBody::GetOutline**
- **pfcSolidBody::IsSheetmetal**
- **pfcSolidBody::GetDensity**
- **pfcSolidBody::GetMassProperty**

The method `pfcSolidBody::GetBodyState` specifies the state of the body using the enumerated type `pfcSolidBodyState`. The valid values are:

- `pfcBODY_STATE_MISSING`—
- `pfcBODY_STATE_CONSUMED`—Shows bodies that are removed by the `Remove Body` command, or the bodies that are merged, subtracted, or intersected during Boolean operations. Such bodies do not contribute in the mass properties calculations.
- `pfcBODY_STATE_NO_CONTR_FEAT`—Shows bodies that are created using the `New Body` command, or the bodies with geometry that is deleted, suppressed, or rerouted to different bodies.
- `pfcBODY_STATE_NO_GEOMETRY`—Shows bodies in which the entire geometry is removed by some features or show bodies that have contributing features but no geometry.
- `pfcBODY_STATE_ACTIVE`

The method `pfcSolidBody::IsConstruction` checks if the body is a construction body. The method returns a `True` if solid body is a construction body.

The method `pfcSolidBody::ListSurfaces` lists all the surfaces in the specified body

The method `pfcSolid::GetDefaultBody` returns the default body in the specified solid.

The method `pfcSolidBody::GetFeatures` lists the features that are associated with the specified body.

Use the method `pfcSolidBody::GetOutline` to retrieve the regeneration outline of a solid body, with respect to the base coordinate system orientation. This outline defines the boundary box of the body.

The method `pfcSolidBody::IsSheetmetal` checks if the specified body is an active sheetmetal body.

The method `pfcSolidBody::GetDensity` determines the density of the body.

Use the method `pfcSolidBody::GetMassProperty` to retrieve the mass properties of a body in the specified coordinate system.

Creating a Solid Body

The object `wfcWSolidBody` represents a solid body.

Methods Introduced:

- **wfcWSolid::CreateBody**
- **wfcWSolid::DeleteBody**
- **wfcWSolidBody::SetDefault**
- **wfcWSolidBody::SetConstruction**

The method `wfcWSolid::CreateBody` creates a new body in the specified solid.

The method `wfcWSolid::DeleteBody` deletes the specified body in the solid.

The method `wfcWSolidBody::SetDefault` sets the solid body as a default body.

The method `wfcWSolidBody::SetConstruction` sets the solid body as a construction body.

External Copy Geometry Feature

The external Copy Geometry feature copies geometry from model to model without copying the geometry in the context of the assembly. Dependency on the assembly and all models along the path between the two components is avoided. When a Copy Geometry feature is created in Part mode, it is automatically defined as external. Source and target components must be relatively positioned, but are independent of the assembly context.

When all the components are assembled, the copied geometry of the target models is coincident to the referenced geometry of the external model. Components can be deleted from the assembly without losing associativity of copied geometry,

which happens with internal Copy Geometry features when the target component model is deleted. In addition, none of the other external Copy Geometry features that reference the component will fail.

An independent external Copy Geometry feature remains frozen in its original state, whether or not its external reference model is in session and its geometry has been updated. It does not fail when copied references are missing from the parent model.

If the external reference model is an assembly, all the geometric references must be chosen from the same model. The first selection determines which model this will be. If the first reference is selected from geometry, an assembly surface, datum, and so forth of the top-level assembly, all subsequent references must be from top-level assembly geometry. If the first reference is chosen from an assembly component model, all subsequent reference selections must be made from that model.

Copy Geometry features can copy reference geometry such as

- Surface
- Datum features
- Edges
- Curves
- Bodies

They are used in a top-down design to reduce the amount of data in a session, thus avoiding the retrieval of entire reference source models.

You can define the properties of the copied references to include in the Copy Geometry feature. All properties in the list that follows are copied by default. You can then decide which properties you do not want to copy to the target.

- `Appearance`—Appearances from the source geometry and applies them to the copied geometry.
- `Parameters`—Parameters of the copied entities.
- `Names`—Names. When cleared, generic names are provided for copied entities.
- `Layers`—Layer assignment and places the copied entities on layers with the same name when they exist in the target part.
- `Materials`—Material assignment is available only for bodies. When the material does not exist in the target part, it is copied to the target part and assigned to the newly created body. When cleared, the newly created body is assigned the master material of the target part and you can explicitly assign a material to the copied body in the target.

Note

Mass properties are copied along with materials.

- `Construction Bodies`—Construction attribute is available only for bodies, whether the body is a construction body or not.

To create a multibody part from an assembly

1. Set the callback action for `Assembly To MultiBody`.
2. Use the method `pfcBaseSession::CreatePart` to create a new empty part.
3. Visit all bodies from the part in an assembly. For each part, a new body is created in the empty part.
4. Select the bodies using the global method `pfcCreateModelSelection`.
5. Copy external geometry from the part to the body.
6. Use the element tree for Body Copy Options to copy the body copy options from part to the body.
7. The part is displayed in the new window using the method `pfcModel::DisplayInNewWindow`.
8. Use the method `pfcBaseSession::GetModelWindow` to get the details of the window in which the model is displayed.

Example 1: Creating a multibody part from an assembly

The sample code in `OTKMultiBody.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_solid` illustrates how to create a multibody part from an assembly.

To create an assembly from a multibody part

1. Set the callback action for `MultiBody To Assembly`
2. Collect bodies from part or component part using the `pfcSelections` class.
3. Build an element tree for body copy option. For more information about element tree for body options, refer to the “The Element Tree for Body Options” chapter in the *Creo Parametric TOOLKIT Users’ Guide*.
4. For each body, an empty part is created and the geometry of the bodies is copied to the respective parts, through the external copy geometry feature and the body copy options.
5. Create an assembly from the parts.

6. The assembly is displayed in the new window using the method `pfcModel::DisplayInNewWindow`.
7. Use the method `pfcBaseSession::GetModelWindow` to get the details of the window in which the model is displayed.

Example 2: Creating an assembly from a multibody part

The sample code in `OTKMultiBody.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_solid` illustrates how to create an assembly from a multibody part.

To set the appearance of bodies

The appearance of a material of a body is based on information such as:

- Color
- Highlight
- Reflection
- Transparency

You can apply and clear appearances to and from a body.

1. Set the callback action for `Identify bodies & Apply Appearance` and `Restore bodies`.
2. Use the method `pfcBaseSession::GetCurrentWindow` to get the information of the current Creo Parametric window.
3. Apply different colors to all bodies in a part using the method `wfcWSelection::SetVisibleAppearance`.
4. Restore colors to all bodies in a part using the method `wfcWSelection:GetVisibleAppearance`.

Example 3: Identifying bodies or applying appearance to bodies

The sample code in `OTKMultiBody.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_solid` illustrates how to apply appearance to bodies

12

Annotations: Annotation Features and Annotations

Overview of Annotation Features

An annotation feature is composed of one or more "annotation elements". Each annotation element is composed of references, parameters and annotations (notes, symbols, geometric tolerances, surface finishes, reference dimensions, driven dimensions, and manufacturing template annotations). The annotation feature allows annotation information to have the same benefits as geometry in Creo Parametric models, that is, parameters can be assigned to these annotation elements, and missing geometric references can cause features to fail in some situations.

Annotation elements may belong to annotation features, or may also be found in data-sharing features (features such as Copy Geometry, Publish Geometry, Merge, Cutout, and Shrinkwrap features).

Creo Object TOOLKIT C++ does not expose the feature element tree for annotation features because some elements in the tree are used for non-standard purposes. Instead, Creo Object TOOLKIT C++ provides specific methods for creating, redefining, and reading the properties of annotation features and annotation elements.

Creating Annotation Features

Methods Introduced:

- **wfcWSelection::CreateAnnotationFeature**

The method `wfcSelection::CreateAnnotationFeature` creates a new annotation feature in the model as a `wfcAnnotationFeature` object. *InvokeUI* specifies a boolean flag that determines how the annotation features will be created. It can have the following values:

- `FALSE`—Indicates that the feature will be a new empty annotation feature with one general annotation element in it. Modify the new annotation element and add others using subsequent steps.
- `TRUE`—Indicates that Creo Parametricwill invoke the annotation feature creation user interface.

Redefining Annotation Features

Redefining an annotation feature involves creation of new annotation elements, deletion of elements that are not required and modification of the feature properties.

Note

The methods in this section are shortcuts to redefining the feature containing the annotation elements. Due to this, Creo Parametric must regenerate the model after making the required changes to the annotation element. The methods include a flag to optionally allow the **Fix Model** User Interface to appear upon a regeneration failure.

Methods Introduced:

- **wfcWSelection::AddAnnotationElement**
- **wfcWSelection::AddElementsInAnnotationFeature**
- **wfcWSelection::CopyAnnotationElement**
- **wfcWSelection::DeleteAnnotationElement**
- **wfcWSelection::DeleteElementsInAnnotationFeature**

The method `wfcWSelection::AddAnnotationElement` adds a new non-graphical annotation element to the feature as a `wfcAnnotationElement` object.

The method `wfcWSelection::AddElementsInAnnotationFeature` adds a series of new annotation elements to the annotation feature as a `wfcAnnotationElements` object. Each element may be created as non-graphical or may be assigned a pre-existing annotation.

Note

In case of Datum Target Annotation Features (DTAFs), you can add only one set datum tag annotation element using the method `wfcWSelection::AddElementsInAnnotationFeature`.

The method `wfcWSelection::CopyAnnotationElement` copies and adds an existing annotation element to the specified annotation feature.

The method `wfcWSelection::DeleteAnnotationElement` deletes an annotation element from the feature. The method deletes the annotation element, its annotations, parameters, references, and application data from the feature.

Note

In case of Datum Target Annotation Features (DTAFs), the method `wfcWSelection::DeleteAnnotationElement` allows you to delete only a Datum Target Annotation Element (DTAE) from a DTAF. This method does not allow deletion of a set datum tag annotation element from the DTAF.

The method `wfcWSelection::DeleteElementsInAnnotationFeature` deletes a series of annotation elements from the feature. The method deletes the annotation element, its annotations, parameters, references, and application data from the feature.

Note

In case of Datum Target Annotation Features (DTAFs), the method `wfcWSelection::DeleteElementsInAnnotationFeature` allows you to delete only a Datum Target Annotation Element (DTAE) from a DTAF. This method does not allow deletion of a set datum tag annotation element from the DTAF.

Accessing Annotations

Methods Introduced:

- **wfcAnnotation::ShowInDrawing**
- **wfcAnnotation::Display**
- **wfcAnnotation::DisplayInDrawing**
- **wfcAnnotation::Undisplay**
- **wfcAnnotation::UndisplayInDrawing**
- **wfcAnnotation::Update**
- **wfcAnnotation::IsInactive**
- **wfcAnnotation::IsUsingXSecReference**
- **wfcWDrawing::EraseAnnotation**
- **wfcWSelection::ShowAnnotations**
- **wfcWSelection::ShowAxes**
- **wfcWSelection::ShowDatumTargets**
- **wfcWView2D::ShowAnnotations**

- **wfcWView2D::ShowAxes**
- **wfcWView2D::ShowDatumTargets**
- **wfcWSelection::ShowAxes**

The method `wfcAnnotation::ShowInDrawing` shows the annotation in the current combined state. The annotation will be visible until it is explicitly erased from the combined state. The method also adds the specified annotation to the current combined state, if it has not been added. If the specified annotation has been erased, then the method changes the display status of the erased annotation and makes it visible in the combined state, that is, it unerases the annotation. This method is also capable of showing an annotation in a drawing view similar to the Creo Parametric command `Show` and `Erase`. The input arguments are:

- *DrawingView*—Specifies the drawing view.
- *CompPath*—Specifies the assembly component path.

The methods `wfcAnnotation::Display` and `wfcAnnotation::Undisplay` temporarily display and remove an annotation from the display for the specified combined state

Similarly, the methods `wfcAnnotation::DisplayInDrawing` and `wfcAnnotation::UndisplayInDrawing` temporarily display and remove an annotation from the display for the specified drawing. Specify the drawing in the input argument *DrawingModel* The undisplay and display methods for combined state and drawing should be used together. To edit a shown annotation, it must be first removed temporarily from display by using the undisplay method, followed by editing the annotation. The annotation must be redisplayed using the display method, so that the updated annotation is visible to the user.

The method `wfcAnnotation::Update` updates the display of an annotation, but does not actually display it. If the annotation is not currently displayed because it is hidden by layer status or inactive geometry, the text extracted from the annotation may include callout symbols, instead of the text shown to the user. The method `wfcAnnotation::Update` informs Creo Parametric to update the contents of the annotation in order to cross-reference these callout values. This method supports 3D model notes and detail notes.

The method `wfcAnnotation::IsInactive` indicates whether the annotation can be shown or not. An annotation becomes inactive if all the weak references it points to have been lost or consumed.

The method `wfcAnnotation::IsUsingXSecReference` identifies if the annotation is created on or attached to a cross-sectional edge.

The method `wfcWDrawing::EraseAnnotation` removes an annotation from the display for the specified drawing. The annotation is not shown in the specified drawing.

Note

The annotation which was removed from the display using the method `wfcWDrawing::EraseAnnotation` will become visible again, only if the method `wfcAnnotation::ShowInDrawing` is called to explicitly display the annotation.

The method `wfcWSelection::ShowAnnotations` displays the annotation of the specified type for the selected feature or component.

The method `wfcWSelection::ShowAxes` displays the axes for the selected feature and component.

The method `wfcWSelection::ShowDatumTargets` displays the datum targets for the selected feature and component.

Note

In case of methods `wfcWSelection::ShowAnnotations`, `wfcWSelection::ShowAxes`, and `wfcWSelection::ShowDatumTargets`, for annotations created in drawing mode and owned by a solid, which can be displayed only in the context of that drawing, specify the name of the drawing in the input argument drawing. You can specify the view in which the annotations for the selected feature and component must be displayed. If you pass the input argument *DrawingView* as `NULL`, the annotations are displayed in all the views.

The method `wfcWView2D::ShowAnnotations` displays the annotation of the specified type in the drawing view by using the enumerated data type `wfcAnnotationType`.

The method `wfcWView2D::ShowAxes` displays the axes in the drawing view.

The method `wfcWView2D::ShowDatumTargets` displays the datum targets in the drawing view.

Accessing and Modifying Annotation Elements

The following methods provide access and modify the properties of an annotation element.

Note

The methods in this section are shortcuts to redefining the feature containing the annotation elements. Because of this, Creo Parametric must regenerate the model after making the indicated changes to the annotation element. The methods include a flag to optionally allow the **Fix Model** User Interface to appear upon a regeneration failure.

Methods Introduced:

- **wfcAnnotation::GetAnnotationElement**
- **wfcAnnotation::GetAnnotationElementWithOption**
- **wfcAnnotationElement::GetAnnotationType**
- **wfcAnnotationElement::IsReadOnly**
- **wfcAnnotationElement::HasMissingReferences**
- **wfcAnnotationElement::IsIncomplete**
- **wfcAnnotationElement::GetCopyFlag**
- **wfcWSelection::SetCopyFlagInAnnotationElement**
- **wfcAnnotationElement::IsDependent**
- **wfcWSelection::SetDependencyFlag**
- **wfcAnnotationElement::GetAnnotation**
- **wfcAnnotationElement::GetAnnotationFeature**
- **wfcAnnotationElement::IsReferenceStrong**
- **wfcAnnotationElement::CollectAnnotationReferences**
- **wfcAnnotationElement::CollectQuiltReferenceSurfaces**
- **wfcAnnotationElement::GetAnnotationReferenceDescription**
- **wfcAnnotationElement::SetAnnotationReferenceDescription**
- **wfcWSelection::SetAnnotationInAnnotationElement**
- **wfcWSelection::GetAutoPropagateFlagInAnnotationElement**
- **wfcWSelection::SetAutoPropagateFlagInAnnotationElement**
- **wfcWSelection::AddAnnotationReferenceInAnnotationElement**
- **wfcWSelection::SetAnnotationReferencesInAnnotationElement**
- **wfcWSelection::RemoveAnnotationReferenceInAnnotationElement**
- **wfcWSelection::SetAnnotationElementReferences**
- **wfcAnnotationReferenceSet::Create**
- **wfcAnnotationReferenceSet::GetAutoPropagate**
- **wfcAnnotationReferenceSet::SetAutoPropagate**

- **wfcAnnotationReferenceSet::GetReference**
- **wfcAnnotationReferenceSet::SetReference**
- **wfcAnnotationReferenceSet::GetStrong**
- **wfcAnnotationReferenceSet::SetStrong**
- **wfcAnnotationReferenceSet::GetDescription**
- **wfcAnnotationReferenceSet::SetDescription**
- **wfcWSelection::StrengthenAnnotationElementReference**
- **wfcWSelection::WeakenAnnotationElementReference**
- **wfcAnnotationReferenceCurveCollectionObject::Create**
- **wfcAnnotationReferenceCurveCollectionObject::GetCurveCollection**
- **wfcAnnotationReferenceCurveCollectionObject::SetCurveCollection**
- **wfcAnnotationReferenceSelectionObject::Create**
- **wfcAnnotationReferenceSelectionObject::GetReference**
- **wfcAnnotationReferenceSelectionObject::SetReference**
- **wfcAnnotationReferenceSurfaceCollectionObject::Create**
- **wfcAnnotationReferenceSurfaceCollectionObject::GetSurfaceCollection**
- **wfcAnnotationReferenceSurfaceCollectionObject::SetSurfaceCollection**

The method `wfcAnnotation::GetAnnotationElement` retrieves the annotation element that contains the annotation as a `wfcAnnotationElement` object.

Use the method `wfcAnnotation::GetAnnotationElementWithOption` to get the annotation element or semi annotation element that contains a given annotation.

The method `wfcAnnotationElement::GetAnnotationType` returns the type of the annotation contained in the annotation element using the enumerated data type `wfcAnnotationType`. The valid values are:

- `wfcANNOTATION_NOTE`—Specifies a note.
- `wfcANNOTATION_GTOL`—Specifies a geometric tolerance.
- `wfcANNOTATION_SURFACE_FINISH`—Specifies a surface finish.
- `wfcANNOTATION_SYMBOL_INSTANCE`—Specifies a symbol.
- `wfcANNOTATION_DIMENSION`—Specifies a driven dimension.
- `wfcANNOTATION_REF_DIMENSION`—Specifies a reference dimension.
- `wfcANNOTATION_SET_DATUM_TAG`—Specifies a set datum tag.

The method `wfcAnnotationElement::IsReadOnly` checks if the annotation element has read only or full access.

The method `wfcAnnotationElement::HasMissingReferences` checks if an annotation element has missing references. Use the input parameters of this method to investigate specific types and sources of references, or to check all references simultaneously. The input arguments are:

- *ReferenceType*—Specifies the type of reference by using the enumerated data type `wfcAnnotationRefFilter`. The valid values are:
  - `wfcANNOTATION_REF_ALL`—Specifies all references.
  - `wfcANNOTATION_REF_WEAK`—Specifies weak references.
  - `wfcANNOTATION_REF_STRONG`—Specifies strong references.
- *ReferenceSource*—Specifies the source of the references by using the enumerated data type `wfcAnnotationRefFromType`. The valid values are:
  - `wfcANNOT_REF_FROM_ALL`—Specifies all references.
  - `wfcANNOT_REF_FROM_ANNOTATION`—Specifies references from annotations.
  - `wfcANNOT_REF_FROM_USER`—Specifies references from users.
- *AtLeastOneMissingRef*—Specifies if the annotation element has at least one missing reference of the specified type and source.

Use the method `wfcAnnotationElement::IsIncomplete` returns a true value if the annotation element is incomplete because of missing strong references.

The method `wfcAnnotationElement::GetCopyFlag` retrieves the copy flag of the annotation elements. This property is not supported for elements in data sharing features. It returns true if the annotation element contains copies of its references.

The method `wfcWSelection::SetCopyFlagInAnnotationElement` sets the copy flag of the annotation element. Set the input argument *InvokeUI* to true to allow the **Fix Model** User Interface to appear upon a regeneration failure. This property is not supported for annotations in data sharing features.

The method `wfcAnnotationElement::IsDependent` retrieves the value of the dependency flag for the annotation element. This property is supported only for the elements in data sharing features. It returns `true` if the annotation element is dependent on its parent.

The method `wfcWSelection::SetDependencyFlag` sets the dependency flag of the annotation element. This property is supported only for annotations in data sharing features.

The method `wfcAnnotationElement::GetAnnotation` returns the annotation contained in an annotation element.

The method `wfcAnnotationElement::GetAnnotationFeature` returns the feature that owns the annotation element as a `pfcFeature` object.

The method `wfcAnnotationElement::IsReferenceStrong` identifies if a reference is weak or strong in a given annotation element.

The method `wfcAnnotationElement::CollectAnnotationReferences` retrieves an array of references contained in the specified annotation element as a `wfcAnnotationReferences` object. The input arguments are:

- *RefStrength*—Specifies the type of reference by using the enumerated data type `wfcAnnotationRefFilter`.
- *RefSource*—Specifies the source of the references by using the enumerated data type `wfcAnnotationRefFromType`.

The method `wfcAnnotationElement::CollectQuiltReferenceSurfaces` returns the surfaces which make up a quilt surface collection reference for the annotation element as a `wfcAnnotationReferences` object.

Note

All the surfaces made inactive by features occurring after the annotation element in the model regeneration are also returned.

The method `wfcAnnotationElement::GetAnnotationReferenceDescription` retrieves the description property for a given annotation element reference.

Note

The description string is the same as that of the tooltip text for the reference name in the Annotation Feature UI.

The method `wfcAnnotationReference::GetAnnotationReferenceType` retrieves the type of the annotation reference by using the enumerated data type `wfcAnnotationRefType`. The valid values are:

- `wfcANNOT_REF_SINGLE`—Specifies a single reference.
- `wfcANNOT_REF_CRV_COLLECTION`—Specifies the collection of curves.
- `wfcANNOT_REF_SRF_COLLECTION`—Specifies the collection of surfaces.

The method `wfcAnnotationElement::SetAnnotationReferenceDescription` sets the description property for a given annotation element reference. The input arguments are:

- *Reference*—Specifies the annotation reference as a `wfcAnnotationReference` object.
- *Description*—Specifies the description for the annotation element reference.

Note

Creo Parametric must regenerate the model after making the indicated changes to the annotation element.

Use the method `wfcWSelection::SetAnnotationInAnnotationElement` to modify the annotation contained in an annotation element. Pass the input argument *Annotation* as `NULL` to modify the annotation element to be a non-graphical annotation.

Note

The above method does not support Datum Target Annotation Elements (DTAEs).

If you modify the annotation contained in the annotation element, the original annotation is automatically removed from the element and is owned by the model.

The methods `wfcWSelection::GetAutoPropagateFlagInAnnotationElement` and `wfcWSelection::SetAutoPropagateFlagInAnnotationElement` get and set the autopropagate flag of the specified annotation element reference.

The method `wfcWSelection::AddAnnotationReferenceInAnnotationElement` adds a strong user-defined reference to the annotation element.

The method `wfcWSelection::SetAnnotationReferencesInAnnotationElement` replaces all the user-defined references in the annotation element with references that are specified in the input argument *Reference*.

The method `wfcWSelection::RemoveAnnotationReferenceInAnnotationElement` removes the user-defined reference from the annotation element.

The method `wfcWSelection::SetAnnotationElementReferences` sets user-defined annotation references in the specified annotation elements. The input arguments follow:

- *Elements*—Annotation Elements specified using the class `wfcAnnotationElements`.
- *References*—User-defined references specified using the class `wfcAnnotationReferenceSetsArray`.
- *InvokeUI*—Specify the Boolean value to indicate whether the UI is invoked.

Use the method `wfcAnnotationReferenceSet::Create` to create an instance of the `wfcAnnotation.AnnotationReferenceSet` object. The input arguments are as follows:

- *Reference*
- *Strong*
- *Description*
- *AutoPropagate*

Use the methods `wfcAnnotationReferenceSet::GetReference` and `wfcAnnotationReferenceSet::SetReference` to get and set the user-defined annotation references using the `wfcAnnotationReference` object.

Use the methods `wfcAnnotationReferenceSet::GetStrong` and `wfcAnnotationReferenceSet::SetStrong` to get and set the strength of the user-defined annotation references.

Use the methods `wfcAnnotationReferenceSet::GetDescription` and `wfcAnnotationReferenceSet::SetDescription` to get and set the description of the user-defined annotation references.

Use the methods `wfcAnnotationReferenceSet::GetAutoPropagate` and `wfcAnnotationReferenceSet::SetAutoPropagate` to get and set the auto propagate flag for the user-defined annotation features.

The method `wfcWSelection::StrengthenAnnotationElementReference` converts a weak reference to a strong reference.

The method `wfcWSelection::WeakenAnnotationElementReference` converts a strong reference to a weak reference.

The method `wfcAnnotationReferenceCurveCollectionObject::Create` creates a curve collection object for the specified user-defined annotation references.

The methods `wfcAnnotationReferenceCurveCollectionObject::GetCurveCollection` and `wfcAnnotationReferenceCurveCollectionObject::SetCurveCollection` retrieves and sets the curve collections for the user-defined set of annotation references using the `wfcCollection` object.

The method `wfcAnnotationReferenceSelectionObject::Create` creates a selection object for the specified user-defined annotation references.

The methods `wfcAnnotationReferenceSelectionObject::GetReference` and `wfcAnnotationReferenceSelectionObject::SetReference` retrieves and sets the references for the user-defined annotation references using the `pfcSelection` object.

The method `wfcAnnotationReferenceSurfaceCollectionObject::Create` creates a surface collection object for the specified user-defined annotation references.

The methods `wfcAnnotationReferenceSurfaceCollectionObject::GetSurfaceCollection` and `wfcAnnotationReferenceSurfaceCollectionObject::SetSurfaceCollection` retrieves and sets the surface collections for the user-defined annotation references using the `wfcCollection` object.

Accessing Reference and Driven Dimensions

Methods Introduced:

- **wfcWSolid::CreateDimension**

The method `wfcWSolid::CreateDimension` creates a new driven dimension. Specify the geometric references and parameters required to construct the required dimension as the input parameters for this method. Once the reference dimension is created, use the method `wfcAnotation::ShowInDrawing` to display it. The input arguments of this method are as follows:

- *AnnotPlane*—Specifies the annotation plane for the dimensions.
- *DimAttachs*—Specifies the points on the model where you want to attach the dimension.

Note

The attachments structure is an array of two `pfcSelection` entities. It is provided to support options such as intersect where two entities must be passed as input. From Creo Parametric 3.0 onward, you can create dimensions that have intersection type of reference. The intersection type of reference is a reference that is derived from the intersection of two entities. Refer to the Creo Parametric Detailed Drawings Help for more information on intersection type of reference.

- *Senses*—Specifies more information about how the dimension attaches to each attachment point of the model, that is, to what part of the entity.
- *OrientHint*—Specifies the orientation of the dimension and has one of the following values:
  - `pfcORIENTATION_HINT_HORIZONTAL`—Specifies a horizontal dimension.
  - `pfcORIENTATION_HINT_VERTICAL`—Specifies a vertical dimension.
  - `pfcORIENTATION_HINT_SLANTED`—Specifies the shortest distance between two attachment points (available only when the dimension is attached to points).
  - `pfcORIENTATION_HINT_ELLIPSE_RADIUS1`—Specifies the start radius for a dimension on an ellipse.
  - `pfcORIENTATION_HINT_ELLIPSE_RADIUS2`—Specifies the end radius for a dimension on an ellipse.
  - `pfcORIENTATION_HINT_ARC_ANGLE`—Specifies the angle of the arc for a dimension of an arc.
  - `pfcORIENTATION_HINT_ARC_LENGTH`—Specifies the length of the arc for a dimension of an arc.
  - `pfcORIENTATION_HINT_LINE_TO_TANGENT_CURVE_ANGLE`—Specifies the value to dimension the angle between the line and the tangent at a curve point (the point on the curve must be an endpoint).
  - `pfcORIENTATION_HINT_RADIAL_DIFF`—Specifies the linear dimension of the radial distance between two concentric arcs or circles.
- *Location*—Specifies the initial location of the dimension text.

Automatic Propagation of Annotation Elements

Automatic local propagation of annotation elements can be done based on some specified conditions, using a Creo TOOLKIT application.

When an appropriate event occurs during a Creo Parametric session, the associated notification method can invoke a local propagation.

Methods Introduced:

- **wfcWSession::AutoPropagate**

The method `wfcWSession::AutoPropagate` causes the local and automatic propagation of annotation elements to the currently selected feature within the same model, after a supported feature has either been created or modified. The propagation behavior is dependant on the standard Creo Parametric algorithm and on the current contents of the selection buffer.

Following are the list of supported features:

- Draft
  - Surface
  - Solid
- Offset
  - Surface
  - With Draft
  - Expand
- Mirror Surface
- Copy Surface
- Move Surface

The Creo TOOLKIT application can be written so as to specify the condition for the automatic propagation, based on created feature-type, subtype or any other required properties.

The method propagates based on the current contents of the selection buffer.

Note

The method `wfcWSession::AutoPropagate` works regardless of the current value for the configuration option, `auto_propagate_ae`. PTC advises that the Creo TOOLKIT application respect the current value of this configuration option; otherwise, duplicate versions of the propagated annotations can result.

Detail Tree

Methods introduced:

- **wfcWSolid::CollapseDetailTree**
- **wfcWSolid::ExpandDetailTree**
- **wfcWSolid::RefreshDetailTree**

Use the method `wfcWSolid::CollapseDetailTree` to collapse all nodes of the detail tree in the Creo Parametric window and make its child nodes invisible.

Use the method `wfcWSolid::ExpandDetailTree` to expand the detail tree in the Creo Parametricwindow.

Use the method `wfcWSolid::RefreshDetailTree` to rebuild the detail tree in the Creo Parametric window that contains the model.

The input argument for the above three methods is *SolidWindow*. It is the window containing the solid. By default, the detail tree in the active window is used.

Converting Annotations to Latest Version

Methods Introduced:

- **wfcAnnotation::NeedsConversion**
- **wfcAnnotation::ConvertLegacy**

The method `wfcAnnotation::NeedsConversion` returns true in the following cases:

- Annotations created in releases earlier than Creo Parametric 4.0 F000
- Annotations created using the deprecated methods `ProGtolCreate()` or `ProSetdatumtagCreate()`

The method `wfcAnnotation::ConvertLegacy` converts annotations to the latest Creo Parametric version.

You can call the method `wfcAnnotation::ConvertLegacy` only if the method `wfcAnnotation::NeedsConversion` returns true.

Annotation Text Styles

Methods Introduced:

- **wfcAnnotation::GetTextStyle**
- **wfcAnnotation::SetTextStyle**
- **wfcAnnotation::GetTextStyleInDrawing**
- **wfcAnnotation::SetTextStyleInDrawing**

- **pfcAnnotationTextStyle::Create**
- **pfcAnnotationTextStyle::GetAngle**
- **pfcAnnotationTextStyle::SetAngle**
- **pfcAnnotationTextStyle::GetColor**
- **pfcAnnotationTextStyle::SetColor**
- **pfcAnnotationTextStyle::GetFont**
- **pfcAnnotationTextStyle::SetFont**
- **pfcAnnotationTextStyle::GetHeight**
- **pfcAnnotationTextStyle::SetHeight**
- **pfcAnnotationTextStyle::GetWidth**
- **pfcAnnotationTextStyle::SetWidth**
- **pfcAnnotationTextStyle::GetThickness**
- **pfcAnnotationTextStyle::SetThickness**
- **pfcAnnotationTextStyle::GetHorizontalJustification**
- **pfcAnnotationTextStyle::SetHorizontalJustification**
- **pfcAnnotationTextStyle::GetVerticalJustification**
- **pfcAnnotationTextStyle::SetVerticalJustification**
- **pfcAnnotationTextStyle::GetSlantAngle**
- **pfcAnnotationTextStyle::SetSlantAngle**
- **pfcAnnotationTextStyle::IsHeightInModelUnits**
- **pfcAnnotationTextStyle::SetHeightInModelUnits**
- **pfcAnnotationTextStyle::IsTextMirrored**
- **pfcAnnotationTextStyle::MirrorText**
- **pfcAnnotationTextStyle::IsTextUnderlined**
- **pfcAnnotationTextStyle::UnderlineText**

The methods `wfcAnnotation::GetTextStyle` and `wfcAnnotation::SetTextStyle` retrieve and set the text style for the specified annotation as a `pfcAnnotationTextStyle` object.

The method `wfcAnnotation::GetTextStyleInDrawing` retrieves the text style for the specified annotation in a drawing as a `pfcAnnotationTextStyle` object. The input arguments are:

- *Drawing*—Specifies a drawing only when the annotation is owned by the solid, but is displayed in the drawing.
- *CompPath*—Specifies the component path to the solid that owns the annotation as a `pfcComponentPath` object.
- *View*—This is reserved for future use. Pass `NULL`.

The method `wfcAnnotation::SetTextStyleInDrawing` sets the text style for the specified annotation in a drawing.

The method `pfcAnnotationTextStyle::Create` creates a data object that contains information about text style for a specified annotation.

The methods `pfcAnnotationTextStyle::GetAngle` and `pfcAnnotationTextStyle::SetAngle` get and set the angle of the text style.

The methods `pfcAnnotationTextStyle::GetColor` and `pfcAnnotationTextStyle::SetColor` retrieve and set the color of the text style as a `pfcColorRGB` object.

The methods `pfcAnnotationTextStyle::GetFont` and `pfcAnnotationTextStyle::SetFont` get and set the font of the text.

The methods `pfcAnnotationTextStyle::GetHeight` and `pfcAnnotationTextStyle::SetHeight` get and set the height of the text style.

The methods `pfcAnnotationTextStyle::GetWidth` and `pfcAnnotationTextStyle::SetWidth` retrieve and set the width of the text style.

The methods `pfcAnnotationTextStyle::GetThickness` and `pfcAnnotationTextStyle::SetThickness` get and set the thickness of the text style.

The methods `pfcAnnotationTextStyle::GetHorizontalJustification` and `pfcAnnotationTextStyle::SetHorizontalJustification` get and set the horizontal justification of the text style using the enumerated data type `pfcHorizontalJustification`. The valid values are:

- `pfcH_JUSTIFY_LEFT`—Aligns the text style object to the left.
- `pfcH_JUSTIFY_CENTER`—Aligns the text style object in the center.
- `pfcH_JUSTIFY_RIGHT`—Aligns the text style object to the right.
- `pfcH_JUSTIFY_DEFAULT`—Aligns the text using the default justification. The justification selected for the first note becomes the default for all successive notes added during the current session.

The methods `pfcAnnotationTextStyle::GetVerticalJustification` and `pfcAnnotationTextStyle::SetVerticalJustification` get and set the vertical justification of the text style using the enumerated data type `pfcVerticalJustification`. The valid values are:

- `pfcV_JUSTIFY_TOP`—Aligns the text style object to the top.
- `pfcV_JUSTIFY_MIDDLE`—Aligns the text style object to the middle.
- `pfcV_JUSTIFY_BOTTOM`—Aligns the text style object to the bottom.

- `pfcV_JUSTIFY_DEFAULT`—Aligns the text using the default justification. The justification selected for the first note becomes the default for all successive notes added during the current session.

The methods `pfcAnnotationTextStyle::GetSlantAngle` and `pfcAnnotationTextStyle::SetSlantAngle` get and set the slant angle of the text style.

The method `pfcAnnotationTextStyle::IsHeightInModelUnits` checks if the text height is in relation to the model units or as a fraction of the screen size. This method is applicable for flat-to-screen annotations only.

The methods `pfcAnnotationTextStyle::IsTextMirrored` checks if the text style is mirrored.

The methods `pfcAnnotationTextStyle::MirrorText` sets the mirroring of the text style. specifies the option to mirror the text style. Specify the input argument *Mirror* to `true` to mirror the text style.

The methods `pfcAnnotationTextStyle::IsTextUnderlined` checks if the text style is underlined.

The methods `pfcAnnotationTextStyle::UnderlineText`specifies the option to underline the text style.

Annotation Orientation

An annotation orientation refers to the annotation plane or the parallel plane in which the annotation lies, the viewing direction, and the text rotation. You can define the annotation orientation using a datum plane or flat surface, a named view, or as flat to screen. If the orientation is defined by a datum plane, you can set its reference to `frozen` or `driven`; where frozen indicates that the reference to the datum plane or named view has been removed.

Methods Introduced:

- **wfcAnnotation::Rotate**
- **wfcWSelection::CreateAnnotationPlane**
- **wfcWSolid::CreateAnnotationPlaneFromView**
- **wfcWSolid::CreateFlatToScreenPlane**
- **wfcAnnotationPlane::GetReference**
- **wfcAnnotationPlane::GetPlaneData**
- **wfcAnnotationPlane::GetFlip**
- **wfcAnnotationPlane::GetNormalVector**
- **wfcAnnotationPlane::GetViewName**
- **wfcAnnotationPlane::GetPlaneType**
- **wfcAnnotationPlane::IsFrozen**

- **wfcAnnotationPlane::SetFrozen**
- **wfcAnnotationPlane::IsForceToPlane**
- **wfcAnnotationPlane::SetForceToPlane**
- **wfcAnnotationPlane::GetPlaneAngle**
- **wfcAnnotationPlane::GetPlaneOrientation**
- **wfcAnnotationPlane::GetNames**
- **wfcAnnotationPlane::AssignName**
- **wfcWModel::GetAnnotationPlanesFromGallery**
- **wfcWModel::GetAnnotationPlanes**
- **wfcWModel::GetAnnotationPlaneByName**
- **wfcWModel::AddAnnotationPlaneToGallery**
- **wfcWModel::RemoveAnnotationPlaneFromGallery**
- **wfcWModel::GetActiveAnnotationPlane**

The method `wfcAnnotation::Rotate` rotates a given annotation by the specified angle. This moves the annotation to a new annotation plane with the appropriate rotation assigned. Other annotations on the annotation's current plane are unaffected by this method.

Note

You can only rotate annotations that belong to annotation elements using the above method.

The method `wfcWSelection::CreateAnnotationPlane` creates a new annotation plane from either a datum plane, a flat surface, or an existing annotation that already contains an annotation plane .

The method `wfcWSolid::CreateAnnotationPlaneFromView` creates a new annotation plane from a saved model view .

The method `wfcWSolid::CreateFlatToScreenPlane` returns an annotation plane representing a flat-to-screen annotation in the model

The method `wfcAnnotationPlane::GetReference` returns the planar surface used as the annotation plane .

The method `wfcAnnotationPlane::GetPlaneData` returns the geometry of the annotation plane in terms of the origin and orientation of the annotation plane as a `wfcWPlaneData` object.

The method `wfcAnnotationPlane::GetFlip` specifies if the annotation plane was flipped during creation.

The method `wfcAnnotationPlane::GetNormalVector` returns the normal vector that determines the viewing direction of the annotation plane .

The method `wfcAnnotationPlane::GetViewName` obtains the name of the view that was originally used to determine the orientation of the annotation plane.

Note

If the named view orientation has been changed after the annotation plane was created, the orientation of the plane will not match the current orientation of the view.

Use the method `wfcAnnotationPlane::GetPlaneType` to obtain the annotation plane type. The valid values are defined in the enumerated data type `wfcAnnotationPlaneType`:

- `wfcANNOTATION_PLANE_REFERENCE`—The annotation plane is created from a datum plane or a flat surface, and can be frozen or be associative to the reference.
- `wfcANNOTATION_PLANE_NAMED_VIEW`—The annotation plane is created from a named view or a view in the drawing.
- `wfcANNOTATION_PLANE_FLATTOSCREEN_BY_MODELPNT`—The annotation plane is flat-to-screen and annotations are located by model units.
- `wfcANNOTATION_PLANE_FLATTOSCREEN_BY_SCREENPNT`—The annotation plane is flat-to-screen and annotations are located by screen points.
- `wfcANNOTATION_PLANE_FLATTOSCREEN_LEGACY`—The annotation uses a legacy flat-to-screen format located in model space.

The methods `wfcAnnotationPlane::IsFrozen` and `wfcAnnotationPlane::SetFrozen` determine and assign, respectively, whether the annotation orientation is frozen or driven by reference to the plane geometry. These methods are applicable only for annotation planes governed by references.

The methods `wfcAnnotationPlane::IsForceToPlane` and `wfcAnnotationPlane::SetForceToPlane` check and assign, respectively, the boolean value of the argument *ForceToPlane* for an annotation plane. If this argument is set to true, then the annotations that reference the annotation plane are placed on that plane. If the annotation orientation is not frozen, that is, driven by the reference plane, and if the reference plane is moved, then the annotations also move along with the plane.

The method `wfcAnnotationPlane::GetPlaneAngle` returns the current rotation angle in degrees for a given annotation plane.

The method `wfcAnnotationPlane::GetPlaneOrientation` returns the text orientation of all annotations on that plane.

The method `wfcAnnotationPlane::GetNames` returns the names of the annotation plane.

The method `wfcAnnotationPlane::AssignName` assigns a name to the specified annotation plane.

The method `wfcWModel::GetAnnotationPlanesFromGallery` collects the names of all the annotation planes in the gallery.

The method `wfcWModel::GetAnnotationPlanes` collects the names of all the named annotation planes in the specified model.

The method `wfcWModel::GetAnnotationPlaneByName` finds and returns the annotation plane with the specified name.

Use the method `wfcWModel::AddAnnotationPlaneToGallery` to add an annotation plane with the specified name to the gallery.

Use the method `wfcWModel::RemoveAnnotationPlaneFromGallery` to remove the annotation plane with the specified name from the gallery.

The method `wfcWModel::GetActiveAnnotationPlane` returns the active annotation plane in the specified model.

Accessing Baseline and Ordinate Dimensions

The methods described in this section enable you to create 3D ordinate driven dimensions in 3D models as model annotations or as annotation elements. They also provide the ability to define a baseline annotation element, and then define model ordinate dimension annotations and ordinate dimension annotation elements that reference the baseline annotation element.

Baseline Dimensions

Methods Introduced:

- **wfcWDimension::IsBaseline**
- **wfcWDimension::GetBaselineDimension**
- **wfcWSelection::CreateAnnotationFeatBaseline**

The method `wfcWDimension::IsBaseline` identifies whether a dimension is a baseline dimension.

The method `wfcWDimension::GetBaselineDimension` obtains the baseline dimension in a drawing.

The method `wfcSelection::CreateAnnotationFeatBaseline` creates an ordinate baseline annotation element and corresponding dimension as a `wfcAnnotationElement` object. The input arguments are:

- *Reference*—Specifies the reference geometry for the ordinate baseline dimension as a `pfcSelection` object.
- *Plane*—Specifies the annotation plane for the baseline dimension and all its related dimensions as a `wfcAnnotationPlane`. Refer to the section Annotation Orientation on page 255 for more information on the `wfcAnnotationPlane` object.
- *DirReference*—Specifies the direction of the dimension witness lines.

Ordinate Dimensions

Methods Introduced:

- **wfcWDimension::IsOrdinate**
- **wfcWDimension::GetOrdinateStandard**
- **wfcWDimension::SetOrdinateStandard**
- **wfcWDimension::SetOrdinateReferences**
- **wfcWDrawing::CreateAutoOrdinateDimensions**
- **wfcWDimension::OrdinateDimensionToLinear**
- **wfcWDimension::LinearDimensionToOrdinate**

The method `wfcWDimension::IsOrdinate` identifies if a dimension is ordinate.

The method `wfcWDimension::GetOrdinateStandard` returns the display standard for the ordinate dimensions in the drawing using the enumerated type `wfcDimOrdinateStandard`. The valid values for display standards are as follows:

- `wfcDIM_ORDSTD_DEFAULT`—Specifies the default style for the ordinate dimensions.
- `wfcDIM_ORDSTD_ANSI`—Specifies the American National Standard style for the ordinate dimension. It places the related ordinate dimensions without a connecting line.
- `wfcDIM_ORDSTD_JIS`—Specifies the Japanese Industrial Standard style for the ordinate dimension. It places the ordinate dimensions along a connecting line that is perpendicular to the baseline and starts with an open circle.
- `wfcDIM_ORDSTD_ISO`—Specifies the International Standard of Organization style for the ordinate dimension.
- `wfcDIM_ORDSTD_DIN`—Specifies the German Institute for Standardization style for the ordinate dimension.

- `wfcDIM_ORDSTD_SAME_AS_3D`—Specifies the ordinate dimension style for 2D drawings. Not used in 3D ordinate dimensions.

The method `wfcWDimension::SetOrdinateStandard` sets the style for the specified ordinate dimension or a set of ordinate dimensions.

Use the method `wfcWDimension::SetOrdinateReferences` to set the dimension attachments and dimension senses.

The method `wfcWDrawing::CreateAutoOrdinateDimensions` creates ordinate dimensions automatically for the selected surfaces. The input arguments are:

- *Surfaces*—Specifies a set of parallel surfaces for which the ordinate dimensions must be created as a `pfcSelections` object.
- *Baseline*—Specifies a reference element used to create the baseline dimension as a `pfcSelection` object. The reference element can be an edge, a curve, or a datum plane.

The method `wfcWDimension::OrdinateDimensionToLinear` converts an existing ordinate dimension to a linear dimension in a solid.

The method `wfcWDimension::LinearDimensionToOrdinate` converts a linear dimension to an ordinate dimension.

Note

The dimension must be first created as an ordinate dimension and then converted to a linear dimension. This method does not work on dimensions which were first created as linear dimensions.

Annotation Associativity

The methods described in this section allow you to ensure associativity between shown annotations in drawings and 3D models. You can independently control the position associativity and attachment associativity of a drawing annotation.

Note

- Drawing annotations can have only uni-directional associativity, that is, changes to the position and attachment of the annotation in the 3D model are reflected for the annotation in the drawing view, but not vice-versa.
- You cannot modify the position associativity and attachment associativity of a drawing annotation from the 3D model.
- You cannot make free, flat-to-screen annotations in a drawing view associative to the annotations in the 3D model.
- Annotation properties such as text, jogs, breaks, skew, witness line clippings, and flip arrow states are not associative.

Methods Introduced:

- **wfcAnnotation::IsAssociative**
- **wfcAnnotation::GetAttachmentAssociativity**
- **wfcAnnotation::UpdatePosition**
- **wfcAnnotation::UpdateAttachment**

The method `wfcAnnotation::IsAssociative` checks if a given annotation in a drawing view is associative to the annotation in the 3D model.

The method `wfcAnnotation::GetAttachmentAssociativity` retrieves the associativity of the attachment using the enumerated data type `wfcAnnotationAttachmentAssociativity`. The valid values are as follows:

- `wfcANNOTATTACH_ASSOCIATIVITY_NONE`—Specifies that the drawing annotation is not associative.
- `wfcANNOTATTACH_ASSOCIATIVITY_PARTIAL`—Specifies that the drawing annotation is partially associative.
- `wfcANNOTATTACH_ASSOCIATIVITY_FULL`— Specifies that the drawing annotation is fully associative.

The method `wfcAnnotation::UpdatePosition` updates the position of the drawing annotation, and makes it associative to the position of the annotation in the 3D model. Pass the input argument *Drawing* of the method as a `pfcDrawing` object.

Note

You can update the associative position only for drawing annotations that have their placement based on model coordinates.

The method `wfcAnnotation::UpdateAttachment` updates the attachment of the drawing annotation, and makes it associative to the attachment of the annotation in the 3D model. Associative attachment of an annotation refers to both – its references and its attachment point on its references.

Note

You can update the associative attachment only for drawing annotations that are attached to the geometry.

Annotation Security

The methods described in this section allow you to manage whether an annotation is designated as a security marking.

Methods Introduced:

- **wfcAnnotation::SetSecurityMarking**
- **wfcAnnotation::GetSecurityMarking**

Use the method `wfcAnnotation::SetSecurityMarking` to set the security marking option for notes and symbols.

Use the method `wfcAnnotation::GetSecurityMarking` to retrieve the security marking option for notes and symbols.

Accessing Set Datum Tags

The methods described in this section provide the ability to access and display set datum tag annotations in 3D models.

Methods Introduced:

- **wfcSetDatumTag::GetAttachment**
- **wfcSetDatumTag::SetAttachment**
- **wfcSetDatumTag::GetAnnotationPlane**
- **wfcSetDatumTag::SetAnnotationPlane**

- **wfcSetDatumTag::Show**
- **wfcSetDatumTag::Delete**
- **wfcSetDatumTag::GetAdditionalText**
- **wfcSetDatumTag::SetAdditionalText**
- **wfcSetDatumTagAdditionalText::GetPosition**
- **wfcSetDatumTagAdditionalText::SetPosition**
- **wfcSetDatumTagAdditionalText::GetText**
- **wfcSetDatumTagAdditionalText::SetText**
- **wfcSetDatumTag::GetAdditionalTextLocation**
- **wfcSetDatumTag::GetTextLocation**
- **wfcSetDatumTag::IsShownInDrawingView**
- **wfcSetDatumTag::EraseFromDrawingView**
- **wfcSetDatumTag::GetDimGTolDisplay**
- **wfcSetDatumTag::SetLabel**
- **wfcSetDatumTag::GetLabel**
- **wfcSetDatumTag::SetASMEDisplay**
- **wfcSetDatumTag::GetASMEDisplay**
- **wfcSetDatumTag::SetElbow**
- **wfcSetDatumTag::GetElbow**
- **wfcSetDatumTag::GetPlacement**
- **wfcSetDatumTag::DeleteReference**
- **wfcSetDatumTag::GetReferences**
- **wfcSetDatumTag::AddReferences**
- **wfcWModel::CreateSetDatumTag**
- **wfcWDrawing::ListSetDatumTags**

The methods `wfcSetDatumTag::GetAttachment` and `wfcSetDatumTag::SetAttachment` get and set the item on which the datum tag is placed .

Note

To specify the datum plane or datum axis as the attachment option, pass the input argument *Reference* as `NULL` in the method `wfcSetDatumTag::SetAttachment`. The datum tag is attached to the datum axis at the default location.

The methods `wfcSetDatumTag::GetAnnotationPlane` and `wfcSetDatumTag::SetAnnotationPlane` get and set the annotation plane for the specified set datum tag.

Once the set datum tag annotation is created, use the method `wfcSetDatumTag::Show` to display it.

Use the method `wfcSetDatumTag::Delete` to delete the specified datum feature tag annotation.

The methods `wfcSetDatumTag::GetAdditionalText` and `wfcSetDatumTag::SetAdditionalText` get and set the additional text for the specified datum feature tag.

The methods `wfcSetDatumTagAdditionalText::GetPosition` and `wfcSetDatumTagAdditionalText::SetPosition` to get and set the position of the additional text around the frame of the datum feature tag. You can set the position of the additional text using the enumerated data type `wfcSetDatumTagAdditionalTextPos` and the valid values are:

- `wfcSDT_ADDL_TEXT_RIGHT`
- `wfcSDT_ADDL_TEXT_BOTTOM`
- `wfcSDT_ADDL_TEXT_LEFT`
- `wfcSDT_ADDL_TEXT_TOP`
- `wfcSDT_ADDL_TEXT_DEFAULT`

The methods `wfcSetDatumTagAdditionalText::GetText` and `wfcSetDatumTagAdditionalText::SetText` to get and set the additional text around the frame of the datum feature tag.

The method `wfcSetDatumTag::GetAdditionalTextLocation` retrieves the location of additional text for the specified datum feature tag.

The method `wfcSetDatumTag::GetTextLocation` retrieves the text point for the specified datum feature tag.

The method `wfcSetDatumTag::IsShownInDrawingView` returns the display status of the set datum tag in the specified view of a drawing.

Use the method `wfcSetDatumTag::EraseFromDrawingView` to set a set datum tag to be erased from the specified view of a drawing. The annotation is not displayed until it is explicitly displayed using the method `wfcAnnotation::ShowInDrawing`.

The method `wfcSetDatumTag::GetDimGTolDisplay` returns display type for datum feature symbol attached to dimension or gtol.

Use the methods `wfcSetDatumTag::SetLabel` and `wfcSetDatumTag::GetLabel` to get and set the label for the specified datum feature tag annotation.

The method `wfcSetDatumTag::SetASMEDisplay` displays the datum feature tag annotation according to the ASME standard.

Use the method `wfcSetDatumTag::GetASMEDisplay` to check if the specified datum feature tag annotation is displayed as per ASME standard.

The method `wfcSetDatumTag::SetElbow` sets or unset Elbow to datum feature tag annotation.

Use the method `wfcSetDatumTag::GetElbow` to check if a specified datum feature tag annotation Elbow is set.

The method `wfcSetDatumTag::GetPlacement` returns the item type, id, and owner on which the set datum tag is placed. Use this method in cases where it is not possible to construct the selection. For example, when the solid owned datum feature tag is attached to a solid owned dimension that is created in a drawing.

Use the method `wfcSetDatumTag::DeleteReference` to delete semantic references in the specified datum feature tag. The references are specified by their index number.

Use the method `wfcSetDatumTag::GetReferences` to retrieve semantic references in the specified datum feature tag.

The method `wfcSetDatumTag::AddReferences` adds semantic references to the specified datum feature tag.

The method `wfcWModel::CreateSetDatumTag` create a new set datum tag annotation. The input arguments follow:

- *Selection*—
- *Label*—The label for the datum feature tag.
- *Plane*—Plane for the annotation.

The method `wfcWDrawing::ListSetDatumTags` enables you to visit the set datum tag annotations in the specified drawing.

Designating Dimensions and Symbols

Methods Introduced:

- **wfcWModel::DesignateSymbol**
- **wfcWModel::IsDesignatedSymbol**
- **wfcWModel::UndesignateSymbol**

The method `wfcWModel::DesignateSymbol` designates a dimension, dimension tolerance, geometric tolerance or surface finish symbol to the specified model.

The method `wfcWModel::IsDesignatedSymbol` determines if a dimension, dimension tolerance, geometric tolerance or surface finish symbol has been designated to a model.

The method `wfcWModel::UndesignateSymbol` undesignates the dimension, dimension tolerance, geometric tolerance or surface finish symbol from the specified model.

Surface Finish Annotations

The methods described in this section provide read access to the properties of the surface finish object. They also allow you to create and modify surface finishes.

The style of surface finishes for releases previous to Pro/ENGINEER Wildfire 2.0 was a flat-to-screen symbol attached to a single surface. From Pro/ENGINEER Wildfire 2.0 onwards, the method for construction of surface finishes has been modified. The new style of surface finish is a symbol instance that may be attached on a surface or with a leader. The following methods support both the old and new surface finish annotations, except where specified.

Methods Introduced:

- **wfcSurfaceFinish::GetValue**
- **wfcSurfaceFinish::SetValue**
- **wfcSurfaceFinish::GetReferences**
- **wfcSurfaceFinish::GetSurfaceCollection**
- **wfcSurfaceFinish::SetSurfaceCollection**
- **wfcSurfaceFinish::Modify**
- **wfcSurfaceFinish::GetSymbolInstructions**
- **wfcSurfaceFinish::Delete**
- **wfcSurfaceFinish::Show**
- **wfcWModel::CreateSurfaceFinish**

The methods `wfcSurfaceFinish::GetValue` and `wfcSurfaceFinish::SetValue` get and set the value of a surface finish annotation.

The method `wfcSurfaceFinish::GetReferences` returns the surface or surfaces referenced by the surface finish .

The method `wfcSurfaceFinish::GetSurfaceCollection` obtains a surface collection which contains the references of the surface finish .

The method `wfcSurfaceFinish::SetSurfaceCollection` assigns a surface collection to be the references of the surface finish. This overwrites all current surface finish references. The following types of surface collections are supported:

- One by one surface set
- Intent surface set
- Excluded surface set
- Seed and Boundary surface set
- Loop surface set
- Solid surface set
- Quilt surface set

The method `wfcSurfaceFinish::Modify` modifies the symbol instance data for the specified surface finish. This method supports new symbol-based surface finishes only.

The method `wfcSurfaceFinish::GetSymbolInstructions` returns the symbol instance data for the surface finish.

The method `wfcSurfaceFinish::Delete` deletes the specified surface finish.

Once the surface finish annotation is created, use the method `wfcSurfaceFinish::Show` to display it.

The method `wfcWModel::CreateSurfaceFinish` creates a new symbol-based surface finish annotation. The method requires a symbol instance data structure for creation. Once the surface finish annotation is created, use the method `wfcAnnotation::ShowInDrawing` to display it.

Symbol Annotations

The methods described in this section provide support for 3D mode symbols. Creo Object TOOLKIT C++ methods for symbol instances are used in both 2D and 3D modes. Symbols for a particular mode must conform to the requirements for that mode.

Creating, Reading and Modifying 3D Symbols

Methods Introduced:

- **wfcWDetailSymbolInstItem::GetAnnotationPlane**
- **wfcWDetailSymbolInstItem::SetAnnotationPlane**

The methods `wfcWDetailSymbolInstItem::GetAnnotationPlane` and `wfcWDetailSymbolInstItem::SetAnnotationPlane` retrieve and set the annotation plane for 3D symbol data . Annotation planes are required for 3D symbol instances but are not applicable for 2D symbol instances.

Locating and Collecting 3D Symbols and Symbol Definitions

Methods Introduced:

- **wfcWDetailSymbolDefItem::VisitNotes**
- **wfcWDetailSymbolDefItem::VisitEntities**
- **wfcWSolid::RetrieveSymbolDefItem**
- **wfcWSolid::ListDetailItems**

The method `wfcWDetailSymbolDefItem::VisitNotes` visits the notes contained in a symbol definition stored in a solid model or a drawing.

The method `wfcWDetailSymbolDefItem::VisitEntities` visits the entities contained in a symbol definition stored in a solid model.

The method `wfcWSolid::RetrieveSymbolDefItem` allows retrieval of a symbol definition into a given solid model. The input arguments for this method are as follows:

- *FileName*— The name of the symbol definition file.
- *Source* — The location to search for the symbol definition file.
- *FilePath*— The path to the file with a symbol definition. If not given, then the symbol definition is located in the designated directory.
- *Version*— The version of the symbol definition file. Pass -1 to get the latest version.
- *UpdateUnconditionally*— Update flag.
  - `xtrue` - Update the existing symbol definition unconditionally.
  - `xfalse`- Do not load new definition if the same symbol exist in the drawing.

The method `wfcWSolid::ListDetailItems` collects and returns a sequence of all the symbol instances used in the specified solid.

Notes

The methods in this section enable you to access the notes created in .Creo

Note

These methods are applicable to solids (parts and assemblies) only.

Note Properties

Methods Introduced:

- **wfcWDetailNoteItem::GetGTol**
- **wfcWDetailNoteItem::GetElbowDirection**
- **wfcWDetailNoteItem::GetLeaderStyle**
- **wfcWDetailNoteItem::SetLeaderStyle**
- **wfcWDetailNoteItem::Get3DLineEnvelope**
- **wfcWDetailNoteItem::GetLegacyLeaderNoteLength**
- **wfcWDetailNoteItem::GetLegacyLeaderNoteDirection**

The method `wfcWDetailNoteItem::GetGTol` returns the detail note that represents a shown geometric tolerance.

The method `wfcWDetailNoteItem::GetElbowDirection` retrieves the direction of elbow in a note in model coordinate system . For flat-to-screen notes, the method retrieves the elbow direction in screen coordinate system.

The methods `wfcWDetailNoteItem::GetLeaderStyle` and `wfcWDetailNoteItem::SetLeaderStyle` retrieve and set the leader style used for the note. The valid values of the leader style are specified by the enumerated data type `wfcLeaderStyle`:

- `wfcLEADER_STYLE_STANDARD`—Specifies that the leader style used for the note is standard.
- `wfcLEADER_STYLE_ISO`—Specifies that the leader style used for the note is ISO.

The method `wfcWDetailNoteItem::Get3DLineEnvelope` retrieves the envelope of a line for a specified note .

The method `wfcWDetailNoteItem::GetLegacyLeaderNoteLength` retrieves the length of a leader line in a note.

The method `wfcWDetailNoteItem::GetLegacyLeaderNoteDirection` retrieves the direction of the leader line in a note .

Note

Creo Parametric adds hard line breaks to the multiple lines drawing notes created in Creo Elements/Pro during retrieval. The hard line breaks display the text of the note on separate lines in the **Note Properties** dialog box as they actually appear in the drawing note.

Accessing Note Placement

The methods described in this section provide access to the properties of a 3D note.

Methods Introduced:

- **wfcWDetailNoteItem::GetLeaderArrowTypes**
- **wfcWDetailNoteItem::GetAnnotationPlane**

The method `wfcWDetailNoteItem::GetLeaderArrowTypes` retrieves the type of arrowhead used for leaders attached to the note .

The method `wfcWDetailNoteItem::GetAnnotationPlane` retrieves the annotation plane assigned to the note attachment data .

Modifying 3D Note Attachments

The actual note created in Creo Parametric will not be modified until the note attachment is assigned to the note.

Methods Introduced:

- **wfcWDetailNoteItem::AddLeader**
- **wfcWDetailNoteItem::AddLeaderWithArrowType**
- **pfcDetailLeaderAttachment::GetLeaderAttachment**
- **pfcDetailLeaderAttachment::SetLeaderAttachment**
- **pfcDetailLeaderAttachment::GetType**
- **wfcWDetailNoteItem::SetLeadersWithArrowType**
- **wfcWDetailNoteItem::RemoveLeader**
- **wfcWDetailNoteItem::SetAnnotationPlane**
- **wfcWSolid::CreateOnItemNote**
- **wfcWSolid::CreateFreeNote**

The method `wfcWDetailNoteItem::AddLeader` adds a new leader to the end of the array of current leaders on a note. .

Use the method `wfcWDetailNoteItem::AddLeaderWithArrowType` to add a new leader to the end of the array of current leaders on a note and specify the type of arrowhead that is to be used for the attached leader. The method takes as input a `pfcDetailLeaderAttachment` object and the type of arrowhead to be used for the leader.

The methods `pfcDetailLeaderAttachment::GetLeaderAttachment` and `pfcDetailLeaderAttachment::SetLeaderAttachment` retrieve and set the leader attachment as a `pfcAttachment` object.

The methods `pfcDetailLeaderAttachment::GetType` and `pfcDetailLeaderAttachment::SetType`retrieves and sets the type of attachment using the enumerated data type `pfcDetailLeaderAttachmentType`. It determines the precise attachment point for the note leader.

The valid values are:

- `pfcLEADER_ATTACH_NONE`—Specifies regular leader attachment.
- `pfcLEADER_ATTACH_NORMAL`—Specifies a normal attachment.
- `pfcLEADER_ATTACH_TANGENT`—Specifies a tangent attachment.

The method `wfcWDetailNoteItem::SetLeadersWithArrowType` sets a new leader to the end of the array of current leaders on a note and specifies the type of arrowhead that is to be used for the attached leader. The input arguments are:

The method `wfcWDetailNoteItem::RemoveLeader` removes a leader from the note .

The method `wfcWDetailNoteItem::SetAnnotationPlane` sets the annotation plane for the notes .

The method `wfcWSolid::CreateOnItemNote` sets the location of an "On Item" note placement. Using this method removes any leaders currently assigned to the note attachment.

The method `wfcWSolid::CreateFreeNote` sets the location of the note text. The input arguments to this method are *TextLines* and *Attach*. The note text is stored in relative model coordinates, where {0.5, 0.5, 0.5} indicates the exact center of the model's display bounding box obtained from `wfcWSolid::GetDisplayOutline`, and {0.0, 0.0, 0.0} and {1.0, 1.0, 1.0} represent the corners of the box.

Text Style Properties

Methods Introduced:

- **pfcTextStyle::GetAngle**
- **pfcTextStyle::SetAngle**
- **pfcTextStyle::GetFontName**
- **pfcTextStyle::SetFontName**
- **pfcTextStyle::GetHeight**
- **pfcTextStyle::SetHeight**
- **pfcTextStyle::GetIsMirrored**
- **pfcTextStyle::SetIsMirrored**
- **pfcTextStyle::GetSlantAngle**

- **pfcTextStyle::SetSlantAngle**
- **pfcTextStyle::GetThickness**
- **pfcTextStyle::SetThickness**
- **pfcTextStyle::GetWidthFactor**
- **pfcTextStyle::SetWidthFactor**
- **pfcTextStyle::GetIsUnderlined**
- **pfcTextStyle::SetIsUnderlined**

The method `pfcTextStyle::GetAngle` and `pfcTextStyle::SetAngle` get and set the angle of rotation for the text style object. If you do not call `pfcTextStyle::SetAngle` when creating the text style, the rotation defaults to 0.0.

The methods `pfcTextStyle::GetFontName` and `pfcTextStyle::SetFontName` get and set the font used to display the text style object.

The methods `pfcTextStyle::GetHeight` and `pfcTextStyle::SetHeight` get and set the height of the text style object. The value `-1.0` means that the text has the default height for text currently specified for the drawing.

The methods `pfcTextStyle::GetIsMirrored` and `pfcTextStyle::SetIsMirrored` get and set the mirroring option specified for the text style object.

The methods `pfcTextStyle::GetSlantAngle` and `pfcTextStyle::SetSlantAngle` get and set the slant angle of the text style object.

The methods `pfcTextStyle::GetThickness` and `pfcTextStyle::SetThickness` get and set the line thickness of the text style object. The value `-1.0` means that the text has the default thickness for text currently specified for the drawing.

The methods `pfcTextStyle::GetWidthFactor` and `pfcTextStyle::SetWidthFactor` get and set the width factor of the text style object. The width factor is the ratio of the width of each character to the height. The value `-1.0` means that the width factor has the default value for text currently specified for the drawing.

The methods `pfcTextStyle::GetIsUnderlined` and `pfcTextStyle::SetIsUnderlined` get and set the underline option for the text style object.

13

Annotations: Geometric Tolerances

The methods in this chapter allow a Creo Object TOOLKIT C++ application to read, modify, and create geometric tolerances (gtols) in a solid or drawing. We recommend that you study the Creo Parametric documentation on geometric tolerances, and develop experience with manipulating geometric tolerances using the Creo Parametric commands before attempting to use these methods.

Reading Geometric Tolerances

Methods Introduced:

- **wfcGTol::GetGTolName**
- **wfcGTol::GetTopModel**
- **wfcGTol::GetCompositeSharedReference**
- **wfcGTol::GetComposite**
- **wfcGTol::GetDatumReferences**
- **wfcGTol::GetIndicators**
- **wfcGTol::GetReferences**
- **wfcGTol::GetGTolType**
- **wfcGTol::IsBoundaryDisplay**
- **wfcGTol::GetUnilateralModifier**
- **wfcGTol::GetValueString**
- **wfcGTol::IsAllAround**
- **wfcGTol::IsAllOver**
- **wfcGTol::IsAddlTextBoxed**

The method `wfcGTol::GetGTolName` retrieves the name of the geometric tolerance (gtol).

The method `wfcGTol::GetTopModel` retrieves the top model that owns the specified gtol. This will usually be the model that contains the gtol; but if the gtol was created in drawing mode and added to a solid in a drawing view, the owner will be the drawing, while the model is the solid.

The method `wfcGTol::GetComposite` retrieves the value and the datum references, that is, the primary, secondary, and tertiary references for the specified composite gtol.

The method `wfcGTol::GetCompositeSharedReference` checks if the datum references are shared between all the rows defined in the composite gtol.

The method `wfcGTol::GetDatumReferences` retrieves the primary, secondary, and tertiary datum references for a gtol.

The method `wfcGTol::GetIndicators` retrieves all the indicators assigned to the specified gtol.

The method `wfcGTol::GetReferences` retrieves a sequence of geometric entities referenced by the specified gtol. The entities are additional references used to create the gtol. This method supports only gtols that are not created as Annotation Elements.

The method `wfcGTol::GetGTolType` retrieves the type of gtol using the enumerated data type `wfcGTolType`. The various types of gtol, are straightness, flatness, and so on.

The method `wfcGTol::IsBoundaryDisplay` checks if the boundary modifier has been set for the specified gtol.

Use the method `wfcGTol::GetUnilateralModifier` to check if the profile boundary has been set to unilateral in the specified gtol. If set to unilateral, the method also checks if the tolerance disposition is in the outward direction of the profile.

The method `wfcGTol::GetValueString` retrieves the value specified in the gtol as a `wchar_t*` string.

The method `wfcGTol::IsAllAround` checks if the **All Around** symbol has been set for the specified gtol.

The method `wfcGTol::IsAllOver` checks if the **All Over** symbol has been set for the specified gtol. The **All Over** symbol and **All Around** symbol specifies that the profile tolerance must be applied to all the three dimensional profile of the part.

The method `wfcGTol::IsAddlTextBoxed` checks if a box has been created around the specified additional text in a geometric tolerance.

Deleting a Geometric Tolerance

Methods Introduced:

- **wfcGTol::Delete**

The method `wfcGTol::Delete` permanently removes a gtol.

Validating a Geometric Tolerance

Methods Introduced:

- **wfcGTol::Validate**

The method `wfcGTol::Validate` checks if the specified geometric tolerance is syntactically correct. For example, when a string is specified instead of a number for a tolerance value, it is considered as syntactically incorrect. The input argument *Type* is specified using the enumerated type `wfcGTolValidityCheckType`.

Geometric Tolerance Layout

The methods described in this section provide access to the layout for the text and symbols in a geometric tolerance.

Methods Introduced:

- **wfcGTol::GetElbow**
- **wfcGTol::GetRightTextEnvelope**
- **wfcGTol::Get3DLineEnvelope**

The method `wfcGTol::GetElbow` retrieves the length and direction of the geometric tolerance leader elbow.

The method `wfcGTol::GetRightTextEnvelope` retrieves the bounding box coordinates for the right text in a specified geometric tolerance.

The method `wfcGTol::Get3DLineEnvelope` retrieves the bounding box coordinates for one line from the geometric tolerance. The input argument *LineNumber* is the line number for which the bounding box coordinates are to be returned.

Note

The above methods support only gtols that are placed on an annotation plane.

Additional Text for Geometric Tolerances

You can place multi-line additional text to the right, left, bottom, and above a geometric tolerance control frame while creating and editing a gtol in both drawing and model modes.

Methods Introduced:

- **wfcGTol::GetBottomText**
- **wfcGTol::SetBottomText**
- **wfcGTol::GetLeftText**
- **wfcGTol::SetLeftText**
- **wfcGTol::GetRightText**
- **wfcGTol::SetRightText**
- **wfcGTol::GetTopText**
- **wfcGTol::SetTopText**

The method `wfcGTol::GetBottomText` retrieves the text added to the bottom of the specified geometric tolerance.

Use the method `wfcGTol::SetBottomText` to set the text to be added to the bottom of the specified geometric tolerance.

The method `wfcGTol::GetLeftText` retrieves the text added to the left of the specified geometric tolerance.

Use the method `wfcGTol::SetLeftText` to set the text to be added to the left of the specified geometric tolerance.

The method `wfcGTol::GetRightText` retrieves the text added to the right of the specified geometric tolerance.

Use the method `wfcGTol::SetRightText` to set the text to be added to the right of the specified geometric tolerance.

The method `wfcGTol::GetTopText` retrieves the text added to the top of the specified geometric tolerance.

Use the method `wfcGTol::SetTopText` to set the text to be added to the top of the specified geometric tolerance.

Geometric Tolerance Text Style

The methods described in this section access the text style properties of a geometric tolerance.

Methods Introduced:

- **wfcGTol::GetBottomTextHorizJustification**
- **wfcGTol::SetBottomTextHorizJustification**
- **wfcGTol::GetTopTextHorizJustification**
- **wfcGTol::SetTopTextHorizJustification**
- **wfcGTol::GetAdditionaltextTextStyle**
- **wfcGTol::SetGTolAdditionaltextTextStyle**
- **wfcGtol::GetAdditionalTextLocation**

The method `wfcGTol::GetBottomTextHorizJustification` retrieves the horizontal justification for the additional text applied to the specified geometric tolerance at the bottom.

The method `wfcGTol::SetBottomTextHorizJustification` sets the horizontal justification for the additional text applied to the specified geometric tolerance at the bottom. The input argument *BottomTextHorizJustification* is specified using the enumerated type `pfcHorizontalJustification` and the valid values are:

- `pfcH_JUSTIFY_LEFT`—Specifies the left horizontal
- `pfcH_JUSTIFY_CENTER`
- `pfcH_JUSTIFY_RIGHT`
- `pfcH_JUSTIFY_DEFAULT`

The method `wfcGTol::GetTopTextHorizJustification` retrieves the horizontal justification for the additional text applied to the specified geometric tolerance at the top.

The method `wfcGTol::SetTopTextHorizJustification` sets the horizontal justification for the additional text applied to the specified geometric tolerance at the top. The input argument *TopTextHorizJustification* is specified using the enumerated type `pfcHorizontalJustification`.

The method `wfcGTol::GetAdditionaltextTextStyle` retrieves the text style of the additional text applied to the specified geometric tolerance.

The method `wfcGTol::SetGTolAdditionaltextTextStyle` assigns the text style of the additional text applied to the specified geometric tolerance. The input parameters are:

- *Type*—Specifies the text type of the additional text using the enumerated type `wfcGTolTextType`.
- *TextStyle*—Specify the text style using the class `pfcAnnotationTextStyle`.

Use the method `wfcGtol::GetAdditionalTextLocation` to get the additional text location for the specified type of text. The input parameter *Type* is specified by the enumerated data type and the valid values are:

- `wfcGTOLTEXT_ON_RIGHT`
- `wfcGTOLTEXT_ON_TOP`
- `wfcGTOLTEXT_ON_BOTTOM`
- `wfcGTOLTEXT_ON_LEFT`

Creating a Geometric Tolerance

Methods Introduced:

- **wfcGTol::SetGTolType**
- **wfcGTol::SetBoundaryDisplay**
- **wfcGTol::SetUnilateralModifier**
- **wfcGTolUnilateralModifier::Create**
- **wfcGTolUnilateralModifier::GetOutside**
- **wfcGTolUnilateralModifier::SetOutside**
- **wfcGTolUnilateralModifier::GetUnilateral**
- **wfcGTolUnilateralModifier::SetUnilateral**
- **wfcGTol::SetValueString**
- **wfcGTol::AddReferences**
- **wfcGTol::DeleteReference**
- **wfcGTol::SetIndicators**
- **wfcGTolIndicator::Create**
- **wfcGTolIndicator::GetDFS**

- **wfcGTolIndicator::SetDFS**
- **wfcGTolIndicator::GetSymbol**
- **wfcGTolIndicator::SetSymbol**
- **wfcGTolIndicator::GetType**
- **wfcGTolIndicator::SetType**
- **wfcGTol::SetElbow**
- **wfcGTolElbow::Create**
- **wfcGTolElbow::GetDirection**
- **wfcGTolElbow::SetDirection**
- **wfcGTolElbow::GetLength**
- **wfcGTolElbow::SetLength**
- **wfcGTol::SetDatumReferences**
- **wfcGTolDatumReferences::Create**
- **wfcGTolDatumReferences::GetPrimary**
- **wfcGTolDatumReferences::SetPrimary**
- **wfcGTolDatumReferences::GetSecondary**
- **wfcGTolDatumReferences::SetSecondary**
- **wfcGTolDatumReferences::GetTertiary**
- **wfcGTolDatumReferences::SetTertiary**
- **wfcGTol::SetCompositeSharedReference**
- **wfcGTol::SetComposite**
- **wfcGTolComposite::Create**
- **wfcGTolComposite::GetPrimary**
- **wfcGTolComposite::SetPrimary**
- **wfcGTolComposite::GetSecondary**
- **wfcGTolComposite::SetSecondary**
- **wfcGTolComposite::GetTertiary**
- **wfcGTolComposite::SetTertiary**
- **wfcGTolComposite::GetValue**
- **wfcGTolComposite::SetValue**
- **wfcGTol::SetAllAround**
- **wfcGTol::SetAllOver**
- **wfcGTol::SetAddlTextBoxed**
- **wfcGTol::GetNotes**

The method `wfcGTol::SetGTolType` sets the type of geometric tolerance using the enumerated data type `wfcGTolType`.

The method `wfcGTol::SetBoundaryDisplay` sets the boundary modifier for the specified gtol.

Use the method `wfcGTol::SetUnilateralModifier` to set the profile boundary as unilateral in the specified gtol using the class `wfcGTolUnilateralModifier`.

The method `wfcGTolUnilateralModifier::Create` creates a unilateral modifier for a specified gtol.

Use the methods `wfcGTolUnilateralModifier::GetOutside` and `wfcGTolUnilateralModifier::SetOutside` to get and set the tolerance disposition to outward direction for the specified gtol.

Use the methods `wfcGTolUnilateralModifier::GetUnilateral` and `wfcGTolUnilateralModifier::SetUnilateral` to get and set the boundary modifier as unilateral for the specified gtol.

The method `wfcGTol::SetValueString` sets the specified value string for a gtol.

Use the method `wfcGTol::AddReferences` to add datum references to the specified gtol.

Use the method `wfcGTol::DeleteReference` to delete the datum references from the specified gtol.

The method `wfcGTol::SetIndicators` sets the indicators for the specified gtol. Specify the input argument *indicators* using the class `wfcGTolIndicators`.

Use the method `wfcGTolIndicator::Create` to create an indicator for a specified gtol. The input arguments are:

- *Type*—Specify a sequence of indicator types.
- *Symbol*—Specifies a sequence of strings for indicator symbols.
- *Df*—Specifies a sequence of strings for datum feature symbols.

Use the methods `wfcGTolIndicator::GetDFS` and `wfcGTolIndicator::SetDFS` to get and set the strings for datum feature symbol.

Use the methods `wfcGTolIndicator::GetSymbol` and `wfcGTolIndicator::SetSymbol` to get and set the strings for indicator symbols.

Use the methods `wfcGTolIndicator::GetType` and `wfcGTolIndicator::SetType` to get and set the type of indicators using the enumerated type `wfcGTolIndicatorType` and the valid values are:

- `wfcGTOL_INDICATOR_DIRECTION_FEAT`
- `wfcGTOL_INDICATOR_COLLECTION_PLANE`

- wfcGTOL_INDICATOR_INTERSECTION_PLANE
- wfcGTOL_INDICATOR_ORIENTATION_PLANE

The method wfcGTol::SetElbow sets the elbow along with its properties for a leader type of gtol. This method is supported for leader type gtols which are placed on the annotation plane. The input argument *elbowLength* specifies the length of the elbow in model coordinates.

Use the method wfcGTolElbow::Create to create an elbow for a specified gtol. The input arguments are:

- *Length*—Specifies the length of the elbow.
- *Direction*—Specifies the direction of the elbow.

The methods wfcGTolElbow::GetDirection and wfcGTolElbow::SetDirection get and set the direction of the elbow in a specified gtol.

The methods wfcGTolElbow::GetLength and wfcGTolElbow::SetLength get and set the length of the elbow in a specified gtol.

Use the method wfcGTol::SetDatumReferences to set the datum references for the specified gtol using the class wfcGTolDatumReferences. The datum references are set as wchar_t* strings.

Use the method wfcGTolDatumReferences::Create to create a datum reference for a specified gtol.

Use the methods wfcGTolDatumReferences::GetPrimary and wfcGTolDatumReferences::SetPrimary to get and set the primary datum references of the gtol.

Use the methods wfcGTolDatumReferences::GetSecondary and wfcGTolDatumReferences::SetSecondary to get and set the secondary datum references of the gtol.

Use the methods wfcGTolDatumReferences::GetTertiary and wfcGTolDatumReferences::SetTertiary to get and set the tertiary datum references of the gtol.

Use the method wfcGTol::SetCompositeSharedReference to specify if datum references in a composite gtol must be shared between all the defined rows.

The method wfcGTol::SetComposite sets the value and datum references for the specified composite gtol using the class wfcGTolComposite.

Use the method wfcGTolComposite::Create to create a composite tolerance for a specified gtol.

Use the methods `wfcGTolComposite::GetPrimary` and `wfcGTolComposite::SetPrimary` to get and set the primary datum strings of the composite tolerance.

Use the methods `wfcGTolComposite::GetSecondary` and `wfcGTolComposite::SetSecondary` to get and set the secondary datum strings of the composite tolerance.

Use the methods `wfcGTolComposite::GetTertiary` and `wfcGTolComposite::SetTertiary` to get and set the tertiary datum strings of the composite tolerance.

Use the methods `wfcGTolComposite::GetValue` and `wfcGTolComposite::SetValue` to get and set the value datum strings of the composite tolerance.

The method `wfcGTol::SetAllAround` sets the **All Around** symbol for the specified geometric tolerance.

The method `wfcGTol::SetAllOver` sets the **All Over** symbol for the specified geometric tolerance.

The method `wfcGTol::SetAddlTextBoxed` creates a box around the specified additional text in a geometric tolerance. Boxes can be created around additional text added above and below the frame of the geometric tolerance. The input arguments are:

- *Type*—Specifies the instance of additional text to access using the enumerated type `wfcGTolTextType`.
- *IsBoxed*—Specifies if the additional text is boxed.

The method `wfcGTol::GetNotes` returns the detail notes that represent a geometric tolerance in the specified drawing. The method returns `pfcDetailNoteItems` array of drawing notes that represents the geometric tolerance.

Attaching the Geometric Tolerances

Methods Introduced:

- **wfcGTol::GetGTolAttach**
- **wfcGTolAttach::GetType**
- **wfcGTol::SetAttachFree**
- **wfcGTolAttachFree::Create**
- **wfcGTolAttachFree::GetLocation**
- **wfcGTolAttachFree::SetLocation**
- **wfcGTolAttachFree::GetAnnotationPlane**
- **wfcGTolAttachFree::SetAnnotationPlane**

- **wfcGTol::SetAttachLeader**
- **wfcGTolAttachLeader::Create**
- **wfcGTolAttachLeader::GetLocation**
- **wfcGTolAttachLeader::SetLocation**
- **wfcGTolAttachLeader::GetLeaders**
- **wfcGTolAttachLeader::GetLeaders**
- **wfcGTolAttachLeader::GetLeaderAttachType**
- **wfcGTolAttachLeader::SetLeaderAttachType**
- **wfcGTolAttachLeader::GetAnnotationPlane**
- **wfcGTolAttachLeader::SetAnnotationPlane**
- **wfcGTolAttachLeader::GetMissingLeaders**
- **wfcGTolLeaderInstructions::Create**
- **wfcGTolLeaderInstructions::GetSelection**
- **wfcGTolLeaderInstructions::SetSelection**
- **wfcGTolLeaderInstructions::GetType**
- **wfcGTolLeaderInstructions::SetType**
- **wfcGTolLeaderInstructions::GetIsZExtension**
- **wfcGTolLeaderInstructions::GetZExtensionPoint**
- **wfcGTol::SetAttachDatum**
- **wfcGTolAttachDatum::Create**
- **wfcGTolAttachDatum::GetDatum**
- **wfcGTolAttachDatum::SetDatum**
- **wfcGTol::SetAttachAnnotation**
- **wfcGTolAttachAnnotation::Create**
- **wfcGTolAttachAnnotation::GetAnnotation**
- **wfcGTolAttachAnnotation::SetAnnotation**
- **wfcGTol::SetAttachOffset**
- **wfcGTolAttachOffset::Create**
- **wfcGTolAttachOffset::GetOffset**
- **wfcGTolAttachOffset::SetOffset**
- **wfcGTolAttachOffset::GetOffsetRef**
- **wfcGTolAttachOffset::SetOffsetRef**
- **wfcGTol::SetAttachMakeDimension**
- **wfcGTolAttachMakeDimension::Create**
- **wfcGTolAttachMakeDimension::GetLocation**
- **wfcGTolAttachMakeDimension::SetLocation**

- **wfcGTolAttachMakeDimension::GetOrientHint**
- **wfcGTolAttachMakeDimension::SetOrientHint**
- **wfcGTolAttachMakeDimension::GetDimSenses**
- **wfcGTolAttachMakeDimension::SetDimSenses**
- **wfcGTolAttachMakeDimension::GetDimAttachments**
- **wfcGTolAttachMakeDimension::SetDimAttachments**
- **wfcGTolAttachMakeDimension::GetAnnotationPlane**
- **wfcGTolAttachMakeDimension::SetAnnotationPlane**

The method `wfcGTol::GetGTolAttach` retrieves all the attachment related information for a gtol using the class `wfcGTolAttach`.

The method `wfcGTolAttach::GetType` retrieves the type of attachment for a specified gtol. It uses the enumerated data type `wfcGTolAttachType` to provide information about the placement of the gtol and has one of the following values:

- `wfcGTOLATTACH_DATUM`—Specifies that the gtol is placed on its reference datum.
- `wfcGTOLATTACH_ANNOTATION`—Specifies that the gtol is attached to an annotation.
- `wfcGTOLATTACH_ANNOTATION_ELBOW`—Specifies that the gtol is attached to the elbow of an annotation.
- `wfcGTOLATTACH_FREE`—Specifies that the gtol is placed as free. It is unattached to the model or drawing.
- `wfcGTOLATTACH_LEADERS`—Specifies that the gtol is attached with one or more leader to geometry such as, edge, dimension witness line, coordinate system, axis center, axis lines, curves, or surface points, vertices, section entities, draft entities, and so on. The leaders are represented using an opaque handle, `wfcGTolAttachLeader`.
- `wfcGTOLATTACH_OFFSET`—Specifies that the gtol frame can be placed at an offset from the following drawing objects: dimension, dimension arrow, gtol, note, and symbol.
- `wfcGTOLATTACH_MAKE_DIM`—Specifies that the gtol frame is attached to a dimension line.

Use the method `wfcGTol::SetAttachFree` to set the attachment options for free type of gtol. The input argument *Attach* is specified using the class `wfcGTolAttachFree`.

Use the method `wfcGTolAttachFree::Create` to create an attachment for a gtol that is placed free. The input arguments are:

- *Location*—Specifies the location of the gtol text in model coordinates.
- *AnnotationPlane*—Specifies the annotation plane in model coordinates.

Use the methods `wfcGTolAttachFree::GetLocation` and `wfcGTolAttachFree::SetLocation` to get and set the location of the gtol text in a specified gtol using the class `pfcPoint3D`.

Use the methods `wfcGTolAttachFree::GetAnnotationPlane` and `wfcGTolAttachFree::SetAnnotationPlane` to get and set the annotation plane in a specified gtol using the class `wfcAnnotationPlane`.

Use the method `wfcGTol::SetAttachLeader` to set the attachment options for leader type of gtol. The input argument *Attach* is specified using the class `wfcGTolAttachLeader`.

Use the method `wfcGTolAttachLeader::Create` to create an attachment for leader type of gtol. The input arguments are:

- *LeaderAttachType*—Specifies the attachment type for the leader.
- *Leaders*—Specifies a sequence of gtol leaders.
- *Location*—Specifies the location of gtol text in model coordinates.
- *AnnotationPlane*—Specifies the annotation plane. For gtols defined in drawing, it returns `NULL`.

Use the methods `wfcGTolAttachLeader::GetLocation` and `wfcGTolAttachLeader::SetLocation` to get and set the location of the gtol text in a specified gtol using the class `pfcPoint3D`.

Use the methods `wfcGTolAttachLeader::GetLeaders` and `wfcGTolAttachLeader::SetLeaders` to get and set the leaders in a specified gtol using the class `wfcGTolLeaderInstructionsSeq`.

Use the methods `wfcGTolAttachLeader::GetLeaderAttachType` and `wfcGTolAttachLeader::SetLeaderAttachType` to get and set the type of leaders that can be attached in a specified gtol. The type of leaders is specified using the enumerated type `wfcGTolLeaderAttachType` and one of the valid values are:

- `wfcGTOL_LEADER`
- `wfcGTOL_NORMAL_LEADER`
- `wfcGTOL_TANGENT_LEADER`

Use the methods `wfcGTolAttachLeader::GetAnnotationPlane` and `wfcGTolAttachLeader::SetAnnotationPlane` to get and set the annotation plane on which the leader is attached to the specified gtol. The plane is specified using the class `wfcAnnotationPlane`.

Use the method `wfcGTolAttachLeader::GetMissingLeaders` to get the number of suppressed leaders due to missing references.

Use the method `wfcGTolLeaderInstructions::Create` to create instructions for a leader type of gtol.

The methods `wfcGTolLeaderInstructions::GetSelection` and `wfcGTolLeaderInstructions::SetSelection` get and set the selection of instructions for a leader type of gtol using the class `pfcSelection`.

The methods `wfcGTolLeaderInstructions::GetType` and `wfcGTolLeaderInstructions::SetType` get and set the type of instructions for a leader type of gtol.

The method `wfcGTolLeaderInstructions::GetIsZExtension` checks if the gtol leader has a Z-Extension line.

The method `wfcGTolLeaderInstructions::GetZExtensionPoint` retrieves the Z-Extension line of the gtol leader. The leader location coordinates are required when the gtol is moved to a different annotation plane.

Use the method `wfcGTol::SetAttachDatum` to set the attachment options for datum symbol type of gtol. The input argument *Attach* is specified using the class `wfcGTolAttachDatum`.

Use the method `wfcGTolAttachDatum::Create` to create an attachment for a specified datum inside a gtol. The input argument *Datum* is specified using the `pfcModelItem` class.

The methods `wfcGTolAttachDatum::GetDatum` and `wfcGTolAttachDatum::SetDatum` get and set the attachments for datum type of gtol.

From Creo Parametric 4.0 F000 onward, datum symbols are defined using datum feature symbol. The methods work with the new datum feature symbol along with the legacy datum tag annotations.

Use the method `wfcGTol::SetAttachAnnotation` to set the attachment options for annotation type of gtol. The input argument *Attach* is specified using the class `wfcGTolAttachAnnotation`.

Use the method `wfcGTolAttachAnnotation::Create` to create an attachment for a specified annotation inside a gtol. The input argument *Annotation* is specified using the `wfcAnnotation` class.

The methods `wfcGTolAttachAnnotation::GetAnnotation` and `wfcGTolAttachAnnotation::SetAnnotation` get and set the attachments for annotation type of gtol.

Use the method `wfcGTol::SetAttachOffset` to set the offset references for the geometric tolerance. The input argument *Attach* is specified using the class `wfcGTolAttachOffset`.

Use the method `wfcGTolAttachOffset::Create` to create an attachment for a gtol with offset references. The input arguments are:

- *OffsetRef*—Specifies the offset reference.
- *Offset*—Specifies the position of the offset reference as model coordinates.

The methods `wfcGTolAttachOffset::GetOffset` and `wfcGTolAttachOffset::SetOffset` get and set the position of the offset reference as model coordinates using the class `pfcVector3D`.

The methods `wfcGTolAttachOffset::GetOffsetRef` and `wfcGTolAttachOffset::SetOffsetRef` get and set the offset reference using the class `pfcSelection`. The reference can be a dimension, arrow of a dimension, another geometric tolerance, note, or a symbol instance. If there are no offset references, the output argument returns `NULL`.

Use the method `wfcGTol::SetAttachMakeDimension` to set all the attachments options to create a geometric tolerance with **Make Dim** type of reference. The input argument *Attach* is specified using the class `wfcGTolAttachMakeDimension`.

Use the method `wfcGTolAttachMakeDimension::Create` to create an attachment for a gtol with **Make Dim** type of reference.

Use the methods `wfcGTolAttachMakeDimension::GetLocation` and `wfcGTolAttachMakeDimension::SetLocation` to get and set the location of the gtol text in a specified gtol using the class `pfcPoint3D`.

Use the methods `wfcGTolAttachMakeDimension::GetOrientHint` and `wfcGTolAttachMakeDimension::SetOrientHint` to get and set the orientation of the gtol using the enumerated type `pfcDimOrientationHint` and one of the valid values are:

- `pfcORIENTATION_HINT_HORIZONTAL`
- `pfcORIENTATION_HINT_VERTICAL`
- `pfcORIENTATION_HINT_SLANTED`
- `pfcORIENTATION_HINT_ELLIPSE_RADIUS1`
- `pfcORIENTATION_HINT_ELLIPSE_RADIUS2`
- `pfcORIENTATION_HINT_ARC_ANGLE`
- `pfcORIENTATION_HINT_ARC_LENGTH`
- `pfcORIENTATION_HINT_LINE_TO_TANGENT_CURVE_ANGLE`
- `pfcORIENTATION_HINT_RADIAL_DIFF`

Use the methods `wfcGTolAttachMakeDimension::GetDimSenses` and `wfcGTolAttachMakeDimension::SetDimSenses` to get and set the information about how the gtol attaches to each attachment point of the model or drawing. The input argument *value* is specified using the class `pfcDimSenses`.

Use the methods `wfcGTolAttachMakeDimension::GetDimAttachments` and `wfcGTolAttachMakeDimension::SetDimAttachments` to get and set the points on the model or drawing where the gtol is attached. The input argument *value* is specified using the class `pfcDimensionAttachments`.

Use the methods `wfcGTolAttachMakeDimension::GetAnnotationPlane` and `wfcGTolAttachMakeDimension::SetAnnotationPlane` to get and set the annotation plane in a specified gtol using the class `wfcAnnotationPlane`.

14

Curve and Surface Collection

This chapter describes the Creo Object TOOLKIT C++ methods to access the details of curve and surface collections for query and modification. Curve and surface collections are required inputs to a variety of Creo tools such as Round, Chamfer, and Draft.

Introduction to Curve and Surface Collection

A curve collection or chain is a group of separate edges or curves that are related, for example, by a common vertex, or tangency. Once selected, these separate entities are identified as a chain so they can be modified as one unit.

The different chain types are as follows:

- One-by-one—a chain of edges, curves or composite curves, each adjacent pair of which has a coincident endpoint. Some applications may place other conditions on the resulting chain.
- Tangent—a chain defined by the selected item and the extent to which adjacent entities are tangent.
- Curve—an entire composite curve or some portion of it that is defined by two component curves of the curve.
- Boundary—an entire loop of one-sided edges that bound a quilt or some portion thereof defined by two edges of the boundary loop.
- Surf Chain—an entire loop of edges that bound a face (solid or surface) or some portion of it that is defined by two edges of the loop.
- Intent Chain—an intent chain entity, usually created as the result of two intersecting features.
- From/To—a chain that begins at a start-point, follows an edge line, and ends at the end-point.

Surface sets are one or more sets of surfaces either for use within a tool, or before entering a tool.

The definition of a surface set may not be independent in all respects from that of any other. In other words, the ability to construct some types of surface sets may depend on the presence of or on the content of others. On this account, we have different surface sets as follows:

- One-by-One Surface Set—Represents a single or a set of single selected surfaces, which belong to solid or surface geometry.
- Intent Surface Set—Represents a single or set of intent surfaces, which are used for the construction of the geometry of features. This instruction facilitates the reuse of the feature construction surface geometry as "intent" reference. This is also known as "logical object surface set".
- All Solid or Quilt Surface Set—Represents all the solid or quilt surfaces in the model.
- Loop Surface Set—Represents all the surfaces in the loop in relation with the selected surface and the edge. This is also known as "neighboring surface set".

- Seed and Boundary Surface Set—Represents all the surfaces between the seed surface and the boundary surface, excluding the boundary surface itself.

Surface set collection can be identified as a gathering a parametric set of surfaces in the context of a tool that specifically requests surface sets and is nearly identical to selection of a surface set.

Chain collection can be identified as gathering a chain in the context of a tool that specifically requests chain objects and is nearly identical to chain selection.

Collection is related to Selection as follows:

Selection is the default method of interaction with Creo system. Selection is performed without the context of any tool. In other words, the system does not know what to do with selected items until the user tells the system what to do. Collection is essentially Selection within the context of a tool. Items are gathered for a specific use or purpose as defined by the tool, which forms the Collection. It is possible to convert the collections into the sets of selections using the collection APIs.

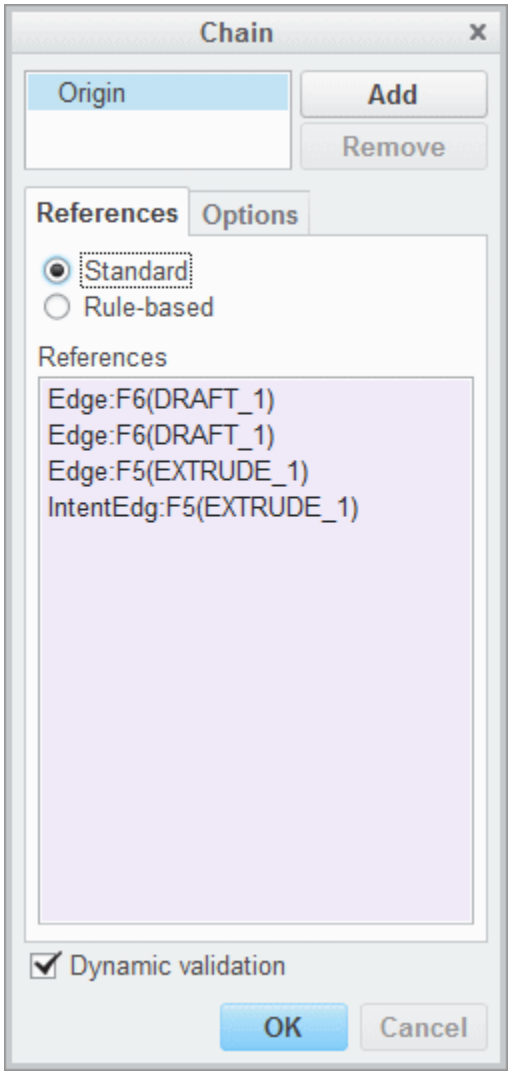
Interactive Collection

Method Introduced:

- **wfcWSolid::CollectCurves**
- **wfcWSolid::CollectSurfaces**

Use the method `wfcWSolid::CollectCurves` to interactively create a collection of curves by invoking a chain collection user interface.

Chain Collection User Interface



The input arguments for the method `wfcWSolid::CollectCurves` are as follows:

- *ChainControls*—Specifies an array defining the permitted instruction types that will be allowed to use in the user interface. The following instruction types are supported:
  - `wfcCHAINCOLLUI_ONE_BY_ONE`—for creating "One by One" chain.
  - `wfcCHAINCOLLUI_TAN_CHAIN`—for creating "Tangent" chain.
  - `wfcCHAINCOLLUI_CURVE_CHAIN`—for creating "Curve" chain.
  - `wfcCHAINCOLLUI_BNDRY_CHAIN`—for creating "Boundary Loop" chain.
  - `wfcCHAINCOLLUI_FROM_TO`—for creating "From-To" chain.

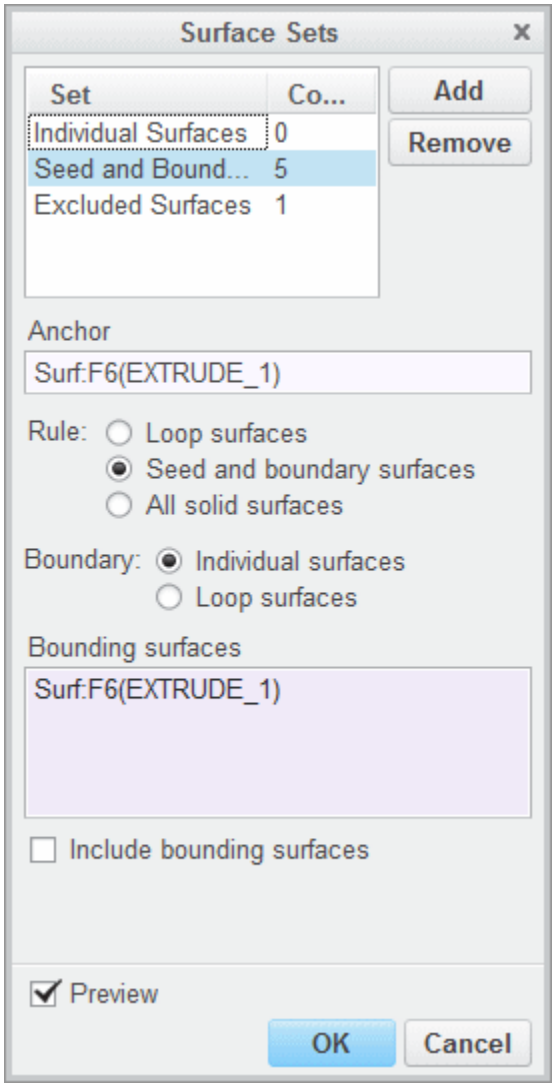
- `wfcCHAINCOLLUI_ALLOW_LENGTH_ADJUSTMENT`—for allowing length adjustment in curves.
- `wfcCHAINCOLLUI_ALLOW_ALL`—for allowing all the supported instruction types.
- `wfcCHAINCOLLUI_ALLOW_EXCLUDED`—for excluding chain.
- `wfcCHAINCOLLUI_ALLOW_APPENDED`—for appending chain.

- *FilterMethod*—Specifies the filter method `wfcCollectCurvesFilter::FilterSelections`. The filter method is called before each selection is accepted. If you want your application to reject a certain selection, *FilterMethod* should return false for that particular selection. You can pass null to skip the filter.
- *AppData*—Specifies the application data that will be passed to the filter method.
- *AppendColl*—Appends the curve collection to the final `wfcCollection` object that will be returned. If this collection already contains instructions, then they will be appended into the details shown in the **Chain** collection user interface. Use the programmatic methods to access curve collections and to extract the required information.

The method returns a handle to `wfcCollection` object describing the current resulting curves and edges from the collection.

Use the method `wfcWSolid::CollectSurfaces` to interactively create a collection of surfaces by invoking a surface sets dialog.

Surface Sets dialog box



The input arguments for the method `wfcWSolid::CollectSurfaces` are:

- *InstrTypes*—Specifies an array defining the permitted instruction types for surfaces using the enumerated type `wfcSurfaceCollectionInstrType`. Surface collection instructions can be of the following types:
  - `wfcSURFCOLL_SINGLE_SURF`—Instruction specifying a set of single surfaces.
  - `wfcSURFCOLL_SEED_N_BND`—Instruction specifying a combination of Seed and Boundary type of surfaces.
  - `wfcSURFCOLL_QUILT_SRFS`—Instruction specifying quilt type of surfaces.

- wfcSURFCOLL_ALL_SOLID_SRFS—Instruction specifying all solid surfaces in the model.
- wfcSURFCOLL_NEIGHBOR—Instruction specifying neighbor type of surfaces (boundary loop).
- wfcSURFCOLL_NEIGHBOR_INC—Instruction specifying neighbor type of surfaces (boundary loop) and also includes the seed surfaces.
- wfcSURFCOLL_ALL_QUILT_SRFS—Instruction specifying all quilts in the model.
- wfcSURFCOLL_ALL_MODEL_SRFS—Instruction specifying all the surfaces in the model.
- wfcSURFCOLL_LOGOBJ_SRFS—Instruction specifying intent surfaces. Intent surfaces are also known as "logical objects".
- wfcSURFCOLL_DTM_PLN—Instruction specifying datum plane selection.
- wfcSURFCOLL_DISALLOW_QLT—Instruction specifying that do not allow selections from quilts.
- wfcSURFCOLL_DISALLOW_SLD—Instruction specifying that do not allow selections from solid geometry.
- wfcSURFCOLL_DONT_MIX—Instruction allowing selections from only solid or only quilt but no mixing.
- wfcSURFCOLL_SAME_SRF_LST—Instruction allowing selections from same solid or same quilt.
- wfcSURFCOLL_USE_BACKUP—Instruction prompting Creo to regenerate using backups.
- wfcSURFCOLL_DONT_BACKUP—Instruction specifying that do not back up copy of references.
- wfcSURFCOLL_DISALLOW_LOBJ—Instruction specifying that do not allow selections from intent surfaces or logical objects.
- wfcSURFCOLL_ALLOW_DTM_PLN—Instruction specifying datum plane selection.
- wfcSURFCOLL_SEED_N_BND_INC_BND—Instruction specifying a combination of Seed and Boundary type of surfaces and also includes the seed surfaces.
- wfcSURFCOLL_GEOM_RULE—Instruction specifying collection of surfaces using geometry rules.
- wfcSURFCOLL_TANG_SRF—Instruction specifying collection of tangent surfaces.

- wfcSURFCOLL_SHAPE_BASED—Instruction specifying collection of shape based surfaces.

- *FilterMethod*—Specifies the filter method wfcCollectSurfacesFilter::FilterSelections. The filter method is called before each selection is accepted. If you want your application to reject a certain selection, *FilterMethod* should return false for that particular selection. You can pass null to skip the filter.
- *AppendColl*—Appends the surface collection to the final wfcCollection object that will be returned. If this collection already contains instructions, then they will be appended into the details shown in the **Surface Sets** collection user interface. Use the programmatic methods to access surface collections and to extract the required information.

Programmatic Access to Collections

The wfcCollection Interface

The wfcCollection interface contains curve and surface collection interfaces for programmatic access to collections. It represents a chain or surface set and extracts the details from an appropriate structure from the Creo application.

The collection object wfcCollection must contain either a collection of curves or a collection of surfaces. It must not contain a combination of both object types.

Methods Introduced:

- **wfcCollection::Create**
- **wfcWModel::GenerateSelectionsFromCollection**

The method wfcCollection::Create creates a data object that contains information about the collection of curves or surfaces.

The method wfcWModel::GenerateSelectionsFromCollection returns a sequence of pfcSelections object depending on the rules and instructions in the curve or surface collection.

Curve Collection

Methods Introduced:

- **wfcCollection::GetCrvCollection**
- **wfcCollection::SetCrvCollection**
- **wfcCurveCollection::Create**

- **wfcCurveCollection::GetInstructions**
- **wfcCurveCollection::SetInstructions**
- **wfcCurveCollectionInstruction::Create**
- **wfcCurveCollectionInstruction::GetAttributes**
- **wfcCurveCollectionInstruction::SetAttributes**
- **wfcCurveCollectionInstruction::GetReferences**
- **wfcCurveCollectionInstruction::SetReferences**
- **wfcCurveCollectionInstruction::GetType**
- **wfcCurveCollectionInstruction::SetType**
- **wfcCurveCollectionInstruction::GetValue**
- **wfcCurveCollectionInstruction::SetValue**
- **wfcCrvCollectionInstrAttribute::Create**
- **wfcCrvCollectionInstrAttribute::GetAttribute**
- **wfcCrvCollectionInstrAttribute::SetAttribute**
- **wfcCollectCurvesFilter::FilterSelections**

The method `wfcCollection::GetCrvCollection` returns the collection of curves for the element `PRO_E_STD_CURVE_COLLECTION_APPL` for the specified model. Use the method `wfcCollection::SetCrvCollection` to set the collection of curves for the element `PRO_E_STD_CURVE_COLLECTION_APPL`.

The method `wfcCurveCollection::Create` creates a data object that contains information about the curve collection instructions.

Use the method `wfcCurveCollection::GetInstructions` and `wfcCurveCollection::SetInstructions` to get and set the instructions from the curve collection.

The method `wfcCurveCollectionInstruction::Create` creates a data object that contains information about the parameters set in the curve collection instructions.

The methods `wfcCurveCollectionInstruction::GetAttributes` and `wfcCurveCollectionInstruction::SetAttributes` get and set the attributes contained in a curve collection instruction.

Use the methods `wfcCurveCollectionInstruction::GetReferences` and `wfcCurveCollectionInstruction::SetReferences` to get and set the references contained in a curve collection instruction.

Use the methods `wfcCurveCollectionInstruction::GetType` and `wfcCurveCollectionInstruction::SetType` to get and set the curve collection instruction type using the enumerated type `wfcCurveCollectionInstrType`.

Curve collection instructions can be of the following types:

- `wfcCURVCOLL_EMPTY_INSTR`—to be used when you do not want to pass any other instruction.
- `wfcCURVCOLL_ADD_ONE_INSTR`—for creating "One by One" chain.
- `wfcCURVCOLL_TAN_INSTR`—for creating "Tangent" chain.
- `wfcCURVCOLL_CURVE_INSTR`—for creating "Curve" chain.
- `wfcCURVCOLL_SURF_INSTR`—for creating "Surface Loop" chain.
- `wfcCURVCOLL_BNDRY_INSTR`—for creating "Boundary Loop" chain.
- `wfcCURVCOLL_LOG_OBJ_INSTR`—for creating "Logical Object" chain.
- `wfcCURVCOLL_PART_INSTR`—for creating chain on all possible references, or to choose from convex or concave only.
- `wfcCURVCOLL_FEATURE_INSTR`—for creating chain from feature curves.
- `wfcCURVCOLL_FROM_TO_INSTR`—for creating "From-To" chain.
- `wfcCURVCOLL_EXCLUDE_ONE_INSTR`—for excluding the entity from the chain.
- `wfcCURVCOLL_TRIM_INSTR`—to trim chain.
- `wfcCURVCOLL_EXTEND_INSTR`—to extend chain.
- `wfcCURVCOLL_START_PNT_INSTR`—to set the chain start point.
- `wfcCURVCOLL_ADD_TANGENT_INSTR`—to add all edges tangent to the ends of the chain.
- `wfcCURVCOLL_ADD_POINT_INSTR`—to add selected point or points to the collection.
- `wfcCURVCOLL_OPEN_CLOSE_LOOP_INSTR`—to add a closed chain that is considered as open.
- `wfcCURVCOLL_QUERY_INSTR`—for creating “Query” chain.
- `wfcCURVCOLL_RESERVED_INSTR`—to determine the number of instructions defined in the curve instruction.
- `wfcCURVCOLL_CNTR_INSTR`—to add contours to the collection.

The methods `wfcCurveCollectionInstruction::GetValue` and `wfcCurveCollectionInstruction::SetValue` get and set the value of a curve collection instruction. These methods are used only for instructions of type `wfcCURVCOLL_TRIM_INSTR` and `wfcCURVCOLL_EXTEND_INSTR`.

The method `wfcCrvCollectionInstrAttribute::Create` creates a data object that contains information about the special attribute in curve collection instruction.

Use the method `wfcCrvCollectionInstrAttribute::GetAttribute` to check whether a special attribute is set for the curve collection instruction using the enumerated type `wfcCurveCollectionInstrAttribute`.

The curve collection instruction attributes can be of the following types:

- `wfcCURVCOLL_NO_ATTR`—applicable when there are no attributes present.
- `wfcCURVCOLL_ALL`—applicable for all edges.
- `wfcCURVCOLL_CONVEX`—applicable for convex edges only.
- `wfcCURVCOLL_CONCAVE`—applicable for concave edges only.

Use the method `wfcCrvCollectionInstrAttribute::SetAttribute` to set a special attribute in the curve collection instruction.

Use the method `wfcCollectCurvesFilter::FilterSelections` to check if the current curve selection satisfies the filter criteria. The curve selection is accepted only if the method returns True.

Surface Collection

Methods Introduced:

- **wfcCollection::GetSurfCollection**
- **wfcCollection::SetSurfCollection**
- **wfcSurfaceCollection::Create**
- **wfcSurfaceCollection::GetInstructions**
- **wfcSurfaceCollection::SetInstructions**
- **wfcSurfaceCollectionInstruction::Create**
- **wfcSurfaceCollectionInstruction::GetInclude**
- **wfcSurfaceCollectionInstruction::SetInclude**
- **wfcSurfaceCollectionInstruction::GetSrfCollectionReferences**
- **wfcSurfaceCollectionInstruction::SetSrfCollectionReferences**
- **wfcSurfaceCollectionInstruction::GetType**
- **wfcSurfaceCollectionInstruction::SetType**
- **wfcSurfaceCollectionInstruction::GetRules**
- **wfcSurfaceCollectionInstruction::SetRules**
- **wfcSurfaceCollectionReference::Create**
- **wfcSurfaceCollectionReference::GetRefType**

- **wfcSurfaceCollectionReference::SetRefType**
- **wfcSurfaceCollectionReference::GetReference**
- **wfcSurfaceCollectionReference::SetReference**
- **wfcCollectSurfacesFilter::FilterSelections**

The method `wfcCollection::GetSurfCollection` returns the collection of surfaces for the element `PRO_E_STD_SURF_COLLECTION_APPL` for the specified model. Use the method `wfcCollection::SetSurfCollection` to set the collection of curves for the element `PRO_E_STD_SURF_COLLECTION_APPL`.

The method `wfcSurfaceCollection::Create` creates a data object that contains information about the surface collection instructions.

Use the methods `wfcSurfaceCollection::GetInstructions` and `wfcSurfaceCollection::SetInstructions` to get and set the instructions from the surface collection.

The method `wfcSurfaceCollectionInstruction::Create` creates a data object that contains information about the parameters set in the surface collection instructions.

Use the method `wfcSurfaceCollectionInstruction::GetInclude` to check the boolean value of the surface collection instruction attribute. If the value is set to `True`, the surfaces generated by this instruction add surfaces to the overall set. If the value is set to `False`, the surfaces generated by this instruction are removed from in the overall set.

Use the methods `wfcSurfaceCollectionInstruction::GetSrfCollectionReferences` and `wfcSurfaceCollectionInstruction::SetSrfCollectionReferences` to get and set the references contained in a surface collection.

Use the methods `wfcSurfaceCollectionInstruction::GetType` and `wfcSurfaceCollectionInstruction::SetType` to get and set the surface collection instruction type using the enumerated type `wfcSurfaceCollectionInstrType`.

The methods `wfcSurfaceCollectionInstruction::GetRules` and `wfcSurfaceCollectionInstruction::SetRules` get and set rules for surface collection instructions using the enumerated type `wfcSurfaceCollectionRule`. The valid values are:

- `wfcSURFCOLL_ALL_GEOM_RULE`—Specifies that all the rules are applied. Otherwise, any applicable geometry rule is applied.
- `wfcSURFCOLL_CO_PLANNAR_GEOM_RULE`—Specifies that the surfaces coplanar to the seed surface should be collected.

- wfcSURFCOLL_PARALLEL_GEOM_RULE—Specifies that the surfaces parallel to the seed surface should be collected.
- wfcSURFCOLL_CO_AXIAL_GEOM_RULE—Specifies that the surfaces coaxial with the seed surface should be collected.
- wfcSURFCOLL_EQ_RADIUS_GEOM_RULE—Specifies that the surfaces with the same radius and type as the seed surface should be collected.
- wfcSURFCOLL_SAME_CONVEXITY_GEOM_RULE—Specifies that the surfaces that have the same convexity and type as the seed surface should be collected.

Note

wfcSURFCOLL_ALL_GEOM_RULE, wfcSURFCOLL_CO_PLANNAR_GEOM_RULE, wfcSURFCOLL_PARALLEL_GEOM_RULE, wfcSURFCOLL_CO_AXIAL_GEOM_RULE, wfcSURFCOLL_EQ_RADIUS_GEOM_RULE, and wfcSURFCOLL_SAME_CONVEXITY_GEOM_RULE are related to wfcSURFCOLL_GEOM_RULE, that is, the geometry rule surface set.

- wfcSURFCOLL_SHAPE_ROUND—Specifies that the surfaces with round shape should be collected. If used alone then only the primary round shapes are added to the collection set.
- wfcSURFCOLL_SHAPE_PROTR_BOSS—Specifies that the surfaces of the boss should be collected. The protrusion surfaces with the secondary shapes are also added to the set.
- wfcSURFCOLL_SHAPE_PROTR_RIB—Specifies that the surfaces of the rib should be collected. The protrusion surfaces without the secondary shapes are added to set.
- wfcSURFCOLL_SHAPE_CUT_POCKET—Specifies that the surfaces of the pocket should be collected. The cut surfaces with the secondary shapes are also added to set.
- wfcSURFCOLL_SHAPE_CUT_SLOT—Specifies that the surfaces of the slot should be collected. The cut surfaces without the secondary shapes are added to set.
- wfcSURFCOLL_SHAPE_MORE_SHAPES—Use with wfcSURFCOLL_SHAPE_ROUND to add the secondary round shapes also.

Note

wfcSURFCOLL_SHAPE_ROUND, wfcSURFCOLL_SHAPE_PROTR_BOSS, wfcSURFCOLL_SHAPE_PROTR_RIB, wfcSURFCOLL_SHAPE_CUT_POCKET, wfcSURFCOLL_SHAPE_CUT_SLOT, and wfcSURFCOLL_SHAPE_MORE_SHAPES are related to wfcSURFCOLL_TANG_SRF, that is, the shape surface set.

- wfcSURFCOLL_TANGENT_NEIGBOURS_ONLY—Specifies that the surfaces that have at least one tangent edge with another surface in the tangent surface set, starting from the seed surface should be collected.

Note

wfcSURFCOLL_TANGENT_NEIGBOURS_ONLY is related to wfcSURFCOLL_TANG_SRF, that is, the tangent surface set.

The method wfcSurfaceCollectionReference::Create creates a data object that contains information about the references in the surface collection instructions.

Use the methods wfcSurfaceCollectionReference::GetRefType and wfcSurfaceCollectionReference::SetRefType to get and set the type of reference contained in the surface collection reference using the enumerated data type wfcSurfaceCollectionRefType.

Surface collection references can be of the following types:

- wfcSURFCOLL_REF_SINGLE—Specifying the collection reference belonging to the "single surface set" type of instruction. This type of reference can belong to single surface type of instruction.
- wfcSURFCOLL_REF_SINGLE_EDGE—Specifying the collection reference belonging to the an "single surface set" edge type of instruction.
- wfcSURFCOLL_REF_SEED—Specifying the collection reference to be the seed surface. This type of reference can belong to seed and boundary type of instruction.
- wfcSURFCOLL_REF_SEED_EDGE—Specifying the collection reference of seed edge type. This type of reference can belong to seed and boundary type of instruction.
- wfcSURFCOLL_REF_BND—Specifying the collection reference to be a boundary surface. This type of reference can belong to seed and boundary type of instruction.

A single seed and boundary type of instruction will have at least one of each seed and boundary type of reference.

- `wfcSURFCOLL_REF_NEIGHBOR`—Specifying the collection reference to be of neighbor type. This type of reference belongs neighbor type of instruction.
- `wfcSURFCOLL_REF_NEIGHBOR_EDGE`—Specifying the collection reference of neighbor edge type. This type of reference belongs to neighbor type of instruction.

  A neighbor type of instruction will have one neighbor and one neighbor edge type of reference.
- `wfcSURFCOLL_REF_GENERIC`—Specifying the collection reference to be of generic type. This type of reference can belong to intent surfaces , quilt and all-solid type of instructions.

Use the methods `wfcSurfaceCollectionReference::GetReference` and `wfcSurfaceCollectionReference::SetReference` to get and set the references contained in a surface collection instruction.

Use the method `wfcCollectSurfacesFilter::FilterSelections` to check if the current surface selection satisfies the filter criteria. The surface selection is accepted only if the method returns True.

15

Windows and Views

Creo Object TOOLKIT C++ provides access to Creo windows and saved views. This chapter describes the methods that provide this access.

Windows

This section describes the Creo Object TOOLKIT C++ methods that access `Window` objects. The topics are as follows:

Getting a Window Object

Methods Introduced:

- **pfcBaseSession::GetCurrentWindow**
- **pfcBaseSession::CreateModelWindow**
- **pfcModel::Display**
- **pfcBaseSession::ListWindows**
- **pfcBaseSession::GetWindow**
- **pfcBaseSession::OpenFile**
- **pfcBaseSession::GetModelWindow**

The method `pfcBaseSession::GetCurrentWindow` provides access to the current active window in Creo application.

The method `pfcBaseSession::CreateModelWindow` creates a new window that contains the model that was passed as an argument.

Note

You must call the method `pfcModel::Display` for the model geometry to be displayed in the window.

Use the method `pfcBaseSession::ListWindows` to get a list of all the current windows in session.

The method `pfcBaseSession::GetWindow` gets the handle to a window given its integer identifier.

The method `pfcBaseSession::OpenFile` returns the handle to a newly created window that contains the opened model.

Note

If a model is already open in a window the method returns a handle to the window.

The method `pfcBaseSession::GetModelWindow` returns the handle to the window that contains the opened model, if it is displayed.

Creating Windows

Method Introduced:

- **wfcWSession::CreateAccessorywindowWithTree**
- **wfcWSession::CreateBarewindow**

In Creo application, when the main window is active, you can open an accessory window for operations such as editing an inserted component or a feature, previewing an object, selecting a reference, and so on. The model tree associated with this Creo object is also displayed in the accessory window.

Use the method `wfcWSession::CreateAccessorywindowWithTree` to open an accessory window containing the specified object. If a window is already open with the specified object, the method returns the identifier of that window. If an empty window is already open, the method uses this window to open the object. The input argument *EnableTree* controls the display of the model tree in the accessory window. If set to true the model tree is displayed in the accessory window.

Note

The accessory window has restricted menu options and does not have any toolbar.

The method `wfcWSession::CreateBarewindow` opens a window containing the specified object. If a window is already open with the specified object, the method returns the identifier of that window. If an empty window is already open, the method uses this window to open the object.

Note

The window does not have any menus or toolbar. All the mouse capabilities for a view such as, zoom, rotate and pan are available.

Window Operations

Methods Introduced:

- **pfcWindow::GetHeight**
- **pfcWindow::GetWidth**
- **pfcWindow::GetXPos**
- **pfcWindow::GetYPos**
- **pfcWindow::GetGraphicsAreaHeight**
- **pfcWindow::GetGraphicsAreaWidth**
- **pfcWindow::Clear**
- **pfcWindow::Repaint**
- **pfcWindow::Refresh**
- **pfcWindow::Close**
- **wfcWWindow::Refit**
- **pfcWindow::Activate**
- **pfcWindow::GetId**
- **pfcBaseSession::FlushCurrentWindow**

The methods `pfcWindow::GetHeight`, `pfcWindow::GetWidth`, `pfcWindow::GetXPos`, and `pfcWindow::GetYPos` retrieve the height, width, x-position, and y-position of the window respectively. The values of these parameters are normalized from 0 to 1.

The methods `pfcWindow::GetGraphicsAreaHeight` and `pfcWindow::GetGraphicsAreaWidth` retrieve the height and width of the Creo graphics area window without the border respectively. The values of these parameters are normalized from 0 to 1. For both the window and graphics area sizes, if the object occupies the whole screen, the window size returned is 1. For example, if the screen is `1024 pixels` wide and the graphics area is `512 pixels`, then the width of the graphics area window is returned as 0.5.

The method `pfcWindow::Clear` removes geometry from the window.

Both `pfcWindow::Repaint` and `pfcWindow::Refresh` repaint solid geometry. However, the `Refresh` method does not remove highlights from the screen and is used primarily to remove temporary geometry entities from the screen.

Use the method `pfcWindow::Close` to close the window. If the current window is the original window created when Creo application started, this method clears the window. Otherwise, it removes the window from the screen.

Use the method `wfcWWindow::Refit` to refit the model in the specified window so that the entire model can be viewed.

The method `pfcWindow::Activate` activates a window. This method is available only in the asynchronous mode.

The method `pfcWindow::GetId` retrieves the ID of the Creo window.

The method `pfcBaseSession::FlushCurrentWindow` flushes the pending display commands on the current window.

Note

It is recommended to call this method only after completing all the display operations. Excessive use of this method will cause major slow down of systems running on Windows Vista and Windows 7.

Embedded Browser

Methods Introduced:

- **pfcWindow::GetURL**
- **pfcWindow::SetURL**
- **pfcWindow::GetBrowserSize**
- **pfcWindow::SetBrowserSize**
- **wfcBrowser::ExecuteScript**
- **wfcBrowser::ExecuteScriptFromFile**
- **wfcBrowser::GetId**
- **wfcBrowser::GetUrl**
- **wfcBrowser::AddActionListener**
- **wfcBrowserListener::OnBeforeDestroy**

The methods `pfcWindow::GetURL` and `pfcWindow::SetURL` enables you to find and change the URL displayed in the embedded browser in the Creo window.

The methods `pfcWindow::GetBrowserSize` and `pfcWindow::SetBrowserSize` enables you to find and change the size of the embedded browser in the Creo window.

Note

The methods `pfcWindow::GetBrowserSize` and `pfcWindow::SetBrowserSize` are not supported if the browser is open in a separate window.

The method `wfcBrowser::ExecuteScript` executes a Java script on the browser page.

The method `wfcBrowser::ExecuteScriptFromFile` executes a Java script file on the browser page.

The method `wfcBrowser::GetId` returns a unique identifier of the browser page.

The method `wfcBrowser::GetUrl` returns a url of the browser page.

The method `wfcBrowser::AddActionListener` adds a listener object for browser event callback. The input arguments are:

- *Listener*— Specifies the listener which is defined by the class `wfcBrowserListener`.
- *EventTypes*—Specifies the type of the events to listen and is defined by the enumerated data type `wfcBrowserEventType`. The valid value `wfcON_BEFORE_DESTROY` specifies the event before destroying the browser window.

Use the method `wfcBrowserListener::OnBeforeDestroy` to return a callback when an embedded browser is destroyed in Creo. The callback is returned for each page of the browser window.

Views

This section describes the Creo Object TOOLKIT C++ methods that access `pfcView` objects. The topics are as follows:

Getting a View Object

Methods Introduced:

- **pfcViewOwner::RetrieveView**
- **pfcViewOwner::GetView**
- **pfcViewOwner::ListViews**
- **pfcViewOwner::GetCurrentView**

Any solid model inherits from the interface `pfcViewOwner`. This will enable you to use these methods on any solid object.

The method `pfcViewOwner::RetrieveView` sets the current view to the orientation previously saved with a specified name.

Use the method `pfcViewOwner::GetView` to get a handle to a named view without making any modifications.

The method `pfcViewOwner::ListViews` returns a list of all the views previously saved in the model.

From Creo Parametric2.0 M120 onward, the method `pfcViewOwner::GetCurrentView` has been deprecated. The method returns a view handle that represents the current orientation. Although this view does not have a name, you can use this view to find or modify the current orientation.

View Operations

Methods Introduced:

- **pfcView::GetName**
- **pfcView::GetIsCurrent**
- **pfcView::Reset**
- **pfcViewOwner::SaveView**

To get the name of a view given its identifier, use the method `pfcView::GetName`.

The method `pfcView::GetIsCurrent` determines if the View object represents the current view.

The `pfcView::Reset` method restores the current view to the default view.

To store the current view under the specified name, call the method `pfcViewOwner::SaveView`.

Coordinate Systems and Transformations

This section describes the various coordinate systems used by Creo application, which are accessible from Creo Object TOOLKIT C++. It also explains how to transform from one coordinate system to another.

Coordinate Systems

Creo applications and Creo Object TOOLKIT C++ use the following coordinate systems:

The following sections describe each of these coordinate systems.

Solid Coordinate System

The solid coordinate system is the three-dimensional, Cartesian coordinate system used to describe the geometry of a Creo solid model. In a part, the solid coordinate system describes the geometry of the surfaces and edges. In an assembly, the solid coordinate system also describes the locations and orientations of the assembly members.

You can visualize the solid coordinate system in Creo application by creating a coordinate system datum with the option **Default**. Distances measured in solid coordinates correspond to the values of dimensions as seen by the Creo user.

Solid coordinates are used by Creo Object TOOLKIT C++ for all the methods that look at geometry and most of the methods that draw three-dimensional graphics.

Screen Coordinate System

The screen coordinate system is two-dimensional coordinate system that describes locations in a Creo window. This is an intermediate coordinate system after which the screen points are transformed to screen pixels. All the models are first mapped to the screen coordinate system. When the user zooms or pans the view, the screen coordinate system follows the display of the solid, so a particular point on the solid always maps to the same screen coordinate. The mapping changes only when the view orientation is changed.

Screen coordinates are used by some of the graphics methods, the mouse input methods, and all methods that draw graphics or manipulate items on a drawing.

Window Coordinate System

The window coordinate system is similar to the screen coordinate system. After mapping the models to the screen coordinate system, they are mapped to the window coordinate before being drawn to screen pixels based on screen resolution. When pan or zoom values are applied to the coordinates in the screen coordinate system, they result in window coordinates. When an object is first displayed in a window, or the option **View ▸ Refit** is used, the screen and window coordinates are the same.

Window coordinates are needed only if you need to take account of zoom and pan —for example, to find out whether a point on the solid is visible in the window, or to draw two-dimensional text in a particular window location, regardless of pan and zoom.

Drawing Coordinate System

The drawing coordinate system is a two-dimensional system that describes the location on a drawing relative to the bottom, left corner, and measured in drawing units. For example, on a U.S. letter-sized, landscape-format drawing sheet that uses inches, the top, right-corner is (11, 8.5) in drawing coordinates.

The Creo Object TOOLKIT C++ methods and properties that manipulate drawings generally use screen coordinates.

Drawing View Coordinate System

The drawing view coordinate system is used to describe the locations of entities in a drawing view.

Assembly Coordinate System

An assembly has its own coordinate system that describes the positions and orientations of the member parts, subassemblies, and the geometry of datum features created in the assembly.

When an assembly is retrieved into memory each member is also loaded and continues to use its own solid coordinate system to describe its geometry.

This is important when you are analyzing the geometry of a subassembly and want to extract or display the results relative to the coordinate system of the parent assembly.

Datum Coordinate System

A coordinate system datum can be created anywhere in any part or assembly, and represents a user-defined coordinate system. It is often a requirement in a Creo Object TOOLKIT C++ application to describe geometry relative to such a datum.

Section Coordinate System

Every sketch has a coordinate system used to locate entities in that sketch. Sketches used in features will use a coordinate system different from that of the solid model.

Transformations

Methods Introduced:

- **pfcTransform3D::Invert**
- **pfcTransform3D::TransformPoint**
- **pfcTransform3D::TransformVector**
- **pfcTransform3D::GetMatrix**
- **pfcTransform3D::SetMatrix**
- **pfcTransform3D::GetOrigin**
- **pfcTransform3D::GetXAxis**
- **pfcTransform3D::GetYAxis**
- **pfcTransform3D::GetZAxis**
- **pfcMakeMatrixOrthonormal**

All coordinate systems are treated in Creo Object TOOLKIT C++ as if they were three-dimensional. Therefore, a point in any of the coordinate systems is always represented by the `pfcPoint3D` class:

Vectors store the same data but are represented for clarity by the `pfcVector3D` class.

Screen coordinates contain a z-value whose positive direction is outwards from the screen. The value of z is not generally important when specifying a screen location as an input to a method, but it is useful in other situations. For example, if you select a datum plane, you can find the direction of the plane by calculating the normal to the plane, transforming to screen coordinates, then looking at the sign of the z-coordinate.

A transformation between two coordinate systems is represented by the `pfcTransform3D` class. This class contains a 4x4 matrix that combines the conventional 3x3 matrix that describes the relative orientation of the two systems, and the vector that describes the shift between them.

The 4x4 matrix used for transformations is as follows:

$$\begin{bmatrix} X' & Y' & Z' & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \begin{bmatrix} \cdots & \cdots & \cdots & 0 \\ \cdots & \cdots & \cdots & 0 \\ \cdots & \cdots & \cdots & 0 \\ Xs & Ys & Zs & 1 \end{bmatrix}$$

The utility method `pfcTransform3D::Invert` inverts a transformation matrix so that it can be used to transform points in the opposite direction.

Creo Object TOOLKIT C++ provides two utilities for performing coordinate transformations. The method `pfcTransform3D::TransformPoint` transforms a three-dimensional point and `pfcTransform3D::TransformVector` transforms a three-dimensional vector.

The method `pfcMakeMatrixOrthonormal` converts a non-orthonormal matrix to an orthonormal matrix with the specified scaling factor. The input arguments follow:

- *Matrix*—The matrix to be converted to orthonormal.
- *Scale*—Scale factor to be applied on the matrix.

Transforming to Screen Coordinates

Methods Introduced:

- **pfcView::GetTransform**
- **pfcView::SetTransform**
- **pfcView::Rotate**
- **pfcViewOwner::GetCurrentViewTransform**
- **pfcViewOwner::SetCurrentViewTransform**
- **pfcViewOwner::CurrentViewRotate**
- **pfcTransform3D::Invert**

The view matrix describes the transformation from solid to screen coordinates. The method `pfcView::GetTransform` provides the view matrix for a model in the view. The method `pfcView::SetTransform` allows you to specify the transformation matrix for a model in the view.

To get and set the transformation matrix for a model in the current view, use the methods `pfcViewOwner::GetCurrentViewTransform` and `pfcViewOwner::SetCurrentViewTransform`.

The method `pfcView::Rotate` rotates an object , relative to the X, Y, or Z axis for the specified rotation angle.

To rotate an object in the current view, use the method `pfcViewOwner::CurrentViewRotate`.

To transform from screen to solid coordinates, invert the transformation matrix using the method `pfcTransform3D::Invert`.

Transforming to Coordinate System Datum Coordinates

Method Introduced:

- **pfcCoordSystem::GetCoordSys**

The method `pfcCoordSystem::GetCoordSys` provides the location and orientation of the coordinate system datum in the coordinate system of the solid that contains it. The location is in terms of the directions of the three axes and the position of the origin.

Transforming Window Coordinates

Methods Introduced

- **pfcWindow::GetScreenTransform**
- **pfcWindow::SetScreenTransform**
- **pfcScreenTransform::SetPanX**
- **pfcScreenTransform::SetPanY**
- **pfcScreenTransform::SetZoom**

You can alter the pan and zoom of a window by using a Screen Transform object. This object contains three attributes. PanX and PanY represent the horizontal and vertical movement. Every increment of 1.0 moves the view point one screen width or height. Zoom represents a scaling factor for the view. This number must be greater than zero.

Transforming Coordinates of an Assembly Member

Method Introduced:

- **pfcComponentPath::GetTransform**

The method `pfcComponentPath::GetTransform` provides the matrix for transforming from the solid coordinate system of the assembly member to the solid coordinates of the parent assembly, or the reverse.

16

ModelItem

This chapter describes the Creo Object TOOLKIT C++ methods that enable you to access and manipulate `ModelItems`.

Solid Geometry Traversal

Solid models are made up of 11 distinct types of `pfcModelItem`, as follows:

- `pfcFeature`
- `pfcSurface`
- `pfcEdge`
- `pfcCurve` (datum curve)
- `pfcAxis` (datum axis)
- `pfcPoint` (datum point)
- `pfcQuilt` (datum quilt)
- `pfcLayer`
- `pfcNote`
- `pfcDimension`
- `pfcRefDimension` All models inherit from the ModelItemOwner

Each model item is assigned a unique identification number that will never change. In addition, each model item can be assigned a string name. Layers, points, axes, dimensions, and reference dimensions are automatically assigned a name that can be changed.

Getting ModelItem Objects

Methods Introduced:

- **pfcModelItemOwner::ListItems**
- **pfcFeature::ListSubItems**
- **pfcLayer::ListItems**
- **pfcModelItemOwner::GetItemById**
- **pfcModelItemOwner::GetItemByName**
- **pfcFamColModelItem::GetRefItem**
- **pfcSelection::GetSelItem**
- **wfcWSelection::EvaluateAngle**

All models inherit from the interface `pfcModelItemOwner`. The method `pfcModelItemOwner::ListItems` returns a sequence of `pfcModelItems` contained in the model. You can specify which type of `pfcModelItem` to collect by passing in one of the enumerated `pfcModelItemType` objects, or you can collect all `pfcModelItems` by passing `null` as the model item type.

If the model has multiple bodies, the method `pfcModelItemOwner::ListItems` returns the exception `pfcXToolkitMultibodyUnsupported`.

The methods `pfcFeature::ListSubItems` and `pfcLayer::ListItems` produce similar results for specific features and layers. These methods return a list of subitems in the feature or items in the layer.

To access specific model items, call the method `pfcModelItemOwner::GetItemById`. This methods enables you to access the model item by identifier.

To access specific model items, call the method `pfcModelItemOwner::GetItemByName`. This method enables you to access the model item by name.

The method `pfcFamColModelItem::GetRefItem` returns the dimension or feature used as a header for a family table.

The method `pfcSelection::GetSelItem` returns the item selected interactively by the user.

The method `wfcWSelection::EvaluateAngle` measures the angle between two geometry items selected in the selection object `wfcWSelection` by the user. Both objects must be straight, solid edges.

ModelItem Information

Methods Introduced:

- **pfcModelItem::GetName**
- **pfcModelItem::SetName**
- **wfcWModelItem::GetDefaultName**
- **wfcWModelItem::IsNameReadOnly**
- **wfcWModelItem::DeleteUserDefinedName**
- **pfcModelItem::GetId**
- **pfcModelItem::GetType**
- **wfcWModelItem::Hide**
- **wfcWModelItem::Unhide**
- **wfcWModelItem::IsHidden**
- **wfcWModelItem::IsZoneFeature**
- **wfcWView::GetModelItemFromView**
- **wfcViewModelitem::GetViewFromModelItem**

Certain pfcModelItems also have a string name that can be changed at any time. The methods pfcModelItem::GetName and pfcModelItem::SetName access this name.

The method wfcWModelItem::GetDefaultName gets the default name assigned to the model item by Creo application at the time of creation.

The method wfcWModelItem::IsNameReadOnly identifies if you can modify the name of the model item.

The method wfcWModelItem::DeleteUserDefinedName deletes the user-defined name of the model item from the Creo database.

The method GetId returns the unique integer identifier for the pfcModelItem.

The pfcModelItem::GetType method returns an enumeration object that indicates the model item type of the specified pfcModelItem. See the section Solid Geometry Traversal on page 317 for the list of possible model item types.

The method wfcWModelItem::Hide hides the specified model item in the model tree. Use the method wfcWModelItem::Unhide to unhide the model item in the model tree. Using these methods is equivalent to using the commands **Hide** and **Unhide** in the Creo user interface.

The method wfcWModelItem::IsHidden identifies if the specified model item is hidden.

The method wfcWModelItem::IsZoneFeature checks if the specified model item is a zone feature.

The method wfcWView::GetModelItemFromView returns the handle to the model item pfcModelItem for the specified view.

The method wfcViewModelitem::GetViewFromModelItem returns the handle to the view pfcView for the specified model item.

Duplicating ModelItems

Methods Introduced:

- **pfcBaseSession::AllowDuplicateModelItems**

You can control the creation of ModelItems more than twice for the same Creo item. The method pfcBaseSession::AllowDuplicateModelItems allows you to turn ON or OFF the option to duplicate model items. By default, this option is OFF. To turn the option ON, set the boolean value to FALSE.

Note

If this option is not handled properly on the application side, it can cause memory corruption. Thus, althought you can turn ON and OFF this option as many times as you want, PTC recommends turning ON and OFF this option only once, right after the session is obtained.

Layer Objects

In Creo Object TOOLKIT C++, layers are instances of `ModelItem`. The following sections describe how to get layer objects and the operations you can perform on them.

Getting Layer Objects

Method Introduced:

- **pfcModel::CreateLayer**

The method `pfcModel::CreateLayer` returns a new layer with the name you specify.

See the section Getting ModelItem Objects on page 317 for other methods that can return layer objects.

Layer Operations

Methods Introduced:

- **pfcLayer::GetStatus**
- **pfcLayer::SetStatus**
- **pfcLayer::ListItems**
- **pfcLayer::AddItem**
- **pfcLayer::RemoveItem**
- **pfcLayer::Delete**
- **pfcLayer::CountUnsupportedItems**
- **wfcLayerItem::GetLayers**
- **wfcLayerItem::IsLayerItemVisible**
- **wfcLayerItem::RemoveNoUpdate**
- **wfcLayerItem::AddNoUpdate**

- **wfcWModel::ExecuteLayerRules**
- **wfcWModel::CopyLayerRules**
- **wfcWModel::MatchLayerRules**

Superseded Method:

- **pfcLayer::HasUnsupportedItems**

The methods pfcLayer::GetStatus and pfcLayer::SetStatus enables you to access the display status of a layer. The display status is contained in the enumerated type pfcDisplayStatus and the valid values are:

- `pfcLAYER_NORMAL`—The selected layer is displayed.
- `pfcLAYER_DISPLAY`—The selected layer is displayed.
- `pfcLAYER_BLANK`—The selected layer is displayed.
- `pfcLAYER_HIDDEN`—The selected layer is displayed.

Use the methods `pfcLayer::ListItems`, `pfcLayer::AddItem`, and `pfcLayer::RemoveItem` to control the contents of a layer.

Note

You cannot add the following items to a layer:

- `pfcITEM_SURFACE`,
- `pfcITEM_EDGE`,
- `pfcITEM_COORD_SYS`,
- `pfcITEM_AXIS`,
- `pfcITEM_SIMPREP`,
- `pfcITEM_DTL_SYM_DEFINITION`,
- `pfcITEM_DTL_OLE_OBJECT`,
- `pfcITEM_EXPLODED_STATE`.

For these items the method will throw the exception `pfcXToolkitInvalidType`.

The method `pfcLayer::Delete` removes the layer (but not the items it contains) from the model.

The method `pfcLayer::CountUnsupportedItems` returns the number of item types not supported as a `pfcModelItem` object in the specified layer. This method deprecates the method `pfcLayer::HasUnsupportedItems`.

Use the method `wfcLayerItem::GetLayers` to find all the layers containing in a given layer item . This method supports layers in solid models and in drawings.

The method `wfcLayerItem::IsLayerItemVisible` returns the visibility status for the specified item.

Use the methods `wfcLayerItem::RemoveNoUpdate` and `wfcLayerItem::AddNoUpdate` to remove and add a specified item from the layer, respectively, without updating the model tree.

Use the method `wfcWModel::ExecuteLayerRules` to execute the layer rules on the specified model. The rules must be enabled in the layers to be executed.

The method `wfcWModel::CopyLayerRules` copies the rules from the reference model to the current model for the specified layer. The input arguments are:

- *LayerName*—Specifies the name of an existing layer in both the models. To copy the layer rules, the name of the layer *LayerName* in both the models must be the same.
- *RefModel*— Specifies the reference model from which the layer rules must be copied.

Use the method `wfcWModel::MatchLayerRules` to compare the rules between the current and reference model for the specified layer. The name of the layer *LayerName* in both the models must be the same, for comparing the layer rules.

Layer State

A layer state stores the display state of existing layers and all the items on the hidden item layer of the top-level assembly. You can create and save one or more layer states and switch between them to change the display of the assembly.

Methods Introduced:

- **wfcLayerState::ActivateLayerState**
- **wfcLayerState::AddLayer**
- **wfcLayerState::DeleteLayerState**
- **wfcLayerState::GetDefaultLayer**
- **wfcLayerState::SetDefaultLayer**
- **wfcLayerState::GetLayerData**
- **wfcLayerState::GetLayerStateName**
- **wfcLayerState::HideLayerItem**
- **wfcLayerState::IsLayerItemHidden**

- **wfcLayerState::RemoveLayer**
- **wfcLayerState::UnhideLayerItem**
- **wfcLayerStateData::Create**
- **wfcLayerStateData::GetDisplayStatuses**
- **wfcLayerStateData::GetLayerItems**
- **wfcLayerStateData::SetLayerItems**
- **wfcLayerStateData::GetLayerStateName**
- **wfcLayerStateData::SetDisplayStatuses**
- **wfcLayerStateData::SetLayerStateName**
- **wfcLayerStateData::GetLayers**
- **wfcLayerStateData::SetLayers**
- **wfcWModel::ListLayers**
- **wfcWModel::SaveLayerDisplayStatus**
- **wfcWSolid::ListLayerStates**
- **wfcWSolid::CreateLayerState**
- **wfcWSolid::GetActiveLayerState**
- **wfcWModel::GetLayerItem**
- **wfcWSolid::UpdateActiveLayerState**

Use the method `wfcLayerState::ActivateLayerState` to activate the specified layer state. The model of the layer state must be the top model in the active window.

The method `wfcLayerState::AddLayer` adds a new layer to an existing layer state. Specify the new layer and its display state as input arguments to this method.

Use the method `wfcLayerState::DeleteLayerState` to delete a specified layer state.

Use the methods `wfcLayerState::GetDefaultLayer` and `wfcLayerState::SetDefaultLayer` to set up a default layer with a specified name and type respectively. The method `wfcLayerState::SetDefaultLayer` requires the default layer type, which is defined in the enumerated type `wfcDefLayerType`.

The method `wfcLayerState::GetLayerData` retrieves the reference data for a specified layer state.

The method `wfcLayerState::GetLayerStateName` retrieves the name of a specified layer state.

Use the method `wfcLayerState::HideLayerItem` to hide a specific item on the specified layer state.

Use the method `wfcLayerState::UnhideLayerItem` to remove a specific item from the list of hidden items on a layer state.

Use the method `wfcLayerState::IsLayerItemHidden` to check if an item is hidden on a layer state.

The method `wfcLayerState::RemoveLayer` removes a specific layer from a specified layer state.

Use the method `wfcLayerStateData::Create` to create a new instance of the `wfcLayerStateData` object. This object contains information about layer state data.

The methods `wfcLayerStateData::GetDisplayStatuses` and `wfcLayerStateData::SetDisplayStatuses`get and set, respectively, the display statuses for the layers in the layer state. The number of display statuses is equal to the number of layers present in the layer state.

The method `wfcWModel::SaveLayerDisplayStatus` saves the changes to the display status of the layers in the specified model and its sub models. For drawings, the display status is saved in the views contained in the drawing.

The methods `wfcLayerStateData::GetLayerItems` and `wfcLayerStateData::SetLayerItems` get and set the value of the specified layer item.

The methods `wfcLayerStateData::GetLayerStateName` and `wfcLayerStateData::SetLayerStateName` get and set the name of the new layer state. The name must only consist of alphanumeric, underscore, and hypen characters.

The methods `wfcLayerStateData::GetLayers` and `wfcLayerStateData::SetLayers` get and set the layer state for an array of layers.

The method`wfcWModel::ListLayers` returns an array containing the layers in the model.

The method `wfcWSolid::ListLayerStates` returns an array of layer states in the specified model.

The method `wfcWSolid::CreateLayerState` creates a new layer state.

The method `wfcWSolid::GetActiveLayerState` retrieves the active layer state in a specified solid model.

Use the method `wfcWModel::GetLayerItem` to initialize a layer item structure. The valid layer item types are defined by the enumerated type `wfcLayerType`. Do not use this function if the layer item owner is a drawing.

The method `wfcWSolid::UpdateActiveLayerState` updates the layer state, which is active in the specified model. If the display statues of layers have changed, then calling this function ensures that the active layer state in the model is updated with the new display statuses of the layers.

17

Feature Element Tree

This chapter explains feature creation in Creo Object TOOLKIT C++.

Overview of Feature Creation

There are many kinds of features in Creo Parametric and each feature can contain a large and varied amount of information. All this information must be complete and consistent before a feature can be used in regeneration and create the geometry.

You must build all the information needed by a feature into a data structure before passing that whole structure to Creo Parametric. This structure is called Feature Element Tree (FET). The FET structure is in the form of a tree containing the data elements. Creo Object TOOLKIT C++ defines this structure as an object that can be allocated and filled using special classes.

You must use the following steps to create a feature in Creo Parametric:

1. Allocate the FET structure as `wfcElementTree`.
2. Fill the FET structure by creating `wfcElement` objects.
3. Pass the FET structure to Creo Parametric to create the feature by calling the function `wfcWSolid::WCreateFeature`.

The feature is created in a sequence of manageable steps with the error checking along the way.

The full FET is represented by a `wfcElementTree` object. The root and branch points in FET are called “elements”. Each element is modeled by `wfcElement` class.

FET contains all the information required to define the feature. It includes the following information:

- All the options and attributes. For example, the material side and depth type for an extrusion or slot, placement method for a hole, and so on.
- All the references to existing geometry items. For example, the placement references, up to surfaces, sketching planes, and so on.
- All the references to Sketcher models used for sections in the feature.
- All dimension values.

The values of dimensions used by the feature are in the FET. However, there are no descriptions or references to the dimension objects themselves.

Each element in the FET is assigned an element ID. The element ID is an unique to every element. No two elements at the same level in the tree can have the same identifier, unless they belong to an array element.

You cannot create all feature types using Creo Object TOOLKIT C++, but the FET structure is capable of defining any feature type. This allows you to extend the range of features.

Note

The Creo Object TOOLKIT C++ is based on the same toolkit that is used to build Creo Parametric. Changes in Creo Parametric may require the definition of the element tree to be altered for some features. PTC will support upward compatibility in most of the cases. However, there may be cases where the old application will not run with the new version of Creo Parametric. You must rewrite the application's code to conform to the new definition of the feature tree.

The Creo Object TOOLKIT C++ and Creo Object TOOLKIT C++ can be used together.

They share the same definitions of FET element IDs and values. To use a specific element, you must refer to

the header files in `<creo_toolkit_loadpoint>/includes` for feature-specific element trees, then insert the actual integer value of the constant which defines the element ID or the element value.

- `PRO_E_FEATURE_TYPE`
- `PRO_E_FEATURE_FORM`
- `PRO_E_EXT_DEPTH`
- `PRO_E_THICKNESS`
- `PRO_E_4AXIS_PLANE`

Element Tree Types

There are four different element types:

- Single-valued
- Multi-valued
- Compound
- Array

A single-valued element can contain various types of value, for example integer, string, double, and so on. The simplest is an integer. An integer can be used to define the type of the feature, or one of the option choices, such as, the material side for a thin protrusion. The wide string can be used define the name of the feature, and a double can be used to define the depth of a blind extrusion. If the element defines a reference to an existing geometry item in the solid, its value contains an entire `pfcSelection` object that allows it to refer to anything in an

entire assembly. If the element represents a collection, its values contain an entire `wfcCollection` object. The collection can be a curve collection or a surface collection.

A multi-valued element contains several values of one of the mentioned types. Multi-valued elements occur at the lowest level of the element tree at the “leaves”.

A compound element is the one that acts as a branch point in the tree. It does not have a value of its own, but acts as a container for elements further down in the hierarchy.

An array element is also a branch point, but one that contains many child elements with the same element ID.

Building Features Using Element Trees

The feature element tree allows you to build a complex feature in stages, with only a small set of functions. However, the form of the tree required for a particular feature needs to be clearly defined. This helps you identify what elements and values must be added. This also helps Creo Object TOOLKIT C++ can check for errors each time you add a new element to the tree.

The header files in `<creo_toolkit_loadpoint>/includes` describe the Feature Element Trees with the following two types of description:

- Feature element tree
- Feature element table

The feature element tree defines the structure of the tree, specifying the element ID (or role) for the elements at all levels in the tree. It also defines which elements are optional. The feature element table defines the following for each of the element IDs in the tree:

- A description of its role in the feature
- The value type it has (that is, whether it is single value or compound; or an array of integer, double, or string)
- The range of values valid for it in this context

Feature Element Values

Methods Introduced:

- **wfcElement::GetValue**
- **wfcElement::SetValue**

The element values are represented by pfcArgValue objects. They can be set and obtained using the methods wfcElement::GetValue and wfcElement::SetValue methods. For more information on pfcArgValue objects, refer to the section Managing Application Arguments on page 596 in Task Based Application Libraries on page 595.

Feature Element Special Values

Methods Introduced:

- **wfcSpecialValue::GetComponentModel**
- **wfcSpecialValue::SetComponentModel**
- **wfcSpecialValue::GetSectionValue**
- **wfcSpecialValue::SetSectionValue**

The method wfcSpecialValue::GetComponentModel returns the value of the element PRO_E_COMPONENT_MODEL for the specified feature. Use the method wfcSpecialValue::SetComponentModel to set the value for the element PRO_E_COMPONENT_MODEL.

The method wfcSpecialValue::GetSectionValue returns the value of the element PRO_E_SKETCHER for the specified feature. Use the method wfcSpecialValue::SetSectionValue to set the value for the element PRO_E_SKETCHER. This value is a object of type wfcSection.

Feature Element Paths

Methods Introduced:

- **wfcElementPath::GetItems**
- **wfcElementPath::SetItems**

An element path is used to describe the location of an element in an element tree. The full path is represented by the class wfcElementPath, which contains the list of wfcElemPathItem objects. Each wfcElemPathItem provides the element ID and its wfcElemPathItemType. The element path from an array element to one of its member arrays contains the array index of that element. The enumerated type wfcElemPathItemType gives the array index. To get the path length, use the method wfcElementPath::GetItems, and then use wfcElemPathItems::getarraysize. Use to set path length using the method wfcElementPath::SetItems path length.

Feature Element Tree

Methods Introduced:

- **wfcElementTree::ListTreeElements**
- **wfcElementTree::GetElement**
- **wfcElementTree::IsElementArray**
- **wfcElementTree::IsElementCompound**
- **wfcElementTree::IsElementMultiVal**
- **wfcElementTree::CreateDtmCsysElemTreeFromFile**
- **wfcElementTree::WriteElementTreeToFile**
- **wfcWSession::CreateElementTreeFromXML**
- **wfcWSession::CreateElementTree**
- **wfcWFeature::GetElementTree**

The Feature Element Tree (FET) is represented by `wfcElementTree` object. This class has methods that can obtain the list of elements in the element tree or obtain a specific feature element by its path, as well as querying element type (array, compound, and multi-valued).

Note

This type is not stored within `wfcElement` object. It is the property of `wfcElementTree`.

The method `wfcElementTree::CreateDtmCsysElemTreeFromFile` allocates required steps of the element tree to create coordinate system from a transformation file.

The input argument *filename* should be name of the file with the extension .trf. The name must be in lowercase only. The file should contain a coordinate transform such as:

```
X1 X2 X3 Tx
Y1 Y2 Y3 Ty
Z1 Z2 Z3 Tz
```

where

- X1 Y1 Z1 is the X-axis direction,
- X2 Y2 Z2 is the Y-axis direction,
- X3 Y3 Z3 is not used (the right hand rule determines the Z direction),
- Tx Ty Tz is the origin of the coordinate system.

Use the method `wfcElementTree::WriteElementTreeToFile` to save the full FET to a file.

Use the method `wfcWSession::CreateElementTreeFromXML` to build the FET from an XML file. The method `wfcWSession::CreateElementTree` builds the element tree from start. If the FET is built from start, all the mandatory elements in the element tree must to be populated and added sequentially in the sequence `wfcElements`.

The method `wfcWFeature::GetElementTree` creates a copy of the feature element tree that describes the contents of a specified feature. The specified feature can be a regular feature or a pattern.

Creating FET Using WCreateFeature

Methods Introduced:

- **wfcWSolid::WCreateFeature**
- **wfcWSelection::CreateFeature**

The `wfcWSolid` object identifies the solid that is to contain the new feature. The method `wfcWSolid::WCreateFeature` creates a feature from the FET.

Note

This method cannot be used to create a feature in the component of an assembly. Use the method `wfcWSelection::CreateFeature` to create a feature in an assembly component.

The syntax of `wfcWSolid::WCreateFeature` is as follows:

```
wfcWFeature_ptr wfcWSolid::WCreateFeature (
   wfcElementTree_ptr Tree,
   wfcFeatCreateOptions_ptr Options,
   optional wfcWRegenInstructions_ptr Instrs);
```

When using the method `wfcWSolid::WCreateFeature` while working with a multi-CAD model, the following scenarios are possible depending on the value of the configuration option `confirm_on_edit_foreign_models`. The default value of the configuration option `confirm_on_edit_foreign_models` is *yes*.

- If the configuration option `confirm_on_edit_foreign_models` is set to *no*, the non-Creo model is modified without any notification.
- If the configuration option `confirm_on_edit_foreign_models` is set to *yes*, or the option is not defined in the configuration file, then in batch mode the application will throw the exception `pfcXToolkitGeneralError`.

- In some situations you may need to provide input in the interactive mode with Creo. Refer to the Creo Parametric Data Exchange online help, for more information.

The method `wfcWSelection::CreateFeature` creates a feature from the Feature Element Tree. Use this method to create a feature in an assembly component. The input arguments are:

- *Tree*— Specifies the Feature Element Tree object.
- *Options*— Specifies the options which must be used to create the feature. The options are specified using the enumerated data type `wfcFeatCreateOption`.
- *Instrs*— Specifies the regeneration instructions as `wfcRegenInstructions` object. Refer to the section Regenerating a Solid on page 207 for more information on regeneration instructions.

Examples of Feature Creation

The folder `<creo_otk_loadpoint_app>\otk_examples\otk_examples_feat` contains `OTKXWriteFeatAsCxx.cxx`, which provides the generator of C++ code for feature creation. To generate the code for a specific feature, you must create a model with that feature, start Creo Parametric with `otk_examples`, open the model and use Save Feature as cxx or Save All Features as cxx option.

Feature Elements

Methods Introduced:

- **wfcWFeature::GetDimensionId**
- **wfcWSession::GetElemWstrOption**
- **wfcWSession::SetElemWstrOption**
- **wfcElementWstringOption::Create**
- **wfcElementWstringOption::GetExpression**
- **wfcElementWstringOption::SetExpression**
- **wfcElementWstringOption::GetPositive**
- **wfcElementWstringOption::SetPositive**
- **wfcElementWstringOption::GetSign**
- **wfcElementWstringOption::SetSign**
- **wfcElement::Create**
- **wfcElement::GetIdAsString**

- **wfcElement::GetIsArray**
- **wfcElement::GetIsCompound**
- **wfcElement::GetIsMultival**
- **wfcElement::GetChildren**
- **wfcElement::GetValueAsString**
- **wfcElement::SetValueAsString**
- **wfcElement::GetId**
- **wfcElement::SetId**
- **wfcElement::GetLevel**
- **wfcElement::SetLevel**
- **wfcElement::GetDecimals**
- **wfcElement::SetDecimals**
- **wfcElement::GetSpecialValueElem**
- **wfcElement::SetSpecialValueElem**
- **wfcElement::GetElemCollection**
- **wfcElement::SetElemCollection**

The method `wfcWFeature::GetDimensionId` returns the integer identifier of the dimension in the Creo Parametric database used to define the value of the specified single-valued element.

The methods `wfcWSession::GetElemWstrOption` and `wfcWSession::SetElemWstrOption` get and set the options used to retrieve the string values of elements. The options set in this method are used by the method `wfcElement::GetValueAsString` to display the string representation of elements.

The method `wfcElementWstringOption::Create` creates a string value of a specified element in a tree.

The method `wfcElementWstringOption::SetExpression` sets the option to retrieve values as expressions or relations, if they exist, instead of string representations of the actual value. This method is applicable only to double value elements.

The method `wfcElementWstringOption::SetPositive` sets the option to retrieve the values as positive. This method is applicable to double and integer value elements.

The method `wfcElementWstringOption::SetSign` sets the option to retrieve values with special sign formatting (+/-), etc. This method is applicable to both double and integer value elements.

The method `wfcElement::Create` creates a new instance of the `wfcElement` object that contains information about the parameters of the element.

The method `wfcElement::GetIdAsString` returns the string representation of the specified element ID.

The methods `wfcElement::GetIsArray` and `wfcElement::GetIsCompound` determine if the specified element contains an array of elements, or is a compound. The methods `wfcElement::GetIsArray`, `wfcElement::GetIsCompound`, and `wfcElement::GetIsMultival` are used to determine the type of the specified element in a tree. The method `wfcElement::GetIsMultival` determines whether the input element can have multiple values.

The method `wfcElement::GetChildren` populates an array of children elements, if the specified element is a compound element, or an array.

The method `wfcElement::GetValueAsString` returns a string value representation for double and integer elements. The options set in the object `wfcElementWstringOption` decide the format of the output.

Use the methods `wfcElement::GetId` and `wfcElement::SetId` to get and set the element identifier for the specified element.

Use the methods `wfcElement::GetLevel` and `wfcElement::SetLevel` to get and set the location of the element in the element tree with respect to the root element.

Use the methods `wfcElement::GetDecimals` and `wfcElement::SetDecimals` obtain and set the number of decimal places to be used for the double value of an element in the feature.

Use the method `wfcElement::GetSpecialValueElem` to obtain the pointer representation for the specified element. The method `wfcElement::SetSpecialValueElem` sets the pointer representation for the specified element.

Use the method `wfcElement::GetElemCollection` to extract a collection object from an element of a feature element tree of the following types:

- `wfcCurveCollection`
- `wfcSurfaceCollection`

Use the method `wfcElement::SetElemCollection` to assign a collection object to an element of a feature element tree of the type `wfcCurveCollection` and `wfcSurfaceCollection`.

Creating Patterns

Methods Introduced:

- **wfcWFeature::CreatePattern**

You can create patterns by calling the method `wfcWFeature::CreatePattern` on the feature.

Redefining Features

Method Introduced:

- **wfcWFeature::RedefineFeature**

You can use the method `wfcWFeature::RedefineFeature` to redefine features.

Element Diagnostics

Methods Introduced:

- **wfcElement::GetDiagnostics**
- **wfcElementDiagnostic::Create**
- **wfcElementDiagnostic::GetDiagnosticMessage**
- **wfcElementDiagnostic::SetDiagnosticMessage**
- **wfcElementDiagnostic::GetSeverity**
- **wfcElementDiagnostic::SetSeverity**

The method `wfcElement::GetDiagnostics` collects the element diagnostics. The diagnostics include warnings and errors about the value of the element within the context of the feature and the remainder of the element tree.

Use the method `wfcElementDiagnostic::Create` to create an element diagnostic in an element tree.

The methods `wfcElementDiagnostic::GetDiagnosticMessage` and `wfcElementDiagnostic::GetSeverity` get the message and severity of the diagnostic item of the element.

18

Element Trees: Sections

A section is a parametric two-dimensional cross section used to define the shape of three-dimensional features, such as extrusions. In the Creo application, you create a section interactively using Sketcher mode. In a Creo Object TOOLKIT C++ application, you can create sections completely programmatically using the methods described in this section.

Overview

Sections fall into two types: 2D and 3D. Both types are represented by the `wfcSection` object and manipulated by the same methods.

The difference between the types arises out of the context in which the section is being used, and affects the requirements for the contents of the section and also of the feature element tree in which it is placed when creating a sketched feature.

Put simply, a 2D section is self-contained, whereas a 3D section contains references to 3D geometry in a parent part or assembly.

You can add section entities programmatically using the Intent Manager mode. This corresponds to creating sections within the Intent Manager mode in the Creo application.

This chapter is concerned with 2D. The extra steps required to construct a 3D section are described in the chapter Element Trees: Sketched Features on page 350.

Creating Section Models

A 2D section, because it is self-contained, can be stored as a Creo model file. It then has the extension `.sec`.

The steps required to create and save a section model using Creo Object TOOLKIT C++ follow closely those used in creating a section interactively using Sketcher mode in the Creo application.

Setting the Intent Manager Mode of a Section

Methods Introduced:

- **wfcSection::GetIntentManagerMode**
- **wfcSection::SetIntentManagerMode**

The Creo Object TOOLKIT C++ methods for 2D and 3D sections work only when the Intent Manager mode is set to ON. Use the method `wfcSection::GetIntentManagerMode` to check if the Intent Manager mode is ON for the specified section. The mode is set to OFF by default.

Use the method `wfcSection::SetIntentManagerMode` to set the Intent Manager mode to ON for the specified section. Call this method before using the other Creo Object TOOLKIT C++ methods that access the sections.

Example 1: Creating a Section Model

The sample code in `OTKXCreateSection2D.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_model` illustrates how to use all the methods described in this chapter to create a section model.

To Create and Save a Section Model

1. Allocate the two-dimensional section and define its name.
2. Set the Intent manager mode to ON using the method `wfcSection::SetIntentManagerMode`.
3. Add section entities (lines, arcs, splines, and so on) to define the section geometry, in section coordinates.
4. Save the section.

When you are creating a section that is to be used in a sketched feature, Steps 1 and 4 will be replaced by different techniques. These techniques are described fully in the chapter on Element Trees: Sketched Features on page 350.

The steps are described in more detail in the following sections.

Allocating a Two-Dimensional Section

Methods Introduced:

- **wfcWSession::CreateSection2D**
- **wfcSection::GetName**
- **wfcSection::SetName**

The method `wfcWSession::CreateSection2D` allocates memory for a new, standalone two dimensional section and outputs a section handle to identify it.

The method `wfcSection::SetName` enables you to set the name of a section. Calling this method places the section in the Creo application namelist and enables it to be recognized by the Creo application as a section model in the database.

Such sections created programmatically have the Intent Manager mode OFF by default.

Copying the Current Section

Methods Introduced:

- **wfcWSession::GetActiveSection**
- **wfcSection::SetActive**

The method `wfcWSession::GetActiveSection` creates a copy of the section that you are using currently. This copy is created within the same Sketcher session. To set the Intent Manager mode to ON, call the method `wfcSection::SetIntentManagerMode` after this method.

Use the method `wfcSection::SetActive` to set the specified section as the current active Sketcher section. The Intent Manager mode must be set to ON when you call this method.

Note

The call to the method `wfcSection::SetActive` makes the **Undo** and **Redo** menu options available in Creo Parametric.

Epsilon Value in Sections

Methods Introduced:

- **wfcSection::GetEpsilon**
- **wfcSection::SetEpsilon**

Epsilon is the tolerance value, which is used to set the proximity for automatic finding of constraints. Use the function `wfcSection::SetEpsilon` to set the value for epsilon. For example, if your section has two lines that differ in length by 0.5, set the epsilon to a value less than 0.5 to ensure that the two lines are not constrained as same length. To get the current epsilon value for the section, use the function `wfcSection::GetEpsilon`.

Please note the following important points related to epsilon:

- Epsilon determines the smallest possible entity in a section. If an entity is smaller than epsilon, then the entity is considered to be a degenerate entity. Degenerate entity is an entity which cannot be solved. It causes solving and regenerating of the section to fail. For example, a circle with radius 0 or line with length 0 are considered as degenerate entities.
- There are many types of constraints, and epsilon has a different meaning for each type. For example, consider two points. In case of constraint for coincident points, , epsilon is the minimum distance between the two points beyond which the points will be treated as separate points. If the distance between the two points is within the epsilon value, the two points are treated as coincident points.
- Creo Parametric has a default value set for epsilon. This value is also used in the Sketcher user interface.
- If the input geometry is accurate and the user does not want the solver to change it by adding constraints, then set the value of epsilon to 1E-9.

- If the input geometry is nearly accurate and the user wants the solver to guess the intent by adding constraints and further aligning the geometry, then in this case epsilon should reflect the maximal proximity between geometry to be constrained.
- You cannot set the value of epsilon to zero.

Section Entities

Methods Introduced:

- **wfcSection::AddEntity**
- **wfcSection::DeleteEntity**
- **wfcSection::GetEntity**
- **wfcSection::ListSectionEntities**
- **wfcSection::GetEntityIds**
- **wfcSectionEntity::GetSectionEntityType**

The method `wfcSection::AddEntity` takes as input the `wfcSectionEntity` object that defines the section entity type using the enumerated type `wfcSection2dEntType`. The following types of entities are defined:

- `wfcSEC_ENTITY_2D_POINT`
- `wfcSEC_ENTITY_2D_LINE`
- `wfcSEC_ENTITY_2D_CENTER_LINE`
- `wfcSEC_ENTITY_2D_ARC`
- `wfcSEC_ENTITY_2D_CIRCLE`
- `wfcSEC_ENTITY_2D_COORD_SYS`
- `wfcSEC_ENTITY_2D_POLYLINE`
- `wfcSEC_ENTITY_2D_SPLINE`
- `wfcSEC_ENTITY_2D_TEXT`
- `wfcSEC_ENTITY_2D_CONSTR_CIRCLE`
- `wfcSEC_ENTITY_2D_BLEND_VERTEX`
- `wfcSEC_ENTITY_2D_ELLIPSE`
- `wfcSEC_ENTITY_2D_CONIC`
- `wfcSEC_ENTITY_2D_SEC_GROUP`

Some classes in Creo Object TOOLKIT C++ allow you to create and modify various types of section entities. The class `wfcSectionEntity` is the parent class for the following entity classes:

- `wfcSectionEntityArc`

- wfcSectionEntityBlendVertex
- wfcSectionEntityCSys
- wfcSectionEntityCenterLine
- wfcSectionEntityCircle
- wfcSectionEntityConic
- wfcSectionEntityEllipse
- wfcSectionEntityLine
- wfcSectionEntityPoint
- wfcSectionEntityPolyline
- wfcSectionEntitySpline
- wfcSectionEntityText

The method wfcSection::AddEntity outputs an integer that is the identifier of the new entity within the section. The Creo Object TOOLKIT C++ application needs these values because they are used to refer to entities when adding dimensions.

The method wfcSection::DeleteEntity enables you to delete a section entity from the specified section.

The method wfcSection::GetEntity takes as input the integer identifier for a section entity and outputs a copy of the section entity object.

Use the method wfcSection::ListSectionEntities to retrieve the list of entities present in the specified section.

The method wfcSection::GetEntityIds returns the array of integer identifiers for the all the entities in the specified section.

The method wfcSectionEntity::GetSectionEntityType returns the section entity type using the enumerated type wfcSection2dEntType.

Section Entity Arc

Methods Introduced:

- **wfcSectionEntityArc::Create**
- **wfcSectionEntityArc::GetCenter**
- **wfcSectionEntityArc::SetCenter**
- **wfcSectionEntityArc::GetEndAngle**
- **wfcSectionEntityArc::SetEndAngle**
- **wfcSectionEntityArc::GetRadius**
- **wfcSectionEntityArc::SetRadius**
- **wfcSectionEntityArc::GetStartAngle**
- **wfcSectionEntityArc::SetStartAngle**

The method `wfcSectionEntityArc::Create` creates an arc entity in a specified section using the center, start and end angles, and radius of the arc as inputs.

The methods `wfcSectionEntityArc::GetCenter` and `wfcSectionEntityArc::SetCenter` retrieve and set the center of the arc using the `pfcPoint2D` object.

The methods `wfcSectionEntityArc::GetEndAngle` and `wfcSectionEntityArc::SetEndAngle` retrieve and set the end angle of the arc.

The methods `wfcSectionEntityArc::GetRadius` and `wfcSectionEntityArc::SetRadius` retrieve and set the radius of the arc.

The methods `wfcSectionEntityArc::GetStartAngle` and `wfcSectionEntityArc::SetStartAngle` retrieve and set the start angle of the arc.

Section Entity Blend Vertex

Methods Introduced:

- **wfcSectionEntityBlendVertex::Create**
- **wfcSectionEntityBlendVertex::GetDepthLevel**
- **wfcSectionEntityBlendVertex::SetDepthLevel**
- **wfcSectionEntityBlendVertex::GetPoint**
- **wfcSectionEntityBlendVertex::SetPoint**

The method `wfcSectionEntityBlendVertex::Create` creates a blend vertex entity in a specified section using the point and depth level of the blend vertex as inputs.

The methods `wfcSectionEntityBlendVertex::GetDepthLevel` and `wfcSectionEntityBlendVertex::SetDepthLevel` retrieve and set the depth level of the blend.

The methods `wfcSectionEntityBlendVertex::GetPoint` and `wfcSectionEntityBlendVertex::SetPoint` retrieve and set the points of the blend vertices using the `pfcPoint2D` object.

Section Entity Coordinate System

Methods Introduced:

- **wfcSectionEntityCSys::Create**
- **wfcSectionEntityCSys::GetCSysPoint**
- **wfcSectionEntityCSys::SetCSysPoint**

The method `wfcSectionEntityCSys::Create` creates a coordinate system entity in a specified section using the center point of the coordinate system as input.

The method `wfcSectionEntityCSys::GetCSysPoint` and `wfcSectionEntityCSys::SetCSysPoint` retrieve and set the center point for the coordinate system using the `pfcPoint2D` object.

Section Entity CenterLine

Methods Introduced:

- **wfcSectionEntityCenterLine::Create**
- **wfcSectionEntityCenterLine::GetCenterLine**
- **wfcSectionEntityCenterLine::SetCenterLine**

The method `wfcSectionEntityCenterLine::Create` creates a centerline entity in a specified section.

The methods `wfcSectionEntityCenterLine::GetCenterLine` and `wfcSectionEntityCenterLine::SetCenterLine` retrieve and set the centerline using the `pfcOutline2D` object.

Section Entity Circle

Methods Introduced:

- **wfcSectionEntityCircle::Create**
- **wfcSectionEntityCircle::GetCenter**
- **wfcSectionEntityCircle::SetCenter**
- **wfcSectionEntityCircle::GetRadius**
- **wfcSectionEntityCircle::SetRadius**

The method `wfcSectionEntityCircle::Create` creates a circle entity in a specified section using the center and radius of the circle as inputs.

The methods `wfcSectionEntityCircle::GetCenter` and `wfcSectionEntityCircle::SetCenter` retrieve and set the center of the circle using the `pfcPoint2D` object.

The methods `wfcSectionEntityCircle::GetRadius` and `wfcSectionEntityCircle::SetRadius` retrieve and set the radius of the circle.

Section Entity Ellipse

Methods Introduced:

- **wfcSectionEntityEllipse::Create**

- **wfcSectionEntityEllipse::GetCenter**
- **wfcSectionEntityEllipse::SetCenter**
- **wfcSectionEntityEllipse::GetXRadius**
- **wfcSectionEntityEllipse::SetXRadius**
- **wfcSectionEntityEllipse::GetYRadius**
- **wfcSectionEntityEllipse::SetYRadius**

The method `wfcSectionEntityEllipse::Create` creates an ellipse entity in a specified section using the center, X-axis and Y-axis radii of the eclipse as inputs.

The methods `wfcSectionEntityEllipse::GetCenter` and `wfcSectionEntityEllipse::SetCenter` retrieve and set the center of the ellipse.

The methods `wfcSectionEntityEllipse::GetXRadius` and `wfcSectionEntityEllipse::GetXRadius` retrieve and set the XRadius of the ellipse.

The methods `wfcSectionEntityEllipse::GetYRadius` and `wfcSectionEntityEllipse::SetYRadius` retrieve and set the YRadius of the ellipse.

Section Entity Conic

Methods Introduced:

- **wfcSectionEntityConic::Create**
- **wfcSectionEntityConic::GetFirstEndPoint**
- **wfcSectionEntityConic::SetFirstEndPoint**
- **wfcSectionEntityConic::GetSecondEndPoint**
- **wfcSectionEntityConic::SetSecondEndPoint**
- **wfcSectionEntityConic::GetParameter**
- **wfcSectionEntityConic::SetParameter**
- **wfcSectionEntityConic::GetShoulderEndPoint**
- **wfcSectionEntityConic::SetShoulderEndPoint**

The method `wfcSectionEntityConic::Create` creates a conic entity in a specified section using the first, second, and shoulder endpoints and parameter of the cone as inputs.

The methods `wfcSectionEntityConic::GetFirstEndPoint` and `wfcSectionEntityConic::SetFirstEndPoint` retrieve and set the first endpoint of the conic entity.

The methods `wfcSectionEntityConic::GetSecondEndPoint` and `wfcSectionEntityConic::SetSecondEndPoint` retrieve and set the second endpoint of the conic entity.

The methods `wfcSectionEntityConic::GetParameter` and `wfcSectionEntityConic::SetParameter` retrieve and set the parameter of the conic entity.

The methods `wfcSectionEntityConic::GetShoulderEndPoint` and `wfcSectionEntityConic::SetShoulderEndPoint` retrieve and set the shoulder endpoint of the conic entity.

Section Entity Line

Methods Introduced:

- **wfcSectionEntityLine::Create**
- **wfcSectionEntityLine::GetLine**
- **wfcSectionEntityLine::SetLine**

The method `wfcSectionEntityLine::Create` creates a line entity in a specified section.

The methods `wfcSectionEntityLine::GetLine` and `wfcSectionEntityLine::SetLine` retrieve and set the lines in a specified section using the `pfcOutline2D` object.

Section Entity Point

Methods Introduced:

- **wfcSectionEntityPoint::Create**
- **wfcSectionEntityPoint::GetPoint**
- **wfcSectionEntityPoint::SetPoint**

The method `wfcSectionEntityPoint::Create` creates a point entity in a section.

The methods `wfcSectionEntityPoint::GetPoint` and `wfcSectionEntityPoint::SetPoint` retrieve and set the points in a specified section using the `pfcPoint2D` object.

Section Entity Polyline

Methods Introduced:

- **wfcSectionEntityPolyline::Create**
- **wfcSectionEntityPolyline::GetPoints**
- **wfcSectionEntityPolyline::SetPoints**

The method `wfcSectionEntityPolyline::Create` creates a polyline entity in a specified section.

The methods `wfcSectionEntityPolyline::GetPoints` and `wfcSectionEntityPolyline::SetPoints` retrieve and set the points using the `pfcPoint2Ds` object.

Section Entity Spline

Methods Introduced:

- **wfcSectionEntitySpline::Create**
- **wfcSectionEntitySpline::GetTangentType**
- **wfcSectionEntitySpline::SetTangentType**
- **wfcSectionEntitySpline::GetPoints**
- **wfcSectionEntitySpline::SetPoints**
- **wfcSectionEntitySpline::GetStartTangentAngle**
- **wfcSectionEntitySpline::SetStartTangentAngle**
- **wfcSectionEntitySpline::GetEndTangentAngle**
- **wfcSectionEntitySpline::SetEndTangentAngle**

The method `wfcSectionEntitySpline::Create` creates a spline entity in a specified section using the tangent type, points, and start and end angles of the tangent of the spline as inputs.

The methods `wfcSectionEntitySpline::GetTangentType` and `wfcSectionEntitySpline::SetTangentType` retrieve and set the type of tangent in the spline using the `wfcSplineTangentType` object.

The methods `wfcSectionEntitySpline::GetPoints` and `wfcSectionEntitySpline::SetPoints` retrieve and set the points in the spline using the `pfcPoint2Ds` object.

The methods `wfcSectionEntitySpline::GetStartTangentAngle` and `wfcSectionEntitySpline::SetStartTangentAngle` retrieve and set the start angle of a tangent in the spline.

The methods `wfcSectionEntitySpline::GetEndTangentAngle` and `wfcSectionEntitySpline::SetEndTangentAngle` retrieve and set the end angle of a tangent in the spline.

Section Entity Text

Methods Introduced:

- **wfcSectionEntityText::Create**
- **wfcSectionEntityText::GetFirstCorner**
- **wfcSectionEntityText::SetFirstCorner**

- **wfcSectionEntityText::GetSecondCorner**
- **wfcSectionEntityText::SetSecondCorner**
- **wfcSectionEntityText::GetFontName**
- **wfcSectionEntityText::SetFontName**
- **wfcSectionEntityText::GetComment**
- **wfcSectionEntityText::SetComment**

The method `wfcSectionEntityText::Create` creates a text entity in a specified section using the first and second corner of the text box and the text to be entered inside the text box as inputs.

The methods `wfcSectionEntityText::GetFirstCorner` and `wfcSectionEntityText::SetFirstCorner` retrieve and set the first corner of the text box using the `pfcPoint2D` class.

The methods `wfcSectionEntityText::GetSecondCorner` and `wfcSectionEntityText::SetSecondCorner` retrieve and set the second corner of the text box using the `pfcPoint2D` class.

The methods `wfcSectionEntityText::GetFontName` and `wfcSectionEntityText::SetFontName` retrieve and set the font of the text to be entered.

The methods `wfcSectionEntityText::GetComment` and `wfcSectionEntityText::SetComment` retrieve and set the comment or text to be entered inside the text box.

Retrieving a Section

Method Introduced:

- **wfcWFeature::GetSections**
- **wfcSection::GetSectionDimensions**
- **wfcSectionDimIds::GetSolidIds**
- **wfcSectionDimIds::GetSectionIds**

The method `wfcWFeature::GetSections` retrieve sections from the specified feature.

Note

The method will not return sections that are not available for use. For example, the selected trajectory in a Sweep feature will not be returned.

The method `wfcSection::GetSectionDimensions` returns corresponding arrays of section dimension identifiers and solid dimension identifiers using the class `wfcSectionDimIds`.

The method `wfcSectionDimIds::GetSolidIds` returns solid dimension identifiers.

The method `wfcSectionDimIds::GetSectionIds` returns section dimension identifiers.

19

Element Trees: Sketched Features

This chapter describes the Creo Object TOOLKIT C++ methods that enable you to work with sketched features.

Sketched features are features that require one or more sections to completely define the feature, such as extruded and revolved protrusions.

This chapter outlines the necessary steps to programmatically create sketched features using Creo Object TOOLKIT C++.

Overview

The chapter Feature Element Tree on page 326 explains how to create a simple feature using the feature element tree, and the documentation in the chapter Element Trees: Sections on page 337 explains how to create a section. This chapter explains how to put these methods together, with a few additional techniques, to create features that contain sketched sections.

Creating Features Containing Sections

The chapter Feature Element Tree on page 326 explained that to create a feature from an element tree, you must build the tree of elements using the `wfcElementTree` object, and then call `wfcWSolid::WCreateFeature` to create the feature using the tree. If the feature is to contain a sketch, the sequence is a little more complex.

As explained in the chapter Element Trees: Sections on page 337, a 2D section stored in a model file can be allocated by calling `wfcWSession::CreateSection2D`. Instead, the Creo application must allocate as part of the initial creation of the sketched feature, a section that will be part of a feature. The allocation is done by calling `wfcWSolid::WCreateFeature` with an element tree which describes at minimum the feature type and form, in order to create an incomplete feature. In creating the feature, the Creo application calculates the location and orientation of the section, and allocates the `wfcSection` object. This section is then retrieved from the value of the `PRO_E_SKETCHER` element that is found in the element tree extracted from the created feature. Fill the empty section using `wfcSection` related methods.

After adding the section contents and the remaining elements in the tree, add the new information to the feature using `wfcWFeature::RedefineFeature`.

To Create Sketched Features Element Trees

1. Build an element tree but do not include the element `PRO_E_SKETCHER`.
2. Call `wfcWSolid::WCreateFeature` with the option `wfcFEAT_CR_INCOMPLETE_FEAT` to create an incomplete feature.
3. Extract the value of the element `PRO_E_SKETCHER` created by the Creo application from an element tree extracted using `wfcWFeature::GetElementTree` on the incomplete feature.
4. Using that value as the `wfcSection` object create the necessary section entities.

5. Add any other elements not previously added to the tree, such as extrusion depth. The depth elements may also be added before the creation of incomplete feature (before step 2).
6. Call `wfcWFeature::RedefineFeature` with the completed element tree.

Example 1: Creating a Sweep Feature

The sample code in `OTKXCreateSweep.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_feat` illustrates how to create a sweep using the Creo Object TOOLKIT C++ methods.

Creating Features with 2D Sections

Sketched features using 2D sections do not require references to other geometry in the Creo model. Some examples of where 2D sections are used are:

- Base features, sometimes called first features. This type of feature must be the first feature created in the model.
- Sketched hole features.

To create 2D sketched features, follow the steps outlined in the section To Create Sketched Features Element Trees on page 351.

Note

For 2D sketched features, you need not specify section references or use projected 3D entities. Entities in a 2D section are dimensioned to themselves only. A 2D section does not require any elements in the tree to setup the sketch plane or the orientation of the sketch. Thus, the `PRO_E_STD_SEC_SETUP_PLANE` subtree is not included.

Creating Features with 3D Sections

A 3D section needs to define its location with respect to the existing geometrical features. The subtree contained in the element `PRO_STD_SEC_SETUP_PLANE` defines the location of the sketch plane edge entities; any other 2D entities in the sketch must be dimensioned to those entities, so that their 3D location is fully defined.

3D Section Location in the Owning Model

Method Introduced:

- **wfcSection::GetLocation**

The Creo application decides where the section will be positioned in 3D for all the features except the first feature and sketched hole feature.

If the section is 3D, the feature tree elements below `PRO_E_STD_SEC_SETUP_PLANE` specifies the sketch plane, the direction from which it is being viewed, an orientation reference, and a direction which that reference represents (TOP, BOTTOM, LEFT or RIGHT). When you call `wfcWSolid::WCreateFeature`, this information is used to calculate the 3D plane in which the section lies, and its orientation in that plane.

The position of the section origin in the plane is not implied by the element tree, and cannot be specified by the Creo Object TOOLKIT C++ application: position is chosen arbitrarily by Creo application. This is because the interactive user of Creo application never deals in absolute coordinates, and does not need to specify, or even know, the location of the origin of the section. In Creo Object TOOLKIT C++ describe all section entities in terms of their coordinate values, so you need to find out where Creo application has put the origin of the section. This is the role of the method `wfcSection::GetLocation`.

`wfcSection::GetLocation` provides the transformation matrix that goes from 2D coordinates within the section to 3D coordinates of the owning part or assembly. This is equivalent to describing the position and orientation of the 2D section coordinate system with respect to the base coordinate system of the 3D model.

`wfcSection::GetLocation` can be called in order to calculate where to position new section entities so that they are in the correct 3D position in the part or assembly.

Example 2: Manipulating a 3D Section

The sample code in `OTKXCreateSection3D.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_feat` illustrates how to use all the methods described in this chapter to create a section model.

20

Holes

This chapter describes how to access the hole properties in Creo Object TOOLKIT C++.

Accessing Threaded Hole Properties

Methods Introduced:

- **wfcWHoleFeature::GetHoleProperties**
- **wfcWHoleFeature::SetHoleProperties**
- **wfcHoleProperties::Create**
- **wfcHoleProperties::GetThreadSeries**
- **wfcHoleProperties::SetThreadSeries**
- **wfcHoleProperties::GetScrewSize**
- **wfcHoleProperties::SetScrewSize**

The methods `wfcWHoleFeature::GetHoleProperties` and `wfcWHoleFeature::SetHoleProperties` retrieve and set the properties of the specified hole feature using the `wfcHoleProperties` object.

The method `wfcHoleProperties::Create` creates a data object that contains information about the thread series and screw size of a hole.

The method `wfcHoleProperties::GetThreadSeries` and `wfcHoleProperties::SetThreadSeries` returns the type of thread for the specified hole feature.

The methods `wfcHoleProperties::GetScrewSize` and `wfcHoleProperties::SetScrewSize` to get and set the size of screw for the specified hole feature.

Note

The screw size depends on the type of thread. Therefore, before you call the method `wfcHoleProperties::SetScrewSize` you must ensure that the thread type is set for the hole feature.

21

Features

All Creo solid models are made up of features. This chapter describes how to program on the feature level using Creo Object TOOLKIT C++.

The actual type of `pfcSolid` objects is `wfcWSolid` and `pfcFeature` object is `wfcWFeature`. Therefore, the methods from `wfcWFeature` and `wfcWSolid` become available to these objects only after applying `wfcWFeature::cast` or `wfcWSolid::cast`. You must check that this cast does not return a null pointer.

Access to Features

Methods Introduced:

- **pfcFeature::ListChildren**
- **pfcFeature::ListParents**
- **pfcFeature::GetGroupDirectHeader**
- **pfcFeatureGroup::GetGroupLeader**
- **pfcFeaturePattern::GetPatternLeader**
- **pfcFeaturePattern::ListMembers**
- **pfcSolid::ListFailedFeatures**
- **wfcWSolid::ListChildOfExternalFailedFeatures**
- **wfcWSolid::ListChildOfFailedFeatures**
- **pfcSolid::ListFeaturesByType**
- **pfcSolid::GetFeatureById**
- **pfcBaseSession::QueryFeatureEdit**

The methods `pfcFeature::ListChildren` and `pfcFeature::ListParents` return a sequence of features that contain all the children or parents of the specified feature.

The method `pfcFeature::GetGroupDirectHeader` returns the direct header of the group.

To get the first feature in the specified group access the method `pfcFeatureGroup::GetGroupLeader`.

The methods `pfcFeaturePattern::GetPatternLeader` and `pfcFeaturePattern::ListMembers` return features that make up the specified feature pattern. See the section Feature Groups and Patterns on page 365 for more information on feature patterns.

The method `pfcSolid::ListFailedFeatures` returns a sequence that contains all the features that failed regeneration.

The method `wfcWSolid::ListChildOfExternalFailedFeatures` returns a list of elements, where each element is a child of an external failed feature.

The method `wfcWSolid:: ListChildOfFailedFeatures` returns a list of elements, where each element is a child of a failed feature.

The method `pfcSolid::ListFeaturesByType` returns a sequence of features contained in the model. You can specify which type of feature to collect by passing in one of the `pfcFeatureType` enumeration objects, or you can

collect all features by passing `void null` as the type. If you list all features, the resulting sequence will include invisible features that Creo creates internally. Use the method's *VisibleOnly* argument to exclude them.

The method `pfcSolid::GetFeatureById` returns the feature object with the corresponding integer identifier.

A feature can be edited with the **Edit Definition** command in Creo Parametric. The method `pfcBaseSession::QueryFeatureEdit` returns a list of all the features that are currently being edited by the **Edit Definition** command.

Feature Information

Methods Introduced:

- **pfcFeature::GetFeatType**
- **pfcFeature::GetStatus**
- **pfcFeature::GetIsVisible**
- **pfcFeature::GetIsReadonly**
- **pfcFeature::GetIsEmbedded**
- **pfcFeature::GetNumber**
- **pfcFeature::GetFeatTypeName**
- **pfcFeature::GetFeatSubType**
- **pfcRoundFeat::GetIsAutoRoundMember**
- **wfcWFeature::IsElementVisible**
- **wfcWFeature::IsElementIncomplete**
- **wfcWFeature::GetStatusFlag**
- **wfcWSolid::CreateZoneSectionFeature**
- **wfcZoneFeatureReference::Create**
- **wfcZoneFeatureReference::GetPlaneId**
- **wfcZoneFeatureReference::SetPlaneId**
- **wfcZoneFeatureReference::GetOperation**
- **wfcZoneFeatureReference::SetOperation**
- **wfcZoneFeatureReference::GetMemberIdTable**
- **wfcZoneFeatureReference::SetMemberIdTable**
- **wfcZoneFeatureReference::GetFlip**
- **wfcZoneFeatureReference::SetFlip**
- **wfcWFeature::GetZoneFeatureReferences**

- **wfcWFeature::GetZoneFeaturePlaneData**
- **wfcWFeature::GetZoneXSectionGeometry**
- **wfcWFeature::IsInFooter**
- **wfcWFeature::MoveToFooter**
- **wfcWFeature::MoveFromFooter**

The enumeration classes `pfcFeatureType` and `pfcFeatureStatus` provide information for a specified feature. The following methods specify this information:

- `pfcFeature::GetFeatType`—Returns the type of a feature.
- `pfcFeature::GetStatus`—Returns whether the feature is suppressed, active, or failed regeneration.

The other methods that gather feature information include the following:

- `pfcFeature::GetIsVisible`—Identifies whether the specified feature will be visible on the screen. The method distinguishes visible features from internal features. Internal features are invisible features used for construction purposes.
- `pfcFeature::GetIsReadonly`—Identifies whether the specified feature can be modified.
- `pfcFeature::GetIsEmbedded`—Specifies whether the specified feature is an embedded datum.
- `pfcFeature::GetNumber`—Returns the feature regeneration number. This method returns `void` `null` if the feature is suppressed.

The method `pfcFeature::GetFeatTypeName` returns a string representation of the feature type.

The method `pfcFeature::GetFeatSubType` returns a string representation of the feature subtype, for example, "Extrude" for a protrusion feature.

The method `pfcRoundFeat::GetIsAutoRoundMember` determines whether the specified round feature is a member of an Auto Round feature.

The method `wfcWFeature::IsElementVisible` determines whether the specified element is visible.

The method `wfcWFeature::IsElementIncomplete` determines whether the specified element is incomplete. If a feature is incomplete, you can use this method to find out which element in the tree is incomplete.

The method `wfcWFeature::IsElementVisible` retrieves the bit status flag object of the feature.

The method `wfcWFeature::GetStatusFlag` retrieves the bit status flag object of the feature.

The method `wfcWSolid::CreateZoneSectionFeature` creates a zone feature. The input arguments are:

- *RefData*—The references to create the zone feature.
- *ZoneName*—The name of the zone feature.

The method `wfcZoneFeatureReference::Create` creates an object of type `wfcZoneFeatureReference` that contains information about the zone references for the specified feature.

Use the method `wfcZoneFeatureReference::GetPlaneId` to retrieve the geometric ID of the reference zone plane. The method `wfcZoneFeatureReference::SetPlaneId` sets the geometric ID for the reference zone plane.

The method `wfcZoneFeatureReference::GetOperation` gets the value of the operation, where 0 specifies intersection of half spaces that is, the AND operator and 1 specifies union of half spaces that is, the OR operator. Use the method `wfcZoneFeatureReference::SetOperation` to set the value of the operation.

The method `wfcZoneFeatureReference::GetMemberIdTable` returns a sequence of component identifiers that form the path to the part to which the reference plane belongs.

Use the method `wfcZoneFeatureReference::SetMemberIdTable` to set the path to the part for the reference plane.

Note

When the feature is owned by a part, pass the value `NULL`.

The methods `wfcZoneFeatureReference::GetFlip` and `wfcZoneFeatureReference::SetFlip` retrieve and set the side of the plane where the model is kept. True indicates positive normal of the plane and false indicates the opposite side.

The method `wfcWFeature::GetZoneFeatureReferences` returns the references used to create the zone feature.

The method `wfcWFeature::GetZoneFeaturePlaneData` returns the planes used to create the zone feature.

Use the method `wfcWFeature::GetZoneXSectionGeometry` to retrieve the array of cross sections in the specified zone feature.

The method `wfcWFeature::IsInFooter` checks if the specified feature is currently located in the model tree footer. The footer is a section of the model tree that lists certain types of features such as, component interfaces, annotation features, zones, reference features, publish geometry, and analysis feature. The

features in the footer are always regenerated at the end of the feature list. You can move features, such as, reference features, annotation features, and so on, to the footer. Some features, such as, component interfaces, zones, and so on, are automatically placed in the footer. Refer to the Creo Parametric online Help for more information on footer. Refer to the Creo Parametric online Help for more information on footer.

Use the method `wfcWFeature::MoveToFooter` to move the specified feature into the model tree footer.

Use the method `wfcWFeature::MoveFromFooter` to move the specified feature out of the model tree footer.

Feature Inquiry

Methods Introduced:

- **wfcFeatureStatusFlag::GetFeatureId**
- **wfcFeatureStatusFlag::GetIsActive**
- **wfcFeatureStatusFlag::GetIsInactive**
- **wfcFeatureStatusFlag::GetIsChildOfExternalFailed**
- **wfcFeatureStatusFlag::GetIsChildOfFailed**
- **wfcFeatureStatusFlag::GetIsFailed**
- **wfcFeatureStatusFlag::GetIsFamtabSuppressed**
- **wfcFeatureStatusFlag::GetIsInvalid**
- **wfcFeatureStatusFlag::GetIsProgramSuppressed**
- **wfcFeatureStatusFlag::GetIsSimprepSuppressed**
- **wfcFeatureStatusFlag::GetIsSuppressed**

The method `wfcFeatureStatusFlag::GetFeatureId` retrieves the feature id of the feature in a part or an assembly.

The method `wfcFeatureStatusFlag::GetIsActive` returns true if the feature is active in a part or an assembly.

The method `wfcFeatureStatusFlag::GetIsInactive` returns true if the feature is inactive in a part or an assembly. If it returns false, it is an active feature.

The method `wfcFeatureStatusFlag::GetIsChildOfExternalFailed` returns true if the feature is a child of an external failed feature in a part or an assembly.

The method `wfcFeatureStatusFlag::GetIsChildOfFailed` returns true if the feature is a child of a failed feature in a part or an assembly.

The method `wfcFeatureStatusFlag::GetIsFailed` returns true if the feature failed regeneration in a part or an assembly.

The method `wfcFeatureStatusFlag::GetIsFamtabSuppressed` returns true if the feature is suppressed due to the family table settings.

The method `wfcFeatureStatusFlag::GetIsInvalid` returns true if the feature status could not be retrieved.

The method `wfcFeatureStatusFlag::GetIsProgramSuppressed` returns true if the feature is suppressed due to a Pro/PROGRAM functionality.

The method `wfcFeatureStatusFlag::GetIsSimprepSuppressed` returns true if the feature is suppressed due to a simplified representation.

The method `wfcFeatureStatusFlag::GetIsSuppressed` returns true if the feature is suppressed.

The method `wfcFeatureStatusFlag::GetIsUnregenerated` returns true if the feature is active and which has not yet been regenerated. This is due to a regeneration failure or if the status is obtained during the regeneration process.

Feature Operations

Methods Introduced:

- **pfcSolid::ExecuteFeatureOps**
- **pfcFeature::CreateSuppressOp**
- **pfcSuppressOperation::SetClip**
- **pfcSuppressOperation::SetAllowGroupMembers**
- **pfcSuppressOperation::SetAllowChildGroupMembers**
- **pfcFeature::CreateDeleteOp**
- **pfcDeleteOperation::SetClip**
- **pfcDeleteOperation::SetAllowGroupMembers**
- **pfcDeleteOperation::SetAllowChildGroupMembers**
- **pfcDeleteOperation::SetKeepEmbeddedDatums**
- **pfcFeature::CreateResumeOp**
- **pfcResumeOperation::SetWithParents**
- **pfcFeature::CreateReorderBeforeOp**
- **pfcReorderBeforeOperation::SetBeforeFeat**
- **pfcFeature::CreateReorderAfterOp**
- **pfcReorderAfterOperation::SetAfterFeat**
- **pfcFeatureOperations::create**
- **wfcWSolid::DeleteFeatures**
- **wfcWSolid::SuppressFeatures**

- **wfcWSolid::ResumeFeatures**
- **wfcWSolid::ReorderFeatures**

The method `pfcSolid::ExecuteFeatureOps` causes a sequence of feature operations to run in order. Feature operations include suppressing, resuming, reordering, and deleting features. The optional `pfcRegenInstructions` argument specifies whether the user will be allowed to fix the model if a regeneration failure occurs.

Note

Regenerating models in `resolve_mode` is not supported. As a result, the method `pfcSolid::ExecuteFeatureOps` is not supported anymore. PTC recommends to use the alternative methods `wfcWSolid::DeleteFeatures`, `wfcWSolid::SuppressFeatures`, `wfcWSolid::ResumeFeatures` and `wfcWSolid::ReorderFeatures` to delete, suppress, resume and reorder a list of features.

You can create an operation that will delete, suppress, reorder, or resume certain features using the methods in the interface `pfcFeature`. Each created operation must be passed as a member of the `pfcFeatureOperations` object to the method `pfcSolid::ExecuteFeatureOps`. You can create a sequence of the `pfcFeatureOperations` object using the method `pfcFeatureOperations::create`.

Some of the operations have specific options that you can modify to control the behavior of the operation:

- `Clip`—Specifies whether to delete or suppress all features after the selected feature. By default, this option is false.

  Use the methods `pfcDeleteOperation::SetClip` and `pfcSuppressOperation::SetClip` to modify this option.
- `AllowGroupMembers`—If this option is set to true and if the feature to be deleted or suppressed is a member of a group, then the feature will be deleted or suppressed out of the group. If this option is set to false, then the entire group containing the feature is deleted or suppressed. By default, this option is false. It can be set to true only if the option `Clip` is set to true.

  Use the methods `pfcSuppressOperation::SetAllowGroupMembers` and `pfcDeleteOperation::SetAllowGroupMembers` to modify this option.

- `AllowChildGroupMembers`—If this option is set to true and if the children of the feature to be deleted or suppressed are members of a group, then the children of the feature will be individually deleted or suppressed out of the group. If this option is set to false, then the entire group containing the feature and its children is deleted or suppressed. By default, this option is false. It can be set to true only if the options `Clip` and `AllowGroupMembers` are set to true.

  Use the methods `pfcSuppressOperation::SetAllowChildGroupMembers` and `pfcDeleteOperation::SetAllowChildGroupMembers` to modify this option.
- `KeepEmbeddedDatums`—Specifies whether to retain the embedded datums stored in a feature while deleting the feature. By default, this option is false.

  Use the method `pfcDeleteOperation::SetKeepEmbeddedDatums` to modify this option.
- `WithParents`—Specifies whether to resume the parents of the selected feature.

  Use the method `pfcResumeOperation::SetWithParents` to modify this option.
- `BeforeFeat`—Specifies the feature before which you want to reorder the features.

  Use the method `pfcReorderBeforeOperation::SetBeforeFeat` to modify this option.
- `AfterFeat`—Specifies the feature after which you want to reorder the features.

  Use the method `pfcReorderAfterOperation::SetAfterFeat` to modify this option.

Use the methods `wfcWSolid::DeleteFeatures`, `wfcWSolid::SuppressFeatures`, `wfcWSolid::ResumeFeatures` and `wfcWSolid::ReorderFeatures` to delete, suppress, resume and reorder a list of features. The input parameters for all the methods are:

- *FeatIDs*—The list of IDs for the features to be deleted, suppressed, reordered or resumed.
- *Options*—The list of options to be used. This input argument is not applicable to the method `wfcWSolid::ReorderFeatures`.
- *Instrs*—Regeneration instructions to be used.

The reorder method takes its second input parameter as:

- *NewFeatNum*—The intended location of the first feature in the specified list.

Feature Groups and Patterns

Patterns are treated as features in Creo applications. A feature type, `pfcFEATTYPE_PATTERN_HEAD`, is used for the pattern header feature.

Note

The pattern header feature is not treated as a leader or a member of the pattern by the methods described in the following section.

Methods Introduced:

- **pfcFeature::GetGroup**
- **pfcFeature::GetPattern**
- **pfcSolid::CreateLocalGroup**
- **pfcFeatureGroup::GetPattern**
- **pfcFeatureGroup::GetGroupLeader**
- **pfcFeaturePattern::GetPatternLeader**
- **pfcFeaturePattern::ListMembers**
- **pfcFeaturePattern::Delete**

The method `pfcFeature::GetGroup` returns a handle to the local group that contains the specified feature.

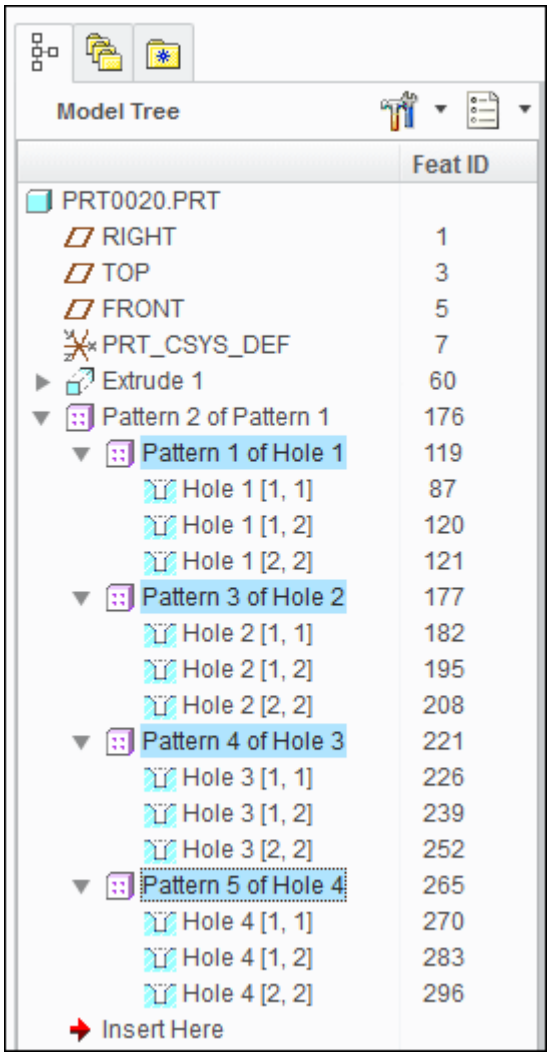
To get the first feature in the specified group call the method `pfcFeatureGroup::GetGroupLeader`.

The methods `pfcFeaturePattern::GetPatternLeader` and `pfcFeaturePattern::ListMembers` return features that make up the specified feature pattern.

A pattern is composed of a pattern header feature and a number of member features. You can pattern only a single feature. To pattern several features, create a local group and pattern this group.

You can also create a pattern of pattern. This creates a multiple level pattern. From Creo Parametric 2.0 M170 onward, for a pattern of pattern, the method `pfcFeaturePattern::ListMembers` returns all the pattern header features created at the first level.

For example, consider a model where a pattern of pattern has been created. The model tree is as shown below:



The method `pfcFeaturePattern::ListMembers` returns the pattern header features with following IDs for a pattern of pattern:

- 119
- 177
- 221
- 265

The methods `pfcFeature::GetPattern` and `pfcFeatureGroup::GetPattern` return the `pfcFeaturePattern` object that contains the corresponding `pfcFeature` or `pfcFeatureGroup`. Use the method `pfcSolid::CreateLocalGroup` to take a sequence of

features and create a local group with the specified name. To delete a `pfcFeaturePattern` object, call the method `pfcFeaturePattern::Delete`.

User Defined Features

Groups in Creo represent sets of contiguous features that act as a single feature for specific operations. Individual features are affected by most operations while some operations apply to an entire group:

- Suppress
- Delete
- Layers
- Patterning

User defined Features (UDFs) are groups of features that are stored in a file. When a UDF is placed in a new model the created features are automatically assigned to a group. A local group is a set of features that have been specifically assigned to a group to make modifications and patterning easier.

Note

All methods in this section can be used for UDFs and local groups.

Read Access to Groups and User Defined Features

Methods Introduced:

- **pfcFeatureGroup::GetUDFName**
- **pfcFeatureGroup::GetUDFInstanceName**
- **pfcFeatureGroup::ListUDFDimensions**
- **pfcUDFDimension::GetUDFDimensionName**

User defined features (UDF's) are groups of features that can be stored in a file and added to a new model. A local group is similar to a UDF except it is available only in the model in which is was created.

The method `pfcFeatureGroup::GetUDFName` provides the name of the group for the specified group instance. A particular group definition can be used more than once in a particular model.

If the group is a family table instance, the method `pfcFeatureGroup::GetUDFInstanceName` supplies the instance name.

The method `pfcFeatureGroup::ListUDFDimensions` traverses the dimensions that belong to the UDF. These dimensions correspond to the dimensions specified as variables when the UDF was created. Dimensions of the original features that were not variables in the UDF are not included unless the UDF was placed using the `Independent` option.

The method `pfcUDFDimension::GetUDFDimensionName` provides access to the dimension name specified when the UDF was created, and not the name of the dimension in the current model. This name is required to place the UDF programmatically using the method `pfcSolid::CreateUDFGroup`.

Creating Features from UDFs

Method Introduced:

- **pfcSolid::CreateUDFGroup**

The method `pfcSolid::CreateUDFGroup` is used to create new features by retrieving and applying the contents of an existing UDF file. It is equivalent to the Creo command `Feature`, `Create`, `User Defined`.

To understand the following explanation of this method, you must have a good knowledge and understanding of the use of UDF's in Creo applications. PTC recommends that you read about UDF's in theCreo online help, and practice defining and using UDF's in Creo application before you attempt to use this method.

When you create a UDF interactively, Creo application prompts you for the information it needs to fix the properties of the resulting features. When you create a UDF from Creo Object TOOLKIT C++, you can provide some or all of this information programmatically by filling several compact data classes that are inputs to the method `pfcSolid::CreateUDFGroup`.

During the call to `pfcSolid::CreateUDFGroup`, Creo application prompts you for the following:

- Information required by the UDF that was not provided in the input data structures
- Correct information to replace erroneous information

Such prompts are a useful way of diagnosing errors when you develop your application. This also means that, in addition to creating UDF's programmatically to provide automatic synthesis of model geometry, you can also use `pfcSolid::CreateUDFGroup` to create UDF's semi-interactively. This can simplify the interactions needed to place a complex UDF making it easier for the user and less prone to error.

Creating UDFs

Creating a UDF requires the following information:

- Name—The name of the UDF you are creating and the instance name if applicable.
- Dependency—Specify if the UDF is independent of the UDF definition or is modified by the changers made to it.
- Scale—How to scale the UDF relative to the placement model.
- Variable Dimension—The new values of the variables dimensions and pattern parameters, those whose values can be modified each time the UDF is created.
- Dimension Display—Whether to show or blank non-variable dimensions created within the UDF group.
- References—The geometrical elements that the UDF needs in order to relate the features it contains to the existing models features. The elements correspond to the picks that Creo application prompts you for when you create a UDF interactively using the prompts defined when the UDF was created. You cannot select an embedded datum as the UDF reference.
- Parts Intersection—When a UDF that is being created in an assembly contains features that modify the existing geometry you must define which parts are affected or intersected. You also need to know at what level in an assembly each intersection is going to be visible.
- Orientations—When a UDF contains a feature with a direction that is defined with respect to a datum plane Creo must know what direction the new feature will point to. When you create such a UDF interactively, Creo application prompts you for this information with a flip arrow.
- Quadrants—When a UDF contains a linearly placed feature that references two datum planes to define it's location in the new model Creo application prompts you to pick the location of the new feature. This is determined by which side of each datum plane the feature must lie. This selection is referred to as the quadrant because the are four possible combinations for each linearly place feature.

Creo Object TOOLKIT C++ uses a special class that prepares and sets all the options and passes them to Creo application.

Creating Interactively Defined UDFs

Method Introduced:

- **pfcUDFPromptCreateInstructions::Create**

This static method is used to create an instructions object that can be used to prompt a user for the required values that will create a UDF interactively.

Creating a Custom UDF

Method Introduced:

- **pfcUDFCustomCreateInstructions::Create**
- **wfcWUDFCustomCreateInstructions::Create**

The method `pfcUDFCustomCreateInstructions::Create` creates a `UDFCustomCreateInstructions` object with a specified name. To set the UDF creation parameters programmatically you must modify this object as described below. The members of this class relate closely to the prompts Creo gives you when you create a UDF interactively. PTC recommends that you experiment with creating the UDF interactively using Creo application before you write the Creo Object TOOLKIT C++ code to fill the structure.

The method `wfcWUDFCustomCreateInstructions::Create` creates a `WUDFCustomCreateInstructions` object with a specified name.

Setting the Family Table Instance Name

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetInstanceName**
- **pfcUDFCustomCreateInstructions::GetInstanceName**

If the UDF contains a family table, this field can be used to select the instance in the table. If the UDF does not contain a family table, or if the generic instance is to be selected, the do not set the string.

Setting Dependency Type

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetDependencyType**
- **pfcUDFCustomCreateInstructions::GetDependencyType**

The `pfcUDFDependencyType` object represents the dependency type of the UDF. The choices correspond to the choices available when you create a UDF interactively. This enumerated type takes the following values:

- `pfcUDFDEP_INDEPENDENT`
- `pfcUDFDEP_DRIVEN`

Note

`pfcUDFDEP_INDEPENDENT` is the default value, if this option is not set.

Setting Scale and Scale Type

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetScaleType**
- **pfcUDFCustomCreateInstructions::GetScaleType**
- **pfcUDFCustomCreateInstructions::SetScale**
- **pfcUDFCustomCreateInstructions::GetScale**

ScaleType specifies the length units of the UDF in the form of the `pfcUDFScaleType` object. This enumerated type takes the following values:

- `pfcUDFSCALE_SAME_SIZE`
- `pfcUDFSCALE_SAME_DIMS`
- `pfcUDFSCALE_CUSTOM`
- `pfcUDFSCALE_nil`

Note

The default value is `pfcUDFSCALE_SAME_SIZE` if this option is not set.

Scale specifies the scale factor. If the *ScaleType* is set to `UDFSCALE_CUSTOM`, `pfcSetScale` assigns the user defined scale factor. Otherwise, this attribute is ignored.

Setting the Appearance of the Non UDF Dimensions

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetDimDisplayType**
- **pfcUDFCustomCreateInstructions::GetDimDisplayType**

The `pfcUDFDimensionDisplayType` object sets the options in Creo for determining the appearance in the model of UDF dimensions and pattern parameters that were not variable in the UDF, and therefore cannot be modified in the model. This enumerated type takes the following values:

- `pfcUDFDISPLAY_NORMAL`

- `pfcUDFDISPLAY_READ_ONLY`
- `pfcUDFDISPLAY_BLANK`

Note

The default value is `pfcUDFDISPLAY_NORMAL` if this option is not set.

Setting the Variable Dimensions and Parameters

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetVariantValues**
- **pfcUDFVariantValues::create**
- **pfcUDFVariantValues::insert**
- **pfcUDFVariantDimension::Create**
- **pfcUDFVariantPatternParam::Create**
- **wfcWSession::GetUDFDataDefaultVariableParameters**
- **wfcUDFVariableParameter::Create**
- **wfcUDFVariableParameter::GetName**
- **wfcUDFVariableParameter::SetName**
- **wfcUDFVariableParameter::GetItemType**
- **wfcUDFVariableParameter::SetItemType**
- **wfcUDFVariableParameter::GetItemId**
- **wfcUDFVariableParameter::SetItemId**
- **wfcUDFVariableParameter::GetValue**
- **wfcUDFVariableParameter::SetValue**
- **wfcWUDFCustomCreateInstructions::SetVariableParameters**
- **wfcWUDFCustomCreateInstructions::GetVariableParameters**

`pfcUDFVariantValues` class represents an array of variable dimensions and pattern parameters.

Use `pfcUDFVariantValues::create` to create an empty object and then use `pfcUDFVariantValues::insert` to add `pfcUDFVariantPatternParam` or `pfcUDFVariantDimension` objects one by one.

`pfcUDFVariantDimension::Create` is a static method creating a `pfcUDFVariantDimension`. It accepts the following parameters:

- *Name*—The symbol that the dimension had when the UDF was originally defined not the prompt that the UDF uses when it is created interactively. To make this name easy to remember, before you define the UDF that you plan to create with the Creo Object TOOLKIT C++, you should modify the symbols of all the dimensions that you want to select to be variable. If you get the name wrong, `pfcCreateUDFGroup` will not recognize the dimension and prompts the user for the value in the usual way does not modify the value.
- *DimensionValue*—The new value.

If you do not remember the name, you can find it by creating the UDF interactively in a test model, then using the `pfcFeatureGroup::ListUDFDimensions` and `pfcUDFDimension::GetUDFDimensionName` to find out the name.

`pfcUDFVariantPatternParam::Create` is a static method which creates a `pfcUDFVariantPatternParam`. It accepts the following parameters:

- *name*—The string name that the pattern parameter had when the UDF was originally defined
- *patternparam*—The new value.

After the `pfcUDFVariantValues` object has been compiled, use `pfcUDFCustomCreateInstructions::SetVariantValues` to add the variable dimensions and parameters to the instructions.

Use the method `wfcWSession::GetUDFDataDefaultVariableParameters` to obtain an array of available variant parameters and/or annotation values that can optionally be set when placing this UDF. The input arguments to this method are:

- *name*
- *instance*

The method `wfcUDFVariableParameter::Create` enables you to create a variable parameter object using the UDF data. The input arguments are:

- *Name*—Specify the name of the variable parameter.
- *ItemType*—Specify the item type of the parameter using the enumerated type `pfcModelItemType`.
- *ItemId*—Specify the item ID of the variable parameter.

The methods `wfcUDFVariableParameter::GetName` and `wfcUDFVariableParameter::SetName`—get and set the name or the symbol of the variant parameter or annotation value.

The methods `wfcUDFVariableParameter::GetItemType` and `wfcUDFVariableParameter::SetItemType`—get and set the item type of the variant parameter or annotation value.

The methods `wfcUDFVariableParameter::GetItemId` and `wfcUDFVariableParameter::SetItemId`—get and set the item id of the variant parameter or annotation value.

The methods `wfcUDFVariableParameter::GetValue` and `wfcUDFVariableParameter::SetValue`—get and set the default value for the variant parameter or annotation value.

Use the method `wfcWUDFCustomCreateInstructions::SetVariableParameters` to set the variable parameter sequence for a UDF feature.

Use the method `wfcWUDFCustomCreateInstructions::GetVariableParameters` to retrieve the variable parameter sequence.

Setting the User Defined References

Methods Introduced:

- **pfcUDFReferences::create**
- **pfcUDFReferences::insert**
- **pfcUDFReference::Create**
- **pfcUDFReference::SetIsExternal**
- **pfcUDFReference::SetReferenceItem**
- **pfcUDFCustomCreateInstructions::SetReferences**

UDFReferences class represents an array of element references. Use `pfcUDFReferences::create` to create an empty object and then use `pfcUDFReferences::insert` to add `pfcUDFReference` objects one by one.

The method `pfcUDFReference::Create` is a static method creating a `UDFReference` object. It accepts the following parameters:

- *PromptForReference*—The prompt defined for this reference when the UDF was originally set up. It indicates which reference this structure is providing. If you get the prompt wrong, `pfcSolid::CreateUDFGroup` will not recognize it and prompts the user for the reference in the usual way.
- *ReferenceItem*—Specifies the `pfcSelection` object representing the referenced element. You can set `pfcSelection` programmatically or prompt the user for a selection separately. You cannot set an embedded datum as the UDF reference.

  There are two types of references:

- Internal—The referenced element belongs directly to the model that will contain the UDF. For an assembly, this means that the element belongs to the top level.
- External—The referenced element belongs to an assembly member other than the placement member.

To set the reference type, use the method `pfcUDFReference::SetIsExternal`.

To set the item to be used for reference, use the method `pfcUDFReference::SetReferenceItem`.

After the `UDFReferences` object has been set, use `pfcUDFCustomCreateInstructions::SetReferences` to add the program-defined references.

Setting the Assembly Intersections

Methods Introduced:

- **pfcUDFAssemblyIntersections::create**
- **pfcUDFAssemblyIntersections::insert**
- **pfcUDFAssemblyIntersection::Create**
- **pfcUDFAssemblyIntersection::SetInstanceNames**
- **pfcUDFCustomCreateInstructions::SetIntersections**

The `pfcUDFAssemblyIntersections` class represents an array of element references.

Use `pfcUDFAssemblyIntersections::create` to create an empty object and then use `pfcUDFAssemblyIntersections::insert` to add `pfcUDFAssemblyIntersection` objects one by one.

`pfcUDFAssemblyIntersection::Create` is a static method creating a `pfcUDFReference` object. It accepts the following parameters:

- *ComponentPath*—Is an `xintsequence_ptr` type object representing the component path of the part to be intersected.
- *Visibility level*—The number that corresponds to the visibility level of the intersected part in the assembly. If the number is equal to the length of the component path the feature is visible in the part that it intersects. If *Visibility level* is 0, the feature is visible at the level of the assembly containing the UDF.

`pfcUDFAssemblyIntersection::SetInstanceNames` sets an array of names for the new instances of parts created to represent the intersection geometry. This method accepts the following parameters:

- *instance names*—is a `xstringsequence_ptr` type object representing the array of new instance names.

After the `pfcUDFAssemblyIntersections` object has been set, use `pfcUDFCustomCreateInstructions::SetIntersections` to add the assembly intersections.

External Symbol: Parameters

The data object for external symbol parameters is `wfcUDFExternalParameter`.

Methods Introduced:

- **wfcUDFExternalParameter::Create**
- **wfcUDFExternalParameter::GetParameter**
- **wfcUDFExternalParameter::SetParameter**
- **wfcUDFExternalParameter::GetPrompt**
- **wfcUDFExternalParameter::SetPrompt**
- **wfcWUDFCustomCreateInstructions::SetExternalParameters**
- **wfcWUDFCustomCreateInstructions::GetExternalParameters**

The method `wfcUDFExternalParameter::Create` enables you to create an external parameter symbol object required by a UDF. The input arguments are:

- *name*—Specify the name of the external parameter symbol.
- *parameter*—Specify the parameter that is used to resolve this external symbol in the placement model.

The methods `wfcUDFExternalParameter::GetParameter` and `wfcUDFExternalParameter::SetParameter` retrieve and set the parameter used to resolve the external symbol.

The methods `wfcUDFExternalParameter::GetPrompt` and `wfcUDFExternalParameter::SetPrompt` retrieve and set the prompt for the external parameter symbol.

Use the method `wfcWUDFCustomCreateInstructions::SetExternalParameters` to the set the external parameters for a UDF feature.

Use the method `wfcWUDFCustomCreateInstructions::GetExternalParameters` to retrieve the external parameters.

External Symbol: Dimensions

The data object for external symbol parameters is `wfcUDFExternalDimension`.

Methods Introduced:

- **wfcUDFExternalDimension::Create**
- **wfcUDFExternalDimension::GetDimension**
- **wfcUDFExternalDimension::SetDimension**
- **wfcUDFExternalDimension::GetPrompt**
- **wfcUDFExternalDimension::SetPrompt**
- **wfcWUDFCustomCreateInstructions::SetExternalDimensions**
- **wfcWUDFCustomCreateInstructions::GetExternalDimensions**

The method `wfcUDFExternalDimension::Create` enables you to create an external dimension symbol object required by a UDF. The input arguments are:

- *name*—Specify the name of the external dimension symbol.
- *dimension*—Specify the dimension that is used to resolve this external symbol in the placement model.

The methods `wfcUDFExternalDimension::GetDimension` and `wfcUDFExternalDimension::SetDimension` retrieve and set the dimension used to resolve the external symbol.

The methods `wfcUDFExternalDimension::GetPrompt` and `wfcUDFExternalDimension::SetPrompt` retrieve and set the prompt used for the external dimension symbol.

Use the method `wfcWUDFCustomCreateInstructions::SetExternalDimensions` to the set the external dimensions for a UDF feature.

Use the method `wfcWUDFCustomCreateInstructions::GetExternalDimensions` to retrieve the external dimensions.

Setting Orientations

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetOrientations**
- **pfcUDFOrientations::create**
- **pfcUDFOrientations::insert**

`pfcUDFOrientations` class represents an array of orientations that provide the answers to Creo applications prompts that use a flip arrow. Each term is a `pfcUDFOrientation` object that takes the following values:

- `pfcUDFORIENT_INTERACTIVE`—Prompt for the orientation using a flip arrow.
- `pfcUDFORIENT_NO_FLIP`—Accept the default flip orientation.

- `pfcUDFORIENT_FLIP`—Invert the orientation from the default orientation.

Use `pfcUDFOrientations::create` to create an empty object and then use `pfcUDFOrientations::insert` to add `pfcUDFOrientation` objects one by one.

The order of orientations should correspond to the order in which Creo prompts for them when the UDF is created interactively. If you do not provide an orientation the default value `NO_FLIP` is used.

After the `pfcUDFOrientations` object has been set use `pfcUDFCustomCreateInstructions::SetOrientations` to add the orientations.

Setting Quadrants

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetQuadrants**

The method `pfcUDFCustomCreateInstructions::SetQuadrants` sets an array of points, which provide the X, Y, and Z coordinates that correspond to the picks answering the Creo prompts for the feature positions. The order of quadrants should correspond to the order in which Creo prompts for them when the UDF is created interactively.

Setting the External References

Methods Introduced:

- **pfcUDFCustomCreateInstructions::SetExtReferences**

The method `pfcUDFCustomCreateInstructions::SetExtReferences` sets an external reference assembly to be used when placing the UDF. This will be required when placing the UDF in the component using references outside of that component. References could be to the top level assembly of another component.

22

Datum Features

This chapter describes the Creo Object TOOLKIT C++ methods that provide read access to the properties of datum features.

Datum Plane Features

The properties of the Datum Plane feature are defined in the `pfcDatumPlaneFeat` data object.

Methods Introduced:

- **pfcDatumPlaneFeat::GetFlip**
- **pfcDatumPlaneFeat::GetConstraints**
- **pfcDatumPlaneConstraint::GetConstraintType**
- **pfcDatumPlaneThroughConstraint::GetThroughRef**
- **pfcDatumPlaneNormalConstraint::GetNormalRef**
- **pfcDatumPlaneParallelConstraint::GetParallelRef**
- **pfcDatumPlaneTangentConstraint::GetTangentRef**
- **pfcDatumPlaneOffsetConstraint::GetOffsetRef**
- **pfcDatumPlaneOffsetConstraint::GetOffsetValue**
- **pfcDatumPlaneOffsetCoordSysConstraint::GetCsysAxis**
- **pfcDatumPlaneAngleConstraint::GetAngleRef**
- **pfcDatumPlaneAngleConstraint::GetAngleValue**
- **pfcDatumPlaneSectionConstraint::GetSectionRef**
- **pfcDatumPlaneSectionConstraint::GetSectionIndex**

The properties of the `pfcDatumPlaneFeat` object are described as follows:

- `Flip`—Specifies whether the datum plane was flipped during creation. Use the method `pfcDatumPlaneFeat::GetFlip` to determine if the datum plane was flipped during creation.
- `Constraints`—Specifies a collection of constraints given by the `pfcDatumPlaneConstraint` object. The method `pfcDatumPlaneFeat::GetConstraints` obtains the collection of constraints defined for the datum plane.

Use the method `pfcDatumPlaneConstraint::GetConstraintType` to obtain the type of constraint. The type of constraint is given by the `pfcDatumPlaneConstraintType` enumerated type. The available types are as follows:

- `pfcDTMPLN_THRU`—Specifies the Through constraint. The `pfcDatumPlaneThroughConstraint` object specifies this constraint. Use the method `pfcDatumPlaneThroughConstraint::GetThroughRef` to get the reference selection handle for the Through constraint.

- `pfcDTMPLN_NORM`—Specifies the Normal constraint. The `pfcDatumPlaneNormalConstraint` object specifies this constraint. Use the method `pfcDatumPlaneNormalConstraint::GetNormalRef` to get the reference selection handle for the Normal constraint.
- `pfcDTMPLN_PRL`—Specifies the Parallel constraint. The `pfcDatumPlaneParallelConstraint` object specifies this constraint. Use the method `pfcDatumPlaneParallelConstraint::GetParallelRef` to get the reference selection handle for the Parallel constraint.
- `pfcDTMPLN_TANG`—Specifies the Tangent constraint. The `pfcDatumPlaneTangentConstraint` object specifies this constraint. Use the method `pfcDatumPlaneTangentConstraint::GetTangentRef` to get the reference selection handle for the Tangent constraint.
- `pfcDTMPLN_OFFS`—Specifies the Offset constraint. The `pfcDatumPlaneOffsetConstraint` object specifies this constraint. Use the method `pfcDatumPlaneOffsetConstraint::GetOffsetRef` to get the reference selection handle for the Offset constraint. Use the method `pfcDatumPlaneOffsetConstraint::GetOffsetValue` to get the offset value.

  An Offset constraint where the offset reference is a coordinate system is given by the `pfcDatumPlaneOffsetCoordSysConstraint` object. Use the method `pfcDatumPlaneOffsetCoordSysConstraint::GetCsysAxis` to get the reference coordinate axis.
- `pfcDTMPLN_ANG`—Specifies the Angle constraint. The `pfcDatumPlaneAngleConstraint` object specifies this constraint. Use the method `pfcDatumPlaneAngleConstraint::GetAngleRef` to get the reference selection handle for the Angle constraint. Use the method `pfcDatumPlaneAngleConstraint::GetAngleValue` to get the angle value.
- `pfcDTMPLN_SEC`—Specifies the Section constraint. The `pfcDatumPlaneSectionConstraint` object specifies this constraint. Use the method `pfcDatumPlaneSectionConstraint::GetSectionRef` to get the reference selection for the Section constraint. Use the method `pfcDatumPlaneSectionConstraint::GetSectionIndex` to get the section index.

Datum Axis Features

The properties of the Datum Axis feature are defined in the `pfcDatumAxisFeat` data object.

Methods Introduced:

- **pfcDatumAxisFeat::GetConstraints**
- **pfcDatumAxisConstraint::GetConstraintType**
- **pfcDatumAxisConstraint::GetConstraintRef**
- **pfcDatumAxisFeat::GetDimConstraints**
- **pfcDatumAxisDimensionConstraint::GetDimOffset**
- **pfcDatumAxisDimensionConstraint::GetDimRef**

The properties of the `pfcDatumAxisFeat` object are described as follows:

- `Constraints`—Specifies a collection of constraints given by the `pfcDatumAxisConstraint` object. The method `pfcDatumAxisFeat::GetConstraints` obtains the collection of constraints applied to the Datum Axis feature.

  This object contains the following attributes:

  - `ConstraintType`—Specifies the type of constraint in terms of the `pfcDatumAxisConstraintType` enumerated type. The constraint type determines the type of datum axis. The constraint types are:

    - `pfcDTMAXIS_NORMAL`—Specifies the Normal datum constraint.
    - `pfcDTMAXIS_THRU`—Specifies the Through datum constraint.
    - `pfcDTMAXIS_TANGENT`—Specifies the Tangent datum constraint.
    - `pfcDTMAXIS_CENTER`—Specifies the Center datum constraint.

    Use the method `pfcDatumAxisConstraint::GetConstraintType` to get the constraint type.
  - `ConstraintRef`—Specifies the reference selection for the constraint. Use the method `pfcDatumAxisConstraint::GetConstraintRef` to get the reference selection handle.
- `DimConstraints`—Specifies a collection of dimension constraints given by the `pfcDatumAxisDimensionConstraint` object. The method `pfcDatumAxisFeat::GetDimConstraints` obtains the collection of dimension constraints applied to the Datum Axis feature.

The pfcDatumAxisDimensionConstraint object contains the following attributes:

- DimOffset—Specifies the offset value for the dimension constraint. Use the method pfcDatumAxisDimensionConstraint::GetDimOffset to get the offset value.
- DimRef—Specifies the reference selection for the dimension constraint. Use the method pfcDatumAxisDimensionConstraint::GetDimRef to get the reference selection handle.

General Datum Point Features

The properties of the General Datum Point feature are defined in the pfcDatumPointFeat data object.

Methods Introduced:

- **pfcDatumPointFeat::GetFeatName**
- **pfcDatumPointFeat::GetPoints**
- **pfcGeneralDatumPoint::GetName**
- **pfcGeneralDatumPoint::GetPlaceConstraints**
- **pfcGeneralDatumPoint::GetDimConstraints**
- **pfcDatumPointConstraint::GetConstraintRef**
- **pfcDatumPointConstraint::GetConstraintType**
- **pfcDatumPointConstraint::GetValue**

The properties of the pfcDatumPointFeat object are described as follows:

- FeatName—Specifies the name of the General Datum Point feature. Use the method pfcDatumPointFeat::GetFeatName to get the name.
- GeneralDatumPoints—Specifies a collection of general datum points (given by the pfcGeneralDatumPoint object). Use the method pfcDatumPointFeat::DatumPointFeat.GetPoints to obtain the collection of general datum points. The pfcGeneralDatumPoint object consists of the following attributes:
  - Name—Specifies the name of the general datum point. Use the method pfcGeneralDatumPoint::GetName to get the name.
  - PlaceConstraints—Specifies a collection of placement constraints given by the pfcDatumPointPlacementConstraint object. Use

the method `pfcGeneralDatumPoint::GetPlaceConstraints` to obtain the collection of placement constraints.

- `DimConstraints`—Specifies a collection of dimension constraints given by the `pfcDatumPointDimensionConstraint` object. Use the method `pfcGeneralDatumPoint::GetDimConstraints` to obtain the collection of dimension constraints.

The constraints for a datum point are given by the `pfcDatumPointConstraint` object. This object contains the following attributes:

- `ConstraintRef`—Specifies the reference selection for the datum point constraint. Use the method to get the reference selection handle.
- `ConstraintType`—Specifies the type of datum point constraint. in terms of the `pfcDatumPointConstraintType` enumerated type. Use the method `pfcDatumPointConstraint::GetConstraintType` to get the constraint type.
- `Value`—Specifies the constraint reference value with respect to the datum point. Use the method `pfcDatumPointConstraint::GetValue` to get the value of the constraint reference with respect to the datum point.

The `pfcDatumPointPlacementConstraint` and `pfcDatumPointDimensionConstraint` objects inherit from the `pfcDatumPointConstraint` object. Use the methods of the `pfcDatumPointConstraint` object for the inherited objects.

Datum Coordinate System Features

The properties of the Datum Coordinate System feature are defined in the `pfcCoordSysFeat` object.

Methods Introduced:

- **pfcCoordSysFeat::GetOriginConstraints**
- **pfcDatumCsysOriginConstraint::GetOriginRef**
- **pfcCoordSysFeat::GetDimensionConstraints**
- **pfcDatumCsysDimensionConstraint::GetDimRef**
- **pfcDatumCsysDimensionConstraint::GetDimValue**
- **pfcDatumCsysDimensionConstraint::GetDimConstraintType**
- **pfcCoordSysFeat::GetOrientationConstraints**
- **pfcDatumCsysOrientMoveConstraint::GetOrientMoveConstraintType**
- **pfcDatumCsysOrientMoveConstraint::GetOrientMoveValue**

- **pfcCoordSysFeat::GetIsNormalToScreen**
- **pfcCoordSysFeat::GetOffsetType**
- **pfcCoordSysFeat::GetOnSurfaceType**
- **pfcCoordSysFeat::GetOrientByMethod**

The properties of the `pfcCoordSysFeat` object are described as follows:

- `OriginConstraints`—Specifies a collection of origin constraints given by the `pfcDatumCsysOriginConstraint` object. Use the method `pfcCoordSysFeat::GetOriginConstraints` to obtain the collection of origin constraints for the coordinate system. This object contains the following attribute:
  - `OriginRef`—Specifies the selection reference for the origin. Use the method `pfcDatumCsysOriginConstraint::GetOriginRef` to get the selection reference handle.
- `DimensionConstraints`—Specifies a collection of dimension constraints (given by the `pfcDatumCsysDimensionConstraint` object). Use the method `pfcCoordSysFeat::GetDimensionConstraints` to obtain the collection of dimension constraints for the coordinate system. This object contains the following attributes:
  - `DimRef`—Specifies the reference selection for the dimension constraint. Use the method `pfcDatumCsysDimensionConstraint::GetDimRef` to get the reference selection handle.
  - `DimValue`—Specifies the value of the reference. Use the method `pfcDatumCsysDimensionConstraint::GetDimValue` to get the value.
  - `DimConstraintType`—Specifies the type of dimension constraint in terms of the `pfcDatumCsysDimConstraintType` enumerated type. Use the method `pfcDatumCsysDimensionConstraint::GetDimConstraintType` to get the constraint type. The constraint types are:
    - `pfcDTMCSYS_DIM_OFFSET`—Specifies the offset type constraint.
    - `pfcDTMCSYS_DIM_ALIGN`—Specifies the align type constraint.
- `OrientationConstraints`—Specifies a collection of orientation constraints (given by the `CoordSysFeat.DatumCsysOrientMoveConstraint` object). Use the method `pfcCoordSysFeat::GetOrientationConstraints` to

obtain the collection of orientation constraints for the coordinate system. This object contains the following attributes:

  - `OrientMoveConstraintType`—Specifies the type of orientation for the constraint. The orientation type is given by the `pfcDatumCsysOrientMoveConstraintType` enumerated type. Use the method `pfcDatumCsysOrientMoveConstraint::GetOrientMoveConstraintType` to get the orientation type.
  - `OrientMoveValue`—Specifies the reference value for the constraint. Use the method `pfcDatumCsysOrientMoveConstraint::GetOrientMoveValue` to get the reference value.

- `IsNormalToScreen`—Specifies if the coordinate system is normal to screen. Use the method `pfcCoordSysFeat::GetIsNormalToScreen` to determine if the coordinate system is normal to screen.
- `OffsetType`—Specifies the offset type of the coordinate system in terms of the `pfcDatumCsysOffsetType` enumerated type. Use the method `pfcCoordSysFeat::GetOffsetType` to get the offset type. The offset types are:
  - `pfcDTMCSYS_OFFSET_CARTESIAN`—Specifies a cartesian coordinate system that has been defined by setting the values for the `pfcDTMCSYS_MOVE_TRAN_X`, `pfcDTMCSYS_MOVE_TRAN_Y`, and `pfcDTMCSYS_MOVE_TRAN_Z` or `pfcDTMCSYS_MOVE_ROT_X`, `pfcDTMCSYS_MOVE_ROT_Y`, and `pfcDTMCSYS_MOVE_ROT_Z` orientation constants.
  - `pfcDTMCSYS_OFFSET_CYLINDRICAL`—Specifies a cylindrical coordinate system that has been defined by setting the values for the `pfcDTMCSYS_MOVE_RAD`, `pfcDTMCSYS_MOVE_THETA`, and `pfcDTMCSYS_MOVE_TRAN_ZI` orientation constants.
  - `pfcDTMCSYS_OFFSET_SPHERICAL`—Specifies a spherical coordinate system that has been defined by setting the values for the `pfcDTMCSYS_MOVE_RAD`, `pfcDTMCSYS_MOVE_THETA`, and `pfcDTMCSYS_MOVE_TRAN_PHI` orientation constants.
- `OnSurfaceType`—Specifies the on surface type for the coordinate system in terms of the `pfcDatumCsysOffsetType` enumerated type. Use the method `pfcCoordSysFeat::GetOnSurfaceType` to get the on surface type property of the coordinate system. The on surface types are:
  - `pfcDTMCSYS_ONSURF_LINEAR`—Specifies a coordinate system placed on the selected surface by using two linear dimensions.
  - `pfcDTMCSYS_ONSURF_RADIAL`—Specifies a coordinate system placed on the selected surface by using a linear dimension and an angular dimension. The radius value is used to specify the linear dimension.

- pfcDTMCSYS_ONSURF_DIAMETER—This type is similar to the pfcDTMCSYS_ONSURF_RADIAL type, except that the diameter value is used to specify the linear dimension. It is available only when planar surfaces are used as the reference.

- OrientByMethod—Specifies the orientation method in terms of the pfcDatumCsysOrientByMethod enumerated type. Use the method pfcCoordSysFeat::GetOrientByMethod to get the orientation method. The available orientation types are:
  - pfcDTMCSYS_ORIENT_BY_SEL_REFS—Specifies the orientation by selected references.
  - pfcDTMCSYS_ORIENT_BY_SEL_CSYS_AXES—Specifies the orientation by corordinate system axes.

23

Cross Sections

The methods in this chapter enable you to create, access, modify, and delete cross sections.

Listing Cross Sections

Methods Introduced:

- **pfcSolid::ListCrossSections**
- **pfcSolid::GetCrossSection**
- **pfcXSection::GetName**
- **pfcXSection::SetName**
- **pfcXSection::GetXSecType**
- **pfcXSecType::GetType**
- **pfcXSecType::GetObjectType**
- **pfcXSection::Delete**
- **pfcXSection::Display**
- **pfcXSection::Regenerate**

The method `pfcSolid::ListCrossSections` returns a sequence of cross section objects represented by the Xsection interface. The method `pfcSolid::GetCrossSection` searches for a cross section given its name.

The method `pfcXSection::GetName` returns the name of the cross section in the Creo application. The method `pfcXSection::SetName` modifies the cross section name.

The method `pfcXSection::GetXSecType` returns the type of cross section as a `pfcXSecType` object, that is planar or offset, and the type of item intersected by the cross section.

The method `pfcXSecType::GetType` returns the type of intersection for the cross section using the enumerated type `pfcXSecCutType`. The valid values are:

- `pfcXSEC_PLANAR`
- `pfcXSEC_OFFSET`

The method `pfcXSecType::GetObjectType` returns the type of item intersected by the cross section using the enumerated type `pfcXSecCutobjType`. The valid values are:

- `pfcXSECTYPE_MODEL`—Specifies that the cross section was created on solid geometry.
- `pfcXSECTYPE_QUILTS`—Specifies that the cross section was created on one quilt surface.
- `pfcXSECTYPE_MODELQUILTS`—Specifies that the cross section was created on solid geometry and all quilt surfaces.
- `pfcXSECTYPE_ONEPART`—Specifies that the cross section was created on one component in the assembly.

The method `pfcXSection::Delete` deletes a cross section.

The method `pfcXSection::Display` forces a display of the cross section in the window.

The method `pfcXSection::Regenerate` regenerates the cross-section of a part or an assembly.

Extracting Cross-Sectional Geometry

Methods Introduced:

- **wfcWXSection::CollectCutComponents**
- **wfcWXSection::GetExcludedItems**
- **wfcXSectionCutComponentItem::GetType**
- **wfcXSectionCutComponentItem::GetId**
- **wfcXSectionCutComponentItem::GetGeometry**
- **wfcXSectionCutComponentItem::SetXHatchStyle**
- **wfcXSectionCutComponentItem::GetXHatchStyle**
- **wfcXSectionCutComponent::GetCutItems**
- **wfcXSectionCutComponent::GetPath**
- **wfcXSectionCutComponent::SetXHatchStyle**
- **wfcXSectionCutComponent::GetXHatchStyle**
- **wfcXSectionExcludeItems::Create**
- **wfcXSectionExcludeItems::GetItems**
- **wfcXSectionExcludeItems::SetItems**
- **wfcXSectionExcludeItems::GetExclude**
- **wfcXSectionExcludeItems::SetExclude**
- **wfcXSectionGeometry::GetGeometry**
- **wfcXSectionGeometry::GetMemberIdTable**
- **wfcXSectionGeometry::GetQuiltId**
- **wfcWXSection::GetPlane**
- **wfcWXSection::GetOffsetXSectionData**
- **wfcWXSection::GetFlip**
- **wfcWXSection::GetComponents**
- **wfcWXSection::GetName**
- **wfcWXSection::SetName**

- **wfcXSectionComponents::GetExclude**
- **wfcXSectionComponents::SetExclude**
- **wfcXSectionComponents::GetComponents**
- **wfcXSectionComponents::SetComponents**
- **wfcZoneXSectionGeometry::GetGeometries**
- **wfcOffsetXSectionData::GetXSectionLines**
- **wfcOffsetXSectionData::GetPlaneData**
- **wfcOffsetXSectionData::GetOneSided**
- **wfcOffsetXSectionData::GetFlip**

Superseded Methods:

- **wfcWXSection::CollectGeometry**

The geometry of a cross section in an assembly is divided into components. Each component corresponds to one of the parts in the assembly that is intersected by the cross section, and describes the geometry of that intersection. A component can have disjoint geometry if the cross section intersects a given part instance in more than one place.

A cross section in a part has a single component.

The components of a cross section are identified by consecutive integer identifiers that always start at 0.

In Creo 7.0.0.0 and later, the method `wfcWXSection::CollectGeometry` is deprecated. Use the method `wfcWXSection::CollectCutComponents` instead.

The method `wfcWXSection::CollectCutComponents` returns the object `XSectionCutComponent` that contains the geometry of all cut components in a specified cross section. A sequence item is created for each body. If no bodies are created, the method returns one `XSectionCutComponent` object for each component. `wfcXSectionCutComponents` retrieves information from the `wfcXSectionCutComponent` object.

Note

The method returns only items containing solid geometry (nonquilt and nonfacetrep) components that intersect their nonhidden, nonsuppressed parts and are inside view borders. If the input argument *drawing view* is null, the method returns all nonempty components.

The method `wfcWXSection::GetExcludedItems` returns paths to the exclude or include components and bodies of the specified cross section using the object `wfcXSectionExcludeItems`.

The method `wfcXSectionCutComponentItem::GetType` returns the type of item - body or quilt.

The method `wfcXSectionCutComponentItem::GetId` returns the Id of the body or the quilt being cut.

The method `wfcXSectionCutComponentItem::GetGeometry` returns the surface geometry created by the cross section by cutting specific body or quilt.

The method `wfcXSectionCutComponentItem::SetXHatchStyle` sets the cross section xhatch style of the body. If the xhatch style exists in the session, it will be set.

The method `wfcXSectionCutComponentItem::GetXHatchStyle` returns the cross section xhatch style of the body.

The method `wfcXSectionCutComponent::GetCutItems` returns the items (body or quilt) that are cut on the component.

The method `wfcXSectionCutComponent::GetPath` returns the path to the component being cut by the cross section.

The method `wfcXSectionCutComponent::SetXHatchStyle` sets the cross section xhatch style on the component. If the xhatch style exists in the session, it will be set.

The method `wfcXSectionCutComponent::GetXHatchStyle` returns the cross section xhatch style of the component.

The method `wfcXSectionExcludeItems::Create` creates the interface for excluded items using the `wfcXSectionExcludeItems` object. The input arguments follow:

- *Items*—The component or bodies selections.
- *Exclude*—`True` if excluded, `False` if not.

The methods `wfcXSectionExcludeItems::GetExclude` and `wfcXSectionExcludeItems::SetExclude` get and set the flag to exclude or include items in the cross section.

The methods `wfcXSectionExcludeItems::GetItems` and `wfcXSectionExcludeItems::SetItems` get and set the items to exclude or include in the cross section.

The method `wfcXSectionGeometry::GetGeometry` returns the geometry of all components in the specified cross section as `pfcSurface` object. Use the methods `wfcXSectionGeometry::GetMemberIdTable` and `wfcXSectionGeometry::GetQuiltId` to get the component and quilt identifiers in the specified cross section.

The method `wfcWXSection::GetPlane` returns the plane geometry for a specified cross section.

The method `wfcWXSection::GetOffsetXSectionData` returns the parameters for a specified offset cross section.

The method `wfcWXSection::GetFlip` returns a boolean value that indicates the direction in which the cross section has been clipped. The value `False` indicates that the cross section has been clipped in the direction of the positive normal to the cross section plane. `True` indicates that the cross section has been clipped in the opposite direction of the positive normal.

The method `wfcWXSection::GetComponents` returns an array of paths to the assembly components that have been included or excluded for the specified cross section.

The methods `wfcWXSection::GetName` and `wfcWXSection::SetName` enable you to retrieve and rename the name of a cross section in an assembly, respectively.

The method `wfcXSectionComponents::GetExclude` specifies if the assembly components were excluded or not.

The method `wfcXSectionComponents::SetExclude` specifies if the assembly components are to be excluded or not.

The methods `wfcXSectionComponents::GetComponents` and `wfcXSectionComponents::SetComponents` get and set the sequence of components that have been included or excluded for the specified cross section definition.

The method `wfcZoneXSectionGeometry::GetGeometries` returns the geometry information of a zone feature in the specified cross section as the `wfcXSectionGeometries` object.

The method `wfcOffsetXSectionData::GetXSectionLines` returns information about the line segment entities in the cross section as a `wfcSectionEntityLine` object.

The method `wfcOffsetXSectionData::GetPlaneData` returns information about the entity datum plane.

The method `wfcOffsetXSectionData::GetOneSided` returns True if the cross section lies on one side of the entity plane. The method returns false if the cross section is both-sided.

If the output argument of the method `wfcOffsetXSectionData::GetFlip` is False, the Creo application removes material from the left of the cross section entities if the viewing direction is from the positive side of the entity plane and if `wfcOffsetXSectionData::GetOneSided` is true, the Creo application removes only the material from positive side of the entity plane.

Creating and Modifying Cross Sections

Methods Introduced:

- **wfcWSolid::CreateParallelXSection**
- **wfcWSolid::CreatePlanarXSectionWithOptions**
- **wfcWModel::CanCreateSectionFeature**
- **wfcWModel::ConvertOldXSectionsToNew**
- **wfcWXSection::IsFeature**
- **wfcWXSection::GetFeature**
- **wfcXSectionComponents::Create**

The method `wfcWSolid::CreateParallelXSection` creates a cross section feature parallel to a given plane.

From Creo 7.0.0.0 and later, the method `wfcWSolid::CreatePlanarXSection` is deprecated. Use the method `wfcWSolid::CreatePlanarXSectionWithOptions` instead.

The method `wfcWSolid::CreatePlanarXSectionWithOptions` creates a cross section feature through a datum plane and also makes the cross section visible. The input arguments follow:

- *Name* —Name of the cross section.
- *Plane* —The cutting plane. The cutting plane must belong to the model.
- *CutObjectType*—Type of the object that will be cut by the cross section. It is specified by the enumerated type `pfcXSecCutobjType`.
- *QuiltOrPart*—Specifies the selection of the quilt or component.
- *Flip*—Direction in which the cross section will be clipped. The value 1 indicates that the cross section will be clipped in the direction of the positive normal to the cutting plane. -1 indicates that the cross section will be clipped in the opposite direction of the positive normal.
- *ExcludeItems*—Selection of selected bodies or parts to be included or excludes from the cross section defined by the class `wfcXSectionExcludeItems`.
- *Data*—Reserved for future use.

Note

- The legacy cross sections, that is, the cross sections created in Pro/ENGINEER, Creo Elements/Pro, and in releases prior to Creo Parametric 2.0 are not supported.
- The methods `wfcWSolid::CreateParallelXSection` and `wfcWSolid::CreatePlanarXSection` automatically convert the legacy cross sections to new cross section features as defined in Creo Parametric 2.0 onward, before creating any new cross section feature.

Use the method `wfcWModel::CanCreateSectionFeature` to check if new cross section features can be created in the specified model. The method returns false if the specified model has legacy cross sections.

The method `wfcWModel::ConvertOldXSectionsToNew` converts the legacy cross sections to new cross section features as defined in Creo Parametric 2.0 onward for the specified model.

Use the method `wfcWXSection::IsFeature` to check whether the cross section is a feature.

The method `wfcWXSection::GetFeature` returns a `pfcFeature` object representing the cross section feature. You must use this method only if `wfcWModel::CanCreateSectionFeature` returns true.

Use the method `wfcXSectionComponents::Create` to create the object `wfcXSectionComponents` that contains interface for excluding or including components from a specified cross section. Pass the following values to this object:

- `Components`—Specify the sequence of components to exclude or include from a cross section definition.
- `Exclude`—Specify the value as `true` to exclude components and `false` to include.

Mass Properties of Cross Sections

Method Introduced:

- **wfcWXSection::GetMassProperty**

The method `wfcWXSection::GetMassProperty` calculates the mass properties of the cross-section in the specified coordinate system. The method needs the name of a coordinate system datum whose X- and Y-axes are parallel to the cross section. The output from this method also refers to the coordinate system datum.

Note

- The method `wfcWXSection::GetMassProperty` is not supported for offset and quilt type of cross-sections.
- The following methods can be called on `MassProperty` object returned by `wfcWXSection::GetMassProperty`:
  - `pfcSolid::MassProperty`
  - `pfcMassProperty::GetSurfaceArea`
  - `pfcMassProperty::SetSurfaceArea`
  - `pfcMassProperty::GetGravityCenter`
  - `pfcMassProperty::SetGravityCenter`
  - `pfcMassProperty::GetCoordSysInertia`
  - `pfcMassProperty::SetCoordSysInertia`

Line Patterns of Cross Section Components

Methods Introduced:

- **wfcWXSection::GetCompXSectionHatches**
- **wfcWXSection::SetCompXSectionHatches**
- **wfcXSectionHatch::Create**
- **wfcXSectionHatch::GetAngle**
- **wfcXSectionHatch::SetAngle**
- **wfcXSectionHatch::GetSpacing**
- **wfcXSectionHatch::SetSpacing**
- **wfcXSectionHatch::GetOffset**
- **wfcXSectionHatch::SetOffset**
- **wfcXSectionHatchStyle::GetType**
- **wfcXSectionHatchStyle::SetType**

- **wfcXSectionHatchStyle::GetOldHatches**
- **wfcXSectionHatchStyle::SetOldHatches**
- **wfcXSectionHatchStyle::GetNewHatches**
- **wfcXSectionHatchStyle::SetNewHatches**
- **wfcXSectionNewHatch::Create**
- **wfcXSectionNewHatch::GetAngle**
- **wfcXSectionNewHatch::SetAngle**
- **wfcXSectionNewHatch::GetXOrigin**
- **wfcXSectionNewHatch::SetXOrigin**
- **wfcXSectionNewHatch::GetYOrigin**
- **wfcXSectionNewHatch::SetYOrigin**
- **wfcXSectionNewHatch::GetXDelta**
- **wfcXSectionNewHatch::SetXDelta**
- **wfcXSectionNewHatch::GetYDelta**
- **wfcXSectionNewHatch::SetYDelta**
- **wfcXSectionNewHatch::GetDash**
- **wfcXSectionNewHatch::SetDash**
- **wfcXSectionNewHatch::GetColor**
- **wfcXSectionNewHatch::SetColor**
- **wfcXSectionOldHatch::Create**
- **wfcXSectionOldHatch::GetAngle**
- **wfcXSectionOldHatch::SetAngle**
- **wfcXSectionOldHatch::GetSpacing**
- **wfcXSectionOldHatch::SetSpacing**
- **wfcXSectionOldHatch::GetOffset**
- **wfcXSectionOldHatch::SetOffset**
- **wfcXSectionOldHatch::GetFont**
- **wfcXSectionOldHatch::SetFont**
- **wfcXSectionOldHatch::GetColor**
- **wfcXSectionOldHatch::SetColor**

The method `wfcWXSection::GetCompXSectionHatches` returns the line patterns of a cross section as a `wfcXSectionHatches` object. The input arguments to the method are the ID of the cross section component and the drawing view containing the cross section component. The information related to

line patterns is obtained from `wfcXSectionHatch`. Use the method `wfcWXSection::SetCompXSectionHatches` to set the line patterns for a cross section.

The method `wfcXSectionHatch::Create` creates a data object that contains information about the line patterns.

Use the methods `wfcXSectionHatch::GetAngle` and `wfcXSectionHatch::SetAngle` to get and set the angle in the line patterns.

The methods `wfcXSectionHatch::GetSpacing` and `wfcXSectionHatch::SetSpacing` get and set the distance between the line patterns.

The methods `wfcXSectionHatch::GetOffset` and `wfcXSectionHatch::SetOffset` get and set the offset of the first line in the pattern.

The methods `wfcXSectionHatchStyle::GetType` and `wfcXSectionHatchStyle::SetType` get and set the type of the hatch style.

The methods `wfcXSectionHatchStyle::GetOldHatches` and `wfcXSectionHatchStyle::SetOldHatches` get and set the old line hatch object sequence using the object `wfcXSectionOldHatches`.

The methods `wfcXSectionHatchStyle::GetNewHatches` and `wfcXSectionHatchStyle::SetNewHatches` get and set the new line hatch object sequence using the object `wfcXSectionNewHatches`.

The method `wfcXSectionNewHatch::Create` creates the interface for new line pattern hatching using the `wfcXSectionNewHatch` object. The input arguments follow:

- *Angle*—Angle for new line pattern.
- *Color*—Color for new line pattern.

The methods `wfcXSectionNewHatch::GetAngle` and `wfcXSectionNewHatch::SetAngle` get and set the angle for new line pattern.

The methods `wfcXSectionNewHatch::GetXOrigin` and `wfcXSectionNewHatch::SetXOrigin` get and set the X origin for the new line pattern.

The methods `wfcXSectionNewHatch::GetYOrigin` and `wfcXSectionNewHatch::SetYOrigin` get and set the Y origin for the new line pattern.

The methods `wfcXSectionNewHatch::GetXDelta` and `wfcXSectionNewHatch::SetXDelta` get and set the X delta for the new line pattern.

The methods `wfcXSectionNewHatch::GetYDelta` and `wfcXSectionNewHatch::SetYDelta` get and set the Y delta for the new line pattern.

The methods `wfcXSectionNewHatch::GetDash` and `wfcXSectionNewHatch::SetDash` get and set the dash for the new line pattern.

The methods `wfcXSectionNewHatch::GetColor` and `wfcXSectionNewHatch::SetColor` get and set the color for the new line pattern.

The method `wfcXSectionOldHatch::Create` creates the interface for old line pattern hatching using the `wfcXSectionOldHatch` object. The input argument *Angle* is the angle for old line pattern.

The methods `wfcXSectionOldHatch::GetAngle` and `wfcXSectionOldHatch::SetAngle` get and set the angle for old line pattern.

The methods `wfcXSectionOldHatch::GetSpacing` and `wfcXSectionOldHatch::SetSpacing` get and set the spacing for the old line pattern.

The methods `wfcXSectionOldHatch::GetOffset` and `wfcXSectionOldHatch::SetOffset` get and set the offset for the old line pattern.

The methods `wfcXSectionOldHatch::GetFont` and `wfcXSectionOldHatch::SetFont` get and set the font for the old line pattern.

The methods `wfcXSectionOldHatch::GetColor` and `wfcXSectionOldHatch::SetColor` get and set the color for the old line pattern.

Example 1: Creating a Planar Cross Section and Editing the Hatch Parameters

The sample code in `OTKXCrossSection.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_solid` shows how to create a planar cross section using the Creo Object TOOLKIT C++ methods. It also shows how to edit hatch parameters for all the cross sections in the current model.

24

External Objects

This chapter describes the Creo Object TOOLKIT C++ methods that enable you to create and manipulate external objects.

Summary of External Objects

External objects are objects created by an application that is external to Creo Parametric. Although these objects can be displayed and selected within a Creo Parametric session, they cannot be independently created by Creo Parametric. Using Creo Object TOOLKIT C++ methods, you can define and manipulate external objects, which are then stored in a model database.

Note

External objects are limited to text and wireframe entities. In addition, external objects can be created for parts and assemblies only. That is, external objects can be stored in a part or assembly database only.

In Creo Object TOOLKIT C++ application, an external object is defined by a `wfcExternalObject` object. This DHandle identifies an external object in the Creo Parametric database, which contains the following information for the object:

- Object class—A class of external objects is a group that contains objects with similar characteristics. All external objects must belong to a class. Object class is contained in the `wfcExternalObjectClass` object.
- Object data—The object data contains information about the display and selection of an external object. Object data is contained in the `wfcExternalObjectData` object.
- Object parameters—External objects can own parameters. You can use the `wfcWParameter` API to get, set, and modify external object parameters.
- Object references—External objects can reference any Creo object. This functionality is useful when changes to Creo objects need to instigate changes in the external objects. The changes are communicated back to your Creo Object TOOLKIT C++ application via the callback methods.
- Callback methods—Creo Object TOOLKIT C++ enables you to specify callback methods for a class of external objects. These methods are called whenever the external object owner or reference is deleted, suppressed, or modified. In this manner, the appearance and behavior of your external objects can depend on the object owner or reference.

External Objects and Object Classes

This section describes the Creo Object TOOLKIT C++ methods that relate to the creation and manipulation of external objects and object classes.

Note

This description does not address the display or selection of the external object. For more information see External Object Data on page 403.

Creating External Objects

Methods introduced:

- **wfcExternalObjectClass::CreateObject**
- **wfcExternalObjectClass::GetName**
- **wfcExternalObjectClass::GetType**

After the object class is registered, you can create the external object by calling the method `wfcExternalObjectClass::CreateObject`. The input arguments to this method is *owner*. (Currently, the owner of the external object can be a part or an assembly only.) To get the information of the newly created object, pass the data handle `wfcExternalObject` to the method `wfcExternalObjectClass::CreateObject`.

When the external object is created, it is assigned an integer identifier that is persistent from session to session. The external object is saved as part of the model database and will be available when the model is retrieved next.

Use the method `wfcExternalObjectClass::GetName` to retrieve the name of the external object class.

Use the method `wfcExternalObjectClass::GetType` to retrieve the type of the external object class.

External Object Owners

Methods introduced:

- **wfcExternalObject::GetOwner**

The owner of an external object is set during the call to `wfcExternalObjectClass::CreateObject`. For example, the “owner” would be the part or assembly where the external object resides.

The method `wfcExternalObject::GetOwner` retrieves the owner of an existing external object. To get the owner to an external object, pass the data handle `pfcModelItem` to the method `wfcExternalObject::GetOwner`.

External Object Data

Simply creating an external object does not allow the object to be displayed or selected in Creo Parametric. For this, you must supply external object data that is used, stored, and retrieved by Creo Parametric. The data is removed from the model database when the external object is deleted.

External object data is described by the opaque workspace handle `wfcExternalObjectData`. The methods required to initialize and modify this object are specific to the type of data being created. That is, creating display data requires one set of methods, whereas creating selection data requires another.

Once you have created a `wfcExternalObjectData` object, the manipulation of the external object data is independent of its contents: the methods required to add or remove data are the same for both display and selection data.

Display Data for External Objects

Display data gives information to Creo Parametric about how the external object appears in the model window. This data must include the color, scale, line type, and transformation of the external object. In addition, display data can include settings that override the user's ability to zoom and spin the external object.

Note

Setting the display data does not result in the external object being displayed. To see the object, you must repaint the model window using the method `pfcWindow::Repaint`.

Methods introduced:

- **wfcExternalObjectDisplayData::Create**

Use the method `wfcExternalObjectDisplayData::Create` to create a display data information for an external object. The input arguments are as follows:

- *Ents*—Specify the entities in the `pfcCurveDescriptors` object in the specified display data.
- *EntityColors*—Specify the entities in the `pfcStdColors` object in the specified display data.

The method `wfcExternalObjectDisplayData::Create` returns the display properties of the external object in the `wfcExternalObjectDisplayData` object.

Creating the External Object Entity

Methods introduced:

- **wfcExternalObjectDisplayData::GetEntities**
- **wfcExternalObjectDisplayData::SetEntities**
- **wfcExternalObjectDisplayData::GetEntityColors**
- **wfcExternalObjectDisplayData::GetEntityColors**

External objects are currently limited to text and wireframe entities. You can specify the entities to be displayed by creating an array of `pfcCurveDescriptors` objects that contain that necessary information. `pfcCurveDescriptors` is a union of specific entity structures, such as line, arrow, arc, circle, spline, and text. Note that when you specify the entities in the `pfcCurveDescriptors` array, the coordinate system used is the default model coordinate system.

Use the method `wfcExternalObjectDisplayData::GetEntities` to retrieve the entities that make up an external object in a specified display data.

After you have created the array of `pfcCurveDescriptors` objects, you can add entities to the display data by calling the method `wfcExternalObjectDisplayData::SetEntities`. The input argument to the method `wfcExternalObjectDisplayData::SetEntities` are the entities in the `pfcCurveDescriptors` object.

Note

The method `wfcExternalObjectDisplayData::SetEntities` supports only `wfcENTITY_LINE` and `wfcENTITY_ARC` entities. However, you can draw polygons as multiple lines, and circles as arcs of extent 2 pi.

The methods `wfcExternalObjectDisplayData::GetEntityColors` and `wfcExternalObjectDisplayData::SetEntityColors` retrieve and set the display data for a list of entities and the color for each entity. The input argument to the method `wfcExternalObjectDisplayData::SetEntityColors` are the list of entities in the `pfcStdColors` object.

External Object Display Properties

Methods introduced:

- **wfcExternalObjectData::GetType**
- **wfcExternalObjectDisplayData::GetProperties**
- **wfcExternalObjectDisplayData::SetProperties**

By default, when users spin or zoom in on a model, external objects are subjected to the same spin and zoom scale as the model. In addition, by default external objects are always displayed, even if the owner or reference objects are suppressed. Setting external object display properties within display data enables you to change these default behaviors.

The method `wfcExternalObjectData::GetType` retrieves the type of property in specified external object data. To specify which type of property you want to retrieve, pass one of the values in the enumerated type `wfcExternalObjectDataType` to this method.

The methods `wfcExternalObjectDisplayData::GetProperties` and `wfcExternalObjectDisplayData::SetProperties` retrieve and set the display properties in the specified display data. The input argument to the method `wfcExternalObjectDisplayData::SetProperties` are the display properties in the `wfcExternalObjectDisplayDataProperties` object.

External Object Color

Methods introduced:

- **wfcExternalObjectDisplayData::GetDisplayColor**
- **wfcExternalObjectDisplayData::SetDisplayColor**

The enumerated type `pfcStdColor` specifies the colors available for external objects.

The methods `wfcExternalObjectDisplayData::GetDisplayColor` and `wfcExternalObjectDisplayData::SetDisplayColor` retrieve and set the object color in the specified display data.

Line Styles for External Objects

Methods introduced:

- **wfcExternalObjectDisplayData::GetLineStyle**
- **wfcExternalObjectDisplayData::SetLineStyle**

The enumerated type `pfcStdLineStyle` specifies the line styles available for specifies the line styles available for external objects.

The methods `wfcExternalObjectDisplayData::GetLineStyle` and `wfcExternalObjectDisplayData::SetLineStyle` retrieve and set the object line style in the specified display data.

External Object Scale

Methods introduced:

- **wfcExternalObjectDisplayData::GetScale**

- **wfcExternalObjectDisplayData::SetScale**

To vary the size of your external object without altering the entities themselves, you must specify an object scale factor as part of the display data.

The methods `wfcExternalObjectDisplayData::GetScale` and `wfcExternalObjectDisplayData::SetScale` retrieve and set the scale factor in the specified display data.

Transformation of the External Object

Methods introduced:

- **wfcExternalObjectDisplayData::GetTransormationMatrix**
- **wfcExternalObjectDisplayData::SetTransormationMatrix**

You can transform the local coordinates from model coordinates using the three-dimensional transformational matrix.

The method `wfcExternalObjectDisplayData::GetTransormationMatrix` retrieve the transformation matrix contained in a particular set of display data.

To a perform a coordinate transformation on an external object, use the method `wfcExternalObjectDisplayData::SetTransormationMatrix` to set the transformation matrix within the associated display data. The input argument to the method `wfcExternalObjectDisplayData::SetTransormationMatrix` is the transformation matrix in the `pfcMatrix3D` object.

Selection Data for External Objects

Methods introduced:

- **wfcExternalObjectSelectionBoxData::Create**
- **wfcExternalObjectSelectionBoxData::SetBoxes**
- **wfcExternalObjectSelectionBoxData::GetBoxes**

Use the method `wfcExternalObjectSelectionBoxData::Create` to create a selection data information for the specified external object. This method returns the selection data for the external object as a `wfcExternalObjectSelectionBoxData` object.

Selection boxes are specified as part of the external object selection data. These selection boxes indicate locations in which mouse selections will cause the external object to be selected. For the selection to be possible, you must designate a set of “hot spots,” or selection boxes for the object.

A selection box is defined by the pair of points contained in a `wfcSelectionBoxes` object. The coordinates of the points are specified in the external object's coordinate system (the default coordinates). The line between the points forms the diagonal of the selection box; the edges of the box lie parallel to the coordinate axes of the external object.

Note

PTC recommends that the size and arrangement of the selection boxes be dependent on the size and shape of the external object. If the external object is compact and uniformly distributed in all coordinate directions, one selection box will probably suffice.

However, if the external object is distributed non-uniformly, or is interfering with other objects, you must designate more specific locations at which selection should occur.

The method `wfcExternalObjectSelectionBoxData::GetBoxes` retrieves the list of selection boxes in a given selection data.

To set the selection boxes within the selection data, call the method `wfcExternalObjectSelectionBoxData::SetBoxes` and pass as input a pointer to a list of `wfcSelectionBoxes` objects. This enables your external object to have more than one associated selection box.

Manipulating External Object Data

Methods introduced:

- **wfcExternalObject::AddData**
- **wfcExternalObject::GetData**
- **wfcExternalObject::ModifyData**
- **wfcExternalObject::RemoveData**
- **wfcExternalObject::GetClass**

The methods in this section enable you to manipulate how the external object data relates to the object itself.

To add new data to an external object, pass the data handle `wfcExternalObjectData` to the method `wfcExternalObject::AddData`.

The method `wfcExternalObject::ModifyData` sets the contents of existing object data.

The method `wfcExternalObject::GetData` retrieves the handle for the display or selection data associated with an external object. To specify which type of data you want to retrieve, pass one of the values in the enumerated type `wfcExternalObjectDataType` to this method.

Use the method `wfcExternalObject::RemoveData` to remove data from an external object.

Use the method `wfcExternalObject::GetClass` to retrieve the class of an external object.

External Object References

You can use external object references to make external objects dependent on model geometry. For example, consider an external object that is modeled as the outward-pointing normal of a surface. Defining the surface as a reference enables the external object to behave appropriately when the surface is modified, deleted, or suppressed.

In general, an external object can reference any of the geometry that belongs to its owner. In addition, if the owner belongs to an assembly, the external object can also reference the geometry of other assembly components, provided that you supply a valid component path.

Note

Setting up the references for an external object does not fully define the dependency between the object and the reference. You must also specify the callback method to be called when some action is taken on the reference.

Creating External Object References

Methods introduced:

- **wfcExternalReferenceInfo::Create**
- **wfcExternalReferenceInfo::GetType**
- **wfcExternalReferenceInfo::SetType**
- **wfcExternalReferenceInfo::GetExtRefs**
- **wfcExternalReferenceInfo::SetExtRefs**

Use the method `wfcExternalReferenceInfo::Create` to create an external reference information object. The input arguments are as follows:

- *type*—Specify the type of the external reference.
- *extRefs*—Specify the sequence of external feature references.

You might need to use “reference types” to differentiate among the references of an external object.

The method `wfcExternalReferenceInfo::GetType` retrieve the reference type of the specified reference in an external reference information object. To specify which type of external reference to want to retrieve, pass one of the values in the enumerated type `wfcExternalReferenceType` to this method.

Use the method `wfcExternalReferenceInfo::SetType` to set a reference type.

The method `wfcExternalReferenceInfo::GetExtRefs` retrieve the sequence of external feature references in an external reference information object.

Use the method `wfcExternalReferenceInfo::SetExtRefs` to set the external feature reference. The input argument are the external feature references in the `wfcWExternalFeatureReferences` object.

25

Geometry Evaluation

This chapter describes geometry representation and discusses how to evaluate geometry using Creo Object TOOLKIT C++.

Geometry Traversal

- A simple rectangular face has one contour and four edges.
- A contour will traverse a boundary so that the part face is always on the right-hand side (RHS). For an external contour the direction of traversal is clockwise. For an internal contour the direction of traversal is counterclockwise.
- If a part is extruded from a sketch that has a U-shaped cross section there will be separate surfaces at each leg of the U-channel.
- If a part is extruded from a sketch that has a square-shaped cross section, and a slot feature is then cut into the part to make it look like a U-channel, there will be one surface across the legs of the U-channel. The original surface of the part is represented as one surface with a cut through it.

Geometry Terms

Following are definitions for some geometric terms:

- Surface—An ideal geometric representation, that is, an infinite plane.
- Face—A trimmed surface. A face has one or more contours.
- Contour—A closed loop on a face. A contour consists of multiple edges. A contour can belong to one face only.
- Edge—The boundary of a trimmed surface.

An edge of a solid is the intersection of two surfaces. The edge belongs to those two surfaces and to two contours. An edge of a datum surface can be either the intersection of two datum surfaces or the external boundary of the surface.

If the edge is the intersection of two datum surfaces it will belong to those two surfaces and to two contours. If the edge is the external boundary of the datum surface it will belong to that surface alone and to a single contour.

Traversing the Geometry of a Solid Block

Methods Introduced:

- **pfcModelItemOwner::ListItems**
- **pfcSurface::ListContours**
- **pfcContour::ListElements**

To traverse the geometry, follow these steps:

1. Starting at the top-level model, use `pfcModelItemOwner::ListItems` with an argument of `pfcITEM_SURFACE`.

2. Use `pfcSurface::ListContours` to list the contours contained in a specified surface.
3. Use `pfcContour::ListElements` to list the edges contained in the contour.

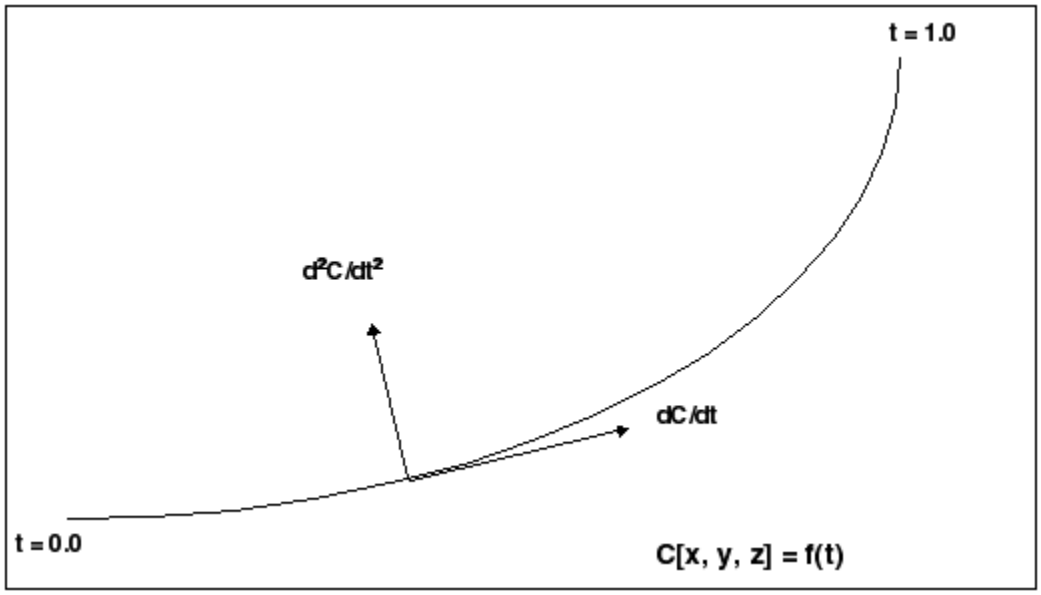
Curves and Edges

Datum curves, surface edges, and solid edges are represented in the same way in Creo Object TOOLKIT C++. You can get edges through geometry traversal or get a list of edges using the methods presented in the chapter ModelItem on page 316.

The t Parameter

The geometry of each edge or curve is represented as a set of three parametric equations that represent the values of x, y, and z as functions of an independent parameter, `t`. The `t` parameter varies from 0.0 at the start of the curve to 1.0 at the end of it.

The following figure illustrates curve and edge parameterization.



Curve and Edge Types

Solid edges and datum curves can be any of the following types:

- LINE—A straight line represented by the class interface `pfcLine`.
- ARC—A circular curve represented by the class interface `pfcArc`.

- SPLINE—A nonuniform cubic spline, represented by the class interface `pfcSpline`.
- B-SPLINE—A nonuniform rational B-spline curve or edge, represented by the class interface `pfcBSpline`.
- COMPOSITE CURVE—A combination of two or more curves, represented by the class interface `pfcCompositeCurve`. This is used for datum curves only.

See the appendix Geometry Representations on page 667 for the parameterization of each curve type. To determine what type of curve a `pfcEdge` or `pfcCurve` object represents, use the method `wfcWEdge::GetEdgeType` or `pfcCurveDescriptor::GetCurveType`.

Because each curve class inherits from `pfcGeomCurve`, you can use all the evaluation methods in `pfcGeomCurve` on any edge or curve.

The following curve types are not used in solid geometry and are reserved for future expansion:

- CIRCLE (`pfcCircle`)
- ELLIPSE (`pfcEllipse`)
- POLYGON (`pfcPolygon`)
- ARROW (`pfcArrow`)
- TEXT (`pfcText`)

Composite Curves

A composite curve is a curve that is made up of more than one segment and has no geometry of its own. A curve descriptor is a data object that describes the geometry of a curve or edge.

Methods Introduced:

- **wfcWCompositeCurveDescriptor::Create**
- **wfcWCompositeCurveDescriptor::SetCompDirections**
- **wfcWCompositeCurveDescriptor::GetCompDirections**
- **wfcWCompositeCurve::GetCompDirections**

Use the method `wfcWCompositeCurveDescriptor::Create` to create an instance of the object `wfcWCompositeCurveDescriptor` that contains information about the curve geometry of the specified composite curve.

In the curve descriptor, use the method `wfcWCompositeCurveDescriptor::SetCompDirections` to set the direction of the component for the composite curve. The valid values for the direction are defined by the enumerated type `wfcCurveDirection` and are as follows:

- `wfcCURVE_NO_FLIP`
- `wfcCURVE_FLIP`

Use the method `wfcWCompositeCurveDescriptor::GetCompDirections` to get the direction of the component of the specified composite curve.

Use the method `wfcWCompositeCurve::GetCompDirections` to get the direction of the component in a composite curve.

Evaluation of Curves and Edges

Methods Introduced:

- **pfcGeomCurve::Eval3DData**
- **pfcGeomCurve::EvalFromLength**
- **pfcGeomCurve::EvalParameter**
- **pfcGeomCurve::EvalLength**
- **pfcGeomCurve::EvalLengthBetween**
- **wfcWCurve::GetCurveData**
- **wfcWCurve::GetColorType**
- **wfcWCurve::SetColorType**
- **wfcWCurve::GetLineStyle**
- **wfcWCurve::SetLineStyle**

The methods in `pfcGeomCurve` provide information about any curve or edge.

The method `pfcGeomCurve::Eval3DData` returns a `pfcCurveXYZData` object with information on the point represented by the input parameter `t`. The method `pfcGeomCurve::EvalFromLength` returns a similar object with information on the point that is a specified distance from the starting point.

The method `pfcGeomCurve::EvalParameter` returns the `t` parameter that represents the input `pfcPoint3D` object.

Both `pfcGeomCurve::EvalLength` and `pfcGeomCurve::EvalLengthBetween` return numerical values for the length of the curve or edge.

Use the method `wfcWCurve::GetCurveData` to retrieve the geometric representation data for the specified curve.

Use the methods `wfcWCurve::GetColorType` and `wfcWCurve::SetColorType` to get and set the color of the specified curve using the enumerated type `pfcStdColor`.

Use the methods `wfcWCurve::GetLineStyle` and `wfcWCurve::SetLineStyle` to get and set the linestyle of the specified curve using the enumerated type `pfcStdLineStyle`.

Solid Edge Geometry

Methods Introduced:

- **pfcEdge::GetSurface1**
- **pfcEdge::GetSurface2**
- **pfcEdge::GetEdge1**
- **pfcEdge::GetEdge2**
- **pfcEdge::EvalUV**
- **pfcEdge::GetDirection**

> **Note**
> The methods in the interface `pfcEdge` provide information only for solid or surface edges.

- **wfcWEdge::GetEdgeType**
- **wfcWEdge::GetEdgeVertexData**

The methods `pfcEdge::GetSurface1` and `pfcEdge::GetSurface2` return the surfaces bounded by this edge. The methods `pfcEdge::GetEdge1` and `pfcEdge::GetEdge2` return the next edges in the two contours that contain this edge.

The method `pfcEdge::EvalUV` evaluates geometry information based on the UV parameters of one of the bounding surfaces.

The method `pfcEdge::GetDirection` returns a positive 1 if the edge is parameterized in the same direction as the containing contour, and –1 if the edge is parameterized opposite to the containing contour.

Use the method `wfcWEdge::GetEdgeType` to return the type of the specified edge.

Use the method `wfcWEdge::GetEdgeVertexData` to get a list of the edges and surfaces that meet at a specified solid vertex using the enumerated type `wfcEdgeEndType`. This method returns `NULL` if the specified edge is not visible in the current geometry.

Curve Descriptors

A curve descriptor is a data object that describes the geometry of a curve or edge. A curve descriptor describes the geometry of a curve without being a part of a specific model.

Methods Introduced:

- **pfcGeomCurve::GetCurveDescriptor**
- **pfcGeomCurve::GetNURBSRepresentation**
- **wfcWSolid::GetCurve**
- **wfcWCurve::GetCurveData**

The method `pfcGeomCurve::GetCurveDescriptor` returns a curve's geometry as a data object.

The method `pfcGeomCurve::GetNURBSRepresentation` returns a Non-Uniform Rational B-Spline Representation of a curve.

Note

Use the method `pfcGeomCurve::GetCurveDescriptor` to get the geometric information for an edge, access the `pfcCurveDescriptor` object for one edge using .

The method `wfcWSolid::GetCurve` returns the curve handle for the specified curve Id.

The method `wfcWCurve::GetCurveData` retrieves the geometric representation data for the specified curve.

Contours

Methods Introduced:

- **pfcSurface::ListContours**
- **pfcContour::GetInternalTraversal**
- **pfcContour::FindContainingContour**
- **pfcContour::EvalArea**
- **pfcContour::EvalOutline**
- **pfcContour::VerifyUV**
- **wfcContourDescriptor::Create**
- **wfcContourDescriptor::SetContourTraversal**

- **wfcContourDescriptor::GetContourTraversal**
- **wfcContourDescriptor::SetEdgeIds**
- **wfcContourDescriptor::GetEdgeIds**

Contours are a series of edges that completely bound a surface. A contour is not a `pfcModelItem`. You cannot get contours using the methods that get different types of `pfcModelItem`. Use the method `pfcSurface::ListContours` to get contours from their containing surfaces.

The method `pfcContour::GetInternalTraversal` returns a `pfcContourTraversal` enumerated type that identifies whether a given contour is on the outside or inside of a containing surface.

Use the method `pfcContour::FindContainingContour` to find the contour that entirely encloses the specified contour.

The method `pfcContour::EvalArea` provides the area enclosed by the contour.

The method `pfcContour::EvalOutline` returns the points that make up the bounding rectangle of the contour.

Use the method `pfcContour::VerifyUV` to determine whether the given `pfcUVParams` argument lies inside the contour, on the boundary, or outside the contour.

Note

To make the methods of `wfcWContour` available in `pfcContour`, cast `pfcContour` to `wfcWContour`. Refer to the section Casting of Creo Object TOOLKIT C++ pfc Classes to wfc Classes on page 21 for more information on casting pfc classes to wfc classes.

Use the method `wfcContourDescriptor::Create` to create an instance of the object `wfcContourDescriptor`.This object contains information about a contour such as its traversal (internal or external) and identifiers of the edges that make up the contour.

Use the method `wfcContourDescriptor::SetContourTraversal` and `wfcContourDescriptor::GetContourTraversal` to set and get the type of contour traversal using the enumerated type `pfcContourTraversal`. The valid values are:

- `pfcCONTOUR_TRAV_INTERNAL`—Specifies the traversal of the internal contours.
- `pfcCONTOUR_TRAV_EXTERNAL`—Specifies the traversal of the external contours.

Use the method `wfcContourDescriptor::SetEdgeIds` and `wfcContourDescriptor::GetEdgeIds` to set and get an array of identifiers of the edges of the contour.

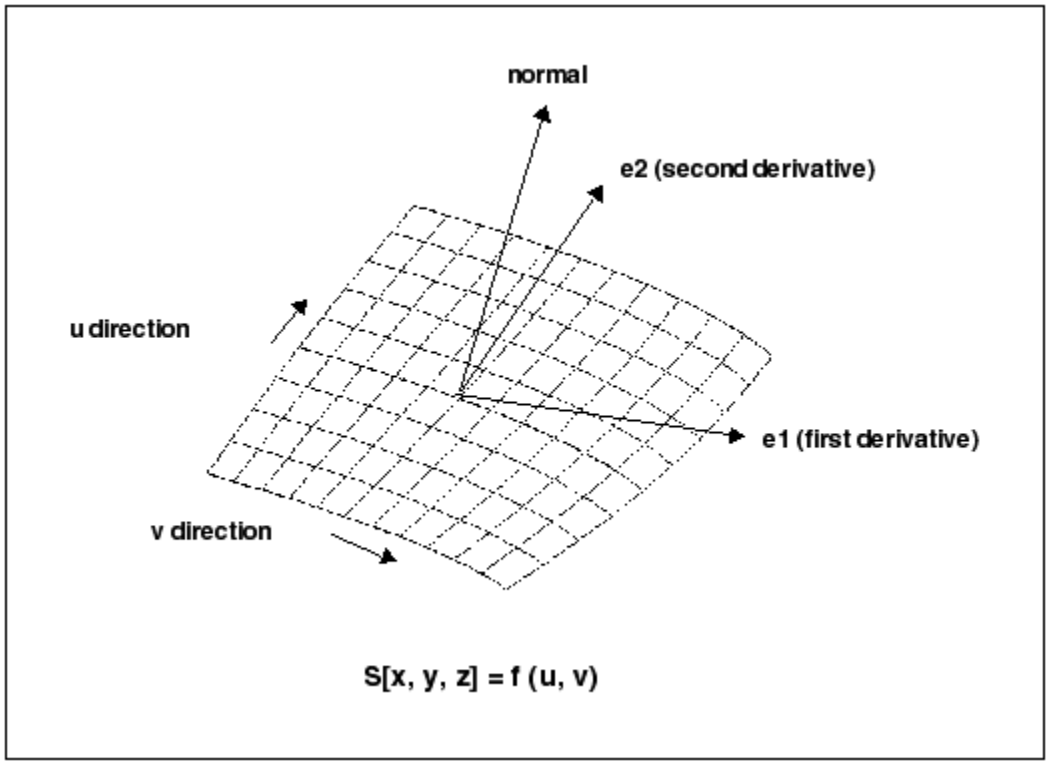
Surfaces

Using Creo Object TOOLKIT C++ you access datum and solid surfaces in the same way.

UV Parameterization

A surface in Creo is described as a series of parametric equations where two parameters, `u` and `v`, determine the x, y, and z coordinates. Unlike the edge parameter, `t`, these parameters need not start at 0.0, nor are they limited to 1.0.

The figure on the following page illustrates surface parameterization.



Surface Types

Surfaces within Creo can be any of the following types:

- PLANE—A planar surface represented by the class interface `pfcPlane`.

- CYLINDER—A cylindrical surface represented by the class interface `pfcCylinder`.
- CONE—A conic surface region represented by the class interface `pfcCone`.
- TORUS—A toroidal surface region represented by the class interface `pfcTorus`.
- REVOLVED SURFACE—Generated by revolving a curve about an axis. This is represented by the class interface `pfcRevSurface`.
- RULED SURFACE—Generated by interpolating linearly between two curve entities. This is represented by the class interface `pfcRuledSurface`.
- TABULATED CYLINDER—Generated by extruding a curve linearly. This is represented by the class interface `pfcTabulatedCylinder`.
- COONS PATCH—A coons patch is used to blend surfaces together. It is represented by the class interface `pfcCoonsPatch`.
- FILLET SURFACE—A filleted surface is found where a round or fillet is placed on a curved edge or an edge with a non-consistant arc radii. On a straight edge a cylinder is used to represent a fillet. This is represented by the class interface `pfcFilletedSurface`.
- SPLINE SURFACE— A nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. This is represented by the class interface `pfcSplineSurface`.
- NURBS SURFACE—A NURBS surface is defined by basic functions (in *u* and *v*), expandable arrays of knots, weights, and control points. This is represented by the class interface `pfcNURBSSurface`.
- CYLINDRICAL SPLINE SURFACE— A cylindrical spline surface is a nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. This is represented by the class interface `pfcCylindricalSplineSurface`.

To determine which type of surface a `pfcSurface` object represents, access the surface type using `pfcSurface::GetSurfaceType`.

Surface Information

Methods Introduced:

- **pfcSurface::GetSurfaceType**
- **pfcSurface::GetXYZExtents**
- **pfcSurface::GetUVExtents**
- **pfcSurface::GetOrientation**

The method `pfcSurface::GetSurfaceType` returns the type of the surface using the enumerated data type `pfcSurfaceType` and the valid values are:

- `pfcSURFACE_PLANE`
- `pfcSURFACE_CYLINDER`
- `pfcSURFACE_CONE`
- `pfcSURFACE_TORUS`
- `pfcSURFACE_RULED`
- `pfcSURFACE_REVOLVED`
- `pfcSURFACE_TABULATED_CYLINDER`
- `pfcSURFACE_FILLET`
- `pfcSURFACE_COONS_PATCH`
- `pfcSURFACE_SPLINE`
- `pfcSURFACE_NURBS`
- `pfcSURFACE_CYLINDRICAL_SPLINE`
- `pfcSURFACE_FOREIGN`
- `pfcSURFACE_SPL2DER`

The method `pfcSurface::GetXYZExtents` returns the XYZ points at the corners of the surface.

The method `pfcSurface::GetUVExtents` returns the UV parameters at the corners of the surface.

The method `pfcSurface::GetOrientation` returns the orientation of the surface using the enumerated data type `pfcSurfaceOrientation` and the valid values are:

- `pfcSURFACEORIENT_NONE`—Surface that does not need orientation. For example, a solid surface needs orientation and therefore cannot be specified.
- `pfcSURFACEORIENT_OUTWARD`—Surface that has oriented outward away from the solid model. du X dv points outward.
- `pfcSURFACEORIENT_INWARD`—Surface that has oriented inward toward the solid model. du X dv points inward.

Evaluation of Surfaces

Surface methods allow you to use multiple surface information to calculate, evaluate, determine, and examine surface functions and problems.

Methods Introduced:

- **pfcSurface::GetOwnerQuilt**
- **pfcSurface::EvalClosestPoint**

- **pfcSurface::EvalClosestPointOnSurface**
- **pfcSurface::Eval3DData**
- **pfcSurface::EvalParameters**
- **pfcSurface::EvalArea**
- **pfcSurface::EvalDiameter**
- **pfcSurface::EvalPrincipalCurv**
- **pfcSurface::VerifyUV**
- **pfcSurface::EvalMaximum**
- **pfcSurface::EvalMinimum**
- **pfcSurface::ListSameSurfaces**
- **wfcWSurface::GetNextSurface**

The method `pfcSurface::GetOwnerQuilt` returns the `pfcQuilt` object that contains the datum surface.

The method `pfcSurface::EvalClosestPoint` projects a three-dimensional point onto the surface. Use the method `pfcSurface::EvalClosestPointOnSurface` to determine whether the specified three-dimensional point is on the surface, within the accuracy of the part. If it is, the method returns the point that is exactly on the surface. Otherwise the method returns null.

The method `pfcSurface::Eval3DData` returns a `pfcSurfXYZData` object that contains information about the surface at the specified *u* and *v* parameters. The method `pfcSurface::EvalParameters` returns the *u* and *v* parameters that correspond to the specified three-dimensional point.

The method `pfcSurface::EvalArea` returns the area of the surface, whereas `pfcSurface::EvalDiameter` returns the diameter of the surface. If the diameter varies the optional `pfcUVParams` argument identifies where the diameter should be evaluated.

The method `pfcSurface::EvalPrincipalCurv` returns a `pfcCurvatureData` object with information regarding the curvature of the surface at the specified *u* and *v* parameters.

Use the method `pfcSurface::VerifyUV` to determine whether the `pfcUVParams` are actually within the boundary of the surface.

The methods `pfcSurface::EvalMaximum` and `pfcSurface::EvalMinimum` return the three-dimensional point on the surface that is the furthest in the direction of (or away from) the specified vector.

The method `pfcSurface::ListSameSurfaces` identifies other surfaces that are tangent and connect to the given surface.

The method `wfcWSurface::GetNextSurface` returns the next surface in the surface list. If no surface is present in the surface list, the method returns the value NULL.

Note

Obtain the next surface again, in case the model geometry has changed.

Surface Descriptors

A surface descriptor is a data object that describes the shape and geometry of a specified surface. A surface descriptor allows you to describe a surface in 3D without an owner ID.

Methods Introduced:

- **pfcSurface::GetSurfaceDescriptor**
- **pfcSurface::GetNURBSRepresentation**
- **wfcWSurfaceDescriptor::SetContourData**
- **wfcWSurfaceDescriptor::GetContourData**
- **wfcWSurfaceDescriptor::SetSurfaceId**
- **wfcWSurfaceDescriptor::GetSurfaceId**

The method `pfcSurface::GetSurfaceDescriptor` returns a surfaces geometry as a data object.

The method `pfcSurface::GetNURBSRepresentation` returns a Non-Uniform Rational B-Spline Representation of a surface.

Use the method `wfcWSurfaceDescriptor::SetContourData` and `wfcWSurfaceDescriptor::GetContourData` to set and get the contour data information from an array of `wfcContourDescriptors`.

Use the method `wfcWSurfaceDescriptor::SetSurfaceId` and `wfcWSurfaceDescriptor::GetSurfaceId` to set and get the Id of the specified surface.

Axes, Coordinate Systems, and Points

Coordinate axes, datum points, and coordinate systems are all model items. Use the methods that return `ModelItems` to get one of these geometry objects. Refer to the chapter ModelItem on page 316 for additional information.

Evaluation of ModelItems

Methods Introduced:

- **pfcAxis::GetSurf**
- **wfcWAxis::GetAxisData**
- **wfcWSolid::GetAxis**
- **pfcCoordSystem::GetCoordSys**
- **wfcWSolid::GetCsys**
- **pfcPoint::GetPoint**

The method `pfcAxis::GetSurf` returns the revolved surface that uses the axis.

Use the method `wfcWAxis::GetAxisData` to return the geometric representation data for the specified axis as a object of the class `pfcCurveDescriptor`.

Use the method `wfcWSolid::GetAxis` to initialize the axis handle. Specify the axis Id as the input parameter for this method.

The method `pfcCoordSystem::GetCoordSys` returns the `pfcTransform3D` object (which includes the origin and x-, y-, and z- axes) that defines the coordinate system.

Use the method `wfcWSolid::GetCsys` to return the coordinate system handle as a object of the class `wfcWCsys`.

The method `pfcPoint::GetPoint` returns the xyz coordinates of the datum point.

Interference

Creo assemblies can contain interferences between components when constraint by certain rules defined by the user. The `pfcInterference` package allows the user to detect and analyze any interferences within the assembly. The analysis of this functionality should be looked at from two standpoints: global and selection based analysis.

Methods Introduced:

- **pfcCreateGlobalEvaluator**
- **pfcGlobalEvaluator::ComputeGlobalInterference**
- **pfcGlobalEvaluator::GetAssem**
- **pfcGlobalEvaluator::SetAssem**
- **pfcGlobalInterference::GetVolume**
- **pfcGlobalInterference::GetSelParts**

To compute all the interferences within an Assembly one has to call `pfcCreateGlobalEvaluator` with a `pfcAssembly` object as an argument. This call returns a`pfcGlobalEvaluator` object. The `pfcCretaeGlobalEvaluator` can be used to extract an assembly object or to set an assembly object for the interference computation.

The methods `pfcGlobalEvaluator::GetAssem` and `pfcGlobalEvaluator::SetAssem` with `pfcAssembly` as an argument allow you to do exactly that.

The method `pfcGlobalEvaluator::ComputeGlobalInterference` determines the set of all the interferences within the assembly.

This method will return a sequence of `pfcGlobalInterference` objects or null if there are no interfering parts. Each object contains a pair of intersecting parts and an object representing the interference volume, which can be extracted by using `pfcGlobalInterference::GetSelParts` and `pfcGlobalInterference::GetVolume` respectively.

Analyzing Interference Information

Methods Introduced:

- **pfcSelectionPair::Create**
- **pfcCreateSelectionEvaluator**
- **pfcSelectionEvaluator::GetSelections**
- **pfcSelectionEvaluator::SetSelections**
- **pfcSelectionEvaluator::ComputeInterference**
- **pfcSelectionEvaluator::ComputeClearance**
- **pfcSelectionEvaluator::ComputeNearestCriticalDistance**

The method `pfcSelectionPair::Create` creates a `pfcSelectionPair` object using two `pfcSelection` objects as arguments.

A return from this method will serve as an argument to `pfcCreateSelectionEvaluator`, which will provide a way to determine the interference data between the two selections.

`pfcSelectionEvaluator::GetSelections` and `pfcSelectionEvaluator::SetSelections` will extract and set the object to be evaluated respectively.

`pfcSelectionEvaluator::ComputeInterference` determines the interfering information about the provided selections. This method will return the `pfcInterferenceVolume` object or null if the selections do no interfere.

`pfcSelectionEvaluator::ComputeClearance` computes the clearance data for the two selection. This method returns a `pfcClearanceData` object, which can be used to obtain and set clearance distance, nearest points between selections, and a boolean `IsInterferening` variable.

`pfcSelectionEvaluator::ComputeNearestCriticalDistance` finds a critical point of the distance function between two selections.

This method returns a `pfcCriticalDistanceData` object, which is used to determine and set critical points, surface parameters, and critical distance between points.

Analyzing Interference Volume

Methods Introduced:

- **pfcInterferenceVolume::ComputeVolume**
- **pfcInterferenceVolume::Highlight**
- **pfcInterferenceVolume::GetBoundaries**

The method `pfcInterferenceVolume::ComputeVolume` will calculate a value for interfering volume.

The method `pfcInterferenceVolume::Highlight` will highlight the interfering volume with the color provided in the argument to the function.

The method `pfcInterferenceVolume::GetBoundaries` will return a set of boundary surface descriptors for the interference volume.

Tessellation

You can calculate tessellation for different types of Creo geometry. The tessellation geometry is made up of:

- Small lines—For edges and curves.
- Triangles—For surfaces and solid models.

Surface Tessellation

Methods Introduced:

- **wfcSurfaceTessellationInput::Create**
- **wfcSurfaceTessellationInput::GetAngleControl**
- **wfcSurfaceTessellationInput::SetAngleControl**
- **wfcSurfaceTessellationInput::GetStepSize**
- **wfcSurfaceTessellationInput::SetStepSize**
- **wfcSurfaceTessellationInput::GetProjection**

- **wfcSurfaceTessellationInput::SetProjection**
- **wfcSurfaceTessellationInput::GetCsysData**
- **wfcSurfaceTessellationInput::SetCsysData**
- **wfcSurfaceTessellationInput::GetChordHeight**
- **wfcSurfaceTessellationInput::SetChordHeight**
- **wfcWSurface::GetTessellation**
- **wfcTessellation::GetFacetVertexIndices**
- **wfcTessellation::GetUVParams**
- **wfcTessellation::GetVectors**
- **wfcTessellation::GetVertices**

Use the method `wfcSurfaceTessellationInput::Create` to create an instance of the object `wfcSurfaceTessellationInput` that contains information about the surface tessellation. Specify the following parameters as input arguments to create the surface tessellation:

- `AngleControl`—Regulates the amount of additional improvement provided along curves with small radii. Specify a value from the range 0.0 to 1.0.
- `StepSize`—Controls the fineness of the triangulations for all surfaces. The values range from five times of the model accuracy to the model size with a default value of (model size)/30.
- `Projection`—Specifies the parameters used to calculate the UV projection for the texture mapping. The types of UV projection are defined in the enumerated type `wfcSurfaceTessellationProjection` and are as follows:
  - `wfcSRFTESS_DEFAULT_PROJECTION`—Provides the UV parameters for the tessellation points that map to a plane whose U and V extents are [0,1] each.
  - `wfcSRFTESS_PLANAR_PROJECTION`—Projects the UV parameters using a planar transform, where u=x, v=y, and z is ignored.
  - `wfcSRFTESS_CYLINDRICAL_PROJECTION`—Projects the UV parameters using a cylindrical transform, where x=r*cos(theta), y=r*sin (theta), u=theta, v=z, and r is ignored.
  - `wfcSRFTESS_SPHERICAL_PROJECTION`—Projects the UV parameters onto a sphere, where x=r*cos(theta)*sin(phi), y=r*sin(theta) *sin(phi), z=r*cos(phi), u=theta, v=phi, and r is ignored.
  - `wfcSRFTESS_NO_PROJECTION`—Provides the unmodified UV parameters for the tessellation points.

  .
- `CsysData`—Specifies the coordinate system data, including the transformation matrix, origin, and axes information.

- `ChordHeight`—Specifies the maximum distance between a chord and a surface.

Use the methods `wfcSurfaceTessellationInput::GetAngleControl` and `wfcSurfaceTessellationInput::SetAngleControl` to get and set the value of the angle control used for surface tessellation.

Use the methods `wfcSurfaceTessellationInput::GetStepSize` and `wfcSurfaceTessellationInput::SetStepSize` to get and set the maximum value for the step size used for the surface tessellation.

Use the methods `wfcSurfaceTessellationInput::GetProjection` and `wfcSurfaceTessellationInput::SetProjection` to get and set the parameters used to calculate the UV projection for the texture mapping using the enumerated value `wfcSurfaceTessellationProjection`.

Use the methods `wfcSurfaceTessellationInput::GetCsysData` and `wfcSurfaceTessellationInput::SetCsysData` to get and set the coordinate system data for the surface tessellation.

Use the methods `wfcSurfaceTessellationInput::GetChordHeight` and `wfcSurfaceTessellationInput::SetChordHeight` to get and set the chord height used for surface tessellation.

Use the method `wfcWSurface::GetTessellation` to calculate the tessellation of a given surface. Specify the object `wfcSurfaceTessellationInput` as the input argument for this method. This method returns an object `wfcTessellation` that contains the tessellation data.

The method `wfcTessellation::GetFacetVertexIndices` returns a sequence of facet vertices for the specified surface tessellation data.

Use the method `wfcTessellation::GetUVParams` to obtain the UV parameters for each of the tessellation vertices

Use the method `wfcTessellation::GetVectors` to obtain the normal vectors for each of the tessellation vertices

Use the method `wfcTessellation::GetVertices` to obtain the vertices for the tessellation for a specified surface.

Curve and Edge Tessellation

Methods Introduced:

- **wfcWCurve::GetCurveTessellation**
- **wfcWEdge::GetEdgeTessellation**

Use the method `wfcWCurve::GetCurveTessellation` to retrieve the curve tessellation for a datum curve as a `wfcCurveTessellation` pointer.

Use the method `wfcWEdge::GetEdgeTessellation` to retrieve the edge tessellation for a specified edge as a `wfcEdgeTessellation` pointer. .

Part and Assembly Tessellation

Methods Introduced:

- **wfcWSolid::Tessellate**
- **wfcSurfaceTessellationData::GetFacetVertexIndices**
- **wfcSurfaceTessellationData::GetNormals**
- **wfcSurfaceTessellationData::GetNumberOfFacets**
- **wfcSurfaceTessellationData::GetNumberOfVertices**
- **wfcSurfaceTessellationData::GetSurface**
- **wfcSurfaceTessellationData::GetVertexVectors**

From Creo 3.0 M110 onward, the method `wfcWPart::Tessellate` has been deprecated. Use the method `wfcWSolid::Tessellate` instead. The method `wfcWSolid::Tessellate` tessellates each surface of the specified model. On parts, `wfcWSolid::Tessellate` acts on all surfaces whereas on assemblies, this method acts only on surfaces that belong to the assembly, that is, it does not tessellate surfaces of the assembly components. Specify the `ChordHeight`, `AngleControl` and `IncludeQuilts` as input parameters to this method. For more information on `ChordHeight`, `AngleControl` and `IncludeQuilts`, refer to the section Surface Tessellation on page 425.

Use the method `wfcSurfaceTessellationData::GetFacetVertexIndices` to obtain an array of facet vertices in the tessellated surface of the specified part.

Use the method `wfcSurfaceTessellationData::GetNormals` to get an array of normal vectors present in the tessellated surface of the specified part.

Use the method `wfcSurfaceTessellationData::GetNumberOfFacets` to obtain the number of facets present in the tessellated surface of the specified part.

Use the method `wfcSurfaceTessellationData::GetNumberOfVertices` to obtain the number of vertices present in the tessellated surface of the specified part.

Use the method `wfcSurfaceTessellationData::GetSurface` to get the surface which has been tessellated in the specified part.

Use the method `wfcSurfaceTessellationData::GetVertexVectors` to get an array of vertex vectors present in the tessellated surface of the specified part.

Geometry Objects

Geometry of Points

Method Introduced:

- **wfcWPoint::GetCoordinates**
- **wfcWSolid::ProjectPoint**
- **wfcProjectionInfo::GetSurface**
- **wfcProjectionInfo::GetUVParam**
- **wfcWPart::FindGeometry**
- **wfcWPoint::FindIntolerance**
- **wfcWPoint::GetCoordinates**
- **wfcWPointTolerance::GetTolerance**
- **wfcWPointTolerance::GetWithinTolerance**

Use the method `wfcWPoint::GetCoordinates` to retrieve the X, Y, and Z coordinates of the specified point.

The method `wfcWSolid::ProjectPoint` projects a point normal on the solid with in the specified maximum distance, and returns the surface where the point is projected along with the UV parameters of the surface as a `wfcProjectionInfo` object.

Note

The method `wfcWSolid::ProjectPoint` is supported only for parts.

The method `wfcProjectionInfo::GetSurface` returns the surface on which the specified point has been projected.

Use the method `wfcProjectionInfo::GetUVParam` to get the UV point on the surface where the specified point has been projected.

Use the method `wfcWPart::FindGeometry` to determine the surfaces or edges on which the specified point is located.

Note

This method does not return the neighboring surfaces, if the specified point lies on an edge.

Use the method `wfcWPoint::FindIntolerance` to determine if two points are co-incident, that is, whether the distance between two points is within the Creo tolerances.

Use the method `wfcWPoint::GetCoordinates` to retrieve the X, Y and Z coordinates of the specified point.

The method `wfcWPointTolerance::GetTolerance` retrieves the amount by which the distance between two points exceeds tolerance.

The method `wfcWPointTolerance::GetWithinTolerance` returns true if distance between points is within tolerance and false if not.

Geometry of Coordinate System Datums

Methods Introduced:

- **wfcWCsysData::Create**
- **wfcWCsysData::GetOrigin**
- **wfcWCsysData::SetOrigin**
- **wfcWCsysData::GetXAxis**
- **wfcWCsysData::SetXAxis**
- **wfcWCsysData::GetYAxis**
- **wfcWCsysData::SetYAxis**
- **wfcWCsysData::GetZAxis**
- **wfcWCsysData::SetZAxis**

The method `wfcWCsysData::Create` creates a data object that contains information about the coordinate system. The input arguments are:

- *XAxis*—Specifies the X-axis of the coordinate system.
- *YAxis*—Specifies the Y-axis of the coordinate system.
- *ZAxis*—Specifies the Z-axis of the coordinate system.
- *Origin*—Specifies the origin of the coordinate system.

Use the methods `wfcWCsysData::GetOrigin` and `wfcWCsysData::SetOrigin` to get and set the origin of the coordinate system using the `pfcVector3D` object.

Use the methods `wfcWCsysData::GetXAxis` and `wfcWCsysData::SetXAxis` to get and set the X–axis of the coordinate system using the `pfcVector3D` object.

Use the methods `wfcWCsysData::GetYAxis` and `wfcWCsysData::SetYAxis` to get and set the Y–axis of the coordinate system using the `pfcVector3D` object.

Use the methods `wfcWCsysData::GetZAxis` and `wfcWCsysData::SetZAxis` to get and set the Z–axis of the coordinate system using the `pfcVector3D` object.

Geometry of Solid Edges

The methods described in this section allow you get and set the parameters of edges in a part. The geometric edges form a contour, which further forms the surface of a part. Thus, the edges define the extent of a surface. Every edge in a part has two surfaces adjacent to it.

Methods Introduced:

- **wfcEdgeDescriptor::Create**
- **wfcEdgeDescriptor::GetId**
- **wfcEdgeDescriptor::SetId**
- **wfcEdgeDescriptor::GetEdgeSurface1**
- **wfcEdgeDescriptor::SetEdgeSurface1**
- **wfcEdgeDescriptor::GetEdgeSurface2**
- **wfcEdgeDescriptor::SetEdgeSurface2**
- **wfcEdgeSurfaceData::Create**
- **wfcEdgeSurfaceData::GetEdgeSurfaceId**
- **wfcEdgeSurfaceData::SetEdgeSurfaceId**
- **wfcEdgeSurfaceData::GetDirection**
- **wfcEdgeSurfaceData::SetDirection**
- **wfcEdgeSurfaceData::GetUVParamsSequence**
- **wfcEdgeSurfaceData::SetUVParamsSequence**
- **wfcEdgeSurfaceData::GetUVCurveData**
- **wfcEdgeSurfaceData::SetUVCurveData**
- **wfcEdgeDescriptor::GetXYZCurveData**
- **wfcEdgeDescriptor::SetXYZCurveData**

The method `wfcEdgeDescriptor::Create` creates a data object that contains information about the edges in a solid surface.

The method `wfcEdgeDescriptor::GetId` returns the ID of the specified edge. Use the method `wfcEdgeDescriptor::SetId` to set ID of the edge.

The method `wfcEdgeDescriptor::GetEdgeSurface1` retrieves all the information of the first surface of an edge as a `wfcEdgeSurfaceData` object. Use the method `wfcEdgeDescriptor::SetEdgeSurface1` to set the parameters of the first surface of an edge.

The method `wfcEdgeDescriptor::GetEdgeSurface2` retrieves all the information of the second surface of an edge as a `wfcEdgeSurfaceData` object. Use the method `wfcEdgeDescriptor::SetEdgeSurface2` to set the parameters of the second surface of an edge.

The method `wfcEdgeSurfaceData::Create` to create a data object that contains information about the surface adjacent to an edge.

The method `wfcEdgeSurfaceData::GetEdgeSurfaceId` returns the ID of the specified surface. Use the method `wfcEdgeSurfaceData::SetEdgeSurfaceId` to set the ID of the specified surface.

The method `wfcEdgeSurfaceData::GetDirection` identifies whether an edge is parameterized along or against the direction of the specified surface. Each edge belongs to two surfaces. The edge will be in the same direction as one surface, and in the opposite direction of the other surface. Use the method `wfcEdgeSurfaceData::SetDirection` to set the direction of the edge along the specified surface.

The method `wfcEdgeSurfaceData::GetUVParamsSequence` returns an array of UV points data for the curve. Use the method `wfcEdgeSurfaceData::SetUVParamsSequence` to set the UV points for the curve.

The method `wfcEdgeSurfaceData::GetUVCurveData` returns an array of UV curve data for the specified surface. Use the method `wfcEdgeSurfaceData::SetUVCurveData` to set the parameters for the UV curve.

The method `wfcEdgeDescriptor::GetXYZCurveData` returns a data object that contains information about the geometry type for the XYZ curve. Use the method `wfcEdgeDescriptor::SetXYZCurveData` to set the geometry type for the XYZ curve.

Geometry of Quilts

Methods Introduced:

- **wfcWQuilt::IsBackupGeometry**
- **wfcWQuilt::GetVolume**
- **wfcWSolid::GetQuilt**
- **wfcQuiltData::Create**
- **wfcQuiltData::SetQuiltId**
- **wfcQuiltData::GetQuiltId**
- **wfcQuiltData::SetSurfaceDescriptors**
- **wfcQuiltData::GetSurfaceDescriptors**

The method `wfcWSolid::GetQuilt` returns a data object which contains information about the quilt for the specified quilt Id.

Use the method `wfcWQuilt::IsBackupGeometry` to identify if the specified quilt belongs to the invisible Copy Geometry backup feature.

Use the method `wfcWQuilt::GetVolume` to get the volume of a closed quilt.

Use the method `wfcQuiltData::Create` to create an instance of the object `wfcQuiltData` that contains information about the quilt data. Specify the quilt ID and a pointer to the surface descriptor as the input arguments for this method.

Use the methods `wfcQuiltData::SetQuiltId` and `wfcQuiltData::GetQuiltId` to set and get the quilt id.

Use the method `wfcQuiltData::GetSurfaceDescriptors` to access information about a quilt surface from an array of `wfcWSurfaceDescriptors`. Use the method `wfcQuiltData::SetSurfaceDescriptors` to set the values for the surface descriptor.

Geometry of Surfaces

Method Introduced:

- **wfcWPart::GetVolumeInfo**
- **wfcVolumeSurfaceInfo::GetNumberOfVolumes**
- **wfcVolumeSurfaceInfo::GetSurfaceInfos**
- **wfcBoundingSurfaceInfo::GetNumberOfSurfaces**
- **wfcBoundingSurfaceInfo::GetSurfaceIds**

The method `wfcWPart::GetVolumeInfo` analyzes and returns the number of connected volumes of a part and the surfaces that bound the volume as a `wfcVolumeSurfaceInfo` object. Connect volumes are disjoint components of a solid. It comprises of all the shells and voids that lie in the same maximal ambient shell. The maximal ambient shell is an external shell and is not located inside any other shell.

The method `wfcVolumeSurfaceInfo::GetNumberOfVolumes` returns the number of volumes in the part.

The method `wfcVolumeSurfaceInfo::GetSurfaceInfos` returns the information of the bounding surfaces as a `wfcBoundingSurfaceInfos` object.

Use the method `wfcBoundingSurfaceInfo::GetNumberOfSurfaces` returns the number of surfaces that bound the connected volumes.

The method `wfcBoundingSurfaceInfo::GetSurfaceIds` returns the IDs of the surfaces that bound the connected volumes.

Geometry of datums

Method Introduced:

- **wfcDatumObject::GetDatumObjectType**
- **wfcDatumData::Create**
- **wfcDatumData::SetDatumObject**
- **wfcDatumData::GetDatumObject**
- **wfcDatumData::SetId**
- **wfcDatumData::GetId**
- **wfcDatumData::SetName**
- **wfcDatumData::GetName**
- **wfcCsysDatumObject::Create**
- **wfcCsysDatumObject::GetCsysData**
- **wfcCsysDatumObject::SetCsysData**
- **wfcCurveDatumObject::Create**
- **wfcCurveDatumObject::GetCurve**
- **wfcCurveDatumObject::SetCurve**
- **wfcPlaneDatumObject::Create**
- **wfcPlaneDatumObject::GetPlaneData**
- **wfcPlaneDatumObject::SetPlaneData**
- **wfcWPlaneData::Create**
- **wfcWPlaneData::GetOrigin**
- **wfcWPlaneData::SetOrigin**
- **wfcWPlaneData::GetXAxis**
- **wfcWPlaneData::SetXAxis**
- **wfcWPlaneData::GetYAxis**
- **wfcWPlaneData::SetYAxis**
- **wfcWPlaneData::GetZAxis**
- **wfcWPlaneData::SetZAxis**

Use the method `wfcDatumObject::GetDatumObjectType` to get the type of the datum object. The valid values for the datum object type are defined by the enumerated type `wfcDatumObjectType` and are as follows:

- `wfcDATUM_CURVE`—Specifies that the datum object is a curve.
- `wfcDATUM_PLANE`—Specifies that the datum object is a plane.
- `wfcDATUM_CSYS`—Specifies that the datum object is a coordinate system.

Use the method `wfcDatumData::Create` to create an instance of the object `wfcDatumData` that contains information about the datum data. Specify the Name, Id and the DatumObject as the input parameters to this method.

Use the methods `wfcDatumData::SetDatumObject` and `wfcDatumData::GetDatumObject` to set and get the datum object as on object of the class `wfcDatumObject`.

Use the methods `wfcDatumData::SetId` and `wfcDatumData::GetId` to set and get the datum Id for the specified datum.

Use the methods `wfcDatumData::SetId` and `wfcDatumData::GetId` to set and get the name of the datum.

The method `wfcCsysDatumObject::Create` creates a coordinate system datum object using `wfcWCsysData` object input argument.

Use the methods `wfcCsysDatumObject::GetCsysData` and `wfcCsysDatumObject::SetCsysData` to get and set the coordinate system data using the `wfcWCsysData` object.

The method `wfcCurveDatumObject::Create` creates a curve datum object using `pfcCurveDescriptor` object input argument.

Use the methods `wfcCurveDatumObject::GetCurve` and `wfcCurveDatumObject::SetCurve` to get and set the curve using the `pfcCurveDescriptor` object.

The method `wfcPlaneDatumObject::Create` creates a plane datum object using `wfcWPlaneData` object input argument.

Use the methods `wfcPlaneDatumObject::GetPlaneData` and `wfcPlaneDatumObject::SetPlaneData` to get and set the plane using the `wfcWPlaneData` object.

The method `wfcWPlaneData::Create` creates a data object that contains information about the plane data. The input arguments are:

- *XAxis*—Specifies the X-axis of the coordinate system.
- *YAxis*—Specifies the Y-axis of the coordinate system.
- *ZAxis*—Specifies the Z-axis of the coordinate system.
- *Origin*—Specifies the origin of the coordinate system.

Use the methods `wfcWPlaneData::GetOrigin` and `wfcWPlaneData::SetOrigin` to get and set the origin of the plane using the `pfcPoint3D` object.

Use the methods `wfcWPlaneData::GetXAxis` and `wfcWPlaneData::SetXAxis` to get and set the X-axis of the plane using the `pfcVector3D` object.

Use the methods `wfcWPlaneData::GetYAxis` and `wfcWPlaneData::SetYAxis` to get and set the Y-axis of the plane using the `pfcVector3D` object.

Use the methods `wfcWPlaneData::GetZAxis` and `wfcWPlaneData::SetZAxis` to get and set the Z-axis of the plane using the `pfcVector3D` object.

Tracing a Ray

Method Introduced:

- **wfcWModel::ComputeRayIntersections**
- **wfcRay::Create**
- **wfcRay::SetPoint**
- **wfcRay::GetPoint**
- **wfcRay::SetVector**
- **wfcRay::GetVector**

The method `wfcWModel::ComputeRayIntersections` returns a list of intersections between a ray and a model as a `pfcSelections` object. The method finds intersections in both directions from the start point of the ray, and assigns each intersection a depth—the distance from the ray start point in the direction defined. The intersections in the reverse direction have a negative depth. The intersections are ordered from the negative depth to the positive depth. The input arguments are:

- *ApertureRadius*—Specifies the aperture value in pixels. If you give a value less than –1.0, the value is taken from the Creo configuration file option `pick_aperture_radius`. If that option is not set, the function uses the default value of 7.0.
- *Ray*—Specifies the ray. A ray is specified in terms of a start location and direction vector as a `wfcRay` object.

  The method `wfcRay::Create` creates a data object that contains information related to the ray. Use the method `wfcRay::SetPoint` to set the starting point for the ray. The method `wfcRay::GetPoint` returns the starting point of the ray.

  The method `wfcRay::SetVector` sets the direction vector for the ray. Use the method `wfcRay::GetVector` to get the direction vector.

Measurement

Method Introduced:

- **wfcWSelection::EvaluateAngle**
- **wfcWContour::Eval3DOutline**
- **wfcWSelection::EvaluateDiameter**

Use the method `wfcWSelection::EvaluateAngle` to measure the angle between two geometry items expressed as `wfcWSelection` objects. Both objects must be straight, solid edges.

Use the method `wfcWContour::Eval3DOutline` to find the 3D bounding box for the inside surface of the specified outer contour. This method takes into account the internal voids present.

Use the method `wfcWSelection::EvaluateDiameter` to get the diameter of the specified surface. Specify only revolved surfaces such as, cylinder, cone, and so on for this method.

26

Dimensions and Parameters

This chapter describes the Creo Object TOOLKIT C++ methods and classes that affect dimensions and parameters.

Overview

Dimensions and parameters in Creo Parametric have similar characteristics but also have significant differences. In Creo Object TOOLKIT C++, the similarities between dimensions and parameters are contained in the `pfcBaseParameter` interface. This interface allows access to the parameter or dimension value and to information regarding a parameter's designation and modification. The differences between parameters and dimensions are recognizable because `pfcDimension` inherits from the interface `pfcModelItem`, and can be assigned tolerances, whereas parameters are not `pfcModelItems` and cannot have tolerances.

The ParamValue Object

Both parameters and dimension objects contain an object of type `pfcParamValue`. This object contains the integer, real, string, or Boolean value of the parameter or dimension. Because of the different possible value types that can be associated with a `pfcParamValue` object there are different methods used to access each value type and some methods will not be applicable for some `pfcParamValue` objects. If you try to use an incorrect method an exception will be thrown.

Accessing a ParamValue Object

Methods Introduced:

- **pfcCreateIntParamValue**
- **pfcCreateDoubleParamValue**
- **pfcCreateStringParamValue**
- **pfcCreateBoolParamValue**
- **pfcCreateNoteParamValue**
- **pfcBaseParameter::GetValue**

The `pfcModelItem` utility class contains methods for creating each type of `pfcParamValue` object. Once you have established the value type in the object, you can change it. The method `pfcBaseParameter::GetValue` returns the `pfcParamValue` associated with a particular parameter or dimension.

A `NotepfcParamValue` is an integer value that refers to the ID of a specified note. To create a parameter of this type the identified note must already exist in the model.

Accessing the ParamValue Value

Methods Introduced:

- **pfcParamValue::Getdiscr**

- **pfcParamValue::GetIntValue**
- **pfcParamValue::SetIntValue**
- **pfcParamValue::GetDoubleValue**
- **pfcParamValue::SetDoubleValue**
- **pfcParamValue::GetStringValue**
- **pfcParamValue::SetStringValue**
- **pfcParamValue::GetBoolValue**
- **pfcParamValue::SetBoolValue**
- **pfcParamValue::GetNoteId**

The method `pfcParamValue::Getdiscr` returns a enumeration object that identifies the type of value contained in the `pfcParamValue` object. Use this information with the `Get` and `Set` methods to access the value. If you use an incorrect `Get` or `Set` method an exception of type `pfcXBadGetParamValue` will be thrown.

Parameter Objects

The following sections describe the Creo Object TOOLKIT C++ methods that access parameters. The topics are as follows:

Creating and Accessing Parameters

Methods Introduced:

- **pfcParameterOwner::CreateParam**
- **pfcParameterOwner::CreateParamWithUnits**
- **pfcParameterOwner::GetParam**
- **pfcParameterOwner::ListParams**
- **pfcParameterOwner::SelectParam**
- **pfcParameterOwner::SelectParameters**
- **pfcFamColParam::GetRefParam**
- **wfcWParameterOwner::ApplyParameterTableset**
- **wfcWParameterOwner::ExportParameterTable**

In Creo Object TOOLKIT C++, models, features, surfaces, and edges inherit from the `pfcParameterOwner` interface, because each of the objects can be assigned parameters in Creo Parametric.

The method `pfcParameterOwner::GetParam` gets a parameter given its name.

The method `pfcParameterOwner::ListParams` returns a sequence of all parameters assigned to the object.

To create a new parameter with a name and a specific value, call the method `pfcParameterOwner::CreateParam`.

To create a new parameter with a name, a specific value, and units, call the method `pfcParameterOwner::CreateParamWithUnits`.

The method `pfcParameterOwner::SelectParam` allows you to select a parameter from the Creo Parametric user interface. The top model from which the parameters are selected must be displayed in the current window.

The method `pfcParameterOwner::SelectParameters` allows you to interactively select parameters from the Creo Parametric **Parameter** dialog box based on the parameter selection options specified by the `pfcParameterSelectionOptions` object. The top model from which the parameters are selected must be displayed in the current window. Refer to the section Parameter Selection Options on page 443 for more information.

The method `pfcFamColParam::GetRefParam` returns the reference parameter from the parameter column in a family table.

The method `wfcWParameterOwner::ApplyParameterTableset` assigns the specified parameter set to the parameter owner `modelitem`. You can create or modify the parameters called by the parameter table set, that is, you can modify the entries contained in this parameter table set, change the label set for the parameter table set and alter the name of the table that owns this parameter table set. The values specified for the parameters, are set as the default values for the parameters.

Note

This method does not regenerate the model on applying the parameter set.

A parameter table simplifies storing and accessing the value sets used for manipulating dimensions and parametric information. Parameter tables maintain design intent, while allowing adjustment for different parameter values. As the global parameters are numeric or have a value of yes or no, they control the dimension values of the components and the mathematical relations between them in an assembly. A parameter table makes it easy to switch between different configuration options of an assembly. The method

wfcWParameterOwner::ExportParameterTable exports a file containing information about the parameter table in Creo Parametric to a specified format. The input parameters for this method are:

- TopMdl—Specify the top level model from which parameters are to be exported.
- Contexts—Specifies a bitmask that contains the context of parameters to list in the exported file. Specify a context only if the argument owner is set to NULL. The list of contexts is specified by the enum pfcParameterSelectionContext and the valid combinations of the context are:
  - pfcPARAMSELECT_ALLOW_SUBITEM_SELECTION used alone—Specifies that all the parameters of the sub items of the top model will be exported.
  - pfcPARAMSELECT_ALLOW_SUBITEM_SELECTION along with any another context—Specifies that only the parameters that belong to the selected context will be exported.
  - pfcPARAMSELECT_MODEL, pfcPARAMSELECT_PART or pfcPARAMSELECT_ASM—Specifies that all the model level parameters in the top model will be exported.
- ExportType—Specify the format of the exported file. The supported formats are defined by the enumerated type wfcParamTableExportType and are as follows:
  - wfcPARAMTABLE_EXPORT_TXT—Specifies that the file will be exported in a .TXT format.
  - wfcPARAMTABLE_EXPORT_CSV—Specifies that the file will be exported in a .CSV format.
- Path—Specify the full path, including the name and the extension of the file to be created during export.
- Columns—Specifies a list that contains the description about the number of columns to be included in the exported file. The columns exported will match the columns and options set by the user in the active session using **File** ▸ **Export** in the **Parameters** dialog box to export the entire parameter table in the Comma Separated Value (CSV) format. This parameter is not applicable for the text format. You can also specify the type of parameter table column to be included in the export file using the enumerated type wfcParamColumn.
- Owner—Specify the owner modelitem of the parameters to be exported. Set this parameter to NULL, if the parameters to be exported are selected by context.

Parameter Selection Options

Parameter selection options in Creo Object TOOLKIT C++ are represented by the `pfcParameterSelectionOptions` interface.

Methods Introduced:

- **pfcParameterSelectionOptions::Create**
- **pfcParameterSelectionOptions::SetAllowContextSelection**
- **pfcParameterSelectionOptions::SetContexts**
- **pfcParameterSelectionOptions::SetAllowMultipleSelections**
- **pfcParameterSelectionOptions::SetSelectButtonLabel**

The method `pfcParameterSelectionOptions::Create` creates a new instance of the `pfcParameterSelectionOptions` object that is used by the method `pfcParameterOwner::SelectParameters`.

The parameter selection options are as follows:

- `AllowContextSelection`—This boolean attribute indicates whether to allow parameter selection from multiple contexts, or from the invoking parameter owner. By default, it is false and allows selection only from the invoking parameter owner. If it is true and if specific selection contexts are not yet assigned, then you can select the parameters from any context.

  Use the method `pfcParameteSelectionOptions::SetAllowContextSelection` to modify the value of this attribute.
- `Contexts`—The permitted parameter selection contexts in the form of the `pfcParameterSelectionContexts` object. Use the method `pfcParameterSelectionOptions::SetContexts` to assign the parameter selection context. By default, you can select parameters from any context.
- The types of parameter selection contexts are as follows:
  - `pfcPARAMSELECT_MODEL`—Specifies that the top level model parameters can be selected.
  - `pfcPARAMSELECT_PART`—Specifies that any part's parameters (at any level of the top model) can be selected.
  - `pfcPARAMSELECT_ASM`—Specifies that any assembly's parameters (at any level of the top model) can be selected.
  - `pfcPARAMSELECT_FEATURE`—Specifies that any feature's parameters can be selected.
  - `pfcPARAMSELECT_EDGE`—Specifies that any edge's parameters can be selected.

- pfcPARAMSELECT_SURFACE—Specifies that any surface's parameters can be selected.
- pfcPARAMSELECT_QUILT—Specifies that any quilt's parameters can be selected.
- pfcPARAMSELECT_CURVE—Specifies that any curve's parameters can be selected.
- pfcPARAMSELECT_COMPOSITE_CURVE—Specifies that any composite curve's parameters can be selected.
- pfcPARAMSELECT_INHERITED—Specifies that any inheritance feature's parameters can be selected.
- pfcPARAMSELECT_SKELETON—Specifies that any skeleton's parameters can be selected.
- pfcPARAMSELECT_COMPONENT—Specifies that any component's parameters can be selected.

- AllowMultipleSelections—This boolean attribute indicates whether or not to allow multiple parameters to be selected from the dialog box, or only a single parameter. By default, it is true and allows selection of multiple parameters.

  Use the method pfcParameterSelectionOptions::SetAllowMultipleSelections to modify this attribute.
- SelectButtonLabel—The visible label for the select button in the dialog box.

  Use the method pfcParameterSelectionOptions::SetSelectButtonLabel to set the label. If not set, the default label in the language of the active Creo Parametric session is displayed.

Parameter Information

Methods Introduced:

- **pfcBaseParameter::GetValue**
- **pfcBaseParameter::SetValue**
- **pfcParameter::GetScaledValue**
- **pfcParameter::SetScaledValue**
- **pfcParameter::GetUnits**
- **pfcBaseParameter::GetIsDesignated**
- **pfcBaseParameter::SetIsDesignated**

- **pfcBaseParameter::GetIsModified**
- **pfcBaseParameter::ResetFromBackup**
- **pfcParameter::GetDescription**
- **pfcParameter::SetDescription**
- **pfcParameter::GetRestriction**
- **pfcParameter::GetDriverType**
- **pfcParameter::Reorder**
- **pfcParameter::Delete**
- **pfcNamedModelItem::GetName**
- **wfcWParameter::GetLockStatus**
- **wfcWParameter::SetLockStatus**
- **wfcWParameter::GetValueWithUnits**
- **wfcWParameter::SetValueWithUnits**
- **wfcParamValueWithUnits::Create**
- **wfcParamValueWithUnits::GetValue**
- **wfcParamValueWithUnits::SetValue**
- **wfcParamValueWithUnits::GetUnits**
- **wfcParamValueWithUnits::SetUnits**
- **wfcParameterData::Create**
- **wfcParameterData::GetName**
- **wfcParameterData::GetUnits**
- **wfcParameterData::GetValue**
- **wfcParameterConflict::Create**
- **wfcParameterConflict::GetConflictDescription**
- **wfcParameterConflict::GetConflictSeverity**
- **wfcParameterConflict::GetParameterName**

Parameters inherit methods from the `pfcBaseParameter`, `pfcParameter` and `pfcNamedModelItem` interfaces.

The method `pfcBaseParameter::GetValue` returns the value of the parameter or dimension.

The method `pfcBaseParameter::SetValue` assigns a particular value to a parameter or dimension.

The method `pfcParameter::GetScaledValue` returns the parameter value in the units of the parameter, instead of the units of the owner model as returned by `pfcBaseParameter::GetValue`.

The method `pfcParameter::SetScaledValue` assigns the parameter value in the units provided, instead of using the units of the owner model as assumed by `pfcBaseParameter::GetValue`.

The method `pfcParameter::GetUnits` returns the units assigned to the parameter.

You can access the designation status of the parameter using the methods `pfcBaseParameter::GetIsDesignated` and `pfcBaseParameter::SetIsDesignated`.

The methods `pfcBaseParameter::GetIsModified` and `pfcBaseParameter::ResetFromBackup` enable you to identify a modified parameter or dimension, and reset it to the last stored value. A parameter is said to be "modified" when the value has been changed but the parameter's owner has not yet been regenerated.

The method `pfcParameter::GetDescription` returns the parameter description, or null, if no description is assigned.

The method `pfcParameter::SetDescription` assigns the parameter description.

The method `pfcParameter::GetRestriction` identifies if the parameter's value is restricted to a certain range or enumeration. It returns the `pfcParameterRestriction` object. Refer to the section Parameter Restrictions on page 448 for more information.

The method `pfcParameter::GetDriverType` returns the driver type for a material parameter. The driver types are as follows:

- `pfcPARAMDRIVER_PARAM`—Specifies that the parameter value is driven by another parameter.
- `pfcPARAMDRIVER_FUNCTION`—Specifies that the parameter value is driven by a function.
- `pfcPARAMDRIVER_RELATION`—Specifies that the parameter value is driven by a relation. This is equivalent to the value obtained using `pfcGetIsRelationDriven` for a parameter object type.

The method `pfcParameter::Reorder` reorders the given parameter to come immediately after the indicated parameter in the **Parameter** dialog box and information files generated by Creo Parametric.

The method `pfcParameter::Delete` permanently removes a specified parameter.

The method `pfcNamedModelItem::GetName` accesses the name of the specified parameter.

The method wfcWParameter::GetLockStatus returns the access state of the specified parameter. Use the function wfcWParameter::SetLockStatus to set the access state for the specified parameter. The access state is defined in the enumerated data type wfcParamLockstatus. The valid values are:

- wfcPARAMLOCKSTATUS_UNLOCKED—Specifies parameters with full access. Full access parameters are user-defined parameters that can be modified from any application.
- wfcPARAMLOCKSTATUS_LIMITED—Specifies parameters with limited access. Full access parameters can be set to have limited access. Limited access parameters can be modified by user, family tables and programs. These parameters cannot be modified by relations.
- wfcPARAMLOCKSTATUS_LOCKED—Specifies parameters with locked access are parameters. The parameters can be locked either by an external application, or by the user. You can modify parameters locked by an external application only from within an external application. You cannot modify user-defined locked parameters from within an external application.

The methods wfcWParameter::GetValueWithUnits and wfcWParameter::SetValueWithUnits reads and sets the value of a parameter specified by the wfcParamValueWithUnits object. These methods also retrieve and set the units of the parameter.

Use the method wfcParamValueWithUnits::Create to create the wfcParamValueWithUnits object, which contains information about the parameter.

The methods wfcParamValueWithUnits::GetValue and wfcParamValueWithUnits::SetValue get and set the value of the parameter. Use the methods wfcParamValueWithUnits::GetUnits and wfcParamValueWithUnits::SetUnits to get and set the units of the parameter.

Use the method wfcParameterData::Create to create the wfcParameterData object, which contains information or data about the parameter. The input parameters are:

- *Name*—Specifies the name of the parameter.
- *Value*—Specifies the value of the parameter.
- *Unit*—Specifies the unit of the parameter.

Use the method wfcParameterData::GetName to retrieve the name of the parameter.

Use the method wfcParameterData::GetUnits to retrieve the units of the parameter.

Use the method wfcParameterData::GetValue to retrieve the value of the parameter.

The method `wfcParameterConflict::Create` creates a report which checks if the restricted value parameters in the model are in agreement with an external file. The input parameters are:

- *Name*— Specifies the name of the parameter in conflict.
- *Severity*—Specifies the severity of the conflict.
- *Description*—Specifies the description of the parameter in conflict.

Use the method `wfcParameterConflict::GetConflictDescription` to retrieve the description of the parameter in conflict.

Use the method `wfcParameterConflict::GetConflictSeverity` to retrieve the severity of the conflict using the enumerated type `wfcParameterConflictSeverity`.

Use the method `wfcParameterConflict::GetParameterName` to retrieve the name of the parameter in conflict.

Parameter Restrictions

Creo Parametric allows users to assign specified limitations to the value allowed for a given parameter (wherever the parameter appears in the model). You can only read the details of the permitted restrictions from Creo Object TOOLKIT C++, but not modify the permitted values or range of values. Parameter restrictions in Creo Object TOOLKIT C++ are represented by the interface `pfcParameterRestriction`.

Method Introduced:

- **pfcParameterRestriction::GetType**

The method `pfcParameterRestriction::GetType` returns the `pfcRestrictionType` object containing the types of parameter restrictions. The parameter restrictions are of the following types:

- `pfcPARAMSELECT_ENUMERATION`—Specifies that the parameter is restricted to a list of permitted values.
- `pfcPARAMSELECT_RANGE`—Specifies that the parameter is limited to a specified range of numeric values.

Enumeration Restriction

The `pfcPARAMSELECT_ENUMERATION` type of parameter restriction is represented by the interface `pfcParameterEnumeration`. It is a child of the `pfcParameterRestriction` interface.

Method Introduced:

- **pfcParameterEnumeration::GetPermittedValues**

The method pfcParameterEnumeration::GetPermittedValues returns a list of permitted parameter values allowed by this restriction in the form of a sequence of the pfcParamValue objects.

Range Restriction

The pfcPARAMSELECT_RANGE type of parameter restriction is represented by the interface pfcParameterRange. It is a child of the pfcParameterRestriction interface.

Methods Introduced:

- **pfcParameterRange::GetMaximum**
- **pfcParameterRange::GetMinimum**
- **pfcParameterLimit::GetType**
- **pfcParameterLimit::GetValue**

The method pfcParameterRange::GetMaximum returns the maximum value limit for the parameter in the form of the pfcParameterLimit object.

The method pfcParameterRange::GetMinimum returns the minimum value limit for the parameter in the form of the pfcParameterLimit object.

The method pfcParameterLimit::GetType returns the pfcParameterLimitType containing the types of parameter limits. The parameter limits are of the following types:

- pfcPARAMLIMIT_LESS_THAN—Specifies that the parameter must be less than the indicated value.
- pfcPARAMLIMIT_LESS_THAN_OR_EQUAL—Specifies that the parameter must be less than or equal to the indicated value.
- pfcPARAMLIMIT_GREATER_THAN—Specifies that the parameter must be greater than the indicated value.
- pfcPARAMLIMIT_GREATER_THAN_OR_EQUAL—Specifies that the parameter must be greater than or equal to the indicated value.

The method pfcParameterLimit::GetValue returns the boundary value of the parameter limit in the form of the pfcParamValue object.

Table Parameters

A parameter table is made up of one or more parameter table sets. Each set represents one or more parameters with their assigned values or assigned ranges. A parameter owner such as model, feature, annotation element or geometry item can only have one parameter table set to create a parameter. In Creo Object TOOLKIT C++, a parameter table set is represented by the type

wfcParamtableSet and is made up of entries, represented by wfcParamTableEntry. A single entry represents a parameter with an assigned value or range.

Methods Introduced:

- **wfcParamTableEntry::GetName**
- **wfcParamTableEntry::GetRange**
- **wfcParamTableEntry::GetValue**
- **wfcWParameter::GetTableset**
- **wfcParamTableset::GetEntries**
- **wfcParamTableset::GetLabel**
- **wfcParamTableset::GetTablePath**

Use the method wfcParamTableEntry::GetName to obtain the name of the parameter in the table set.

Use the method wfcParamTableEntry::GetRange to obtain the permitted range for the parameter in the table set. This method returns an object of the class pfcParameterRange. Use the methods pfcParameterRange::GetMaximum and pfcParameterRange::GetMinimum to get the minimum and maximum value for the parameter. For more information on these methods refer to the section Parameter Restrictions on page 448.

Use the method wfcParamTableEntry::GetValue to get the value set for a parameter in the table set.

Use the method wfcWParameter::GetTableset to obtain the parameter table set that contains the specified parameter.

Use the method wfcParamTableset::GetEntries to get the list of parameters present in the parameter table set. This method returns an array of parameter table set which contains information about all the parameters defined.

Use the method wfcParamTableset::GetLabel to get the set label parameter defined for the parameter table set. A parameter that describes an entire set of parameters and their values is called a set label parameter.

Use the method wfcParamTableset::GetTablePath to get the name of the table that owns that parameter table set. If the parameter table set has been loaded from a table file, this method returns the full path of the table. It returns the table name, if the table parameter table set is stored in the model directly.

Driven and Driving Parameters

Driven or dependent parameters are controlled by the equation you define. Driven parameters enable you to set particular values for a dimension, drive the value of one dimension based on the behavior of another dimension, and dynamically

suppress features based on changes in the part. Driving or independent parameters on the other hand are capable of controlling an activity and their value does not change often unlike driving parameters. The methods described below provide access to the item, that is, parameter or method driving model parameters. You can use Creo Parametric parameters to define the characteristics of your material properties with either driven or driving parameters.

Methods Introduced:

- **wfcWParameter::GetDrivingParam**
- **wfcWParameter::SetDrivingParam**

If the driver type defined by `pfcParameterDriverType` is set to `pfcPARAMDRIVER_PARAM`, the method `wfcWParameter::GetDrivingParam` returns the name of the driving parameter for the specified material parameter.

Use the method `wfcWParameter::SetDrivingParam` to assign the driving parameter for a material parameter. This method will set the driver type to `pfcPARAMDRIVER_PARAM`.

Dimension Objects

Dimension objects include standard Creo Parametric dimensions as well as reference dimensions. Dimension objects enable you to access dimension tolerances and enable you to set the value for the dimension. Reference dimensions allow neither of these actions.

Getting Dimensions

Dimensions and reference dimensions are Creo Parametric model items. See the section Getting ModelItem Objects on page 317 for methods that can return `pfcDimension` and `pfcRefDimension` objects.

Dimension Information

Methods Introduced:

- **pfcBaseParameter::GetValue**
- **pfcBaseParameter::SetValue**
- **pfcBaseDimension::GetDimValue**
- **pfcBaseDimension::SetDimValue**
- **pfcBaseParameter::GetIsDesignated**
- **pfcBaseParameter::SetIsDesignated**
- **pfcBaseParameter::GetIsModified**

- **pfcBaseParameter::ResetFromBackup**
- **pfcBaseParameter::GetIsRelationDriven**
- **pfcBaseDimension::GetDimType**
- **pfcBaseDimension::GetSymbol**
- **pfcBaseDimension::GetTexts**
- **pfcBaseDimension::SetTexts**
- **wfcWDimension::GetBound**
- **wfcWDimension::GetNominalValue**
- **wfcWDimension::GetSymbolModeText**
- **wfcWDimension::IsFractional**
- **wfcWDimension::IsBasic**
- **wfcWDimension::IsInspection**
- **wfcWDimension::GetOwnerFeature**
- **wfcWDimension::IsDisplayedValueRounded**
- **wfcWDimension::DisplayValueAsRounded**
- **wfcWDimension::GetDisplayedValue**
- **wfcWDimension::GetOverrideValue**
- **wfcWDimension::GetDisplayedValueType**
- **wfcWDimension::IsSignDriven**
- **wfcWDimension::IsAccessibleInModel**
- **wfcWDimension::GetSignificantDigits**
- **wfcWDimension::GetDenominator**

All the `pfcBaseParameter` methods are accessible to Dimensions as well as Parameters. See the section Parameter Objects on page 440 for brief descriptions.

Note

You cannot set the value or designation status of reference dimension objects.

The methods `pfcBaseDimension::GetDimValue` and `pfcBaseDimension::SetDimValue` access the dimension value as a double. These methods provide a shortcut for accessing the dimensions' values without using a `ParamValue` object.

The `pfcBaseParameter::GetIsRelationDriven` method identifies whether the part or assembly relations control a dimension.

The method `pfcBaseDimension::GetDimType` returns an enumeration object that identifies whether a dimension is linear, radial, angular, or diametrical.

The method `pfcBaseDimension::GetSymbol` returns the dimension or reference dimension symbol (that is, “d#” or “rd#”).

The methods `pfcBaseDimension::GetTexts` and `pfcBaseDimension::SetTexts` allow access to the text strings that precede or follow the dimension value.

The method `wfcWDimension::GetBound` returns the bound values of a dimension using the enumerated data type `wfcDimBound`. When you design a model, the actual part dimensions must be within certain predetermined limits of size. These limits of size—the upper and lower dimension boundaries—are known as dimension bounds. Refer to the section Modifying Dimensions on page 455 for more information on bound values.

The method `wfcWDimension::GetNominalValue` returns the nominal value of a dimension. The method returns the nominal value even if the dimension is set to the upper or lower bound. The nominal value is returned in degrees for an angular dimension and in the system of units for other types of dimensions.

The method `wfcWDimension::GetSymbolModeText` returns the text of the dimension in symbol mode.

The method `wfcWDimension::IsFractional` checks whether the dimension is expressed in terms of a fraction rather than a decimal.

The method `wfcWDimension::IsBasic` identifies if the specified dimension is a basic dimension.

The method `wfcWDimension::IsInspection` identifies if the specified dimension is an inspection dimension.

The method `wfcWDimension::GetOwnerFeature` returns the feature that owns the specified dimension.

Use the method `wfcWDimension::IsDisplayedValueRounded` to determine whether the specified dimension is set to display its rounded off value.

In Creo TOOLKIT, a rounded off value is a decimal value that contains only the desired number of digits after the decimal point. For example, if a dimension has the stored value `10.34132` and you want to display only two digits after the decimal point, you must round off the stored value to two decimal places. Thus, rounding off converts `10.34132` to `10.34`.

Use the method `wfcWDimension::DisplayValueAsRounded` to set the attribute of the given dimension to display either the rounded off value or the stored value.

You can use this method for all dimensions, except angular dimensions created prior to Pro/ENGINEER Wildfire 4.0, ordinate baseline dimensions, and dimensions of legacy type. For these dimensions, the method throws an exception `pfcXToolkitNotValid`.

If you choose to display the rounded off value, the method `wfcWDimension::GetDisplayedValue` retrieves the displayed rounded value of the specified dimension. Otherwise, it retrieves the stored value.

The method `wfcWDimension::GetOverrideValue` returns the override value for a dimension. The default override value is zero.

Note

The override value is available only for driven dimensions.

Use the method `wfcWDimension::GetDisplayedValueType` to obtain the type of value displayed for a dimension using the enumerated type `wfcDimValueDisplay`. The valid types are:

- `wfcDIMVALUEDISPLAY_NOMINAL`—Displays the actual value of the dimension along with the tolerance value.
- `wfcDIMVALUEDISPLAY_OVERRIDE`—Displays the override value for the dimension along with the tolerance value.
- `wfcDIMVALUEDISPLAY_HIDE`—Displays only the tolerance value for the dimension.

When you set a negative value to a dimension, it will either change the dimension to this negative value, or flip the direction around its reference and show a positive value dimension instead. Use the method `wfcWDimension::IsSignDriven` to check this. The method returns the following values:

- `true`—When the negative sign in the dimension value is used to flip the direction.
- `false`—When the negative sign is used to indicate a negative value, that is, the dimension is negative.

The configuration option `show_dim_sign` when set to `yes` allows you to display negative dimensions in the Creo Parametric user interface.

When the option is set `no`, the dimensions always show positive value. However, in this case, if you set a negative value for the dimension, the direction is flipped.

Note

Some feature types, such as, dimensions for coordinate systems and datum point offsets, always show negative or positive values, even if the option is set to `no`. These features do not depend on the configuration option.

The method `wfcWDimension::IsAccessibleInModel` identifies if a specified dimension is owned by the model. When a model owns the dimension, then by default, the dimension is accessible in the model.

The method `wfcWDimension::GetSignificantDigits` retrieves the number of decimals digits that are significant for a dimension or tolerance. If you specify the input argument *Tolerance* as `false`, the method retrieves the number of decimals digits that are significant for the dimension.

If you want to get the number of decimal places shown for the upper and lower values of the dimension tolerance, specify the input argument *Tolerance* as `true`.

The method `wfcWDimension::GetDenominator` retrieves the value of the largest possible denominator that is used to define a fractional value or tolerance. If you specify the input argument *Tolerance* as `false`, the method returns the value of the largest possible denominator used to define the fractional value.

If you want to get the value for the largest possible denominator for the upper and lower tolerance values, specify the input argument *Tolerance* as `true`. By default, this value is defined by the `config.pro` option, `dim_fraction_denominator`.

Modifying Dimensions

Methods Introduced:

- **wfcWDimension::SetBound**
- **wfcWDimension::SetAsBasic**
- **wfcWDimension::SetAsInspection**
- **wfcWDimension::SetOverrideValue**
- **wfcWDimension::SetDisplayedValueType**
- **wfcWDimension::SetElbowLength**
- **wfcWDimension::CreateSimpleBreak**
- **wfcWDimension::CreateJog**
- **wfcWDimension::EraseWitnessLine**
- **wfcWDimension::ShowWitnessLine**
- **wfcWDimension::SetSignificantDigits**
- **wfcWDimension::SetDenominator**
- **wfcWDimension::SetDimensionArrowType**
- **wfcWDimension::EnvelopeGet**

The method `wfcWDimension::SetBound` sets the bound status of the dimension using the enumerated type `wfcDimBound`. The valid values are:

- `wfcDIM_BOUND_NOMINAL`—Sets the dimension value to the nominal value. It generates geometry based on exact ideal dimensions.

- `wfcDIM_BOUND_UPPER`—Sets the dimension value to its maximum value. It generates geometry based on a nominal dimension value plus the tolerance.
- `wfcDIM_BOUND_LOWER`—Sets the dimension value to its minimum value. It generates geometry based on a nominal dimension value minus the tolerance.
- `wfcDIM_BOUND_MIDDLE`—Sets the dimension value to the nominal value plus the mean of the upper and lower tolerance values.

The methods `wfcWDimension::SetAsBasic` and `wfcWDimension::SetAsInspection` set the basic and inspection notations of the dimension respectively. These methods are applicable to both driven and driving dimensions.

Note

The basic and inspection notations of the dimension are not available when only the tolerance value for a dimension is displayed.

The method `wfcWDimension::SetOverrideValue` assigns the override value for a dimension. This value is restricted to real numbers. The default override value is zero.

Note

You can set the override value only for driven dimensions.

The method `wfcWDimension::SetDisplayedValueType` sets the type of value to be displayed for a dimension using the enumerated data type `wfcDimValueDisplay`.

The method `wfcWDimension::SetElbowLength` sets the length of the elbow for the specified dimension in a solid. The method can also be used to set the length of the elbow for a dimension in a drawing, where the dimension is created in a solid and is displayed in a drawing. To work with dimensions shown in a drawing, pass the name of the drawing in the input argument *Drw*.

The method `wfcWDimension::CreateSimpleBreak` creates a simple break on an existing dimension witness line. The input arguments are:

- *Drawing*—Specifies the drawing in which the dimension is present. You can specify a `NULL` value.
- *WitnessLineIndex*—Specifies the index of the witness line. Depending on which side of the dimension the witness line lies, specify the value as `1` or `2`. Use the methods

`wfcDimLocation::GetFirstWitnessLineLocation` or `wfcDimLocation::GetSecondWitnessLineLocation` to get the location of the witness line end points for a dimension.

Note

This argument is not applicable for ordinate, radius, and diameter dimensions.

- *BreakStart*—Specifies the start point of the break.
- *BreakEnd*—Specifies the end point of the break.

The method `wfcWDimension::CreateJog` creates a jog on an existing dimension witness line. The input arguments are:

- *Drawing*—Specifies the drawing in which the dimension is present. You can specify a `NULL` value.
- *WitnessLineIndex*—Specifies the index of the witness line. Depending on which side of the dimension the witness line lies, specify the value as `1` or `2`. Use the methods `wfcDimLocation::GetFirstWitnessLineLocation` or `wfcDimLocation::GetSecondWitnessLineLocation` to get the location of the witness line end points for a dimension.

Note

This argument is not applicable for ordinate, radius, and diameter dimensions.

- *JogPoints*—Specifies an array of points to position the jog. If the specified witness line has no jog added to it, then you must specify minimum two points that is, the start point and end point of the jog.

Note

The methods `wfcWDimension::CreateSimpleBreak` and `wfcWDimension::CreateJog` throw an exception `pfcXToolkitInvalidType` when breaks and jogs are not supported for the specified dimension type, for example, diameter dimension. The methods throw an exception `pfcXToolkitAbort` when it is not possible to create breaks or jogs for the specified dimension witness line. For example, if you add a jog that is duplicate to an existing jog on the dimension witness line.

When you create a dimension, witness lines are created based on the dimension placement location and dimension references. These witness lines do not overlap with the reference geometry.

The method `wfcWDimension::EraseWitnessLine` erases a specified witness line from the dimension. The input arguments are:

- *Drawing*—Specifies the drawing in which the dimension is displayed. To erase witness line from a solid, specify this argument as `NULL`.
- *WitnessLineIndex*—Specifies the index of the witness line. Specify the value as `1` or `2` depending on which side of the dimension the witness line lies. Use the methods `wfcDimLocation::GetFirstWitnessLineLocation` or `wfcDimLocation::GetSecondWitnessLineLocation` to get the location of the witness line end points for a dimension.

Use the method `wfcWDimension::ShowWitnessLine` to show the erased witness line for the specified dimension.

Note

The methods `wfcWDimension::EraseWitnessLine` and `wfcWDimension::ShowWitnessLine` erase and show the witness lines of dimensions and reference dimensions, respectively. These methods work with both drawings and solids.

The method `wfcWDimension::SetSignificantDigits` sets the number of decimals digits that are significant for a dimension or tolerance. If you specify the input argument *Tolerance* as `false`, the method sets the number of decimal places for a decimal dimension.

- If the number of decimal places required to display the stored value of the dimension is greater than the number of decimal places specified in the method `wfcWDimension::SetSignificantDigits` and the round displayed value attribute of the dimension is ON, the stored value is unchanged. Only the displayed number of decimal places is changed and the displayed value is updated accordingly. For example, consider a dimension with its stored value as `12.12323` and the round displayed value attribute of the dimension is set to ON. If the method `wfcWDimension::SetSignificantDigits` sets the number of decimal places to 3, the stored value of the dimension is unchanged, that is, the stored value will be `12.12323`. The displayed value of the dimension is rounded to 3 decimal places, that is, `12.123`. The round displayed value attribute is not changed.
- If the number of decimal places required to display the stored value of the dimension is greater than the number of decimal places specified in the method `wfcWDimension::SetSignificantDigits` and the round

displayed value attribute of the dimension is OFF, the number of decimal places of the dimension is modified and the stored value is rounded to the specified number of decimal places. For example, consider a dimension with its stored value as `12.12323` and the round displayed value attribute of the dimension is OFF. If the method `wfcWDimension::SetSignificantDigits` sets the dimension to 3 decimal places, then the stored value of the dimension is rounded to 3 decimal places and is modified to `12.123`. The dimension is displayed as `12.123`.

- If the number of decimal places required to display the stored value of the dimension is less than the number of decimal places specified in the method `wfcWDimension::SetSignificantDigits`, the number of decimal places is set to the specified value. The status of the round displayed value attribute is not considered, as no change or an increase to the number of decimal places will have no effect on the stored value. For example, consider a dimension with its stored value as `12.12323`. If the method `wfcWDimension::SetSignificantDigits` sets the dimension to 8 decimal places and if trailing zeros are displayed, then the dimension is displayed as `12.12323000`.

For a driven dimension:

- If the number of decimal places set by the method is greater than or equal to the number of decimal places required to display the stored value of the dimension, the decimal places value is changed and no change to the round displayed value attribute is made.
- If the number of decimal places of the dimension is less than the number required to display the stored value of the dimension, the round displayed value attribute is automatically set to ON as it is not possible to change the stored value of a driven dimension.

If you want to set the number of decimal places shown for the upper and lower values of the dimension tolerance, specify the input argument *Tolerance* as `true`. Thus, the decimals of the dimension tolerance can be set independent of the number of dimension decimals. By default, the number of decimal places for tolerance values is calculated based upon the “linear_tol” settings of the model.

Note

Specify a non-negative number as input for the argument *Digits* in the method `wfcWDimension::SetSignificantDigits`. It should be such that when you apply either the upper or lower values of tolerance to the given dimension, the total number of digits before and after the decimal point in the resulting values must not exceed `13`.

The method `wfcWDimension::SetDenominator` sets the denominator for fractional dimensions and tolerances. If you specify the input argument *Tolerance* as `false`, the method sets the denominator for fractional dimensions. When you call the method `wfcWDimension::SetDenominator`:

- The stored value remains unchanged if,
  - it can be expressed as an exact fraction with the given denominator, regardless of whether the round-off attribute is set or not.
  - the stored value cannot be expressed as an exact fraction, but the round-off attribute is set. In this case, the fraction is the approximate representation of the stored value.
- The stored value changes to the nearest fraction and triggers a regeneration of the model, if it cannot be expressed as an exact fraction with the given denominator and the round-off attribute is not set.

If you want to set the value for the largest possible denominator for the upper and lower tolerance values, specify the input argument *Tolerance* as `true`. By default, this value is defined by the `config.pro` option, `dim_fraction_denominator`.

Use the method `wfcWDimension::SetDimensionArrowType` to set the style for the arrow head of a leader for a specified dimension. The input arguments are:

- *ArrowIndex*—Specifies the index of the witness line. Depending on which side of the dimension the witness line lies, specify the value as `1` or `2`. For diameter dimensions, this value determines which of the two arrows to change. For other dimensions, the value of `1` indicates the arrow on the first witness line, and the value of `2` indicates the arrow on the second witness line. For ordinate and radius dimensions, this value is ignored. Use the method `wfcDimLocation::GetFirstWitnessLineLocation` or `wfcDimLocation::GetSecondWitnessLineLocation` to get the location of the witness line end points for a dimension.
- *ArrowType*—Specifies the type of arrow head using the enumerated data type `wfcLeaderArrowType`.
- *Drawing*—Optional argument. Specifies the name of the drawing. For dimensions created in drawing mode and owned by a solid, which can be displayed only in the context of that drawing, specify the name of the drawing in the input argument drawing.

The function `wfcWDimension::GetEnvelope` returns the envelope of a line in the specified dimension. While retrieving coordinates of the dimension in a specified solid, if the dimension is displayed in the solid as well as in the drawing, the drawing must not be active. The input arguments follow:

- *drawing*—The value for this input argument must be passed only if the solid dimension is shown in the drawing. Else, pass it as `NULL`.
- *line*—The line number of the dimension.

Cleaning Up Dimensions

You can clean up the placement of dimensions in a drawing to meet the industry standards, and enable easier reading of your model detailing. You can adjust the location and display of dimensions by setting controls on the placement of a dimension. You can also set the cosmetic attributes, like flip the direction of arrow when the arrows do not fit between the witness lines and center the dimension text between two witness lines.

Methods Introduced:

- **wfcWDrawing::CleanupDimensions**

Use the method `wfcWDrawing::CleanupDimensions` to clean up the dimensions in a drawing. The input argument is:

- *View*—Specifies the view in which the dimensions must be cleaned as a `pfcView2D` object. If you pass the value as `NULL`, the dimensions are cleaned for all the views in the specified drawing.

The dimensions are cleaned using the default values set in the **Clean Dimensions** dialog box in Creo Parametric user interface.

Dimension Tolerances

Methods Introduced:

- **pfcDimension::GetTolerance**
- **pfcDimension::SetTolerance**
- **pfcDimTolPlusMinus::Create**
- **pfcDimTolSymmetric::Create**
- **pfcDimTolLimits::Create**
- **pfcDimTolSymSuperscript::Create**
- **pfcDimTolISODIN::Create**
- **wfcWSolid::GetTolerance**
- **wfcWSolid::SetTolerance**
- **wfcWSession::GetDefaultTolerance**
- **wfcWSolid::GetModelClass**
- **wfcWSolid::SetModelClass**
- **wfcWSolid::LoadToleranceClass**
- **wfcWDimension::IsToleranceDisplayed**
- **wfcWDimension::GetDisplayedUpperLimitTolerance**
- **wfcWDimension::GetDisplayedLowerLimitTolerance**

Only true dimension objects can have geometric tolerances.

The methods `pfcDimension::GetTolerance` and `pfcDimension::SetTolerance` enable you to access the dimension tolerance. The object types for the dimension tolerance are:

- `pfcDimTolLimits`—Displays dimension tolerances as upper and lower limits.

> **Note**
>
> This format is not available when only the tolerance value for a dimension is displayed.

- `pfcDimTolPlusMinus`—Displays dimensions as nominal with plus-minus tolerances. The positive and negative values are independent.
- `pfcDimTolSymmetric`—Displays dimensions as nominal with a single value for both the positive and the negative tolerance.
- `pfcDimTolSymSuperscript`—Displays dimensions as nominal with a single value for positive and negative tolerance. The text of the tolerance is displayed in a superscript format with respect to the dimension text.
- `pfcDimTolISODIN`—Displays the tolerance table type, table column, and table name, if the dimension tolerance is set to a hole or shaft table (DIN/ISO standard).

A `null` value is similar to the nominal option in Creo Parametric.

To determine whether a given tolerance is plus/minus, symmetric, limits, or superscript use the method `pfcDimTolerance::GetType`.

The method `wfcWSolid::GetTolerance` returns the tolerance value for the specified solid. The input arguments are:

- *Type*—Specifies the type of tolerance as linear or angular using the enumerated type `wfcToleranceType`.
- *Decimals*—Specifies the number of decimal places to identify the tolerance value.

Use the method `wfcWSolid::SetTolerance` to set the geometric tolerance for the solid. Specify the tolerance type, number of decimal places for the tolerance value, and the tolerance value as the input arguments.

The method `wfcWSession::GetDefaultTolerance` returns the default value for the specified linear or angular tolerance value. The default value is set in the Creo Parametric configuration files. Specify the tolerance type and number of decimal places to identify the tolerance value as the input arguments.

All the user specified information for a tolerance is saved in a tolerance table for ISO and DIN standards. You can retrieve and set the data for tolerance tables using Creo Object TOOLKIT C++ methods. A model with ISO or DIN standard

has an extra attribute called the tolerance class which determines the general coarseness of the model. The method `wfcWSolid::GetModelClass` returns the type of coarseness in a model using the enumerated type `wfcModelClass`. The valid values are:

- `wfcMODELCLASS_NONE`
- `wfcMODELCLASS_FINE`
- `wfcMODELCLASS_MEDIUM`
- `wfcMODELCLASS_COARSE`
- `wfcMODELCLASS_VERY_COARSE`

Use the method `wfcWSolid::SetModelClass` to set the tolerance class for a solid.

The method `wfcWSolid::LoadToleranceClass` loads the hole or shaft tolerance table for a model with ISO or DIN standard in the current session. Pass the tolerance table name `ToleranceClassName` as the input argument.

The method `wfcWDimension::IsToleranceDisplayed` checks whether the tolerances of the specified dimension are currently displayed. Refer to the Creo Parametric Detailed Drawings Help for more information.

If the round off attribute for the given dimension is set, the methods `wfcWDimension::GetDisplayedUpperLimitTolerance` and `wfcWDimension::GetDisplayedLowerLimitTolerance` retrieve the displayed rounded values of the upper and lower limits of the specified dimension. Otherwise, it retrieves the stored values of the tolerances as done by the method `wfcWSolid::GetTolerance`. For example, consider a dimension that is set to round off to two decimal places and has the upper and lower tolerances `0.123456`. By default, the tolerance values displayed are also rounded off to two decimal places. In this case, the methods `wfcWDimension::GetDisplayedUpperLimitTolerance` and `wfcWDimension::GetDisplayedLowerLimitTolerance` retrieve the upper and lower values as `0.12`.

Dimension Prefix and Suffix

Methods Introduced:

- **wfcWDimension::GetPrefix**
- **wfcWDimension::SetPrefix**
- **wfcWDimension::GetSuffix**
- **wfcWDimension::SetSuffix**

The method `wfcWDimension::GetPrefix` retrieves the prefix assigned to the specified dimension.

Use the method `wfcWDimension::SetPrefix` to set the specified prefix for a dimension.

The method `wfcWDimension::GetSuffix` retrieves the suffix assigned to the specified dimension.

Use the method `wfcWDimension::SetSuffix` to set the specified suffix for a dimension.

Dimension Location

The methods described in this section extract the dimension location and geometry in 3D space for solid model dimensions.

Dimension Entity Location

The following methods extract the locations of geometric endpoints for the dimension. You can calculate the dimension location plane, witness line, and dimension orientation vectors from these points. The location of the points is specified in the same coordinate system as the solid model.

Methods Introduced:

- **wfcDimLocation::GetNormal**
- **wfcDimLocation::GetCenterLeaderInformation**
- **wfcCenterLeaderInformation::GetCenterLeaderType**
- **wfcCenterLeaderInformation::GetElbowLength**
- **wfcCenterLeaderInformation::GetElbowDirection**
- **wfcCenterLeaderInformation::GetLeaderArrowType**
- **wfcDimLocation::GetFirstZExtensionLineLocation**
- **wfcDimLocation::GetSecondZExtensionLineLocation**
- **wfcDimLocation::GetFirstArrowheadLocation**
- **wfcDimLocation::GetSecondArrowheadLocation**
- **wfcDimLocation::GetElbowLength**
- **wfcDimLocation::GetFirstWitnessLineLocation**
- **wfcDimLocation::GetSecondWitnessLineLocation**
- **wfcDimLocation::GetLocation**
- **wfcDimLocation::HasElbow**

The method `wfcDimLocation::GetNormal` returns the vector normal to the dimensioning plane for a radial or diameter dimension. This normal vector should correspond to the axis normal to the arc being measured by the radial or diameter dimension.

The method `wfcDimLocation::GetCenterLeaderInformation` obtains the information about the center leader as a `wfcCenterLeaderInformation` object. The type of center leader is determined by the orientation of the dimension text.

Use the method `wfcCenterLeaderInformation::GetCenterLeaderType` to get the type of center leader. The valid values are defined in the enumerated data type `wfcDimCenterLeaderType`:

- `wfcDIM_CLEADER_CENTERED_ELBOW`—Specifies that the dimension text is placed next to and centered about the elbow of the center leader.
- `wfcDIM_CLEADER_ABOVE_ELBOW`—Specifies that the dimension text is placed next to and above the elbow of the center leader.
- `wfcDIM_CLEADER_ABOVE_EXT_ELBOW`—Specifies that the dimension text is placed above the extended elbow of the center leader.
- `wfcDIM_PARALLEL_ABOVE`—Specifies that the dimension text is placed parallel to and above the center leader.
- `wfcDIM_PARALLEL_BELOW`—Specifies that the dimension text is placed parallel to and below the center leader.

The method `wfcCenterLeaderInformation::GetElbowLength` and `wfcCenterLeaderInformation::GetElbowDirection` returns the length and direction of the elbow used by the center leader and the leader end symbol.

The method `wfcCenterLeaderInformation::GetLeaderArrowType` returns the type of arrow for the leader.

Note

A center leader type is available only for linear and diameter dimensions.

The methods `wfcDimLocation::GetFirstZExtensionLineLocation` and `wfcDimLocation::GetSecondZExtensionLineLocation` obtains the endpoints of the first and second Z-extension lines created for a specified dimension. Z-extension lines are automatically created whenever the dimension's attachment does not intersect its reference in the Z-Direction. The Z-extension line is attached at the edge of the surface at the closest distance from the dimension witness line.

The methods `wfcDimLocation::GetFirstArrowheadLocation` and `wfcDimLocation::GetSecondArrowheadLocation` returns the location of the first and second arrow heads for a dimension.

The method `wfcDimLocation::GetElbowLength` returns the length of the elbow for a dimension.

The methods `wfcDimLocation::GetFirstWitnessLineLocation` and `wfcDimLocation::GetSecondWitnessLineLocation` gets the location of the first and second witness line end points for a dimension.

The method `wfcDimLocation::GetLocation` returns the location of the elements that make up a solid dimension or reference dimension.

The method `wfcDimLocation::HasElbow` specifies if a dimension has an elbow. The method returns the following values:

- `True`—If the dimension has an elbow.
- `False`—If the dimension does not have an elbow.

Dimension Orientation

Methods Introduced:

- **wfcWDimension::SetAnnotationPlane**
- **wfcWDimension::GetAnnotationPlane**

The method `wfcWDimension::SetAnnotationPlane` assigns an annotation plane as the orientation of a specified dimension stored in an annotation element.

The method `wfcWDimension::GetAnnotationPlane` obtains the orientation of a specified dimension stored in an annotation element.

Driving Dimension Annotation Elements

You can convert driving dimensions created by features into annotation elements and place them on annotation planes. However, you can create the driving dimension annotation elements only in the features that own the dimensions. These annotation elements cannot have any user defined or system references.

Methods Introduced:

- **wfcWDimension::CreateAnnotationElement**
- **wfcWDimension::DeleteAnnotationElement**

The method `wfcWDimension::CreateAnnotationElement` creates an annotation element for a specified driving dimension based on the specified annotation orientation.

The method `wfcWDimension::DeleteAnnotationElement` removes the annotation element containing the driving dimension. It deletes all the parameters and relations associated with the annotation element.

Accessing Reference and Driven Dimensions

The methods described in this section provide additional access to reference and driven dimension annotations.

Many methods listed in the previous sections that are applicable for driving dimensions are also applicable for reference and driven dimensions.

Methods Introduced:

- **wfcWDimension::CanRegenerate**
- **wfcWDimension::Delete**
- **wfcWDimension::IsDriving**
- **wfcWDimension::GetDimensionAttachPoints**
- **wfcWDimension::GetDimensionSenses**
- **wfcWDimension::GetOrientationHint**
- **wfcWDimension::SetDimensionAttachPoints**

The method `wfcWDimension::CanRegenerate` identifies if a driven dimension can be regenerated.

The method `wfcWDimension::Delete` deletes the driven or reference dimension. Dimensions stored in annotation elements should be deleted using `wfcWSelection::DeleteAnnotationElement`.

The method `wfcWDimension::IsDriving` determines if a dimension is driving geometry or is driven by it. If a dimension drives geometry, its value can be modified and the model regenerated with the given change. If a dimension is driven by geometry, its value is fixed but it can be deleted and redefined as necessary. A driven dimension may also be included in an annotation element.

The method `wfcWDimension::GetDimensionAttachPoints` gets the entities to which a dimension is attached. This method supports dimensions that are created with intersection type of reference.

The method `wfcWDimension::GetDimensionSenses` gets information on how dimensions attach to the entity, that is, to what part of the entity and in what direction the dimension runs. The method returns a `pfcDimSenses` object for the driven or reference dimension. This method supports dimensions that are created with intersection type of reference. Refer to the section Creating Drawing Dimensions on page 140 for more information.

The method `wfcWDimension::GetOrientationHint` gets the orientation of the driven or reference dimensions in cases where this cannot be deduced from the attachments themselves. This method supports dimensions that are created with intersection type of reference. The orientation of the dimension is given by the enumerated type `pfcDimOrientationHint`. The valid values are:

- `pfcORIENTATION_HINT_HORIZONTAL`—Specifies a horizontal dimension.

- `pfcORIENTATION_HINT_VERTICAL`—Specifies a vertical dimension.
- `pfcORIENTATION_HINT_SLANTED`—Specifies the shortest distance between two attachment points (available only when the dimension is attached to points).
- `pfcORIENTATION_HINT_ELLIPSE_RADIUS1`—Specifies the start radius for a dimension on an ellipse.
- `pfcORIENTATION_HINT_ELLIPSE_RADIUS2`—Specifies the end radius for a dimension on an ellipse.
- `pfcORIENTATION_HINT_ARC_ANGLE`—Specifies the angle of the arc for a dimension of an arc.
- `pfcORIENTATION_HINT_ARC_LENGTH`—Specifies the length of the arc for a dimension of an arc.
- `pfcORIENTATION_HINT_LINE_TO_TANGENT_CURVE_ANGLE`—Specifies the value to dimension the angle between the line and the tangent at a curve point (the point on the curve must be an endpoint).
- `pfcORIENTATION_HINT_RADIAL_DIFF`—Specifies the linear dimension of the radial distance between two concentric arcs or circles.

The method `wfcWDimension::SetDimensionAttachPoints` sets the geometric references and parameters of the driven or reference dimension. This method supports dimensions that are created with intersection type of reference. The input arguments are:

- *DimAttachments*—Specifies the points on the model where you want to attach the dimension.
- *DimSenses*—Specifies how the dimension attaches to each attachment point of the model, that is, to what part of the entity.
- *OrientHint*—Specifies the orientation of the dimension and has one of the values given by the enumerated type `pfcDimOrientationHint`.
- *AnnotPlane*—Specifies the annotation plane for the dimensions.

45 Degree Chamfer Dimensions

You can create 45-degree chamfer dimensions by referencing one of the following items:

- Edges, including solid or surface edges, silhouette edges, curves, and sketches.
- Surfaces
- Revolve surfaces

The methods described in this section provide access to the display style of 45-degree chamfer dimensions in a solid. These methods can also be used to access the display style of the chamfer dimension in a drawing, where the dimension is

created in a solid and is displayed in a drawing. To work with dimensions shown in a drawing, pass the name of the drawing in the input argument *drawing* in the methods.

Note

The default display of a 45-degree chamfer dimension depends upon the setting of the `config.pro` option, `default_chamfer_text`.

Methods Introduced:

- **wfcWDimension::GetChamferLeaderStyle**
- **wfcWDimension::SetChamferLeaderStyle**
- **wfcWDimension::GetConfiguration**
- **wfcWDimension::SetConfiguration**
- **wfcWDimension::GetChamferStyle**
- **wfcWDimension::SetChamferStyle**

The methods `wfcWDimension::GetChamferLeaderStyle` and `wfcWDimension::SetChamferLeaderStyle` retrieve and set the style of the leader for the specified 45-degree chamfer dimension using the enumerated type `wfcDimChamferLeaderStyle`. The valid values are as follows:

- `wfcDIM_CHMFR_LEADER_STYLE_NORMAL`—Defines the leader of the chamfer dimension normal to the chamfer edge (ASME, ANSI, JIS, ISO Standard).
- `wfcDIM_CHMFR_LEADER_STYLE_DEFAULT`—Defines that the chamfer dimension leader style should be displayed using the default value.

The methods `wfcWDimension::GetConfiguration` and `wfcWDimension::SetConfiguration` retrieve and set the dimension configuration for chamfer dimensions using the enumerated type `wfcDimLeaderConfig`. The dimension configuration defines the style in which the dimension must be displayed. The valid values are as follows:

- `wfcDIMCONFIG_LEADER`—Creates the dimension with a leader.
- `wfcDIMCONFIG_LINEAR`—Creates a linear dimension.
- `wfcDIMCONFIG_CENTER_LEADER`—Creates the dimension with the leader note attached to the center of the dimension leader line.

The methods `wfcWDimension::GetChamferStyle` and `wfcWDimension::SetChamferStyle` retrieve and set the dimension scheme for the specified 45-degree chamfer dimension using the enumerated type `wfcDimChamferStyle`. The valid values are as follows:

- `wfcDIM_CHMFRSTYLE_CD`—Specifies that the chamfer dimension text should be displayed in the C(Dimension value) format (JIS/GB Standard).
- `wfcDIM_CHMFRSTYLE_D_X_45`—Specifies that the chamfer dimension text should be displayed in the (Dimension value) X 45 format (ISO/DIN Standards).
- `wfcDIM_CHMFRSTYLE_DEFAULT`—Specifies that the chamfer dimension text should be displayed using the default value set in the drawing detail option `default_chamfer_text`.
- `wfcDIM_CHMFRSTYLE_45_X_D`—Specifies that the chamfer dimension text should be displayed in the 45 X (Dimension value) format (ASME/ANSI Standards).

27

Relations

This chapter describes how to access relations on all models and model items in Creo application using the methods provided in Creo Object TOOLKIT C++.

Accessing Relations

In Creo Object TOOLKIT C++, the set of relations on any model or model item is represented by the `pfcRelationOwner` interface. Models, features, surfaces, and edges inherit from this interface, because each object can be assigned relations in Creo application.

Methods Introduced:

- **pfcRelationOwner::RegenerateRelations**
- **pfcRelationOwner::DeleteRelations**
- **pfcRelationOwner::GetRelations**
- **pfcRelationOwner::SetRelations**
- **pfcRelationOwner::EvaluateExpression**
- **wfcWRelationOwner::EvaluateExpressionWithUnits**
- **wfcWRelationOwner::UseUnits**
- **wfcWRelationOwner::UnitsUsed**

The method `pfcRelationOwner::RegenerateRelations` regenerates the relations assigned to the owner item. It also determines whether the specified relation set is valid.

The method `pfcRelationOwner::DeleteRelations` deletes all the relations assigned to the owner item.

The method `pfcRelationOwner::GetRelations` returns the list of initial relations assigned to the owner item as a sequence of strings.

The method `pfcRelationOwner::SetRelations` assigns the sequence of strings as the new relations to the owner item.

The method `pfcRelationOwner::EvaluateExpression` evaluates the given relations-based expression, and returns the resulting value in the form of the `pfcParamValue` object. Refer to the section The ParamValue Object on page 439 in the chapter Dimensions and Parameters on page 438 for more information on this object.

The method `wfcWRelationOwner::EvaluateExpression` has been deprecated. Use the method `wfcWRelationOwner::EvaluateExpressionWithUnits` instead.

Use the method `wfcWRelationOwner::EvaluateExpressionWithUnits` if you want the units of the relation to be considered while evaluating the expression. Specify the input argument `Consider_units` as true to consider the units. In this case, the result of the relation is returned along with its unit as

wfcParamValueWithUnits object. Refer to the section The ParamValue Object on page 439 in the chapter Dimensions and Parameters on page 438 for more information on this object.

The method wfcWRelationOwner::UseUnits specifies if units must be considered while solving the specified relation. Use the method wfcWRelationOwner::UnitsUsed to check if units will be considered while solving the relation.

Accessing Post Regeneration Relations

Method Introduced:

- **pfcModel::GetPostRegenerationRelations**
- **pfcModel::RegeneratePostRegenerationRelations**
- **pfcModel::DeletePostRegenerationRelations**

The method pfcModel::GetPostRegenerationRelations lists the post-regeneration relations assigned to the model. It can be NULL, if not set.

Note

To work with post-regeneration relations, use the post-regeneration relations attribute in the methods pfcRelationOwner::SetRelations, pfcRelationOwner::RegenerateRelations and pfcRelationOwner::DeleteRelations.

You can regenerate the relation sets post-regeneration in a model using the method pfcModel::RegeneratePostRegenerationRelations.

To delete all the post-regeneration relations in the specified model, call the method pfcModel::DeletePostRegenerationRelations.

Adding a Customized Function to the Relations Dialog Box

Methods Introduced:

- **pfcBaseSession::RegisterRelationFunction**

The method pfcBaseSession::RegisterRelationFunction registers a custom function that is included in the function list of the **Relations** dialog box in Creo Parametric. You can add the custom function to relations that are added to models, features, or other relation owners. The registration method takes the following input arguments:

- *Name*—The name of the custom function.
- `RelationFunctionOptions`—This object contains the options that determine the behavior of the custom relation function. Refer to the section Relation Function Options on page 474 for more information.
- `RelationFunctionListener`—This object contains the action listener methods for the implementation of the custom function. Refer to the section Relation Function Listeners on page 475 for more information.

Note

Creo Object TOOLKIT C++ relation functions are valid only when the custom function has been registered by the application. If the application is not running or not present, models that contain user-defined relations cannot evaluate these relations. In this situation, the relations are marked as errors. However, these errors can be commented until needed at a later time when the relations functions are reactivated in a Creo Parametric session.

Relation Function Options

Methods Introduced:

- **pfcRelationFunctionOptions::Create**
- **pfcRelationFunctionOptions::SetArgumentTypes**
- **pfcRelationFunctionArgument::Create**
- **pfcRelationFunctionArgument::SetType**
- **pfcRelationFunctionArgument::SetIsOptional**
- **pfcRelationFunctionOptions::SetEnableTypeChecking**
- **pfcRelationFunctionOptions::SetEnableArgumentCheckMethod**
- **pfcRelationFunctionOptions::SetEnableExpressionEvaluationMethod**
- **pfcRelationFunctionOptions::SetEnableValueAssignmentMethod**

Use the method `pfcRelationFunctionOptions::Create` to create the `pfcRelationFunctionOptions` object containing the options to enable or disable various relation function related features. Use the methods listed above to access and modify the options. These options are as follows:

- *ArgumentTypes*—The types of arguments in the form of the `pfcRelationFunctionArgument` object. By default, this parameter is null, indicating that no arguments are permitted.

Use the method `pfcRelationFunctionArgument::Create` to create the `pfcRelationFunctionArgument` object containing the attributes of the arguments passed to the custom relation function.

These attributes are as follows:

- *Type*—The type of argument value such as double, integer, and so on in the form of the `pfcParamValueType` object.
- *IsOptional*—This boolean attribute specifies whether the argument is optional, indicating that it can be skipped when a call to the custom relation function is made. The optional arguments must fall at the end of the argument list. By default, this attribute is false.

- *EnableTypeChecking*—This boolean attribute determines whether or not to check the argument types internally. By default, it is false. If this attribute is set to false, Creo does not need to know the contents of the arguments array. The custom function must handle all user errors in such a situation.
- *EnableArgumentCheckMethod*—This boolean attribute determines whether or not to enable the arguments check listener function. By default, it is false.
- *EnableExpressionEvaluationMethod*—This boolean attribute determines whether or not to enable the evaluate listener function. By default, it is true.
- *EnableValueAssignmentMethod*—This boolean attribute determines whether or not to enable the value assignment listener function. By default, it is false.

Relation Function Listeners

The interface `pfcRelationFunctionListener` provides the method signatures to implement a custom relation function.

Methods Introduced:

- **pfcRelationFunctionListener::CheckArguments**
- **pfcRelationFunctionListener::AssignValue**
- **pfcRelationFunctionListener::EvaluateFunction**

The method `pfcRelationFunctionListener::CheckArguments` checks the validity of the arguments passed to the custom function. This listener method takes the following input arguments:

- The owner of the relation being evaluated
- The custom function name
- A sequence of arguments passed to the custom function

If the implementation of this method determines that the arguments are not valid for the custom function, then the listener method returns false. Otherwise, it returns true.

The method `pfcRelationFunctionListener::EvaluateFunction` evaluates a custom relation function invoked on the right hand side of a relation. This listener method takes the following input arguments:

- The owner of the relation being evaluated
- The custom function name
- A sequence of arguments passed to the custom function

You must return the computed result of the custom relation function.

The method `pfcRelationFunctionListener::AssignValue` evaluates a custom relation function invoked on the left hand side of a relation. It allows you to initialize properties to be stored and used by your application. This listener method takes the following input arguments:

- The owner of the relation being evaluated
- The custom function name
- A sequence of arguments passed to the custom function
- The value obtained by Creo from evaluating the right hand side of the relation

28

Assemblies and Components

This chapter describes the Creo Object TOOLKIT C++ methods that access the methods of a Creo assembly. You must be familiar with the following before you read this section:

- The Selection Object
- Coordinate Systems
- The Geometry section

Structure of Assemblies and Assembly Objects

The object `pfcAssembly` is an instance of `pfcSolid`. The `pfcAssembly` object can therefore be used as input to any of the `pfcSolid` and `pfcModel` methods applicable to assemblies. However assemblies do not contain solid geometry items. The only geometry in the assembly is datums (points, planes, axes, coordinate systems, curves, and surfaces). Therefore solid assembly features such as holes and slots will not contain active surfaces or edges in the assembly model.

The solid geometry of an assembly is contained in its components. A component is a feature of type `pfcComponentFeat`, which is a reference to a part or another assembly, and a set of parametric constraints for determining its geometrical location within the parent assembly.

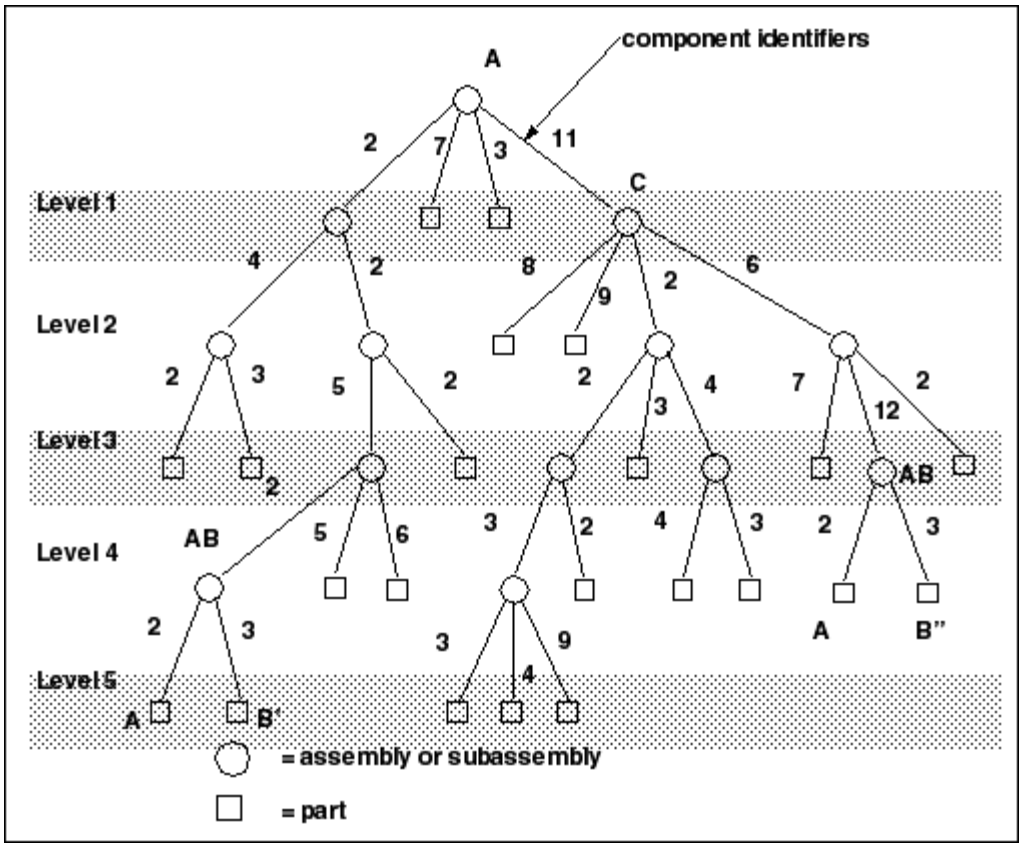
Assembly features that are solid, such as holes and slots, and therefore affect the solid geometry of parts in the assembly hierarchy, do not themselves contain the geometry items that describe those modifications. These items are always contained in the parts whose geometry is modified, within local features created for that purpose.

The important Creo Object TOOLKIT C++ methods for assemblies are those that operate on the components of an assembly. The object `pfcComponentFeat`, which is an instance of `pfcFeature` is defined for that purpose. Each assembly component is treated as a variety of feature, and the integer identifier of the component is also the feature identifier.

An assembly can contain a hierarchy of assemblies and parts at many levels, in which some assemblies and parts may appear more than once. To identify the role of any database item in the context of the root assembly, it is not sufficient to have the integer identifier of the item and the handle to its owning part or assembly, as would be provided by its `pfcFeature` description.

It is also necessary to give the full path of the assembly-component references down from the root assembly to the part or assembly that owns the database item. This is the purpose of the object `pfcComponentPath`, which is used as the input to Creo Object TOOLKIT C++ assembly methods.

The following figure shows an assembly hierarchy with two examples of the contents of a `pfcComponentPath` object.



In the assembly shown in the figure, sub-assembly C is component identifier 11 within assembly A, Part B is component identifier 3 within assembly AB, and so on. The sub-assembly AB occurs twice. To refer to the two occurrences of part B, use the following:

```
(?)Component B'                          Component B"
ComponentIds->get(0) will return 2   ComponentIds->get(1) will
return 11
ComponentIds->get(1) will return 2   ComponentIds->get(2) will
return 6
ComponentIds->get(2) will return 5   ComponentIds->get(3) will
return 12
ComponentIds->get(3) will return 2   ComponentIds->get(4) will
return 3
ComponentIds->get(4) will return 3
```

The object `pfcComponentPath` is one of the main portions of the `pfcSelection` object.

Assembly Components

Methods Introduced:

- **pfcComponentFeat::GetIsBulkitem**

- **pfcComponentFeat::GetIsSubstitute**
- **pfcComponentFeat::GetCompType**
- **pfcComponentFeat::SetCompType**
- **pfcComponentFeat::GetModelDescr**
- **pfcComponentFeat::GetIsPlaced**
- **pfcComponentFeat::SetIsPlaced**
- **pfcComponentFeat::GetIsPackaged**
- **pfcComponentFeat::GetIsUnderconstrained**
- **pfcComponentFeat::GetIsFrozen**
- **pfcComponentFeat::GetPosition**
- **pfcComponentFeat::CopyTemplateContents**
- **pfcComponentFeat::CreateReplaceOp**
- **wfcWComponentFeat::MakeUniqueSubAssembly**
- **wfcWComponentFeat::RemoveUniqueSubAssembly**
- **wfcWComponentFeat::IsUnplaced**

The method `pfcComponentFeat::GetIsBulkitem` identifies whether an assembly component is a bulk item. A bulk item is a non-geometric assembly feature that should appear in an assembly bill of materials.

The method `pfcComponentFeat::GetIsSubstitute` returns a true value if the component is substituted, else it returns a false. When you substitute a component in a simplified representation, you temporarily exclude the substituted component and superimpose the substituting component in its place.

The method `pfcComponentFeat::GetCompType` returns the type of the assembly component.

The method `pfcComponentFeat::SetCompType` enables you to set the type of the assembly component. The component type identifies the purpose of the component in a manufacturing assembly.

The method `pfcComponentFeat::GetModelDescr` returns the model descriptor of the component part or sub-assembly.

Note

The method `pfcComponentFeat::GetModelDescr` throws an exception `pfcXtoolkitCantOpen` if called on an assembly component whose immediate generic is not in session. Handle this exception and typecast the assembly component as `pfcSolid`, which in turn can be typecast as `pfcFamilyMember`, and use the method `pfcFamilyMember::GetImmediateGenericInfo` to get the model descriptor of the immediate generic model.

If you wish to switch to the pre-Wildfire 4.0 mode, set the configuration option `retrieve_instance_dependencies` to `instance_and_generic_deps`.

The method `pfcComponentFeat::GetIsPlaced` determines whether the component is placed.

The method `pfcComponentFeat::SetIsPlaced` forces the component to be considered placed. The value of this parameter is important in assembly Bill of Materials.

Note

Once a component is constrained or packaged, it cannot be made unplaced again.

A component of an assembly that is either partially constrained or unconstrained is known as a packaged component. Use the method `pfcComponentFeat::GetIsPackaged` to determine if the specified component is packaged.

The method `pfcComponentFeat::GetIsUnderconstrained` determines if the specified component is underconstrained, that is, it possesses some constraints but is not fully constrained.

The method `pfcComponentFeat::GetIsFrozen` determines if the specified component is frozen. The frozen component behaves similar to the packaged component and does not follow the constraints that you specify.

The method `pfcComponentFeat::GetPosition` retrieves the component's initial position before constraints and movements have been applied. If the component is packaged this position is the same as the constraint's actual position. This method modifies the assembly component data but does not regenerate the assembly component. To regenerate the component, use the method `pfcComponentFeat::Regenerate`.

The method `pfcComponentFeat::CopyTemplateContents` copies the template model into the model of the specified component.

The method `pfcComponentFeat::CreateReplaceOp` creates a replacement operation used to swap a component automatically with a related component. The replacement operation can be used as an argument to `pfcSolid::ExecuteFeatureOps`.

The method `wfcWComponentFeat::MakeUniqueSubAssembly` creates a unique instance of the sub-assembly by specifying the path to the sub-assembly. Use the method `wfcWComponentFeat::RemoveUniqueSubAssembly` to remove the instance of the sub-assembly.

Use the method `wfcWComponentFeat::IsUnplaced` checks if the specified component is unplaced. Unplaced components belong to an assembly without being assembled or packaged. If the method returns true, the component is unplaced.

Regenerating an Assembly Component

Method Introduced:

- **pfcComponentFeat::Regenerate**

The method `pfcComponentFeat::Regenerate` regenerates an assembly component. The method regenerates the assembly component just as in an interactive Creo session.

Creating a Component Path

Methods Introduced

- **pfcCreateComponentPath**

The method `pfcCreateComponentPath` returns a component path object, given the Assembly model and the integer id path to the desired component.

Component Path Information

Methods Introduced:

- **pfcComponentPath::GetRoot**
- **pfcComponentPath::SetRoot**
- **pfcComponentPath::GetComponentIds**
- **pfcComponentPath::SetComponentIds**
- **pfcComponentPath::GetLeaf**
- **pfcComponentPath::GetTransform**

- **pfcComponentPath::SetTransform**
- **pfcComponentPath::GetIsVisible**
- **wfcSubstituteComponent::GetSubCompPath**
- **wfcSubstituteComponent::GetSubCompFeat**
- **wfcWComponentPath::GetSubstituteComponent**
- **wfcWComponentPath::GetSubstitutionType**

The method `pfcComponentPath::GetRoot` returns the assembly at the head of the component path object.

The method `pfcComponentPath::SetRoot` setsthe assembly at the head of the component path object as the root assembly.

The method `pfcComponentPath::GetComponentIds` returns the sequence of ids which is the path to the particular component.

The method `pfcComponentPath::SetComponentIds` sets the path from the root assembly to the component through various subassemblies containing this component.

The method `pfcComponentPath::GetLeaf` returns the solid model at the end of the component path.

The method `pfcComponentPath::GetTransform` returns the coordinate system transformation between the assembly and the particular component. It has an option to provide the transformation from bottom to top, or from top to bottom. This method describes the current position and the orientation of the assembly component in the root assembly.

The method `pfcComponentPath::SetTransform` applies a temporary transformation to the assembly component, similar to the transformation that takes place in an exploded state. The transformation will only be applied if the assembly is using DynamicPositioning.

The method `pfcComponentPath::GetIsVisible` identifies if a particular component is visible in any simplified representation.

The methods `wfcSubstituteComponent::GetSubCompPath` and `wfcSubstituteComponent::GetSubCompFeat` return the component path and handle to the component feature of the substituted component.

The method `wfcWComponentPath::GetSubstituteComponent` returns the component path and handle to the substituted component, when the replacing component is a simplified representation. The method `wfcWComponentPath::GetSubstitutionType` returns the substitution type of the simplified representation using the enumerated type `wfcSubstitutionType`:

- `wfcSUBSTITUTE_NONE`—Specifies that no substitution type has been defined.

- `wfcSUBSTITUTE_INTERCHG`—Specifies that the component is substituted with an interchange assembly component or a family table.
- `wfcSUBSTITUTE_PRT_REP`—Specifies that the part is substituted with a simplified representation.
- `wfcSUBSTITUTE_ASM_REP`—Specifies that the assembly is substituted with a simplified representation.
- `wfcSUBSTITUTE_ENVELOPE`—Specifies that the assembly is substituted with an envelope.
- `wfcSubstitutionType_nil`—`NULL` value.

Displayed Entities

Methods Introduced:

- **wfcWComponentPath::ListDisplayedPoints**
- **wfcWComponentPath::ListDisplayedCsyses**
- **wfcWComponentPath::ListDisplayedCurves**
- **wfcWComponentPath::ListDisplayedQuilts**

The methods in this section return the list of entities, that is, points, coordinate systems, datum curves, and quilts that are currently displayed in an assembly.

Assembling Components

Methods Introduced:

- **pfcAssembly::AssembleComponent**
- **pfcAssembly::AssembleByCopy**
- **pfcComponentFeat::GetConstraints**
- **pfcComponentFeat::SetConstraints**
- **pfcComponentFeat::GetConstraintsWithCompPath**
- **wfcWComponentFeat::RemoveConstraint**
- **wfcWAssembly::AutoInterchange**
- **wfcWAssembly::CreateAssemblyItem**
- **wfcWAssembly::GetConnectors**
- **wfcWAssembly::GetHarnesses**
- **wfcWAssembly::GetLinestocks**
- **wfcLineStock::GetName**
- **wfcLineStock::SetName**

- **wfcWAssembly::GetSpools**
- **wfcSpool::GetName**
- **wfcSpool::SetName**
- **wfcWAssembly::ListDisplayedComponents**
- **wfcExternalFeatRefAsmComp::GetPathToOwner**
- **wfcExternalFeatRefAsmComp::SetPathToOwner**
- **wfcExternalFeatRefAsmComp::GetPathToRef**
- **wfcExternalFeatRefAsmComp::SetPathToRef**
- **wfcWExternalFeatureReference::GetAsmcomponents**
- **wfcWExternalFeatureReference::GetFeature**

The method `pfcAssembly::AssembleComponent` adds a specified component model to the assembly at the specified initial position. The position is specified in the format defined by the interface `pfcTransform3D`. Specify the orientation of the three axes and the position of the origin of the component coordinate system, with respect to the target assembly coordinate system.

Note

If the transform matrix passed as the initial position of the component is incorrect and non-orthonormal, the method `pfcAssembly::AssembleComponent` returns the error `pfcXToolkitBadInputs`. In such scenario, you can use the method `pfcMakeMatrixOrthonormal` to convert this non-orthonormal matrix to an orthonormal matrix.

The method `pfcAssembly::AssembleByCopy` creates a new component in the specified assembly by copying from the specified component. If no model is specified, then the new component is created empty. The input parameters for this method are:

- *LeaveUnplaced*—If true the component is unplaced. If false the component is placed at a default location in the assembly. Unplaced components belong to an assembly without being assembled or packaged. These components appear in the model tree, but not in the graphic window. Unplaced components can be constrained or packaged by selecting them from the model tree for redefinition. When its parent assembly is retrieved into memory, an unplaced component is also retrieved.
- *ModelToCopy*—Specify the model to be copied into the assembly
- *NewModelName*—Specify a name for the copied model

The method `pfcComponentFeat::GetConstraints` retrieves the constraints for a given assembly component.

The method `pfcComponentFeat::SetConstraints` allows you to set the constraints for a specified assembly component. The input parameters for this method are:

- *Constraints*—Constraints for the assembly component. These constraints are explained in detail in the later sections.
- *ReferenceAssembly*—The path to the owner assembly, if the constraints have external references to other members of the top level assembly. If the constraints are applied only to the assembly component then the value of this parameter should be null.

This method modifies the component feature data and regenerates the assembly component.

The method `pfcComponentFeat::GetConstraintsWithCompPath` retrieves the constraints for a given assembly component using the input argument *CompPath* which is the path to the owner assembly. Pass this input argument *CompPath*, if the constraints have references to other members of the top level assembly. Pass it as `Null`, if the constraints have references only to the owner assembly.

The method `wfcWComponentFeat::RemoveConstraint` removes one or all the constraints for the specified assembly component. It takes the index of the constraint as its input argument.

The method `wfcWAssembly::AutoInterchange` interchanges an assembly component with another component that contains equivalent assembly constraints. The input parameters are:

- *ComponentIDs*—Specifies the component identifiers of the replaced members from the assembly nodes.
- *ReplacementModel*—Specifies the replacing component, which can be a part or a sub-assembly.

The method `wfcWAssembly::CreateAssemblyItem` creates an assembly item that defines the flexible components. Refer to the section Flexible Components and Inheritance Features in an Assembly on page 494 for more information on flexible items.

The method `wfcWAssembly::GetConnectors` returns the list of connectors defined for the specified assembly.

The method `wfcWAssembly::GetHarnesses` returns the list of harnesses defined for the specified assembly.

The method `wfcWAssembly::GetLinestocks` returns the list of linestocks defined for the specified assembly. Use the methods `wfcLineStock::GetName` and `wfcLineStock::SetName` to get and set the name of linestock in an assembly.

The method `wfcWAssembly::GetSpools` returns the list of spools defined for the specified assembly. Use the methods `wfcSpool::GetName` and `wfcSpool::SetName` to get and set the name of spool in an assembly.

Use the method `wfcWAssembly::ListDisplayedComponents` to retrieve a list of all the currently displayed components in a solid.

The methods `wfcExternalFeatRefAsmComp::GetPathToOwner` and `wfcExternalFeatRefAsmComp::SetPathToOwner` retrieve and set the path from the external specified reference to the component that owns the specified external reference.

The methods `wfcExternalFeatRefAsmComp::GetPathToRef` and `wfcExternalFeatRefAsmComp::SetPathToRef` retrieve and set the path from the external specified reference to the component from which the external reference was created.

The method `wfcWExternalFeatureReference::GetAsmcomponents` retrieves from the specified external reference a path to the component from which the reference was created. It also returns a path to the component that owns the specified external reference.

The method `wfcWExternalFeatureReference::GetFeature` retrieves from the specified external reference a feature referred to by the external reference.

Constraint Attributes

Methods Introduced:

- **pfcConstraintAttributes::Create**
- **pfcConstraintAttributes::GetForce**
- **pfcConstraintAttributes::SetForce**
- **pfcConstraintAttributes::GetIgnore**
- **pfcConstraintAttributes::SetIgnore**

The method `pfcConstraintAttributes::Create` returns the constraint attributes object based on the values of the following input parameters:

- *Ignore*—Constraint is ignored during regeneration. Use this capability to store extra constraints on the component, which allows you to quickly toggle between different constraints.
- *Force*—Constraint has to be forced for line and point alignment.

- *None*—No constraint attributes. This is the default value.

Use the `Get` methods to retrieve the values of the input parameters specified above and the `Set` methods to modify the values of these input parameters.

Assembling a Component Parametrically

You can position a component relative to its neighbors (components or assembly features) so that its position is updated as its neighbors move or change. This is called parametric assembly. Creo allows you to specify constraints to determine how and where the component relates to the assembly. You can add as many constraints as you need to make sure that the assembly meets the design intent.

Methods Introduced:

- **pfcComponentConstraint::Create**
- **pfcComponentConstraint::GetType**
- **pfcComponentConstraint::SetType**
- **pfcComponentConstraint::SetAssemblyReference**
- **pfcComponentConstraint::GetAssemblyReference**
- **pfcComponentConstraint::SetAssemblyDatumSide**
- **pfcComponentConstraint::GetAssemblyDatumSide**
- **pfcComponentConstraint::SetComponentReference**
- **pfcComponentConstraint::GetComponentReference**
- **pfcComponentConstraint::SetComponentDatumSide**
- **pfcComponentConstraint::GetComponentDatumSide**
- **pfcComponentConstraint::SetOffset**
- **pfcComponentConstraint::GetOffset**
- **pfcComponentConstraint::SetAttributes**
- **pfcComponentConstraint::GetAttributes**
- **pfcComponentConstraint::SetUserDefinedData**
- **pfcComponentConstraint::GetUserDefinedData**

The method `pfcComponentConstraint::Create` returns the component constraint object having the following parameters:

- *ComponentConstraintType*—Using the `TYPE` options, you can specify the placement constraint types. They are as follows:
  - `pfcASM_CONSTRAINT_MATE`—Use this option to make two surfaces touch one another, that is coincident and facing each other.

- pfcASM_CONSTRAINT_MATE_OFF—Use this option to make two planar surfaces parallel and facing each other.
- pfcASM_CONSTRAINT_ALIGN—Use this option to make two planes coplanar, two axes coaxial and two points coincident. You can also align revolved surfaces or edges.
- pfcASM_CONSTRAINT_ALIGN_OFF—Use this option to align two planar surfaces at an offset.
- pfcASM_CONSTRAINT_INSERT—Use this option to insert a "male" revolved surface into a ``female" revolved surface, making their respective axes coaxial.
- pfcASM_CONSTRAINT_ORIENT—Use this option to make two planar surfaces to be parallel in the same direction.
- pfcASM_CONSTRAINT_CSYS—Use this option to place a component in an assembly by aligning the coordinate system of the component with the coordinate system of the assembly.
- pfcASM_CONSTRAINT_TANGENT—Use this option to control the contact of two surfaces at their tangents.
- pfcASM_CONSTRAINT_PNT_ON_SRF—Use this option to control the contact of a surface with a point.
- pfcASM_CONSTRAINT_EDGE_ON_SRF—Use this option to control the contact of a surface with a straight edge.
- pfcASM_CONSTRAINT_DEF_PLACEMENT—Use this option to align the default coordinate system of the component to the default coordinate system of the assembly.
- pfcASM_CONSTRAINT_SUBSTITUTE—Use this option in simplified representations when a component has been substituted with some other model
- pfcASM_CONSTRAINT_PNT_ON_LINE—Use this option to control the contact of a line with a point.
- pfcASM_CONSTRAINT_FIX—Use this option to force the component to remain in its current packaged position.
- pfcASM_CONSTRAINT_AUTO—Use this option in the user interface to allow an automatic choice of constraint type based upon the references.
- pfcASM_CONSTRAINT_EXPLICIT

- *AssemblyReference*—A reference in the assembly.

- *AssemblyDatumSide*—Orientation of the assembly. This can have the following values:
  - `Yellow`—The primary side of the datum plane which is the default direction of the arrow.
  - `Red`—The secondary side of the datum plane which is the direction opposite to that of the arrow.
- *ComponentReference*—A reference on the placed component.
- *ComponentDatumSide*—Orientation of the assembly component. This can have the following values:
  - `Yellow`—The primary side of the datum plane which is the default direction of the arrow.
  - `Red`—The secondary side of the datum plane which is the direction opposite to that of the arrow.
- *Offset*—The mate or align offset value from the reference.
- *Attributes*—Constraint attributes for a given constraint.
- *UserDefinedData*—A string that specifies user data for the given constraint.

Use the `Get` methods to retrieve the values of the input parameters specified above and the `Set` methods to modify the values of these input parameters.

Redefining and Rerouting Assembly Components

These methods enable you to reroute previously assembled components, just as in an interactive Creo session.

Methods Introduced:

- **pfcComponentFeat::RedefineThroughUI**
- **pfcComponentFeat::MoveThroughUI**

The method `pfcComponentFeat::RedefineThroughUI` must be used in interactive Creo Object TOOLKIT C++ applications. This method displays the dialog box for constraints. This enables the end user to redefine the constraints interactively. The control returns to Creo Object TOOLKIT C++ application when the user selects **OK** or **Cancel** and the dialog box is closed.

The method `pfcComponentFeat::MoveThroughUI` invokes a dialog box that prompts the user to interactively reposition the components. This interface enables the user to specify the translation and rotation values. The control returns to Creo Object TOOLKIT C++ application when the user selects **OK** or **Cancel** and the dialog box is closed.

Exploded Assemblies

These methods enable you to determine and change the explode status of the assembly object.

Methods Introduced:

- **pfcAssembly::GetIsExploded**
- **pfcAssembly::Explode**
- **pfcAssembly::UnExplode**
- **pfcAssembly::GetActiveExplodedState**
- **pfcAssembly::GetDefaultExplodedState**
- **pfcExplodedState::Activate**

The methods `pfcAssembly::Explode` and `pfcAssembly::UnExplode` enable you to determine and change the explode status of the assembly object.

The method `pfcAssembly::GetIsExploded` reports whether the specified assembly is currently exploded. Use this method in the assembly mode only. The exploded status of an assembly depends on the mode. If an assembly is opened in the drawing mode, the state of the assembly in the drawing view is displayed. The drawing view does not represent the actual exploded state of the assembly.

The method `pfcAssembly::GetActiveExplodedState` returns the current active explode state.

The method `pfcAssembly::GetDefaultExplodedState` returns the default explode state.

The method `pfcExplodedState::Activate` activates the specified explode state representation.

Accessing Exploded States

Methods Introduced:

- **wfcWAssembly::GetExplodeStateFromName**
- **wfcWAssembly::GetExplodeStateFromId**
- **wfcWAssembly::SelectExplodedState**
- **wfcWExplodedState::GetExplodedStateName**
- **wfcWExplodedState::SetExplodedStateName**
- **wfcWExplodedState::GetExplodedcomponents**
- **wfcWExplodedState::GetExplodedStateMoves**
- **wfcWExplodedState::SetExplodedStateMoves**
- **wfcExplodedAnimationMoveInstruction::Create**

- **wfcExplodedAnimationMoveInstruction::GetCompSet**
- **wfcExplodedAnimationMoveInstruction::SetCompSet**
- **wfcExplodedAnimationMoveInstruction::GetMove**
- **wfcExplodedAnimationMoveInstruction::SetMove**
- **wfcExplodedAnimationMove::Create**
- **wfcExplodedAnimationMove::GetMoveType**
- **wfcExplodedAnimationMove::SetMoveType**
- **wfcExplodedAnimationMove::GetStartPoint**
- **wfcExplodedAnimationMove::SetStartPoint**
- **wfcExplodedAnimationMove::GetDirVector**
- **wfcExplodedAnimationMove::SetDirVector**
- **wfcExplodedAnimationMove::GetValue**
- **wfcExplodedAnimationMove::SetValue**

The methods `wfcWAssembly::GetExplodeStateFromName` and `wfcWAssembly::GetExplodeStateFromId` return the exploded state representation of a solid with the specified name and ID respectively.

The method `wfcWAssembly::SelectExplodedState` enables you to select a specific exploded state from the list of defined exploded states.

The method `wfcWExplodedState::GetExplodedStateName` returns the name of the exploded state. Use the method `wfcWExplodedState::SetExplodedStateName` to set the name of the exploded state.

The method `wfcWExplodedState::GetExplodedcomponents` returns an array of assembly component paths that are included in the exploded state.

The method `wfcWExplodedState::GetExplodedStateMoves` retrieves an array of moves for the specified exploded state. The sequence of moves defines the exploded position of an assembly component or a set of assembly components. For example, you can move an assembly component over the X-axis, rotate it over a selected edge, and then move over the Y-axis. The final position of the assembly component is attained by performing these three moves. Use the method `wfcWExplodedState::SetExplodedStateMoves` to set the array of moves of an exploded state.

The method `wfcExplodedAnimationMoveInstruction::Create` creates a `wfcExplodedAnimationMoveInstruction` data object that contains information about the moves of an exploded state.

The method `wfcExplodedAnimationMoveInstruction::GetCompSet` returns an array that contains the full path to the assembly component. Use the method `wfcExplodedAnimationMoveInstruction::SetCompSet` to set an array of paths to the assembly components.

The methods `wfcExplodedAnimationMoveInstruction::GetMove` and `wfcExplodedAnimationMoveInstruction::SetMove` retrieve and set the move of the exploded state.

The method `wfcExplodedAnimationMove::Create` creates the `wfcExplodedAnimationMove` data object.

The method `wfcExplodedAnimationMove::GetMoveType` returns the type of move for the exploded state. The move can have one of the following values:

- `wfcEXPLDANIM_MOVE_TRANSLATE`
- `wfcEXPLDANIM_MOVE_ROTATE`

Use the method `wfcExplodedAnimationMove::SetMoveType` to set the move type using the enumerated type `wfcExplodedAnimationMoveType`.

The methods `wfcExplodedAnimationMove::GetStartPoint` and `wfcExplodedAnimationMove::SetStartPoint` get and set the start location of the transitional direction or the rotational axis, depending upon the selected move type.

The methods `wfcExplodedAnimationMove::GetDirVector` and `wfcExplodedAnimationMove::SetDirVector` get and set the direction vector for the transitional direction or the rotational axis, depending upon the selected move type.

Depending upon the selected move type, the methods `wfcExplodedAnimationMove::GetValue` and `wfcExplodedAnimationMove::SetValue` get and set the translational distance or the rotation angle.

Manipulating Exploded States

Methods Introduced:

- **wfcWAssembly::CreateExplodedState**
- **wfcWAssembly::DeleteExplodedState**

The method `wfcWAssembly::CreateExplodedState` creates a new exploded state based on the values of the following input arguments:

- *name*—Specifies the name of the exploded state. This argument cannot be `NULL`.

- *AnimMoveInstructions*—Specifies an array of `wfcExplodedAnimationMoveInstruction` objects.

Use the method `wfcWAssembly::DeleteExplodedState` to delete a specified exploded state.

Skeleton Models

Skeleton models are a 3-dimensional layout of the assembly. These models are holders or distributors of critical design information, and can represent space requirements, important mounting locations, and motion.

Methods Introduced:

- **pfcAssembly::AssembleSkeleton**
- **pfcAssembly::AssembleSkeletonByCopy**
- **pfcAssembly::GetSkeleton**
- **pfcAssembly::DeleteSkeleton**
- **pfcSolid::GetIsSkeleton**

The method `pfcAssembly::AssembleSkeleton` adds an existing skeleton model to the specified assembly.

The method `pfcAssembly::GetSkeleton` returns the skeleton model of the specified assembly.

The method `pfcAssembly::DeleteSkeleton` deletes a skeleton model component from the specified assembly.

The method `pfcAssembly::AssembleSkeletonByCopy` adds a specified skeleton model to the assembly. The input parameters for this method are:

- *SkeletonToCopy*—Specify the skeleton model to be copied into the assembly
- *NewSkeletonName*—Specify a name for the copied skeleton model

The method `pfcSolid::GetIsSkeleton` determines if the specified part model is a skeleton model or a concept model. It returns a true if the model is a skeleton else it returns a false.

Flexible Components and Inheritance Features in an Assembly

A flexible component allows variance of items such as features, dimensions, annotations, and parameters of a model in the context of an assembly. The methods in this section describe the properties for the flexible component.

An Inheritance feature allows one-way associative propagation of geometry and feature data from a reference part to target part within an assembly. The reference part is the original part and the target part contains the inheritance features. Inheritance features are always created by referencing existing parts. An inheritance feature begins with all of its geometry and data identical to the reference part from which it is derived.

Use inheritance features or flexible components to create variations of a model in an assembly. This section refers collectively to inheritance features and flexible components as "variant features".

Methods Introduced:

- **wfcWComponentFeat::CreateFlexibleModel**
- **wfcWComponentFeat::CreatePredefinedFlexibilityComponent**
- **wfcWComponentFeat::IsFlexible**
- **wfcWComponentFeat::SetAsFlexible**
- **wfcWComponentFeat::UnsetAsFlexible**

The method `wfcWComponentFeat::CreateFlexibleModel` creates a flexible model from the specified flexible model component.

The method `wfcWComponentFeat::CreatePredefinedFlexibilityComponent` converts the specified assembly component to a flexible component. It uses the variant items with predefined flexibility to create the flexible component.

Use the method `wfcWComponentFeat::IsFlexible` to identify if the specified assembly component is a flexible component. If the method returns true, the component is a flexible component.

The method `wfcWComponentFeat::SetAsFlexible` converts a specified assembly component to a flexible component by using the variant items specified in the input argument.

The method `wfcWComponentFeat::UnsetAsFlexible` converts a flexible component to a regular component.

Variant Feature Model

Method Introduced:

- **wfcWModel::IsVariantFeatModel**

The method `wfcWModel::IsVariantFeatModel` returns a boolean to identify if a model pointer is from an inheritance feature or a flexible component. The method returns true, if the model pointer is from an inheritance feature, else it returns false.

Variant Items for Flexible Components

Varied items define component flexibility. You define components, dimensions, features, parameters, references, gtols, and so on in the original part. The methods described in this section enable you to assign values to the varied items to define component flexibility in the assembly

Methods Introduced:

- **wfcAssemblyItemInstructions::Create**
- **wfcAssemblyItemInstructions::GetItemOwner**
- **wfcAssemblyItemInstructions::SetItemOwner**
- **wfcAssemblyItemInstructions::GetItemCompPath**
- **wfcAssemblyItemInstructions::SetItemCompPath**
- **wfcAssemblyItemInstructions::GetItemId**
- **wfcAssemblyItemInstructions::SetItemId**
- **wfcAssemblyItemInstructions::GetItemName**
- **wfcAssemblyItemInstructions::SetItemName**
- **wfcAssemblyItemInstructions::GetItemType**
- **wfcAssemblyItemInstructions::SetItemType**
- **wfcWFeature::ListVariedItems**
- **wfcWFeature::ListVariedParameters**

The method `wfcAssemblyItemInstructions::Create` creates a new instance of the object `wfcAssemblyItemInstruction` that contains the instructions to define a variant item for a flexible component. Specify the model owner, item type, and item ID as input arguments of this method. The input arguments are:

- *ItemOwner*— Specifies the model owner of the item or parameter.
- *ItemType*— Specifies the item type. The value of this argument is ignored for parameter. For parameter, pass this value as `pfcModelItemType_nil`.
- *ItemId*— Specifies the item identifier.
- *ItemName*— Specifies the parameter name.

The method `wfcAssemblyItemInstructions::GetItemOwner` gets the name of the model owner for the specified variant item or parameter.

The method `wfcAssemblyItemInstructions::SetItemCompPath` gets the component path for the variant item. Use the method `wfcAssemblyItemInstructions::SetItemCompPath` to set the component path for the variant item.

The methods `wfcAssemblyItemInstructions::GetItemId` and `wfcAssemblyItemInstructions::SetItemId` get and set the identifier for the variant item or parameter.

The methods `wfcAssemblyItemInstructions::GetItemName` and `wfcAssemblyItemInstructions::SetItemName` get and set the name of the variant parameter.

The methods `wfcAssemblyItemInstructions::GetItemType` and `wfcAssemblyItemInstructions::SetItemType` get and set the type of variant item using the enumerated type `pfcModelItemType`. This value is not required for variant parameter. If the object is a variant parameter, pass this value as `pfcModelItemType_nil`.

The method `wfcWFeature::ListVariedItems` lists the variant items, that is dimensions, features, and annotations owned by an inheritance feature or flexible component.

The method `wfcWFeature::ListVariedParameters` lists the variant parameters owned by an inheritance feature or flexible component.

Gathering Components by Rule

Creo application provides tools to search for components within large assemblies. This section describes how to access some of the functionality through Creo Object TOOLKIT C++.

You can specify different types of rules to search and generate a list of components that follow these rules. You can gather components using one or more of the following rules:

- By model name
- By parameters, using an expression
- By location with a zone
- By distance from a point
- By size
- By an existing simplified representation

Method Introduced:

- **wfcAssemblyRule::GetRuleType**

Use the method `wfcAssemblyRule::GetRuleType` to get the type of rule that was used to search for the component. The types of rules are:

- *wfcRULE_NONE*Specifies that no rule has been set.
- *wfcRULE_NAME*—Specifies the rule to search components by model name.
- *wfcRULE_EXPR*—Specifies the rule to search components by parameters, using an expression.

- *wfcRULE_ZONE*—Specifies the rule to search components by location with a zone.
- *wfcRULE_DIST*—Specifies the rule to search components by distance from a point.
- *wfcRULE_SIZE*—Specifies the rule to search components by size.
- *wfcRULE_SIMP_REP*—Specifies the rule to search components by an existing simplified representation.

Gathering Components by Model Name

Methods Introduced:

- **wfcAssemblyNameRule::Create**
- **wfcAssemblyNameRule::SetNameMask**
- **wfcAssemblyNameRule::GetNameMask**

The class `wfcAssemblyNameRule` specifies the rule to gather components by model name. This class is an interface that can be used to define the name rule and contains the methods described below:

Use the method `wfcAssemblyNameRule::Create` to create a rule to search for components by name. Specify the search string as the input parameter *NameMask* for this method. Use wildcards to improve the search results.

- Use the method `wfcAssemblyNameRule::SetNameMask` to set the search string for the name rule.
- Use the method `wfcAssemblyNameRule::GetNameMask` to get the search string used in the name rule.

Note

The attribute 'NameMask' can be a wildcard character that is, you can specify wildcard characters for object names, their extensions, and directory names. For more information, refer to the online Help.

Gathering Components by Size

Methods Introduced:

- **wfcAssemblySizeRule::Create**
- **wfcAssemblySizeRule::SetAbsolute**
- **wfcAssemblySizeRule::GetAbsolute**
- **wfcAssemblySizeRule::SetGreaterThan**

- **wfcAssemblySizeRule::GetGreaterThan**
- **wfcAssemblySizeRule::SetIncludeDatums**
- **wfcAssemblySizeRule::GetIncludeDatums**
- **wfcAssemblySizeRule::SetValue**
- **wfcAssemblySizeRule::GetValue**

The class `wfcAssemblySizeRule` is an interface that can be used to define the rule to gather components by their size.

Use the method `wfcAssemblySizeRule::Create` to create a rule to search for components by size. The input parameters of this method are as follows:

- *Absolute*—If set to `true`, compares the absolute size of the model with respect to the assembly, else compares the relative size of the model with respect to the assembly.
- *GreaterThan*—If set to `true`, searches for components that are larger than the specified size, else searches for components that are smaller.
- *IncludeDatums*—If set to `true`, gather the model volume using the bounding box, else use the regeneration outline to gather the model volume.
- *Value*—Specifies the actual size against which the size of the model will be compared.

Use the methods `wfcAssemblySizeRule::SetAbsolute` and `wfcAssemblySizeRule::GetAbsolute` to set and get the absolute or relative size of the model respectively. Set the value of the input parameter to `true`, to compare the absolute size of the model with the size of the top-level assembly. Specify `false`, to compare the relative size of the model with respect to the assembly. For the relative size, specify a value in the range of 0.0 to 1.0. The method compares the component size to that of the top-level assembly and uses this ratio to determine whether the component should be gathered.

Use the methods `wfcAssemblySizeRule::SetGreaterThan` and `wfcAssemblySizeRule::GetGreaterThan` to set and get the absolute or relative size of the model respectively. To search for components greater than the specified size, set the parameter *GreaterThan* to `true`. If you set the parameter to `false`, the method gathers the components that are smaller than the specified size.

Use the methods `wfcAssemblySizeRule::SetIncludeDatums` and `wfcAssemblySizeRule::GetIncludeDatums` to set and get the model size using the bounding box or regeneration outline. Specify the value `true` to use bounding box, or `false` for regeneration outline.

Use the methods `wfcAssemblySizeRule::SetValue` and `wfcAssemblySizeRule::GetValue` to set the get the size against which the specified model will be compared. The valid range for this parameter is from 0.0 to 1.0 only if the *Absolute* attribute is set to `false`.

Gathering Components by Simplified Representation

Methods Introduced:

- **wfcAssemblySimpRepRule::Create**
- **wfcAssemblySimpRepRule::GetRuleSimpRep**
- **wfcAssemblySimpRepRule::SetRuleSimpRep**

The class `wfcAssemblySimpRepRule` specifies the rule to gather components that belong to the specified simplified representation. This class is an interface that can be used to define the simplified representation rule and contains the methods described below:

Use the method `wfcAssemblySimpRepRule::Create` to create a rule to search for components by simplified representation.

Use the methods `wfcAssemblySimpRepRule::GetRuleSimpRep` and `wfcAssemblySimpRepRule::SetRuleSimpRep` to get and set the simplified representation to be used for the rule.

Gathering Components by Parameters

Methods Introduced:

- **wfcAssemblyExpressionRule::Create**
- **wfcAssemblyExpressionRule::GetExpressions**
- **wfcAssemblyExpressionRule::SetExpressions**

The class `wfcAssemblyExpressionRule` specifies the rule to gather components by parameter. This class is an interface that can be used to define the parameter rule and contains the methods described below:

Use the method `wfcAssemblyExpressionRule::Create` to create a rule to search for components by parameter expressions. The parameter of this method is given below:

- `Expressions`—Specifies the expression created using the parameters and logical operators.

Use the methods `wfcAssemblyExpressionRule::GetExpressions` and `wfcAssemblyExpressionRule::SetExpressions` to get and set the parameter expressions. You can specify an expression in the relations format to search for components of a particular parameter value. For example, consider the following expression:

```
type == "electrical" | cost <= 10
```

When you supply this expression to the rule, it searches for components that have a “cost” parameter of less than or equal to 10, or for components whose type parameter is set to “electrical.”

Gathering Components by Zone

Methods Introduced:

- **wfcAssemblyZoneRule::Create**
- **wfcAssemblyZoneRule::GetZoneFeature**
- **wfcAssemblyZoneRule::SetZoneFeature**

The class `wfcAssemblyZoneRule` specifies the rule to gather components by the specified zone feature. This class is an interface that can be used to define the zone rule and contains the methods described below:

Use the method `wfcAssemblyZoneRule::Create` to create a rule to search for components by the zone feature. The parameter of this method is given below:

- `ZoneFeature`—Gathers all the components that belong to the specified zone feature.

Use the methods `wfcAssemblyZoneRule::GetZoneFeature` and `wfcAssemblyZoneRule::SetZoneFeature` to get and set the zone feature. When you create a zone, the method creates a zone feature in the top-level assembly.

Gathering Components by Distance from a Point

Methods Introduced:

- **wfcAssemblyDistanceRule::Create**
- **wfcAssemblyDistanceRule::GetCenter**
- **wfcAssemblyDistanceRule::SetCenter**
- **wfcAssemblyDistanceRule::GetDistance**
- **wfcAssemblyDistanceRule::SetDistance**
- **wfcAssemblyDistanceRule::GetIncludeDatums**
- **wfcAssemblyDistanceRule::SetIncludeDatums**

The class `wfcAssemblyDistanceRule` specifies the rule to gather components within specified distance from a point. This class is an interface that can be used to define the distance rule and contains the methods described below:

Use the method `wfcAssemblyDistanceRule::Create` to create a rule to search for components within specified distance. The parameters of this method are given below:

- `Center`—Specifies the centre point of the specified region.
- `Distance`—Specifies the distance from center point.
- `IncludeDatums`—Specifies the type of datum to be included.

Use the methods `wfcAssemblyDistanceRule::GetCenter` and `wfcAssemblyDistanceRule::SetCenter` to get and set the center point from which the distance is measured.

Use the method `wfcAssemblyDistanceRule::GetDistance` to get the distance against which the components will be gathered. The method `wfcAssemblyDistanceRule::SetDistance` sets the distance from the centre point of the model.

Use the methods `wfcAssemblyDistanceRule::GetIncludeDatums` and `wfcAssemblyDistanceRule::SetIncludeDatums` to set and get the model size respectively. If set to `true`, gather the model volume using the bounding box, else use the regeneration outline to gather the model volume.

Listing Components By Rule

Method Introduced:

- **wfcWAssembly::ListComponentsByAssemblyRule**

Use the method `wfcWAssembly::ListComponentsByAssemblyRule` to list all the components that satisfy the specified rule. The input parameter for this method is:

- *AsmRule* —Specify the rule, against which the components will be searched.

29

Family Tables

This chapter describes how to use Creo Object TOOLKIT C++ classes and methods to access and manipulate family table information.

Working with Family Tables

Creo Object TOOLKIT C++ provides several methods for accessing family table information. Because every model inherits from the interface `pfcFamilyMember`, every model can have a family table associated with it.

Accessing Instances

Methods Introduced:

- **pfcFamilyMember::GetParent**
- **pfcFamilyMember::GetImmediateGenericInfo**
- **pfcFamilyMember::GetTopGenericInfo**
- **pfcFamilyTableRow::CreateInstance**
- **pfcFamilyMember::ListRows**
- **pfcFamilyMember::GetRow**
- **pfcFamilyMember::RemoveRow**
- **pfcFamilyTableRow::GetInstanceName**
- **pfcFamilyTableRow::GetIsLocked**
- **pfcFamilyTableRow::SetIsLocked**

To get the generic model for an instance, call the method `pfcFamilyMember::GetParent`.

When you now call the method `pfcFamilyMember::GetParent`, it throws an exception `pfcXToolkitCantOpen`, if the immediate generic of a model instance in a nested family table is currently not in session. Handle this exception and use the method `pfcFamilyMember::GetImmediateGenericInfo` to get the model descriptor of the immediate generic model. This information can be used to retrieve the immediate generic model. If you wish to switch to the pre-Wildfire 4.0 mode, set the configuration option `retrieve_instance_dependencies` to `instance_and_generic_deps`.

To get the model descriptor of the top generic model, call the method `pfcFamilyMember::GetTopGenericInfo`.

Similarly, the method `pfcFamilyTableRow::CreateInstance` returns an instance model created from the information stored in the `pfcFamilyTableRow` object.

The method `pfcFamilyMember::ListRows` returns a sequence of all rows in the family table, whereas `pfcFamilyMember::GetRow` gets the row object with the name you specify.

Use the method `pfcFamilyMember::RemoveRow` to permanently delete the row from the family table.

The method `pfcFamilyTableRow::GetInstanceName` returns the name that corresponds to the invoking row object.

To control whether the instance can be changed or removed, call the methods `pfcFamilyTableRow::GetIsLocked` and `pfcFamilyTableRow::SetIsLocked`.

Accessing Columns

Methods Introduced:

- **pfcFamilyMember::ListColumns**
- **pfcFamilyMember::GetColumn**
- **pfcFamilyMember::RemoveColumn**
- **pfcFamilyTableColumn::GetSymbol**
- **pfcFamilyTableColumn::GetType**
- **pfcFamColModelItem::GetRefItem**
- **pfcFamColParam::GetRefParam**

The method `pfcFamilyMember::ListColumns` returns a sequence of all columns in the family table.

The method `pfcFamilyMember::GetColumn` returns a family table column, given its symbolic name.

To permanently delete the column from the family table and all changed values in all instances, call the method `pfcFamilyMember::RemoveColumn`.

The method `pfcFamilyTableColumn::GetSymbol` returns the string symbol at the top of the column, such as `D4` or `F5`.

The method `pfcFamilyTableColumn::GetType` returns an enumerated value indicating the type of parameter governed by the column in the family table.

The method `pfcFamColModelItem::GetRefItem` returns the `pfcModelItem` (`pfcFeature` or `pfcDimension`) controlled by the column, whereas `pfcFamColParam::GetRefParam` returns the `pfcParameter` controlled by the column.

Accessing Cell Information

Methods Introduced:

- **pfcFamilyMember::GetCell**
- **pfcFamilyMember::GetCellIsDefault**
- **pfcFamilyMember::SetCell**
- **pfcParamValue::GetStringValue**

- **pfcParamValue::GetIntValue**
- **pfcParamValue::GetDoubleValue**
- **pfcParamValue::GetBoolValue**

The method `pfcFamilyMember::GetCell` returns a string `pfcParamValue` that corresponds to the cell at the intersection of the row and column arguments. Use the method `pfcFamilyMember::GetCellIsDefault` to check if the value of the specified cell is the default value, which is the value of the specified cell in the generic model.

The method `pfcFamilyMember::SetCell` assigns a value to a column in a particular family table instance.

The `pfcParamValue::GetStringValue`, `pfcParamValue::GetIntValue`, `pfcParamValue::GetDoubleValue`, and `pfcParamValue::GetBoolValue` methods are used to get the different types of parameter values.

Creating Family Table Instances

Methods Introduced:

- **pfcFamilyMember::AddRow**
- **pfcCreateStringParamValue**
- **pfcCreateIntParamValue**
- **pfcCreateDoubleParamValue**
- **pfcCreateBoolParamValue**

Use the method `pfcFamilyMember::AddRow` to create a new instance with the specified name, and, optionally, the specified values for each column. If you do not pass in a set of values, the value `*` will be assigned to each column. This value indicates that the instance uses the generic value.

Creating Family Table Columns

Methods Introduced:

- **pfcFamilyMember::CreateDimensionColumn**
- **pfcFamilyMember::CreateParamColumn**
- **pfcFamilyMember::CreateFeatureColumn**
- **pfcFamilyMember::CreateComponentColumn**
- **pfcFamilyMember::CreateCompModelColumn**

- **pfcFamilyMember::CreateGroupColumn**
- **pfcFamilyMember::CreateMergePartColumn**
- **pfcFamilyMember::CreateColumn**
- **pfcFamilyMember::AddColumn**
- **pfcCreateStringParamValue**
- **pfcParamValues::create**

The above methods initialize a column based on the input argument. These methods assign the proper symbol to the column header.

The method `pfcFamilyMember::CreateColumn` creates a new column given a properly defined symbol and column type. The results of this call should be passed to the method `pfcFamilyMember::AddColumn` to add the column to the model's family table.

The method `pfcFamilyMember::AddColumn` adds the column to the family table. You can specify the values; if you pass nothing for the values, the method assigns the value `*` to each instance to accept the column's default value.

Operations on Family Table Instances

Methods Introduced:

- **wfcWFamilyTableRow::GetModelFromDisk**
- **wfcWFamilyTableRow::GetModelFromSession**
- **wfcWFamilyTableRow::IsFlatState**
- **wfcWFamilyTableRow::IsModifiable**
- **wfcWFamilyMember::SelectRows**

Use the method `wfcWFamilyTableRow::GetModelFromDisk` to retrieve an instance of a model from the disk as an object of the class `wfcWModel`.

Use the method `wfcWFamilyTableRow::GetModelFromSession` to retrieve the handle to the instance model for the given instance if the model is in session.

Use the method `wfcWFamilyTableRow::IsFlatState` to identify if the family table instance, that is, a completely unbent instance of a sheet metal part. This method returns the value `true` if the family table instance is a flat state instance.

Use the method `wfcWFamilyTableRow::IsModifiable` to check if the given instance of a family table can be modified. This method returns the value `true` if the instance is modifiable. The input parameter for this method is:

- `ShowUI`—Specifies whether the **Conflicts** dialog box should be shown to resolve the conflicts, if detected. Pass the value `true` to show the dialog box.

Use the method `wfcWFamilyMember::SelectRows` to select one or more instances from the specified family table. The input parameter for this method is:

- `AllowMultiSelect`—Specify the value `true` to enable the selection of more than one instance in the family table.

Family Table Utilities

Methods Introduced:

- **wfcWFamilyMember::EditFamilyTable**
- **wfcWFamilyMember::EraseFamilyTable**
- **wfcWFamilyMember::GetFamilyTableStatus**
- **wfcWFamilyMember::IsModifiable**
- **wfcWFamilyMember::ShowFamilyTable**

Use the method `wfcWFamilyMember::EditFamilyTable` to edit the specified family table using the Pro/TABLE or another text editor.

Use the method `wfcWFamilyMember::EraseFamilyTable` to erase the specified family table.

Use the method `wfcWFamilyMember::GetFamilyTableStatus` to determine the validity status of the family table.

Use the method `wfcWFamilyMember::IsModifiable` to check whether the specified family table can be modified. This method returns the value `true` if the family table is modifiable. The input parameter for this method is:

- `ShowUI`—Specifies whether the **Conflicts** dialog box should be shown to resolve the conflicts, if detected. Pass the value `true` to show the dialog box.

Use the method `wfcWFamilyMember::ShowFamilyTable` to display the family table using Pro/TABLE or another text editor.

30

Action Listeners

This chapter describes the Creo Object TOOLKIT C++ methods that enable you to use action listeners.

Creo Object TOOLKIT C++ Action Listeners

An `ActionListener` in Creo Object TOOLKIT C++ is a class that is assigned to respond to certain events. You can assign action listeners to respond to events involving the following tasks:

- Changing windows
- Changing working directories
- Model operations
- Regenerating
- Creating, deleting, and redefining features
- Checking for regeneration failures

All action listeners in Creo Object TOOLKIT C++ are defined by a named pfc<Object>ActionListener. This interface defines the methods that can respond to various events.

Besides, for some listeners a default implementation class is provided, named pfcDefault<Object>ActionListener. This class has every available method overridden by an empty implementation. You create your own action listeners by extending the default class and overriding the methods for events that interest you. Default implementations are provided for the following listeners:

- `pfcFeatureActionListener`
- `pfcSolidActionListener`
- `pfcSessionActionListener`
- `pfcModelEventActionListener`
- `pfcModelActionListener`

Note

When notifications are set in Creo Object TOOLKIT C++ applications, every time an event is triggered, notification messages are added to the trail files. From Creo Parametric 2.0 M210 onward, a new environment variable `PROTK_LOG_DISABLE` enables you to disable this behavior. When set to `true`, the notifications messages are not added to the trail files.

Creating an ActionListener Implementation

You can create a proper `ActionListener` class using either of the following methods:

Define a separate class within the Creo Object TOOLKIT C++ file.

To use your action listener in different applications, define it in a separate file.

Action Sources

Methods introduced:

- **pfcActionSource::AddActionListener**
- **pfcActionSource::AddActionListenerWithType**
- **pfcActionSource::RemoveActionListener**

Many Creo Object TOOLKIT C++ classes inherit the `pfcActionSource` interface, but only the following classes currently make calls to the methods of registered `ActionListeners`:

- `pfcSession`
  - Session Action Listener
  - Model Action Listener
  - Solid Action Listener
  - Model Event Action Listener
  - Feature Action Listener
- `pfcUICommand`
  - UI Action Listener
- `pfcModel` (and it's subclasses)
  - Model Action Listener
  - Parameter Action Listener
- `pfcSolid` (and it's subclasses)
  - Solid Action Listener
  - Feature Action Listener
- `pfcFeature` (and it's subclasses)
  - Feature Action Listener

Note

- Assigning an action listener to a source not related to it will not cause an error but the listener method will never be called.
- It is recommended to use smart pointer to create an instance of pfc Action Listener class. The smart pointer can be destroyed by calling the `pfcActionSource::RemoveActionListener` method. The keyword `delete` is not required to delete the smart pointer.
- The method `pfcActionSource::AddActionListener` adds action listeners to notify you of all the events.
- The method `pfcActionSource::AddActionListenerWithType` adds action listeners to notify you about events that are specified in the input argument *ATypes*.

Types of Action Listeners

The following sections describe the different kinds of action listeners: session, UI command, solid, and feature.

Dimension Level Action Listeners

Methods Introduced:

- **wfcDimensionActionListener::OnBeforeDimensionValueModify**

The method `wfcDimensionActionListener::OnBeforeDimensionValueModify` is called before the dimension value is modified.

Session Level Action Listeners

Methods introduced:

- **pfcSessionActionListener::OnAfterDirectoryChange**
- **pfcSessionActionListener::OnAfterWindowChange**
- **pfcSessionActionListener::OnAfterModelDisplay**
- **pfcSessionActionListener::OnBeforeModelErase**
- **pfcSessionActionListener::OnBeforeModelDelete**
- **pfcSessionActionListener::OnBeforeModelRename**

- **pfcSessionActionListener::OnBeforeModelSave**
- **pfcSessionActionListener::OnBeforeModelPurge**
- **pfcSessionActionListener::OnBeforeModelCopy**
- **pfcSessionActionListener::OnAfterModelPurge**
- **wfcWSession::AddBrowserMessageListener**

The `pfcSessionActionListener::OnAfterDirectoryChange` method activates after the user changes the working directory. This method takes the new directory path as an argument.

The `pfcSessionActionListener::OnAfterWindowChange` method activates when the user activates a window other than the current one. Pass the new window to the method as an argument.

The `pfcSessionActionListener::OnAfterModelDisplay` method activates every time a model is displayed in a window.

Note

Model display events happen when windows are moved, opened and closed, repainted, or the model is regenerated. The event can occur more than once in succession.

The methods `pfcSessionActionListener::OnBeforeModelErase`, `pfcSessionActionListener::OnBeforeModelRename`, `pfcSessionActionListener::OnBeforeModelSave`, and `pfcSessionActionListener::OnBeforeModelCopy` take special arguments. They are designed to allow you to fill in the arguments and pass this data back to Creo Parametric. The model names placed in the descriptors will be used by Creo Parametric as the default names in the user interface.

The method `wfcWSession::AddBrowserMessageListener` adds a listener callback to receive browser message with specified key. The input arguments are:

- *key*—Unique key that should be specified in the Java Script message with the syntax as:

  ```
  window.external.ptc ('ToolkitJSBridge=' + key + '?' + message)
  ```

- *Listener*—Listener object.

UI Command Action Listeners

Methods introduced:

- **pfcSession::UICreateCommand**

- **pfcUICommandActionListener::OnCommand**

The `pfcSession::UICreateCommand` method takes a `pfcUICommandActionListener` argument and returns a `UICommand` action source with that action listener already registered. This `UICommand` object is subsequently passed as an argument to the `pfcSession::AddUIButton` method that adds a command button to a Creo Parametric menu. The `pfcUICommandActionListener::OnCommand` method of the registered `pfcUICommandActionListener` is called whenever the command button is clicked.

Model Level Action listeners

Methods introduced:

- **pfcModelActionListener::OnAfterModelSave**
- **pfcModelEventActionListener::OnAfterModelCopy**
- **pfcModelEventActionListener::OnAfterModelRename**
- **pfcModelEventActionListener::OnAfterModelErase**
- **pfcModelEventActionListener::OnAfterModelDelete**
- **pfcModelActionListener::OnAfterModelRetrieve**
- **pfcModelActionListener::OnBeforeModelDisplay**
- **pfcModelActionListener::OnAfterModelCreate**
- **pfcModelActionListener::OnAfterModelSaveAll**
- **pfcModelEventActionListener::OnAfterModelCopyAll**
- **pfcModelActionListener::OnAfterModelEraseAll**
- **pfcModelActionListener::OnAfterModelDeleteAll**
- **pfcModelActionListener::OnAfterModelRetrieveAll**
- **wfcWSession::AddBeforeModelRetrieveListener**
- **wfcBeforeModelRetrieveActionListener::OnBeforeModelRetrieve**
- **wfcBeforeModelRetrieveInstructions::Create**
- **wfcBeforeModelRetrieveInstructions::GetRetrieveOptions**
- **wfcBeforeModelRetrieveInstructions::SetRetrieveOptions**
- **wfcBeforeModelRetrieveInstructions::GetModelFilePath**
- **wfcBeforeModelRetrieveInstructions::SetModelFilePath**
- **wfcBeforeModelSaveAllListener::OnBeforeModelSave**
- **wfcModelParamActionListener::OnBeforeParameterCreate**
- **wfcModelParamActionListener::OnBeforeParameterModify**

- **wfcModelParamActionListener::OnAfterParameterModify**
- **wfcModelParamActionListener::OnAfterParameterDelete**
- **wfcModelAfterRenameAllActionListener::OnAfterModelRenameAll**
- **wfcModelReplaceActionListener::OnAfterModelReplace**

Methods ending in `All` are called after any event of the specified type. The call is made even if the user did not explicitly request that the action take place. Methods that do not end in `All` are only called when the user specifically requests that the event occurs.

The method `pfcModelActionListener::OnAfterModelSave` is called after successfully saving a model.

The method `pfcModelEventActionListener::OnAfterModelCopy` is called after successfully copying a model.

The method `pfcModelEventActionListener::OnAfterModelRename` is called after successfully renaming a model.

The method `pfcModelEventActionListener::OnAfterModelErase` is called after successfully erasing a model.

The method `pfcModelEventActionListener::OnAfterModelDelete` is called after successfully deleting a model.

The method `pfcModelActionListener::OnAfterModelRetrieve` is called after successfully retrieving a model.

The method `pfcModelActionListener::OnBeforeModelDisplay` is called before displaying a model.

The method `pfcModelActionListener::OnAfterModelCreate` is called after the successful creation of a model.

The method `wfcWSession::AddBeforeModelRetrieveListener` supersedes the method `wfcWSession::AddModelRetrievePreListener`. The method creates a listener. This listener blocks the standard Creo **File Open** dialog box.

The method `wfcBeforeModelRetrieveActionListener::OnBeforeModelRetrieve` supersedes the method `wfcBeforeModelRetrieveListener::OnBeforeModelRetrieve`. The method `wfcBeforeModelRetrieveActionListener::OnBeforeModelRetrieve` is called when you activate the **File ▸ Open** menu. This method contains its own code for handling the File open event. You must replace the standard **File Open** dialog box with the dialog box created by your application to open files. The method also retrieves the model specified in the object

wfcBeforeModelRetrieveInstructions. Use the method wfcBeforeModelRetrieveInstructions::Create to create an instance of wfcBeforeModelRetrieveInstructions object, which specifies the instructions for retrieving the model. The method wfcBeforeModelRetrieveInstructions::SetRetrieveOptions uses the enumerated data type wfcWModelRetrieveOption to specify how the model must be retrieved. Use the method wfcBeforeModelRetrieveInstructions::GetRetrieveOptions to get the type of option used to retrieve the model. The model can be retrieved using the following options:

- wfcMODEL_RETRIEVE_NORMAL—Retrieves the models normally.
- wfcMODEL_RETRIEVE_SIMP_REP—Retrieves the models as a simplified representation.
- wfcMODEL_RETRIEVE_VIEW_ONLY—Used for drawings only. Retrieves the model in view only mode.

Use the methods wfcBeforeModelRetrieveInstructions::GetModelFilePath and wfcBeforeModelRetrieveInstructions::SetModelFilePath to get and set the file path of the model that must be retrieved. The file path includes the path, file name, extension, and version of the model.

The method wfcBeforeModelSaveAllListener::OnBeforeModelSave is called before a model has been saved. This method provides more functionality than the existing method pfcSessionActionListener::OnBeforeModelSave. The method wfcBeforeModelSaveAllListener::OnBeforeModelSave is called on the current model and also its dependents. For example, before saving an assembly, the method is called on the assembly and also on all the assembly components. It is also called for various user actions such as, saving a copy of the model, checkin of a model, and so on. During conflict resolution, the method may be called more than once.

The methods in class wfcModelParamActionListener are triggered whenever these events occur for any parameter in the model.

The method wfcModelParamActionListener::OnBeforeParameterCreate is called before a parameter is created.

The method wfcModelParamActionListener::OnBeforeParameterModify is called before the parameter is modified.

The method wfcModelParamActionListener::OnAfterParameterModify is called after the parameter has been successfully modified.

The method `wfcModelParamActionListener::OnAfterParameterDelete` is called after the parameter has been deleted.

The method `wfcModelAfterRenameAllActionListener::OnAfterModelRenameAll` is called after all the models described using the `pfcModelDescriptor` object have been renamed.

The method `wfcModelReplaceActionListener::OnAfterModelReplace` is called after successfully replacing a model.

Solid Level Action Listeners

Methods introduced:

- **pfcSolidActionListener::OnBeforeRegen**
- **pfcSolidActionListener::OnAfterRegen**
- **pfcSolidActionListener::OnBeforeUnitConvert**
- **pfcSolidActionListener::OnAfterUnitConvert**
- **pfcSolidActionListener::OnBeforeFeatureCreate**
- **pfcSolidActionListener::OnAfterFeatureCreate**
- **pfcSolidActionListener::OnAfterFeatureDelete**

The `pfcSolidActionListener::OnBeforeRegen` and `pfcSolidActionListener::OnAfterRegen` methods occur when the user regenerates a solid object within the `pfcActionSource` to which the listener is assigned. These methods take the first feature to be regenerated and a handle to the `Solid` object as arguments. In addition, the method `pfcSolidActionListener::OnAfterRegen` includes a Boolean argument that indicates whether regeneration was successful.

Note

- It is not recommended to modify geometry or dimensions using the `pfcSolidActionListener::OnBeforeRegen` method call.
- A regeneration that did not take place because nothing was modified is identified as a regeneration failure.

The `pfcSolidActionListener::OnBeforeUnitConvert` and `pfcSolidActionListener::OnAfterUnitConvert` methods activate when a user modifies the unit scheme (by selecting the Creo Parametric command

Set Up, **Units**). The methods receive the `Solid` object to be converted and a Boolean flag that identifies whether the conversion changed the dimension values to keep the object the same size.

Note

`SolidActionListeners` can be registered with the session object so that its methods are called when these events occur for any solid model that is in session.

The `pfcSolidActionListener::OnBeforeFeatureCreate` method activates when the user starts to create a feature that requires the **Feature Creation** dialog box. Because this event occurs only after the dialog box is displayed, it will not occur at all for datums and other features that do not use this dialog box. This method takes two arguments: the solid model that will contain the feature and the `ModelItem` identifier.

The `pfcSolidActionListener::OnAfterFeatureCreate` method activates after any feature, including datums, has been created. This method takes the new `Feature` object as an argument.

The `pfcSolidActionListener::OnAfterFeatureDelete` method activates after any feature has been deleted. The method receives the solid that contained the feature and the (now defunct) `pfcModelItem` identifier.

Selection Level Action Listeners

Method introduced:

- **pfcSelectionBufferListener::OnAfterSelBufferChange**

The method **pfcSelectionBufferListener::OnAfterSelBufferChange** is called after the selection buffer is changed.

Feature Level Action Listeners

Methods introduced:

- **pfcFeatureActionListener::OnBeforeDelete**
- **pfcFeatureActionListener::OnBeforeSuppress**
- **pfcFeatureActionListener::OnAfterSuppress**
- **pfcFeatureActionListener::OnBeforeRegen**
- **pfcFeatureActionListener::OnAfterRegen**
- **pfcFeatureActionListener::OnRegenFailure**

- **pfcFeatureActionListener::OnBeforeRedefine**
- **pfcFeatureActionListener::OnAfterCopy**
- **pfcFeatureActionListener::OnBeforeParameterDelete**
- **wfcFeatureParamActionListener::OnBeforeParameterCreate**
- **wfcFeatureParamActionListener::OnBeforeParameterModify**
- **wfcFeatureParamActionListener::OnAfterParameterModify**
- **wfcFeatureParamActionListener::OnAfterParameterDelete**

Each method in `pfcFeatureActionListener` takes as an argument the feature that triggered the event.

`pfcFeatureActionListeners` can be registered with the object so that the action listener's methods are called whenever these events occur for any feature that is in session or with a solid model to react to changes only in that model.

The method `pfcFeatureActionListener::OnBeforeDelete` is called before a feature is deleted.

The method `pfcFeatureActionListener::OnBeforeSuppress` is called before a feature is suppressed.

The method `pfcFeatureActionListener::OnAfterSuppress` is called after a successful feature suppression.

The method `pfcFeatureActionListener::OnBeforeRegen` is called before a feature is regenerated.

The method `pfcFeatureActionListener::OnAfterRegen` is called after a successful feature regeneration.

The method `pfcFeatureActionListener::OnRegenFailure` is called when a feature fails regeneration.

The method `pfcFeatureActionListener::OnBeforeRedefine` is called before a feature is redefined.

The method `pfcFeatureActionListener::OnAfterCopy` is called after a feature has been successfully copied.

The method `pfcFeatureActionListener::OnBeforeParameterDelete` is called before a feature parameter is deleted.

The methods in `wfcFeatureParamActionListener` are triggered whenever these events occur for any parameter in the feature.

The method `wfcFeatureParamActionListener::OnBeforeParameterCreate` is called before a parameter is created.

The method `wfcFeatureParamActionListener::OnBeforeParameterModify` is called before the parameter is modified.

The method `wfcFeatureParamActionListener::OnAfterParameterModify` is called after the parameter has been successfully modified.

The method `wfcFeatureParamActionListener::OnAfterParameterDelete` is called after the parameter has been deleted.

Cancelling an ActionListener Operation

Creo Object TOOLKIT C++ allows you to cancel certain notification events, registered by the action listeners.

Methods Introduced:

- **pfcXCancelProEAction::Throw**

The static method `pfcXCancelProEAction::Throw` must be called from the body of an action listener to cancel the impending Creo Parametric operation. This method will throw a Creo Object TOOLKIT C++ exception signalling to Creo Parametric to cancel the listener event.

Note: Your application should not catch the Creo Object TOOLKIT C++ exception, or should rethrow it if caught, so that Creo Parametric is forced to handle it.

The following events can be cancelled using this technique:

- `pfcSessionActionListener::OnBeforeModelErase`
- `pfcSessionActionListener::OnBeforeModelDelete`
- `pfcSessionActionListener::OnBeforeModelRename`
- `pfcSessionActionListener::OnBeforeModelSave`
- `pfcSessionActionListener::OnBeforeModelPurge`
- `pfcSessionActionListener::OnBeforeModelCopy`
- `pfcModelActionListener::OnBeforeParameterCreate`
- `pfcModelActionListener::OnBeforeParameterDelete`
- `pfcModelActionListener::OnBeforeParameterModify`
- `pfcFeatureActionListener::OnBeforeDelete`
- `pfcFeatureActionListener::OnBeforeSuppress`
- `pfcFeatureActionListener::OnBeforeParameterDelete`
- `pfcFeatureActionListener::OnBeforeParameterCreate`

- `pfcFeatureActionListener::OnBeforeRedefine`

31

Interface

This chapter describes various methods of importing and exporting files in Creo Object TOOLKIT C++.

Exporting Files and 2D Models

Method Introduced:

- **pfcModel::Export**

The method `pfcModel::Export` exports model data to a file. The exported files are placed in the current Creo working directory. The input parameters are:

- *filename*—Output file name including extensions
- *exportdata*—The `pfcExportInstructions` object that controls the export operation. The type of data that is exported is given by the `pfcExportType` object.

There are four general categories of files to which you can export models:

- File types whose instructions inherit from `pfcGeomExportInstructions`.

  These instructions export files that contain precise geometric information used by other CAD systems.
- File types whose instructions inherit from `pfcCoordSysExportInstructions`.

  These instructions export files that contain coordinate information describing faceted, solid models (without datums and surfaces).
- File types whose instructions inherit from `pfcFeatIdExportInstructions`.

  These instructions export information about a specific feature.
- General file types that inherit only from `pfcExportInstructions`.

  These instructions provide conversions to file types such as BOM (bill of materials).

For information on exporting to a specific format, see the Creo Object TOOLKIT C++ APIWizard and theCreo Help.

Export Instructions

Methods Introduced:

- **pfcRelationExportInstructions::Create**
- **pfcModelInfoExportInstructions::Create**
- **pfcProgramExportInstructions::Create**
- **pfcIGESFileExportInstructions::Create**
- **pfcDXFExportInstructions::Create**

- **pfcRenderExportInstructions::Create**
- **pfcSTLASCIIExportInstructions::Create**
- **pfcSTLBinaryExportInstructions::Create**
- **pfcBOMExportInstructions::Create**
- **pfcDWGSetupExportInstructions::Create**
- **pfcFeatInfoExportInstructions::Create**
- **pfcMFGFeatCLExportInstructions::Create**
- **pfcMFGOperCLExportInstructions::Create**
- **pfcMaterialExportInstructions::Create**
- **pfcCGMFILEExportInstructions::Create**
- **pfcInventorExportInstructions::Create**
- **pfcFIATExportInstructions::Create**
- **pfcConnectorParamExportInstructions::Create**
- **pfcCableParamsFileInstructions::Create**
- **pfcCATIAFacetsExportInstructions::Create**
- **pfcVRMLModelExportInstructions::Create**
- **pfcSTEP2DExportInstructions::Create**
- **pfcMedusaExportInstructions::Create**
- **pfcCADDSExportInstructions::Create**
- **pfcSliceExportData::Create**
- **pfcNEUTRALFileExportInstructions::Create**
- **pfcProductViewExportInstructions::Create**
- **pfcBaseSession::ExportDirectVRML**

Export Instructions Table

Interface	Used to Export
`pfcRelationExportInstructions`	A list of the relations and parameters in a part or assembly
`pfcModelInfoExportInstructions`	Information about a model, including units information, features, and children
`pfcProgramExportInstructions`	A program file for a part or assembly that can be edited to change the model
`pfcIGESExportInstructions`	A drawing in `IGES` format
`pfcDXFExportInstructions`	A drawing in `DXF` format
`pfcRenderExportInstructions`	A part or assembly in `RENDER` format
`pfcSTLASCIIExportInstructions`	A part or assembly to an `ASCII STL` file
`pfcSTLBinaryExportInstructions`	A part or assembly in a binary `STL` file

Interface	Used to Export
pfcBOMExportInstructions	A BOM for an assembly
pfcDWGSetupExportInstructions	A drawing setup file
pfcFeatInfoExportInstructions	Information about one feature in a part or assembly
pfcMfgFeatCLExportInstructions	A cutter location (CL) file for one Creo NC sequence in a manufacturing assembly
pfcMfgOperClExportInstructions	A cutter location (CL) file for all the Creo NC sequences in a manufacturing assembly
pfcMaterialExportInstructions	A material from a part
pfcCGMFILEExportInstructions	A drawing in CGM format
pfcInventorExportInstructions	A part or assembly in Inventor format
pfcFIATExportInstructions	A part or assembly in FIAT format
pfcConnectorParamExportInstructions	The parameters of a connector to a text file
pfcCableParamsFileInstructions	Cable parameters from an assembly
CATIAFacetsExportInstructions	A part or assembly in CATIA format (as a faceted model)
pfcVRMLModelExportInstructions	A part or assembly in VRML format
pfcSTEP2DExportInstructions	A two-dimensional STEP format file
pfcMedusaExportInstructions	A drawing in MEDUSA file
pfcCADDSExportInstructions	A CADDS5 solid model
pfcNEUTRALFileExportInstructions	A Creo part to neutral format
pfcProductViewExportInstructions	A part, assembly, or drawing in Creo Viewformat
pfcSliceExportData	A slice export format

Exporting Drawing Sheets

Methods Introduced:

- **pfcDXFExportInstructions::GetOptionValue**
- **pfcDXFExportInstructions::SetOptionValue**
- **pfcExport2DOption::Create**
- **pfcExport2DOption::SetExportSheetOption**
- **pfcExport2DOption::SetModelSpaceSheet**
- **pfcExport2DOption::SetSheets**

When you export a drawing to DXF format, use the methods `pfcDXFExportInstructions::GetOptionValue` and `pfcDXFExportInstructions::SetOptionValue` to get and set the options that are required to export multiple sheets.

The options required to export multiple sheets of a drawing are specified using the `pfcExport2DOption` object.

The method `pfcExport2DOptions::Create` creates a new instance of the `pfcExport2DOption` object. This object contains the following options:

- *pfcExportSheetOption*—Specifies the option for exporting multiple drawing sheets. Use the method `pfcExport2DOption::SetExportSheetOption` to set the option for exporting multiple drawing sheets. The options are given by the `pfcExport2DSheetOption` class and can be of the following types:
  - `pfcEXPORT_CURRENT_TO_MODEL_SPACE`—Exports only the drawing's current sheet as model space to a single file. This is the default type.
  - `pfcEXPORT_CURRENT_TO_PAPER_SPACE`—Exports only the drawing's current sheet as paper space to a single file. This type is the same as `pfcEXPORT_CURRENT_TO_MODEL_SPACE` for formats that do not support the concept of model space and paper space.
  - `pfcEXPORT_ALL`—Exports all the sheets in a drawing to a single file as paper space, if applicable for the format type.
  - `pfcEXPORT_SELECTED`—Exports selected sheets in a drawing as paper space and one sheet as model space.
- *pfcModelSpaceSheet*—Specifies the sheet number that needs be exported as model space. This option is applicable only if the export formats support the concept of model space and paper space and if *pfcExportSheetOption* is set to `pfcEXPORT_SELECTED`. Use the method `pfcExport2DOption::SetModelSpaceSheet` to set this option.
- *Sheets*—Specifies the sheet numbers that need to be exported as paper space. This option is applicable only if *pfcExportSheetOption* is set to `pfcEXPORT_SELECTED`. Use the method `pfcExport2DOption::SetSheets` to set this option.

Exporting to Faceted Formats

The methods described in this section support the export of Creo drawings and solid models to faceted formats like `CATIA CGR`.

Methods Introduced:

- **pfcTriangulationInstructions::GetAngleControl**
- **pfcTriangulationInstructions::SetAngleControl**
- **pfcTriangulationInstructions::GetChordHeight**
- **pfcTriangulationInstructions::SetChordHeight**
- **pfcTriangulationInstructions::GetStepSize**
- **pfcTriangulationInstructions::SetStepSize**
- **pfcTriangulationInstructions::GetFacetControlOptions**

- **pfcTriangulationInstructions::SetFacetControlOptions**

The methods `pfcTriangulationInstructions::GetAngleControl` and `pfcTriangulationInstructions::SetAngleControl` gets and sets the angle control for the exported facet drawings and models. You can set the value between 0.0 to 1.0.

Use the methods `pfcTriangulationInstructions::GetChordHeight` and `pfcTriangulationInstructions::SetChordHeight` to get and set the chord height for the exported facet drawings and models.

The methods `pfcTriangulationInstructions::GetStepSize` and `pfcTriangulationInstructions::SetStepSize` allow you to control the step size for the exported files. The default value is 0.0.

Note

You must pass the value of Step Size as `NULL`, if you specify the Quality value.

The methods `pfcTriangulationInstructions::GetFacetControlOptions` and `pfcTriangulationInstructions::SetFacetControlOptions` get and set the flags that control the facet export options. You can set the bit flags using the `pfcFacetControlFlag` object. It has the following values:

- `pfcFACET_STEP_SIZE_ADJUST`—Adjusts the step size according to the component size.
- `pfcFACET_CHORD_HEIGHT_ADJUST`—Adjusts the chord height according to the component size.
- `pfcFACET_USE_CONFIG`—CONFIG—If this flag is set, values of the flags `pfcFACET_STEP_SIZE_OFF`, `pfcFACET_STEP_SIZE_ADJUST`, and `pfcFACET_CHORD_HEIGHT_ADJUST` are ignored and the configuration settings from the Creo user interface are used during the export operation.
- `pfcFACET_CHORD_HEIGHT_DEFAULT`—Uses the default value set in the Creo user interface for the chord height.
- `pfcFACET_ANGLE_CONTROL_DEFAULT`—Uses the default value set in the Creo user interface for the angle control.
- `pfcFACET_STEP_SIZE_DEFAULT`—Uses the default value set in the Creo user interface for the step size.
- `pfcFACET_STEP_SIZE_OFF`—Switches off the step size control.
- `pfcFACET_FORCE_INTO_RANGE`—Forces the out-of-range parameters into range. If any of the `pfcFACET_*_DEFAULT` option is set, then the option `pfcFACET_FORCE_INTO_RANGE` is not applied on that parameter.

- `pfcFACET_STEP_SIZE_FACET_INCLUDE_QUILTS`—Includes quilts in the export of Creo model to the specified format.
- `pfcEXPORT_INCLUDE_ANNOTATIONS`—Includes annotations in the export of Creo model to the specified format.

> **Note**
>
> To include annotations, during the export of Creo model, you must call the method `pfcModel::Display` before calling `pfcModel::Export`.

Exporting Using Coordinate System

The methods described in this section support the export of files with information about the faceted solid models (without datums and surfaces). The files are exported in reference to the coordinate-system feature in the model being exported.

Methods Introduced:

- **pfcCoordSysExportInstructions::GetCsysName**
- **pfcCoordSysExportInstructions::SetCsysName**
- **pfcCoordSysExportInstructions::GetQuality**
- **pfcCoordSysExportInstructions::SetQuality**
- **pfcCoordSysExportInstructions::GetMaxChordHeight**
- **pfcCoordSysExportInstructions::SetMaxChordHeight**
- **pfcCoordSysExportInstructions::GetAngleControl**
- **pfcCoordSysExportInstructions::SetAngleControl**
- **pfcCoordSysExportInstructions::GetSliceExportData**
- **pfcCoordSysExportInstructions::SetSliceExportData**
- **pfcCoordSysExportInstructions::GetStepSize**
- **pfcCoordSysExportInstructions::SetStepSize**
- **pfcCoordSysExportInstructions::GetFacetControlOptions**
- **pfcCoordSysExportInstructions::SetFacetControlOptions**
- **pfcSelection::SetIntf3DCsys()**

The method `pfcCoordSysExportInstructions::GetCsysName` returns the name of the name of a coordinate system feature in the model being exported. It is recommended to use the coordinate system that places the part or assembly in its upper-right quadrant, so that all position and distance values of the

exported assembly or part are positive. The method `pfcCoordSysExportInstructions::SetCsysName` allows you to set the coordinate system feature name.

The methods `pfcCoordSysExportInstructions::GetQuality` and `pfcCoordSysExportInstructions::SetQuality` can be used instead of `pfcCoordSysExportInstructions::GetMaxChordHeight` and `pfcCoordSysExportInstructions::GetMaxChordHeight` and `pfcCoordSysExportInstructions::GetAngleControl` and `pfcCoordSysExportInstructions::SetAngleControl`. You can set the value between 1 and 10. The higher the value you pass, the lower is the Maximum Chord Height setting and higher is the Angle Control setting the method uses. The default Quality value is 1.0.

Note

You must pass the value of Quality as `NULL`, if you use Maximum Chord Height and Angle Control values. If Quality, Maximum Chord Height, and Angle Control are all `NULL`, then the Quality setting of 3 is used.

Use the methods `pfcCoordSysExportInstructions::GetMaxChordHeight` and `pfcCoordSysExportInstructions::SetMaxChordHeight` to work with the maximum chord height for the exported files. The default value is 0.1.

Note

You must pass the value of Maximum Chord Height as `NULL`, if you specify the Quality value.

The methods `pfcCoordSysExportInstructions::GetAngleControl` and `pfcCoordSysExportInstructions::SetAngleControl` allow you to work with the angle control setting for the exported files. The default value is 0.1.

Note

You must pass the value of Angle Control as `NULL`, if you specify the Quality value.

The methods `pfcCoordSysExportInstructions::GetSliceExportData` and `pfcCoordSysExportInstructions::SetSliceExportData` get and set the `pfcModelSliceExportData` data object that specifies data for the slice export. The options in this object are described as follows:

- *CompIds*—Specifies the sequence of integers that identify the components that form the path from the root assembly down to the component part or assembly being referred to. Use the methods `pfcSliceExportData::GetCompIds` and `pfcSliceExportData::SetCompIds` to work with the component IDs.

The methods `pfcCoordSysExportInstructions::GetStepSize` and `pfcCoordSysExportInstructions::SetStepSize` allow you to control the step size for the exported files. The default value is 0.0.

Note

You must pass the value of Step Size as `NULL`, if you specify the Quality value.

The methods `pfcCoordSysExportInstructions::GetFacetControlOptions` and `pfcCoordSysExportInstructions::SetFacetControlOptions` get and set the flags that control the facet export options using the `pfcFacetControlFlag` object. For more information on the bit flag values, please refer to the section Exporting to Faceted Formats on page 526

The method `pfcSelection::SetIntf3DCsys()` sets the reference coordinate system for the export. The input argument *ReferenceCsys* is the reference coordinate system selection. Call this method without any argument to set default coordinate system. Reference coordinate system is not supported for CADDS and NEUTRAL file types.

Exporting to PDF and U3D

The methods described in this section support the export of Creo drawings and solid models to Portable Document Format (PDF) and U3D format. You can export a drawing or a 2D model as a 2D raster image embedded in a `PDF` file. You can export Creo solid models in the following ways:

- As a `U3D` model embedded in a one-page `PDF` file
- As 2D raster images embedded in the pages of a `PDF` file representing saved views
- As a standalone `U3D` file

While exporting multiple sheets of a Creo drawing to a PDF file, you can choose to export all sheets, the current sheet, or selected sheets.

These methods also allow you to insert a variety of non-geometric information to improve document content, navigation, and search.

Methods Introduced:

- **pfcPDFExportInstructions::Create**
- **pfcPDFExportInstructions::GetOptions**
- **pfcPDFExportInstructions::SetOptions**
- **pfcPDFExportInstructions::GetFilePath**
- **pfcPDFExportInstructions::SetFilePath**
- **pfcPDFExportInstructions::GetProfilePath**
- **pfcPDFExportInstructions::SetProfilePath**
- **pfcPDFOption::Create**
- **pfcPDFOption::SetOptionType**
- **pfcPDFOption::SetOptionValue**

The method pfcPDFExportInstructions::Create creates a new instance of the pfcPDFExportInstructions data object that describes how to exportCreo drawings or solid models to the PDF and U3D formats. The options in this object are described as follows:

- *FilePath*—Specifies the name of the output file. Use the method pfcPDFExportInstructions::SetFilePath to set the name of the output file.
- *Options*—Specifies a collection of PDF export options of the type pfcPDFOption. Create a new instance of this object using the method pfcPDFOption::Create. This object contains the following attributes:
  - *OptionType*—Specifies the type of option in terms of the pfcPDFOptionType class. Set this option using the method pfcPDFOption::SetOptionType.
  - *OptionValue*—Specifies the value of the option in terms of the pfcArgValue object. Set this option using the method pfcPDFOption::SetOptionValue.

  Use the method pfcPDFExportInstructions::SetOptions to set the collection of PDF export options.
- *ProfilePath*—Specifies the export profile path. Use the method pfcPDFExportInstructions::SetProfilePath to set the profile path. When you set the profile path, the PDF export options set in the object

pfcPDFExportInstructions are ignored when the method pfcModel::Export is called. You can set the profile path as NULL.

Note

You can specify the profile path only for drawings.

The types of options (given by the pfcPDFOptionType class) available for export to PDF and U3D formats are described as follows:

- pfcPDFOPT_FONT_STROKE—Allows you to switch between using TrueType fonts or "stroking" text in the resulting document. This option is given by the pfcPDFFontStrokeMode class and takes the following values:
  - pfcPDF_USE_TRUE_TYPE_FONTS—Specifies TrueType fonts. This is the default type.
  - pfcPDF_STROKE_ALL_FONTS—Specifies the option to stroke all fonts.
- pfcPDFOPT_COLOR_DEPTH—Allows you to choose between color, grayscale, or monochrome output. This option is given by the pfcPDFColorDepth class and takes the following values:
  - pfcPDF_CD_COLOR—Specifies color output. This is the default value.
  - pfcPDF_CD_GRAY—Specifies grayscale output.
  - pfcPDF_CD_MONO—Specifies monochrome output.
- pfcPDFOPT_HIDDENLINE_MODE—Enables you to set the style for hidden lines in the resulting PDF document. This option is given by the pfcPDFHiddenLineMode class and takes the following values:
  - pfcPDF_HLM_SOLID—Specifies solid hidden lines.
  - pfcPDF_HLM_DASHED—Specifies dashed hidden lines. This is the default type.
- pfcPDFOPT_SEARCHABLE_TEXT—If true, stroked text is searchable. The default value is true.
- pfcPDFOPT_RASTER_DPI—Allows you to set the resolution for the output of any shaded views in DPI. It can take a value between 100 and 600. The default value is 300.
- pfcPDFOPT_LAUNCH_VIEWER—If true, launches the Adobe Acrobat Reader. The default value is true.

- pfcPDFOPT_LAYER_MODE—Enables you to set the availability of layers in the document. It is given by the pfcPDFLayerMode class and takes the following values:
  - pfcPDF_LAYERS_ALL—Exports the visible layers and entities. This is the default.
  - pfcPDF_LAYERS_VISIBLE—Exports only visible layers in a drawing.
  - pfcPDF_LAYERS_NONE—Exports only the visible entities in the drawing, but not the layers on which they are placed.
- pfcPDFOPT_PARAM_MODE—Enables you to set the availability of model parameters as searchable metadata in the PDF document. It is given by the pfcPDFParameterMode class and takes the following values:
  - pfcPDF_PARAMS_ALL—Exports the drawing and the model parameters to PDF. This is the default.
  - pfcPDF_PARAMS_DESIGNATED—Exports only the specified model parameters in the PDF metadata.
  - pfcPDF_PARAMS_NONE—Exports the drawing to PDF without the model parameters.
- pfcPDFOPT_HYPERLINKS—Sets hyperlinks to be exported as label text only or sets the underlying hyperlink URLs as active. The default value is true, specifying that the hyperlinks are active.
- pfcPDFOPT_BOOKMARK_ZONES—If true, adds bookmarks to the PDF showing zoomed in regions or zones in the drawing sheet. The zone on an A4-size drawing sheet is ignored.
- pfcPDFOPT_BOOKMARK_VIEWS—If true, adds bookmarks to the PDF document showing zoomed in views on the drawing.
- pfcPDFOPT_BOOKMARK_SHEETS—If true, adds bookmarks to the PDF document showing each of the drawing sheets.
- pfcPDFOPT_BOOKMARK_FLAG_NOTES—If true, adds bookmarks to the PDF document showing the text of the flag note.
- pfcPDFOPT_TITLE—Specifies a title for the PDF document.
- pfcPDFOPT_AUTHOR—Specifies the name of the person generating the PDF document.
- pfcPDFOPT_SUBJECT—Specifies the subject of the PDF document.
- pfcPDFOPT_KEYWORDS—Specifies relevant keywords in the PDF document.

- pfcPDFOPT_PASSWORD_TO_OPEN—Sets a password to open the PDF document. By default, this option is NULL, which means anyone can open the PDF document without a password.
- pfcPDFOPT_MASTER_PASSWORD—Sets a password to restrict or limit the operations that the viewer can perform on the opened PDF document. By default, this option is NULL, which means you can make any changes to the PDF document regardless of the settings of the modification flags pfcPDFOPT_ALLOW_*.
- pfcPDFOPT_RESTRICT_OPERATIONS—If true, enables you to restrict or limit operations on the PDF document. By default, is is false.
- pfcPDFOPT_ALLOW_MODE—Enables you to set the security settings for the PDF document. This option must be set if pfcPDFOPT_RESTRICT_OPERATIONS is set to true. It is given by the pfcPDFRestrictOperationsMode class and takes the following values:
  - pfcPDF_RESTRICT_NONE—Specifies that the user can perform any of the permitted viewer operations on the PDF document. This is the default value.
  - pfcPDF_RESTRICT_FORMS_SIGNING—Restricts the user from adding digital signatures to the PDF document.
  - pfcPDF_RESTRICT_INSERT_DELETE_ROTATE—Restricts the user from inserting, deleting, or rotating the pages in the PDF document.
  - pfcPDF_RESTRICT_COMMENT_FORM_SIGNING—Restricts the user from adding or editing comments in the PDF document.
  - pfcPDF_RESTRICT_EXTRACTING—Restricts the user from extracting pages from the PDF document.
- pfcPDFOPT_ALLOW_PRINTING—If true, allows you to print the PDF document. By default, it is true.
- pfcPDFOPT_ALLOW_PRINTING_MODE—Enables you to set the print resolution. It is given by the pfcPDFPrintingMode class and takes the following values:
  - pfcPDF_PRINTING_LOW_RES—Specifies low resolution for printing.
  - pfcPDF_PRINTING_HIGH_RES—Specifies high resolution for printing. This is the default value.
- pfcPDFOPT_ALLOW_COPYING—If true, allows you to copy content from the PDF document. By default, it is true.
- pfcPDFOPT_ALLOW_ACCESSIBILITY—If true, enables visually-impaired screen reader devices to extract data independent of the value given

by the pfcPDFRestrictOperationsMode class. The default value is true.

- pfcPDFOPT_PENTABLE—If true, uses the standard Creo pentable to control the line weight, line style, and line color of the exported geometry. The default value is false.
- pfcPDFOPT_LINECAP—Enables you to control the treatment of the ends of the geometry lines exported to PDF. It is given by the pfcPDFLinecap class and takes the following values:
  - pfcPDF_LINECAP_BUTT—Specifies the butt cap square end. This is the default value.
  - pfcPDF_LINECAP_ROUND—Specifies the round cap end.
  - pfcPDF_LINECAP_PROJECTING_SQUARE—Specifies the projecting square cap end.
- pfcPDFOPT_LINEJOIN—Enables you to control the treatment of the joined corners of connected lines exported to PDF. It is given by the pfcPDFLinejoin class and takes the following values:
  - pfcPDF_LINEJOIN_MITER—Specifies the miter join. This is the default.
  - pfcPDF_LINEJOIN_ROUND—Specifies the round join.
  - pfcPDF_LINEJOIN_BEVEL—Specifies the bevel join.
- pfcPDFOPT_SHEETS—Allows you to specify the sheets from a Creo drawing that are to be exported to PDF. It is given by the pfcPrintSheets class and takes the following values:
  - pfcPRINT_CURRENT_SHEET—Only the current sheet is exported to PDF
  - pfcPRINT_ALL_SHEETS—All the sheets are exported to PDF. This is the default value.
  - pfcPRINT_SELECTED_SHEETS—Sheets of a specified range are exported to PDF. If this value is assigned, then the value of the option pfcPDFOPT_SHEET_RANGE must also be known.
- pfcPDFOPT_SHEET_RANGE—Specifies the range of sheets in a drawing that are to be exported to PDF. If this option is set, then the option pfcPDFOPT_SHEETS must be set to the value pfcPRINT_SELECTED_SHEETS.
- pfcPDFOPT_EXPORT_MODE—Enables you to select the object to be exported to PDF and the export format. It is given by the pfcPDFExportMode class and takes the following values:

  - pfcPDF_2D_DRAWING—Only drawings are exported to PDF. This is the default value.
  - pfcPDF_3D_AS_NAMED_VIEWS—3D models are exported as 2D raster images embedded in PDF files.
  - pfcPDF_3D_AS_U3D_PDF—3D models are exported as U3D models embedded in one-page PDF files.
  - pfcPDF_3D_AS_U3D—A 3D model is exported as a U3D (.u3d) file. This value ignores the options set for the pfcPDFOptionType class.
- pfcPDFOPT_LIGHT_DEFAULT—Enables you to set the default lighting style used while exporting 3D models in the U3D format to a one-page PDF file, that is when the option pfcPDFOPT_EXPORT_MODE is set to pfcPDF_3D_AS_U3D. The values for this option are given by the pfcPDFU3DLightingMode class.
- pfcPDFOPT_RENDER_STYLE_DEFAULT—Enables you to set the default rendering style used while exporting Creo models in the U3D format to a one-page PDF file, that is when the option pfcPDFOPT_EXPORT_MODE is set to pfcPDF_3D_AS_U3D. The values for this option are given by the pfcPDFU3DRenderMode class.
- pfcPDFOPT_SIZE—Allows you to specify the page size of the exported PDF file. The values for this option are given by the pfcPlotPaperSize class. If the value is set to VARIABLESIZEPLOT, you also need to set the options pfcPDFOPT_HEIGHT and pfcPDFOPT_WIDTH.
- pfcPDFOPT_HEIGHT—Enables you to set the height for a user-defined page size of the exported PDF file. The default value is 0.0.
- pfcPDFOPT_WIDTH—Enables you to set the width for a user-defined page size of the exported PDF file. The default value is 0.0.
- pfcPDFOPT_ORIENTATION—Enables you to specify the orientation of the pages in the exported PDF file. It is given by the pfcSheetOrientation class.
  - pfcORIENT_PORTRAIT—Exports the pages in portrait orientation. This is the default value.
  - pfcORIENT_LANDSCAPE—Exports the pages in landscape orientation.
- pfcPDFOPT_TOP_MARGIN—Allows you to specify the top margin of the view port. The default value is 0.0.
- pfcPDFOPT_LEFT_MARGIN—Allows you to specify the left margin of the view port. The default value is 0.0.

- pfcPDFOPT_BACKGROUND_COLOR_RED—Specifies the default red background color that appears behind the U3D model. You can set any value within the range of 0.0 to 1.0. The default value is 1.0.
- pfcPDFOPT_BACKGROUND_COLOR_GREEN—Specifies the default green background color that appears behind the U3D model. You can set any value within the range of 0.0 to 1.0. The default value is 1.0.
- pfcPDFOPT_BACKGROUND_COLOR_BLUE—Specifies the default blue background color that appears behind the U3D model. You can set any value within the range of 0.0 to 1.0. The default value is 1.0.
- pfcPDFOPT_ADD_VIEWS—If true, allows you to add view definitions to the U3D model from a file. By default, it is true.
- pfcPDFOPT_VIEW_TO_EXPORT—Specifies the view or views to be exported to the PDF file. It is given by the pfcPDFSelectedViewMode class and takes the following values:
  - pfcPDF_VIEW_SELECT_CURRENT—Exports the current graphical area to a one-page PDF file.
  - pfcPDF_VIEW_SELECT_ALL—Exports all the views to a multi-page PDF file. Each page contains one view with the view name displayed at the bottom center of the view port.
  - pfcPDF_VIEW_SELECT_BY_NAME—Exports the selected view to a one-page PDF file with the view name printed at the bottom center of the view port. If this value is assigned, then the option pfcPDFOPT_SELECTED_VIEW must also be set.
- pfcPDFOPT_SELECTED_VIEW—Sets the option pfcPDFOPT_VIEW_TO_EXPORT to the value pfcPDF_VIEW_SELECT_BY_NAME, if the corresponding view is successfully found.
- pfcPDFOPT_PDF_SAVE—Specifies the PDF save options. It is given by the pfcPDFSaveMode class and takes the following values:
  - pfcPDF_ARCHIVE_1—Applicable only for the value pfcPDF_2D_DRAWING. Saves the drawings as PDF with the following conditions:
    - The value of pfcPDFLayerMode is set to pfcPDF_LAYERS_NONE.
    - The value of pfcPDFOPT_HYPERLINKS is set to FALSE.
    - The shaded views in the drawings will not have transparency and may overlap other data in the PDF.
    - The value of pfcPDFOPT_PASSWORD_TO_OPEN is set to NULL.
    - The value of pfcPDFOPT_MASTER_PASSWORD is set to NULL.

- `pfcPDF_FULL`—Saves the PDF with the values set by you. This is the default value.

Exporting 3D Geometry

Creo Object TOOLKIT C++ allows you to export three dimensional geometry to various formats.

Export Instructions

Methods Introduced:

- **pfcModel::ExportIntf3D**
- **pfcBaseSession::ExportProfileLoad**
- **pfcGeometryFlags::Create**
- **pfcInclusionFlags::Create**
- **pfcLayerExportOptions::Create**
- **pfcTriangulationInstructions::Create**

From Creo Parametric 5.0 F000 onward, the following interfaces along with their methods have been deprecated. Use the method `pfcModel::ExportIntf3D` instead to export Creo Parametric models to other file formats. All the options that can be set with these interfaces and methods, can also be set using the export profile option in Creo Parametric. Refer to the Creo Parametric Data Exchange Online Help for more information.

- `pfcExport3DInstructions`
- `pfcACIS3DExportInstructions`
- `pfcCATIAModel3DExportInstructions`
- `pfcCATIASession3DExportInstructions`
- `pfcCatiaPart3DExportInstructions`
- `pfcCatiaProduct3DExportInstructions`
- `pfcCatiaCGR3DExportInstructions`
- `pfcDXF3DExportInstructions`
- `pfcDWG3DExportInstructions`
- `pfcIGES3DNewExportInstructions`
- `pfcJT3DExportInstructions`
- `pfcParaSolid3DExportInstructions`
- `pfcSTEP3DExportInstructions`
- `pfcSWPart3DExportInstructions`

- `pfcSWAsm3DExportInstructions`
- `pfcUG3DExportInstructions`
- `pfcVDA3DExportInstructions`

The method `pfcModel::ExportIntf3D` exports a Creo Parametric model to the specified output format using the default export profile. The export options must be set using the export profile option in Creo Parametric.

The method `pfcBaseSession::ExportProfileLoad` loads the specified profile for export. You can use this function when you want to use the export profile of your choice instead of the default export profile in a particular Creo Parametric session. The input argument *ProfileFile* is the full path to the profile along with the profile name and extension.

Note

Once the export profile file is loaded in a Creo Parametric session, it will be active in the interactive mode as well.

The method `pfcTriangulationInstructions::Create` creates a object that will be used to define the parameters for faceted exports.

Export Utilities

Methods Introduced:

- **pfcBaseSession::IsConfigurationSupported**
- **pfcBaseSession::IsGeometryRepSupported**

The method `pfcBaseSession::IsConfigurationSupported` checks whether the specified assembly configuration is valid for a particular model and the specified export format. The input parameters for this method are:

- *Configuration*—Specifies the structure and content of the output files.
- *Type*—Specifies the output file type to create.

The method returns a true value if the configuration is supported for the specified export type.

The method `pfcBaseSession::IsGeometryRepSupported` checks whether the specified geometric representation is valid for a particular export format. The input parameters are :

- *Flags*—The type of geometry supported by the export operation.
- *Type*—The output file type to create.

The method returns a true value if the geometry combination is valid for the specified model and export type.

The methods `pfcBaseSession::IsConfigurationSupported` and `pfcBaseSession::IsGeometryRepSupported` must be called before exporting an assembly to the specified export formats except for the `CADDS` and `STEP2D` formats. The return values of both the methods must be true for the export operation to be successful.

Use the method`pfcModel::Export` to export the assembly to the specified output format.

Shrinkwrap Export

To improve performance in a large assembly design, you can export lightweight representations of models called shrinkwrap models. A shrinkwrap model is based on the external surfaces of the source part or asssembly model and captures the outer shape of the source model.

You can create the following types of nonassociative exported shrinkwrap models:

- Surface Subset—This type consists of a subset of the original model's surfaces.
- Faceted Solid—This type is a faceted solid representing the original solid.
- Merged Solid—The external components from the reference assembly model are merged into a single part representing the solid geometry in all collected components.

Methods Introduced:

- **pfcSolid::ExportShrinkwrap**

You can export the specified solid model as a shrinkwrap model using the method `pfcSolid::ExportShrinkwrap`. This method takes the `pfcShrinkwrapExportInstructions` object as an argument.

Use the appropriate interface given in the following table to create the required type of shrinkwrap. All the interfaces have their own static method to create an object of the specified type. The object created by these interfaces can be used as an object of type `pfcShrinkwrapExportInstructions` or `pfcShrinkwrapModelExportInstructions`.

Type of Shrinkwrap Model	Interface to Use
Surface Subset	`pfcShrinkwrapSurfaceSubset Instructions`
Faceted Part	`pfcShrinkwrapFacetedPartInstruc tions`
Faceted VRML	`pfcShrinkwrapFacetedVRMLInstruc tions`

Type of Shrinkwrap Model	Interface to Use
Faceted STL	`pfcShrinkwrapFacetedSTLInstructions`
Merged Solid	`pfcShrinkwrapMergedSolidInstructions`

Setting Shrinkwrap Options

The interface `pfcShrinkwrapModelExportInstructions` contains the general methods available for all the types of shrinkwrap models. The object created by any of the interfaces specified in the preceeding table can be used with these methods.

Methods Introduced:

- **pfcShrinkwrapModelExportInstructions::GetMethod**
- **pfcShrinkwrapModelExportInstructions::GetQuality**
- **pfcShrinkwrapModelExportInstructions::SetQuality**
- **pfcShrinkwrapModelExportInstructions::GetAutoHoleFilling**
- **pfcShrinkwrapModelExportInstructions::SetAutoHoleFilling**
- **pfcShrinkwrapModelExportInstructions::GetIgnoreSkeleton**
- **pfcShrinkwrapModelExportInstructions::SetIgnoreSkeleton**
- **pfcShrinkwrapModelExportInstructions::GetIgnoreQuilts**
- **pfcShrinkwrapModelExportInstructions::SetIgnoreQuilts**
- **pfcShrinkwrapModelExportInstructions::GetAssignMassProperties**
- **pfcShrinkwrapModelExportInstructions::SetAssignMassProperties**
- **pfcShrinkwrapModelExportInstructions::GetIgnoreSmallSurfaces**
- **pfcShrinkwrapModelExportInstructions::SetIgnoreSmallSurfaces**
- **pfcShrinkwrapModelExportInstructions::GetSmallSurfPercentage**
- **pfcShrinkwrapModelExportInstructions::SetSmallSurfPercentage**
- **pfcShrinkwrapModelExportInstructions::GetDatumReferences**
- **pfcShrinkwrapModelExportInstructions::SetDatumReferences**

The method `pfcShrinkwrapModelExportInstructions::GetMethod` returns the method used to create the shrinkwrap. The types of shrinkwrap methods are:

- `pfcSWCREATE_SURF_SUBSET`—Surface Subset
- `pfcSWCREATE_FACETED_SOLID`—Faceted Solid
- `pfcSWCREATE_MERGED_SOLID`—Merged Solid

The method `pfcShrinkwrapModelExportInstructions::GetQuality` specifies the quality level for the system to use when identifying surfaces or components that contribute to the shrinkwrap model. Quality ranges from 1 which produces the coarsest representation of the model in the fastest time, to 10 which produces the most exact representation. Use the method `pfcShrinkwrapModelExportInstructions::SetQuality` to set the quality level for the system during the shrinkwrap export. The default value is 1.

The method `pfcShrinkwrapModelExportInstructions::GetAutoHoleFilling` returns true if auto hole filling is enabled during Shrinkwrap export. The method `pfcShrinkwrapModelExportInstructions::SetAutoHoleFilling` sets a flag that forces Creo application to identify all holes and surfaces that intersect a single surface and fills those holes during shrinkwrap. The default value is true.

The methods `pfcShrinkwrapModelExportInstructions::GetIgnoreSkeleton` and `pfcShrinkwrapModelExportInstructions::SetIgnoreSkeleton` determine whether the skeleton model geometry must be included in the shrinkwrap model.

The methods `pfcShrinkwrapModelExportInstructions::GetIgnoreQuilts` and `pfcShrinkwrap.ShrinkwrapModelExportInstructions::SetIgnoreQuilts` determine whether external quilts must be included in the shrinkwrap model.

The method `pfcShrinkwrapModelExportInstructions::GetAssignMassProperties` determines the mass property of the model. The method `pfcShrinkwrapModelExportInstructions::SetAssignMassProperties` assign mass properties to the shrinkwrap model. The default value is false and the mass properties of the original model is assigned to the shrinkwrap model. If the value is set to true, the user must assign a value for the mass properties.

The method `pfcShrinkwrapModelExportInstructions::GetIgnoreSmallSurfaces` specifies whether small surfaces are ignored during the creation of a shrinkwrap model. The method `pfcShrinkwrapModelExportInstructions::SetIgnoreSmallSurfaces` sets a flag that forces Creo application to skip surfaces smaller than a certain size. The default value is false. The size of the surface is specified as a percentage of the model's size. This size can be modified using the methods `pfcShrinkwrapModelExportInstructions::GetSmallSurfPer`

centage and pfcShrinkwrapModelExportInstructions::SetSmallSurfPercentage.

The methods pfcShrinkwrapModelExportInstructions::GetDatumReferences and pfcShrinkwrapModelExportInstructions::SetDatumReferences specify and select the datum planes, points, curves, axes, and coordinate system references to be included in the shrinkwrap model.

Surface Subset Options

Methods Introduced:

- **pfcShrinkwrapSurfaceSubsetInstructions::Create**
- **pfcShrinkwrapSurfaceSubsetInstructions::GetAdditionalSurfaces**
- **pfcShrinkwrapSurfaceSubsetInstructions::SetAdditionalSurfaces**
- **pfcShrinkwrapSurfaceSubsetInstructions::GetOutputModel**
- **pfcShrinkwrapSurfaceSubsetInstructions::SetOutputModel**

The static method pfcShrinkwrapSurfaceSubsetInstructions::Create returns an object used to create a shrinkwrap model of surface subset type. Specify the name of the output model in which the shrinkwrap is to be created as an input to this method.

The method pfcShrinkwrapSurfaceSubsetInstructions::GetAdditionalSurfaces specifies the surfaces included in the shrinkwrap model while the method pfcShrinkwrapSurfaceSubsetInstructions::SetAdditionalSurfaces selects individual surfaces to be included in the shrinkwrap model.

The method pfcShrinkwrapSurfaceSubsetInstructions::GetOutputModel returns the template model where the shrinkwrap geometry is to be created while the method pfcShrinkwrapSurfaceSubsetInstructions::SetOutputModel sets the template model.

Faceted Solid Options

The pfcShrinkwrapFacetedFormatInstructions interface consists of the following types:

- pfcSWFACETED_PART—Creo part with normal geometry. This is the default format type.

- `pfcSWFACETED_STL`—An `STL` file.
- `pfcSWFACETED_VRML`—A `VRML` file.

Use the `Create` method to create the object of the specified type. Upcast the object to use the general methods available in this interface .

Methods Introduced:

- **pfcShrinkwrapFacetedFormatInstructions::GetFormat**
- **pfcShrinkwrapFacetedFormatInstructions::GetFramesFile**
- **pfcShrinkwrapFacetedFormatInstructions::SetFramesFile**

The method `pfcShrinkwrapFacetedFormatInstructions::GetFormat` returns the output file format of the shrinkwrap model.

The methods `pfcShrinkwrapFacetedFormatInstructions::GetFramesFile` and `pfcShrinkwrapFacetedFormatInstructions::SetFramesFile` enable you to select a frame file to create a faceted solid motion envelope model that represents the full motion of the mechanism captured in the frame file. Specify the name and complete path of the frame file.

Faceted Part Options

Methods Introduced:

- **pfcShrinkwrapFacetedPartInstructions::Create**
- **pfcShrinkwrapFacetedPartInstructions::GetLightweight**
- **pfcShrinkwrapFacetedPartInstructions::SetLightweight**

The static method `pfcShrinkwrapFacetedPartInstructions::Create` returns an object used to create a shrinkwrap model of shrinkwrap faceted type. The input parameters of this method are:

- *OutputModel*—Specify the output model where the shrinkwrap must be created.
- *Lightweight*—Specify this value as True if the shrinkwrap model is a Lightweight Creo part.

The method `pfcShrinkwrapFacetedPartInstructions::GetLightweight` returns a true value if the output file format of the shrinkwrap model is a LightweightCreo part. The method `pfcShrinkwrapFacetedPartInstructions::SetLightweight` specifies if theCreo part is exported as a light weight faceted geometry.

VRML Export Options

Methods Introduced:

- **pfcShrinkwrapVRMLInstructions::Create**
- **pfcShrinkwrapVRMLInstructions::GetOutputFile**
- **pfcShrinkwrapVRMLInstructions::SetOutputFile**

The static method `pfcShrinkwrapVRMLInstructions::Create` returns an object used to create a shrinkwrap model of shrinkwrap `VRML` format. Specify the name of the output model as an input to this method.

The method `pfcShrinkwrapVRMLInstructions::GetOutputFile` returns the name of the output file to be created and the method `pfcShrinkwrapVRMLInstructions::SetOutputFile` specifies the name of the output file to be created.

STL Export Options

Methods Introduced:

- **pfcShrinkwrapVRMLInstructions::Create**
- **pfcShrinkwrapVRMLInstructions::GetOutputFile**
- **pfcShrinkwrapVRMLInstructions::SetOutputFile**

The static method `pfcShrinkwrapVRMLInstructions::Create` returns an object used to create a shrinkwrap model of shrinkwrap `STL` format. Specify the name of the output model as an input to this method.

The method `pfcShrinkwrapSTLInstructions::GetOutputFile` returns the name of the output file to be created and the method `pfcShrinkwrapSTLInstructions::SetOutputFile` specifies the name of the output file to be created.

Merged Solid Options

Methods Introduced:

- **pfcShrinkwrapMergedSolidInstructions::Create**
- **pfcShrinkwrapMergedSolidInstructions::GetAdditionalComponents**
- **pfcShrinkwrapMergedSolidInstructions::SetAdditionalComponents**

The static method `pfcShrinkwrapMergedSolidInstructions::Create` returns an object used to create a shrinkwrap model of merged solids format. Specify the name of the output model as an input to this method.

The methods `pfcShrinkwrapMergedSolidInstructions::`

`GetAdditionalComponents` specifies individual components of the assembly to be merged into the shrinkwrap model. Use the method `pfcShrinkwrapMergedSolidInstructions::SetAdditionalComponents` to select individual components of the assembly to be merged into the shrinkwrap model.

Importing Files

Method Introduced:

- **pfcModel::Import**

The method `pfcModel::Import` reads a file into Creo. The format must be the same as it would be if these files were created by Creo. The parameters are:

- *FilePath*—Absolute path of the file to be imported along with its extension.
- *ImportData*—The `pfcImportInstructions` object that controls the import operation.

Import Instructions

Methods Introduced:

- **pfcRelationImportInstructions::Create**
- **pfcIGESSectionImportInstructions::Create**
- **pfcProgramImportInstructions::Create**
- **pfcConfigImportInstructions::Create**
- **pfcDWGSetupImportInstructions::Create**
- **pfcSpoolImportInstructions::Create**
- **pfcConnectorParamsImportInstructions::Create**
- **pfcASSEMTreeCFGImportInstructions::Create**
- **pfcWireListImportInstructions::Create**
- **pfcCableParamsImportInstructions::Create**
- **pfcSTEPImport2DInstructions::Create**
- **pfcIGESImport2DInstructions::Create**
- **pfcDXFImport2DInstructions::Create**
- **pfcDWGImport2DInstructions::Create**

The methods described in this section create an instructions data object to import a file of a specified type into Creo application. The details are as shown in the table below:

Interface	Used to Import
pfcRelationImportInstructions	A list of relations and parameters in a part or assembly.
pfcIGESSectionImportInstructions	A section model in IGES format.
pfcProgramImportInstructions	A program file for a part or assembly that can be edited to change the model.
pfcConfigImportInstructions	Configuration instructions.
pfcDWGSetupImportInstructions	A drawing s/u file.
pfcSpoolImportInstructions	Spool instructions.
pfcConnectorParamsImportInstructions	Connector parameter instructions.
pfcASSEMTreeCFGImportInstructions	Assembly tree CFG instructions.
pfcWireListImportInstructions	Wirelist instructions.
pfcCableParamsImportInstructions	Cable parameters from an assembly.
pfcSTEPImport2DInstructions	A part or assembly in STEP format.
pfcIGESImport2DInstructions	A part or assembly in IGES format.
pfcDXFImport2DInstructions	A drawing in DXF format.
pfcDWGImport2DInstructions	A drawing in DWG format.

Note

- The method pfcModel::Import does not support importing of CADAM type of files.
- If a model or the file type STEP, IGES, DWX, or SET already exists, the imported model is appended to the current model. For more information on methods that return models of the types STEP, IGES, DWX, and SET, refer to Getting a Model Object on page 104.

Importing 2D Models

Method Introduced:

- **pfcBaseSession::Import2DModel**

The method pfcBaseSession::Import2DModel imports a two dimensional model based on the following parameters:

- *NewModelName*—Specifies the name of the new model.
- *Type*—Specifies the type of the model. The type can be one of the following:
  - STEP
  - IGES
  - DXF

- DWG
- SET

- *FilePath*—Specifies the location of the file to be imported along with the file extension
- *Instructions*—Specifies the `pfcImport2DInstructions` object that controls the import operation.

  The interface `pfcpfcImport2DInstructions` contains the following attributes:

  - *Import2DViews*—Defines whether to import 2D drawing views.
  - *ScaleToFit*—If the current model has a different sheet size than that specified by the imported file, set the parameter to true to retain the current sheet size. Set the parameter to false to retain the sheet size of the imported file.
  - *FitToLeftCorner*—If this parameter is set to true, the bottom left corner of the imported file is adjusted to the bottom left corner of the current model. If it is set to false, the size of imported file is retained.

Note

The method `pfcBaseSession::Import2DModel` does not support importing of `CADAM` type of files.

Importing 3D Geometry

Methods Introduced:

- **pfcBaseSession::GetImportSourceType**
- **pfcBaseSession::ImportNewModel**
- **pfcLayerImportFilter::OnLayerImport**

For some input formats, the method `pfcBaseSession::GetImportSourceType` returns the type of model that can be imported using a designated file. The input parameters of this method are:

- *FileToImport*—Specifies the path of the file along with its name and extension.
- *NewModelImportType*—Specifies the type of model to be imported.

The method `pfcBaseSession::ImportNewModel` is used to import an external 3D format file and creates a new model or set of models of type `pfcModel`. The input parameters of this method are:

- *FileToImport*—Specifies the path to the file along with its name and extension
- `pfcNewModelImportType`—Specifies the type of model to be imported. The types of models that can be imported are as follows:
  - `pfcIMPORT_NEW_IGES`
  - `pfcIMPORT_NEW_VDA`
  - `pfcIMPORT_NEW_NEUTRAL`
  - `pfcIMPORT_NEW_CADDS`
  - `pfcIMPORT_NEW_STEP`
  - `pfcIMPORT_NEW_STL`
  - `pfcIMPORT_NEW_VRML`
  - `pfcIMPORT_NEW_POLTXT`
  - `pfcIMPORT_NEW_CATIA_SESSION`
  - `pfcIMPORT_NEW_CATIA_MODEL`
  - `pfcIMPORT_NEW_DXF`
  - `pfcIMPORT_NEW_ACIS`
  - `pfcIMPORT_NEW_PARASOLID`
  - `pfcIMPORT_NEW_ICEM`
  - `pfcIMPORT_NEW_CATIA_PART`
  - `pfcIMPORT_NEW_CATIA_PRODUCT`
  - `pfcIMPORT_NEW_UG`
  - `pfcIMPORT_NEW_PRODUCTVIEW`
  - `pfcIMPORT_NEW_CATIA_CGR`
  - `pfcIMPORT_NEW_JT`
  - `pfcIMPORT_NEW_SW_PART`
  - `pfcIMPORT_NEW_SW_ASSEM`
  - `pfcIMPORT_NEW_INVENTOR_PART`
  - `pfcIMPORT_NEW_INVENTOR_ASSEM`
  - `pfcIMPORT_NEW_CC`
  - `pfcIMPORT_NEW_SEDGE_PART`
  - `pfcIMPORT_NEW_SEDGE_ASSEMBLY`
  - `pfcIMPORT_NEW_SEDGE_SHEETMETAL_PART`

- pfcIMPORT_NEW_3MF

- pfcModelType—Specifies the type of the model. It can be a part, assembly or drawing.
- *NewModelName*—Specifies a name for the imported model.
- pfcLayerImportFilter—Specifies the layer filter. This parameter is optional.

The interface pfcLayerImportFilter has a call back method pfcLayerImportFilter::OnLayerImport. Creo application passes the object pfcImportedLayer describing each imported layer to the layer filter to allow you to perform changes on each layer as it is imported.

The method pfcXCancelProEAction::Throw can be called from the body of the method pfcLayerImportFilter::OnLayerImport to end the filtering of the layers.

Modifying the Imported Layers

Layers help you organize model items so that you can perform operations on those items collectively. These operations primarily include ways of showing the items in the model, such as displaying or blanking, selecting, and suppressing. The methods described in this section modify the attributes of the imported layers.

Methods Introduced:

- **pfcImportedLayer::GetName**
- **pfcImportedLayer::SetNewName**
- **pfcImportedLayer::GetSurfaceCount**
- **pfcImportedLayer::GetCurveCount**
- **pfcImportedLayer::GetTrimmedSurfaceCount**
- **pfcImportedLayer::SetAction**

Layers are identified by their names. The method pfcImportedLayer::GetName returns the name of the layer while the method pfcImportedLayer::SetNewName can be used to set the name of the layer. The name can be numeric or alphanumeric.

The method pfcImportedLayer::GetSurfaceCount returns the number of curves on the layer.

The method pfcImportedLayer::GetTrimmedSurfaceCount returns the number of trimmed surfaces on the layer and the method pfcImportedLayer::GetCurveCount returns the number of curves on the layer.

The method `pfcImportedLayer::SetAction` sets the display of the imported layers. The input parameter for this method is ImportAction. The types of actions that can be performed on the imported layers are:

- `pfcIMPORT_LAYER_DISPLAY`—Displays the imported layer.
- `pfcIMPORT_LAYER_SKIP`—Does not import entities on this layer.
- `pfcIMPORT_LAYER_BLANK`—Blanks the selected layer.
- `pfcIMPORT_LAYER_IGNORE`—Imports only entities on this layer but not the layer.

The default action type is `pfcIMPORT_LAYER_DISPLAY`.

Import Feature Properties

The methods defined in this section get the properties of the import feature.

Methods Introduced:

- **wfcWFeature::GetIdMap**
- **wfcWFeature::GetUserIds**
- **wfcWFeature::GetItemIds**
- **wfcImportFeatureIdMap::GetItemId**
- **wfcImportFeatureIdMap::SetItemId**
- **wfcImportFeatureIdMap::GetItemType**
- **wfcImportFeatureIdMap::SetItemType**
- **wfcImportFeatureIdMap::GetUserId**
- **wfcImportFeatureIdMap::SetUserId**
- **wfcWFeature::GetImportFeatureData**
- **wfcImportFeatureData::GetIntfType**
- **wfcImportFeatureData::GetFileName**
- **wfcImportFeatureData::GetCsys**
- **wfcImportFeatureData::GetAttributes**

The method `wfcWFeature::GetIdMap` returns an array that contains the mapping between user defined IDs and the IDs assigned by Creo application to the entity items in the import feature.

The method `wfcWFeature::GetUserIds` converts a Creo item ID to an array of user defined IDs. For example, if the edges defined by you are created as a single edge by Creo application and are assigned a single item ID. When you

pass this single item ID assigned by Creo application to the method `wfcWFeature::GetUserIds`, it will return an array of user IDs against each edge defined. The input parameters are:

- *ItemId*—Specifies the ID assigned by Creo application for the geometry item.
- *ItemType*—Specifies the type of geometry item. The following types are supported:
  - Surface (`pfcITEM_SURFACE`)
  - Edge (`pfcITEM_EDGE`)
  - Quilt (`pfcITEM_QUILT`)

Use the method `wfcWFeature::GetItemIds` to get the IDs assigned by Creo application for the specified user ID. The input parameters are:

- *UserId*—Specifies the user ID for the geometry item.
- *ItemType*—Specifies the type of geometry item. The following types are supported:
  - Surface (`pfcITEM_SURFACE`)
  - Edge (`pfcITEM_EDGE`)
  - Quilt (`pfcITEM_QUILT`)

The methods `wfcImportFeatureIdMap::GetItemId` and `wfcImportFeatureIdMap::SetItemId` retrieve and set the item id of an import feature id map.

The methods `wfcImportFeatureIdMap::GetItemType` and `wfcImportFeatureIdMap::SetItemType` retrieve and set item type of the import feature id map using the enumerated type `pfcModelItemType`.

The methods `wfcImportFeatureIdMap::GetUserId` and `wfcImportFeatureIdMap::GetUserId` retrieve and set the id or ids assigned by Creo Parametric.

The method `wfcWFeature::GetImportFeatureData` returns information about the parameters assigned to the specified import feature as a `wfcImportFeatureData` object.

The method `wfcImportFeatureData::GetIntfType` returns the file format type of the specified import feature.

The method `wfcImportFeatureData::GetFileName` returns the name of the file from which the import feature was created.

The method `wfcImportFeatureData::GetCsys` returns the coordinate system with respect to which the import feature is aligned.

The method `wfcImportFeatureData::GetAttributes` returns the attributes assigned to the import feature as a `wfcImportFeatAttributes` object.

Import Feature Attributes

Attributes define the action to be taken when creating the import feature.

Methods Introduced:

- **wfcImportFeatAttributes::Create**
- **wfcImportFeatAttributes::MakeSolid**
- **wfcImportFeatAttributes::SetSolid**
- **wfcImportFeatAttributes::IsAdded**
- **wfcImportFeatAttributes::SetAdded**
- **wfcImportFeatAttributes::AreSurfacesJoined**
- **wfcImportFeatAttributes::SetSurfacesJoined**
- **wfcImportFeatAttributes::GetAddBodies**
- **wfcImportFeatAttributes::SetAddBodies**
- **wfcImportFeatAttributes::GetBodyUseOpts**
- **wfcImportFeatAttributes::SetBodyUseOpts**
- **wfcImportFeatAttributes::GetBodyArray**
- **wfcImportFeatAttributes::SetBodyArray**

The method `wfcImportFeatAttributes::Create` creates a data object that contains information about the attributes of the import feature.

The method `wfcImportFeatAttributes::MakeSolid` returns a boolean value that indicates if the import feature was created as a solid or a surface type. Use the method `wfcImportFeatAttributes::SetSolid` to specify whether the import feature must be created as a solid or a surface type. You must specify True to create the import feature as solid and False to create it as a surface.

Note

If the import feature is an open surface, you cannot create the import feature of solid type even if you set the boolean value to the method `wfcImportFeatAttributes::SetSolid` to True.

The method `wfcImportFeatAttributes::IsAdded` returns a boolean value that indicates if the import feature is created as a cut or a protrusion. You can create the import feature as a cut or protrusion only if the import feature is of solid type. Use the method `wfcImportFeatAttributes::SetAdded` to specify if the import feature must be created as a cut or protrusion. Specify True to create as a cut and false to create as a protrusion.

The method `wfcImportFeatAttributes::AreSurfacesJoined` returns a boolean value that indicates if the import feature is created as a single quilt (joined surface) or separate surfaces (as it was in the original file). You can create the import feature as single quilt or separate surfaces only when the import feature is of surface type. Use the method `wfcImportFeatAttributes::SetSurfacesJoined` to specify if the import feature must be created as a single quilt or separate surfaces. You must specify True to create as single quilt and false to create as separate surfaces.

Use the methods `wfcImportFeatAttributes::GetAddBodies` and `wfcImportFeatAttributes::SetAddBodies` to retrieve and set the option for creating the same body structure as is present in the source file.

The methods `wfcImportFeatAttributes::GetBodyUseOpts` and `wfcImportFeatAttributes::SetBodyUseOpts` retrieve and specifies the body options you can use while importing a feature and is defined by the enumerated data type `wfcImportBodyUseOpts`. The valid values are:

- `wfcIMPORT_BODY_USE_DEFAULT`—Imports feature in the default body.
- `wfcIMPORT_BODY_USE_NEW`—Imports feature in a new body.
- `wfcIMPORT_BODY_USE_ALL`—Currently not supported.
- `wfcIMPORT_BODY_USE_SELECTED`—Imports feature in a selected body.

The methods `wfcImportFeatAttributes::GetBodyArray` and `wfcImportFeatAttributes::SetBodyArray` get and set the array of bodies to be selected. By default, the size is `1`. You must not call the method `wfcImportFeatAttributes::SetBodyArray` if you do not want to use any bodies in the import operation.

Redefining the Import Feature

The methods defined in this section allow you to redefine the import feature.

Methods Introduced:

- **wfcWFeature::RedefineImportFeature**
- **wfcImportFeatureRedefSource::GetRedefOperationType**
- **wfcWSession::GetImportFeatRedefSourceType**

The method `wfcWFeature::RedefineImportFeature` redefines the import feature. The input argument *Source* contains the data about the source files and operation type to be used for redefinition of import feature as a `wfcImportFeatureRedefSource` object.

Use the method `wfcImportFeatureRedefSource::GetRedefOperationType` to get the operation type for the redefinition of the import feature as an enumerated data type `wfcImportFeatRedefOperationType`. The types of operation are:

- `wfcIMPORT_FEAT_REDEF_CHANGE_ATTR`—Specifies if the attributes of the existing import feature must be changed.
- `wfcIMPORT_FEAT_REDEF_SUBSTITUTE`—Specifies if the existing import feature is substituted with a new import feature.

The method `wfcWSession::GetImportFeatRedefSourceType` determines the type of data source that must be passed as input for redefining the import feature. The type of data source depends on the type of operation. The data source can be of following types:

- `wfcIMPORT_FEAT_REDEF_DATA_SOURCE_NONE`—Specifies that the existing data source must be reloaded.
- `wfcIMPORT_FEAT_REDEF_DATA_SOURCE_ATTR`—Specifies that the data source must contain information about the attributes of the import feature.
- `wfcIMPORT_FEAT_REDEF_DATA_SOURCE_NEW`—Specifies that the data source must contain information about the new part model to be imported into Creo application.

Extracting Geometry as Interface Data

The methods defined in this section allow you to extract interface data from Creo geometry. You can use this data to create the import feature.

Methods Introduced:

- **wfcWPart::GetInterfaceData**
- **wfcConversionOptions::Create**
- **wfcConversionOptions::GetEdgeRepresentation**
- **wfcConversionOptions::SetEdgeRepresentation**
- **wfcEdgeRepresentation::Create**
- **wfcEdgeRepresentation::GetUVPoints**
- **wfcEdgeRepresentation::SetUVPoints**
- **wfcEdgeRepresentation::GetUVCurves**
- **wfcEdgeRepresentation::SetUVCurves**
- **wfcEdgeRepresentation::GetXYZCurves**
- **wfcEdgeRepresentation::SetXYZCurves**
- **wfcConversionOptions::GetCurveConversionOption**

- **wfcConversionOptions::SetCurveConversionOption**
- **wfcConversionOptions::GetSurfaceConversionOption**
- **wfcConversionOptions::SetSurfaceConversionOption**
- **wfcInterfaceData::Create**
- **wfcInterfaceData::GetSurfaceData**
- **wfcInterfaceData::SetSurfaceData**
- **wfcInterfaceData::GetEdgeDescriptor**
- **wfcInterfaceData::SetEdgeDescriptor**
- **wfcInterfaceData::GetQuiltData**
- **wfcInterfaceData::SetQuiltData**
- **wfcInterfaceData::GetDatumData**
- **wfcInterfaceData::SetDatumData**
- **wfcInterfaceData::GetAccuracytype**
- **wfcInterfaceData::SetAccuracytype**
- **wfcInterfaceData::GetAccuracy**
- **wfcInterfaceData::SetAccuracy**
- **wfcInterfaceData::GetOutline**
- **wfcInterfaceData::SetOutline**
- **wfcWSession::GetDataSourceType**

The method `wfcWPart::GetInterfaceData` extracts information about the geometry of a part as a `wfcInterfaceData` object. The interface data can be used to extract all geometric data in order to convert it to another geometric format. Pass the information about the curves, edges, and surfaces of a part as an object of type `wfcConversionOptions`.

The method `wfcConversionOptions::Create` creates a data object of type `wfcConversionOptions` that contains information about the curves, edges, and surfaces of a part.

The method `wfcConversionOptions::GetEdgeRepresentation` gets the information about the representation of edges in a part as a `wfcEdgeRepresentation` object. Use the method `wfcConversionOptions::SetEdgeRepresentation` to set the parameters for the representation of edges in a part.

The method `wfcEdgeRepresentation::Create` creates a data object that contains information about the representation of edges in a part. The parameters of this method are:

- *UVPoints*—Specifies the representation of the edge using UV points. Use the methods `wfcEdgeRepresentation::GetUVPoints` and `wfcEdgeRepresentation::SetUVPoints` to get and set the UV points on the surface for edge representation.
- *UVCurves*—Specifies the representation of the edge using UV curves. Use the methods `wfcEdgeRepresentation::GetUVCurves` and `wfcEdgeRepresentation::SetUVCurves` to get and set the edge UV curves on the surface for edge representation.
- *XYZCurves*—Specifies the representation of the edge using XYZ curves. Use the methods `wfcEdgeRepresentation::GetXYZCurves` and `wfcEdgeRepresentation::SetXYZCurves` to get and set the XYZ curves on the surface for edge representation.

The method `wfcConversionOptions::GetCurveConversionOption` returns the conversion option set for curves during the data exchange. Use the method `wfcConversionOptions::SetCurveConversionOption` to set the curve conversion option.

The method `wfcConversionOptions::GetSurfaceConversionOption` returns the conversion option set for surfaces during the data exchange. Use the method `wfcConversionOptions::SetSurfaceConversionOption` to set the surface conversion option.

The method `wfcInterfaceData::Create` creates a data object that contains the interface data.

The method `wfcInterfaceData::GetSurfaceData` returns an array of surface data for the specified interface data. Use the method `wfcInterfaceData::SetSurfaceData` to set the surface data for a part import.

The method `wfcInterfaceData::GetEdgeDescriptor` returns an array of edge data for the specified interface data. Use the method `wfcInterfaceData::SetEdgeDescriptor` to set the edge data for a part import.

The method `wfcInterfaceData::GetQuiltData` returns an array of quilt data for the specified interface data. Use the method `wfcInterfaceData::SetQuiltData` to set the quilt data for a part import.

The method `wfcInterfaceData::GetDatumData` returns an array of datum data for the specified interface data. Use the method `wfcInterfaceData::SetDatumData` to set the datum data for a part import.

The method `wfcInterfaceData::GetAccuracytype` gets the type of accuracy for the interface data using the enumerated type `wfcAccuracytype`. Use the method `wfcInterfaceData::SetAccuracytype` to set the type of accuracy. The valid values are:

- `wfcACCU_RELATIVE`—Specifies the comparative ratio of the smallest model dimension to the part size. Creo application can display geometry equal to or greater than the ratio without any error. This is the default accuracy type in a model.
- `wfcACCU_ABSOLUTE`—Specifies the absolute accuracy of a model that defines the smallest allowable size of the unit that Creo application can display or interpret without any error.

The method `wfcInterfaceData::GetAccuracy` returns the value of the accuracy for the specified model. Use the method `wfcInterfaceData::SetAccuracy` to set the accuracy for the specified model.

The method `wfcInterfaceData::GetOutline` returns the maximum and minimum values of x, y, and z coordinates for the display outline of the bounding box that contains the interface data. Use the method `wfcInterfaceData::SetOutline` to set the display outline for the bounding box to contain the interface data.

The method `wfcWSession::GetDataSourceType` returns the type of the interface source data in session using the enumerated type `wfcIntfDataSourceType`. The valid values are:

- `wfcINTF_DATA_SOURCE_FILE`—Specifies that the data source is from a file.
- `wfcINTF_DATA_SOURCE_MEMORY`—Specifies that the data source is of neutral type.

Extracting Interface Data for Neutral Files

The methods defined in this section allow you to extract interface data for neutral data files. You can use this data to create an import feature.

Methods Introduced:

- **wfcWIntfNeutral::Create**
- **wfcWIntfNeutral::GetInterfaceData**
- **wfcWIntfNeutral::SetInterfaceData**

The method `wfcWIntfNeutral::Create` creates a data object of type `wfcWIntfNeutral` that contains information about the interface data for the neutral file type. The input parameters for this method are:

- *FileName*—Specifes the name of the neutral file.

- *Data*—Specifies the interface data for the specified neutral file as an object of type `wfcInterfaceData`. Use the method `wfcWIntfNeutral::SetInterfaceData` to set the interface data for the specified neutral file. The method `wfcWIntfNeutral::GetInterfaceData` gets the interface data for the specified neutral file. Refer to the section Extracting Geometry as Interface Data on page 555, for more information on interface data object `wfcInterfaceData`.

Associative Topology Bus Enabled Models and Features

Associative Topology Bus (ATB) propagates changes made to the original CAD system data in the heterogeneous design environment. All geometric IDs preserved by the native system after the change to the native file are also preserved in the imported geometry by the ATB update. With ATB, you can work with Creo part or assembly that is:

- A Translated Image Model (TIM) representation of a model imported from the ATB interface, such as, CADDS or CATIA.
- A Creo assembly containing one or more components which are models imported from an ATB interface, such as, CADDS or CATIA.
- ACreo part containing an Import feature that is imported from an ATB interface such as, ICEM.

Only import operations in Creo application create TIM parts and assemblies. You can open CATIA, CADDS model files as TIMs. Neutral part files and files of other ATB-enabled formats are imported as native Creo parts with ATB-enabled features.

The TIM parts and assemblies store their ATB information at the model level. However, ATB-enabled import features store ATB information at the feature-level. The TIMs are displayed in the Model Tree with ATB icons that indicate their status with respect to their reference file as up-to-date, out-of-date, and so on.

These methods related to ATB models or features enable you to perform the following actions on a TIM model or ATB-enabled feature or the entire geometry of the imported model:

- Check the status of the TIMs or the ATB-enabled features.
- Update TIMs or ATB-enabled features that are identified as out-of-date.
- Change the link of a TIM or ATB-enabled feature.
- Break the association between a TIM or the ATB-enabled feature and the original reference model.

Methods Introduced:

- **wfcWModel::GetTIMInfo**
- **wfcTIMInfo::IsModelTIM**
- **wfcTIMInfo::ModelHasTIMFeats**
- **wfcTIMInfo::GetFeatIds**
- **wfcWModel::VerifyATB**
- **wfcATBVerificationResults::GetOutOfDateModels**
- **wfcATBVerificationResults::GetUnlinkedModels**
- **wfcATBVerificationResults::GetOldVersionModels**
- **wfcWModel::MarkATBModelAsOutOfDate**
- **wfcWModel::UpdateATB**
- **wfcWModel::RelinkATB**

The method `wfcWModel::GetTIMInfo` returns information about TIM in the specified model as a `wfcTIMInfo` object.

The method `wfcTIMInfo::IsModelTIM` checks if the specified model is a TIM.

The method `wfcTIMInfo::ModelHasTIMFeats` checks if the specified model contains a TIM feature.

The method `wfcTIMInfo::FeatIds` lists all the TIMs or ATB-enabled features present in the specified model. This method can be called after the method `wfcTIMInfo::ModelHasTIMFeats` which determines if the specified model has one or more TIM features.

The method `wfcWModel::VerifyATB` verifies if the specified ATB model is out of date with the source CAD model. The method first checks if the specified model is a TIM. If the model is not a TIM, this method checks if the ATB-enabled model was created by importing or appending ICEM or neutral surfaces to existingCreo part models. The method `wfcWModel::VerifyATB` returns the `wfcATBVerificationResults` object that represents the status of the TIMs.

You can specify a Creo part or assembly that is—

- A Translated Image Model (TIM) representation of a model imported from the ATB interface, such as, CADDS or CATIA.
- A Creo assembly containing one or more components which are models imported from an ATB Interface, such as, CADDS or CATIA.
- A Creo part containing an Import feature that is imported from an ATB interface such as, ICEM.

The input parameters for this method are:

- *FeatIds*—Specify an array of feature ids for the ATB-enabled features in the model. If a model contains more than one ATB-enabled feature, the verify method works only on the specified feature. If you do not specify a feature id, the method `wfcWModel::VerifyATB` verifies the entire model including TIMs from non-native CAD models.
- *SearchPaths*—Specify the complete location to the source CAD model. You can specify multiple directories to search for the model. If no search path is specified, then the method will search in current working directory or locations set in the configuration option `atb_search_path`.

Use the method `wfcATBVerificationResults::GetOutOfDateModels` to get an array of TIMs or the ATB-enabled features that are out-of-date with the source model and require an update. These TIMs or the ATB-enabled features can be relinked. Such models are represented by a red icon in the Model Tree in the Creo user interface.

Use the method `wfcATBVerificationResults::GetUnlinkedModels` to get an array of TIMs or the ATB-enabled features that have missing links because the reference model is missing from the designated search path. These models are represented by a yellow icon in the Model Tree in the Creo user interface.

Use the method `wfcATBVerificationResults::GetOldVersionModels` to get an array of TIMs for which the source CAD model is older than the one with which the TIM was last updated. These models are represented by a yellow icon in the Model Tree in the Creo user interface.

The method `wfcWModel::MarkATBModelAsOutOfDate` identifies all the ATB-enabled features that are out of date for the update operation.

The method `wfcWModel::UpdateATB` updates the ATB-enabled models or features that are displayed in the session. The update action synchronizes the derived structure and the contents of the TIMs with the primary structure and the

content of the source non-native CAD models. This method returns an error if there are non-displayed models in the session or if the input model is not displayed.

Note

- If the link of a TIM or ATB-enabled feature is broken, you cannot re-establish the link or update the part that is independent and has lost its association with the reference model.
- The geometry added or removed from the model before the update is added or removed from the TIM after the update.
- ATB incorrectly identifies the imported geometry as up-to-date based on the old reference file which is found before the updated reference file.

The method `wfcWModel::RelinkATB` relinks a TIM to a source CAD model specified by the input argument *MasterModelPath*. This method relinks all those models or features that have lost their association or link with their master model. In order to relink a model, provide the name and location of the master model, using *MasterModelPath* to which the specified model or feature is to be linked. If the master model with the same name is found, the Creo TIM model is linked to that master model.

Printing Files

The printer instructions for printing a file are defined in `pfcPrinterInstructions` data object.

Methods Introduced:

- **pfcPrinterInstructions::Create**
- **pfcPrinterInstructions::SetPrinterOption**
- **pfcPrinterInstructions::SetPlacementOption**
- **pfcPrinterInstructions::SetModelOption**
- **pfcPrinterInstructions::SetWindowId**

The method `pfcPrinterInstructions::Create` creates a new instance of the `pfcPrinterInstructions` object. The object contains the following instruction attributes:

- *PrinterOption*—Specifies the printer settings for printing a file in terms of the `pfcPrintPrinterOption` object. Set this attribute using the method `pfcPrinterInstructions::SetPrinterOption`.

- *PlacementOption*—Specifies the placement options for printing purpose in terms of the `pfcPrintMdlOption` object. Set this attribute using the method `pfcPrinterInstructions::SetPlacementOption`.
- *ModelOption*—Specifies the model options for printing purpose in terms of the `pfcPrintPlacementOption` object. Set this attribute using the method `pfcPrinterInstructions::SetModelOption`.
- *WindowId*—Specifies the current window identifier. Set this attribute using the method `pfcPrinterInstructions::SetWindowId`.

Printer Options

The printer settings for printing a file are defined in the `pfcPrintPrinterOption` object.

Methods Introduced:

- **pfcPrintPrinterOption::Create**
- **pfcBaseSession::GetPrintPrinterOptions**
- **pfcPrintPrinterOption::SetDeleteAfter**
- **pfcPrintPrinterOption::SetFileName**
- **pfcPrintPrinterOption::SetPaperSize**
- **pfcPrintSize::Create**
- **pfcPrintSize::SetHeight**
- **pfcPrintSize::SetWidth**
- **pfcPrintSize::SetPaperSize**
- **pfcPrintPrinterOption::SetPenTable**
- **pfcPrintPrinterOption::SetPrintCommand**
- **pfcPrintPrinterOption::SetPrinterType**
- **pfcPrintPrinterOption::SetQuantity**
- **pfcPrintPrinterOption::SetRollMedia**
- **pfcPrintPrinterOption::SetRotatePlot**
- **pfcPrintPrinterOption::SetSaveMethod**
- **pfcPrintPrinterOption::SetSaveToFile**
- **pfcPrintPrinterOption::SetSendToPrinter**
- **pfcPrintPrinterOption::SetSlew**
- **pfcPrintPrinterOption::SetSwHandshake**
- **pfcPrintPrinterOption::SetUseTtf**

The method `pfcPrintPrinterOption::Create` creates a new instance of the `pfcPrintPrinterOption` object.

The method `pfcBaseSession::GetPrintPrinterOptions` retrieves the printer settings.

The `pfcPrintPrinterOption` object contains the following options:

- *DeleteAfter*—Determines if the file is deleted after printing. Set it to true to delete the file after printing. Use the method `pfcPrintPrinterOption::SetDeleteAfter` to assign this option.
- *FileName*—Specifies the name of the file to be printed. Use the method `pfcPrintPrinterOption::SetFileName` to set the name.

> **Note**
>
> If the method `pfcModel::Export` is called for `pfcExportType` object, then the argument *FileName* is ignored, and can be passed as `NULL`. You must use the method `pfcModel::Export` to set the *FileName*.

- *PaperSize*—Specifies the parameters of the paper to be printed in terms of the `pfcPrintSize` object. The method `pfcPrintPrinterOption::SetPaperSize` assigns the *PaperSize* option. Use the method `pfcPrintSize::Create` to create a new instance of the `pfcPrintSize` object. This object contains the following options:
  - *Height*—Specifies the height of paper. Use the method `pfcPrintSize::SetHeight` to set the paper height.
  - *Width*—Specifies the width of paper. Use the method `pfcPrintSize::SetWidth` to set the paper width.
  - *PaperSize*—Specifies the size of the paper used for the plot in terms of the `pfcPlotPaperSize` object. Use the method `pfcPrintSize::SetPaperSize` to set the paper size.

Note

If you want to plot a layout without adding a border on the paper, use the following paper sizes defined in the enumerated data type `pfcPlotPaperSize`:

- `CEEMPTYPLOT`—The paper size is 22.5 x 36 in
- `CEEMPTYPLOT_MM`—The paper size is 625 x 1000 mm

- *PenTable*—Specifies the file containing the pen table. Use the method `pfcPrintPrinterOption::SetPenTable` to set this option.
- *PrintCommand*—Specifies the command to be used for printing. Use the method `pfcPrintPrinterOption::SetPrintCommand` to set the command.
- *PrinterType*—Specifies the printer type. Use the method `pfcPrintPrinterOption::SetPrinterType` to assign the type.
- *Quantity*—Specifies the number of copies to be printed. Use the method `pfcPrintPrinterOption::SetQuantity` to assign the quantity.
- *RollMedia*—Determines if roll media is to be used for printing. Set it to true to use roll media. Use the method `pfcPrintPrinterOption::SetRollMedia` to assign this option.
- *RotatePlot*—Determines if the plot is rotated by 90 degrees. Set it to true to rotate the plot. Use the method `pfcPrintPrinterOption::SetRotatePlot` to set this option.
- *SaveMethod*—Specifies the save method in terms of the `pfcPrintSaveMethod` class. Use the method `pfcPrintPrinterOption::SetSaveMethod` to specify the save method. The available methods are as follows:
  - `pfcPRINT_SAVE_SINGLE_FILE`—Plot is saved to a single file.
  - `pfcPRINT_SAVE_MULTIPLE_FILE`—Plot is saved to multiple files.
  - `pfcPRINT_SAVE_APPEND_TO_FILE`—Plot is appended to a file.
- *SaveToFile*—Determines if the file is saved after printing. Set it to true to save the file after printing. Use the method `pfcPrintPrinterOption::SetSaveToFile` to assign this option.
- *SendToPrinter*—Determines if the plot is directly sent to the printer. Set it to true to send the plot to the printer. Use the method `pfcPrintPrinterOption::SetSendToPrinter` to set this option.

- *Slew*—Specifies the speed of the pen in centimeters per second in X and Y direction. Use the method `pfcPrintPrinterOption::SetSlew` to set this option.
- *SwHandshake*—Determines if the software handshake method is to be used for printing. Set it to true to use the software handshake method. Use the method `pfcPrintPrinterOption::SetSwHandshake` to set this option.
- *UseTtf*—Specifies whether `TrueType` fonts or stroked text is used for printing. Set this option to true to use `TrueType` fonts and to false to stroke all text. Use the method `pfcPrintPrinterOption::SetUseTtf` to set this option.

Placement Options

The placement options for printing purpose are defined in the `pfcPrintPlacementOption` object.

Methods Introduced:

- **pfcPrintPlacementOption::Create**
- **pfcBaseSession::GetPrintPlacementOptions**
- **pfcPrintPlacementOption::SetBottomOffset**
- **pfcPrintPlacementOption::SetClipPlot**
- **pfcPrintPlacementOption::SetKeepPanzoom**
- **pfcPrintPlacementOption::SetLabelHeight**
- **pfcPrintPlacementOption::SetPlaceLabel**
- **pfcPrintPlacementOption::SetScale**
- **pfcPrintPlacementOption::SetShiftAllCorner**
- **pfcPrintPlacementOption::SetSideOffset**
- **pfcPrintPlacementOption::SetX1ClipPosition**
- **pfcPrintPlacementOption::SetX2ClipPosition**
- **pfcPrintPlacementOption::SetY1ClipPosition**
- **pfcPrintPlacementOption::SetY2ClipPosition**

The method `pfcPrintPlacementOption::Create` creates a new instance of the `pfcPrintPlacementOption` object.

The method `pfcBaseSession::GetPrintPlacementOptions` retrieves the placement options.

The `pfcPrintPlacementOption` object contains the following options:

- *BottomOffset*—Specifies the offset from the lower-left corner of the plot. Use the method `pfcPrintPlacementOption::SetBottomOffset` to set this option.
- *ClipPlot*—Specifies whether the plot is clipped. Set this option to true to clip the plot or to false to avoid clipping of plot. Use the method `pfcPrintPlacementOption::SetClipPlot` to set this option.
- *KeepPanzoom*—Determines whether pan and zoom values of the window are used. Set this option to true use pan and zoom and false to skip them. Use the method `pfcPrintPlacementOption::SetKeepPanzoom` to set this option.
- *LabelHeight*—Specifies the height of the label in inches. Use the method `pfcPrintPlacementOption::SetLabelHeight` to set this option.
- *PlaceLabel*—Specifies whether you want to place the label on the plot. Use the method `pfcPrintPlacementOption::SetPlaceLabel` to set this option.
- *Scale*—Specifies the scale used for the plot. Use the method `pfcPrintPlacementOption::SetScale` to set this option.
- *ShiftAllCorner*—Determines whether all corners are shifted. Set this option to true to shift all corners or to false to skip shifting of corners. Use the method `pfcPrintPlacementOption::SetShiftAllCorner` to set this option.
- *SideOffset*—Specifies the offset from the sides. Use the method `pfcPrintPlacementOption::SetSideOffset` to set this option.
- *X1ClipPosition*—Specifies the first X parameter for defining the clip position. Use the method `pfcPrintPlacementOption::SetX1ClipPosition` to set this option.
- *X2ClipPosition*—Specifies the second X parameter for defining the clip position. Use the method `pfcPrintPlacementOption::SetX2ClipPosition` to set this option.
- *Y1ClipPosition*—Specifies the first Y parameter for defining the clip position. Use the method `pfcPrintPlacementOption::SetY1ClipPosition` to set this option.
- *Y2ClipPosition*—Specifies the second Y parameter for defining the clip position. Use the method `pfcPrintPlacementOption::SetY2ClipPosition` to set this option.

Model Options

The model options for printing purpose are defined in the `pfcPrintMdlOption` object.

Methods Introduced:

- **pfcPrintMdlOption::Create**
- **pfcBaseSession::GetPrintMdlOptions**
- **pfcPrintMdlOption::SetDrawFormat**
- **pfcPrintMdlOption::SetFirstPage**
- **pfcPrintMdlOption::SetLastPage**
- **pfcPrintMdlOption::SetLayerName**
- **pfcPrintMdlOption::SetLayerOnly**
- **pfcPrintMdlOption::SetMdl**
- **pfcPrintMdlOption::SetQuality**
- **pfcPrintMdlOption::SetSegmented**
- **pfcPrintMdlOption::SetSheets**
- **pfcPrintMdlOption::SetUseDrawingSize**
- **pfcPrintMdlOption::SetUseSolidScale**

The method `pfcPrintMdlOption::Create` creates a new instance of the `pfcPrintMdlOption` object.

The method `pfcBaseSession::GetPrintMdlOptions` retrieves the model options.

The `pfcPrintMdlOption` object contains the following options:

- *DrawFormat*—Displays the drawing format used for printing. Use the method `pfcPrintMdlOption::SetDrawFormat` to set this option.
- *FirstPage*—Specifies the first page number. Use the method `pfcPrintMdlOption::SetFirstPage` to set this option.
- *LastPage*—Specifies the last page number. Use the method `pfcPrintMdlOption::SetLastPage` to set this option.
- *LayerName*—Specifies the name of the layer. Use the method `pfcPrintMdlOption::SetLayerName` to set the name.
- *LayerOnly*—Prints the specified layer only. Set this option to `true` to print the specified layer. Use the method `pfcPrintMdlOption::SetLayerOnly` to set this option.
- *Mdl*—Specifies the model to be printed. Use the method `pfcPrintMdlOption::SetMdl` to set this option.

- *Quality*—Determines the quality of the model to be printed. It checks for no line, no overlap, simple overlap, and complex overlap. Use the `pfcPrintMdlOption::SetQuality` to set this option.
- *Segmented*—If set to `true`, the printer prints the drawing in full size, but in segments that are compatible with the selected paper size. This option is available only if you are plotting a single page. Use the method `pfcPrintMdlOption::SetSegmented` to set this option.
- *Sheets*—Specifies the sheets that need to be printed in terms of the `pfcPrintSheets` class. Use the method `pfcPrintMdlOption::SetSheets` to specify the sheets. The sheets can be of the following types:
  - `pfcPRINT_CURRENT_SHEET`—Only the current sheet is printed.
  - `pfcPRINT_ALL_SHEETS`—All the sheets are printed.
  - `pfcPRINT_SELECTED_SHEETS`—Sheets of a specified range are printed.
- *UseDrawingSize*—Overrides the paper size specified in the printer options with the drawing size. Set this option to `true` to use the drawing size. Use the method `pfcPrintMdlOption::SetUseDrawingSize` to set this option.
- *UseSolidScale*—Prints with the scale used in the solid model. Set this option to `true` to use solid scale. Use the method `pfcPrintMdlOption::SetUseSolidScale` to set this option.

Plotter Configuration File (PCF) Options

The printing options for PCF file are defined in the `pfcPrinterPCFOptions` object.

Methods Introduced:

- **pfcPrinterPCFOptions::Create**
- **pfcPrinterPCFOptions::SetPrinterOption**
- **pfcPrinterPCFOptions::SetPlacementOption**
- **pfcPrinterPCFOptions::SetModelOption**

The method `pfcPrinterPCFOptions::Create` creates a new instance of the `pfcPrinterPCFOptions` object.

The `pfcPrinterPCFOptions` object contains the following options:

- *PrinterOption*—Specifies the printer settings for printing a file in terms of the `pfcPrintPrinterOption` object. Set this attribute using the method `pfcPrinterPCFOptions::SetPrinterOption`.

- *PlacementOption*—Specifies the placement options for printing purpose in terms of the `pfcPrintMdlOption` object. Set this attribute using the method `pfcPrinterPCFOptions::SetPlacementOption`.
- *ModelOption*—Specifies the model options for printing purpose in terms of the `pfcPrintPlacementOption` object. Set this attribute using the method `pfcPrinterPCFOptions::SetModelOption`.

Automatic Printing of 3D Models

Creo Object TOOLKIT C++ provides the capability of automatically creating and plotting a drawing of a solid model. The Creo Object TOOLKIT C++ application needs only to supply instructions for the print activity, and Creo application will automatically create the drawing, print it, and then discard it.

The methods listed here are analogous to the command **File ▸ Print ▸ Quick Drawing** in Creo Parametric user interface.

Method Introduced:

- **wfcQuickPrintGeneralViewInsructions::Create**
- **wfcQuickPrintGeneralViewInsructions::GetGeneralViewLocation**
- **wfcQuickPrintGeneralViewInsructions::SetGeneralViewLocation**
- **wfcQuickPrintGeneralViewInsructions::GetScale**
- **wfcQuickPrintGeneralViewInsructions::SetScale**
- **wfcQuickPrintGeneralViewInsructions::GetViewDisplayStyle**
- **wfcQuickPrintGeneralViewInsructions::SetViewDisplayStyle**
- **wfcQuickPrintGeneralViewInsructions::GetViewName**
- **wfcQuickPrintGeneralViewInsructions::SetViewName**
- **wfcQuickPrintInstructions::Create**
- **wfcQuickPrintInstructions::GetDrawingTemplate**
- **wfcQuickPrintInstructions::SetDrawingTemplate**
- **wfcQuickPrintInstructions::GetGeneralViewInstructions**
- **wfcQuickPrintInstructions::SetGeneralViewInstructions**
- **wfcQuickPrintInstructions::GetLayoutType**
- **wfcQuickPrintInstructions::SetLayoutType**
- **wfcQuickPrintInstructions::GetManualLayoutType**
- **wfcQuickPrintInstructions::SetManualLayoutType**
- **wfcQuickPrintInstructions::GetOrientation**
- **wfcQuickPrintInstructions::SetOrientation**

- **wfcQuickPrintInstructions::GetPaperSize**
- **wfcQuickPrintInstructions::SetPaperSize**
- **wfcQuickPrintInstructions::GetPrintFlatToScreen**
- **wfcQuickPrintInstructions::SetPrintFlatToScreen**
- **wfcQuickPrintInstructions::GetProjectionViewLocations**
- **wfcQuickPrintInstructions::SetProjectionViewLocations**

The method `wfcQuickPrintGeneralViewInsructions::Create` creates a new instance of the `wfcQuickPrintGeneralViewInsructions` object that contains the quick drawing print instructions for general view.

The methods `wfcQuickPrintGeneralViewInsructions::GetGeneralViewLocation` and `wfcQuickPrintGeneralViewInsructions::SetGeneralViewLocation` get and set the location of the view being added for projected view layout. This option is ignored for a manual view layout. You can set the view location using the enumerated type `wfcQuickPrintGeneralViewLocation`:

- `wfcQPRINT_PROJ_GENVIEW_MAIN`
- `wfcQPRINT_PROJ_GENVIEW_NW`
- `wfcQPRINT_PROJ_GENVIEW_SW`
- `wfcQPRINT_PROJ_GENVIEW_SE`
- `wfcQPRINT_PROJ_GENVIEW_NE`

Note

The general view location options are analogous to the images in the **Quick Drawing** dialog box under **View Layout**.

The methods `wfcQuickPrintGeneralViewInsructions::GetScale` and `wfcQuickPrintGeneralViewInsructions::SetScale` get and set the view scale.

The methods `wfcQuickPrintGeneralViewInsructions::GetViewDisplayStyle` and `wfcQuickPrintGeneralViewInsructions::SetViewDisplayStyle` get and set the display styles being used in a view using the object `wfcDrawingViewDisplay` For more information on view display styles, refer to section Drawing View Display Information on page 137.

The methods `wfcQuickPrintGeneralViewInsructions::GetViewName` and `wfcQuickPrintGeneralViewInsructions::SetViewName` get and set the saved view name.

The method `wfcQuickPrintInstructions::Create` creates a new instance of the `wfcQuickPrintInstructions` object that contains the quick drawing print instructions.

The methods `wfcQuickPrintInstructions::GetDrawingTemplate` and `wfcQuickPrintInstructions::SetDrawingTemplate` get and set the path to the drawing template file to be used for the quick drawing print operation. The methods are applicable only to views with layout type `wfcQPRINT_LAYOUT_TEMPLATE`.

The methods `wfcQuickPrintInstructions::GetGeneralViewInstructions` and `wfcQuickPrintInstructions::SetGeneralViewInstructions` get and set the quick drawing print instructions for general view using the object `wfcQuickPrintGeneralViewInsructions`.

Use the methods `wfcQuickPrintInstructions::GetLayoutType` and `wfcQuickPrintInstructions::SetLayoutType` to get and set the layout type for print operation. You can set the layout type using the enumerated type `wfcQuickPrintLayoutType`:

- `wfcQPRINT_LAYOUT_PROJ`—Specifies a projected view-type layout.
- `wfcQPRINT_LAYOUT_MANUAL`—Specifies a manually arranged layout.
- `wfcQPRINT_LAYOUT_TEMPLATE`—Specifies the use of a drawing template to define the layout. If this option is specified, only the template name is required to define the print; other options are not used.

The methods `wfcQuickPrintInstructions::GetManualLayoutType` and `wfcQuickPrintInstructions::SetManualLayoutType` get and set the layout type when three views are being used in a manual layout. These methods are applicable only to views with layout type `wfcQPRINT_LAYOUT_MANUAL`.

You can set the layout type using the enumerated type `wfcQuickPrintManual3View`. The layout can be of the following types:

- `wfcQPRINTMANUAL_3VIEW_1_23VERT`
- `wfcQPRINTMANUAL_3VIEW_23_VERT1`

- wfcQPRINTMANUAL_3VIEW_123_HORIZ

Note

The general view location options are analogous to the images in the **Quick Drawing** dialog box under **View Layout**.

The methods wfcQuickPrintInstructions::GetOrientation and wfcQuickPrintInstructions::SetOrientation allow you to get and set the sheet orientation for the quick print operation. You can set the orientation using the enumerated type wfcQuickPrintOrientation:

- wfcQPRINT_ORIENTATION_PORTRAIT
- wfcQPRINT_ORIENTATION_LANDSCAPE

The methods wfcQuickPrintInstructions::GetPaperSize and wfcQuickPrintInstructions::SetPaperSize get and set the size of the print for the print operation using the object pfcPrintSize. For more information on print options, see the section Printer Options on page 563.

Use the method wfcQuickPrintInstructions::SetPrintFlatToScreen to set the ProBoolean flag to print the flat-to-screen annotations. The flat-to-screen annotations created at screen locations in the Creo graphics window are printed at their relative locations in the drawing. You can print flat-to-screen annotations such as notes, symbols, and surface finish symbols.

The methods wfcQuickPrintInstructions::GetProjectionViewLocations and wfcQuickPrintInstructions::SetProjectionViewLocations get and set the projected views to be included in the print operation. The methods define the projected views using the object wfcQuickPrintProjectionViewLocations. These methods are applicable only to views with layout type wfcQPRINT_LAYOUT_PROJ. You can set the following projections types using the enumerated type wfcQuickPrintProjectionViewLocation:

- wfcQPRINT_PROJ_TOP_VIEW
- wfcQPRINT_PROJ_RIGHT_VIEW
- wfcQPRINT_PROJ_LEFT_VIEW
- wfcQPRINT_PROJ_BOTTOM_VIEW
- wfcQPRINT_PROJ_BACK_NORTH
- wfcQPRINT_PROJ_BACK_EAST
- wfcQPRINT_PROJ_BACK_SOUTH

- `wfcQPRINT_PROJ_BACK_WEST`

Solid Operations

Method Introduced:

- **pfcSolid::CreateImportFeat**
- **wfcWSolid::ImportAsFeat**
- **wfcWSession::ImportAsModel**

The method `pfcSolid::CreateImportFeat` creates a new import feature in the solid and takes the following input arguments:

- *IntfData*—Specifies the source of data from which to create the import feature. It is given by the `pfcIntfDataSource` object. The type of source data that can be imported is given by the `pfcIntfType` class and can be of the following types:
  - `pfcINTF_NEUTRAL`
  - `pfcINTF_NEUTRAL_FILE`
  - `pfcINTF_IGES`
  - `pfcINTF_STEP`
  - `pfcINTF_VDA`
  - `pfcINTF_ICEM`
  - `pfcINTF_ACIS`
  - `pfcINTF_DXF`
  - `pfcINTF_CDRS`
  - `pfcINTF_STL`
  - `pfcINTF_VRML`
  - `pfcINTF_PARASOLID`
  - `pfcINTF_AI`
  - `pfcINTF_CATIA_PART`
  - `pfcINTF_UG`
  - `pfcINTF_PRODUCTVIEW`
  - `pfcINTF_CATIA_CGR`
  - `pfcINTF_JT`
  - `pfcINTF_INVENTOR_PART`
  - `pfcINTF_INVENTOR_ASM`

- pfcINTF_IBL
- pfcINTF_PTS
- pfcINTF_SE_PART
- pfcINTF_SE_SHEETMETAL_PART

- *CoordSys*—Specifies the pointer to a reference coordinate system. If this is NULL, the method uses the default coordinate system.
- *FeatAttr*—Specifies the attributes for creation of the new import feature given by the pfcImportFeatAttr object. If this pointer is NULL, the method uses the default attributes.

The method wfcWSolid::ImportAsFeat creates a new import feature in the solid. It takes the following input arguments:

- *IntfData*—Specifies the source of data using which the import feature is created. You can specify the data source as a file or interface data of only neutral type.
  - For a file, the data source is specified as a pfcIntfDataSource object. The type of source data that can be imported is given by the enumerated data type pfcIntfType.
  - For the interface data, the data source is specified as a wfcWIntfNeutral object. Refer to the section Extracting Interface Data for Neutral Files on page 558, for more information on wfcWIntfNeutral object type.
- *CoordSys*—Specifies the pointer to a reference coordinate system. If this is NULL, the method uses the default coordinate system.
- *CutOrAdd*—Specifies whether the import feature must be created as a cut or a protrusion. The default option is to add and has the value PRO_B_FALSE. If NULL, the method performs an add operation.
- *Profile*—Specifies the import profile path. It can be NULL.

The method wfcWSession::ImportAsModel imports a model in the solid. It takes the following arguments:

- *FileToImport*—Specifies the path of the file along with its name and extension.
- *NewModelType*—Specifies the type of the file to be imported.
- *Type*—Specifies the type of the model to be created. It can be a part, assembly, or drawing.
- *NewModelName*—Specifies a name for the imported model.
- *ModelRepType*—Specifies the representation type for the new imported model.
- *profile*—Specifies the import profile path. It can be NULL.

Note

The input argument *profile* allows you to include the import of Creo Elements/Direct containers, face parts, wire parts, and empty parts.

- *Filter*—Specifies the filter string in the form of callback method. The method determines the display and mapping of layers of the imported model. It can be `NULL`.

Window Operations

Method Introduced:

- **pfcWindow::ExportRasterImage**

The method `pfcWindow::ExportRasterImage` outputs a standard Creo raster output file.

Creating Import Features from Files

To create import features in Creo Parametric from external format files use the methods described in this section.

Methods Introduced:

- **pfcIntfNeutralFile::Create**
- **wfcIntfIBL::Create**
- **wfcIntfInventorAsm::Create**
- **wfcIntfInventorPart::Create**
- **wfcIntfPTS::Create**
- **wfcIntfSEdgePart::Create**
- **wfcIntfSEdgeSheetmetal::Create**

Use the method `pfcIntfNeutralFile::Create` to create a new object that represents the neutral file that creates an import feature.

Use the method `wfcIntfIBL::Create` to create a new object that represents the IBL file that creates an import feature.

Use the method `wfcIntfInventorAsm::Create` to create a new object that represents the Inventor Assembly file that creates an import feature.

Use the method `wfcIntfInventorPart::Create` to create a new object that represents the Inventor pat file that creates an import feature.

Use the method `wfcIntfPTS::Create` to create a new object that represents the PTS file that creates an import feature.

Use the method `wfcIntfSEdgePart::Create` to create a new object that represents the solid Edge part file that creates an import feature.

Use the method `wfcIntfSEdgeSheetmetal::Create` to create a new object that represents the solid Edge sheetmetal file that creates an import feature.

The output of the above methods is the value of the input argument *IntfData* passed to the method `wfcSolid::ImportAsFeat`.

32

Simplified Representations

Creo Object TOOLKIT C++ gives programmatic access to all the simplified representation functionality of Creo application. Create simplified representations either permanently or on the fly and save, retrieve, or modify them by adding or deleting items.

Overview

Using Creo Object TOOLKIT C++, you can create and manipulate assembly simplified representations just as you can using Creo application interactively.

Note

Creo Object TOOLKIT C++ supports simplified representation of assemblies only, not parts.

Simplified representations are identified by the `pfcSimRep` class. This class is a child of `pfcModelItem`, so you can use the methods dealing with `pfcModelItems` to collect, inspect, and modify simplified representations.

The information required to create and modify a simplified representation is stored in a class called `pfcSimpRepInstructions` which contains several data objects and fields, including:

- `String`—The name of the simplified representation
- `pfcSimpRepAction`—The rule that controls the default treatment of items in the simplified representation.
- `pfcSimpRepItem`—An array of assembly components and the actions applied to them in the simplified representation.

A `pfcSimpRepItem` is identified by the assembly component path to that item. Each `pfcSimpRepItem` has it's own `pfcSimpRepAction` assigned to it. `pfcSimpRepAction` is a visible data object that includes a field of type `pfcSimpRepActionType`. You can use the method `pfcSimpRepAction` to set the actions. To delete an existing item, you must set the action as NULL.

`pfcSimpRepActionType` is an enumerated type that specifies the possible treatment of items in a simplified representation. The possible values are as follows

Values	Action
`pfcSIMPREP_NONE`	No action is specified.
`pfcSIMPREP_REVERSE`	Reverse the default rule for this component (for example, include it if the default rule is exclude).
`pfcSIMPREP_INCLUDE`	Include this component in the simplified representation.
`pfcSIMPREP_EXCLUDE`	Exclude this component from the simplified representation.
`pfcSIMPREP_SUBSTITUTE`	Substitute the component in the simplified representation.
`pfcSIMPREP_GEOM`	Use only the geometrical representation of the component.
`pfcSIMPREP_GRAPHICS`	Use only the graphics representation of the component.

Values	Action
pfcSIMPREP_SYMB	Use the symbolic representation of the component.
pfcSIMPREP_BOUNDBOX	Use the boundary box representation of the component.
pfcSIMPREP_DEFENV	Use the default envelope representation of the component.
pfcSIMPREP_LIGHT_GRAPH	Use the light weight graphics representation of the component.
pfcSIMPREP_AUTO	Use the automatic representation of the component.

Retrieving Simplified Representations

Methods Introduced:

- **pfcBaseSession::RetrieveAssemSimpRep**
- **pfcBaseSession::RetrieveGeomSimpRep**
- **pfcBaseSession::RetrieveGraphicsSimpRep**
- **pfcBaseSession::RetrieveSymbolicSimpRep**
- **pfcRetrieveExistingSimpRepInstructions::Create**
- **wfcWSession::RetrieveDefaultEnvelopeSimprep**
- **wfcWSession::LoadModelRepresentation**

You can retrieve a named simplified representation from a model using the method `pfcBaseSession::RetrieveAssemSimpRep`, which is analogous to the Assembly mode option **Retrieve Rep** in the **SIMPLFD REP** menu. This method retrieves the object of an existing simplified representation from an assembly without fetching the generic representation into memory. The method takes two arguments, the name of the assembly and the simplified representation data.

To retrieve an existing simplified representation, pass an instance of `pfcRetrieveExistingSimpRepInstructions::Create` and specify its name as the second argument to this method. Creo retrieves that representation and any active submodels and returns the object to the simplified representation as a `pfcAssembly::Assembly` object.

You can retrieve geometry, graphics, and symbolic simplified representations into session using the methods `pfcBaseSession::RetrieveGeomSimpRep`, `pfcBaseSession::RetrieveGraphicsSimpRep`, and `pfcBaseSession::RetrieveSymbolicSimpRep` respectively. Like `pfcBaseSession::RetrieveAssemSimpRep`, these methods retrieve the simplified representation without bringing the master representation into memory. Supply the name of the assembly whose simplified representation is to be retrieved as the input parameter for these methods. The methods output the assembly. They do not display the simplified representation.

The method `wfcWSession::RetrieveDefaultEnvelopeSimprep` retrieves the simplified representation of the default envelope of an assembly in the session. This method is not supported for parts.

The method `wfcWSession::LoadModelRepresentation` retrieves the specified simplified representation of a model into memory.

Creating and Deleting Simplified Representations

Methods Introduced:

- **pfcCreateNewSimpRepInstructions::Create**
- **pfcSolid::CreateSimpRep**
- **pfcSolid::DeleteSimpRep**

To create a simplified representation, you must allocate and fill a `pfcSimpRepInstructions` object by calling the method `pfcCreateNewSimpRepInstructions::Create`. Specify the name of the new simplified representation as an input to this method. You should also set the default action type and add `pfcSimpRepItems` to the object.

To generate the new simplified representation, call `pfcSolid::CreateSimpRep`. This method returns the `pfcSimpRep` object for the new representation.

The method `pfcSolid::DeleteSimpRep` deletes a simplified representation from its model owner. The method requires only the `pfcSimpRep` object as input.

Extracting Information About Simplified Representations

Methods Introduced:

- **pfcSimpRep::GetInstructions**
- **pfcSimpRepInstructions::GetDefaultAction**
- **pfcCreateNewSimpRepInstructions::GetNewSimpName**
- **pfcSimpRepInstructions::GetIsTemporary**
- **pfcSimpRepInstructions::GetItems**
- **wfcWSimpRep::GetSimprepSubstitutionName**
- **wfcWSimpRep::IsSimpRepInstructionDefault**
- **wfcWSimpRep::SetSimpRepInstructionDefaultAction**

Given the object to a simplified representation, `pfcSimpRep::GetInstructions` fills out the `pfcSimpRepInstructions` object.

The `pfcSimpRepInstructions::GetDefaultAction`, `pfcCreateNewSimpRepInstructions::GetNewSimpName`, and `pfcSimpRepInstructions::GetIsTemporary` methods return the associated values contained in the `pfcSimpRepInstructions` object.

The method `pfcSimpRepInstructions::GetItems` returns all the items that make up the simplified representation.

The method `wfcWSimpRep::GetSimprepSubstitutionName()` returns the name of the substituted representation at the given assembly path. This method returns the name even when the substituted representation is deleted from the model at the given path.

The method `wfcWSimpRep::IsSimpRepInstructionDefault` determines if the specified simplified representation is the default representation for the owner model.

Use the method `wfcWSimpRep::SetSimpRepInstructionDefaultAction` to set the default action for the simplified representation using the enumerated type `pfcSimpRepActionType`.

Modifying Simplified Representations

Methods Introduced:

- **pfcSimpRep::GetInstructions**
- **pfcSimpRep::SetInstructions**
- **pfcSimpRepInstructions::SetDefaultAction**
- **pfcCreateNewSimpRepInstructions::SetNewSimpName**
- **pfcSimpRepInstructions::SetIsTemporary**
- **wfcWSimpRep::GetSimprepdataTempvalue**
- **wfcWSimpRep::SetSimprepdataName**

Using Creo Object TOOLKIT C++, you can modify the attributes of existing simplified representations. After you create or retrieve a simplified representation, you can make calls to the methods listed in this section to designate new values for the fields in the `pfcSimpRepInstructions` object.

To modify an existing simplified representation retrieve it and then get the `pfcSimpRepInstructions` object by calling `pfcSimpRep::GetInstructions`. If you created the representation programmatically within the same application, the

pfcSimpRepInstructions object is already available. Once you have modified the data object, reassign it to the corresponding simplified representation by calling the method pfcSimpRep::SetInstructions.

The method wfcWSimpRep::GetSimprepdataTempvalue returns a boolean value that specifies if the simplified representation is a temporary one.

Use the method wfcWSimpRep::SetSimprepdataName to set the name of the simplified representation in the pfcSimpRepInstructions object.

Adding Items to and Deleting Items from a Simplified Representation

Methods Introduced:

- **pfcSimpRepInstructions::SetItems**
- **pfcSimpRepItem::Create**
- **pfcSimpRep::SetInstructions**
- **pfcSimpRepReverse::Create**
- **pfcSimpRepInclude::Create**
- **pfcSimpRepExclude::Create**
- **pfcSimpRepSubstitute::Create**
- **pfcSimpRepGeom::Create**
- **pfcSimpRepGraphics::Create**
- **wfcWSimpRep::DeleteSimpRepInstructionItem**

You can add and delete items from the list of components in a simplified representation using Creo Object TOOLKIT C++. If you created a simplified representation using the option **Exclude** as the default rule, you would generate a list containing the items you want to include. Similarly, if the default rule for a simplified representation is **Include**, you can add the items that you want to be excluded from the simplified representation to the list, setting the value of the pfcSimpRepActionType to SIMPREP_EXCLUDE.

The method wfcWSimpRep::DeleteSimpRepInstructionItem deletes the specified item from pfcSimpRepInstructions object.

How to Add Items

1. Get the pfcSimpRepInstructions object, as described in the previous section.
2. Specify the action to be applied to the item with a call to one of following methods.

3. Initialize a `pfcSimpRepItem` object for the item by calling the method `pfcSimpRepItem::Create`.
4. Add the item to the `pfcSimpRepItem` sequence. Put the new `pfcSimpRepInstructions` using `pfcSimpRepInstructions::SetItems`.
5. Reassign the `pfcSimpRepInstructions` object to the corresponding `pfcSimpRep` object by calling `pfcSimpRep::SetInstructions`

How to Remove Items

Follow the procedure above, except remove the unwanted `pfcSimpRepItem` from the sequence.

Simplified Representation Utilities

Methods Introduced:

- **pfcModelItemOwner::ListItems**
- **pfcModelItemOwner::GetItemById**
- **pfcSolid::GetSimpRep**
- **pfcSolid::SelectSimpRep**
- **pfcSolid::ActivateSimpRep**
- **pfcSolid::GetActiveSimpRep**
- **wfcWSolid::ActivateAutomaticSimpRep**
- **wfcWSolid::ConvertAutomaticSimpRep**
- **wfcWSession::RetrieveAutomaticSimpRep**

This section describes the utility methods that relate to simplified representations.

The method `pfcModelItemOwner::ListItems` can list all of the simplified representations in a Solid.

The method `pfcModelItemOwner::GetItemById` initializes a `pfcSimpRep::SimpRep` object. It takes an integer id.

Note

Creo Object TOOLKIT C++ supports simplified representation of Assemblies only, not Parts.

The method `pfcSolid::GetSimpRep` initializes a `pfcSimpRep` object. The method takes the following arguments:

- *SimpRepname*— The name of the simplified representation in the solid. If you specify this argument, the method ignores the *rep_id*.

The method `pfcSolid::SelectSimpRep` creates a Creo menu to enable interactive selection. The method takes the owning solid as input, and outputs the object to the selected simplified representation. If you choose the **Quit** menu button, the method throws an exception `XToolkitUserAbort`

The methods `pfcSolid::GetActiveSimpRep` and `pfcSolid::ActivateSimpRep` enable you to find and get the currently active simplified representation, respectively. Given an assembly object, `pfcSolid::GetActiveSimpRep` returns the object to the currently active simplified representation. If the current representation is the master representation, the return is null.

The method `pfcSolid::ActivateSimpRep` activates the requested simplified representation.

To set a simplified representation to be the currently displayed model, you must also call `pfcModel::Display`.

The method `wfcWSolid::ActivateAutomaticSimpRep` activates a user-defined representation as automatic simplified representation.

The method `wfcWSolid::ConvertAutomaticSimpRep` converts a user-defined representation to automatic simplified representation while maintaining the excluded or substituted components in the representation.

Use the method `wfcWSession::RetrieveAutomaticSimpRep` to retrieve a user-defined simplified representation as automatic simplified representation. The input arguments follow:

- *AssemName*—Name of the assembly to retrieve.
- *FileType*—File type of the assembly to retrieve that is specified using by the enumerated data type `wfcMdlfileType`.
- *UDSrepName*—Name of the user defined simplified representation to retrieve as automatic.

Expanding Light Weight Graphics Simplified Representations

Methods Introduced:

- **wfcWAssembly::ExpandLightweightGraphicsSimprep**

Use the method `wfcWAssembly::ExpandLightweightGraphicsSimprep` to expand the light weight graphics representation to the specified level. The input arguments of this method are:

- TreeItem—Specify the model feature whose light weight graphic representation is to be expanded.
- LWGLevel—Specify the level up to which the expansion should take place using the enumerated type `LightweightGraphicsSimprepLevel`. The valid values for this enumerated type are:
  - `wfcLWG_SIMPREP_LEVEL_NEXT`—Specifies the expansion of the 3D Thumbnail to the next level.
  - `wfcLWG_SIMPREP_LEVEL_ALL`-Specifies the expansion of the 3D Thumbnail to all levels.

33

Asynchronous Mode

This chapter explains how to use Creo Object TOOLKIT C++ in Asynchronous Mode.

Overview

Asynchronous mode is a multiprocess mode in which the Creo Object TOOLKIT C++ application and Creo Parametric can perform concurrent operations. Unlike synchronous mode, the asynchronous mode uses remote procedure calls (rpc) as the means of communication between the application and Creo Parametric.

Another important difference between synchronous and asynchronous modes is in the startup of the Creo Object TOOLKIT C++ application. In synchronous mode, the application is started by Creo Parametric, based on information contained in the registry file. In asynchronous mode, the application is started independently of Creo Parametric and subsequently either starts or connects to a Creo Parametric process. The application can contain its own `main()` or `wmain()` function. Use `wmain()` if the application needs to receive command-line arguments as `wchar_t`, for example if the input contains non-usascii characters.

Note

An asynchronous application that starts Creo Parametric will not appear in the **Auxiliary Applications** dialog box.

The section How Creo Object TOOLKIT C++ Works on page 20 in Overview of Creo Object TOOLKIT C++ on page 14 chapter describes two modes—DLL and multiprocess (or “spawned”). These modes are synchronous modes in the sense that the Creo Object TOOLKIT C++ application and Creo Parametric do not perform operations concurrently. In spawn mode, each process can send a message to the other to ask for some operation, but each waits for a returning message that reports that the operation is complete. Control alternates between the two processes, one of which is always in a wait state.

Asynchronous mode applications operate with the same method of communication as spawn mode (multiprocess). The use of rpc in spawn mode causes this mode to perform significantly slower than DLL communications. For this reason, you should be careful not to apply asynchronous mode when it is not needed. Note that asynchronous mode is not the only mode in which your application can have explicit control over Creo Parametric.

An asynchronous application can also use the `user_initialize()` function, which will be called back by Creo Parametric when the application starts the connection. Note that in the asynchronous mode the callback to `user_initialize()` happens twice.

Setting up an Asynchronous Creo Object TOOLKIT Application

For your asynchronous application to communicate with Creo Parametric, you must set the environment variable `PRO_COMM_MSG_EXE` to the full path of the executable `pro_comm_msg`.

On Windows systems, set `PRO_COMM_MSG_EXE` in the **Environment** section of the **System** window that you access from the **Control Panel**.

Simple Asynchronous Mode

A simple asynchronous application does not implement a way to handle requests from Creo Parametric. Therefore, Creo Object TOOLKIT C++ cannot plant listeners to be notified when events happen in Creo Parametric. Consequently, Creo Parametric cannot invoke the methods that must be supplied when you add, for example, menu buttons to Creo Parametric.

Despite this limitation, a simple asynchronous mode application can be used to automate processes in Creo Parametric. The application may either start or connect to an existing Creo Parametric session, and may access Creo Parametric in interactive or in a non graphical, non interactive mode. When Creo Parametric is running with graphics, it is an interactive process available to the user.

When you design a Creo Object TOOLKIT C++ application to run in simple asynchronous mode, keep the following points in mind:

- The Creo Parametric process and the application perform operations concurrently.
- None of the application's listener methods can be invoked by Creo Parametric.

Simple asynchronous mode supports normal Creo Object TOOLKIT C++ methods but does not support callbacks. These considerations imply that the Creo Object TOOLKIT C++ application does not know the state (the current mode, for example) of the Creo Parametric process at any moment.

Starting and Stopping Creo Parametric

The following methods are used to start and stop Creo Parametric when using Creo Object TOOLKIT C++ applications.

Methods Introduced:

- **pfcAsyncConnection::Start**
- **pfcAsyncConnection::End**

A simple asynchronous application can spawn and connect to a Creo Parametric process with the method `pfcAsyncConnection::Start`. The Creo Parametric process listens for requests from the application and acts on the requests at suitable breakpoints, usually between commands.

Unlike applications running in synchronous mode, asynchronous applications are not terminated when Creo Parametric terminates. This is useful when the application needs to perform Creo Parametric operations intermittently, and therefore, must start and stop Creo Parametric more than once during a session.

The application can connect to or start only one Creo Parametric session at any time. If the Creo Object TOOLKIT C++ application spawns a second session, connection to the first session is lost.

To end any Creo Parametric process that the application is connected to, call the method `pfcAsyncConnection::End`.

Setting Up a Noninteractive Session

You can spawn a Creo Parametric session that is both noninteractive and nongraphical. In asynchronous mode, include the following strings in the Creo Parametric start or connect call to `pfcAsyncConnection::Start`:

- `-g:no_graphics`—Turn off the graphics display.
- `-i:rpc_input`—Causes Creo Parametric to expect input from your asynchronous application only.

> **Note**
>
> Both of these arguments are required, but the order is not important.

The syntax of the call for a noninteractive, nongraphical session is as follows:

```
pfcAsyncConnection::Start
("pro -g:no_graphics -i:rpc_input",<text_dir>);
```

where `pro` is the command to start Creo Parametric.

Connecting to a Creo Parametric Process

Methods Introduced:

- **pfcAsyncConnection::Connect**
- **pfcAsyncConnection::GetActiveConnection**
- **pfcAsyncConnection::Disconnect**

A simple asynchronous application can also connect to a Creo Parametric process that is already running on a local computer. The method `pfcAsyncConnection::Connect` performs this connection. This method fails to connect if multiple Creo Parametric sessions are running. If several versions of Creo Parametric are running on the same computer, try to connect by specifying user and display parameters. However, if several versions of Creo Parametric are running in the same user and display parameters, the connection may not be possible.

`pfcAsyncConnection::GetActiveConnection` returns the current connection to a Creo Parametric session.

To disconnect from a Creo Parametric process, call the method `pfcAsyncConnection::Disconnect`. This method can be called only if you used the method `pfcAsyncConnection::Connect` to get the connection.

The connection to a Creo Parametric process uses information provided by the name service daemon. The name service daemon accepts and supplies information about the processes running on the specified hosts. The application manager, for example, uses the name service when it starts up Creo Parametric and other processes. The name service daemon is set up as part of the Creo Parametric installation.

Connecting Via Connection ID

Methods Introduced:

- **pfcAsyncConnection::GetConnectionId**
- **pfcConnectionId::GetExternalRep**
- **pfcBaseSession::GetConnectionId**
- **pfcConnectionId::Create**
- **pfcAsyncConnection::ConnectById**

Each Creo Parametric process maintains a unique identity for communications purposes. Use this ID to reconnect to a Creo Parametric process.

The method `pfcAsyncConnection::GetConnectionId` returns a data structure containing the connection ID.

If the connection id must be passed to some other application the method `pfcConnectionId::GetExternalRep` provides the string external representation for the connection ID.

The method `pfcBaseSession::GetConnectionId` provides access to the asynchronous connection ID for the current Creo Parametric session. This ID can be passed to any asynchronous mode application that needs to connect to the current session of Creo Parametric.

The method `pfcConnectionId::Create` takes a string representation and creates a `ConnectionId` data object. The method `pfcAsyncConnection::ConnectById` connects to Creo Parametric at the specified connection ID.

Note

Connection IDs are unique for each Creo Parametric process and are not maintained after you quit Creo Parametric.

Status of a Creo Parametric Process

Method Introduced:

- **pfcAsyncConnection::IsRunning**

To find out whether a Creo Parametric process is running, use the method `pfcAsyncConnection::IsRunning`.

Getting the Session Object

Method Introduced:

- **pfcAsyncConnection::GetSession**

The method `pfcAsyncConnection::GetSession` returns the session object representing the Creo Parametric session. Use this object to access the contents of the Creo Parametric session. See the Session Objects on page 53 chapter for additional information.

Full Asynchronous Mode

Full asynchronous mode is identical to the simple asynchronous mode except in the way the Creo Object TOOLKIT C++ application handles requests from Creo Parametric. In simple asynchronous mode, it is not possible to process these requests. In full asynchronous mode, the application implements a control loop that ‘listens’ for messages from Creo Parametric. As a result, Creo Parametric can call functions in the application, including callback functions for menu buttons and notifications.

Note

Using full asynchronous mode requires starting or connecting to Creo Parametric using the methods described in the previous sections. The difference is that the application must provide an event loop to process calls from menu buttons and listeners.

Methods Introduced:

- **pfcAsyncConnection::EventProcess**
- **pfcAsyncConnection::WaitForEvents**
- **pfcAsyncConnection::InterruptEventProcessing**
- **pfcAsyncActionListener::OnTerminate**

The control loop of an application running in full asynchronous mode must contain a call to the method `pfcAsyncConnection::EventProcess`, which takes no arguments. This method allows the application to respond to messages sent from Creo Parametric. For example, if the user selects a menu button that is added by your application, `pfcAsyncConnection::EventProcess` processes the call to your listener and returns when the call completes. For more information on listeners and adding menu buttons, see the Session Objects on page 53 chapter.

The method `pfcAsyncConnection::WaitForEvents` provides an alternative to the development of an event processing loop in a full asynchronous mode application. Call this function to have the application wait in a loop for events to be passed from Creo Parametric. No other processing takes place while the application is waiting. The loop continues until `pfcAsyncConnection::InterruptEventProcessing` is called from a Creo Object TOOLKIT C++ callback action, or until the application detects the termination of Creo Parametric.

It is often necessary for your full asynchronous application to be notified of the termination of the Creo Parametric process. In particular, your control loop need not continue to listen for Creo Parametric messages if Creo Parametric is no longer running.

An `AsyncConnection` object can be assigned an Action Listener to bind a termination action that is executed upon the termination of Creo Parametric. The method `pfcAsyncActionListener::OnTerminate` handles the termination that you must override. It sends a member of the class `pfcAsyncConnection::TerminationStatus`, which is one of the following:

- `pfcTERM_EXIT`—Normal exit (the user clicks **Exit** on the menu).

- `pfcTERM_ABNORMAL`—Quit with error status.
- `pfcTERM_SIGNAL`—Fatal signal raised.

Your application can interpret the termination type and take appropriate action. For more information on Action Listeners, see the Action Listeners on page 509 chapter.

Troubleshooting Asynchronous Creo Object TOOLKIT

General Problems

pfcXToolkitNotFound exception on the first call to pfcAsyncConnection::Start on Windows.

Make sure your Creo Parametric command is correct. If it's not a full path to a script/executable, make sure `$PATH` is set correctly. Try full path in the command: if it works, then your `$PATH`is incorrect.

pfcXToolkitGeneralError or pfcXToolkitCommError on the first call to pfcAsyncConnection::Start or pfcAsyncConnection::Connect

- Make sure the environment variable `PRO_COMM_MSG_EXE` is set to full path to `pro_comm_msg`, including file name, including `.exe` on Windows.
- Make sure the environment variable `PRO_DIRECTORY` is set to Creo Parametric installation directory.
- Make sure name service () is running.

pfcAsyncConnection::Start hangs, even though Creo Parametric already started.

Make sure name service () is also started with Creo Parametric. Open **Task Manager** and look for `nmsd.exe` in the process listing.

34

Task Based Application Libraries

Applications created using different Creo API products are interoperable. These products use the Creo application as the medium of interaction, eliminating the task of writing native-platform specific interactions between different programming languages.

Application interoperability allows Creo Object TOOLKIT C++ applications to call into Creo TOOLKIT from areas not covered in the native interface. It allows you to put a Creo Object TOOLKIT C++ front end on legacy Creo TOOLKIT applications and also allows you to use Creo Object TOOLKIT C++ applications and listeners in conjunction with a asynchronous Creo Object TOOLKIT C++ application.

Creo Object TOOLKIT C++ can call Creo Parametric web pages belonging to Web.Link, and functions in Creo TOOLKIT DLLs. Creo Object TOOLKIT C++ synchronous applications can also register tasks for use by other applications.

Managing Application Arguments

Creo Object TOOLKIT C++ passes application data to and from tasks in other applications as members of a sequence of `pfcArgument` objects. Application arguments consist of a label and a value. The value may be of any one of the following types:

- Integer
- Double
- Boolean
- ASCII string (a non-encoded string, provided for compatibility with arguments provided from C applications)
- String (a fully encoded string)
- `pfcSelection` (a selection of an item in a Creo session)
- `pfcTransform3D` (a coordinate system transformation matrix)

Methods Introduced:

- **pfcCreateIntArgValue**
- **pfcCreateDoubleArgValue**
- **pfcCreateBoolArgValue**
- **pfcCreateASCIIStringArgValue**
- **pfcCreateStringArgValue**
- **pfcCreateSelectionArgValue**
- **pfcCreateTransformArgValue**
- **pfcArgValue::Getdiscr**
- **pfcArgValue::GetIntValue**
- **pfcArgValue::SetIntValue**
- **pfcArgValue::GetDoubleValue**
- **pfcArgValue::SetDoubleValue**
- **pfcArgValue::GetBoolValue**
- **pfcArgValue::SetBoolValue**
- **pfcArgValue::GetASCIIStringValue**
- **pfcArgValue::SetASCIIStringValue**
- **pfcArgValue::GetStringValue**
- **pfcArgValue::SetStringValue**
- **pfcArgValue::GetSelectionValue**
- **pfcArgValue::SetSelectionValue**

- **pfcArgValue::GetTransformValue**
- **pfcArgValue::SetTransformValue**

The class `pfcArgValue` contains one of the seven types of values. Creo Object TOOLKIT C++ provides different methods to create each of the seven types of argument values.

The method `pfcArgValue::Getdiscr` returns the type of value contained in the argument value object.

Use the methods listed above to access and modify the argument values.

Modifying Arguments

Methods Introduced:

- **pfcArgument::Create**
- **pfcArguments::create**
- **pfcArgument::GetLabel**
- **pfcArgument::SetLabel**
- **pfcArgument::GetValue**
- **pfcArgument::SetValue**

The method `pfcArgument::Create` creates a new argument. Provide a name and value as the input arguments of this method.

The method `pfcArguments::create` creates a new empty sequence of task arguments.

The method `pfcArgument::GetLabel` returns the label of the argument. The method `pfcArgument::Argument.SetLabel` sets the label of the argument.

The method `pfcArgument::GetValue` returns the value of the argument. The method `pfcArgument::SetValue` sets the value of the argument.

Launching a Creo TOOLKIT DLL

The methods described in this section enable a Creo Object TOOLKIT C++ user to register and launch a Creo TOOLKIT DLL from a Creo Object TOOLKIT C++ application.

Methods Introduced:

- **pfcBaseSession::LoadProToolkitDll**
- **pfcBaseSession::LoadProToolkitLegacyDll**
- **pfcBaseSession::GetProToolkitDll**

- **pfcDll::ExecuteFunction**
- **pfcDll::GetId**
- **pfcDll::IsActive**
- **pfcDll::Unload**

Use the method `pfcBaseSession::LoadProToolkitDll` to register and start a Creo TOOLKIT DLL. The input parameters of this method are similar to the fields of a registry file and are as follows:

- *ApplicationName*—The name of the application to initialize.
- *DllPath*—The full path to the DLL binary file.
- *TextPath*—The path to the application's message and user interface text files.
- *UserDisplay*—Set this parameter to `true` to register the application in the Creo user interface and to see error messages if the application fails. If this parameter is `false`, the application will be invisible to the user.

The application's `user_initialize()` function is called when the application is started. The method returns a handle to the loaded Creo TOOLKIT DLL.

In order to register and start a legacy Pro/TOOLKIT DLL that is not Unicode-compliant, use the method `pfcBaseSession::LoadProToolkitLegacyDll`. This method conveys to Creo Parametric that the loaded DLL application is not Unicode-compliant and built in the pre-Wildfire 4.0 environment. It takes the same input parameters as the earlier method `pfcBaseSession::LoadProToolkitDll`.

Use the method `pfcBaseSession::GetProToolkitDll` to obtain a Creo TOOLKIT DLL handle. Specify the *Application_Id*, that is, the DLL's identifier string as the input parameter of this method. The method returns the DLL object or null if the DLL was not in session. The *Application_Id* can be determined as follows:

- Use the function `ProToolkitDllIdGet()` within the DLL application to get a string representation of the DLL application. Pass `NULL` to the first argument of `ProToolkitDllIdGet()` to get the string identifier for the calling application.
- Use the `Get` method for the `Id` attribute in the DLL interface. The method `pfcDll::GetId` returns the DLL identifier string.

Use the method `pfcDll::ExecuteFunction` to call a properly designated function in the Creo TOOLKIT DLL library. The input parameters of this method are:

- *FunctionName*—Name of the function in the Creo TOOLKIT DLL application.
- *InputArguments*—Input arguments to be passed to the library function.

The method returns an object of interface `pfcFunctionReturn`. This interface contains data returned by a Creo TOOLKIT function call. The object contains the return value, as integer, of the executed function and the output arguments passed back from the function call.

The method `pfcDll::IsActive` determines whether a Creo TOOLKIT DLL previously loaded by the method `pfcBaseSession::LoadProToolkitDll` is still active.

The method `pfcDll::Unload` is used to shutdown a Creo TOOLKIT DLL previously loaded by the method `pfcBaseSession::LoadProToolkitDll` and the application's `user_terminate()` function is called.

Launching Tasks from Creo Object TOOLKIT C++ Task Libraries

The methods described in this section allow you to launch tasks from a predefined Creo Object TOOLKIT C++ task library.

Methods Introduced:

- **pfcBaseSession::StartJLinkApplication**
- **pfcJLinkApplication::ExecuteTask**
- **pfcJLinkApplication::IsActive**
- **pfcJLinkApplication::Stop**

Use the method `pfcBaseSession::StartJLinkApplication` to start a Creo Object TOOLKIT C++ application. The input parameters of this method are similar to the fields of a registry file and are as follows:

- *ApplicationName*—Assigns a unique name to this Creo Object TOOLKIT C++ application.
- *ClassName*—Specifies the name of the Creo Object TOOLKIT C++ class that contains the Creo Object TOOLKIT C++ application's start and stop method. This should be a fully qualified Creo Object TOOLKIT C++ package and class name.
- *StartMethod*—Specifies the start method of the Creo Object TOOLKIT C++ application.
- *StopMethod*—Specifies the stop method of the Creo Object TOOLKIT C++ application.
- *AdditionalClassPath*—Specifies the locations of packages and classes that must be loaded when starting this Creo Object TOOLKIT C++ application. If this parameter is specified as null, the default classpath locations are used.

- *TextPath*—Specifies the application text path for menus and messages. If this parameter is specified as null, the default text locations are used.
- *UserDisplay*—Specifies whether to display the application in the **Auxiliary Applications** dialog box in the Creo application.

Upon starting the application, the static `start()` method is invoked. The method returns a `pfcJLinkApplication` referring to the Creo Object TOOLKIT C++ application.

The method `pfcJLinkApplication::ExecuteTask` calls a registered task method in a Creo Object TOOLKIT C++ application. The input parameters of this method are:

- Name of the task to be executed.
- A sequence of name value pair arguments contained by the interface `pfcArguments`.

The method outputs an array of output arguments. These arguments are returned by the task's implementation of the `pfcJLinkTaskListener::OnExecute` call back method.

The method `pfcJLinkApplication::IsActive` returns a `True` value if the application specified by the `pfcJLinkApplication` object is active.

The method `pfcJLinkApplication::Stop` stops the application specified by the `pfcJLinkApplication` object. This method activates the application's static `Stop()` method.

35

Graphics

This chapter covers Creo Object TOOLKIT C++ graphics including displaying lists, displaying text, using the mouse and object displays.

Overview

The methods described in this section allow you to draw temporary graphics in a display window. Methods that are identified as 2D are used to draw entities (arcs, polygons, and text) in screen coordinates. Other entities may be drawn using the current model's coordinate system or the screen coordinate system's lines, circles, and polylines. Methods are also included for manipulating text properties and accessing mouse inputs.

Getting Mouse Input

The following methods are used to read the mouse position in screen coordinates with the mouse button depressed. Each method outputs the position and an enumerated type description of which mouse button was pressed when the mouse was at that position. These values are contained in the interface `pfcMouseStatus`. The enumerated values are defined in `pfcMouseButton`.

Methods Introduced:

- **pfcSession::UIGetNextMousePick**
- **pfcSession::UIGetCurrentMouseStatus**

The method `pfcSession::UIGetNextMousePick` returns the mouse position when you press a mouse button. The input argument is the mouse button that you expect the user to select.

The method `pfcSession::UIGetCurrentMouseStatus` returns a value whenever the mouse is moved or a button is pressed. With this method a button does not have to be pressed for a value to be returned. You can use an input argument to flag whether or not the returned positions are snapped to the window grid.

Drawing a Mouse Box

This method allows you to draw a mouse box.

Method Introduced:

- **pfcSession::UIPickMouseBox**

The method `pfcSession::UIPickMouseBox` draws a dynamic rectangle from a specified point in screen coordinates to the current mouse position until the user presses the left mouse button. The return value for this method is of the type `pfcOutline3D`.

You can supply the first corner location programmatically or you can allow the user to select both corners of the box.

Cosmetic Properties

You can enhance your model using Creo Object TOOLKIT C++ methods that change the surface properties, or set different light sources. The following section describes these methods in detail.

Surface Properties

Methods Introduced:

- **wfcWSelection::GetVisibleAppearance**
- **wfcWSelection::SetVisibleAppearance**
- **wfcAppearance::Create**
- **wfcAppearance::GetAmbient**
- **wfcAppearance::SetAmbient**
- **wfcAppearance::GetDescription**
- **wfcAppearance::SetDescription**
- **wfcAppearance::GetDiffuse**
- **wfcAppearance::SetDiffuse**
- **wfcAppearance::GetHighlight**
- **wfcAppearance::SetHighlight**
- **wfcAppearance::GetHighlightColor**
- **wfcAppearance::SetHighlightColor**
- **wfcAppearance::GetKeywords**
- **wfcAppearance::SetKeywords**
- **wfcAppearance::GetLabel**
- **wfcAppearance::SetLabel**
- **wfcAppearance::GetName**
- **wfcAppearance::SetName**
- **wfcAppearance::GetRGBColor**
- **wfcAppearance::SetRGBColor**
- **wfcAppearance::GetReflection**
- **wfcAppearance::SetReflection**
- **wfcAppearance::GetShininess**
- **wfcAppearance::SetShininess**
- **wfcAppearance::GetTransparency**

- **wfcAppearance::SetTransparency**
- **wfcWSurface::GetNextSurface**
- **wfcWSelection::SetVisibleTextures**
- **wfcWSelection::GetVisibleTextures**
- **wfcTexture::Create**
- **wfcTexture::GetType**
- **wfcTexture::SetType**
- **wfcTexture::GetFilePath**
- **wfcTexture::SetFilePath**
- **wfcTexture::GetPlacement**
- **wfcTexture::SetPlacement**
- **wfcTexturePlacement::Create**
- **wfcTexturePlacement::GetTextureProjectionType**
- **wfcTexturePlacement::SetTextureProjectionType**
- **wfcTexturePlacement::GetLocalSys**
- **wfcTexturePlacement::SetLocalSys**
- **wfcTexturePlacement::GetHorizontalOffset**
- **wfcTexturePlacement::SetHorizontalOffset**
- **wfcTexturePlacement::GetVerticalOffset**
- **wfcTexturePlacement::SetVerticalOffset**
- **wfcTexturePlacement::GetRotate**
- **wfcTexturePlacement::SetRotate**
- **wfcTexturePlacement::GetHorizontalScale**
- **wfcTexturePlacement::SetHorizontalScale**
- **wfcTexturePlacement::GetVerticalScale**
- **wfcTexturePlacement::SetVerticalScale**
- **wfcTexturePlacement::GetBumpHeight**
- **wfcTexturePlacement::SetBumpHeight**
- **wfcTexturePlacement::GetDecalIntensity**
- **wfcTexturePlacement::SetDecalIntensity**
- **wfcTexturePlacement::GetHorizontalFlip**
- **wfcTexturePlacement::SetHorizontalFlip**
- **wfcTexturePlacement::GetVerticalFlip**

- **wfcTexturePlacement::SetVerticalFlip**
- **wfcWSession::IsTextureFileEmbedded**
- **wfcWSession::GetTextureFilePath**

The methods `wfcWSelection::GetVisibleAppearance` and `wfcWSelection::SetVisibleAppearance` enable you to retrieve and set the appearance properties for the specified model using an object of the class `wfcAppearance`.

Note

To set the default surface appearance properties, pass the value NULL to the `AppearanceProps` parameter of the `wfcWSelection::SetVisibleAppearance` method.

The class `wfcAppearance` is an interface to the appearance properties and contains the methods described below.

Use the method `wfcAppearance::Create` to create an instance of the `wfcAppearance` object.

- `Ambient`—Specifies the indirect, scattered light the model receives from its surroundings. The valid range is from 0.0 to 1.0.
- `Description`—Specifies the model description.
- `Diffuse`—Specifies the reflected light that comes from directional, point, or spot lights. The valid range is from 0.0 to 1.0.
- `Highlight`—Specifies the intensity of the light reflected from a highlighted surface area. The valid range is from 0.0 to 1.0.
- `HighlightColor`—Specifies the highlight color, in terms of red, green, and blue. The valid range is from 0.0 to 1.0.
- `Keywords`—Specifies the keywords in the specified model.
- `Label`—Specifies the labels in the model.
- `Name`— Specifies the name of the model.
- `RGBColor`— Specifies the color, in the RGB format. The valid range is from 0.0. to 1.0.
- `Reflection`—Specifies how reflective the surface is. The valid range is from 0, that is dull to 100, that is shiny.
- `Shininess`—Specifies the properties of a highlighted surface area. A plastic model would have a lower shininess value, while a metallic model would have a higher value. The valid range is from 0.0. to 1.0.
- `Transparency`— Specifies the transparency value, which is between 0 that is completely opaque to 1.0 that is completely transparent.

Use the methods `wfcAppearance::GetAmbient` and `wfcAppearance::SetAmbient` to get and set the ambience value.

Use the methods `wfcAppearance::GetDescription` and `wfcAppearance::SetDescription` to get and set the description for the specified model.

Use the methods `wfcAppearance::GetDiffuse` and `wfcAppearance::SetDiffuse` to get and set the value for the diffused light.

Use the methods `wfcAppearance::GetHighlight` and `wfcAppearance::SetHighlight` to get and set the intensity of the highlighted light.

Use the methods `wfcAppearance::GetHighlightColor` and `wfcAppearance::SetHighlightColor` to get and set the highlight color.

Use the methods `wfcAppearance::GetKeywords` and `wfcAppearance::SetKeywords` to get and set the keywords.

Use the methods `wfcAppearance::GetLabel` and `wfcAppearance::SetLabel` to get and set labels in the specified model.

Use the methods `wfcAppearance::GetName` and `wfcAppearance::SetName` to get and set the model name for the specified model.

Use the methods `wfcAppearance::GetRGBColor` and `wfcAppearance::SetRGBColor` to get and set the RGBColor.

Use the methods `wfcAppearance::GetReflection` and `wfcAppearance::SetReflection` to get and set the reflectivity of the surface.

Use the methods `wfcAppearance::GetShininess` and `wfcAppearance::SetShininess` to get and set the properties of a highlighted surface area.

Use the methods `wfcAppearance::GetTransparency` and `wfcAppearance::SetTransparency` to get and set the transparency value.

Use the methods `wfcWSelection::GetVisibleTextures` and `wfcWSelection::SetVisibleTextures` to get and set the properties related to the surface texture respectively using an object of the class `wfcTexture`. The class `wfcTexture` is an interface to the texture property and contains the methods described below.

Use the method `wfcTexture::Create` to create an instance of the `wfcTexture` object. The parameters of this method are given below:

- `Type`—Specifies the type of the texture.
- `FilePath`—Specifies the full path to the texture.

- `Placement`—Specifies the properties related to the placement of surface texture.

Use the methods `wfcTexture::GetType` and `wfcTexture::SetType` to get and set the type of the texture using the enumerated type `wfcTextureType`. The valid values of the enumerated type `wfcTextureType` are as follows:

- `NULL_TEXTURE`—Specifies that no texture map is assigned.
- `BUMP_TEXTURE`—Specifies the single-channel texture maps that represent height fields. This texture results in shaded surface and has a wrinkled or irregular appearance.
- `COLOR_TEXTURE`—Specifies the surface color such as woodgrain, geometric patterns, and pictures applied to the texture.
- `DECAL_TEXTURE`—Specifies the specialized texture maps such as company logos or text that are applied to a surface. A decal is a texture with an alpha or transparency mask.

Use the methods `wfcTexture::GetFilePath` and `wfcTexture::SetFilePath` to get and set the full path to the texture map.

Use the methods `wfcTexture::GetPlacement` and `wfcTexture::SetPlacement` to get and set the sequence of placements properties using an object of the class `wfcTexturePlacement`. The parameters of the class `wfcTexturePlacement` are as follows:

Use the method `wfcTexturePlacement::Create` to create an instance of the object `wfcTexturePlacement`. The parameters of this method are as follows:

- `TextureProjectionType`—Specifies the projection type of the texture on the selected surface or surfaces.
- `LineEnvelope`—Specifies the coordinate system that defines the direction for the planar projection, or the center for the other projection types.
- `HorizontalOffset`—Specifies the percentage of horizontal shift of the texture map on the surface.
- `VerticalOffset`—Specifies the percentage of vertical shift of the texture map on the surface.
- `Rotate`—Specifies the angle to rotate the texture map on the surface.
- `HorizontalScale`—Specifies the horizontal scaling of the texture map.
- `VerticalScale`—Specifies the vertical scaling of the texture map.
- `BumpHeight`—Specifies the height of the bump on the surface of the texture map.
- `DecalIntensity`—Specifies the alpha or transparency mask intensity on the surface.
- `HorizontalFlip`—Specifies that the texture map on the surface should be flipped horizontally.

- `VerticalFlip`—Specifies that the texture map on the surface should be flipped vertically.

Use the methods `wfcTexturePlacement::GetTextureProjectionType` and `wfcTexturePlacement::SetTextureProjectionType` to get and set the texture projection type using the enumerated type `wfcTextureProjectionType`.

Use the methods `wfcTexturePlacement::GetLocalSys` and `wfcTexturePlacement::SetLocalSys` to get and set the line envelope.

Use the methods `wfcTexturePlacement::GetHorizontalOffset` and `wfcTexturePlacement::SetHorizontalOffset` to get and set the horizontal offset.

Use the methods `wfcTexturePlacement::GetVerticalOffset` and `wfcTexturePlacement::SetVerticalOffset` to get and set the vertical offset.

Use the methods `wfcTexturePlacement::GetRotate` and `wfcTexturePlacement::SetRotate` to get and set the rotating angle.

Use the methods `wfcTexturePlacement::GetHorizontalScale` and `wfcTexturePlacement::SetHorizontalScale` to get and set the horizontal scale.

Use the methods `wfcTexturePlacement::GetVerticalScale` and `wfcTexturePlacement::SetVerticalScale` to get and set the vertical scale.

Use the methods `wfcTexturePlacement::GetBumpHeight` and `wfcTexturePlacement::SetBumpHeight` to get and set the height of the bump.

Use the methods `wfcTexturePlacement::GetDecalIntensity` and `wfcTexturePlacement::SetDecalIntensity` to get and set the decal intensity.

Use the methods `wfcTexturePlacement::GetHorizontalFlip` and `wfcTexturePlacement::SetHorizontalFlip` to get and set the horizontal flip.

Use the methods `wfcTexturePlacement::GetVerticalFlip` and `wfcTexturePlacement::SetVerticalFlip` to get and set the vertical flip.

The method `wfcWSession::IsTextureFileEmbedded` checks if the specified texture, decal, or bump map file is embedded in the model.

The method `wfcWSession::GetTextureFilePath` returns the full path of the specified texture, decal, or bump map file and loads the file from the path. If you specify the input argument *CreateTempFile* as true, the method creates a temporary copy of the texture file.

Item Properties

Methods Introduced:

- **wfcItemAppearanceAndTextures::GetAppearance**
- **wfcItemAppearanceAndTextures::GetTextures**
- **wfcItemAppearanceAndTextures::GetSelection**

The method `wfcItemAppearanceAndTextures::GetAppearance` retrieves the appearance properties for the specified item using the `wfcAppearance` object.

The method `wfcItemAppearanceAndTextures::GetTextures` retrieves the texture properties for the specified item using the `wfcTextures` object.

Use the method `wfcItemAppearanceAndTextures::GetSelection` to retrieve the selection handle of the item using the `pfcSelection` object.

For more information, see Surface Properties on page 603.

Setting Light Sources

The methods in this section allow you to set the light sources to render a model in the specified graphics window.

Methods Introduced:

- **wfcWWindow::GetLightInstructions**
- **wfcWWindow::SetLightInstructions**
- **wfcLightSourceInstruction::Create**
- **wfcLightSourceInstruction::GetName**
- **wfcLightSourceInstruction::SetName**
- **wfcLightSourceInstruction::GetType**
- **wfcLightSourceInstruction::SetType**
- **wfcLightSourceInstruction::GetColor**
- **wfcLightSourceInstruction::SetColor**
- **wfcLightSourceInstruction::GetPosition**
- **wfcLightSourceInstruction::SetPosition**

- **wfcLightSourceInstruction::GetDirection**
- **wfcLightSourceInstruction::SetDirection**
- **wfcLightSourceInstruction::GetSpreadAngle**
- **wfcLightSourceInstruction::SetSpreadAngle**
- **wfcLightSourceInstruction::GetCastShadows**
- **wfcLightSourceInstruction::SetCastShadows**
- **wfcLightSourceInstruction::GetIsActive**
- **wfcLightSourceInstruction::SetIsActive**

The method `wfcWWindow::GetLightInstructions` retrieves the information about the light sources in the specified Creo window. Use the method `wfcWWindow::SetLightInstructions` to set the parameters for a light source in the specified window.

Use the method `wfcLightSourceInstruction::Create` to create an instance of the object `wfcLightSourceInstruction` that contains all the information related to light sources. The parameters of this method are as follows:

- *inName*—Specifies the name of the light source.
- *inType*—Specifies the type of light using the enumerated data type `wfcLightType`. The light types are —ambient, direction, point, spot, or HDRI.
- *inColor*—Specifies the color of the light in red, green, and blue values.
- *inIsActive*—Specifies if the light source is active.

Use the methods `wfcLightSourceInstruction::GetName` and `wfcLightSourceInstruction::SetName` to get and set the name of the light source.

Use the methods `wfcLightSourceInstruction::GetType` and `wfcLightSourceInstruction::SetType` to get and set the type of light source using the enumerated data type `wfcLightType`. The light types are:

- `wfcLIGHT_AMBIENT`—Ambient light illuminates all the surfaces equally. The position of the light in the room has no effect on the rendering. Ambient light exists by default and cannot be created or set in Creo application.
- `wfcLIGHT_DIRECTION`—Directional light casts parallel light rays illuminating all surfaces at the same angle, regardless of the position of the model.
- `wfcLIGHT_POINT`—Point light is similar to the light from a bulb in a room where the light radiates from the center. The light reflected off a surface varies, depending on the surface position with respect to the light.
- `wfcLIGHT_SPOT`—Spotlight is similar to a light bulb but with its rays confined within a cone called the spot angle.

- `wfcLIGHT_HDRI`—High Dynamic Range Image (HDRI) is used to illuminate the model. The light type cannot be set in Creo application.

Use the methods `wfcLightSourceInstruction::GetColor` and `wfcLightSourceInstruction::SetColor` to get and set the color of light as red, green, and blue values.

The method `wfcLightSourceInstruction::GetIsActive` checks if the light source is active. Use the method `wfcLightSourceInstruction::SetIsActive` to set the light source as active.

Use the methods `wfcLightSourceInstruction::GetPosition` and `wfcLightSourceInstruction::SetPosition` to get and set the position of the light source. Specify the position only for point and spot lights.

Use the methods `wfcLightSourceInstruction::GetDirection` and `wfcLightSourceInstruction::SetDirection` to get and set the direction of the light source. Specify the direction only for direction and spot lights.

Use the methods `wfcLightSourceInstruction::GetSpreadAngle` and `wfcLightSourceInstruction::SetSpreadAngle` to get and set the angle to which the light is spread in radians. Specify the spread angle only for spot light.

Use the method `wfcLightSourceInstruction::GetCastShadows` to identify if the light source casts a shadow. The `wfcLightSourceInstruction::SetCastShadows` sets the light source to cast a shadow. Shadows can be applied only in Creo Render Studio. Refer to the Creo Render Studio Help for more information.

Example 1: Adding a Light Source

The sample code in `OTKXLights.cxx` located at `<creo_otk_loadpoint_app>/otk_examples/otk_examples_graphics` shows how to add a light source using the Creo Object TOOLKIT C++ methods. It also shows how to extract and display information for a light source.

Graphics Colors

Creo application uses several predefined colors in its color map which correspond to the system colors presented to the user through the user interface. The names of the types generally indicate what they are used for in the Creo application.

Note

PTC reserves the right to change both the definitions of the predefined colormap and also of the assignment of entities to members of the color map as required by improvements to the user interface. PTC recommends not relying on the predefined RGB color for displaying of Creo Object TOOLKIT C++ entities or graphics, and also recommends against relying on the relationship between certain colormap entries and types of entities. The following sections describe how to construct your application so that it does not rely on potentially variant properties in Creo application.

Setting Colors to Match Existing Entities

Method Introduced:

- **wfcWDisplay::GetColorByObjectType**

The method `wfcWDisplay::GetColorByObjectType` returns the standard color used to display the specified entity in the Creo application.

The association between objects and colormap entries may change in a new release of Creo. This causes the association between the application entities and the Creo entities to be lost. You can use this method to get the display color of the entity.

Version of Color Map

Methods Introduced:

- **wfcWDisplay::GetColorRGBVersion**
- **wfcWDisplay::SetColorRGBVersion**

The method `wfcWDisplay::GetColorRGBVersion` returns the version of the Creo color map. Use the method `wfcWDisplay::SetColorRGBVersion` to set the version of the color map using the enumerated data type `wfcColorRGBVersion`. To specify a color map from releases prior to the Pro/ENGINEER Wildfire release specify the value as:

- `wfcCOLORRGB_STANDARD_VERSION`—Specifies the `appearance.dmt` files for the color map.
- `wfcCOLORRGB_PRE_WILDFIRE_VERSION`—Specifies the `color.map` files for the color map from releases prior to the Pro/ENGINEER Wildfire release.

Color Schemes

Methods Introduced:

- **wfcWDisplay::GetColorRGBAlternateScheme**
- **wfcWDisplay::SetColorRGBAlternateScheme**

The method `wfcWDisplay::SetColorRGBAlternateScheme` allow you to change the color scheme of Creo application to a predefined alternate color scheme. The alternate color scheme defined in the enumerated data type `wfcColorRGBAlternateScheme` has the following valid values:

- `wfcCOLORRGB_ALT_DEFAULT`—Resets the color scheme to the default color scheme of light to dark blue gradient background.
- `wfcCOLORRGB_ALT_BLACK_ON_WHITE`—Displays black entities on a white background.
- `wfcCOLORRGB_ALT_WHITE_ON_BLACK`—Displays white entities on a black background.
- `wfcCOLORRGB_ALT_WHITE_ON_GREEN`—Displays white entities on a dark green background.
- `wfcCOLORRGB_OPTIONAL1`—Represents the color scheme with a dark background.
- `wfcCOLORRGB_OPTIONAL2`—Represents the color scheme with a medium background.
- `wfcCOLORRGB_CLASSIC_WF`—Resets the color scheme to the light to dark grey background. This was the default color scheme till Pro/ENGINEER Wildfire 4.0.

Use the method `wfcWDisplay::GetColorRGBAlternateScheme` to get the current alternate color scheme.

Line Styles for Graphics

Methods Introduced:

- **wfcWSession::GetLineStyleData**
- **wfcLineStyleData::GetName**
- **wfcLineStyleData::GetDefinition**
- **wfcLineStyleData::GetCapStyle**
- **wfcLineStyleData::GetJoinStyle**
- **wfcLineStyleData::GetDashOffset**
- **wfcLineStyleData::GetDashList**
- **wfcLineStyleData::GetFillStyle**

- **wfcLineStyleData::GetFillRule**

The method `wfcWSession::GetLineStyleData` returns information about the specified line style as a `wfcLineStyleData` object.

The method `wfcLineStyleData::GetName` returns the name of the specified line style.

The method `wfcLineStyleData::GetDefinition` returns the definition of the line style as a series of dashes (-) and spaces ( ).

The method `wfcLineStyleData::GetCapStyle` returns the cap style used for line ends. The method `wfcLineStyleData::GetJoinStyle` returns the style in which line ends are joined. The cap style and join line settings are used from the pen table file for printing a drawing file as a PDF document.

The method `wfcLineStyleData::GetDashOffset` returns the distance between the two dashes in the pattern.

The method `wfcLineStyleData::GetDashList` returns a list of dashes defined in the specified line style.

The method `wfcLineStyleData::GetFillStyle` returns the name of the line font for the specified line style.

The method `wfcLineStyleData::GetFillRule` returns the rule defined to fill a unit of length with the line style pattern.

Displaying Graphics

All the methods in this section draw graphics in the Creo current window and use the color and linestyle set by calls to `pfcBaseSession::SetStdColorFromRGB` and `pfcBaseSession::SetLineStyle`. The methods draw the graphics in the Creo graphics color. The default graphics color is white.

The methods in this section are called using the interface `pfcDisplay`. This interface is extended by the `pfcBaseSession` interface. This architecture allows you to call all these methods on any `pfcSession` object.

By default graphic elements are not stored in the Creo display list. Thus, they do not get redrawn by Creo when the user uses repaint and orientation commands. However, if you store graphic elements in either 2-D or 3-D display lists, Creo application will redraw them when appropriate. See the section on Display Lists and Graphics on page 617 for more information.

Methods Introduced:

- **pfcDisplay::SetPenPosition**
- **pfcDisplay::DrawLine**
- **pfcDisplay::DrawPolyline**

- **wfcWDisplay::DrawPolylines**
- **pfcDisplay::DrawCircle**
- **pfcDisplay::DrawArc2D**
- **pfcDisplay::DrawPolygon2D**

The method `pfcDisplay::SetPenPosition` sets the point at which you want to start drawing a line. The method `pfcDisplay::DrawLine` draws a line to the given point from the position given in the last call to either of the two methods. Call `pfcDisplay::SetPenPosition` for the start of the polyline, and `pfcDisplay::DrawLine` for each vertex. If you use these methods in two-dimensional modes, use screen coordinates instead of solid coordinates.

The method `pfcDisplay::DrawCircle` uses solid coordinates for the center of the circle and the radius value. The circle will be placed to the XY plane of the model.

The method `pfcDisplay::DrawPolyline` also draws polylines, using an array to define the polyline.

Use the method `wfcWDisplay::DrawPolylines` to draw multiple polylines. Pass the sequence of polylines as a `wfcPolylines` object as the input argument.

In two-dimensional models the Display Graphics methods draw graphics at the specified screen coordinates.

The method `pfcDisplay::DrawPolygon2D` draws a polygon in screen coordinates. The method `pfcDisplay::DrawArc2D` draws an arc in screen coordinates.

Controlling Graphics Display

Methods Introduced:

- **pfcDisplay::GetCurrentGraphicsColor**
- **pfcDisplay::SetCurrentGraphicsColor**
- **pfcDisplay::GetCurrentGraphicsMode**
- **pfcDisplay::SetCurrentGraphicsMode**

The method `pfcDisplay::GetCurrentGraphicsColor` returns the standard color used to display graphics in the Creo application. TheCreo default is `pfcCOLOR_DRAWING` (white). The method `pfcDisplay::SetCurrentGraphicsColor` allows you to change the color used to draw subsequent graphics.

The method `pfcDisplay::GetCurrentGraphicsMode` returns the mode used to draw graphics:

- `pfcDRAW_GRAPHICS_NORMAL`—Creo application draws graphics in the required color in each invocation.
- `pfcDRAW_GRAPHICS_COMPLEMENT`—Creo application draws graphics normally, but will erase graphics drawn a second time in the same location. This allows you to create rubber band lines.

The method `pfcDisplay::GetCurrentGraphicsMode` allows you to set the current graphics mode.

Displaying Text in the Graphics Window

Method Introduced:

- **pfcDisplay::DrawText2D**

The method `pfcDisplay::DrawText2D` places text at a position specified in screen coordinates. If you want to add text to a particular position on the solid, you must transform the solid coordinates into screen coordinates by using the view matrix.

Text items drawn are not known to Creo and therefore are not redrawn when you select the repaint command. To notify the Creo application of these objects, create them inside the `OnDisplay()` method of the Display Listener.

Controlling Text Attributes

Methods Introduced:

- **pfcDisplay::GetTextHeight**
- **pfcDisplay::SetTextHeight**
- **pfcDisplay::GetWidthFactor**
- **pfcDisplay::SetWidthFactor**
- **pfcDisplay::GetRotationAngle**
- **pfcDisplay::SetRotationAngle**
- **pfcDisplay::GetSlantAngle**
- **pfcDisplay::SetSlantAngle**

These methods control the attributes of text added by calls to `pfcDisplay::DrawText2D`.

You can get and set the following information:

- Text height (in screen coordinates)
- Width ratio of each character, including the gap, as a proportion of the height
- Rotation angle of the whole text, in counterclockwise degrees

- Slant angle of the text, in clockwise degrees

Controlling Text Fonts

Methods Introduced:

- **pfcDisplay::GetDefaultFont**
- **pfcDisplay::GetCurrentFont**
- **pfcDisplay::SetCurrentFont**
- **pfcDisplay::GetFontById**
- **pfcDisplay::GetFontByName**

The method `pfcDisplay::GetDefaultFont` returns the default Creo text font. The text fonts are identified in Creo application by names and by integer identifiers. To find a specific font, use the methods `pfcDisplay::GetFontById` or `pfcDisplay::GetFontByName`.

Display Lists and Graphics

When generating a display of a solid in a window,Creo application maintains two display lists. A display list contains a set of vectors that are used to represent the shape of the solid in the view. A 3D display list contains a set of three-dimensional vectors that represent an approximation to the geometry of the edges of the solid. This list gets rebuilt every time the solid is regenerated.

A 2D display list contains the two-dimensional projections of the edges of the solid 3D display list onto the current window. It is rebuilt from the 3D display list when the orientation of the solid changes. The methods in this section enable you to add your own vectors to the display lists, so that the graphics will be redisplayed automatically by Creo application until the display lists are rebuilt.

When you add graphics items to the 2D display list, they will be regenerated after each repaint (when zooming and panning) and will be included in plots created by Creo. When you add graphics to the 3D display list, you get the further benefit that the graphics survive a change to the orientation of the solid and are displayed even when you spin the solid dynamically.

Methods Introduced:

- **pfcDisplayListener::OnDisplay**
- **pfcDisplay::CreateDisplayList2D**
- **pfcDisplay::CreateDisplayList3D**
- **pfcDisplayList2D::Display**
- **pfcDisplayList3D::Display**
- **pfcDisplayList2D::Delete**

- **pfcDisplayList3D::Delete**

A display listener is a class that acts similarly to an action listener. You must implement the method inherited from the `pfcDisplayListener` interface. The implementation should provide calls to methods on the provided `pfcDisplay` object to produce 2D or 3D graphics.

In order to create a display list in Creo application, you must call `pfcDisplay::CreateDisplayList2D` or `pfcDisplay::CreateDisplayList3D` to tell Creo application to use your listener to create the display list vectors.

`pfcDisplayList2D::Display` or `pfcDisplay::List3D.Display` will display or redisplay the elements in your display list. The application should delete the display list data when it is no longer needed.

The methods `pfcDisplayList2D::Delete` and the method `pfcDisplayList3D::Delete` will remove both the specified display list from a session.

Note

The method `pfcWindow::Refresh` does not cause either of the display lists to be regenerated, but simply repaints the window using the 2D display list.

Exceptions

Possible exceptions that might be thrown by displaying graphics methods are shown in the following table:

Exception	Reason
`XToolkitNotExist`	The display list is empty.
`XToolkitNotFound`	The method could not find the display list or the font specified in a previous call to `pfcDisplay::SetCurrentFont` was not found.
`XToolkitCantOpen`	The use of display lists is disabled.
`XToolkitAbort`	The display was aborted.
`XToolkitNotValid`	The specified display list is invalid.
`XToolkitInvalidItem`	There is an invalid item in the display list.
`XToolkitGeneralError`	The specified display list is already in the process of being displayed.

36

External Data

This chapter explains using External Data in Creo Object TOOLKIT C++.

External Data

This chapter describes how to store and retrieve external data. External data enables a Creo Object TOOLKIT C++ application to store its own data in a Creo database in such a way that it is invisible to the Creo user. This method is different from other means of storage accessible through the Creo user interface.

Introduction to External Data

External data provides a way for the Creo application to store its own private information about a Creo model within the model file. The data is built and interrogated by the application as a workspace data structure. It is saved to the model file when the model is saved, and retrieved when the model is retrieved. The external data is otherwise ignored by Creo; the application has complete control over form and content.

The external data for a specific Creo model is broken down into classes and slots. A class is a named "bin" for your data, and identifies it as yours so no other Creo API application (or other classes in your own application) will use it by mistake. An application usually needs only one class. The class name should be unique for each application and describe the role of the data in your application.

Each class contains a set of data slots. Each slot is identified by an identifier and optionally, a name. A slot contains a single data item of one of the following types:

Creo Object TOOLKIT C++ Type	Data
`pfcEXTDATA_INTEGER`	integer
`pfcEXTDATA_DOUBLE`	double
`pfcEXTDATA_STRING`	string

The Creo Object TOOLKIT C++ interfaces used to access external data in Creo are:

Creo Object TOOLKIT C++ Type	Data Type
`pfcExternalDataAccess`	This is the top level object and is created when attempting to access external data.
`pfcExternalDataClass`	This is a class of external data and is identified by a unique name.
`pfcExternalDataSlot`	This is a container for one item of data. Each slot is stored in a class.
`pfcExternalData`	This is a compact data structure that contains either an integer, double or string value.

Compatibility with Creo TOOLKIT

Creo Object TOOLKIT C++ and Creo TOOLKIT share external data in the same manner. Creo Object TOOLKIT C++ external data is accessible by Creo TOOLKIT and the reverse is also true. However, an error will result if Creo Object TOOLKIT C++ attempts to access external data previously stored by Creo TOOLKIT as a stream.

Accessing External Data

Methods Introduced:

- **pfcModel::AccessExternalData**
- **pfcModel::TerminateExternalData**
- **pfcExternalDataAccess::IsValid**

The method `pfcModel::AccessExternalData` prepares Creo application to read external data from the model file. It returns the `pfcExternalDataAccess` object that is used to read and write data. This method should be called only once for any given model in session.

The method `pfcModel::TerminateExternalData` stops Creo application from accessing external data in a model. When you use this method all external data in the model will be removed. Permanent removal will occur when the model is saved.

Note

If you need to preserve the external data created in session, you must save the model before calling this function. Otherwise, your data will be lost.

The method `pfcExternalDataAccess::IsValid` determines if the `pfcExternalDataAccess` object can be used to read and write data.

Storing External Data

Methods Introduced:

- **pfcExternalDataAccess::CreateClass**
- **pfcExternalDataClass::CreateSlot**
- **pfcExternalDataSlot::SetValue**

The first step in storing external data in a new class and slot is to set up a class using the method `pfcExternalDataAccess::CreateClass`, which provides the class name. The method outputs `pfcExternalDataClass`, used by the application to reference the class.

The next step is to use `pfcExternalDataClass::CreateSlot` to create an empty data slot and input a slot name. The method outputs a `pfcExternalDataSlot` object to identify the new slot.

Note

Slot names cannot begin with a number.

The method `pfcExternalDataSlot::SetValue` specifies the data type of a slot and writes an item of that type to the slot. The input is a `pfcExternalData` object that you can create by calling any one of the methods in the next section.

Initializing Data Objects

Methods Introduced:

- **pfcCreateIntExternalData**
- **pfcCreateDoubleExternalData**
- **pfcCreateStringExternalData**

These methods initialize a `pfcExternalData` object with the appropriate data inputs.

Retrieving External Data

Methods Introduced:

- **pfcExternalDataAccess::LoadAll**
- **pfcExternalDataAccess::ListClasses**
- **pfcExternalDataClass::ListSlots**
- **pfcExternalDataSlot::GetValue**
- **pfcExternalData::Getdiscr**
- **pfcExternalData::GetIntegerValue**
- **pfcExternalData::GetDoubleValue**
- **pfcExternalData::GetStringValue**

For improved performance, external data is not loaded automatically into memory with the model. When the model is in session, call the method `pfcExternalDataAccess::LoadAll` to retrieve all the external data for the specified model from the Creo model file and put it in the workspace. The method needs to be called only once to retrieve all the data.

The method `pfcExternalDataAccess::ListClasses` returns a sequence of `pfcExternalDataClasses` registered in the model. The method `pfcExternalDataClass::ListSlots` provide a sequence of `pfcExternalDataSlots` existing for each class.

The method `pfcExternalDataSlot::GetValue` reads the `pfcExternalData` from a specified slot.

To find out a data type of a `pfcExternalData`, call `pfcExternalData::Getdiscr` and then call one of these methods to get the data, depending on the data type:

- `pfcExternalData::GetIntegerValue`
- `pfcExternalData::GetDoubleValue`
- `pfcExternalData::GetStringValue`

Selecting the Node from the External Application Tree

The tree created by an external application is similar to the Creo model tree. Each node of this tree represents an external object that has been created by the application. The external objects could be different types of entities, such as, light sources, light sensors, and so on.

Methods Introduced:

- **wfcWSession::RegisterExternalSelectionHighlight**
- **wfcExternalSelectionHighlight::StartNotify**
- **wfcExternalSelectionHighlight::Action**
- **wfcExternalSelectionHighlight::EndNotify**
- **wfcWSelection::RecordExternalSelection**
- **wfcWSession::ReleaseExternalSelectionHighlight**

The method `wfcWSession::RegisterExternalSelectionHighlight` registers the callback methods when you select or deselect a node in the user tree or an object in the graphics window. The notification method `wfcExternalSelectionHighlight::StartNotify` is called when the method `pfcBaseSession::Select` is activated. It notifies the application about entering `pfcBaseSession::Select`. The callback method

wfcExternalSelectionHighlight::Action is called when you select or deselect an external object. The Creo Object TOOLKIT C++ application will highlight the external object or remove the highlight according to the selection. The notification method wfcExternalSelectionHighlight::EndNotify is called when the applications is about to exit the method pfcBaseSession::Select.

On clicking a tree node, the application creates a wfcWSelection object and uses the method wfcWSelection::RecordExternalSelection to pass it to pfcBaseSession::Select. The input arguments of this method is enumerated data type wfcSelectionRecordAction. The valid values are:

- wfcSELECT_OVERRIDE—Specifies that the previous selection is overridden.
- wfcSELECT_TOGGLE—Specifies that the last two selections are toggled.

Use the method wfcWSession::ReleaseExternalSelectionHighlight to release the memory of the client interface wfcExternalSelectionHighlight in the method wfcWSession::RegisterExternalSelectionHighlight. After the client interface is released it cannot be used by the application.

Exceptions

Most exceptions thrown by external data methods in Creo Object TOOLKIT C++ extend pfcXExternalDataError, which is a subclass of pfcXToolkitError.

An additional exception thrown by external data methods is pfcXBadExternalData. This exception signals an error accessing data. For example, external data access might have been terminated or the model might contain stream data from Creo TOOLKIT.

The following table lists these exceptions.

Exception	Cause
pfcXExternalDataInvalidObject	Generated when a model or class is invalid.
pfcXExternalDataClassOrSlotExists	Generated when creating a class or slot and the proposed class or slot already exists.
pfcXExternalDataNamesTooLong	Generated when a class or slot name is too long.
pfcXExternalDataSlotNotFound	Generated when a specified class or slot does not exist.
pfcXExternalDataEmptySlot	Generated when the slot you are attempting to read is empty.

Exception	Cause
`pfcXExternalDataInvalidSlotName`	Generated when a specified slot name is invalid.
`pfcXBadGetExternalData`	Generated when you try to access an incorrect data type in a `External.ExternalData` object.

37

Windchill Connectivity APIs

Creo has the capability to be directly connected to Windchill solutions, including ProjectLink and PDMLink servers. This access allows users to manage and control the product data seamlessly from within the Creo application.

This chapter lists Creo Object TOOLKIT C++ APIs that support Windchill servers and server operations in a connected Creo session.

Introduction

The methods introduced in this chapter provide support for the basic Windchill server operations from within Creo. With these methods, operations such as registering a Windchill server, managing workspaces, and check in or check out of objects will be possible via Creo Object TOOLKIT C++. The capabilities of these APIs are similar to the operations available from within the Creo client, with some restrictions.

Some of these APIs are supported from a non-interactive, that is, batch mode application or asynchronous application. For more information on batch mode application or asynchronous application, refer to the chapter Asynchronous Mode on page 587.

Accessing a PTC Windchill Server from a Creo Session

Creo application allows you to register Windchill servers as a connection between the Windchill database and Creo application. Although the represented Windchill database can be from Windchill ProjectLink or PDMLink, all types of databases are represented in the same way.

You can use the following identifiers when referring to Windchill servers in Creo Object TOOLKIT C++:

- Codebase URL—This is the root portion of the URL that is used to connect to a Windchill server. For example `<server.mycompany.com>/Windchill`..
- Server Alias—A server alias is used to refer to the server after it has been registered. The alias is also used to construct paths to files in the server workspaces and commonspaces. The server alias is chosen by the user or application and it need not have any direct relationship to the codebase URL. An alias can be any normal name, such as `my_alias`.

Accessing Information Before Registering a Server

To start working with a Windchill server, you must establish a connection by registering the server in the Creo application. The methods described in this section allow you to connect to a Windchill server and access information related to the server.

Methods Introduced:

- **pfcBaseSession::AuthenticateBrowser**
- **pfcBaseSession::GetServerLocation**
- **pfcServerLocation::GetClass**

- **pfcServerLocation::GetLocation**
- **pfcServerLocation::GetVersion**
- **pfcServerLocation::ListContexts**
- **pfcServerLocation::CollectWorkspaces**

Use the method `pfcBaseSession::AuthenticateBrowser` to set the authentication context using a valid username and password. A successful call to this method allows the Creo session to register with any server that accepts the username and password combination. A successful call to this method also ensures that an authentication dialog box does not appear during the registration process. You can call this method any number of times to set the authentication context for any number of Windchill servers, provided that you register the appropriate servers or servers immediately after setting the context.

The method `pfcServerLocation::GetLocation` returns a `pfcServer::ServerLocation` object representing the codebase URL for a possible server. The server may not have been registered yet, but you can use this object and the methods it contains to gather information about the server prior to registration.

The method `pfcServerLocation::GetClass` returns the class of the server or server location. The values are:

- `Windchill`—Denotes a Windchill PDMLink server.
- `ProjectLink`—Denotes Windchill ProjectLink type of servers.

The method `pfcServerLocation::GetVersion` returns the version of Windchill that is configured on the server or server location, for example,`9.0` or `10.0`. This method accepts the server codebase URL as the input.

Note

`pfcServerLocation::GetVersion` works only for Windchill servers and throws the `pfcXToolkitUnsupported` exception, if the server is not a Windchill server.

The method `pfcServerLocation::ListContexts` gives a list of all the available contexts for a specified server. A context is used to associate a workspace with a product, project, or library.

The method `pfcServerLocation::CollectWorkspaces` returns the list of available workspaces for the specified server. The workspace objects returned contain the name of each workspace and its context.

Registering and Activating a Server

From Creo 2.0 onward, the Creo Object TOOLKIT C++ methods call the same underlying API as Creo application to register and unregister servers. Hence, registering the servers usingCreo Object TOOLKIT C++ methods is similar to registering the servers using the Creo user interface. Therefore, the servers registered by Creo Object TOOLKIT C++ are available in the Creo Server Registry. The servers are also available in other locations in the Creo user interface such as, the **Folder Navigator** and the embedded browser.

Methods Introduced:

- **pfcBaseSession::RegisterServer**
- **pfcServer::Activate**
- **pfcServer::Unregister**

The method `pfcBaseSession::RegisterServer` registers the specified server with the codebase URL. You can automate the registration of servers in interactive mode. To preregister the servers use the standard `config.fld` setup. If you do not want the servers to be preregistered in batch mode, set the environment variable `PTC_WF_ROOT` to an empty directory before starting the Creo application.

A successful call to `pfcBaseSession::AuthenticateBrowser` with a valid username and password is essential for `pfcBaseSession::RegisterServer` to register the server without launching the authentication dialog box. Registration of the server establishes the server alias. You must designate an existing workspace to use when registering the server. After the server has been registered, you may create a new workspace.

The method `pfcServer::Activate` sets the specified server as the active server in the Creo session.

The method `pfcServer::Unregister` unregisters the specified server.

Accessing Information From a Registered Server

Methods Introduced:

- **pfcServer::GetIsActive**
- **pfcServer::GetAlias**
- **pfcServer::GetContext**

The method `pfcServer::GetIsActive` specifies if the server is active.

The method `pfcServer::GetAlias` returns the alias of a server if you specify the codebase URL.

The method `pfcServer::GetContext` returns the active context of the active server.

Information on Servers in Session

Methods Introduced:

- **pfcBaseSession::GetActiveServer**
- **pfcBaseSession::GetServerByAlias**
- **pfcBaseSession::GetServerByUrl**
- **pfcBaseSession::ListServers**

The method `pfcBaseSession::GetActiveServer` returns the active server handle.

The method `pfcBaseSession::GetServerByAlias` returns the handle to the server matching the given server alias, if it exists in session.

The method `pfcBaseSession::GetServerByUrl` returns the handle to the server matching the given server URL and workspace name, if it exists in session.

The method `pfcBaseSession::ListServers` returns a list of servers registered in this session.

Accessing Workspaces

For every workspace, a new distinct storage location is maintained in the user's personal folder on the server (server-side workspace) and on the client (client-side workspace cache). Together, the server-side workspace and the client-side workspace cache make up the workspace.

Methods Introduced:

- **pfcWorkspaceDefinition::Create**
- **pfcWorkspaceDefinition::GetWorkspaceName**
- **pfcWorkspaceDefinition::GetWorkspaceContext**
- **pfcWorkspaceDefinition::SetWorkspaceName**
- **pfcWorkspaceDefinition::SetWorkspaceContext**

The interface `pfcWorkspaceDefinition` contains the name and context of the workspace. The method `pfcServerLocation::CollectWorkspaces` returns an array of workspace data. Workspace data is also required for the method `pfcServer::CreateWorkspace` to create a workspace with a given name and a specific context.

The method `pfcWorkspaceDefinition::Create` creates a new workspace definition object suitable for use when creating a new workspace on the server.

The method `pfcWorkspaceDefinition::GetWorkspaceName` retrieves the name of the workspace.

The method `pfcWorkspaceDefinition::GetWorkspaceContext` retrieves the context of the workspace.

The method `pfcWorkspaceDefinition::SetWorkspaceName` sets the name of the workspace.

The method `pfcWorkspaceDefinition::SetWorkspaceContext` sets the context of the workspace.

Creating and Modifying the Workspace

Methods Introduced:

- **pfcServer::CreateWorkspace**
- **pfcServer::GetActiveWorkspace**
- **pfcServer::SetActiveWorkspace**
- **pfcServerLocation::DeleteWorkspace**

The method `pfcServer::CreateWorkspace` creates and activates a new workspace.

The method `pfcServer::GetActiveWorkspace` retrieves the name of the active workspace.

The method `pfcServer::SetActiveWorkspace` sets a specified workspace as an active workspace.

The method `pfcServerLocation::DeleteWorkspace` deletes the specified workspace. The method deletes the workspace only if the following conditions are met:

- The workspace is not the active workspace.
- The workspace does not contain any checked out objects.

Use one of the following techniques to delete an active workspace:

- Make the required workspace inactive using `pfcServer::SetActiveWorkspace` with the name of some other workspace and then call `pfcServerLocation::DeleteWorkspace`.
- Unregister the server using `pfcServer::Unregister` and delete the workspace.

Workflow to Register a Server

To Register a Server with an Existing Workspace

Perform the following steps to register a Windchill server with an existing workspace:

1. Set the appropriate authentication context using the method `pfcBaseSession::AuthenticateBrowser` with a valid username and password.
2. Look up the list of workspaces using the method `pfcServerLocation::CollectWorkspaces`. If you already know the name of the workspace on the server, then ignore this step.
3. Register the workspace using the method `pfcBaseSession::RegisterServer` with an existing workspace name on the server.
4. Activate the server using the method `pfcServer::Activate`.

To Register a Server with a New Workspace

Perform the following steps to register a Windchill server with a new workspace:

1. Perform steps 1 to 4 in the preceding section to register the Windchill server with an existing workspace.
2. Use the method `pfcServerLocation::ListContexts` to choose the required context for the server.
3. Create a new workspace with the required context using the method `pfcServer::CreateWorkspace`. This method automatically makes the created workspace active.

Note

You can create a workspace only after the server is registered.

Aliased URL

An aliased URL serves as a handle to the server objects. You can access the server objects in the commonspace (shared folders) and the workspace using an aliased URL. An aliased URL is a unique identifier for the server object and its format is as follows:

- Object in workspace has a prefix `wtws`
  `wtws://<server_alias>/<workspace_name>/<object_server_name>`

  where `<object_server_name>` includes `<object_name>.<object_extension>`

  For example,
  `wtws://my_server/my_workspace/abcd.prt`,
  `wtws://my_server/my_workspace/intf_file.igs`

  where

  `<server_alias>` is `my_server`

  `<workspace_name>` is `my_workspace`
- Object in commonspace has a prefix `wtpub`
  `wtpub://<server_alias>/<folder_location>/<object_server_name>`

  For example,
  `wtpub://my_server/path/to/cs_folder/abcd.prt`

  where

  `<server_alias>` is `my_server`

  `<folder_location>` is `path/to/cs_folder`

Note

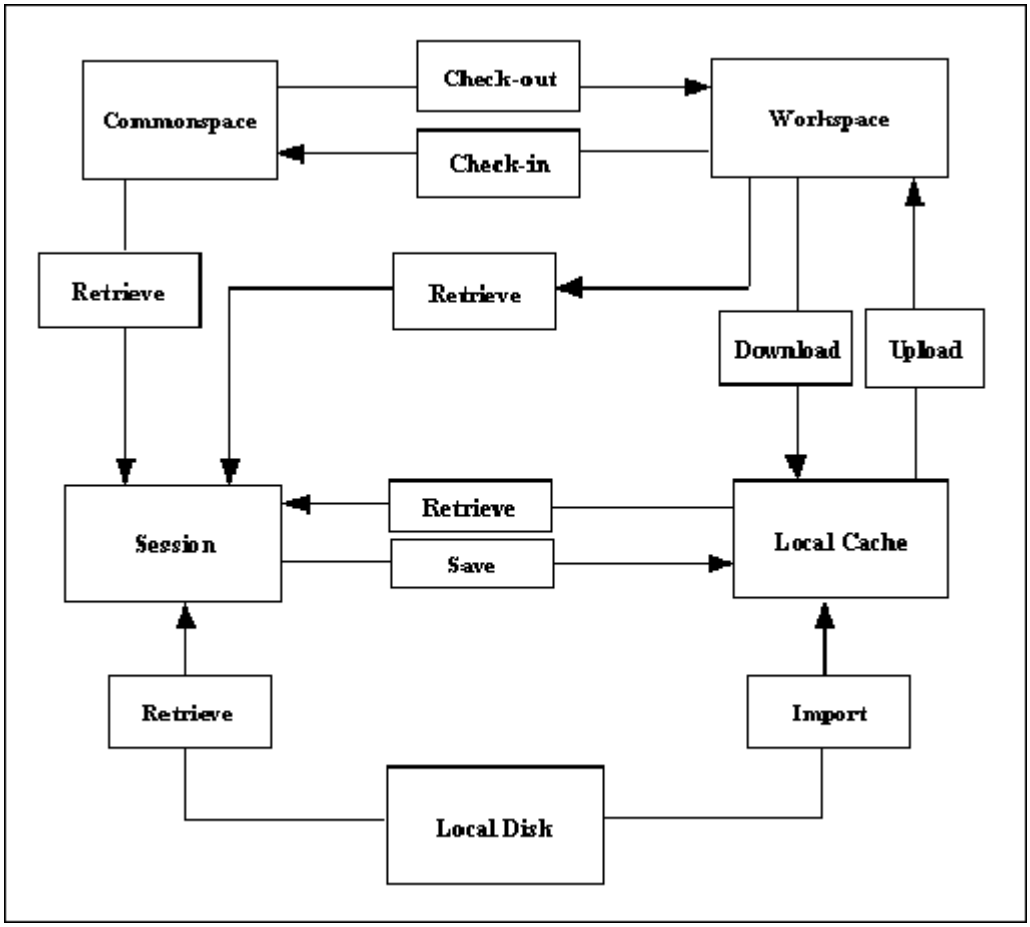
- `object_server_name` must be in lowercase.
- The APIs are case-sensitive to the aliased URL.
- `<object_extension>` should not contain Creo versions, for example, .1 or .2, and so on.

Server Operations

After registering the Windchill server with Creo application, you can start accessing the data on the Windchill servers. The Creo interaction with Windchill servers leverages the following locations:

- Commonspace (Shared folders)
- Workspace (Server-side workspace)
- Workspace local cache (Client-side workspace)
- Creo session
- Local disk

The methods described in this section enable you to perform the basic server operations. The following illustration shows how data is transferred among these locations.



Save

Methods Introduced:

- **pfcModel::Save**

The method `pfcModel::Save` stores the object from the session in the local workspace cache, when a server is active.

Upload

An upload transfers Creo files and any other dependencies from the local workspace cache to the server-side workspace.

Methods Introduced:

- **pfcServer::UploadObjects**

- **pfcServer::UploadObjectsWithOptions**
- **pfcUploadOptions::Create**

The method `pfcServer::UploadObjects` uploads the object to the workspace. The object to be uploaded must be present in the current Creo session. You must save the object to the workspace using `pfcModel::Save` before attempting to upload it.

The method `pfcServer::UploadObjectsWithOptions` uploads objects to the workspace using the options specified in the `pfcUploadOptions` interface. These options allow you to upload the entire workspace, auto-resolve missing references, and indicate the target folder location for the new content during the upload. You must save the object to the workspace using `pfcModel::Save`, or import it to the workspace using `pfcBaseSession::ImportToCurrentWS` before attempting to upload it.

Create the `pfcUploadOptions` object using the method `pfcUploadOptions::Create`.

The methods available for setting the upload options are described in the following section.

CheckIn

After you have finished working on objects in your workspace, you can share the design changes with other users. The checkin operation copies the information and files associated with all changed objects from the workspace to the Windchill database.

Methods Introduced:

- **pfcServer::CheckinObjects**
- **pfcCheckinOptions::Create**
- **pfcUploadBaseOptions::SetDefaultFolder**
- **pfcUploadBaseOptions::SetNonDefaultFolderAssignments**
- **pfcUploadBaseOptions::SetAutoresolveOption**
- **pfcCheckinOptions::SetBaselineName**
- **pfcCheckinOptions::SetBaselineNumber**
- **pfcCheckinOptions::SetBaselineLocation**
- **pfcCheckinOptions::GetBaselineLifecycle**
- **pfcCheckinOptions::SetKeepCheckedout**

The method `pfcServer::CheckinObjects` checks in an object into the database. The object to be checked in must be present in the current Creo session. Changes made to the object are not included unless you save the object to the workspace using the method `pfcModel::Save` before you check it in.

If you pass `NULL` as the value of the *options* parameter, the checkin operation is similar to the **Check-In** option in Creo application. For more details on **Check-In**, refer to the Creo online help.

Use the method `pfcCheckinOptions::Create` to create a new `pfcCheckinOptions` object.

By using an appropriately constructed *options* argument, you can control the checkin operation. Use the APIs listed above to access and modify the checkin options. The checkin options are as follows:

- *DefaultFolder*—Specifies the default folder location on the server for the automatic checkin operation.
- *NonDefaultFolderAssignment*—Specifies the folder location on the server to which the objects will be checked in.
- *AutoresolveOption*—Specifies the option used for auto-resolving missing references. These options are defined in the `pfcServerAutoresolveOption` class, and are as follows:
  - `pfcSERVER_DONT_AUTORESOLVE`—Model references missing from the workspace are not automatically resolved. This may result in a conflict upon checkin. This option is used by default.
  - `pfcSERVER_AUTORESOLVE_IGNORE`—Missing references are automatically resolved by ignoring them.
  - `pfcSERVER_AUTORESOLVE_UPDATE_IGNORE`—Missing references are automatically resolved by updating them in the database and ignoring them if not found.
- *Baseline*—Specifies the baseline information for the objects upon checkin. The baseline information for a checkin operation is as follows:
  - *BaselineName*—Specifies the name of the baseline.
  - *BaselineNumber*—Specifies the number of the baseline.

  The default format for the baseline name and baseline number is `Username + time (GMT) in milliseconds`.
  - *BaselineLocation*—Specifies the location of the baseline.
  - *BaselineLifecycle*—Specifies the name of the lifecycle.
- *KeepCheckedout*—If the value specified is `true`, then the contents of the selected object are checked into the Windchill server and automatically checked out again for further modification.

Retrieval

Standard Creo Object TOOLKIT C++ provides several methods that are capable of retrieving models. When using these methods with Windchill servers, remember that these methods do not check out the object to allow modifications.

Methods Introduced:

- **pfcBaseSession::RetrieveModel**
- **pfcBaseSession::RetrieveModelWithOpts**
- **pfcBaseSession::OpenFile**

The methods `pfcBaseSession::RetrieveModel`, `pfcBaseSession::RetrieveModelWithOpts`, and `pfcBaseSession::OpenFile` load an object into a session given its name and type. The methods search for the object in the active workspace, the local directory, and any other paths specified by the `search_path` configuration option.

Checkout and Download

To modify an object from the commonspace, you must check out the object. The process of checking out communicates your intention to modify a design to the Windchill server. The object in the database is locked, so that other users can obtain read-only copies of the object, and are prevented from modifying the object while you have checked it out.

Checkout is often accompanied by a download action, where the objects are brought from the server-side workspace to the local workspace cache. In Creo Object TOOLKIT C++, both operations are covered by the same set of methods.

Methods Introduced:

- **pfcServer::CheckoutObjects**
- **pfcServer::CheckoutMultipleObjects**
- **pfcCheckoutOptions.Create**
- **pfcCheckoutOptions::SetDependency**
- **pfcCheckoutOptions::SetSelectedIncludes**
- **pfcCheckoutOptions::SetIncludeInstances**
- **pfcCheckoutOptions::SetVersion**
- **pfcCheckoutOptions::SetDownload**
- **pfcCheckoutOptions::SetReadonly**

The method `pfcServer::CheckoutObjects` checks out and optionally downloads the object to the workspace based on the configuration specifications of the workspace. The input arguments of this method are as follows:

- *Mdl*—Specifies the object to be checked out. This is applicable if the model has already been retrieved without checking it out.
- *File*—Specifies the top-level object to be checked out.
- *Checkout*—The checkout flag. If you specify the value of this argument as `true`, the selected object is checked out. Otherwise, the object is downloaded without being checked out. The download action enables you to bring read-only copies of objects into your workspace. This allows you to examine the object without locking it.
- *Options*—Specifies the checkout options object. If you pass `NULL` as the value of this argument, then the default Creo checkout rules apply. Use the method `pfcCheckoutOptions.Create` to create a new `pfcCheckoutOptions` object.

Use the method `pfcServer::CheckoutMultipleObjects` to check out and download multiple objects to the workspace based on the configuration specifications of the workspace. This method takes the same input arguments as listed above, except for *Mdl* and *File*. Instead it takes the argument *Files* that specifies the sequence of the objects to check out or download.

Note

Creo Object TOOLKIT C++ methods do not support the `AS_STORED` configuration.

By using an appropriately constructed *options* argument in the above functions, you can control the checkout operation. Use the APIs listed above to modify the checkout options. The checkout options are as follows:

- *Dependency*—Specifies the dependency rule used while checking out dependents of the object selected for checkout. The types of dependencies given by the `ServerDependency` class are as follows:
  - `pfcSERVER_DEPENDENCY_ALL`—All the objects that are dependent on the selected object are downloaded, that is, they are added to the workspace.
  - `pfcSERVER_DEPENDENCY_REQUIRED`—All the objects that are required to successfully retrieve the selected object in the CAD application are downloaded, that is, they are added to workspace.
  - `pfcSERVER_DEPENDENCY_NONE`—None of the dependent objects from the selected object are downloaded, that is, they are not added to workspace.

- *IncludeInstances*—Specifies the rule for including instances from the family table during checkout. The type of instances given by the `pfcServerIncludeInstances` class are as follows:
  - `pfcSERVER_INCLUDE_ALL`—All the instances of the selected object are checked out.
  - `pfcSERVER_INCLUDE_SELECTED`—The application can select the family table instance members to be included during checkout.
  - `pfcSERVER_INCLUDE_NONE`—No additional instances from the family table are added to the object list.
- *SelectedIncludes*—Specifies the sequence of URLs to the selected instances, if *IncludeInstances* is of type `SERVER_INCLUDE_SELECTED`.
- *Version*—Specifies the version of the checked out object. If this value is set to `NULL`, the object is checked out according to the current workspace configuration.
- *Download*—Specifies the checkout type as `download` or `link`. The value `download` specifies that the object content is downloaded and checked out, while `link` specifies that only the metadata is downloaded and checked out.
- *Readonly*—Specifies the checkout type as a read-only checkout. This option is applicable only if the checkout type is `link`.

The following truth table explains the dependencies of the different control factors in the method `pfcServer::CheckoutObjects` and the effect of different combinations on the end result.

Argument *checkout* in `pfcServer::CheckoutObjects`	`pfcCheckoutOptions::SetDownload`	`pfcCheckoutOptions::SetReadonly`	Result
true	true	NA	Object is checked out and its content is downloaded.
true	false	NA	Object is checked out but content is not downloaded.
false	NA	true	Object is downloaded without checkout and as read-only.
false	NA	false	Not supported

Undo Checkout

Method Introduced:

- **pfcServer::UndoCheckout**

Use the method `pfcServer::UndoCheckout` to undo a checkout of the specified object. When you undo a checkout, the changes that you have made to the content and metadata of the object are discarded and the content, as stored in the server, is downloaded to the workspace. This method is applicable only for the model in the active Creo session.

Import and Export

Creo Object TOOLKIT C++ provides you with the capability of transferring specified objects to and from a workspace. Import and export operations must take place in a session with no models. An import operation transfers a file from the local disk to the workspace.

Methods Introduced:

- **pfcBaseSession::ExportFromCurrentWS**
- **pfcBaseSession::ImportToCurrentWS**
- **pfcWSImportExportMessage::GetDescription**
- **pfcWSImportExportMessage::GetFileName**
- **pfcWSImportExportMessage::GetMessageType**
- **pfcWSImportExportMessage::GetResolution**
- **pfcWSImportExportMessage::GetSucceeded**
- **pfcBaseSession::SetWSExportOptions**
- **pfcWSExportOptions::Create**
- **pfcWSExportOptions::SetIncludeSecondaryContent**

The method `pfcBaseSession::ExportFromCurrentWS` exports the specified objects from the current workspace to a disk in a linked session of Creo application.

The method `pfcBaseSession::ImportToCurrentWS` imports the specified objects from a disk to the current workspace in a linked session of Creo application.

Both `pfcBaseSession::ExportFromCurrentWS` and `pfcBaseSession::ImportToCurrentWS` allow you to specify a dependency criterion to process the following items:

- All external dependencies
- Only required dependencies
- No external dependencies

Both `pfcBaseSession::ExportFromCurrentWS` and `pfcBaseSession::ImportToCurrentWS` return the messages generated during the export or import operation in the form of the

pfcWSImportExportMessages object. Use the APIs listed above to access the contents of a message. The message specified by the pfcWSImportExportMessage object contains the following items:

- Description—Specifies the description of the problem or the message information.
- FileName—Specifies the object name or the name of the object path.
- MessageType—Specifies the severity of the message in the form of the pfcWSImportExportMessageType class. The severity is one of the following types:
  - pfcWSIMPEX_MSG_INFO—Specifies an informational type of message.
  - pfcWSIMPEX_MSG_WARNING—Specifies a low severity problem that can be resolved according to the configured rules.
  - pfcWSIMPEX_MSG_CONFLICT—Specifies a conflict that can be overridden.
  - pfcWSIMPEX_MSG_ERROR—Specifies a conflict that cannot be overridden or a serious problem that prevents processing of an object.
- Resolution—Specifies the resolution applied to resolve a conflict that can be overridden. This is applicable when the message is of the type pfcWSIMPEX_MSG_CONFLICT.
- Succeeded—Determines whether the resolution succeeded or not. This is applicable when the message is of the type pfcWSIMPEX_MSG_CONFLICT.

The method pfcBaseSession::SetWSExportOptions sets the export options used while exporting the objects from a workspace in the form of the pfcWSExportOptions object. Create this object using the method pfcWSExportOptions::Create. The export options are as follows:

- *Include Secondary Content*—Indicates whether or not to include secondary content while exporting the primary Creo model files. Use the method pfcWSExportOptions::SetIncludeSecondaryContent to set this option.

File Copy

Creo Object TOOLKIT C++ provides you with the capability of copying a file from the workspace or target folder to a location on the disk and vice-versa.

Methods Introduced:

- **pfcBaseSession::CopyFileToWS**
- **pfcBaseSession::CopyFileFromWS**

Use the method `pfcBaseSession::CopyFileToWS` to copy a file from the disk to the workspace. The file can optionally be added as secondary content to a given workspace file. If the viewable file is added as secondary content, a dependency is created between the Creo model and the viewable file.

Use the method `pfcBaseSession::CopyFileFromWS` to copy a file from the workspace to a location on disk.

When importing or exporting Creo models, PTC recommends that you use methods `pfcBaseSession::ImportToCurrentWS` and `pfcBaseSession::ExportFromCurrentWS`, respectively, to perform the import or export operation. Methods that copy individual files do not traverse Creo model dependencies, and therefore do not copy a fully retrievable set of models at the same time.

Additionally, only the methods `pfcBaseSession::ImportToCurrentWS` and `pfcBaseSession::ExportFromCurrentWS` provide full metadata exchange and support. That means `pfcBaseSession::ImportToCurrentWS` can communicate all the Creo designated parameters, dependencies, and family table information to a PDM system while `pfcBaseSession::ExportFromCurrentWS` can update exported Creo data with PDM system changes to designated and system parameters, dependencies, and family table information. Hence PTC recommends the use of `pfcBaseSession::CopyFileToWS` and `pfcBaseSession::CopyFileFromWS` to process only non-Creo files.

Server Object Status

Methods Introduced:

- **pfcServer::IsObjectCheckedOut**
- **pfcServer::IsObjectModified**
- **pfcServer::IsServerObjectModified**
- **pfcServerObjectStatus::GetIsCheckedOut**
- **pfcServerObjectStatus::GetIsModifiedInWorkspace**
- **pfcServerObjectStatus::GetIsModifiedLocally**

The methods described in this section verify the current status of the object in the workspace. The method `pfcServer::IsObjectCheckedOut` specifies whether the object is checked out for modification. The value `true` indicates that the specified object is checked out to the active workspace.

The value `false` indicates one of the following statuses:

- The specified object is not checked out
- The specified object is only uploaded to the workspace, but was never checked in

- The specified object is only saved to the local workspace cache, but was never uploaded

The method `pfcServer::IsObjectModified` specifies whether the object has been modified in the workspace. This method returns `true` if the object was modified locally.

Use the method `pfcServer::IsServerObjectModified` to check if the specified object has been modified in workspace or is modified locally. This method returns an object of the class `pfcServerObjectStatus`.

Use the method `pfcServerObjectStatus::GetIsCheckedOut` to identify whether the object has been checked out or not. This method returns the value `true` if the object is checked out.

Use the method `pfcServerObjectStatus::GetIsModifiedInWorkspace` to identify whether the object has been modified in workspace. This method returns the value `true` if the object is modified in workspace.

Use the method `pfcServerObjectStatus::GetIsModifiedLocally` to identify whether the object has been modified locally or not. This method returns the value `true` if the object is modified locally.

Delete Objects

Method Introduced:

- **pfcServer::RemoveObjects**

The method `pfcServer::RemoveObjects` deletes the array of objects from the workspace. When passed with the *ModelNames* array as `NULL`, this method removes all the objects in the active workspace.

Conflicts During Server Operations

Method Introduced:

- **pfcXToolkitCheckoutConflict::GetConflictDescription**

An exception is provided to capture the error condition while performing the following server operations using the specified APIs:

Operation	API
Checkin an object or workspace	`pfcServer::CheckinObjects`
Checkout an object	`pfcServer::CheckoutObjects`
Undo checkout of an object	`pfcServer::UndoCheckout`
Upload object	`pfcServer::UploadObjects`
Download object	`pfcServer::CheckoutObjects` (with download as `true`)

Operation	API
Delete workspace	`pfcServerLocation::DeleteWorkspace`
Remove object	`pfcServer::RemoveObjects`

These APIs throw a common exception `pfcXToolkitCheckoutConflict` if an error is encountered during server operations. Use the method `pfcXToolkitCheckoutConflict::GetConflictDescription` to extract details of the error condition. This description is similar to the description displayed by the Creo HTML user interface in the conflict report.

Utility APIs

The methods specified in this section enable you to obtain the handle to the server objects to access them. The handle may be the aliased URL or the model name of the http URL. These utilities enable the conversion of one type of handle to another.

Methods Introduced:

- **pfcServer::GetAliasedUrl**
- **pfcBaseSession::GetModelNameFromAliasedUrl**
- **pfcBaseSession::GetAliasFromAliasedUrl**
- **pfcBaseSession::GetUrlFromAliasedUrl**

The method `pfcServer::GetAliasedUrl` enables you to search for a server object by its name. Specify the complete filename of the object as the input, for example, `test_part.prt`. The method returns the aliased URL for a model on the server. For more information regarding the aliased URL, refer to the section Aliased URL on page 632. During the search operation, the workspace takes precedence over the shared space.

You can also use this method to search for files that are not in the Creo format. For example, `my_text.txt`, `prodev.dat`, `intf_file.stp`, and so on.

The method `pfcBaseSession::GetModelNameFromAliasedUrl` returns the name of the object from the given aliased URL on the server.

The method `pfcBaseSession::GetUrlFromAliasedUrl` converts an aliased URL to a standard URL for the objects on the server.

For example, `wtws://my_alias/Creo Parametric/abcd.prt` is converted to an appropriate URL on the server as `<server.mycompany.com>/Windchill`..

The method `pfcBaseSession::GetAliasFromAliasedUrl` returns the server alias from aliased URL.

38

Technical Summary of Changes

Technical Summary of Changes for Creo 9.0.0.0

This chapter describes the critical and miscellaneous technical changes in Creo 9.0.0.0 and Creo Object TOOLKIT C++. It also lists the new and superseded functions for this release.

New Functions

This section describes new functions for Creo Object TOOLKIT C++ for Creo 9.0.0.0.

Action Listeners

New Function	Description
pfcSelectionBufferListener:: OnAfterSelBufferChange	This method is called after the selection buffer is changed. This method is available by calling the method `pfcActionSource::AddActionListenerWithType` with the value of the notify type as `pfcSELBUFFER_CHANGE_POST`.

Annotations

New Function	Description
wfcSetDatumTag::GetAdditionalText wfcSetDatumTag::SetAdditionalText	Gets and sets the additional text for the specified datum feature tag.
wfcSetDatumTagAdditionalText:: GetPosition wfcSetDatumTagAdditionalText:: SetPosition	Gets and sets the position of the additional text around the frame of the datum feature tag.
wfcSetDatumTagAdditionalText:: GetText wfcSetDatumTagAdditionalText:: SetText	Gets and sets the additional text around the frame of the datum feature tag.
wfcSetDatumTag:: GetAdditionalTextLocation	Retrieves the location of additional text for the specified datum feature tag.
wfcSetDatumTag::GetTextLocation	Retrieves the text point for the specified datum feature tag.
wfcSetDatumTag:: IsShownInDrawingView	Returns the display status of the set datum tag in the specified view of a

New Function	Description
	drawing.
wfcSetDatumTag:: EraseFromDrawingView	Sets a set datum tag to be erased from the specified view of a drawing.
wfcSetDatumTag::GetDimGTolDisplay	Returns display type for datum feature symbol attached to dimension or gtol.
wfcSetDatumTag::SetLabel wfcSetDatumTag::GetLabel	Gets and sets the label for the specified datum feature tag annotation.
wfcSetDatumTag::SetASMEDisplay	Displays the datum feature tag annotation according to the ASME standard.
wfcSetDatumTag::GetASMEDisplay	Checks if the specified datum feature tag annotation is displayed as per ASME standard.
wfcSetDatumTag::SetElbow	Sets or unsets Elbow to datum feature tag annotation.
wfcSetDatumTag::GetElbow	Checks if a specified datum feature tag annotation Elbow is set.
wfcSetDatumTag::GetPlacement	Returns the item type, id, and owner on which the set datum tag is placed.
wfcSetDatumTag::DeleteReference	Deletes semantic references in the specified datum feature tag.
wfcSetDatumTag::GetReferences	Retrieves semantic references in the specified datum feature tag.
wfcSetDatumTag::AddReferences	Adds semantic references to the specified datum feature tag.
wfcSetDatumTag::Delete	Deletes the specified datum feature tag annotation.
wfcWDrawing::ListSetDatumTags	Enables you to visit the set datum tag annotations in the specified drawing.
wfcWModel::CreateSetDatumTag	Create a new set datum tag annotation.
wfcWDetailNoteItem::GetGTol	Returns the detail note that represents a shown geometric tolerance.
wfcAnnotation::IsUsingXSecReference	Identifies if the annotation is created on or attached to a cross-sectional edge.
wfcAnnotation:: GetAnnotationElementWithOption	Gets the annotation element or semi annotation element that contains a given annotation.
wfcWSelection::	Sets user-defined annotation references

New Function	Description
SetAnnotationElementReferences	in the specified annotation elements.
wfcAnnotationReferenceSet::Create	Creates an instance of the `wfcAnnotation.AnnotationReferenceSet` object.
wfcAnnotationReferenceSet::GetReference wfcAnnotationReferenceSet::SetReference	Gets and sets the user-defined annotation references using the `wfcAnnotationReference` object.
wfcAnnotationReferenceSet::GetStrong wfcAnnotationReferenceSet::SetStrong	Gets and sets the strength of the user-defined annotation references.
wfcAnnotationReferenceSet::GetDescription wfcAnnotationReferenceSet::SetDescription	Gets and Sets the description of the user-defined annotation references.
wfcAnnotationReferenceSet::GetAutoPropagate wfcAnnotationReferenceSet::SetAutoPropagate	Gets and sets auto the propagate flag for the user-defined annotation features.
wfcAnnotation::NeedsConversion	Returns true if the annotation is created in releases earlier than Creo Parametric4.0 F000 or is created using the deprecated methods `ProGtolCreate()` or `ProSetdatumtagCreate()` and needs conversion.
wfcAnnotation::ConvertLegacy	Converts annotations to the latest Creo Parametric version.
wfcAnnotationPlane::GetFlip	Specifies if the annotation plane was flipped during creation.
wfcWModel::GetActiveAnnotationPlane	Returns the active annotation plane in the specified model.
wfcAnnotation::GetSecurityMarking wfcAnnotation::SetSecurityMarking	Gets and sets the security marking option for notes and symbols.

Annotations: Geometric Tolerances

New Function	Description
wfcGtol::GetAdditionalTextLocation	Gets the additional text location for the specified type of text.
wfcGTol::GetNotes	Returns the detail notes that represent a geometric tolerance in the specified drawing.
wfcGTolAttachLeader:: GetMissingLeaders	Gets the number of suppressed leaders due to missing references.
wfcGTolLeaderInstructions:: GetIsZExtension	Checks if the gtol leader has a Z-Extension line.
wfcGTolLeaderInstructions:: GetZExtensionPoint	Retrieves the Z-Extension line of the gtol leader.

Dimensions

New Function	Description
wfcWDimension::GetEnvelope	Returns the envelope of a line in the specified dimension.

Element Trees: Sections

New Function	Description
wfcSection::GetSectionDimensions	Returns corresponding arrays of section dimension identifiers and solid dimension identifiers using the class `wfcSectionDimIds`.
wfcSectionDimIds::GetSolidIds	Returns solid dimension identifiers.
wfcSectionDimIds::GetSectionIds	Returns section dimension identifiers.

Solids

New Function	Description
wfcWSolid:: GetAnnotationsOfActiveState	Gets annotations of active combined state.
wfcCombState:: GetStateOfSupplGeometry	Checks if the display of supplementary geometry is controlled by the specified combined state or layers.

New Function	Description
wfcCombState::SetStateOfAnnotations wfcCombState:: SetStateOfSupplGeometry	Allows you to change the display of annotations and supplementary geometry by the combined state or layers.
wfcWRegenInstructions:: SetTopAsmOnly wfcWRegenInstructions:: GetTopAsmOnly	Forces only top level assembly to regenerate.

Miscellaneous Technical Changes

The following changes in Creo 9.0.0.0 can affect the functional behavior of Creo Object TOOLKIT C++. PTC does not anticipate that these changes cause critical issues with existing Creo Object TOOLKIT C++ applications.

Support for Spline Surfaces with 2 Derivatives

The value `pfcSURFACE_SPL2DER` is added to the enumerated data type `pfcSurfaceType`, which provides support for spline surfaces with 2 derivatives.

Support for Datum Curve End in Attachment Selection

The value `pfcITEM_CRV_START` and `pfcITEM_CRV_END` is added to the enumerated data type `pfcModelItemType`, which supports selection of specific dimension attachment.

Obsolete Functions

In Creo Parametric 9.0.0.0 and later, the methods `pfcModelDescriptor::GetHost` and `pfcModelDescriptor::SetHost` are obsoleted and will not have any support.

Full Version of Creo Object TOOLKIT C++ Release Notes

To see a full version of the *Creo Object TOOLKIT C++ Release Notes*, visit the page Creo Object TOOLKIT C++ Release Notes. The full version contains information from all the past release notes for Creo Object TOOLKIT C++.

Technical Summary of Changes for Creo 9.0.4.0

This chapter describes the critical and miscellaneous technical changes in Creo 9.0.4.0 and Creo Object TOOLKIT C++. It also lists the new and superseded functions for this release.

Miscellaneous Technical Changes

The following changes in Creo 9.0.4.0 can affect the functional behavior of Creo Object TOOLKIT C++. PTC does not anticipate that these changes cause critical issues with existing Creo Object TOOLKIT C++ applications.

Rebuild Requirement for Creo Object TOOLKIT C++ Applications

Some functions were obsoleted in Creo 9.0.0.0. As a result, the library file `otk_cpp.lib` was updated. For your Creo Object TOOLKIT C++ applications to run properly, you must rebuild your applications.

Full Version of Creo Object TOOLKIT C++ Release Notes

To see a full version of the *Creo Object TOOLKIT C++ Release Notes*, visit the page Creo Object TOOLKIT C++ Release Notes. The full version contains information from all the past release notes for Creo Object TOOLKIT C++.

A

Creo Object TOOLKIT C++ Registry File

This appendix describes how to use the Registry file to have a foreign program communicate with Creo.

Note

The Creo Object TOOLKIT C++ applications are supported with Creo Parametric.

Registry File

The registry file is a simple text file, where each line consists of one keyword from a predefined set of keywords, followed by a value.

Registry File Fields

The following table lists the fields in the registry file `creotk.dat` or `protk.dat`.

Field	Description
`name`	Assigns a unique name to the Creo Object TOOLKIT C++ application. The name is used to identify the application if there is more than one. The name can be the product name and does not have to be the same as the executable name. This field has a limit of `PRO_NAME_SIZE-1` wide characters (`wchar_t`).
`startup`	Specifies the method Creo should use to communicate with the Creo Object TOOLKIT C++ application. This field can take one of two values; `spawn` or `dll`. • `spawn`—If the value is `spawn`, Creo starts the foreign program using interprocess communications. • `dll`—If the value is `dll`, Creo loads the foreign program as a DLL. The default value is `spawn`.
`toolkit`	Specifies the name of the Toolkit which was used to create the customization. The valid values for this field are `object` and `protoolkit`. An application created in Creo Object TOOLKIT C++ must always have the value of this field set as `object`. **Note** This field can also be used to indicate other toolkits. Its default value is `protoolkit`, which specifies that the customizing application was created in Creo TOOLKIT. If you set the value for this field as `protoolkit`, or omit this field, then the application can be used only with Creo Parametric.
`creo_type`	Specifies the Creo applications that support the Creo Object TOOLKIT C++ applications. The valid values for this field are: • `parametric`—This is the default value. Specify `parametric` to load the Creo Object TOOLKIT C++ application in Creo Parametric. **Note** Other Creo applications will be supported in future releases.
`fail_tol`	Specifies the action of Creo if the call to `user_initialize()` in the foreign program returns non-zero, or if the foreign program

Field	Description
	subsequently fails. If this is TRUE, Creo continues as normal. If this field is missing or is set to FALSE, Creo shuts down the application and other foreign programs.
exec_file	Specifies the full path and name of the file produced by compiling and linking the Creo Object TOOLKIT C++ application. In DLL mode, this is a dynamically linkable library; in spawn mode, it is a complete executable. This field has a limit of PRO_PATH_SIZE-1 wide characters (wchar_t).
text_dir	Specifies the full path name to text directory that contains the language-specific directories. The language-specific directories contain the message files, menu files, resource files and UI bitmaps in the language supported by the Creo Object TOOLKIT C++ application. Please refer to the Ribbon Tabs, Groups, and Menu Items on page 82 chapter for more information. The text_dir does not need to include the trailing /text; it is added automatically by Creo. The search priority for messages and menu files is as follows: 1. Current working directory 2. text_dir\text 3. <creo_loadpoint>\Common Files\<datecode>\<machine type>\text The text_dir should be different from the Creo Parametric text tree. This field has a limit of PRO_PATH_SIZE-1 wide characters (wchar_t).
rbn_path	Specifies the name of the ribbon file along with its path, which must be loaded when you open the Creo application. The location of the ribbon file is relative to the location of the text directory. The field text_dir specifies the path for the text directory. If the field is not specified, by default, the ribbon file with its location, text_dir/toolkitribbonui.rbn is used.
delay_start	If you set this to TRUE, Creo does not invoke the Creo Object TOOLKIT C++ application as it starts up, but enables you to choose when to start the application. If this field is missing or is set to FALSE, the Creo Object TOOLKIT C++ application starts automatically.
description	Acts as a help line for your auxiliary application. If you leave the cursor on an application in the Start/Stop GUI, Creo displays the description text (up to 80 characters). You can use non-ASCII characters, as in menu files. To make the description appear in multiple languages, you must use separate protk.dat files in <hierarchy>/<platform>/<text>/<language>.

Field	Description
allow_stop	If you set this to TRUE, you can stop the application during the session. If this field is missing or is set to FALSE, you cannot stop the application, regardless of how it was started.
end	Indicates the end of the description of the Creo Object TOOLKIT C++ application. It is possible to add further statements that define other foreign applications. All of these applications are initialized by Creo.

Sample Registry Files

This section shows an example that illustrates the various ways to have a foreign program communicate with Creo.

Example 1

File: creotk.dat

[Start of file on next line]

```
NAME          otk_examples
TOOLKIT                      OBJECT [optional; omitted means
protoolkit]
CREO_TYPE               DIRECT [optional; omitted means
PARAMETRIC]
EXEC_FILE                 otk_examples_dll.dll
TEXT_DIR                   ./text
STARTUP                   dll [dll/spawn/java]
END
```

[End of file on previous line]

B

Creo Object TOOLKIT C++ Library Types

This appendix describes the various libraries available in a Creo Object TOOLKIT C++ installation.

Overview

The libraries available in a Creo Object TOOLKIT C++ installation have been classified under:

- Standard Libraries on page 658
- Alternate Libraries on page 658

From Creo 4.0 F000 onward, the libraries listed in the following table are no longer supported and will not be available with the software. The New Library Name column provides a list of the equivalent libraries that are now available. Apart from the compatibility issues explained in the chapter Version Compatibility: Creo Parametric and Creo Object TOOLKIT C++ on page 44, applications based on Creo 3.0 and previous versions will continue to run successfully with Creo 4.0.

Old Library Name	New Library Name
protk_dll.lib	protk_dll_NU.lib
protoolkit.lib	protoolkit_NU.lib
protk_dllmd.lib	protk_dllmd_NU.lib
protkmd.lib	protkmd_NU.lib

Linking the Applications

Before you run an existing application in Creo Parametric 8.0, link it to the new libraries and the import libraries: `ucore.lib` and `udata.lib`. The new libraries do not link the application to the Unicode libraries but use `ucore.lib` and `udata.lib` at runtime to resolve the Unicode dependencies, which also reduces the size of the application. Since these libraries are import libraries, the application must resolve Unicode dependencies at runtime by loading the actual libraries `ucore64.dll` and `udata64.dll`. These dlls are located at `<creo_load_point>/Common Files/<platform>/obj` and `<creo_load_point>/Common Files/<platform>/lib`.

For synchronous applications, the references to `ucore64.dll` and `udata64.dll` are resolved by Creo Parametric when the application is started.

For asynchronous applications, these references must be resolved by the application. For linking asynchronous applications, add the path where the dlls `ucore64.dll` and `udata64.dll` are located, that is `<creo_load_point>/Common Files/<platform>/lib` to the environment variable `PATH`.

All the sample makefiles available with Creo Object TOOLKIT C++ use the new libraries. For instance, the sample example `make_otk_examples` created for Creo Object TOOLKIT C++ applications contains information on how to use `protk_dll_NU.lib`. The sample file is located at `<creo_otk_loadpoint>/<platform>/obj`.

Standard Libraries

Creo Object TOOLKIT C++ applications will be linked with the following libraries:

Library Name	Purpose
`protoolkit_NU.lib otk_cpp.lib ucore.lib udata.lib`	Spawn mode library
`pt_asynchronous.lib protoolkit_NU.lib otk_cpp_async.lib`	Asynchronous mode library
`protk_dll_NU.lib otk_cpp.lib ucore.lib udata.lib`	DLL mode library
`otk_222.lib`	If you have the advanced license option 222, include this library in your application.
`otk_no222.lib`	If you do not have the advanced license option 222, include this library in your application.

A specific platform will also require various system libraries in the application link list. Please refer to the makefiles in the folder `<creo_otk_loadpoint_app>\<platform>\obj`.

Alternate Libraries

Creo Object TOOLKIT C++ provides alternate libraries, `otk_cpp_md.lib` and `otk_cpp_async_md.lib` that may be used for applications compiled with `/MD` flag and built with `msvcrt.lib`. These libraries must be used together with alternate libraries from Creo TOOLKIT:

- `otk_cpp_md.lib` with `protkmd.lib` for IPC mode
- `protk_dllmd_NU.lib` for DLL mode,
- `otk_cpp_async_md.lib` with `ptasyncmd.lib` and `protkmd_NU.lib`

Makefile `make_otk_examples_md` shows how to use the library `otk_cpp_md.lib` and `make_otk_async_examples_md` shows how to use `otk_cpp_async_md.lib`.

Note

- Although `/MD` provides compatibility with multi-threaded components, Creo Object TOOLKIT C++ calls must be made within a single thread. Creo does not respond to calls made from multiple threads. Extra threads may be created by applications only to do tasks which do not directly call Creo Object TOOLKIT C++ functions.
- Compiling Creo Object TOOLKIT C++ applications with `/MTd` and `/MDd` flags is not supported.

C

Advanced Licensing Options

This appendix describes the licensing requirements for advanced options in Creo Object TOOLKIT C++.

Advance Licensing Options for Creo Object TOOLKIT C++

To use some of the functionality in Creo Object TOOLKIT C++ you must have advanced development license options.

For every function that requires an advanced license, the comment “LICENSE: 222” has been added in the Creo Object TOOLKIT C++ APIWizard. Advanced licenses are required in the following situations. These are applicable when calls are made to Creo Object TOOLKIT C++ or Creo TOOLKIT.

- To run a locked application, Creo Parametric requires the basic toolkit option. Advanced toolkit option is required by specific functions called by the application. If the application contains calls to such functions, Creo Parametric checks out the corresponding advanced license option on demand.
- To unlock an application, the unlock utility requires the basic toolkit option and any advanced toolkit options required by specific functions called by the application. The utility will not hold any of the advanced options, as it does the basic Creo Object TOOLKIT C++ option, after unlock is completed.
- Creo Parametric does not require any of the toolkit licenses to run a properly unlocked application.

Applications are assigned requirements for advanced options based on whether the application is coded to use any functions requiring the advanced option. It does not matter if an application does not use the function that requires licensing during a particular invocation of the application. The licensing requirements are resolved the moment the application is started by or connects to Creo Parametric, not at the first time an advanced function is invoked.

For more information on how to unlock an application, refer to the section Unlocking the Creo Object TOOLKIT C++ Application on page 17.

D

Sample Applications

This appendix describes the sample applications provided with Creo Object TOOLKIT C++.

Installing Creo Object TOOLKIT C++

Creo Object TOOLKIT C++ is a part of the Creo installation. When you install the product, one of the optional components is **API Toolkits**. In **API Toolkits**, select Creo Object TOOLKIT C++ to install it.

The header files, platform-specific libraries, example applications, and documentation, specific to Creo Object TOOLKIT C++ are installed in the loadpoints as explained below :

- Application loadpoint `<creo_otk_loadpoint_app>`—The location is `<creo_loadpoint>\<datecode>\Common Files\otk\otk_cpp`. The Creo Object TOOLKIT C++ application files, such as, platform-specific libraries, header files, example applications, and so on, are installed in this location.
- Documentation loadpoint `<creo_otk_loadpoint_doc>`—The location is `<creo_loadpoint>\<datecode>\Common Files\otk_cpp_doc`. The Creo Object TOOLKIT C++ documentation files such as, *Creo Object TOOLKIT User's Guide*, Creo Object TOOLKIT C++ APIWizard, and so on, are installed in this location.

In Creo 6.0.0.0 and later, these sample applications are digitally signed.

Sample Applications

The Creo Object TOOLKIT C++ sample applications are available at `<creo_otk_loadpoint_app>` for both synchronous and asynchronous modes.

otk_examples

The application `otk_examples` is a collection of example source files for *Creo Object TOOLKIT C++ User's Guide*. The examples cover most of the Creo Object TOOLKIT C++ functionality for synchronous mode of application.

For example, the `otk_examples_drw` application demonstrates the drawing functionality.

The example `otk_examples_cip` demonstrates basic operations in Creo Object TOOLKIT C++ using `xstring`.

otk_async_examples

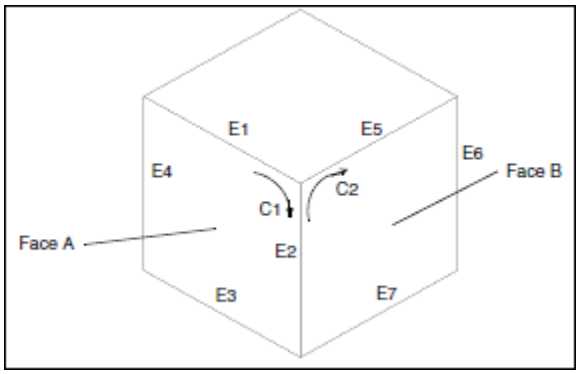
The application `otk_async_examples` is a collection of example source files for *Creo Object TOOLKIT C++ User's Guide* for asynchronous mode of application.

E

Geometry Traversal

This appendix illustrates the relationships between faces, contours, and edges. Examples E-1 through E-5 show some sample parts and list the information about their surfaces, faces, contours, and edges.

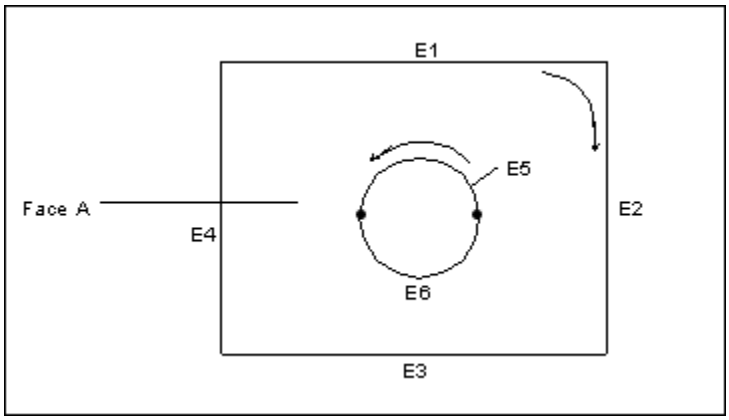
Example 1



This part has 6 faces.

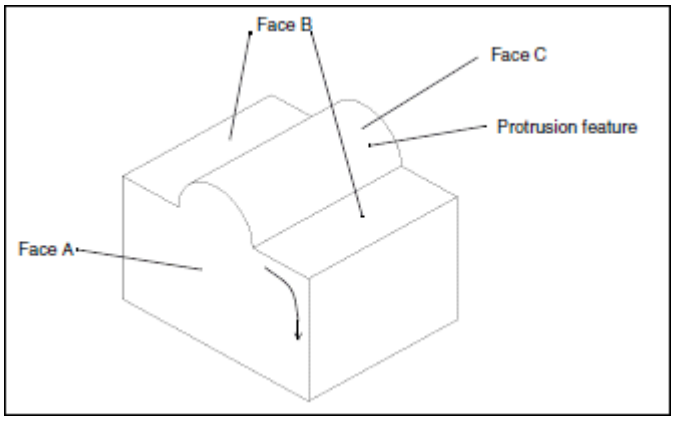
- Face A has 1 contour and 4 edges.
- Edge E2 is the intersection of faces A and B.
- Edge E2 is a component of contours C1 and C2.

Example 2



Face A has 2 contours and 6 edges.

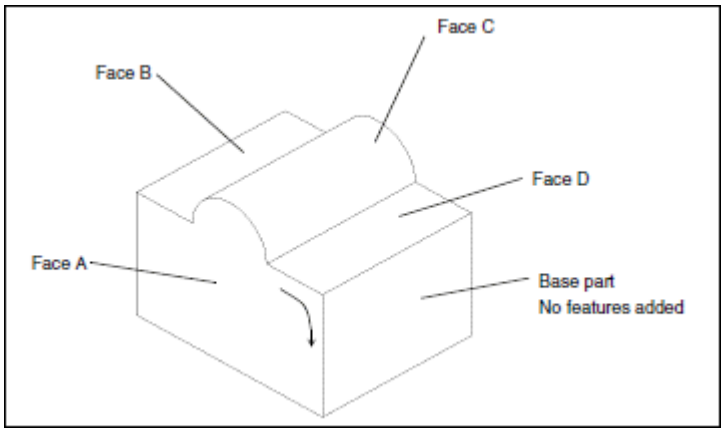
Example 3



This part was extruded from a rectangular cross section. The feature on the top was added later as an extruded protrusion in the shape of a semicircle.

- Face A has 1 contour and 6 edges.
- Face B has 2 contours and 8 edges.
- Face C has 1 contour and 4 edges.

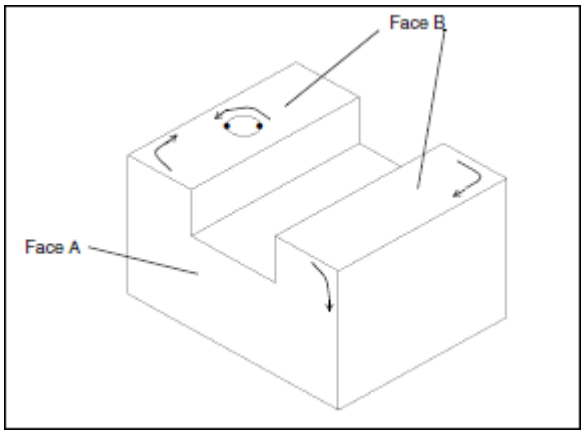
Example 4



This part was extruded from a cross section identical to Face A. In the Sketcher, the top boundary was sketched with two lines and an arc. The sketch was then extruded to form the base part, as shown.

- Face A has 1 contour and 6 edges.
- Face B has 1 contour and 4 edges.
- Face C has 1 contour and 4 edges.
- Face D has 1 contour and 4 edges.

Example 5



This part was extruded from a rectangular cross section. The slot and hole features were added later.

- Face A has 1 contour and 8 edges.
- Face B has 3 contours and 10 edges.

F

Geometry Representations

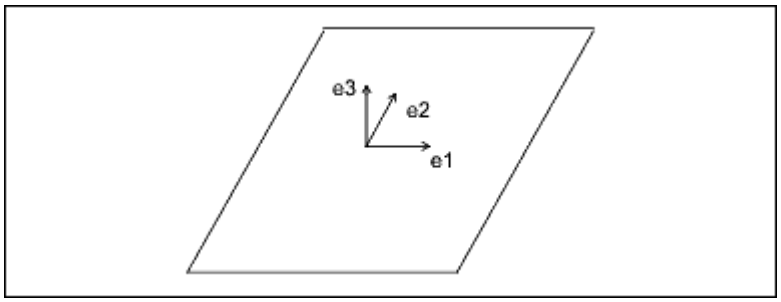
This appendix describes the geometry representations of the data used by Creo Object TOOLKIT C++.

Surface Parameterization

A surface in Creo contains data that describes the boundary of the surface, and a pointer to the primitive surface on which it lies. The primitive surface is a three-dimensional geometric surface parameterized by two variables (*u* and *v*). The surface boundary consists of closed loops (contours) of edges. Each edge is attached to two surfaces, and each edge contains the *u* and *v* values of the portion of the boundary that it forms for both surfaces. Surface boundaries are traversed clockwise around the outside of a surface, so an edge has a direction in each surface with respect to the direction of traversal.

This section describes the surface parameterization. The surfaces are listed in order of complexity. For ease of use, the alphabetical listing of the data structures is as follows:

Plane



The plane entity consists of two perpendicular unit vectors (`e1` and `e2`), the normal to the plane (`e3`), and the origin of the plane.

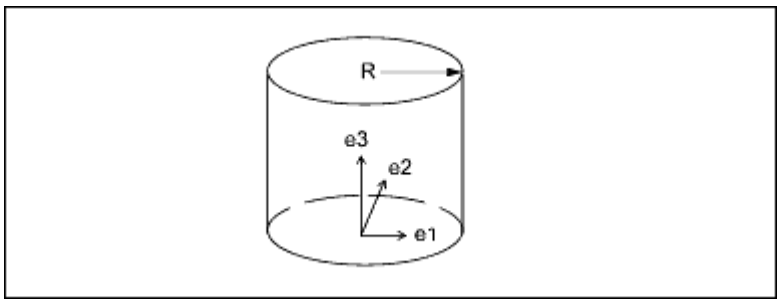
Data Format:

```
e1[3]       Unit vector, in the u direction
e2[3]       Unit vector, in the v direction
e3[3]       Normal to the plane
origin[3]   Origin of the plane
```

Parameterization:

```
(x, y, z) = u * e1 + v * e2 + origin
```

Cylinder



The generating curve of a cylinder is a line, parallel to the axis, at a distance `R` from the axis. The radial distance of a point is constant, and the height of the point is `v`.

Data Format:

```
e1[3]       Unit vector, in the u direction
e2[3]       Unit vector, in the v direction
e3[3]       Normal to the plane
origin[3]   Origin of the cylinder
radius      Radius of the cylinder
```

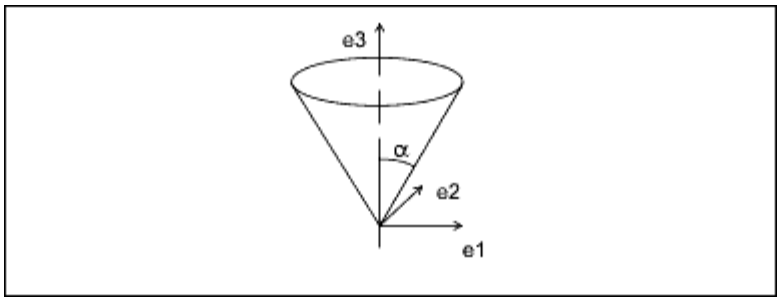
Parameterization:

```
(x, y, z) = radius * [cos(u) * e1 + sin(u) * e2] +
            v * e3 + origin
```

Engineering Notes:

For the cylinder, cone, torus, and general surface of revolution, a local coordinate system is used that consists of three orthogonal unit vectors (`e1`, `e2`, and `e3`) and an origin. The curve lies in the plane of `e1` and `e3`, and is rotated in the direction from `e1` to `e2`. The `u` surface parameter determines the angle of rotation, and the `v` parameter determines the position of the point on the generating curve.

Cone



The generating curve of a cone is a line at an angle alpha to the axis of revolution that intersects the axis at the origin. The `v` parameter is the height of the point along the axis, and the radial distance of the point is `v * tan(alpha)`.

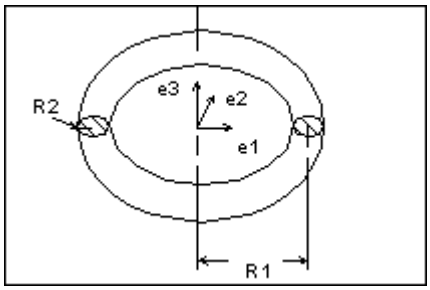
Data Format:

```
e1[3]      Unit vector, in the u direction
e2[3]      Unit vector, in the v direction
e3[3]      Normal to the plane
origin[3]  Origin of the cone
alpha      Angle between the axis of the cone
           and the generating line
```

Parameterization:

```
(x, y, z) = v * tan(alpha) * [cos(u) * e1 +
            sin(u) * e2] + v * e3 + origin
```

Torus



The generating curve of a torus is an arc of radius `R2` with its center at a distance `R1` from the origin. The starting point of the generating arc is located at a distance `R1 + R2` from the origin, in the direction of the first vector of the local coordinate system. The radial distance of a point on the torus is `R1 + R2 * cos(v)`, and the height of the point along the axis of revolution is `R2 * sin(v)`.

Data Format:

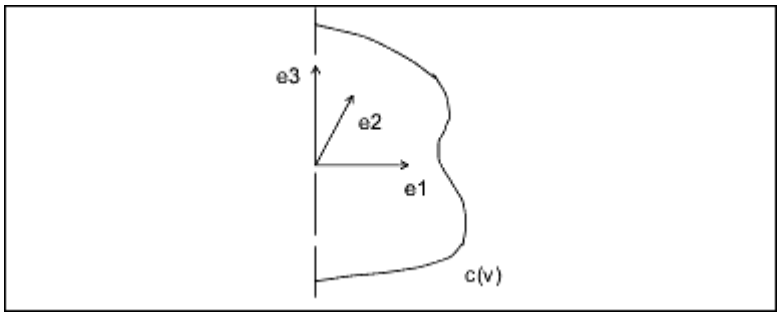
```
e1[3]      Unit vector, in the u direction
e2[3]      Unit vector, in the v direction
e3[3]      Normal to the plane
origin[3]  Origin of the torus
radius1    Distance from the center of the
```

```
           generating arc to the axis of
           revolution
radius2    Radius of the generating arc
```

Parameterization:

```
(x, y, z) = (R1 + R2 * cos(v)) * [cos(u) * e1 +
            sin(u) * e2] + R2 * sin(v) * e3 +
            origin
```

General Surface of Revolution



A general surface of revolution is created by rotating a curve entity, usually a spline, around an axis. The curve is evaluated at the normalized parameter `v`, and the resulting point is rotated around the axis through an angle `u`. The surface of revolution data structure consists of a local coordinate system and a curve structure.

Data Format:

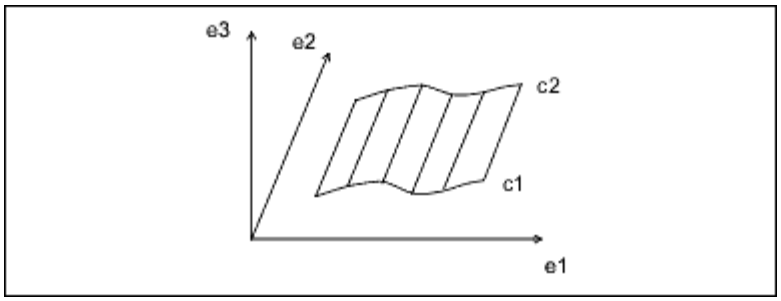
```
e1[3]       Unit vector, in the u direction
e2[3]       Unit vector, in the v direction
e3[3]       Normal to the plane
origin[3]   Origin of the surface of revolution
curve       Generating curve
```

Parameterization:

```
curve(v) = (c1, c2, c3) is a point on the curve.

(x, y, z) = [c1 * cos(u) - c2 * sin(u)] * e1 +
            [c1 * sin(u) + c2 * cos(u)] * e2 +
            c3 * e3 + origin
```

Ruled Surface



A ruled surface is the surface generated by interpolating linearly between corresponding points of two curve entities. The `u` coordinate is the normalized parameter at which both curves are evaluated, and the `v` coordinate is the linear parameter between the two points. The curves are not defined in the local coordinate system of the part, so the resulting point must be transformed by the local coordinate system of the surface.

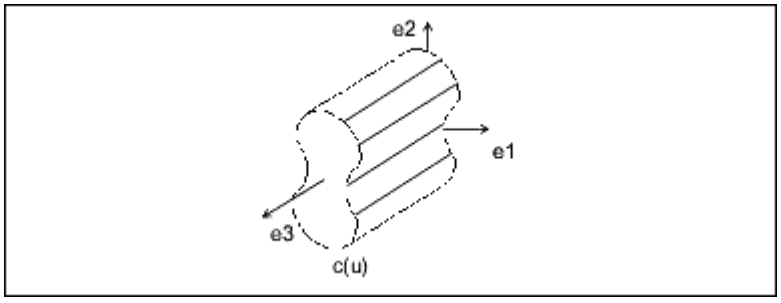
Data Format:

```
e1[3]      Unit vector, in the u direction
e2[3]      Unit vector, in the v direction
e3[3]      Normal to the plane
origin[3]  Origin of the ruled surface
curve_1    First generating curve
curve_2    Second generating curve
```

Parameterization:

```
(x', y', z') is the point in local coordinates.
(x', y', z') = (1 - v) * C1(u) + v * C2(u)
(x, y, z) = x' * e1 + y' * e2 + z' * e3 + origin
```

Tabulated Cylinder



A tabulated cylinder is calculated by projecting a curve linearly through space. The curve is evaluated at the `u` parameter, and the `z` coordinate is offset by the `v` parameter. The resulting point is expressed in local coordinates and must be transformed by the local coordinate system to be expressed in part coordinates.

Data Format:

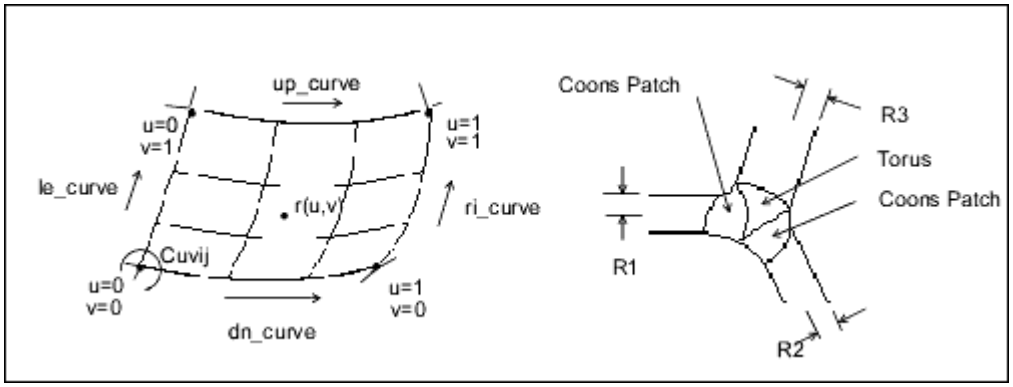
```
e1[3]      Unit vector, in the u direction
```

```
e2[3]       Unit vector, in the v direction
e3[3]       Normal to the plane
origin[3]   Origin of the tabulated cylinder
curve       Generating curve
```

Parameterization:

```
(x', y', z') is the point in local coordinates.
(x', y', z') = C(u) + (0, 0, v)
(x, y, z) = x' * e1 + y' * e2 + z' * e3 + origin
```

Coons Patch

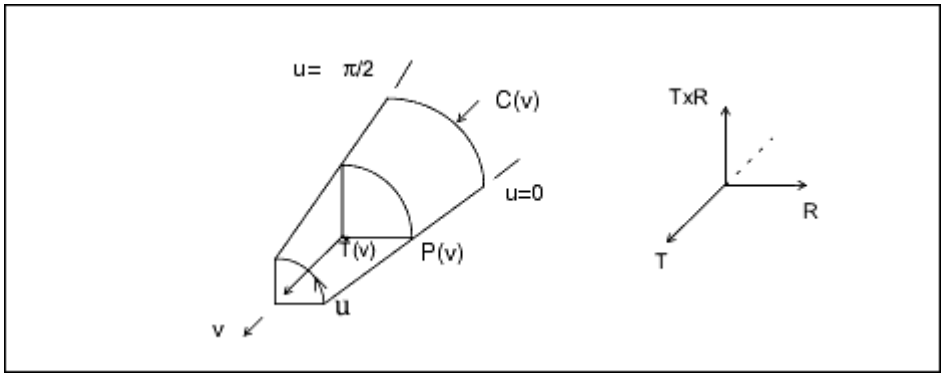


A Coons patch is used to blend surfaces together. For example, you would use a Coons patch at a corner where three fillets (each of a different radius) meet.

Data Format:

```
le_curve                u = 0 boundary
ri_curve                u = 1 boundary
dn_curve                v = 0 boundary
up_curve                v = 1 boundary
point_matrix[2][2]      Corner points
uvder_matrix[2][2]      Corner mixed derivatives
```

Fillet Surface



A fillet surface is found where a round or a fillet is placed on a curved edge, or on an edge with non-constant arc radii. On a straight edge, a cylinder would be used to represent the fillet.

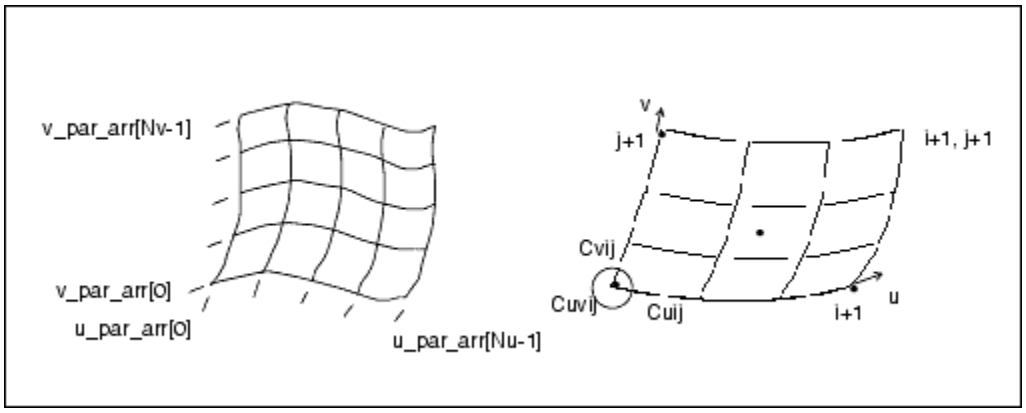
Data Format:

```
pnt_spline   P(v) spline running along the u = 0 boundary
ctr_spline   C(v) spline along the centers of the
             fillet arcs
tan_spline   T(v) spline of unit tangents to the
             axis of the fillet arcs
```

Parameterization:

```
R(v) = P(v) - C(v)
(x,y,z) = C(v) + R(v) * cos(u) + T(v) X R(v) *
          sin(u)
```

Spline Surface



The parametric spline surface is a nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. The grid is curvilinear in uv space. Use this for bicubic blending between corner points.

Data Format:

```
u_par_arr[]     Point parameters, in the u
                direction, of size Nu
v_par_arr[]     Point parameters, in the v
                direction, of size Nv
point_arr[][3]  Array of interpolant points, of
                size Nu x Nv
u_tan_arr[][3]  Array of u tangent vectors
                at interpolant points, of size
                Nu x Nv
v_tan_arr[][3]  Array of v tangent vectors at
                interpolant points, of size
                Nu x Nv
uvder_arr[][3]  Array of mixed derivatives at
                interpolant points, of size
```

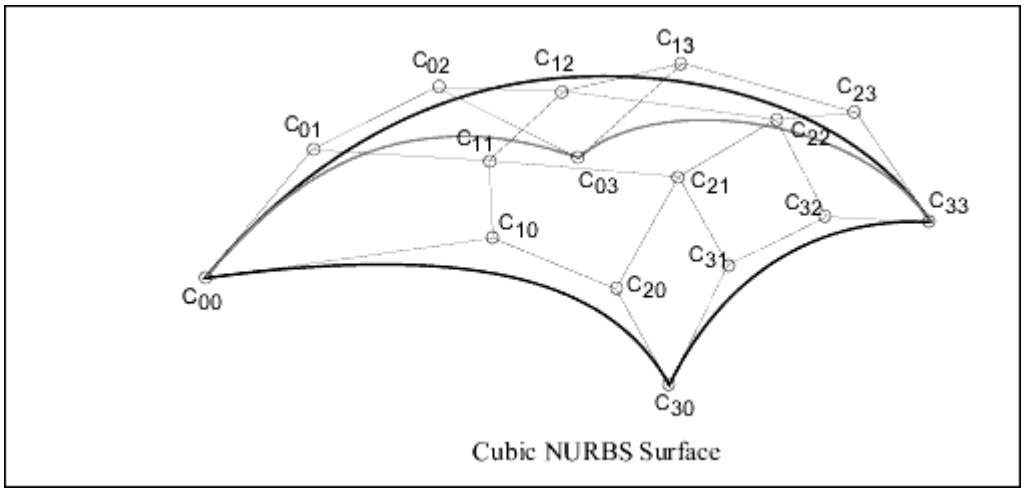
```
Nu x Nv
```

Engineering Notes:

- Allows for a unique 3x3 polynomial around every patch.
- There is second order continuity across patch boundaries.
- The point and tangent vectors represent the ordering of an array of `[i][j]`, where u varies with `j`, and `v` varies with `j`. In walking through the `point_arr[][3]`, you will find that the innermost variable representing `v(j)` varies first.

NURBS Surface

The NURBS surface is defined by basis functions (in `u` and `v`), expandable arrays of knots, weights, and control points.



Cubic NURBS Surface

Data Format:

```
deg[2]              Degree of the basis
                    functions (in u and v)
u_par_arr[]         Array of knots on the
                    parameter line u
v_par_arr[]         Array of knots on the
                    parameter line v
wghts[]             Array of weights for
                    rational NURBS, otherwise
                    NULL
c_point_arr[][3]    Array of control points
```

Definition:

$$R(u, v) = \frac{\sum_{i=0}^{N1}\sum_{j=0}^{N2} C_{i,j} \times B_{i,k}(u) \times B_{j,l}(v)}{\sum_{i=0}^{N1}\sum_{j=0}^{N2} w_{i,j} \times B_{i,k}(u) \times B_{j,l}(v)}$$

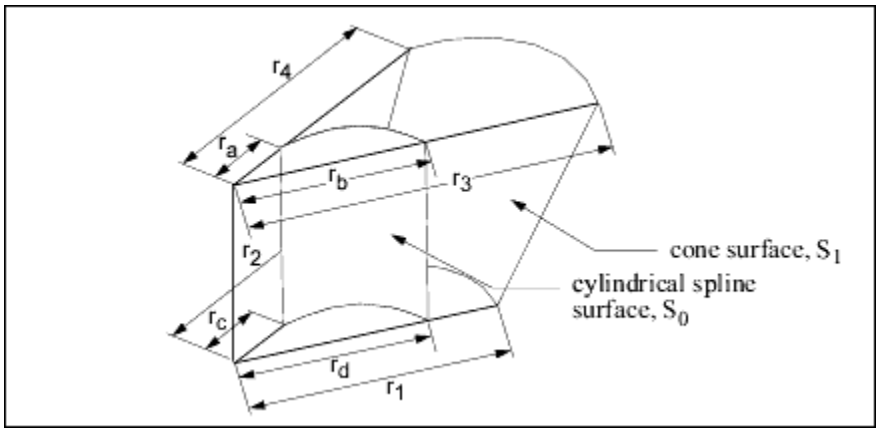
```
k   = degree in u
l   = degree in v
N1  = (number of knots in u) - (degree in u) - 2
N2  = (number of knots in v) - (degree in v) - 2
Bi,k = basis function in u
Bj, l  = basis function in v
wij  = weights
Ci, j   = control points (x,y,z) * wi,j
```

Engineering Notes:

The `weights` and `c_points_arr` arrays represent matrices of size `wghts[N1+1] [N2+1]` and `c_points_arr [N1+1] [N2+1]`. Elements of the matrices are packed into arrays in row-major order.

Cylindrical Spline Surface

The cylindrical spline surface is a nonuniform bicubic spline surface that passes through a grid with tangent vectors given at each point. The grid is curvilinear in modeling space.



Data Format:

```
e1[3]       x' vector of the local coordinate
            system
e2[3]       y' vector of the local coordinate
            system
e3[3]       z' vector of the local coordinate
            system, which corresponds to the
            axis of revolution of the surface
```

```
origin[3]  Origin of the local coordinate
           system
splsrf     Spline surface data structure
```

The spline surface data structure contains the following fields:

```
u_par_arr[]          Point parameters, in the
                     u direction, of size Nu
v_par_arr[]          Point parameters, in the
                     v direction, of size Nv
point_arr[][3]       Array of points, in
                     cylindrical coordinates,
                     of size Nu x Nv. The array
                     components are as follows:
                         point_arr[i][0] - Radius
                         point_arr[i][1] - Theta
                         point_arr[i][2] - Z
u_tan_arr[][3]       Array of u tangent vectors.
                     in cylindrical coordinates,
                     of size Nu x Nv
v_tan_arr[][3]       Array of v tangent vectors,
                     in cylindrical coordinates,
                     of size Nu x Nv
uvder_arr[][3]       Array of mixed derivatives,
                     in cylindrical coordinates,
                     of size Nu x Nv
```

Engineering Notes:

If the surface is represented in cylindrical coordinates (`r`, `theta`, `z`), the local coordinate system values (`x'`,`y'`, `z'`) are interpreted as follows:

```
x' = r cos (theta)
y' = r sin (theta)
z' = z
```

A cylindrical spline surface can be obtained, for example, by creating a smooth rotational blend (shown in the figure on the previous page).

In some cases, you can replace a cylindrical spline surface with a surface such as a plane, cylinder, or cone. For example, in the figure, the cylindrical spline surface S_1 was replaced with a cone (`r1 = r2, r3 = r4, and r1 ≠ r3`).

If a replacement cannot be done (such as for the surface S_0 in the figure (`ra ≠ rb or rc ≠ rd`)), leave it as a cylindrical spline surface representation.

Edge and Curve Parameterization

This parameterization represents edges (line, arc, and spline) as well as the curves (line, arc, spline, and NURBS) within the surfaces.

This section describes edges and curves, arranged in order of complexity. For ease of use, the alphabetical listing is as follows:

- Arc on page 678

- Line on page 678
- NURBS on page 679
- Spline on page 678

Line

Data Format:

```
end1[3]   Starting point of the line
end2[3]   Ending point of the line
```

Parameterization:

```
(x, y, z) = (1 - t) * end1 + t * end2
```

Arc

The arc entity is defined by a plane in which the arc lies. The arc is centered at the origin, and is parameterized by the angle of rotation from the first plane unit vector in the direction of the second plane vector. The start and end angle parameters of the arc and the radius are also given. The direction of the arc is counterclockwise if the start angle is less than the end angle, otherwise it is clockwise.

Data Format:

```
vector1[3]    First vector that defines the
              plane of the arc
vector2[3]    Second vector that defines the
              plane of the arc
origin[3]     Origin that defines the plane
              of the arc
start_angle   Angular parameter of the starting
              point
end_angle     Angular parameter of the ending
              point
radius        Radius of the arc.
```

Parameterization:

```
t' (the unnormalized parameter) is
   (1 - t) * start_angle + t * end_angle
(x, y, z) = radius * [cos(t') * vector1 +
   sin(t') * vector2] + origin
```

Spline

The spline curve entity is a nonuniform cubic spline, defined by a series of three-dimensional points, tangent vectors at each point, and an array of unnormalized spline parameters at each point.

Data Format:

```
par_arr[]     Array of spline parameters
              (t) at each point.
pnt_arr[][3]  Array of spline interpolant points
tan_arr[][3]  Array of tangent vectors at
              each point
```

Parameterization:

`x`, `y`, and `z` are a series of unique cubic functions, one per segment, fully determined by the starting and ending points, and tangents of each segment.

Let `p_max` be the parameter of the last spline point. Then, `t`, the unnormalized parameter, is `t * p_max`.

Locate the `i`th spline segment such that:

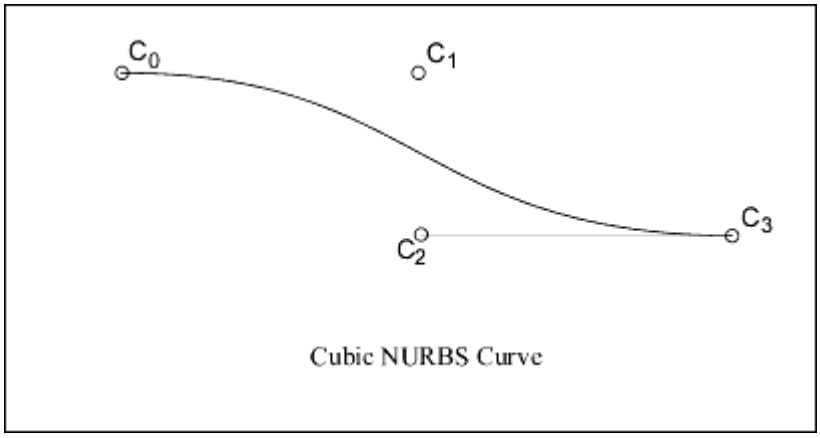
```
par_arr[i] < t' < par_arr[i+1]
```

(If `t` < 0 or `t` > +1, use the first or last segment.)

```
t0 = (t' - par_arr[i])   / (par_arr[i+1] - par_arr[i])
t1 = (par_arr[i+1] - t') / (par_arr[i+1] - par_arr[i])
```

NURBS

The NURBS (nonuniform rational B-spline) curve is defined by expandable arrays of knots, weights, and control points.

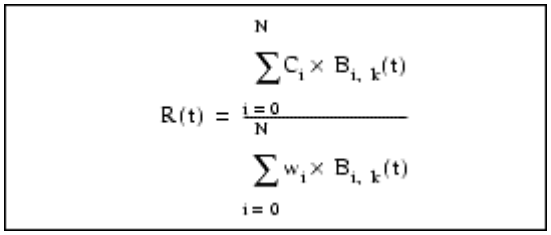


Cubic NURBS Curve

Data Format:

```
degree        Degree of the basis function
params[]      Array of knots
weights[]     Array of weights for rational
              NURBS, otherwise NULL.
c_pnts[][3]   Array of control points
```

Definition:



$$R(t) = \frac{\sum_{i=0}^{N} C_i \times B_{i,k}(t)}{\sum_{i=0}^{N} w_i \times B_{i,k}(t)}$$

```
k     = degree of basis function
N    = (number of knots) - (degree) - 2
wi   = weights
Ci    = control points (x, y, z) * wi
Bi,k = basis functions
```

By this equation, the number of control points equals `N+1`.

References:

Faux, I.D., M.J. Pratt. Computational Geometry for Design and Manufacture. Ellis Harwood Publishers, 1983.

Mortenson, M.E. Geometric Modeling. John Wiley & Sons, 1985.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W