Copyright © 2013 PTC Inc. and/or Its Subsidiary Companies. All Rights Reserved.

User and training guides and related documentation from PTC Inc. and its subsidiary companies (collectively "PTC") are subject to the copyright laws of the United States and other countries and are provided under a license agreement that restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed software user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the PTC copyright notice and any other proprietary notice provided by PTC. Training materials may not be copied without the express written consent of PTC. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of PTC and no authorization is granted to make copies for such purposes.

Information described herein is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC.

UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION. PTC regards software piracy as the crime it is, and we view offenders accordingly. We do not tolerate the piracy of PTC software products, and we pursue (both civilly and criminally) those who do so using all legal means available, including public and private surveillance resources. As part of these efforts, PTC uses data monitoring and scouring technologies to obtain and transmit data on users of illegal copies of our software. This data collection is not performed on users of legally licensed software from PTC and its authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

**Important Copyright, Trademark, Patent, and Licensing Information:** See the About Box, or copyright notice, of your PTC software.

**UNITED STATES GOVERNMENT RESTRICTED RIGHTS LEGEND**

This document and the software described herein are Commercial Computer Documentation and Software, pursuant to FAR 12.212(a)-(b) (OCT’95) or DFARS 227.7202-1(a) and 227.7202-3(a) (JUN’95), and are provided to the US Government under a limited commercial license only. For procurements predating the above clauses, use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 (OCT’88) or Commercial Computer Software-Restricted Rights at FAR 52.227-19(c)(1)-(2) (JUN’87), as applicable. 01282013

**PTC Inc., 140 Kendrick Street, Needham, MA 02494 USA**
Contents

About This Guide ................................................................................................................................................. 9

Overview of Object TOOLKIT C++ .................................................................................................................. 13
  Setting Up Object TOOLKIT C++ .................................................................................................................. 14
  Installing Object TOOLKIT C++ .................................................................................................................. 14
  Unlocking the Object TOOLKIT C++ Application ....................................................................................... 14
  Object TOOLKIT C++ Registry File ............................................................................................................. 14
  How Object TOOLKIT C++ Works ................................................................................................................ 14
  Categories of Object TOOLKIT C++ Classes ............................................................................................... 16
  Using Object TOOLKIT C++ with Creo Parametric TOOLKIT .................................................................. 27
  Object TOOLKIT C++ Support for Creo Applications ................................................................................ 28
  Creating Applications ................................................................................................................................ 28
  Version Compatibility: Creo Parametric and Object TOOLKIT C++ ........................................................ 33
  Retrieving Creo Parametric Datecode ........................................................................................................ 34
  Threading in Object TOOLKIT C++ Applications ....................................................................................... 34

The Object TOOLKIT C++ Online Browser .................................................................................................... 35
  Online Documentation for Object TOOLKIT C++ APIWizard .................................................................. 36

Session Objects .................................................................................................................................................. 47
  Overview of Session Objects ......................................................................................................................... 48
  Getting the Session Object ............................................................................................................................ 48
  Directories ...................................................................................................................................................... 49
  Initializing Objects ....................................................................................................................................... 51
  Accessing the Creo Parametric Interface .................................................................................................... 52

Selection ........................................................................................................................................................... 63
  Interactive Selection ..................................................................................................................................... 64
  Accessing Selection Data .............................................................................................................................. 65
  Programmatic Selection ............................................................................................................................... 66
  Selection Buffer .......................................................................................................................................... 68

Ribbon Tabs, Groups, and Menu Items .............................................................................................................. 71
  Creating Ribbon Tabs, Groups, and Menu Items .......................................................................................... 72
  About the Ribbon Definition File ................................................................................................................ 73
  Localizing the Ribbon User Interface Created by the Object TOOLKIT C++ Applications ................................ 77

Menus, Commands, and Pop-up Menus ............................................................................................................. 79
  Introduction .................................................................................................................................................. 80
  Menu Bar Definitions .................................................................................................................................. 80
  Menu Buttons and Menus ............................................................................................................................. 80
  Designating Commands ............................................................................................................................... 83
  Pop-up Menus .............................................................................................................................................. 85

Models ............................................................................................................................................................... 89
  Overview of Model Objects ......................................................................................................................... 90
Getting a Model Object .......................................................... 90
Model Descriptors ................................................................... 90
Retrieving Models ................................................................. 91
Model Information ................................................................ 92
Model Operations ................................................................ 95
Running ModelCHECK ......................................................... 97

Drawings .............................................................................. 103
Overview of Drawings in Object TOOLKIT C++ .................... 104
Creating Drawings from Templates ........................................ 104
Obtaining Drawing Models .................................................. 106
Drawing Information ............................................................. 107
Drawing Operations ............................................................. 107
Drawing Sheets .................................................................... 108
Drawing Views ...................................................................... 111
Drawing Dimensions ........................................................... 118
Drawing Tables ..................................................................... 124
Detail Items .......................................................................... 130
Detail Entities ........................................................................ 131
OLE Objects .......................................................................... 134
Detail Notes ........................................................................... 135
detail Groups ....................................................................... 138
detail Symbols ....................................................................... 140
detail Attachments ............................................................... 150

Solid ..................................................................................... 155
Getting a Solid Object .......................................................... 156
Solid Information ................................................................. 156
Solid Operations ................................................................... 156
Solid Units ............................................................................ 160
Mass Properties .................................................................... 166
Annotations ........................................................................... 166
cross Sections ....................................................................... 167
Materials ................................................................................ 168

Curve and Surface Collection .................................................. 175
Introduction to Curve and Surface Collection ......................... 176
Interactive Collection ........................................................... 177
Programmatic Access to Collections ....................................... 179

Windows and Views ............................................................... 187
Windows ............................................................................... 188
Embedded Browser ............................................................. 190
Views .................................................................................... 190
Coordinate Systems and Transformations ................................. 191

ModellItem ............................................................................ 197
Solid Geometry Traversal ....................................................... 198
Getting ModellItem Objects .................................................. 198
ModellItem Information ........................................................ 199
Duplicating ModellItems ....................................................... 199
Layer Objects .......................................................................... 200

Feature Element Tree ........................................................... 203
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Feature Creation</td>
<td>204</td>
</tr>
<tr>
<td>Feature Element Values</td>
<td>206</td>
</tr>
<tr>
<td>Feature Element Special Values</td>
<td>206</td>
</tr>
<tr>
<td>Feature Element Paths</td>
<td>207</td>
</tr>
<tr>
<td>Feature Element Tree</td>
<td>207</td>
</tr>
<tr>
<td>Creating FET Using WCreateFeature</td>
<td>208</td>
</tr>
<tr>
<td>Examples of Feature Creation</td>
<td>209</td>
</tr>
<tr>
<td>Feature Elements</td>
<td>209</td>
</tr>
<tr>
<td>Creating Patterns</td>
<td>210</td>
</tr>
<tr>
<td>Redefining Features</td>
<td>210</td>
</tr>
<tr>
<td>Element Diagnostics</td>
<td>211</td>
</tr>
<tr>
<td>Features</td>
<td>213</td>
</tr>
<tr>
<td>Access to Features</td>
<td>214</td>
</tr>
<tr>
<td>Feature Information</td>
<td>215</td>
</tr>
<tr>
<td>Feature Operations</td>
<td>216</td>
</tr>
<tr>
<td>Feature Groups and Patterns</td>
<td>219</td>
</tr>
<tr>
<td>User Defined Features</td>
<td>220</td>
</tr>
<tr>
<td>Creating Features from UDFs</td>
<td>221</td>
</tr>
<tr>
<td>Datum Features</td>
<td>229</td>
</tr>
<tr>
<td>Datum Plane Features</td>
<td>230</td>
</tr>
<tr>
<td>Datum Axis Features</td>
<td>232</td>
</tr>
<tr>
<td>General Datum Point Features</td>
<td>234</td>
</tr>
<tr>
<td>Datum Coordinate System Features</td>
<td>235</td>
</tr>
<tr>
<td>Element Trees: Sections</td>
<td>239</td>
</tr>
<tr>
<td>Overview</td>
<td>240</td>
</tr>
<tr>
<td>Creating Section Models</td>
<td>240</td>
</tr>
<tr>
<td>Element Trees: Sketched Features</td>
<td>245</td>
</tr>
<tr>
<td>Overview</td>
<td>246</td>
</tr>
<tr>
<td>Creating Features Containing Sections</td>
<td>246</td>
</tr>
<tr>
<td>Creating Features with 2D Sections</td>
<td>247</td>
</tr>
<tr>
<td>Creating Features with 3D Sections</td>
<td>247</td>
</tr>
<tr>
<td>Example 2: Manipulating a 3D Section</td>
<td>248</td>
</tr>
<tr>
<td>Holes</td>
<td>249</td>
</tr>
<tr>
<td>Accessing Threaded Hole Properties</td>
<td>250</td>
</tr>
<tr>
<td>Geometry Evaluation</td>
<td>251</td>
</tr>
<tr>
<td>Geometry Traversal</td>
<td>252</td>
</tr>
<tr>
<td>Curves and Edges</td>
<td>253</td>
</tr>
<tr>
<td>Contours</td>
<td>256</td>
</tr>
<tr>
<td>Surfaces</td>
<td>256</td>
</tr>
<tr>
<td>Axes, Coordinate Systems, and Points</td>
<td>260</td>
</tr>
<tr>
<td>Interference</td>
<td>261</td>
</tr>
<tr>
<td>Dimensions and Parameters</td>
<td>263</td>
</tr>
<tr>
<td>Overview</td>
<td>264</td>
</tr>
<tr>
<td>The ParamValue Object</td>
<td>264</td>
</tr>
<tr>
<td>Parameter Objects</td>
<td>265</td>
</tr>
<tr>
<td>Dimension Objects</td>
<td>272</td>
</tr>
<tr>
<td>Relations</td>
<td>275</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Accessing Relations</td>
<td>276</td>
</tr>
<tr>
<td>Accessing Post Regeneration Relations</td>
<td>276</td>
</tr>
<tr>
<td>Adding a Customized Function to the Relations Dialog Box in Creo</td>
<td></td>
</tr>
<tr>
<td>- Parametric</td>
<td>277</td>
</tr>
<tr>
<td>Assemblies and Components</td>
<td>281</td>
</tr>
<tr>
<td>- Structure of Assemblies and Assembly Objects</td>
<td>282</td>
</tr>
<tr>
<td>- Assembling Components</td>
<td>288</td>
</tr>
<tr>
<td>- Redefining and Rerouting Assembly Components</td>
<td>293</td>
</tr>
<tr>
<td>- Exploded Assemblies</td>
<td>293</td>
</tr>
<tr>
<td>- Skeleton Models</td>
<td>294</td>
</tr>
<tr>
<td>- Flexible Components</td>
<td>294</td>
</tr>
<tr>
<td>- Flexible Component Variant Items</td>
<td>295</td>
</tr>
<tr>
<td>Family Tables</td>
<td>297</td>
</tr>
<tr>
<td>- Working with Family Tables</td>
<td>298</td>
</tr>
<tr>
<td>- Creating Family Table Instances</td>
<td>300</td>
</tr>
<tr>
<td>- Creating Family Table Columns</td>
<td>300</td>
</tr>
<tr>
<td>Action Listeners</td>
<td>303</td>
</tr>
<tr>
<td>- Object TOOLKIT C++ Action Listeners</td>
<td>304</td>
</tr>
<tr>
<td>- Creating an ActionListener Implementation</td>
<td>304</td>
</tr>
<tr>
<td>- Action Sources</td>
<td>304</td>
</tr>
<tr>
<td>- Types of Action Listeners</td>
<td>305</td>
</tr>
<tr>
<td>- Cancelling an ActionListener Operation</td>
<td>310</td>
</tr>
<tr>
<td>Interface</td>
<td>311</td>
</tr>
<tr>
<td>- Exporting Files and 2D Models</td>
<td>312</td>
</tr>
<tr>
<td>- Exporting to PDF and U3D</td>
<td>319</td>
</tr>
<tr>
<td>- Exporting 3D Geometry</td>
<td>326</td>
</tr>
<tr>
<td>- Shrinkwrap Export</td>
<td>329</td>
</tr>
<tr>
<td>- Importing Files</td>
<td>335</td>
</tr>
<tr>
<td>- Importing 3D Geometry</td>
<td>338</td>
</tr>
<tr>
<td>- Associative Topology Bus Enabled Models and Features</td>
<td>340</td>
</tr>
<tr>
<td>- Printing Files</td>
<td>343</td>
</tr>
<tr>
<td>- Automatic Printing of 3D Models</td>
<td>350</td>
</tr>
<tr>
<td>- Solid Operations</td>
<td>354</td>
</tr>
<tr>
<td>- Window Operations</td>
<td>356</td>
</tr>
<tr>
<td>Simplified Representations</td>
<td>357</td>
</tr>
<tr>
<td>- Overview</td>
<td>358</td>
</tr>
<tr>
<td>- Retrieving Simplified Representations</td>
<td>359</td>
</tr>
<tr>
<td>- Creating and Deleting Simplified Representations</td>
<td>359</td>
</tr>
<tr>
<td>- Extracting Information About Simplified Representations</td>
<td>360</td>
</tr>
<tr>
<td>- Modifying Simplified Representations</td>
<td>360</td>
</tr>
<tr>
<td>- Simplified Representation Utilities</td>
<td>362</td>
</tr>
<tr>
<td>Asynchronous Mode</td>
<td>365</td>
</tr>
<tr>
<td>- Overview</td>
<td>366</td>
</tr>
<tr>
<td>- Simple Asynchronous Mode</td>
<td>367</td>
</tr>
<tr>
<td>- Starting and Stopping Creo Parametric</td>
<td>367</td>
</tr>
<tr>
<td>- Connecting to a Creo Parametric Process</td>
<td>368</td>
</tr>
<tr>
<td>- Full Asynchronous Mode</td>
<td>370</td>
</tr>
<tr>
<td>- Troubleshooting Asynchronous Object TOOLKIT</td>
<td>372</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Task Based Application Libraries</td>
<td>373</td>
</tr>
<tr>
<td>Managing Application Arguments</td>
<td>374</td>
</tr>
<tr>
<td>Launching a Creo Parametric TOOLKIT DLL</td>
<td>375</td>
</tr>
<tr>
<td>Launching Tasks from Object TOOLKIT C++ Task Libraries</td>
<td>377</td>
</tr>
<tr>
<td>Graphics</td>
<td>379</td>
</tr>
<tr>
<td>Overview</td>
<td>380</td>
</tr>
<tr>
<td>Getting Mouse Input</td>
<td>380</td>
</tr>
<tr>
<td>Displaying Graphics</td>
<td>381</td>
</tr>
<tr>
<td>Display Lists and Graphics</td>
<td>383</td>
</tr>
<tr>
<td>External Data</td>
<td>387</td>
</tr>
<tr>
<td>External Data</td>
<td>388</td>
</tr>
<tr>
<td>Exceptions</td>
<td>391</td>
</tr>
<tr>
<td>Windchill Connectivity APIs</td>
<td>393</td>
</tr>
<tr>
<td>Introduction</td>
<td>394</td>
</tr>
<tr>
<td>Accessing a Windchill Server from a Creo Parametric Session</td>
<td>394</td>
</tr>
<tr>
<td>Accessing Workspaces</td>
<td>397</td>
</tr>
<tr>
<td>Workflow to Register a Server</td>
<td>399</td>
</tr>
<tr>
<td>Aliased URL</td>
<td>400</td>
</tr>
<tr>
<td>Server Operations</td>
<td>401</td>
</tr>
<tr>
<td>Utility APIs</td>
<td>411</td>
</tr>
<tr>
<td>Appendix A. Summary of Technical Changes</td>
<td>413</td>
</tr>
<tr>
<td>Appendix B. Object TOOLKIT C++ Library Types</td>
<td>431</td>
</tr>
<tr>
<td>Appendix C. Digital Rights Management</td>
<td>433</td>
</tr>
<tr>
<td>Appendix D. Geometry Traversal</td>
<td>439</td>
</tr>
<tr>
<td>Appendix E. Geometry Representations</td>
<td>443</td>
</tr>
<tr>
<td>Index</td>
<td>457</td>
</tr>
</tbody>
</table>
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 Object TOOLKIT C++, an object-based C++ toolkit API for Creo Parametric. Object TOOLKIT C++ makes possible the development of C++ programs that access the internal components of a Creo Parametric session, to customize Creo Parametric models.

Note

Object TOOLKIT C++ is supported only 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 Object TOOLKIT C++ includes the following:

- *Object TOOLKIT C++ User’s Guide*
- An online browser that describes the syntax of the 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.

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

Setting Up Object TOOLKIT C++........................................................................................................... 14
Installing Object TOOLKIT C++ .................................................................................................................. 14
Unlocking the Object TOOLKIT C++ Application ...................................................................................... 14
Object TOOLKIT C++ Registry File .............................................................................................................. 14
How Object TOOLKIT C++ Works .............................................................................................................. 14
Categories of Object TOOLKIT C++ Classes .............................................................................................. 16
Using Object TOOLKIT C++ with Creo Parametric TOOLKIT ................................................................. 27
Object TOOLKIT C++ Support for Creo Applications ............................................................................... 28
Creating Applications .................................................................................................................................. 28
Version Compatibility: Creo Parametric and Object TOOLKIT C++ ......................................................... 33
Retrieving Creo Parametric Datecode .......................................................................................................... 34
Threading in Object TOOLKIT C++ Applications ....................................................................................... 34

This chapter provides an overview of Object TOOLKIT C++.
Setting Up Object TOOLKIT C++

Object TOOLKIT is a Creo Parametric 1.0 object-based C++ toolkit API. Object TOOLKIT C++ can be used with Creo Parametric TOOLKIT inside the same C++ based application.

Installing Object TOOLKIT C++

Object TOOLKIT C++ is a part of the Creo Parametric 1.0 installation, and is installed automatically, if selected.

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

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

Unlocking the Object TOOLKIT C++ Application

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

*Note*

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

Object TOOLKIT C++ Registry File

Object TOOLKIT C++ based applications use the same format of `creotk.dat` file as Creo Parametric TOOLKIT based applications.

How Object TOOLKIT C++ Works

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

Object TOOLKIT C++ also supports a second method of integration: the “multiprocess,” or spawned mode. In this mode, the Object TOOLKIT C++ application code is compiled and linked to form a separate executable. This executable is designed to be spawned by Creo Parametric and runs as a child process of the Creo Parametric session. In DLL mode, the exchanges between the Object TOOLKIT C++ application and Creo Parametric 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 Object TOOLKIT C++ application makes frequent calls to 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 Object TOOLKIT C++ application with a source-code debugger without also loading the whole Creo Parametric executable into the debugger. You can use an 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 Object TOOLKIT C++ application within a single session of Creo Parametric, 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 Creo Parametric or 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 an Object TOOLKIT C++ application that provides this ability, called “asynchronous mode”. For more information on asynchronous mode, see the chapter Asynchronous Mode on page 365. The DLL and multiprocess modes are given the general name “synchronous mode”. An asynchronous 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 Object TOOLKIT C++ calls running in either synchronous (DLL or multiprocess) mode or asynchronous mode always clear the Undo/Redo stack in the Creo Parametric session. The Creo Parametric user interface reflects this by making the Undo and Redo menu options unavailable.

### Categories of Object TOOLKIT C++ Classes

Object TOOLKIT C++ is made up of a number of classes. 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 Parametric-Related Interfaces**—Contains unique methods and attributes that are directly related to the functions in Creo Parametric. See the section Creo Parametric-Related Interfaces on page 17 for additional information.
- **Compact Data Classes**—Classes containing data needed as arguments to some Object TOOLKIT C++ methods. See the section Compact Data Classes on page 19 for additional information.
- **Union Classes**—Classes with a potential for multiple types of values. See the section Union Classes on page 20 for additional information.
- **Sequence Classes**—Expandable arrays of objects or primitive data types. See the section Sequence Classes on page 20 for more information.
- **Array Classes**—Arrays that are limited to a certain size. See the section Array Classes on page 22 for more information.
- **Enumeration Classes**—Classes that define the enumerated types. See the section Enumeration Classes on page 23 for more information.
• **ActionListener Classes**—Classes that enable you to specify callbacks that will run only if certain events in Creo Parametric take place. See the section [Action Listener Classes on page 24](#) for more information.

• **Utility Classes**—Classes that contain the static methods used to initialize certain Object TOOLKIT C++ objects. See the section [Utilities on page 26](#) for more information.

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

**Creo Parametric-Related Interfaces**

The Creo Parametric-related interfaces contain methods that directly manipulate objects in Creo Parametric. 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 Parametric-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:

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

The following calls are illegal:

```cpp
pfcWindow window; // pfcWindow is a pure virtual class, C++ constructor cannot be used
window.Activate();
window.Repaint(); // There is no invoking object.
```

Inheritance

All Creo Parametric-related objects 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:

```cpp
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:

```cpp
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 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 Object TOOLKIT C++ methods. They do not represent the actual objects in Creo Parametric.

Initialization

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

```cpp
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**:
  ```cpp
  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:

```cpp
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 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 Object TOOLKIT C++ union:

class pfcParamValue : public xobject
{
xsDeclare (pfcParamValue)
public:
    pfcParamValueType Getdiscr () ;
    xstring GetStringValue () ;
    xint GetIntValue () ;
    xbool GetBoolValue () ;
    xreal GetDoubleValue () ;
    xint GetNoteId () ;
    void SetStringValue (xstring value) ;
    void SetIntValue (xint value) ;
    void SetBoolValue (xbool value) ;
    void SetDoubleValue (xreal value) ;
    void SetNoteId (xint value) ;
private:
    pfcParamvalueValue mProParamvalue ;
public:
    pfcParamValue (const pfcParamvalueValue &inProParamvalue) ;
    pfcParamValue () ;

    const pfcParamvalueValue &GetProParamvalue () ;
};

Sequence Classes

Sequences are expandable arrays of primitive data types or objects in 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 Object TOOLKIT C++ classes. Therefore, you cannot cast sequence objects to any other type of 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:

```cpp
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++ Object TOOLKIT C++.

```cpp
// 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 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 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 Parametric TOOLKIT
Since Object TOOLKIT C++ and Creo Parametric TOOLKIT calls can be combined in the same code, there is no need in introducing special enumerations and constants for 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 Object TOOLKIT C++ to assign programmed reactions to events that occur within Creo Parametric. Object TOOLKIT C++ defines a set of action listener interfaces that can be implemented enabling Creo Parametric to call your Object TOOLKIT C++ application when specific events occur. These interfaces are designed to respond to events from action sources in Creo Parametric. Examples of action sources include the session, user-interface commands, models, solids, parameters, and features.

**Initialization**

Object TOOLKIT C++ provides a few Action Listener classes, whose actions map most of notification types in Creo Parametric TOOLKIT ProNotifyType. 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 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() method of the action source. When done with using the action listener, destroy it with the C++ keyword delete.

**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 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
{
  xainclude (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, xbool WasSuccessful, pfcFeature_ptr Feat) = 0;
}
virtual void OnAfterFeatureDelete (pfcSolid_ptr Sld, xint FeatId) = 0;
};

Utilities

Each package in 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 Object TOOLKIT C++ objects.

Inheritance

Utilities do not inherit from any other 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
}

Using Object TOOLKIT C++ with Creo Parametric TOOLKIT

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

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) );
Object TOOLKIT C++ Support for Creo Applications

Object TOOLKIT C++ applications in synchronous and asynchronous modes are supported only with the Creo Parametric application. They are not supported with the other Creo applications, such as, Creo Direct, Creo Layout, Creo Simulate and so on.

In the asynchronous mode, the methods `pfcAsyncConnection::Connect` and `pfcAsyncConnection::Start` return an error when the Object TOOLKIT C++ application attempts to connect to a Creo application other than Creo Parametric.

For Object TOOLKIT C++ in synchronous mode, the non-Creo Parametric applications ignore the registry files without any warnings. The Auxiliary Applications dialog box is also not available within the non-Creo Parametric applications.

Creating Applications

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

- Application Hierarchy on page 28
- Exception Handling on page 29

Application Hierarchy

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

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

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

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

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.

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

PFC/WFC Exceptions

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

<table>
<thead>
<tr>
<th>Exception</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcXBadExternalData</td>
<td>Indicates an attempt to read contents of an external data object which has been terminated.</td>
</tr>
<tr>
<td>pfcXBadGetArgValue</td>
<td>Indicates an attempt to read the wrong type of data from the ArgValue union.</td>
</tr>
<tr>
<td>pfcXBadGetExternalData</td>
<td>Indicates an attempt to read the wrong type of data from the ExternalData union.</td>
</tr>
<tr>
<td>pfcXBadGetParamValue</td>
<td>Indicates an attempt to read the wrong type of data from the ParamValue union.</td>
</tr>
<tr>
<td>pfcXBadOutlineExcludeType</td>
<td>Indicates that an invalid type of item was passed to the outline calculation method.</td>
</tr>
<tr>
<td>pfcXCANCELPROEAction</td>
<td>This exception type will not be thrown by Object TOOLKIT C++ methods, but you may instantiate and throw this from certain ActionListener methods to cancel the corresponding action in Creo Parametric.</td>
</tr>
<tr>
<td>pfcXCannotAccess</td>
<td>Indicates that the contents of an Object TOOLKIT C++ object cannot be accessed in this situation.</td>
</tr>
<tr>
<td>pfcXEmptyString</td>
<td>Indicates an empty string was passed to a method that does not accept this type of input.</td>
</tr>
<tr>
<td>pfcXInvalidEnumValue</td>
<td>Indicates an invalid value for a specified enumeration class.</td>
</tr>
<tr>
<td>Exception</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pfcXInvalidFileName</td>
<td>Indicates that a file name passed to a method was incorrectly structured.</td>
</tr>
<tr>
<td>pfcXInvalidFileType</td>
<td>Indicates a model descriptor contained an invalid file type for a requested operation.</td>
</tr>
<tr>
<td>pfcXInvalidModelItem</td>
<td>Indicates that the item requested to be used is no longer usable (for example, it may have been deleted).</td>
</tr>
<tr>
<td>pfcXInvalidSelection</td>
<td>Indicates that the selection passed is invalid or is missing a needed piece of information. For example, its component path, drawing view, or parameters.</td>
</tr>
<tr>
<td>pfcXModelNotInSession</td>
<td>Indicates that the model is no longer in session; it may have been erased or deleted.</td>
</tr>
<tr>
<td>pfcXNegativeNumber</td>
<td>Numeric argument was negative.</td>
</tr>
<tr>
<td>pfcXNumberTooLarge</td>
<td>Numeric argument was too large.</td>
</tr>
<tr>
<td>pfcXProEWasNotConnected</td>
<td>The Creo Parametric session is not available so the operation failed.</td>
</tr>
<tr>
<td>pfcXSequenceTooLong</td>
<td>Indicates that the sequence argument was too long.</td>
</tr>
<tr>
<td>pfcXStringTooLong</td>
<td>Indicates that the string argument was too long.</td>
</tr>
<tr>
<td>pfcXUnimplemented</td>
<td>Indicates unimplemented method.</td>
</tr>
<tr>
<td>pfcXUnknownModelExtension</td>
<td>Indicates that a file extension does not match a known Creo Parametric model type.</td>
</tr>
</tbody>
</table>

### Creo Parametric TOOLKIT Errors

The pfcXToolkitError exception provides access to error codes from Creo Parametric TOOLKIT functions that 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 Object TOOLKIT C++ is built on top of Creo Parametric TOOLKIT. The following table lists the integer values that can be returned by the pfcXToolkitError::GetErrorCode() method and shows the corresponding Creo Parametric 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.

<table>
<thead>
<tr>
<th>pfcXToolkitError Child Class</th>
<th>Creo Parametric TOOLKIT Error</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcXToolkitGeneralError</td>
<td>PRO_TK_GENERAL_ERROR</td>
<td>-1</td>
</tr>
<tr>
<td>pfcXToolkitBadInputs</td>
<td>PRO_TK_BAD_INPUTS</td>
<td>-2</td>
</tr>
<tr>
<td>pfcXToolkitUserAbort</td>
<td>PRO_TK_USER_ABORT</td>
<td>-3</td>
</tr>
<tr>
<td>pfcXToolkitNotFound</td>
<td>PRO_TK_E_NOT_FOUND</td>
<td>-4</td>
</tr>
<tr>
<td>pfcXToolkitFound</td>
<td>PRO_TK_E_FOUND</td>
<td>-5</td>
</tr>
<tr>
<td>pfcXToolkitLineTooLong</td>
<td>PRO_TK_LINE_TOO_LONG</td>
<td>-6</td>
</tr>
<tr>
<td>pfcXToolkitContinue</td>
<td>PRO_TK_CONTINUE</td>
<td>-7</td>
</tr>
<tr>
<td>pfcXToolkitBadContext</td>
<td>PRO_TK_BAD_CONTEXT</td>
<td>-8</td>
</tr>
<tr>
<td>pfcXToolkitNotImplemented</td>
<td>PRO_TK_NOT_IMPLEMENTED</td>
<td>-9</td>
</tr>
<tr>
<td>pfcXToolkitOutOfMemory</td>
<td>PRO_TK_OUT_OF_MEMORY</td>
<td>-10</td>
</tr>
<tr>
<td>pfcXToolkitCommError</td>
<td>PRO_TK_COMM_ERROR</td>
<td>-11</td>
</tr>
<tr>
<td>pfcXToolkitNoChange</td>
<td>PRO_TK_NO_CHANGE</td>
<td>-12</td>
</tr>
<tr>
<td>pfcXToolkitError Child Class</td>
<td>Creo Parametric TOOLKIT Error</td>
<td>#</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>pfcXToolkitSuppressedParents</td>
<td>PRO_TK_SUPP_PARENTS</td>
<td>-13</td>
</tr>
<tr>
<td>pfcXToolkitPickAbove</td>
<td>PRO_TK_PICK_ABOVE</td>
<td>-14</td>
</tr>
<tr>
<td>pfcXToolkitInvalidDir</td>
<td>PRO_TK_INVALID_DIR</td>
<td>-15</td>
</tr>
<tr>
<td>pfcXToolkitInvalidFile</td>
<td>PRO_TK_INVALID_FILE</td>
<td>-16</td>
</tr>
<tr>
<td>pfcXToolkitCantWrite</td>
<td>PRO_TK_CANT_WRITE</td>
<td>-17</td>
</tr>
<tr>
<td>pfcXToolkitInvalidType</td>
<td>PRO_TK_INVALID_TYPE</td>
<td>-18</td>
</tr>
<tr>
<td>pfcXToolkitInvalidPtr</td>
<td>PRO_TK_INVALID_PTR</td>
<td>-19</td>
</tr>
<tr>
<td>pfcXToolkitUnavailableSection</td>
<td>PRO_TK_UNAV_SEC</td>
<td>-20</td>
</tr>
<tr>
<td>pfcXToolkitInvalidMatrix</td>
<td>PRO_TK_INVALID_MATRIX</td>
<td>-21</td>
</tr>
<tr>
<td>pfcXToolkitInvalidName</td>
<td>PRO_TK_INVALID_NAME</td>
<td>-22</td>
</tr>
<tr>
<td>pfcXToolkitNotExist</td>
<td>PRO_TK_NOT_EXIST</td>
<td>-23</td>
</tr>
<tr>
<td>pfcXToolkitCantOpen</td>
<td>PRO_TK_CANT_OPEN</td>
<td>-24</td>
</tr>
<tr>
<td>pfcXToolkitAbort</td>
<td>PRO_TK_ABORT</td>
<td>-25</td>
</tr>
<tr>
<td>pfcXToolkitNotValid</td>
<td>PRO_TK_NOT_VALID</td>
<td>-26</td>
</tr>
<tr>
<td>pfcXToolkitInvalidItem</td>
<td>PRO_TK_INVALID_ITEM</td>
<td>-27</td>
</tr>
<tr>
<td>pfcXToolkitMsgNotFound</td>
<td>PRO_TK_MSG_NOT_FOUND</td>
<td>-28</td>
</tr>
<tr>
<td>pfcXToolkitMsgNoTrans</td>
<td>PRO_TK_MSG_NO_TRANS</td>
<td>-29</td>
</tr>
<tr>
<td>pfcXToolkitMsgFmtError</td>
<td>PRO_TK_MSG_FMT_ERROR</td>
<td>-30</td>
</tr>
<tr>
<td>pfcXToolkitMsgUserQuit</td>
<td>PRO_TK_MSG_USER_QUIT</td>
<td>-31</td>
</tr>
<tr>
<td>pfcXToolkitMsgTooLong</td>
<td>PRO_TK_MSG_TOO_LONG</td>
<td>-32</td>
</tr>
<tr>
<td>pfcXToolkitCantAccess</td>
<td>PRO_TK_CANT_ACCESS</td>
<td>-33</td>
</tr>
<tr>
<td>pfcXToolkitObsoleteFunc</td>
<td>PRO_TK_OBSOLETE_FUNC</td>
<td>-34</td>
</tr>
<tr>
<td>pfcXToolkitNoCoordSystem</td>
<td>PRO_TK_NO_COORD_SYSTEM</td>
<td>-35</td>
</tr>
<tr>
<td>pfcXToolkitAmbiguous</td>
<td>PRO_TK_E_AMBIGUOUS</td>
<td>-36</td>
</tr>
<tr>
<td>pfcXToolkitDeadLock</td>
<td>PRO_TK_E_DEADLOCK</td>
<td>-37</td>
</tr>
<tr>
<td>pfcXToolkitBusy</td>
<td>PRO_TK_E_BUSY</td>
<td>-38</td>
</tr>
<tr>
<td>pfcXToolkitInUse</td>
<td>PRO_TK_E_IN_USE</td>
<td>-39</td>
</tr>
<tr>
<td>pfcXToolkitNoLicense</td>
<td>PRO_TK_NO_LICENSE</td>
<td>-40</td>
</tr>
<tr>
<td>pfcXToolkitBsplUnsuitableDegree</td>
<td>PRO_TK_BSPL_UNSUITABLE_DEGREE</td>
<td>-41</td>
</tr>
<tr>
<td>pfcXToolkitBsplNonStdEndKnots</td>
<td>PRO_TK_BSPL NON_STD_END_KNOTS</td>
<td>-42</td>
</tr>
<tr>
<td>pfcXToolkitBsplMultiInnerKnots</td>
<td>PRO_TK_BSPL MULTI INNER_KNOTS</td>
<td>-43</td>
</tr>
<tr>
<td>pfcXToolkitBadSrfCrv</td>
<td>PRO_TK_BAD_SRF_CRV</td>
<td>-44</td>
</tr>
<tr>
<td>pfcXToolkitEmpty</td>
<td>PRO_TK_EMPTY</td>
<td>-45</td>
</tr>
<tr>
<td>pfcXToolkitBadDimAttach</td>
<td>PRO_TK_BAD_DIM.Attach</td>
<td>-46</td>
</tr>
<tr>
<td>pfcXToolkitNotDisplayed</td>
<td>PRO_TK_NOT_DISPLAYED</td>
<td>-47</td>
</tr>
<tr>
<td>pfcXToolkitCantModify</td>
<td>PRO_TK_CANT MODIFY</td>
<td>-48</td>
</tr>
<tr>
<td>pfcXToolkitCheckoutConflict</td>
<td>PRO_TK_CHECKOUT_CONFLICT</td>
<td>-49</td>
</tr>
<tr>
<td>pfcXToolkitCreateViewBadSheet</td>
<td>PRO_TK_CRE_VIEW BAD SHEET</td>
<td>-50</td>
</tr>
<tr>
<td>pfcXToolkitCreateViewBadModel</td>
<td>PRO_TK_CRE_VIEW BAD MODEL</td>
<td>-51</td>
</tr>
<tr>
<td>pfcXToolkitCreateViewBadParent</td>
<td>PRO_TK_CRE_VIEW BAD_PARENT</td>
<td>-52</td>
</tr>
<tr>
<td>pfcXToolkitCreateViewBadType</td>
<td>PRO_TK_CRE VIEW BAD TYPE</td>
<td>-53</td>
</tr>
<tr>
<td>pfcXToolkitCreateViewBadExplode</td>
<td>PRO_TK_CRE_VIEW BAD_EXPLODE</td>
<td>-54</td>
</tr>
<tr>
<td>pfcXToolkitUnattachedFeats</td>
<td>PRO_TK_UNATTACHED_FEATS</td>
<td>-55</td>
</tr>
<tr>
<td>pfcXToolkitRegenerateAgain</td>
<td>PRO_TK_REGEN AGAIN</td>
<td>-56</td>
</tr>
<tr>
<td>pfcXToolkitDrawingCreateErrors</td>
<td>PRO_TK_DWGCREATE_ERRORS</td>
<td>-57</td>
</tr>
<tr>
<td>pfcXToolkitUnsupported</td>
<td>PRO_TK_UNSUPPORTED</td>
<td>-58</td>
</tr>
<tr>
<td>pfcXToolkitNoPermission</td>
<td>PRO_TK_NO_PERMISSION</td>
<td>-59</td>
</tr>
</tbody>
</table>

Overview of Object TOOLKIT C++
<table>
<thead>
<tr>
<th>pfcXToolkitError Child Class</th>
<th>Creo Parametric TOOLKIT Error</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcXToolkitAuthenticationFailure</td>
<td>PRO_TK_AUTHENTICATION_FAILURE</td>
<td>-60</td>
</tr>
<tr>
<td>pfcXToolkitAppNoLicense</td>
<td>PRO_TK_APP_NO_LICENSE</td>
<td>-92</td>
</tr>
<tr>
<td>pfcXToolkitAppExcessCallbacks</td>
<td>PRO_TK_APP_XS_CALLBACKS</td>
<td>-93</td>
</tr>
<tr>
<td>pfcXToolkitAppStartupFailed</td>
<td>PRO_TK_APP_STARTUP_FAIL</td>
<td>-94</td>
</tr>
<tr>
<td>pfcXToolkitAppInitializationFailed</td>
<td>PRO_TK_APP_INIT_FAIL</td>
<td>-95</td>
</tr>
<tr>
<td>pfcXToolkitAppVersionMismatch</td>
<td>PRO_TK_APP_VERSION_MISMATCH</td>
<td>-96</td>
</tr>
<tr>
<td>pfcXToolkitAppCommunicationFailure</td>
<td>PRO_TK_APP_COMM_FAILURE</td>
<td>-97</td>
</tr>
<tr>
<td>pfcXToolkitAppNewVersion</td>
<td>PRO_TK_APP_NEW_VERSION</td>
<td>-98</td>
</tr>
</tbody>
</table>

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 Object TOOLKIT C++ Exception Handling

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

```cpp
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:

```cpp
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 Object TOOLKIT C++

In many situations it will be inconvenient or impossible to ensure that the users of your Object TOOLKIT C++ application use the same build of Creo Parametric used to compile and link the Object TOOLKIT C++ application. This section summarizes the rules for mixing Object TOOLKIT C++ and Creo Parametric versions. The Object TOOLKIT C++ version is the Creo Parametric CD version from which the user installed the 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 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 2.0 is 31.

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

• Creo Parametric release newer than an Object TOOLKIT C++ release:

  This works in many, but not all, cases. The communication method used to link 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 an Object TOOLKIT C++ application in order that it continues to work correctly. Whether you need to convert Object TOOLKIT C++ applications depends on what functionality it uses and what functionality changed in Creo Parametric and Object TOOLKIT C++. PTC makes every effort to keep
these effects to a minimum. The Release Notes for Object TOOLKIT C++ detail any conversion work that could be necessary for that release.

- Creo Parametric build newer than Object TOOLKIT C++ build
  This is always supported.

Retrieving Creo Parametric Datecode

Method Introduced:

- wfcWSession::GetDisplayDateCode

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

Threading in Object TOOLKIT C++ Applications

Calling Object TOOLKIT C++ applications from within multiple threads of any application in any mode is not supported. Extra threads created by the application are to be used only for completion of tasks that do not directly call the Object TOOLKIT C++ methods.

Method Introduced:

- wfcWSession::EnableMultiThreadMode

Call the method wfcWSession::EnableMultiThreadMode from within the initialization function user_initialize(), if your application creates additional threads for processing. This method notifies Creo Parametric to execute in the multithread enabled mode. Running in this mode eliminates the possibility of a memory corruption due to interaction between Creo Parametric’s thread libraries and the threads created by your application. This method does not work for multiprocess and asynchronous mode applications.

**Note**

Running Creo Parametric in the multithread enabled mode may slow down performance. Therefore, wfcWSession::EnableMultiThreadMode should be used only for applications that actually create multiple threads.
This chapter describes how to use the online browser provided with Object TOOLKIT C++.
Online Documentation for Object TOOLKIT C++ APIWizard

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

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

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

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

Note

The Object TOOLKIT C++ User’s Guide is also available in PDF format. This file is located at:
<creo_loadpoint>\Common Files\<datecode>\otk\otkug.pdf

Installing the APIWizard

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

Starting the APIWizard

Start the Object TOOLKIT C++ APIWizard by pointing your browser to:
<creo_otk_loadpoint>\otk_cppdoc\index.html

Your web browser will display the Object TOOLKIT C++ APIWizard data in a new window.
Web Browser Environments

The APIWizard supports Netscape Firefox and Internet Explorer.

For APIWizard use with Internet Explorer, the recommended browser environment requires installation of the Java plug-in.

For Netscape Firefox, the recommended browser environment requires installation of the Java Swing foundation class. If this class is not loaded on your computer, the APIWizard can load it for you. This takes several minutes, and is not persistant between sessions.

Automatic Index Tree Updating

With your browser environment configured correctly, following a link in an APIWizard HTML file causes the tree in the Selection frame to update and scroll the tree reference that corresponds to the newly displayed page. This is automatic tree scrolling.

APIWizard Interface

The APIWizard interface consists of two frames. The next sections describe how to display and use these frames in your Web browser.

Library/Class/Exception/Enumeration/Topic Selection Frame

This frame, located on the left of the screen, controls what is presented in the Display frame. Specify what data you want to view by choosing either Object TOOLKIT C++ Libraries, Object TOOLKIT C++ Classes, Object TOOLKIT C++ Exceptions, Object TOOLKIT C++ Enumerations, or the Object TOOLKIT C++ User's Guide.

In Object TOOLKIT C++ Libraries mode, this frame displays an alphabetical list of the Object TOOLKIT C++ libraries. A library is a logical subdivision of functionality within Object TOOLKIT C++; for example, the pfcFamily library contains classes related to family table operations. This frame can also display Object TOOLKIT C++ classes, enumerations and methods as subnodes of the library.

In Object TOOLKIT C++ Classes mode, this frame displays an alphabetical list of the Object TOOLKIT C++ Classes. It can also display Object TOOLKIT C++ methods as subnodes of the classes.
In **Object TOOLKIT C++ Exceptions** mode, this frame displays an alphabetical list of named exceptions in the Object TOOLKIT C++ library. It can also display the methods for the exceptions as the subnodes.

In **Object TOOLKIT C++ Enumerations** mode, this frame displays an alphabetical list of the Object TOOLKIT C++ enumerated type classes.

In **Object TOOLKIT C++ User’s Guide** mode, this frame displays the *Object TOOLKIT C++ User’s Guide* table of contents in a tree structure. All chapters are displayed as subnodes of the main *Object TOOLKIT C++ User’s Guide* node.

The **Library/Class/Enumerations/Topic** frame includes a **Find** button for data searches of the *Object TOOLKIT C++ User’s Guide* or of API specifications taken from header files. See the section **APIWizard Search Feature (Find)** on page 43 for more information on the **Find** feature.

**Display Frame**

This frame, located on the right of the screen, displays:

- Object TOOLKIT C++ library definitions
- Object TOOLKIT C++ classes definitions and their hierarchical relationships
- Object TOOLKIT C++ method descriptions
- User's Guide content
- Code examples for Object TOOLKIT C++ methods

The following figure displays the APIWizard interface layout.
Overview of Object TOOLKIT C++

This section provides an overview of Object TOOLKIT C++.

Setting Up Object TOOLKIT C++

Object TOOLKIT is a Creo Parametric 1.0 object-based C++ toolkit API. Object TOOLKIT C++ can be used with Creo Parametric TOOLKIT inside the same C++ based application.

Installing Object TOOLKIT C++

Object TOOLKIT C++ is a part of the Creo Parametric 1.0 installation, and is installed automatically, if selected.

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

- **otk/otk_cpp/includes**—Contains all the header files specific to Object TOOLKIT C++.
- **otk_cpp/platform**—Contains platform-specific libraries of Object TOOLKIT C++ which must be used with the Creo Parametric TOOLKIT libraries.
Navigating the Library/Class/Exception/Enumeration/Topic Selection Tree

Access all Object TOOLKIT C++ APIWizard online documentation data for libraries, classes, methods, enumerated types or the Object TOOLKIT C++ User’s Guide from the Library/Class/Exception/Enumeration Selection frame. This frame displays a tree structure of the data. Expand and collapse the tree to navigate this data.

To expand the tree structure, first select Object TOOLKIT C++ Libraries, Object TOOLKIT C++ Classes, Object TOOLKIT C++ Exceptions, Object TOOLKIT C++ Enumerations, or the Object TOOLKIT C++ User’s Guide at the top of the frame. The APIWizard displays the tree structure in a collapsed form. The switch icon to the far left of a node (i.e. a library, a class, an exception or chapter name) signifies that this node contains subnodes. If a node has no switch icon, it has no subnodes. Clicking the switch icon (or double-clicking on the node text) moves the switch to the down position. The APIWizard then expands the tree to display the subnodes. Select a node or subnode, and the APIWizard displays the online data in the Display frame.

Browsing the Object TOOLKIT C++ Libraries

View the Object TOOLKIT C++ libraries by choosing Object TOOLKIT C++ Libraries at the top of the Library/Classes/Exception/Topic frame. In this mode, the APIWizard displays all the Object TOOLKIT C++ libraries in the alphabetical order.

The Display frame for each Object TOOLKIT C++ library displays the information about all the classes and enumerated types that belong to that library.

Click the switch icon next to the desired library name, or double-click the library name text to view the classes or enumerated types that belong to that library. You can also view the methods for each class in the expanded tree by clicking the switch icon next to the class name, or by double-clicking the name.

The following figure shows the collapsed tree layout for the Object TOOLKIT C++ libraries.
Browsing the Object TOOLKIT C++ User’s Guide

View the *Object TOOLKIT C++ User’s Guide* by clicking the *Object TOOLKIT C++ User’s Guide* at the top of the *Library/Class/Interface/Exception/Topic* frame. In this mode, the APIWizard displays the section headings of the User’s Guide.

View a section by clicking the switch icon next to the desired section name or by double-clicking the section name. The APIWizard then displays a tree of subsections under the selected section. The text for the selected section and
its subsections appear in the **Display** frame. Click the switch icon again (or double-click the node text) to collapse the subnodes listed and display only the main nodes.

The following figure shows the collapsed tree layout for the table of contents of the *Object TOOLKIT C++ User’s Guide*.
APIWizard Search Feature (Find)

The APIWizard supports searches for specified strings against both the Object TOOLKIT C++ User’s Guide and API definition files. Click the Find button on the Library/Class/Exception/Enumeration/Topic frame to display the APIWizard Search dialog.

**Note**

The APIWizard Search feature is slow when accessed through Internet Explorer’s Default Virtual Machine. For better performance, access the APIWizard through Internet Explorer’s Java2 plug-in.

The Search dialog box contains the following fields, buttons, and frames:

- **Enter Search String(s)**
  Enter the specific search string or strings in this field. By default, the browser performs a non-case-sensitive search.

- **Search/Stop**
  Select the Search button to begin a search. During a search, this button name changes to Stop. Select the Stop button to stop a search.

- **Help**
Select this button for help about the APIWizard search feature. The APIWizard presents this help data in the **Display** frame.

- **Case Sensitive**
  Select this button to specify a case-sensitive search.

- **Search API References**
  Select this button to search for data on API methods. Select the **API Names** button to search for method names only. Select the **Definitions** button to search the API method names and definitions for specific strings.

- **Search Manuals**
  Select this button to search the *Object TOOLKIT C++ User’s Guide* data. Select the **Table of Contents** button to search on TOC entries only. Select the **Index** button to search only the Index. Select the **Contents** button to search on all text in the *Object TOOLKIT C++ User’s Guide*.

- **Name**
  This frame displays a list of strings found by the APIWizard search.

- **Found Under**
  This frame displays the location in the online help data where the APIWizard found the string.

**Supported Search Types**

The APIWizard Search supports the following:

- Case sensitive searches
- Search of API definitions, *Object TOOLKIT C++ User’s Guide* data, or both
- Search of API data by API names only or by API names and definitions
- Search of *Object TOOLKIT C++ User’s Guide* by Table of Contents only, by TOC and section titles, or on the User’s Guide contents (the entire text).
- Wildcard searches—valid characters are:
  - * (asterisk) matches zero or more non-whitespace characters
  - ? (question mark) matches one and only one non-whitespace character

To search for any string containing the characters Get, any number of other characters, and the characters Name

Get*Name

To search for any string containing the characters Get, one other character, and the characters Name

Get?Name
To search for any string containing the characters Get, one or more other characters, and the characters Name

Get*Name

To search on the string Feature, followed by an *

Feature\*

To search on the string Feature, followed by a ?

Feature\?

To search on the string Feature, followed by a \\

Feature\\

- Search string containing white space—Search on strings that contain space characters (white space) by placing double- or single-quote characters around the string.

"family table"
'Model* methods'

- Search on multiple strings—Separate multiple search strings with white space (tabs or spaces). Note that the default logical relationship between multiple search strings is OR.

To return all strings matching GetName OR GetId, enter:

Get*Name Get*Id

**Note**

This search specification also returns strings that match both specified search targets.

For example:

**fullName**

returns pfcModel::GetName and
pfcModelDescriptor::GetFullName

If a string matches two or more search strings, the APIWizard displays only one result in the search table, for example:

**fullName* Name**

returns only one entry for each **fullName** property found.

Mix quoted and non-quoted strings as follows:

Get*Name "family table"

returns all instances of strings containing Get and Name, or strings containing family table.
Performing an APIWizard Search

Follow these steps to search for information in the APIWizard online help data:

• Select the Find icon at the top of the Library/Class/Exception/Enumeration/Topic Selection frame.

• Specify the string or strings to be searched for in the Enter Search String field.

• Select Case Sensitive to specify a case-sensitive search. Note that the default search is non-case-sensitive.

• Select either or both of the Search API References and Search User’s Guide buttons. Select the options under these buttons as desired.

• Select the Search button. The APIWizard turns this button red and is renames it Stop for the duration of the search.

• If the APIWizard finds the search string in the specified search area(s), it displays the string in the Name frame. In the Where Found frame, the APIWizard displays links to the online help data that contains the found string.

• During the search, or after the search ends, select an entry in the Name or Where Found frames to display the online help data for that string. The APIWizard first updates the Library/Class/Exception/Enumeration/Topic Selection frame tree, and then presents in the Display frame the online help data for the selected string.
Session Objects

Overview of Session Objects ........................................................................................................ 48
Getting the Session Object .......................................................................................................... 48
Directories .................................................................................................................................. 49
Initializing Objects ..................................................................................................................... 51
Accessing the Creo Parametric Interface .................................................................................. 52

This chapter describes how to program on the session level using Object TOOLKIT C++. 
Overview of Session Objects

The Creo Parametric Session object (contained in the class pfcSession) is the highest level object in Object TOOLKIT C++. Any program that accesses data from Creo Parametric 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 Parametric 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:

• pfcGetProESession

The method pfcGetProESession gets a Session object in synchronous mode.

*Note*

You can make multiple calls to this method but each call will give 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 Parametric 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 Parametric executable.
The method `pfcGetProEVersion` returns a string that represent the Creo Parametric version.

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

**Note**

The preceding methods can only access information in synchronous mode.

## Directories

Methods Introduced:

- `pfcBaseSession::GetCurrentDirectory`
- `pfcBaseSession::ChangeDirectory`

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

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

## File Handling

Methods Introduced:

- `pfcBaseSession::ListFiles`
- `pfcBaseSession::ListSubdirectories`

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.

### 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 Parametric.

### Macros

Method Introduced:

• `pfcBaseSession::RunMacro`

The method `pfcBaseSession::RunMacro` runs a macro string. A 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.

Macros are executed from synchronous mode only when control returns to Creo Parametric from the Object TOOLKIT C++ program. Macros in synchronous mode are stored in reverse order (last in, first out).

Macros are executed in asynchronous mode as soon as they are registered. Macros in asynchronous mode are run in the same order that they are saved.

**Colors and Line Styles**

Methods Introduced:

- `pfcBaseSession::SetStdColorFromRGB`
- `pfcBaseSession::GetRGBFromStdColor`
- `pfcBaseSession::SetTextColor`
- `pfcBaseSession::SetLineStyle`

These methods control the general display of a Creo Parametric session.

Use the method `pfcBaseSession::SetStdColorFromRGB` to customize any of the Creo Parametric 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 allow you to initialize session objects.
Methods Introduced:

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

The method \texttt{wfcCreateMatrix3D} initializes a three-dimensional matrix with the specified values.

Use the methods \texttt{wfcCreatePoint2D} and \texttt{wfcCreatePoint3D} to initialize a two-dimensional and three-dimensional point respectively with the specified values.

The methods \texttt{wfcCreateOutline2D} and \texttt{wfcCreateOutline3D} initialize a two-dimensional and three-dimensional line respectively with the specified values.

Use the methods \texttt{wfcCreateVector2D} and \texttt{wfcCreateVector3D} to initialize a two-dimensional and three-dimensional vector respectively with the specified values.

**Accessing the Creo Parametric Interface**

The \texttt{Session} object has methods that work with the Creo Parametric interface. These methods provide access to the menu bar and message window. For more information on accessing menus, refer to the chapter \textit{Menus, Commands, and Pop-up Menus} on page 79.

**The Text Message File**

A text message file is where you define strings that are displayed in the Creo Parametric user interface. This includes the strings on the command buttons that you add to the Creo Parametric number, the help string that displays when the user’s 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 Parametric. Duplicate message file names or message key strings can cause Creo Parametric to exhibit unexpected behavior. To avoid conflicts with the names of Creo Parametric 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 Parametric only once during a session. If you make a change to the message file while Creo Parametric is running you must exit and restart Creo Parametric 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 Parametric 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 Object TOOLKIT C++. The topics are as follows:

• **Writing Messages to the Message Window** on page 54
• **Writing Messages to an Internal Buffer** on page 55

### 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 up to 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 Parametric 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 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.*

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

- **Prompt**—This 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 Object TOOLKIT C++ message is preceded by a blue dot. Info message types contain information such as user requests or feedback from Object TOOLKIT C++ or Creo Parametric. 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 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 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 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 messages 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 `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.

**Displaying Feature Parameters**

Method Introduced:

- `pfcSession::UIDisplayFeatureParams`

The method `pfcSession::UIDisplayFeatureParams` forces Creo Parametric 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`
- `pfcSession::UISaveFile`
- `pfcFileSaveOptions::Create`
- `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 relevant Creo Parametric dialog box for opening files and browsing directories. The method lets you specify several options through the input arguments pfcFileOpenOptions and pfcFileUIOptions.

Use the method pfcFileOpenOptions::Create to create a new instance of the pfcFileOpenOptions object. This object contains the following options:

• FilterString—Specifies the filter string for the type of file accepted by the dialog box. Multiple file types should be listed with wildcards and separated by commas, for example, *.prt, *.asm. Use the method pfcFileOpenOptions::SetFilterString to set this option.

• PreselectedItem—Specifies the name of an item to preselect in the dialog box. Use the method pfcFileOpenOptions::SetPreselectedItem to set this option.

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 `pfcSession::UIOpenFile` returns the file selected by you. The application must use other methods or techniques to perform the desired action on the file.

The method `pfcSession::UISaveFile` opens the Creo Parametric dialog box for saving a file. 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 `pfcSession::UISaveFile` returns the name of the file selected by you; the application must use other methods or techniques to perform the desired action on the file.

The method `pfcSession::UISelectDirectory` prompts the user to select a directory using the Creo Parametric 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 Parametric. 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 *File ▶ Save a Copy* dialog box in Creo Parametric. 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 *File ▶ 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 *File ▶ 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 *File ▶ Save a Copy* dialog box. The method pfcFileSaveRegisterListener::OnFileSaveRegister is called on clicking *OK* for the newly registered file type.

**Customizing the Creo Parametric Navigation Area**

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

To add custom Web pages to the navigation area, the 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 subsequent 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 Parametric .BIF, .GIF, .JPG, or .PNG. The new pane is displayed with the icon image. If you specify the value as NULL, the default Creo Parametric icon is used.

The default search paths for finding the icons are:

- `<creo_loadpoint>\Common Files\<datecode>\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.
This chapter describes how to use Interactive Selection in Object TOOLKIT C++.
Interactive Selection

Methods Introduced:

- `pfcBaseSession::Select`
- `pfcSelectionOptions::Create`
- `pfcSelectionOptions::SetMaxNumSels`
- `pfcSelectionOptions::SetOptionKeywords`

The method `pfcBaseSession::Select` activates the standard Creo Parametric menu structure 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`.

<table>
<thead>
<tr>
<th>Creo Parametric Database Item</th>
<th>String Identifier</th>
<th>ModelItem Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datum point</td>
<td><code>point</code></td>
<td><code>pfcITEM_POINT</code></td>
</tr>
<tr>
<td>Datum axis</td>
<td><code>axis</code></td>
<td><code>pfcITEM_AXIS</code></td>
</tr>
<tr>
<td>Datum plane</td>
<td><code>datum</code></td>
<td><code>pfcITEM_SURFACE</code></td>
</tr>
<tr>
<td>Coordinate system datum</td>
<td><code>csys</code></td>
<td><code>pfcITEM_COORD_SYS</code></td>
</tr>
<tr>
<td>Feature</td>
<td><code>feature</code></td>
<td><code>pfcITEM_FEATURE</code></td>
</tr>
<tr>
<td>Edge (solid or datum surface)</td>
<td><code>edge</code></td>
<td><code>pfcITEM_EDGE</code></td>
</tr>
<tr>
<td>Edge (solid only)</td>
<td><code>sldedge</code></td>
<td><code>pfcITEM_EDGE</code></td>
</tr>
<tr>
<td>Edge (datum surface only)</td>
<td><code>qltedge</code></td>
<td><code>pfcITEM_EDGE</code></td>
</tr>
<tr>
<td>Datum curve</td>
<td><code>curve</code></td>
<td><code>pfcITEM_CURVE</code></td>
</tr>
<tr>
<td>Composite curve</td>
<td><code>comp_crv</code></td>
<td><code>pfcITEM_CURVE</code></td>
</tr>
<tr>
<td>Surface (solid or quilt)</td>
<td><code>surface</code></td>
<td><code>pfcITEM_SURFACE</code></td>
</tr>
<tr>
<td>Surface (solid)</td>
<td><code>sldface</code></td>
<td><code>pfcITEM_SURFACE</code></td>
</tr>
<tr>
<td>Surface (datum surface)</td>
<td><code>qltface</code></td>
<td><code>pfcITEM_SURFACE</code></td>
</tr>
<tr>
<td>Quilt</td>
<td><code>dimqtl</code></td>
<td><code>pfcITEM_QUILT</code></td>
</tr>
<tr>
<td>Dimension</td>
<td><code>dimension</code></td>
<td><code>pfcITEM_DIMENSION</code></td>
</tr>
<tr>
<td>Reference dimension</td>
<td><code>ref_dim</code></td>
<td><code>pfcITEM_REF_DIMENSION</code></td>
</tr>
<tr>
<td>Integer parameter</td>
<td><code>ipar</code></td>
<td><code>pfcITEM_DIMENSION</code></td>
</tr>
<tr>
<td>Part</td>
<td><code>part</code></td>
<td>N/A</td>
</tr>
<tr>
<td>Part or subassembly</td>
<td><code>prt_or_asm</code></td>
<td>N/A</td>
</tr>
<tr>
<td>Creo Parametric Database Item</td>
<td>String Identifier</td>
<td>ModelItemType</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>Assembly component model</td>
<td>component</td>
<td>N/A</td>
</tr>
<tr>
<td>Component or feature</td>
<td>membfeat</td>
<td>pfcITEM_FEATURE</td>
</tr>
<tr>
<td>Detail symbol</td>
<td>dtl_symbol</td>
<td>pfcITEM_DTL_SYM_INSTANCE</td>
</tr>
<tr>
<td>Note</td>
<td>any_note</td>
<td>pfcITEM_NOTE, pfcITEM_DTL_NOTE</td>
</tr>
<tr>
<td>Draft entity</td>
<td>draft_ent</td>
<td>pfcITEM_DTL_ENTITY</td>
</tr>
<tr>
<td>Table</td>
<td>dwg_table</td>
<td>pfcITEM_TABLE</td>
</tr>
<tr>
<td>Table cell</td>
<td>table_cell</td>
<td>pfcITEM_TABLE</td>
</tr>
<tr>
<td>Drawing view</td>
<td>dwg_view</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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

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

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

**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`

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.

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 Parametric when selecting an item—it just repaints the wire-frame display in the new color. The highlight is removed if you use the View, 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
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`
- `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 156 in the Solid on page 155 chapter.

Some 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 Parametric, 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 Parametric offers the default selection buffer, the Edit selection buffer, and other more specialized buffers. Other Creo Parametric 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.

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`

Use the method `pfcSelectionBuffer::AddSelection` to add an item to the currently active 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.
This chapter describes the 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 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 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 true before customizing the ribbon user interface using the Object TOOLKIT C++ application. When you run Creo Parametric, the toolkitribbonui.rbn file is loaded along with the Object TOOLKIT C++ application and the commands created by the Object TOOLKIT C++ application appear in the ribbon user interface. Refer to the section About the Ribbon Definition File on page 73 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 Creo Parametric 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

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

1. Create a 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 Parametric application and load this Object TOOLKIT C++ application. The commands created by the 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 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 Object TOOLKIT C++ application registry file. For more information refer to the section on **Ribbon Definition File** on page 73.

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

9. Click **Apply**. The custom settings are saved to the `toolkitribbonui.rbn` file.

10. Reload the Object TOOLKIT C++ application or restart Creo Parametric. The `toolkitribbonui.rbn` file will be loaded along with the 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 C++ Applications** on page 77.

**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 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 C++ Applications on page 77.

Set the configuration option tk_enable_ribbon_custom_save to true 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 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 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 true.*

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 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 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 Object TOOLKIT C++ registry file.*

---

## Loading Multiple Applications Using the Ribbon Definition File

Creo Parametric supports loading of multiple `.rbn` files in the same session. You can develop multiple 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 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 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 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 Auxiliary Application User Interface**. The **Save UI Customization** dialog box opens.
7. Select a Object TOOLKIT C++ application and Click **Save**. The modified layout is saved to the **.rbn** file of the specified Object TOOLKIT C++ application.

The **Save UI Customization** dialog box is shown in the following image:
Localizing the Ribbon User Interface
Created by the Object TOOLKIT C++ Applications

The labels for the custom tabs, groups, and cascade menus belonging to the 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 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 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>
```
This chapter describes the methods provided by Object TOOLKIT C++ to create and modify menus, buttons, and pop-up menus in the Creo Parametric user interface.

Refer to the chapter Ribbon Tabs, Groups, and Menu Items on page 71 for more information. Also, refer to the Creo Parametric Help for more information on customizing the ribbon user interface.
Introduction

The Object TOOLKIT C++ classes enable you to supplement the Creo Parametric ribbon user interface. Once the 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>\Common Files\<datecode>\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 `pfcXCcancelProEAction` exception from the body of the listener method using `pfcXCcancelProEAction::Throw`.

Designating Commands

Using 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 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 71 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**— `<icon_name_16X16.ext>`
- **Large icon**— `<icon_name_32X32.ext>`

**Note**

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>\Common Files\<datecode>\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 **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 71 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, Object TOOLKIT C++ applications must attach a button to the pop-up menu during initialization of the pop-up menu. The 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`
- `pfcPopupmenuListener::OnPopupmenuCreate`

Use the method `pfcActionSource::AddActionListener` 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.
This chapter describes how to program on the model level using Object TOOLKIT C++. 
Overview of Model Objects

Models can be any Creo Parametric 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 Parametric 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 155.

Model Descriptors

Methods Introduced:
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 Creo Parametric 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 cause Creo Parametric to retrieve the model that corresponds to the `pfcModelDescriptor` argument.

The method `pfcBaseSession::RetrieveModel` retrieves the specified model into the Creo Parametric 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 ParametricNGINER 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.

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::GetFileName`
• `pfcModel::GetCommonName`
• `pfcModel::IsCommonNameModifiable`
• `pfcModel::GetFullName`
• `pfcModel::GetGenericName`
• `pfcModel::GetInstanceName`
• `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`

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 `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.

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 Parametric 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 Parametric workspace in the form of the `pfcDependencies` object.

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

**Model Operations**

Methods Introduced:

- `pfcModel::Backup`
- `pfcModel::Copy`
- `pfcModel::CopyAndRetrieve`
- `pfcModel::Rename`
- `pfcModel::Save`
- `pfcModel::Erase`
- `pfcModel::EraseWithDependencies`
• `pfcModel::Delete`  
• `pfcModel::Display`  
• `pfcModel::SetCommonName`  

These model operations duplicate most of the commands available in the Creo Parametric `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 Parametric from the Object TOOLKIT C++ application. Therefore, after calling them the control must be returned to Creo Parametric before calling any other function, otherwise the behavior of Creo Parametric 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`.  

---

96  
*Creo® Parametric 2.0 Object TOOLKIT C++ User’s Guide*
Running ModelCHECK

ModelCHECK is an integrated application that runs transparently within Creo Parametric. ModelCHECK uses a configurable list of company design standards and best modeling practices. You can configure 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 ModelCHECK from an external application using the method `pfcBaseSession::ExecuteModelCheck`. This method takes the model `Model` on which you want to run 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 ModelCHECK configuration files are used.
- **Mode**—Specifies the mode in which you want to run 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 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 ModelCHECK instructions described above.
Use the methods `pfcModelCheckInstructions::SetConfigDir`, `pfcModelCheckInstructions::SetMode`, `pfcModelCheckInstructions::SetOutputDir`, and `pfcModelCheckInstructions::SetShowInBrowser` to modify the ModelCHECK instructions.

The method `pfcBaseSession::ExecuteModelCheck` returns the results of the 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 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 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 ModelCHECK report when you select the name.
- DSC_<checkname>—Specifies the name of the check as it appears in the 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 ModelCHECK configuration file.

4. Register the ModelCHECK check from the Object TOOLKIT C++ application.

**Note**

*Other than the requirements listed above, Object TOOLKIT C++ custom checks do not have access to the rest of the values in the 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 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 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 100** 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 ModelCHECK report.

![ModelCHECK](image)

**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 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 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 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 ModelCHECK report with an **Action** button that invokes the `pfcModelCheckCustomCheckListener::OnCustomCheckAction` function.
This chapter describes how to program drawing functions using Object TOOLKIT C++.
Overview of Drawings in Object TOOLKIT C++

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

Unless otherwise specified, 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 Object TOOLKIT C++. Creo Parametric 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.
- pfcDWGCreative_ERR_NO_STEP_NUM—The process step number does not exist.
- pfcDWGCreative_ERR TEMPLATE USED—The template does not exist.
- pfcDWGCreative_ERR_NO_PARENT_VIEW FOR PROJ—There is no possible parent view for this projected view.
- pfcDWGCreative_ERR_CANT_GET_PROJ_PARENT—Could not get the projected parent for a drawing view.
- pfcDWGCreative_ERR_SEC_NOT_PARALLEL—The designated cross section was not parallel to the created view.
- pfcDWGCreative_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.

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` sets 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 118.

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 111.

## Drawing Sheets

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

- `pfcSheetOwner::GetSheetTransform`
- `pfcSheetOwner::GetSheetInfo`
• `pfcSheetOwner::GetSheetScale`
• `pfcSheetOwner::GetSheetFormat`
• `pfcSheetOwner::GetSheetFormatDescr`
• `pfcSheetOwner::GetSheetBackgroundView`
• `pfcSheetOwner::GetNumberOfSheets`
• `pfcSheetOwner::GetCurrentSheetNumber`
• `pfcSheetOwner::GetSheetUnits`

Superseded Method:
• `pfcSheetOwner::GetSheetData`

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.

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`

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.

**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` creates 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`
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.

**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.

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 `wfcDisplayStyle`:

- `wfcDISPSTYLE_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`.
- `wfcDISPSTYLE_WIREFRAME`—Shows all edges in wireframe style.
- `wfcDISPSTYLE_HIDDEN_LINE`—Shows all edges in hidden line style.
- `wfcDISPSTYLE_NO_HIDDEN`—Removes all hidden edge from view display.
- `wfcDISPSTYLE_SHADED`—Shows the view in shaded display mode.
- `wfcDISPSTYLE_FOLLOW_ENVIRONMENT`—Uses the setting from Creo Parametric Options dialog box, under Entity Display, Geometry display settings, or the view display style icon in the Creo Parametric graphics window.
- `wfcDisplayStyle_nil`—NULL value.
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.
• wfcTangentEdgeDisplay_nil—NULL value.

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 Object TOOLKIT C++ 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:
• **pfcDIMLINA_OCTANGENT_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.

• **pfcDIMLINA_OCTANGENT_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.

• **pfcDIMLINA_OCTANGENT_LEFT1**—The tangent is to the left of the line, and is on the opposite side of the line.

• **pfcDIMLINA_OCTANGENT_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:

• **pfcDimension2D::GetIsAssociative**

• **pfcDimension2D::GetIsReference**

• **pfcDimension2D::GetIsDisplayed**
• pfcDimension2D::GetAttachmentPoints
• pfcDimension2D::GetDimensionSenses
• pfcDimension2D::GetOrientationHint
• pfcDimension2D::GetBaselineDimension
• pfcDimension2D::GetLocation
• pfcDimension2D::GetView
• pfcDimension2D::GetTolerance
• pfcDimension2D::GetIsToleranceDisplayed

The method pfcDimension2D::GetIsAssociative returns whether the dimension or reference dimension in a drawing is associative.

The method pfcDimension2D::GetIsReference determines whether the drawing dimension is a reference dimension.

The method pfcDimension2D::GetIsDisplayed determines whether the dimension will be displayed in the drawing.

The method pfcDimension2D::GetAttachmentPoints returns a sequence of attachment points. The dimension senses array returned by the method pfcDimension2D::GetDimensionSenses gives more information on how these attachments are interpreted.

The method pfcDimension2D::GetDimensionSenses returns a sequence of dimension senses, describing how the dimension is attached to each attachment returned by the method pfcDimension2D::GetAttachmentPoints.

The method pfcDimension2D::GetOrientationHint returns the orientation hint for placing the drawing dimensions. The orientation hint determines how Creo Parametric will orient the dimension with respect to the attachment points.

Note
This methods described above are applicable only for dimensions created in the drawing mode. It does not support dimensions created at intersection points of entities.

The method pfcDimension2D::GetBaselineDimension returns an ordinate baseline drawing 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 pfcDimension2D::GetLocation returns the placement location of the dimension.
The method `pfcDimension2D::GetView` returns the drawing view in which the dimension is displayed. This method applies to dimensions stored in the solid or in the drawing.

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.

Use the method `pfcDimension2D::GetIsToleranceDisplayed` determines whether or not the dimension’s tolerance is displayed in the drawing.

**Drawing Dimensions Operations**

Methods Introduced:

- `pfcDimension2D::ConvertToLinear`
- `pfcDimension2D::ConvertToOrdinate`
- `pfcDimension2D::ConvertToBaseline`
- `pfcDimension2D::SetLocation`
- `pfcDimension2D::SwitchView`
- `pfcDimension2D::SetTolerance`
- `pfcDimension2D::EraseFromModel2D`
- `pfcModel2D::SetViewDisplaying`

The method `pfcDimension2D::ConvertToLinear` converts an ordinate drawing dimension to a linear drawing dimension. The drawing containing the dimension must be displayed.

The method `pfcDimension2D::ConvertToOrdinate` converts a linear drawing dimension to an ordinate baseline dimension.

The method `pfcDimension2D::ConvertToBaseline` converts a location on a linear drawing dimension to an ordinate baseline dimension. The method returns the newly created baseline dimension.

**Note**

_The method updates the display of the dimension only if it is currently displayed._

The method `pfcDimension2D::SetLocation` sets the placement location of a dimension or reference dimension in a drawing.

The method `pfcDimension2D::SwitchView` changes the view where a dimension created in the drawing is displayed.

The method `pfcDimension2D::SetTolerance` assigns the upper and lower tolerance limits of the drawing dimension.
The method `pfcDimension2D::EraseFromModel2D` permanently erases the dimension from the drawing.

The method `pfcModel2D::SetViewDisplaying` changes the view where a dimension created in a solid model is displayed.

## Drawing Tables

A drawing table in 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 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`
• `pfcTableOwner::RetrieveTable`

The method `pfcTableRetrieveInstructions::Create` creates the `pfcTableRetrieveInstructions` data object that describes how to retrieve a drawing table using the method `pfcTableOwner::RetrieveTable`. 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**—The location of left top corner of the retrieved 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 data object contains information on the table to retrieve and is returned by the method `pfcTableRetrieveInstructions::Create`.

### 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` verifies if the drawing table was created using the format of the drawing sheet specified by the sheet number. 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`

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 Parametric will delay the repaint, allowing you to perform several operations before showing changes on the screen.*

### 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`

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.

### 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`.

**Detail Items**

The methods described in this section operate on detail items.

In 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 Parametric. 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 Parametric 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.

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 a Detail Item**

Methods Introduced:

• `pfcDetailItemOwner::CreateDetailItem`
• `pfcDetailGroupInstructions::Create`

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.

**Detail Entities**

A detail entity in 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 255.

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 Parametric. You can create and insert supported OLE objects into a two-dimensional Creo Parametric 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 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.

**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`, etc.
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*

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 Parametric 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::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::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 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 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::RetrieveSymbolDefinition`

The method `pfcDetailItemOwner::RetrieveSymbolDefinition` retrieves a symbol definition from the disk.

The input parameters of this method are:

- `FileName`—Name of the symbol definition file
- `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 Parametric 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 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 Properties dialog box in the Creo Parametric 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 147 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 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 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.
Most of the objects and methods in 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

MethodsIntroduced:

- pfcBaseSession::CreatePart
- pfcBaseSession::CreateAssembly
- pfcComponentPath::GetRoot
- pfcComponentPath::GetLeaf
- pfcMFG::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.

Solid Information

MethodsIntroduced:

- 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.*

Solid Operations

Methods Introduced:
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, 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.
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**—Creo Parametric

  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**—Creo Parametric **Model Tree**

  Use the method `pfcRegenInstructions::SetRefreshModelTree` to modify this parameter.

- **ResumeExcludedComponents**—Creo Parametric

  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.
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**—Gets and sets the state of regeneration failure window in a model using the methods `wfcWRegenInstructions::GetAllowConfirm` and `wfcWRegenInstructions::SetAllowConfirm`
- **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.

The method `pfcSolid::GetGeomOutline` returns the three-dimensional bounding box for the specified solid. The method `pfcSolid::Solid.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.
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 Gram Second (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 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 Parametric, 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 Parametric Default—Inch lbm Second. This is the default system followed by Creo Parametric.
  • FPS—Foot Pound Second
  • IPS—Inch Pound Second
  • mmNS—Millimeter Newton Second

In 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

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

\[ F = a + (C \times b) \]

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 Parametric 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.

### 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 160.

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:

- **pfcSolid::SetPrincipalUnits**
- **pfcUnitConversionOptions::Create**
- **pfcUnitConversionOptions::SetDimensionOption**
- **pfcUnitConversionOptions::SetIgnoreParamUnits**

The method `pfcSolid::SetPrincipalUnits` changes the principal system of units assigned to the solid model based on 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 to 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

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).

Annotations
Methods Introduced:
• pfcNote::GetLines
• pfcNote::SetLines
• pfcNote::GetText
• pfcNote::GetURL
• pfcNote::SetURL
• pfcNote::Display
• pfcNote::Delete
• pfcNote::GetOwner
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.

## Cross Sections

Methods Introduced:

- pfcSolid::ListCrossSections
- pfcSolid::GetCrossSection
- pfcXSection::GetName
- pfcXSection::SetName
- pfcXSection::GetXSecType
- 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 Creo Parametric. The method pfcXSection::SetName modifies the cross section name.

The method pfcXSection::GetXSecType returns the type of cross section, that is planar or offset, and the type of item intersected by the cross section.

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 a cross section.

**Materials**

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`

The method `pfcMaterial::Save` writes to a material file that can be imported into any Creo Parametric part.

The method `pfcMaterial::Delete` removes material from the part.

The method `pfcPart::GetCurrentMaterial` returns the currently assigned material for the part.

The method `pfcPart::SetCurrentMaterial` sets the material assigned to the part.
Note
By default, while assigning a 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 calculations of the sheetmetal part. However, you
can avoid this behavior by setting the value of the configuration option
material_update_smt_bend_table to never_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_MATERIAL_NAME is a part of the family table.
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.

Accessing Material Types
Methods Introduced:
• pfcMaterial::GetStructuralMaterialType
• pfcMaterial::SetStructuralMaterialType
• pfcMaterial::GetThermalMaterialType
• pfcMaterial::SetThermalMaterialType
• pfcMaterial::GetSubType
• pfcMaterial::SetSubType
• pfcMaterial::GetPermittedSubTypes
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 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.

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.

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.

**Accessing Material Properties**

The methods listed in this section enable you to access material properties.

Methods Introduced:

- `pfcMaterialProperty::Create`
- `pfcMaterial::GetPropertyValues`
• 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

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
\texttt{pfcMaterial::SetFailureCriterion} to set the reduction factor for the
failure strength of the material.

Use the method \texttt{pfcMaterial::GetPermittedFailureCriteria} to
retrieve a list of permitted string values for the material failure criterion.

The methods \texttt{pfcMaterial::GetHardness} and
\texttt{pfcMaterial::SetHardness} return and set the hardness for the specified
material respectively.

The methods \texttt{pfcMaterial::GetHardnessType} and
\texttt{pfcMaterial::SetHardnessType} return and set the hardness
type for the specified material respectively.

The methods \texttt{pfcMaterial::GetCondition} and
\texttt{pfcMaterial::GetCondition} return and set the condition for the specified
material respectively.

The methods \texttt{pfcMaterial::GetBendTable} and
\texttt{pfcMaterial::SetBendTable} return and set the bend table for the specified
material respectively.

The methods \texttt{pfcMaterial::GetCrossHatchFile} and
\texttt{pfcMaterial::SetCrossHatchFile} return and set the file containing the
crosshatch pattern for the specified material respectively.

The methods \texttt{pfcMaterial::GetMaterialModel} and
\texttt{pfcMaterial::SetMaterialModel} return and set the type of hyperelastic
isotropic material model respectively.

Use the method \texttt{pfcMaterial::GetPermittedMaterialModels} to
retrieve a list of the permitted string values for the material model.

The methods \texttt{pfcMaterial::GetModelDefByTests} determines whether
the hyperelastic isotropic material model has been defined using experimental
data for stress and strain.

Use the method \texttt{pfcMaterial::SetModelDefByTests} to define the
hyperelastic isotropic material model using experimental data for stress and strain.

**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
\texttt{pfcMaterial} is a child of \texttt{pfcParameterOwner}, which provides access to
user-defined parameters and properties of materials through the methods in that
interface.
10

Curve and Surface Collection

Introduction to Curve and Surface Collection ................................................................. 176
Interactive Collection......................................................................................................... 177
Programmatic Access to Collections .................................................................................. 179

This chapter describes the 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 Parametric 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 Parametric 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

Use the method wfcWSolid::CollectCurves to interactively create a collection of curves by invoking a 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 by 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.

Programmatic Access to Collections

The wfcCollection Interface

The wfcCollection interface contains surface and curve collection interfaces for programmatic access to collections. It represents a chain or surface set and extracts the details from an appropriate structure from Creo Parametric.

Method Introduced:

• wfcCollection::Create

The method wfcCollection::Create creates a data object that contains information about the collection of curves and surfaces.

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`
- `wfcSurfaceCollectionReference::Create`
- `wfcSurfaceCollectionReference::GetRefType`
- `wfcSurfaceCollectionReference::SetRefType`
• `wfcSurfaceCollectionReference::GetReference`
• `wfcSurfaceCollectionReference::SetReference`

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 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`.

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.
- `wfcSURFCOLLSAME_SRF_LST`—Instruction allowing selections from same solid or same quilt.
- `wfcSURFCOLL_USE_BACKUP`—Instruction prompting Creo Parametric 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.

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 "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.
Object TOOLKIT C++ provides access to Creo Parametric windows and saved views. This chapter describes the methods that provide this access.
Windows

This section describes the Object TOOLKIT C++ methods that access Window objects. The topics are as follows:

- Getting a Window Object on page 188
- Window Operations on page 189

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 Parametric.

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.
**Window Operations**

Methods Introduced:

- `pfcWindow::GetHeight`
- `pfcWindow::GetWidth`
- `pfcWindow::GetXPos`
- `pfcWindow::GetYPos`
- `pfcWindow::GetGraphicsAreaHeight`
- `pfcWindow::GetGraphicsAreaWidth`
- `pfcWindow::Clear`
- `pfcWindow::Repaint`
- `pfcWindow::Refresh`
- `pfcWindow::Close`
- `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 Parametric 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 Parametric started, this method clears the window. Otherwise, it removes the window from the screen.

The method `pfcWindow::Activate` activates a window. This function is available only in the asynchronous mode.
The method `pfcWindow::GetId` retrieves the ID of the Creo Parametric 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`

The methods `pfcWindow::GetURL` and `pfcWindow::SetURL` enables you to find and change the URL displayed in the embedded browser in the Creo Parametric window.

The methods `pfcWindow::GetBrowserSize` and `pfcWindow::SetBrowserSize` enables you to find and change the size of the embedded browser in the Creo Parametric window.

**Note**

*The methods `pfcWindow::GetBrowserSize` and `pfcWindow::SetBrowserSize` are not supported if the browser is open in a separate window.*

### Views

This section describes the Object TOOLKIT C++ methods that access `pfcView` objects. The topics are as follows:

- Getting a View Object on page 190
- View Operations on page 191

### 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.

The method pfcViewOwner::GetCurrentView 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 Parametric and accessible from Object TOOLKIT C++ and how to transform from one coordinate system to another.
Coordinate Systems

Creo Parametric and Object TOOLKIT C++ use the following coordinate systems:

- Solid Coordinate System on page 192
- Screen Coordinate System on page 192
- Window Coordinate System on page 193
- Drawing Coordinate System on page 193
- Drawing View Coordinate System on page 193
- Assembly Coordinate System on page 193
- Datum Coordinate System on page 193
- Section Coordinate System on page 194

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 Parametric 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 Parametric 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 Parametric user.

Solid coordinates are used by 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 Parametric window. 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 nominal pixel counts. The bottom, left corner of the default window is at (0, 0) and the top, right corner is at (1000, 864).

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, except it is not affected by zoom and pan. When an object is first displayed in a window, or the option **View, Pan/Zoom, Reset** is used, the screen and window coordinates are the same.

Window coordinates are needed only if you take account of zoom and pan. For example, you can find out whether a point on the solid is visible in the window or you can 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 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 an 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

All coordinate systems are treated in 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:

\[
\begin{bmatrix}
X' & Y' & Z' & 1 \\
\end{bmatrix} = \begin{bmatrix}
X & Y & Z & 1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \begin{bmatrix}
... & ... & 0 \\
... & ... & 0 \\
... & ... & 0 \\
Xs & Ys & Zs & 1 \\
\end{bmatrix}
\]
The utility method `pfcTransform3D::Invert` inverts a transformation matrix so that it can be used to transform points in the opposite direction.

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.

**Transforming to Screen Coordinates**

Methods Introduced:

- `pfcView::GetTransform`
- `pfcView::SetTransform`
- `pfcView::Rotate`

The view matrix describes the transformation from solid to screen coordinates. The method `pfcView::GetTransform` provides the view matrix for the specified view. The method `pfcView::SetTransform` allows you to specify a matrix for the view.

The method `pfcView::Rotate` rotates a view, relative to the X, Y, or Z axis, in the amount that you specify.

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.
This chapter describes the 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` interface. 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`

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.
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 method 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.

**ModellItem Information**

Methods Introduced:

- `pfcModelItem::GetName`
- `pfcModelItem::SetName`
- `pfcModelItem::GetId`
- `pfcModelItem::GetType`

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 `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 198 for the list of possible model item types.

**Duplicating ModelItems**

Methods Introduced:
• **pfcBaseSession::AllowDuplicateModelItems**

You can control the creation of ModelItems more than twice for the same Creo Parametric 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, although 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 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 198* 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**

Superseded Method:
• **pfcLayer::HasUnsupportedItems**

The methods `pfcLayer::GetStatus` and `pfcLayer::SetStatus` enables you to access the display status of a layer. The corresponding enumeration class is `pfcDisplayStatus` and the possible values are `Normal`, `Displayed`, `Blank`, or `Hidden`.

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`,
• `pfcITEMCOORD_SYS`,
• `pfcITEM_AXIS`,
• `pfcITEMSIMPREP`,
• `pfcITEM_DTL_SYM_DEFINITION`,
• `pfcITEM_DTL_OLE_OBJECT`,
• `pfcITEMEXPLODED_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`. 
13

Feature Element Tree

Overview of Feature Creation ................................................................. 204
Feature Element Values ........................................................................ 206
Feature Element Special Values ............................................................ 206
Feature Element Paths ........................................................................ 207
Feature Element Tree ........................................................................... 207
Creating FET Using WCreateFeature ..................................................... 208
Examples of Feature Creation ............................................................... 209
Feature Elements .................................................................................. 209
Creating Patterns ................................................................................. 210
Redefining Features .............................................................................. 210
Element Diagnostics ............................................................................ 211

This chapter explains feature creation in 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. 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 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 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 Object TOOLKIT C++ and Creo Parametric TOOLKIT can be used together. They share the same definitions of FET element IDs and values. You must refer to the header files in `protoolkit\includes` for feature-specific element trees.

- 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 Object TOOLKIT C++ can check for errors each time you add a new element to the tree.

The header files in protookit\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 374 in Task Based Application Libraries on page 373.

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::WriteElementTreeToFile
• wfcWSession::CreateElementTreeFromFile
• 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.*

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**

The wfcWSolid object identifies the solid that is to contain the new feature. The method wfcWSolid::WCreateFeature creates a feature from the FET.

The syntax of wfcWSolid::WCreateFeature is as follows:

```cpp
wfcWFeature_ptr wfcWSolid::WCreateFeature ( 
    wfcElementTree_ptr Tree,
    wfcFeatCreateOptions_ptr Options,
    optional wfcWRegenInstructions_ptr Instrs);
```

---

208 Creo® Parametric 2.0 Object TOOLKIT C++ User’s Guide
Examples of Feature Creation

The folder `< creo_otk_loadpoint>\otk_examples\otk_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`
- `wfcElement::GetIdAsString`
- `wfcElement::GetIsArray`
- `wfcElement::GetIsCompound`
- `wfcElement::GetIsMultival`
- `wfcElement::GetChildren`
- `wfcWSession::GetElemWstrOption`
- `wfcWSession::SetElemWstrOption`
- `wfcElementWstringOption::GetExpression`
- `wfcElementWstringOption::SetExpression`
- `wfcElementWstringOption::GetPositive`
- `wfcElementWstringOption::SetPositive`
- `wfcElementWstringOption::GetSign`
- `wfcElementWstringOption::SetSign`
- `wfcElement::GetValueAsString`
- `wfcElement::SetValueAsString`

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 method `wfcElement::GetIdAsString` returns the string representation of the specified element ID.

The methods `wfcElement::GetIsArray`, `wfcElement::GetIsCompound`, and `wfcElement::GetIsMultival` are used to determine the type of the specified element in a tree. The methods `wfcElement::GetIsArray` and `wfcElement::GetIsCompound`
determine if the specified element contains an array of elements, or is a compound. 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 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::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::GetValueAsString returns a string value representation for double and integer elements. The options set in the object wfcElementWstringOption decide the format of the output.

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::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.

The methods `wfcElementDiagnostic::GetDiagnosticMessage` and `wfcElementDiagnostic::GetSeverity` get the message and severity of the diagnostic item of the element.
All Creo Parametric solid models are made up of features. This chapter describes how to program on the feature level using 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`
• `pfcFeatureGroup::GetGroupLeader`
• `pfcFeaturePattern::GetPatternLeader`
• `pfcFeaturePattern::ListMembers`
• `pfcSolid::ListFailedFeatures`
• `wfcWSolid::ListChildOfExternalFailedFeatures`
• `wfcWSolid::ListChildOfFailedFeatures`
• `pfcSolid::ListFeaturesByType`
• `pfcSolid::GetFeatureById`

The methods `pfcFeature::ListChildren` and `pfcFeature::ListParents` return a sequence of features that contain all the children or parents of the specified feature.

To get the first feature in the specified group access the method `pfcFeatureGroup::GetGroupLeader`.

The methods `pfcFeaturePattern::GetPatternLeader` and the method `pfcFeaturePattern::ListMembers` return features that make up the specified feature pattern. See the section Feature Groups and Patterns on page 219 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 Parametric creates internally. Use the method’s `VisibleOnly` argument to exclude them.

The method `pfcSolid::GetFeatureById` returns the feature object with the corresponding integer identifier.
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`

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.
- `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.

## 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**

The method `pfcSolid::ExecuteFeatureOps` is not supported in the No-Resolve mode. It throws an exception `pfcXToolkitBadContext`. 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. Refer to the Solid Operations on page 156 section in the Solid on page 155 chapter for more information on the No-Resolve mode.

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:

- \textit{NewFeatNum}—The intended location of the first feature in the specified list.

### Feature Groups and Patterns

Patterns are treated as features in Creo Parametric. A feature type, \texttt{pfcFEATTYPE\_PATTERN\_HEAD}, is used for the pattern header feature.

\textit{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:

- \texttt{pfcFeature::GetGroup}
- \texttt{pfcFeature::GetPattern}
- \texttt{pfcSolid::CreateLocalGroup}
- \texttt{pfcFeatureGroup::GetPattern}
- \texttt{pfcFeatureGroup::GetGroupLeader}
- \texttt{pfcFeaturePattern::GetPatternLeader}
- \texttt{pfcFeaturePattern::ListMembers}
- \texttt{pfcFeaturePattern::Delete}

The method \texttt{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 \texttt{pfcFeatureGroup::GetGroupLeader}.

The methods \texttt{pfcFeaturePattern::GetPatternLeader} and \texttt{pfcFeaturePattern::ListMembers} return features that make up the specified feature pattern.

The methods \texttt{pfcFeature::GetPattern} and \texttt{pfcFeatureGroup::GetPattern} return the \texttt{pfcFeaturePattern} object that contains the corresponding \texttt{pfcFeature} or \texttt{pfcFeatureGroup}. Use the method \texttt{pfcSolid::CreateLocalGroup} to take a sequence of features and create a local group with the specified name. To delete a \texttt{pfcFeaturePattern} object, call the method \texttt{pfcFeaturePattern::Delete}. 
User Defined Features

Groups in Creo Parametric 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 Parametric 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 Parametric. PTC recommends that you read about UDF’s in the Creo Parametric on-line help, and practice defining and using UDF’s in Creo Parametric before you attempt to use this method.

When you create a UDF interactively, Creo Parametric prompts you for the information it needs to fix the properties of the resulting features. When you create a UDF from 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 Parametric 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 Parametric 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 in respect to a datum plane Creo Parametric must know what direction the new feature will point to. When you create such a UDF interactively Creo Parametric prompt 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 Parametric 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.

To pass all the above values to Creo Parametric, Object TOOLKIT C++ uses a special class that prepares and sets all the options and passes them to Creo Parametric.

**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`

This method 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 Parametric gives you when you create a UDF interactively. PTC recommends that you experiment with creating the UDF interactively using Creo Parametric before you write the Object TOOLKIT C++ code to fill the structure.

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:

• `pfcUDFSCALESAME_SIZE`
• `pfcUDFSCALESAME_DIMS`
• `pfcUDFSCALECUSTOM`
• `pfcUDFSCALE nil`

**Note**
The default value is `pfcUDFSCALESAME_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 Parametric 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:

• `pfcUDFDISPLAYNORMAL`
• `pfcUDFDISPLAYREADONLY`
• `pfcUDFDISPLAYBLANK`

**Note**
The default value is `pfcUDFDISPLAYNORMAL` if this option is not set.

### Setting the Variable Dimensions and Parameters

Methods Introduced:

• `pfcUDFCustomCreateInstructions::setVariantValues`
• `pfcUDFVariantValues::create`
• `pfcUDFVariantValues::insert`
• `pfcUDFVariantDimension::create`
• `pfcUDFVariantPatternParam::create`
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 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.

**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` programatically or prompt the user for a selection separately. You cannot set an embedded datum as the UDF reference.

There are two types of reference:

- **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.

### Setting Orientations

Methods Introduced:

- `pfcUDFCustomCreateInstructions::SetOrientations`
- `pfcUDFOrientations::create`
- `pfcUDFOrientations::insert`

`pfcUDFOrientations` class represents an array of orientations that provide the answers to Creo Parametric 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 Parametric prompts for them when the UDF is created interactively. If you do not provide an orientation that Creo Parametric needs, it uses the default value NO_FLIP.

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 Parametric prompts for the feature positions. The order of quadrants should correspond to the order in which Creo Parametric 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.
15 Datum Features

Datum Plane Features ................................................................. 230
Datum Axis Features .................................................................. 232
General Datum Point Features .................................................... 234
Datum Coordinate System Features ............................................. 235

This chapter describes the 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::Create
- pfcDatumPlaneThroughConstraint::GetThroughRef
- pfcDatumPlaneNormalConstraint::Create
- pfcDatumPlaneNormalConstraint::GetNormalRef
- pfcDatumPlaneParallelConstraint::Create
- pfcDatumPlaneParallelConstraint::GetParallelRef
- pfcDatumPlaneTangentConstraint::Create
- pfcDatumPlaneTangentConstraint::GetTangentRef
- pfcDatumPlaneOffsetConstraint::Create
- pfcDatumPlaneOffsetConstraint::GetOffsetRef
- pfcDatumPlaneOffsetConstraint::GetOffsetValue
- pfcDatumPlaneOffsetCoordSysConstraint::Create
- pfcDatumPlaneOffsetCoordSysConstraint::GetCsysAxis
- pfcDatumPlaneAngleConstraint::Create
- pfcDatumPlaneAngleConstraint::GetAngleRef
- pfcDatumPlaneAngleConstraint::GetAngleValue
- pfcDatumPlaneSectionConstraint::Create
- pfcDatumPlaneSectionConstraint::GetSectionRef
- pfcDatumPlaneSectionConstraint::GetSectionIndex
- pfcDatumPlaneDefaultXConstraint::Create
- pfcDatumPlaneDefaultYConstraint::Create
- pfcDatumPlaneDefaultZConstraint::Create

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::Create to create a new object. 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::Create to create a new object. 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 pfcDatumPlaneFeatDatumPlaneParallelConstraint::Create to create a new object. Use the method pfcDatumPlaneParallelConstraint::GetParallelRef to get the reference selection handle for the Parallel constraint.

- **pfcDTMPLN_TANG**—Specifies the Tangent constraint. The pfcDatumPlaneTangentConstraint specifies this constraint. Use the method pfcDatumPlaneTangentConstraint::Create to create a new object. 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::Create to create a new object. 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::Create` to create a new 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::Create` to create a new object. 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::Create` to create a new object. Use the method `pfcDatumPlaneSectionConstraint::GetSectionRef` to get the reference selection for the Section constraint. Use the method `pfcDatumPlaneSectionConstraint::GetSectionIndex` to get the section index.

- **pfcDTMPLN_DEF_X**—Specifies the default RIGHT constraint for the datum plane. The `pfcDatumPlaneDefaultXConstraint` object specifies this constraint. Use the method `pfcDatumPlaneDefaultXConstraint::Create` to create a new object.

- **pfcDTMPLN_DEF_Y**—Specifies the default TOP constraint for the datum plane. The `pfcDatumPlaneDefaultYConstraint` object specifies this constraint. Use the method `pfcDatumPlaneDefaultYConstraint::Create` to create a new object.

- **pfcDTMPLN_DEF_Z**—Specifies the default FRONT constraint for the datum plane. The `pfcDatumPlaneDefaultZConstraint` object specifies this constraint. Use the method `pfcDatumPlaneDefaultZConstraint::Create` to create a new object.

**Datum Axis Features**

The properties of the Datum Axis feature are defined in the `pfcDatumAxisFeat` data object.
Methods Introduced:

- `pfcDatumAxisFeat::GetConstraints`
- `pfcDatumAxisConstraint::Create`
- `pfcDatumAxisConstraint::GetConstraintType`
- `pfcDatumAxisConstraint::GetConstraintRef`
- `pfcDatumAxisFeat::GetDimConstraints`
- `pfcDatumAxisDimensionConstraint::Create`
- `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.

  Use the method `pfcDatumAxisConstraint::Create` to create a new `pfcDatumAxisConstraint` object. 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.
Use the method pfcDatumAxisDimensionConstraint::Create to create a new pfcDatumAxisDimensionConstraint object. This 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
- pfcDatumPointPlacementConstraint::Create
- pfcGeneralDatumPoint::GetPlaceConstraints
- pfcDatumPointDimensionConstraint::Create
- 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 pfcDatumPointPlacementConstraint::Create to create a new 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 pfcDatumPointDimensionConstraint::Create to create a new 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::Create
- pfcDatumCsysOriginConstraint::GetOriginRef
- pfcCoordSysFeat::GetDimensionConstraints
- pfcDatumCsysDimensionConstraint::Create
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. Use the method pfcDatumCsysOriginConstraint::Create to create a new pfcDatumCsysOriginConstraint object. 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. Use the method pfcDatumCsysDimensionConstraint::Create to create a new pfcDatumCsysDimensionConstraint object. 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 \texttt{pfcDatumCsSysDimensionConstraint::GetDimConstraintType} to get the constraint type. The constraint types are:

- \texttt{pfcDTMCSYS\_DIM\_OFFSET}—Specifies the offset type constraint.
- \texttt{pfcDTMCSYS\_DIM\_ALIGN}—Specifies the align type constraint.

- OrientationConstraints—Specifies a collection of orientation constraints (given by the \texttt{CoordSysFeat.DatumCsSysOrientMoveConstraint} object). Use the method \texttt{pfcCoordSysFeat::GetOrientationConstraints} to obtain the collection of orientation constraints for the coordinate system. Use the method \texttt{pfcDatumCsSysOrientMoveConstraint::Create} to create a new \texttt{pfcDatumCsSysOrientMoveConstraints} object. This object contains the following attributes:
  - \texttt{OrientMoveConstraintType}—Specifies the type of orientation for the constraint. The orientation type is given by the \texttt{pfcDatumCsSysOrientMoveConstraintType} enumerated type. Use the method \texttt{pfcDatumCsSysOrientMoveConstraint::GetOrientMoveConstraintType} to get the orientation type.
  - \texttt{OrientMoveValue}—Specifies the reference value for the constraint. Use the method \texttt{pfcDatumCsSysOrientMoveConstraint::GetOrientMoveValue} to get the reference value.

- \texttt{IsNormalToScreen}—Specifies if the coordinate system is normal to screen. Use the method \texttt{pfcCoordSysFeat::GetIsNormalToScreen} to determine if the coordinate system is normal to screen.

- \texttt{OffsetType}—Specifies the offset type of the coordinate system in terms of the \texttt{pfcDatumCsSysOffsetType} enumerated type. Use the method \texttt{pfcCoordSysFeat::GetOffsetTable} to get the offset type. The offset types are:
  - \texttt{pfcDTMCSYS\_OFFSET\_CARTESIAN}—Specifies a cartesian coordinate system that has been defined by setting the values for the \texttt{pfcDTMCSYS\_MOVE\_TRAN\_X}, \texttt{pfcDTMCSYS\_MOVE\_TRAN\_Y}, and \texttt{pfcDTMCSYS\_MOVE\_TRAN\_Z} or \texttt{pfcDTMCSYS\_MOVE\_ROT\_X}, \texttt{pfcDTMCSYS\_MOVE\_ROT\_Y}, and \texttt{pfcDTMCSYS\_MOVE\_ROT\_Z} orientation constants.
  - \texttt{pfcDTMCSYS\_OFFSET\_CYLINDRICAL}—Specifies a cylindrical coordinate system that has been defined by setting the values for the \texttt{pfcDTMCSYS\_MOVE\_RAD}, \texttt{pfcDTMCSYS\_MOVE\_THETA}, and \texttt{pfcDTMCSYS\_MOVE\_TRAN\_ZI} orientation constants.
  - \texttt{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 coordinate system axes.
Element Trees: Sections

Overview........................................................................................................................................ 240
Creating Section Models .................................................................................................................. 240

A section is a parametric two-dimensional cross section used to define the shape of three-dimensional features, such as extrusions. In Creo Parametric, you create a section interactively using Sketcher mode. In an 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 Creo Parametric.

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 245.

Creating Section Models

A 2D section, because it is self-contained, can be stored as a Creo Parametric model file. It then has the extension .sec.

The steps required to create and save a section model using Object TOOLKIT C++ follow closely those used in creating a section interactively using Sketcher mode in Creo Parametric.

Setting the Intent Manager Mode of a Section

Methods Introduced:

• wfcSection::GetIntentManagerMode
• wfcSection::SetIntentManagerMode

The 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 Object TOOLKIT C++ methods that access the sections.
Example 1: Creating a Section Model

The sample code in OTKXCreateSection2D.cxx located at
<creo_toolkit_loadpoint>/otk_appls/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 245.

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 Parametric namelist and enables it to be recognized by Creo Parametric 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`

The methods `wfcSection::GetEpsilon` and `wfcSection::SetEpsilon` allow you to get and set the epsilon value for the section.

*Note*

*You cannot set the epsilon value 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`
Some classes in 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 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 all the entities in the specified section.

The method `wfcSectionEntity::GetSectionEntityTypeName` returns the section entity type using the enumerated type `wfcSection2dEntType`.

**Retrieving a Section**

Method Introduced:

- `wfcWFeature::GetSections`

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.*
Element Trees: Sketched Features

Overview ................................................................. 246
Creating Features Containing Sections .............................................. 246
Creating Features with 2D Sections .................................................. 247
Creating Features with 3D Sections .................................................. 247
Example 2: Manipulating a 3D Section .............................................. 248

This chapter describes the 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 Object TOOLKIT C++.
Overview

The chapter Feature Element Tree on page 203 explains how to create a simple feature using the feature element tree, and the documentation in the chapter Element Trees: Sections on page 239 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 203 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 239, a 2D section stored in a model file can be allocated by calling wfcWSession::CreateSection2D. Instead, Creo Parametric 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, Creo Parametric 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 Creo Parametric 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_toolkit_loadpoint>/otk_apps/otk_examples/otk_examples_feat` illustrates how to create a sweep using the Object TOOLKIT C++ methods.

**Creating Features with 2D Sections**

Sketched features using 2D sections do not require references to other geometry in the Creo Parametric 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 246.

**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**

Creo Parametric 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 Object TOOLKIT C++ application: position is chosen arbitrarily by Creo Parametric. This is because the interactive user of Creo Parametric never deals in absolute coordinates, and does not need to specify, or even know, the location of the origin of the section. In Object TOOLKIT C++ describe all section entities in terms of their coordinate values, so you need to find out where Creo Parametric 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_toolkit_loadpoint>/otk_appls/otk_examples/otk_examples_feat illustrates how to use all the methods described in this chapter to create a section model.
18

Holes

Accessing Threaded Hole Properties ................................................................. 250

This chapter describes how to access the hole properties in 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.*
Geometry Evaluation

Geometry Traversal ........................................................................................................ 252
Curves and Edges ........................................................................................................... 253
Contours .......................................................................................................................... 256
Surfaces .......................................................................................................................... 256
Axes, Coordinate Systems, and Points ........................................................................... 260
Interference ..................................................................................................................... 261

This chapter describes geometry representation and discusses how to evaluate
geometry using 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 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 197.

**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 443 for the parameterization of each curve type. To determine what type of curve a pfcEdge or pfcCurve object represents, use the Object TOOLKIT C++ instanceof operator.

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)

Evaluation of Curves and Edges

Methods Introduced:
• pfcGeomCurve::Eval3DData
• pfcGeomCurve::EvalFromLength
• pfcGeomCurve::EvalParameter
• pfcGeomCurve::EvalLength
• pfcGeomCurve::EvalLengthBetween

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.

**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.*

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.

**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`

**Note**

*To get geometric information for an edge, access the `pfcCurveDescriptor` object for one edge using `pfcGeomCurve::GetCurveDescriptor`.*
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.

## Contours

Methods Introduced:
- `pfcSurface::ListContours`
- `pfcContour::GetInternalTraversal`
- `pfcContour::FindContainingContour`
- `pfcContour::EvalArea`
- `pfcContour::EvalOutline`
- `pfcContour::VerifyUV`

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.

## Surfaces

Using Object TOOLKIT C++ you access datum and solid surfaces in the same way.
UV Parameterization

A surface in Creo Parametric 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 Parametric 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`.

• **QUILT**—A combination of two or more surfaces. This is represented by the class interface `pfcQuilt`.

  **Note**
  *This is used only for datum surfaces.*

• **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-consistent 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`

### 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`

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.

**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`

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.

**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 197` for additional information.

**Evaluation of ModelItems**

Methods Introduced:

- `pfcAxis::GetSurf`
- `pfcCoordSystem::GetCoordSys`
- `pfcPoint::GetPoint`

The method `pfcAxis::GetSurf` returns the revolved surface that uses the axis.

The method `pfcCoordSystem::GetCoordSys` returns the `pfcTransform3D` object (which includes the origin and x-, y-, and z- axes) that defines the coordinate system.

The method `pfcPoint::GetPoint` returns the xyz coordinates of the datum point.
Interference

Creo Parametric 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 pfcCreateGlobalEvaluator 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 not interfere.

`pfcSelectionEvaluator::ComputeClearance` computes the clearance data for the two selections. 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.
This chapter describes the 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 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 Object TOOLKIT C++ methods that access parameters. The topics are as follows:

- Creating and Accessing Parameters on page 265
- Parameter Selection Options on page 266
- Parameter Information on page 268
- Parameter Restrictions on page 270

Creating and Accessing Parameters

Methods Introduced:

- `pfcParameterOwner::CreateParam`
- `pfcParameterOwner::CreateParamWithUnits`
- `pfcParameterOwner::GetParam`
- `pfcParameterOwner::ListParams`
- `pfcParameterOwner::SelectParam`
In 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 266 for more information.

The method pfcFamColParam::GetRefParam returns the reference parameter from the parameter column in a family table.

### Parameter Selection Options

Parameter selection options in 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 `pfcParameterSelectionOptions::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::GetNamed
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 270 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.

### 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 Object TOOLKIT C++, but not modify the permitted values or range of values. Parameter restrictions in 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.
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 198 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`

All the `pfcBaseParameter` methods are accessible to Dimensions as well as Parameters. See the section Parameter Objects on page 265 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.

**Dimension Tolerances**

Methods Introduced:

- `pfcDimension::GetTolerance`
- `pfcDimension::SetTolerance`
- `pfcDimTolPlusMinus::Create`
- `pfcDimTolSymmetric::Create`
- `pfcDimTolLimits::Create`
- `pfcDimTolSymSuperscript::Create`
- `pfcDimTolISODIN::Create`

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 `instanceof`. 
This chapter describes how to access relations on all models and model items in Creo Parametric using the methods provided in Object TOOLKIT C++.
Accessing Relations

In 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 Parametric.

Methods Introduced:

- pfcRelationOwner::RegenerateRelations
- pfcRelationOwner::DeleteRelations
- pfcRelationOwner::GetRelations
- pfcRelationOwner::SetRelations
- pfcRelationOwner::EvaluateExpression

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 264 in the chapter Dimensions and Parameters on page 263 for more information on this object.

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 in Creo Parametric

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 277 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 279 for more information.

Note
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 Parametric 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 Parametric from evaluating the right hand side of the relation
Assemblies and Components

Structure of Assemblies and Assembly Objects .......................................................... 282
Assembling Components ................................................................................................ 288
Redefining and Rerouting Assembly Components ...................................................... 293
Exploded Assemblies .................................................................................................... 293
Skeleton Models ............................................................................................................. 294
Flexible Components .................................................................................................... 294
Flexible Component Variant Items ............................................................................... 295

This chapter describes the Object TOOLKIT C++ functions that access the functions of a Creo Parametric 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 Object TOOLKIT C++ functions 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 Object TOOLKIT C++ assembly functions.

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'
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 Parametric session.

Creating a Component Path

Methods Introduced

- `pfcCreateComponentPath`

The method `pfcAssembly::CreateComponentPath` 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` sets the 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::ListPoints
• wfcWComponentPath::ListCsyes
• wfcWComponentPath::ListCurves
• wfcWComponentPath::ListQuilts

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
• wfcWComponentFeat::RemoveConstraint
• wfcWAssembly::AutoInterchange
• wfcWAssembly::CreateAssemblyItem
• wfcWAssembly::GetConnectors
• wfcWAssembly::GetHarnesses
• wfcWAssembly::GetLinestocks
• wfcLineStock::GetName
• wfcLineStock::SetName
• wfcWAssembly::GetSpools
• wfcSpool::GetName
• wfcSpool::SetName

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.

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 but does not regenerate the assembly component. To regenerate the assembly use the method `pfcSolid::Regenerate`.

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 instance of assembly item that defines the flexible components using object `wfcAssemblyItemInstructions`. A flexible component allows variance of items such as features, dimensions, annotations, and parameters of a model in the context of an assembly.
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.

**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 Parametric 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.

• 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 functions enable you to reroute previously assembled components, just as in an interactive Creo Parametric session.

Methods Introduced:

- `pfcComponentFeat::RedefineThroughUI`
- `pfcComponentFeat::MoveThroughUI`

The method `pfcComponentFeat::RedefineThroughUI` must be used in interactive Object TOOLKIT C++ applications. This method displays the Creo Parametric **Constraint** dialog box. This enables the end user to redefine the constraints interactively. The control returns to 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 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.
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.

**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**

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.
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.

Flexible Component Variant Items

The methods in this section supports access to the variant properties of flexible components.

Methods Introduced:

- `wfcAssemblyItemInstructions::Create`
- `wfcAssemblyItemInstructions::GetItemOwner`
- `wfcAssemblyItemInstructions::SetItemOwner`
- `wfcAssemblyItemInstructions::GetItemCompPath`
- `wfcAssemblyItemInstructions::SetItemCompPath`
- `wfcAssemblyItemInstructions::GetItemId`
- `wfcAssemblyItemInstructions::SetItemId`
- `wfcAssemblyItemInstructions::GetItemType`
- `wfcAssemblyItemInstructions::SetItemType`
The method `wfcAssemblyItemInstructions::Create` creates a new instance of the object `wfcAssemblyItemInstruction` that contains the instructions to create a variant item for a flexible component.

The method `wfcAssemblyItemInstructions::GetItemOwner` gets the name of the model owner for the specified variant item.

The method `wfcAssemblyItemInstructions::SetItemCompPath` sets the component path for the variant item.

The methods `wfcAssemblyItemInstructions::GetItemId` and `wfcAssemblyItemInstructions::SetItemId` get and set the identifier for the variant item.

The methods `wfcAssemblyItemInstructions::GetItemType` and `wfcAssemblyItemInstructions::SetItemType` get and set the type of variant item using the enumerated type `pfcModelItemType`.
This chapter describes how to use Object TOOLKIT C++ classes and methods to access and manipulate family table information.
Working with Family Tables

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.
This chapter describes the Object TOOLKIT C++ methods that enable you to use action listeners.
Object TOOLKIT C++ Action Listeners

An ActionListener in Object TOOLKIT C++ is a class that is assigned to respond to certain events. In Object TOOLKIT C++, 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 Object TOOLKIT C++ are defined by these classes:

- Interface—Named <Object> ActionListener. This interface defines the methods that can respond to various events.
- Default class—Named Default <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.

Creating an ActionListener Implementation

You can create a proper ActionListener class using either of the following methods:

Define a separate class within the Object TOOLKIT C++ file.

To use your action listener in different Object TOOLKIT C++ applications, define it in a separate file.

Action Sources

Methods introduced:

- pfcActionSource::AddActionListener
- pfcActionSource::RemoveActionListener

Many 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 its subclasses)
  - Model Action Listener
  - Parameter Action Listener

- pfcSolid (and its subclasses)
  - Solid Action Listener
  - Feature Action Listener

- pfcFeature (and its 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.

Types of Action Listeners
The following sections describe the different kinds of action listeners: session, UI command, solid, and feature.

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

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.

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

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.
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.

### Feature Level Action Listeners

Methods introduced:

- `pfcFeatureActionListener::OnBeforeDelete`
- `pfcFeatureActionListener::OnBeforeSuppress`
- `pfcFeatureActionListener::OnAfterSuppress`
- `pfcFeatureActionListener::OnBeforeRegen`
- `pfcFeatureActionListener::OnAfterRegen`
- `pfcFeatureActionListener::OnRegenFailure`
- `pfcFeatureActionListener::OnBeforeRedefine`
- `pfcFeatureActionListener::OnAfterCopy`
- `pfcFeatureActionListener::OnBeforeParameterDelete`

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.

**Cancelling an ActionListener Operation**

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 Object TOOLKIT C++ exception signalling to Creo Parametric to cancel the listener event.

Note: Your application should not catch the 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`
25

Interface

Exporting Files and 2D Models ................................................................. 312
Exporting to PDF and U3D ...................................................................... 319
Exporting 3D Geometry ....................................................................... 326
Shrinkwrap Export ............................................................................... 329
Importing Files ..................................................................................... 335
Importing 3D Geometry ....................................................................... 338
Associative Topology Bus Enabled Models and Features ..................... 340
Printing Files ......................................................................................... 343
Automatic Printing of 3D Models ......................................................... 350
Solid Operations ................................................................................... 354
Window Operations ............................................................................... 356

This chapter describes various methods of importing and exporting files in
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 Parametric 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 Object TOOLKIT C++ APIWizard and online help for the Creo Parametric interface.

Export Instructions

Methods Introduced:

- **pfcRelationExportInstructions::Create**
- **pfcModelInfoExportInstructions::Create**
- **pfcProgramExportInstructions::Create**
- **pfcIGESFileExportInstructions::Create**

*Creo® Parametric 2.0 Object TOOLKIT C++ User’s Guide*
• 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

<table>
<thead>
<tr>
<th>Interface</th>
<th>Used to Export</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcRelationExportInstructions</td>
<td>A list of the relations and parameters in a part or assembly</td>
</tr>
<tr>
<td>pfcModelInfoExportInstructions</td>
<td>Information about a model, including units information, features, and children</td>
</tr>
<tr>
<td>pfcProgramExportInstructions</td>
<td>A program file for a part or assembly that can be edited to change the model</td>
</tr>
<tr>
<td>pfcIGESExportInstructions</td>
<td>A drawing in IGES format</td>
</tr>
<tr>
<td>pfcDXFExportInstructions</td>
<td>A drawing in DXF format</td>
</tr>
<tr>
<td>pfcRenderExportInstructions</td>
<td>A part or assembly in RENDER format</td>
</tr>
<tr>
<td>pfcSTLASCIIExportInstructions</td>
<td>A part or assembly to an ASCII STL file</td>
</tr>
<tr>
<td>pfcSTLBinaryExportInstructions</td>
<td>A part or assembly in a binary STL file</td>
</tr>
<tr>
<td>pfcBOMExportInstructions</td>
<td>A BOM for an assembly</td>
</tr>
<tr>
<td>Interface</td>
<td>Used to Export</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pfcDWGSetupExportInstructions</td>
<td>A drawing setup file</td>
</tr>
<tr>
<td>pfcFeatInfoExportInstructions</td>
<td>Information about one feature in a part or assembly</td>
</tr>
<tr>
<td>pfcMfgFeatCLExportInstructions</td>
<td>A cutter location (CL) file for one NC sequence in a manufacturing assembly</td>
</tr>
<tr>
<td>pfcMfgOperCLExportInstructions</td>
<td>A cutter location (CL) file for all the NC sequences in a manufacturing assembly</td>
</tr>
<tr>
<td>pfcMaterialExportInstructions</td>
<td>A material from a part</td>
</tr>
<tr>
<td>pfcCGMFILEExportInstructions</td>
<td>A drawing in CGM format</td>
</tr>
<tr>
<td>pfcInventorExportInstructions</td>
<td>A part or assembly in Inventor format</td>
</tr>
<tr>
<td>pfcFIATExportInstructions</td>
<td>A part or assembly in FIAT format</td>
</tr>
<tr>
<td>pfcConnectorParamExportInstructions</td>
<td>The parameters of a connector to a text file</td>
</tr>
<tr>
<td>pfcCableParamsFileInstructions</td>
<td>Cable parameters from an assembly</td>
</tr>
<tr>
<td>CATIAFacetsExportInstructions</td>
<td>A part or assembly in CATIA format (as a faceted model)</td>
</tr>
<tr>
<td>pfcVRMLModel1ExportInstructions</td>
<td>A part or assembly in VRML format</td>
</tr>
<tr>
<td>pfcSTEP2DExportInstructions</td>
<td>A two-dimensional STEP format file</td>
</tr>
<tr>
<td>pfcMedusaExportInstructions</td>
<td>A drawing in MEDUSA file</td>
</tr>
<tr>
<td>pfcCADDSExportInstructions</td>
<td>A CADDSS solid model</td>
</tr>
<tr>
<td>pfcNEUTRALFileExportInstructions</td>
<td>A Creo Parametric part to neutral format</td>
</tr>
<tr>
<td>pfcProductViewExportInstructions</td>
<td>A part, assembly, or drawing in Creo View format</td>
</tr>
<tr>
<td>pfcSliceExportData</td>
<td>A slice export format</td>
</tr>
</tbody>
</table>

Exporting Drawing Sheets

The options required to export multiple sheets of a drawing are given by the `pfcExport2DOption` object.

Methods Introduced:

- `pfcExport2DOption::Create`
- `pfcExport2DOption::SetExportSheetOption`
- `pfcExport2DOption::SetModelSpaceSheet`
- `pfcExport2DOption::SetSheets`

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.
<ul>
  <li>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.</li>
  <li>pfcEXPORT_ALL—Exports all the sheets in a drawing to a single file as paper space, if applicable for the format type.</li>
  <li>pfcEXPORT_SELECTED—Exports selected sheets in a drawing as paper space and one sheet as model space.</li>
</ul>

- 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 Parametric 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 Parametric user interface are used during the export operation.
- **pfcFACET_CHORD_HEIGHT_DEFAULT**—Uses the default value set in the Creo Parametric user interface for the chord height.
- **pfcFACET_ANGLE_CONTROL_DEFAULT**—Uses the default value set in the Creo Parametric user interface for the angle control.
- **pfcFACET_STEP_SIZE_DEFAULT**—Uses the default value set in the Creo Parametric 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.
- **pfcFacetControlFlag_nil**—Nil option.
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`

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
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 **NULL**, if you specify the Quality value.

The methods

```
pfcCoordSysExportInstructions::GetAngleControl
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 **NULL**, if you specify the Quality value.

The methods

```
pfcCoordSysExportInstructions::GetSliceExportData
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
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 **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 315**

**Exporting to PDF and U3D**

The methods described in this section support the export of Creo Parametric 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 Parametric 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 Parametric 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 export Creo Parametric 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, it 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 Parametric 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 Parametric 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 Parametric 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 pfcPDF_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
Object TOOLKIT C++ allows you to export three dimensional geometry to various formats. Pass the instructions object containing information about the desired export file to the method pfcModel::Export.

Export Instructions
Methods Introduced:
• pfcExport3DInstructions::GetConfiguration
• pfcExport3DInstructions::SetConfiguration
• pfcExport3DInstructions::GetReferenceSystem
• pfcExport3DInstructions::SetReferenceSystem
• pfcExport3DInstructions::GetGeometry
• pfcExport3DInstructions::SetGeometry
• pfcExport3DInstructions::GetIncludedEntities
• pfcExport3DInstructions::SetIncludedEntities
• pfcExport3DInstructions::GetLayerOptions
• pfcExport3DInstructions::SetLayerOptions
• pfcGeometryFlags::Create
• pfcInclusionFlags::Create
• pfcLayerExportOptions::Create
• pfcSTEP3DExportInstructions::Create
• pfcVDA3DExportInstructions::Create
• pfcIGES3DNewExportInstructions::Create
• pfcACIS3DExportInstructions::Create
• pfcCATIAModel3DExportInstructions::Create
• pfcCatiaPart3DExportInstructions::Create
• pfcCatiaProduct3DExportInstructions::Create
• pfcCatiaCGR3DExportInstructions::Create
• pfcDXF3DExportInstructions::Create
• pfcDWG3DExportInstructions::Create
• pfcJT3DExportInstructions::Create
• pfcParaSolid3DExportInstructions::Create
• pfcUG3DExportInstructions::Create
• pfcTriangulationInstructions::Create

The interface pfcExport3DInstructions contains data to export a part or an assembly to a specified 3D format. The fields of this interface are:

• AssemblyConfiguration—While exporting an assembly you can specify the structure and contents of the output files. The options are:
  ○ pfcIMPORT_ASM_FLAT_FILE—Exports all the geometry of the assembly to a single file as if it were a part.
  ○ pfcIMPORT_ASM_SINGLE_FILE—Exports an assembly structure to a file with external references to component files. This file contains only top-level geometry.
  ○ pfcIMPORT_ASM_MULTI_FILE—Exports an assembly structure to a single file and the components to component files. It creates component parts and subassemblies with their respective geometry and external references. This option supports all levels of hierarchy.
  ○ pfcIMPORT_ASM_ASSEMBLY_FILE—Exports an assembly as multiple files containing geometry information of its components and assembly features.

• CoordSystem—The reference coordinate system used for export. If this value is null, the system uses the default coordinate system.

• GeometryFlags—The object describing the type of geometry to export. The pfcGeometryFlags::Create returns this instruction object. The types of geometry supported by the export operation are:
  ○ Wireframe—Export edges only.
  ○ Solid—Export surfaces along with topology.
  ○ Surfaces—Export all model surfaces.
  ○ Quilts—Export as quilt.

• InclusionFlags—The object returned by the method pfcInclusionFlags::Create that determines whether to include certain entities. The entities are:
Datums—Determines whether datum curves are included when exporting files. If true the datum curve information is included during export. The default value is false.

Blanked—Determines whether entities on blanked layers are exported. If true entities on blanked layers are exported. The default value is false.

LayerExportOptions—The instructions object returned by the method pfcLayerExportOptions::Create that describes how to export layers. To export layers you can specify the following:

UseAutoId—Enables you to set or remove an interface layer ID. A layer is recognized with this ID when exporting the file to a specified output format. If true, automatically assigns interface IDs to layers not assigned IDs and exports them. The default value is false.

LayerSetupFile—Specifies the name and complete path of the layer setup file. This file contains the layer assignment information which includes the name of the layer, its display status, the interface ID and number of sub layers.

The method pfcTriangulationInstructions::Create creates a object that will be used to define the parameters for faceted exports.

Export 3D Instructions Table

<table>
<thead>
<tr>
<th>Interface</th>
<th>Used to Export</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcSTEP3DExportInstructions</td>
<td>A part or assembly in STEP format</td>
</tr>
<tr>
<td>pfcVDA3DExportInstructions</td>
<td>A part or assembly in VDA format</td>
</tr>
<tr>
<td>pfcIGES3DNewExportInstructions</td>
<td>A part or assembly in IGES format</td>
</tr>
<tr>
<td>pfcCATIAModel3DExportInstructions</td>
<td>A part or assembly in CATIA MODEL format</td>
</tr>
<tr>
<td>pfcACTIS3DExportInstructions</td>
<td>A part or assembly in ACIS format</td>
</tr>
<tr>
<td>pfcCatiaPart3DExportInstructions</td>
<td>A part or assembly in CATIA PART format</td>
</tr>
<tr>
<td>pfcCatiaProduct3DExportInstructions</td>
<td>A part or assembly in CATIA PRODUCT format</td>
</tr>
<tr>
<td>pfcCatiaCGR3DExportInstructions</td>
<td>A part or assembly in CATIA CGR format</td>
</tr>
<tr>
<td>pfcJT3DExportInstructions</td>
<td>A part or assembly in JT format</td>
</tr>
<tr>
<td>pfcParaSolid3DExportInstructions</td>
<td>A part or assembly in PARASOLID format</td>
</tr>
<tr>
<td>pfcUG3DExportInstructions</td>
<td>A part or assembly in UG format</td>
</tr>
<tr>
<td>pfcDWG3DExportInstructions</td>
<td>A part or assembly in DWG format</td>
</tr>
<tr>
<td>pfcDXF3DExportInstructions</td>
<td>A part or assembly in DXF format</td>
</tr>
<tr>
<td>pfcTriangulationInstructions</td>
<td>A part or assembly in faceted format</td>
</tr>
</tbody>
</table>

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 assembly 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`.

<table>
<thead>
<tr>
<th>Type of Shrinkwrap Model</th>
<th>Interface to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Subset</td>
<td><code>pfcShrinkwrapSurfaceSubsetInstructions</code></td>
</tr>
<tr>
<td>Faceted Part</td>
<td><code>pfcShrinkwrapFacetedPartInstructions</code></td>
</tr>
<tr>
<td>Faceted VRML</td>
<td><code>pfcShrinkwrapFacetedVRMLInstructions</code></td>
</tr>
<tr>
<td>Faceted STL</td>
<td><code>pfcShrinkwrapFacetedSTLInstructions</code></td>
</tr>
<tr>
<td>Merged Solid</td>
<td><code>pfcShrinkwrapMergedSolidInstructions</code></td>
</tr>
</tbody>
</table>

**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 Parametric 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 `pfcShrinkwrapModelExportInstructions::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` assigns 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 Parametric 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::GetSmallSurfPercentage` 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 Parametric 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 Parametric part.

The method `pfcShrinkwrapFacetedPartInstructions::GetLightweight` returns a true value if the output file format of the shrinkwrap model is a Lightweight Creo Parametric part. The method `pfcShrinkwrapFacetedPartInstructions::SetLightweight` specifies if the Creo Parametric part is exported as a lightweight 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 Parametric. The format must be the same as it would be if these files were created by Creo Parametric. 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 Parametric. The details are as shown in the table below:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Used to Import</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pfcRelationImportInstructions</code></td>
<td>A list of relations and parameters in a part or assembly.</td>
</tr>
<tr>
<td><code>pfcIGESSectionImportInstructions</code></td>
<td>A section model in IGES format.</td>
</tr>
<tr>
<td><code>pfcProgramImportInstructions</code></td>
<td>A program file for a part or assembly that can be edited to change the model.</td>
</tr>
<tr>
<td><code>pfcConfigImportInstructions</code></td>
<td>Configuration instructions.</td>
</tr>
<tr>
<td><code>pfcDWGSetupImportInstructions</code></td>
<td>A drawing s/u file.</td>
</tr>
<tr>
<td><code>pfcSpoolImportInstructions</code></td>
<td>Spool instructions.</td>
</tr>
<tr>
<td><code>pfcConnectorParamsImportInstructions</code></td>
<td>Connector parameter instructions.</td>
</tr>
<tr>
<td><code>pfcASSEMTreeCFGImportInstructions</code></td>
<td>Assembly tree CFG instructions.</td>
</tr>
<tr>
<td><code>pfcWireListImportInstructions</code></td>
<td>Wirelist instructions.</td>
</tr>
<tr>
<td><code>pfcCableParamsImportInstructions</code></td>
<td>Cable parameters from an assembly.</td>
</tr>
<tr>
<td><code>pfcSTEPImport2DInstructions</code></td>
<td>A part or assembly in STEP format.</td>
</tr>
<tr>
<td><code>pfcIGESImport2DInstructions</code></td>
<td>A part or assembly in IGES format.</td>
</tr>
<tr>
<td><code>pfcDXFImport2DInstructions</code></td>
<td>A drawing in DXF format.</td>
</tr>
<tr>
<td><code>pfcDWGImport2DInstructions</code></td>
<td>A drawing in DWG format.</td>
</tr>
</tbody>
</table>
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 90.

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 pfcImport2DInstructions 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
- pfcNewModelImportType_nil

- 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 Parametric 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 pfcXCancelProEAAction::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 
\texttt{pfcImportedLayer::GetName} returns the name of the layer while the 
method \texttt{pfcImportedLayer::SetNewName} can be used to set the name of 
the layer. The name can be numeric or alphanumeric.

The method \texttt{pfcImportedLayer::GetSurfaceCount} returns the number 
of curves on the layer.

The method \texttt{pfcImportedLayer::GetTrimmedSurfaceCount} 
returns the number of trimmed surfaces on the layer and the method 
\texttt{pfcImportedLayer::GetCurveCount} returns the number of curves on 
the layer.

The method \texttt{pfcImportedLayer::SetAction} sets the display of the 
imported layers. The input parameter for this method is \texttt{ImportAction}. The types 
of actions that can be performed on the imported layers are:

- \texttt{pfcIMPORT\_LAYER\_DISPLAY}—Displays the imported layer.
- \texttt{pfcIMPORT\_LAYER\_SKIP}—Does not import entities on this layer.
- \texttt{pfcIMPORT\_LAYER\_BLANK}—Blanks the selected layer.
- \texttt{pfcIMPORT\_LAYER\_IGNORE}—Imports only entities on this layer but not 
  the layer.

The default action type is \texttt{pfcIMPORT\_LAYER\_DISPLAY}.

**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 Parametric 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 Parametric assembly containing one or more components which are 
  models imported from an ATB interface, such as, CADDS or CATIA.
- A Creo Parametric part containing an Import feature that is imported from an 
  ATB interface such as, ICEM.

Only import operations in Creo Parametric 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 Parametric 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 existing Creo Parametric part models. The method `wfcWModel::VerifyATB` returns the `wfcATBVerificationResults` object that represents the status of the TIMs.

You can specify a Creo Parametric 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 Parametric assembly containing one or more components which are models imported from an ATB Interface, such as, CADDs or CATIA.
- A Creo Parametric part containing an Import feature that is imported from an ATB interface such as, ICEM.

The input arguments 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 Parametric 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 Parametric 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 Parametric 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 Parametric 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.

• **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 to 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 method `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

Object TOOLKIT C++ provides the capability of automatically creating and plotting a drawing of a solid model. The Object TOOLKIT C++ application needs only to supply instructions for the print activity, and Creo Parametric 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:
The method `wfcQuickPrintGeneralViewInstructions::Create` creates a new instance of the `wfcQuickPrintGeneralViewInstructions` object that contains the quick drawing print instructions for general view.

The methods
`wfcQuickPrintGeneralViewInstructions::GetGeneralViewLocation` and
`wfcQuickPrintGeneralViewInstructions::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
- wfcQuickPrintGeneralViewLocation-nil

**Note**

*The general view location options are analogous to the images in the Quick Drawing dialog box under View Layout.*

The methods wfcQuickPrintGeneralViewInstructions::GetScale and wfcQuickPrintGeneralViewInstructions::SetScale get and set the view scale.

The methods wfcQuickPrintGeneralViewInstructions::GetViewDisplayStyle and wfcQuickPrintGeneralViewInstructions::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 115.

The methods wfcQuickPrintGeneralViewInstructions::GetViewName and wfcQuickPrintGeneralViewInstructions::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
wfcQuickPrintGeneralViewInstructions.

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.
- wfcQuickPrintLayoutType_nil— NULL value.

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
- wfcQuickPrintManual3View_nil

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
- wfcQuickPrintOrientation_nil
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 344.

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 Parametric 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`
- `wfcQuickPrintProjectionViewLocation_nil`

### 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`

• **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 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 enumerated data type `pfcIntfType`.
- **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 Parametric raster output file.
Object TOOLKIT C++ gives programmatic access to all the simplified representation functionality of Creo Parametric. Create simplified representations either permanently or on the fly and save, retrieve, or modify them by adding or deleting items.
Overview

Using Object TOOLKIT C++, you can create and manipulate assembly simplified representations just as you can using Creo Parametric interactively.

Note

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 its 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.

pfcSimpActionType is an enumerated type that specifies the possible treatment of items in a simplified representation. The possible values are as follows

<table>
<thead>
<tr>
<th>Values</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcSIMPREP_NONE</td>
<td>No action is specified.</td>
</tr>
<tr>
<td>pfcSIMPREP_REVERSE</td>
<td>Reverse the default rule for this component (for example, include it if the default rule is exclude).</td>
</tr>
<tr>
<td>pfcSIMPREP_INCLUDE</td>
<td>Include this component in the simplified representation.</td>
</tr>
<tr>
<td>pfcSIMPREP_EXCLUDE</td>
<td>Exclude this component from the simplified representation.</td>
</tr>
<tr>
<td>pfcSIMPREP_SUBSTITUTE</td>
<td>Substitute the component in the simplified representation.</td>
</tr>
<tr>
<td>pfcSIMPREP_GEOM</td>
<td>Use only the geometrical representation of the component.</td>
</tr>
<tr>
<td>pfcSIMPREP_GRAPHICS</td>
<td>Use only the graphics representation of the component.</td>
</tr>
<tr>
<td>pfcSIMPREP_SYMB</td>
<td>Use the symbolic representation of the component.</td>
</tr>
</tbody>
</table>
Retrieving Simplified Representations

Methods Introduced:

- `pfcBaseSession::RetrieveAssemSimpRep`
- `pfcBaseSession::RetrieveGeomSimpRep`
- `pfcBaseSession::RetrieveGraphicsSimpRep`
- `pfcBaseSession::RetrieveSymbolicSimpRep`
- `pfcRetrieveExistingSimpRepInstructions::Create`

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 Parametric 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.

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`

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.

### Modifying Simplified Representations

Methods Introduced:

- `pfcSimpRep::GetInstructions`
- `pfcSimpRep::SetInstructions`
- `pfcSimpRepInstructions::SetDefaultAction`
• pfcCreateNewSimpRepInstructions::SetNewSimpName
• pfcSimpRepInstructions::SetIsTemporary

Using 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.

### 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

You can add and delete items from the list of components in a simplified representation using 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.
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

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

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 Parametric 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 `pfcModelDisplay`.
Asynchronous Mode

Overview........................................................................................................... 366
Simple Asynchronous Mode ................................................................................ 366
Starting and Stopping Creo Parametric ............................................................. 367
Connecting to a Creo Parametric Process .......................................................... 368
Full Asynchronous Mode ...................................................................................... 370
Troubleshooting Asynchronous Object TOOLKIT ............................................ 372

This chapter explains how to use Object TOOLKIT C++ in Asynchronous Mode.
Overview

Asynchronous mode is a multiprocessing mode in which the 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 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 (containing its own main() function) is started independently of Creo Parametric and subsequently either starts or connects to a Creo Parametric process.

**Note**

*An asynchronous application that starts Creo Parametric will not appear in the Auxiliary Applications dialog box.*

The section How Object TOOLKIT C++ Works on page 14 in Overview of Object TOOLKIT C++ on page 13 chapter describes two modes—DLL and multiprocessing (or “spawned”). These modes are synchronous modes in the sense that the 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 (multiprocessing). 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 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, 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 an Object TOOLKIT 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 Object TOOLKIT methods but does not support callbacks. These considerations imply that the Object TOOLKIT 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 Object TOOLKIT 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 Object TOOLKIT 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:

```cpp
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.

\texttt{pfcAsyncConnection::GetActiveConnection} returns the current connection to a Creo Parametric session.

To disconnect from a Creo Parametric process, call the method \texttt{pfcAsyncConnection::Disconnect}. This method can be called only if you used the method \texttt{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:

- \texttt{pfcAsyncConnection::GetConnectionId}
- \texttt{pfcConnectionId::GetExternalRep}
- \texttt{pfcBaseSession::GetConnectionId}
- \texttt{pfcConnectionId::Create}
- \texttt{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 \texttt{pfcAsyncConnection::GetConnectionId} returns a data structure containing the connection ID.

If the connection id must be passed to some other application the method \texttt{pfcConnectionId::GetExternalRep} provides the string external representation for the connection ID.

The method \texttt{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 47 chapter for additional information.

### Full Asynchronous Mode

Full asynchronous mode is identical to the simple asynchronous mode except in the way the Object TOOLKIT 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:
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 47 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 an Object TOOLKIT 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 303 chapter.
Troubleshooting Asynchronous Object TOOLKIT

General Problems

pfcX ToolkitNotFound 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.

pfcX ToolkitGeneralError or pfcX ToolkitCommError 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.
Applications created using different Creo Parametric API products are interoperable. These products use Creo Parametric as the medium of interaction, eliminating the task of writing native-platform specific interactions between different programming languages.

Application interoperability allows Object TOOLKIT C++ applications to call into Creo Parametric TOOLKIT from areas not covered in the native interface. It allows you to put a Object TOOLKIT C++ front end on legacy Creo Parametric TOOLKIT applications and also allows you to use Object TOOLKIT C++ applications and listeners in conjunction with a asynchronous Object TOOLKIT C++ application.

Object TOOLKIT C++ can call Creo Parametric web pages belonging to Web.Link, and functions in Creo Parametric TOOLKIT DLLs. Object TOOLKIT C++ synchronous applications can also register tasks for use by other applications.
Managing Application Arguments

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 Parametric session)
- pfcTransform3D (a coordinate system transformation matrix)

Methods Introduced:

- pfcCreateIntArgValue
- pfcCreateDoubleArgValue
- pfcCreateBoolArgValue
- pfcCreateASCIIStringArgValue
- pfcCreateStringArgValue
- pfcCreateSelectionArgValue
- pfcCreateTransformArgValue
- pfcArgValue::Getdiscer
- 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. 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 Parametric TOOLKIT DLL**

The methods described in this section enable a Object TOOLKIT C++ user to register and launch a Creo Parametric TOOLKIT DLL from a Object TOOLKIT C++ application.

Methods Introduced:
Use the method `pfcBaseSession::LoadProToolkitDll` to register and start a Creo Parametric 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 Parametric 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 Parametric 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 Parametric 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 Parametric TOOLKIT DLL library. The input parameters of this method are:

- **FunctionName**—Name of the function in the Creo Parametric 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 Parametric 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 Parametric TOOLKIT DLL previously loaded by the method `pfcBaseSession::LoadProToolkitDll` is still active.

The method `pfcDll::Unload` is used to shutdown a Creo Parametric TOOLKIT DLL previously loaded by the method `pfcBaseSession::LoadProToolkitDll` and the application's `user_terminate()` function is called.

**Launching Tasks from Object TOOLKIT C++ Task Libraries**

The methods described in this section allow you to launch tasks from a predefined Object TOOLKIT C++ task library.

Methods Introduced:

- `pfcBaseSession::StartJLinkApplication`
- `pfcJLinkApplication::ExecuteTask`
- `pfcJLinkApplication::IsActive`
- `pfcJLinkApplication::Stop`

Use the method `pfcBaseSession::StartJLinkApplication` to start a 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 Object TOOLKIT C++ application.
- **ClassName**—Specifies the name of the Object TOOLKIT C++ class that contains the Object TOOLKIT C++ application’s start and stop method. This should be a fully qualified Object TOOLKIT C++ package and class name.
• **StartMethod**—Specifies the start method of the Object TOOLKIT C++ application.

• **StopMethod**—Specifies the stop method of the Object TOOLKIT C++ application.

• **AdditionalClassPath**—Specifies the locations of packages and classes that must be loaded when starting this 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 Creo Parametric.

Upon starting the application, the static `start()` method is invoked. The method returns a `pfcJLinkApplication` referring to the Object TOOLKIT C++ application.

The method `pfcJLinkApplication::ExecuteTask` calls a registered task method in a 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.
Overview ................................................................. 380
Getting Mouse Input .................................................. 380
Displaying Graphics .................................................... 381
Display Lists and Graphics .......................................... 383

This chapter covers Object TOOLKIT C++ graphics including displaying lists, displaying text and using the mouse.
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.
Displaying Graphics

All the methods in this section draw graphics in the Creo Parametric 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 Parametric 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 Parametric display list. Thus, they do not get redrawn by Creo Parametric when the user selects View, Repaint or View, Orientation. However, if you store graphic elements in either 2-D or 3-D display lists, Creo Parametric will redraw them when appropriate. See the section on Display Lists and Graphics on page 383 for more information.

Methods Introduced:

- `pfcDisplay::SetPenPosition`
- `pfcDisplay::DrawLine`
- `pfcDisplay::DrawPolyline`
- `pfcDisplay::DrawCircle`
- `pfcDisplay::DrawArc2D`
- `pfcDisplay::DrawPolygon2D`

The method `pfcDisplay::SetPenPosition` sets the point at which you want to start drawing a line. The function `pfcDisplay::DrawLine` draws a line to the given point from the position given in the last call to either of the two functions. 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.

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 Creo Parametric standard color used to display graphics. The Creo Parametric 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 Parametric draws graphics in the required color in each invocation.
- `pfcDRAW_GRAPHICS_COMPLEMENT`— Creo Parametric 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 Parametric and therefore are not redrawn when you select View, Repaint. To notify the Creo Parametric of these objects, create them inside the `OnDisplay()` method of the Display Listener.

Controlling Text Attributes

Methods Introduced:
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 Parametric text font. The text fonts are identified in Creo Parametric 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 Parametric 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 Parametric 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 Parametric. 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 Parametric, you call `pfcDisplay::CreateDisplayList2D` or `pfcDisplay::CreateDisplayList3D` to tell Creo Parametric to use your listener to create the display list vectors.

`pfcDisplayList2D::Display` or `pfcDisplayList3D::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:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>XToolkitNotExist</td>
<td>The display list is empty.</td>
</tr>
<tr>
<td>XToolkitNotFound</td>
<td>The function could not find the display list or the font specified in a previous call to pfcDisplay::SetCurrentFont was not found.</td>
</tr>
<tr>
<td>XToolkitCantOpen</td>
<td>The use of display lists is disabled.</td>
</tr>
<tr>
<td>XToolkitAbort</td>
<td>The display was aborted.</td>
</tr>
<tr>
<td>XToolkitNotValid</td>
<td>The specified display list is invalid.</td>
</tr>
<tr>
<td>XToolkitInvalidItem</td>
<td>There is an invalid item in the display list.</td>
</tr>
<tr>
<td>XToolkitGeneralError</td>
<td>The specified display list is already in the process of being displayed.</td>
</tr>
</tbody>
</table>

Graphics
This chapter explains using External Data in Object TOOLKIT C++.
External Data

This chapter describes how to store and retrieve external data. External data enables a Object TOOLKIT C++ application to store its own data in a Creo Parametric database in such a way that it is invisible to the Creo Parametric user. This method is different from other means of storage accessible through the Creo Parametric user interface.

Introduction to External Data

External data provides a way for the Creo Parametric application to store its own private information about a Creo Parametric 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 Parametric; the application has complete control over form and content.

The external data for a specific Creo Parametric 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 Parametric 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:

<table>
<thead>
<tr>
<th>Object TOOLKIT C++ Type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcEXTDATA_INTEGER</td>
<td>integer</td>
</tr>
<tr>
<td>pfcEXTDATA_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>pfcEXTDATA_STRING</td>
<td>string</td>
</tr>
</tbody>
</table>

The Object TOOLKIT C++ interfaces used to access external data in Creo Parametric are:

<table>
<thead>
<tr>
<th>Object TOOLKIT C++ Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcExternalDataAccess</td>
<td>This is the top level object and is created when attempting to access external data.</td>
</tr>
<tr>
<td>pfcExternalDataClass</td>
<td>This is a class of external data and is identified by a unique name.</td>
</tr>
<tr>
<td>pfcExternalDataSlot</td>
<td>This is a container for one item of data. Each slot is stored in a class.</td>
</tr>
<tr>
<td>pfcExternalData</td>
<td>This is a compact data structure that contains either an integer, double or string value.</td>
</tr>
</tbody>
</table>
Compatibility with Creo Parametric TOOLKIT

Object TOOLKIT C++ and Creo Parametric TOOLKIT share external data in the same manner. Object TOOLKIT C++ external data is accessible by Creo Parametric TOOLKIT and the reverse is also true. However, an error will result if Object TOOLKIT C++ attempts to access external data previously stored by Creo Parametric TOOLKIT as a stream.

Accessing External Data

Methods Introduced:

- `pfcModel::AccessExternalData`
- `pfcModel::TerminateExternalData`
- `pfcExternalDataAccess::IsValid`

The method `pfcModel::AccessExternalData` prepares Creo Parametric 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 Parametric 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 Parametric 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`

**Exceptions**

Most exceptions thrown by external data methods in 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 Parametric TOOLKIT.

The following table lists these exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pfcXExternalDataInvalidObject</code></td>
<td>Generated when a model or class is invalid.</td>
</tr>
<tr>
<td><code>pfcXExternalDataClassOrSlotExists</code></td>
<td>Generated when creating a class or slot and the proposed class or slot already exists.</td>
</tr>
<tr>
<td><code>pfcXExternalDataNamesTooLong</code></td>
<td>Generated when a class or slot name is too long.</td>
</tr>
<tr>
<td><code>pfcXExternalDataSlotNotFound</code></td>
<td>Generated when a specified class or slot does not exist.</td>
</tr>
<tr>
<td><code>pfcXExternalDataEmptySlot</code></td>
<td>Generated when the slot you are attempting to read is empty.</td>
</tr>
<tr>
<td><code>pfcXExternalDataInvalidSlotName</code></td>
<td>Generated when a specified slot name is invalid.</td>
</tr>
<tr>
<td><code>pfcXBadGetExternalData</code></td>
<td>Generated when you try to access an incorrect data type in a <code>External.ExternalData</code> object.</td>
</tr>
</tbody>
</table>
Windchill Connectivity APIs

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>394</td>
</tr>
<tr>
<td>Accessing a Windchill Server from a Creo Parametric Session</td>
<td>394</td>
</tr>
<tr>
<td>Accessing Workspaces</td>
<td>397</td>
</tr>
<tr>
<td>Workflow to Register a Server</td>
<td>399</td>
</tr>
<tr>
<td>Aliased URL</td>
<td>400</td>
</tr>
<tr>
<td>Server Operations</td>
<td>401</td>
</tr>
<tr>
<td>Utility APIs</td>
<td>411</td>
</tr>
</tbody>
</table>

Creo Parametric has the capability to be directly connected to Windchill solutions, including Windchill ProjectLink and PDMLink servers. This access allows users to manage and control the product data seamlessly from within Creo Parametric.

This chapter lists Object TOOLKIT C++ APIs that support Windchill servers and server operations in a connected Creo Parametric session.
Introduction

The methods introduced in this chapter provide support for the basic Windchill server operations from within Creo Parametric. With these methods, operations such as registering a Windchill server, managing workspaces, and check in or check out of objects will be possible via Object TOOLKIT C++. The capabilities of these APIs are similar to the operations available from within the Creo Parametric client, with some restrictions.

Non-Interactive Mode Operations

Some of the APIs specified in this section operate only in batch mode and cannot be used in the normal Creo Parametric interactive mode. This restriction is mainly centered around the Object TOOLKIT C++ registered servers, that is, servers registered by Object TOOLKIT C++ are not available in the Creo Parametric Server Manager dialog box or in other locations in the Creo Parametric user interface such as the Folder Navigator and embedded browser. If a Object TOOLKIT C++ customization requires the user to have interactive access to the server, the server must be registered via the normal Creo Parametric techniques, that is, either by entry in the Server Manager dialog box or by automatic registration of a previously registered server.

All of these APIs are supported from a non-interactive, that is, batch mode application or asynchronous application.

Accessing a Windchill Server from a Creo Parametric Session

Creo Parametric allows you to register Windchill servers as a connection between the Windchill database and Creo Parametric. Although the represented Windchill database can be from Windchill ProjectLink or Windchill PDMLink, all types of databases are represented in the same way.

You can use the following identifiers when referring to Windchill servers in Object TOOLKIT C++:

- **Codebase URL**—This is the root portion of the URL that is used to connect to a Windchill server. For example http://wcservr.company.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 Creo Parametric. 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 Parametric 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**

The methods described in this section are restricted to the non-interactive mode only. Refer to the section, *Non-Interactive Mode Operations* on page 394, for more information.

Methods Introduced:

- *pfcBaseSession::RegisterServer*
- *pfcServer::Activate*
- *pfcServer::Unregister*

The method *pfcBaseSession::RegisterServer* registers the specified server with the codebase URL. 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 Parametric session.

The method *pfcServer::Unregister* unregisters the specified server. This is similar to *Server Manager ▶ Delete* through the user interface.

**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

All methods described in this section, except pfcServer::GetActiveWorkspace, are permitted only in the non-interactive mode. Refer to the section, Non-Interactive Mode Operations on page 394, for more information.

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 Parametric versions, for example, .1 or .2, and so on.

Server Operations

After registering the Windchill server with Creo Parametric, you can start accessing the data on the Windchill servers. The Creo Parametric interaction with Windchill servers leverages the following locations:

- Commonspace (Shared folders)
- Workspace (Server-side workspace)
- Workspace local cache (Client-side workspace)
- Creo Parametric 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 Parametric 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 Parametric 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 Parametric 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 Auto Check-In option in Creo Parametric. For more details on Auto Check-In, refer to the online help for Creo Parametric.

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 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:

- \texttt{pfcBaseSession::RetrieveModel}
- \texttt{pfcBaseSession::RetrieveModelWithOpts}
- \texttt{pfcBaseSession::OpenFile}

The methods \texttt{pfcBaseSession::RetrieveModel}, \texttt{pfcBaseSession::RetrieveModelWithOpts}, and \texttt{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 \texttt{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 Object TOOLKIT C++, both operations are covered by the same set of methods.

Methods Introduced:

- \texttt{pfcServer::CheckoutObjects}
- \texttt{pfcServer::CheckoutMultipleObjects}
- \texttt{pfcCheckoutOptions.Create}
- \texttt{pfcCheckoutOptions::SetDependency}
- \texttt{pfcCheckoutOptions::SetSelectedIncludes}
- \texttt{pfcCheckoutOptions::SetIncludeInstances}
- \texttt{pfcCheckoutOptions::SetVersion}
- \texttt{pfcCheckoutOptions::SetDownload}
- \texttt{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 Parametric 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.

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.

<table>
<thead>
<tr>
<th>Argument checkout in <code>pfcServer::CheckoutObjects</code></th>
<th><code>pfcCheckoutOptions::SetDownload</code></th>
<th><code>pfcCheckoutOptions::SetReadonly</code></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>NA</td>
<td>Object is checked out and its content is downloaded.</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>NA</td>
<td>Object is checked out but content is not downloaded.</td>
</tr>
<tr>
<td>false</td>
<td>NA</td>
<td>true</td>
<td>Object is downloaded without checkout.</td>
</tr>
<tr>
<td>false</td>
<td>NA</td>
<td>false</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

**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 Parametric session.
Import and Export

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 Parametric.

The method `pfcBaseSession::ImportToCurrentWS` imports the specified objects from a disk to the current workspace in a linked session of Creo Parametric.

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 Parametric model files. Use the method pfcWSExportOptions::SetIncludeSecondaryContent to set this option.

File Copy
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 Parametric 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 Parametric 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 Parametric 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 Parametric designated parameters, dependencies, and family table information to a PDM system while `pfcBaseSession::ExportFromCurrentWS` can update exported Creo Parametric 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 Parametric files.

**Server Object Status**

Methods Introduced:

- `pfcServer::IsObjectCheckedOut`
- `pfcServer::IsObjectModified`

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 status:

- 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 since checkout. This method returns the value `false` if newly created objects have not been uploaded.
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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkin an object or workspace</td>
<td><code>pfcServer::CheckinObjects</code></td>
</tr>
<tr>
<td>Checkout an object</td>
<td><code>pfcServer::CheckoutObjects</code></td>
</tr>
<tr>
<td>Undo checkout of an object</td>
<td><code>pfcServer::UndoCheckout</code></td>
</tr>
<tr>
<td>Upload object</td>
<td><code>pfcServer::UploadObjects</code></td>
</tr>
<tr>
<td>Download object</td>
<td><code>pfcServer::CheckoutObjects (with download as true)</code></td>
</tr>
<tr>
<td>Delete workspace</td>
<td><code>pfcServerLocation::DeleteWorkspace</code></td>
</tr>
<tr>
<td>Remove object</td>
<td><code>pfcServer::RemoveObjects</code></td>
</tr>
</tbody>
</table>

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 Parametric 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 400. 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 Parametric 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 http://server.my-company.com/Windchill.

The method pfcBaseSession::GetAliasFromAliasedUrl returns the server alias from aliased URL.
Summary of Technical Changes

Critical Technical Changes ................................................................. 415
Change in Behavior of pfcServer::IsObjectCheckedOut .......................... 415
Object TOOLKIT C++ Support for Creo Applications ............................ 415
Support for Asynchronous Mode ......................................................... 415
Support for Creo Elements/Direct ....................................................... 416
New Functions ................................................................................. 416
Assembly ........................................................................................... 416
Associative Topology Bus (ATB)-Enabled Features ............................... 419
Creo Parametric Datecode .................................................................... 420
Creo Parametric Version Number ......................................................... 420
Drawing Sheet Information .................................................................. 420
Drawing View Display Styles ............................................................... 421
Export 3D Instructions ....................................................................... 422
Export to PDF .................................................................................... 422
Facet Control Options ....................................................................... 422
Import Feature .................................................................................. 423
Import Model ..................................................................................... 423
Interactive Curve Collection ............................................................... 423
Layer Items ....................................................................................... 423
Models ............................................................................................... 423
Multiple Threads Mode ...................................................................... 423
Post Regeneration Relations ............................................................... 424
Quick Drawing Options ...................................................................... 424
Sections ............................................................................................. 426
Session Objects ................................................................................ 427
Simplified Representation in Views ..................................................... 427
Sketcher Element Data ....................................................................... 427
Step Size for Exported Files ............................................................... 428
Threaded Hole Properties .................................................................. 428
Superseded Functions ...................................................................... 428
Drawing Sheet Information ................................................................. 429
Layer Items ....................................................................................... 429
Miscellaneous Technical Changes ..................................................... 429
This chapter describes the critical and miscellaneous technical changes in Creo Parametric 2.0 and Object TOOLKIT C++. It also lists the new and superseded functions for this release.
Critical Technical Changes

This section describes the changes in Creo Parametric 2.0 and Object TOOLKIT C++ that might require alteration of existing Object TOOLKIT C++ applications.

Change in Behavior of pfcServer::IsObjectCheckedOut

From Creo Parametric 2.0 M040 onward, the behavior of the method pfcServer::IsObjectCheckedOut has been fixed. The method now returns the value false to indicate one of the following statuses for the specified object:

- the object was only saved, but never uploaded
- the object was only uploaded, but never checked in

Object TOOLKIT C++ Support for Creo Applications

Object TOOLKIT C++ applications in synchronous and asynchronous modes are supported with the Creo Parametric application only. They are not supported with the other Creo applications, such as, Creo Direct, Creo Layout, Creo Layout, Creo Simulate and so on.

The methods pfcAsyncConnection::Connect and pfcAsyncConnection::Start return an error when the Creo Parametric TOOLKIT application attempts to connect to a Creo application other than Creo Parametric. Refer to the section Object TOOLKIT C++ Support for Creo Applications in the Creo Parametric 2.0 Object TOOLKIT C++ User’s Guide.

Support for Asynchronous Mode

The Object TOOLKIT C++ methods now support the asynchronous mode of operation.
Support for Creo Elements/Direct

You can import the Creo Elements/Direct assemblies and parts using the Object TOOLKIT C++ method wfcWSession::ImportAsModel. Creo Parametric 2.0 supports the import of following file types for Creo Elements/Direct:

- bundle (.bdl)
- modeling (soliddesigner file types .sda, .sdp, .sdac, and .sdpc)
- package (.pkg)

A new import type pfcIMPORT_NEW_CC has been added to the enumerated type pfcNewModelImportType.

New Functions

This section describes new functions for Object TOOLKIT C++ for Creo Parametric 2.0.

Assembly

New functions have been added for assemblies.

Component Path

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcSubstituteComponent::GetSubCompPath</td>
<td>Returns more information about the component paths, substitution types, and displayed entities in an assembly.</td>
</tr>
<tr>
<td>wfcSubstituteComponent::GetSubCompFeat</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentPath::GetSubstituteComponent</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentPath::GetSubstitutionType</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentPath::ListPoints</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentPath::ListCsyses</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentPath::ListCurves</td>
<td></td>
</tr>
</tbody>
</table>
### New Function Description

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWComponentPath::ListQuilts</td>
<td></td>
</tr>
</tbody>
</table>

### Flexible Component Variant Items

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcAssemblyItem Instructions::Create</td>
<td>Instructions to create a variant item for a flexible component.</td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::GetItemOwner</td>
<td></td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::SetItemOwner</td>
<td></td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::GetItemComp Path</td>
<td></td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::SetItemComp Path</td>
<td></td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::GetItemId</td>
<td></td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::SetItemId</td>
<td></td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::GetItemType</td>
<td></td>
</tr>
<tr>
<td>wfcAssemblyItem Instructions::SetItemType</td>
<td></td>
</tr>
</tbody>
</table>
### Flexible Components

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWComponentFeat::CreateFlexibleModel</td>
<td>Provides read and write access to the flexible assembly components.</td>
</tr>
<tr>
<td>wfcWComponentFeat::CreatePredefinedFlexibilityComponent</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentFeat::IsFlexible</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentFeat::SetAsFlexible</td>
<td></td>
</tr>
<tr>
<td>wfcWComponentFeat::UnsetAsFlexible</td>
<td></td>
</tr>
</tbody>
</table>

### Interchange Components

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWAssembly::AutoInterchange</td>
<td>Interchanges an assembly component with another component.</td>
</tr>
</tbody>
</table>

### Listing Components

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWAssembly::GetConnectors</td>
<td>Lists the components for specified assembly.</td>
</tr>
<tr>
<td>wfcWAssembly::GetHarnesses</td>
<td></td>
</tr>
<tr>
<td>wfcWAssembly::GetLinestocks</td>
<td></td>
</tr>
<tr>
<td>wfcLineStock::GetName</td>
<td></td>
</tr>
<tr>
<td>wfcLineStock::SetName</td>
<td></td>
</tr>
<tr>
<td>wfcWAssembly::GetSpools</td>
<td></td>
</tr>
<tr>
<td>wfcSpool::GetName</td>
<td></td>
</tr>
<tr>
<td>wfcSpool::SetName</td>
<td></td>
</tr>
</tbody>
</table>
### Remove Component Constraint

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWComponentFeat::RemoveConstraint</td>
<td>Removes the constraints for the specified assembly component.</td>
</tr>
</tbody>
</table>

### Unique Instance of a Subassembly

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWComponentFeat::MakeUniqueSubAssembly</td>
<td>Creates and removes the unique instance of a subassembly.</td>
</tr>
<tr>
<td>wfcWComponentFeat::RemoveUniqueSubAssembly</td>
<td>Creates and removes the unique instance of a subassembly.</td>
</tr>
</tbody>
</table>

### Unplaced Components

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWComponentFeat::IsUnplaced</td>
<td>Determines if the component is unplaced.</td>
</tr>
</tbody>
</table>

### Associative Topology Bus (ATB)-Enabled Features

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWModel::VerifyATB</td>
<td>Verifies if the specified ATB model is out-of-date.</td>
</tr>
<tr>
<td>wfcATBVerificationResults::GetOutOfDateModels</td>
<td>Lists the Translated Image Models (TIMs) or the ATB-enabled features that are out-of-date.</td>
</tr>
<tr>
<td>wfcATBVerificationResults::GetUnlinkedModels</td>
<td>Lists the TIMs or the ATB-enabled features that have missing links.</td>
</tr>
<tr>
<td>wfcATBVerificationResults::GetOldVersionModels</td>
<td>Lists TIMs for which the source CAD model is older than the one with which the TIM was last updated.</td>
</tr>
<tr>
<td>wfcWModel::MarkATBModelAsOutOfDate</td>
<td>Marks the specified ATB model or feature as out-of-date.</td>
</tr>
<tr>
<td>wfcWModel::UpdateATB</td>
<td>Updates the specified ATB enabled models or features.</td>
</tr>
</tbody>
</table>
### New Function Description

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wfcWModel::RelinkATB</code></td>
<td>Relinks all the models or features that have lost their association or link with their master model.</td>
</tr>
<tr>
<td><code>wfcWModel::GetTIMInfo</code></td>
<td>Returns information about TIM in the specified model.</td>
</tr>
<tr>
<td><code>wfcTIMInfo::IsModelTIM</code></td>
<td>Checks if the specified feature is a TIM.</td>
</tr>
<tr>
<td><code>wfcTIMInfo::ModelHasTIMFeats</code></td>
<td>Checks if the specified model has a TIM feature.</td>
</tr>
<tr>
<td><code>wfcTIMInfo::FeatIds</code></td>
<td>Lists the feature IDs of all the ATB-enabled TIM features in the specified model.</td>
</tr>
</tbody>
</table>

### Creo Parametric Datecode

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wfcWSession::GetDisplayDateCode</code></td>
<td>Returns the user-visible datecode from Creo Parametric.</td>
</tr>
</tbody>
</table>

### Creo Parametric Version Number

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wfcWSession::GetReleaseNumericVersion</code></td>
<td>Returns the version number of the Creo Parametric executable to which the Object TOOLKIT C++ application is connected.</td>
</tr>
</tbody>
</table>

### Drawing Sheet Information

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pfcSheetOwner::GetSheetInfo</code></td>
<td>Returns information about the drawing sheet.</td>
</tr>
</tbody>
</table>
## Drawing View Display Styles

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcDXF3DExport Instructions::Create</td>
<td>Contains information about the display styles being used in a view.</td>
</tr>
<tr>
<td>pfcDWG3DExport Instructions::Create</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::Create</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::GetCableDisp</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::SetCableDisp</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::GetConceptModel</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::SetConceptModel</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::GetDispStyle</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::SetDispStyle</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::GetQuiltHLR</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::SetQuiltHLR</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::GetTanEdgeDisplay</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::SetTanEdgeDisplay</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::GetWeldXSec</td>
<td></td>
</tr>
<tr>
<td>wfcDrawingViewDisplay::SetWeldXSec</td>
<td></td>
</tr>
</tbody>
</table>
Export 3D Instructions

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcDXF3DExport Instructions::Create</td>
<td>Creates instructions objects used to export a model to DWG and DXF formats.</td>
</tr>
<tr>
<td>pfcDWG3DExport Instructions::Create</td>
<td></td>
</tr>
</tbody>
</table>

Export to PDF

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcPDFExportInstructions::GetProfilePath</td>
<td>Provides read and write access to the profile path in PDF export.</td>
</tr>
<tr>
<td>pfcPDFExportInstructions::SetProfilePath</td>
<td></td>
</tr>
</tbody>
</table>

Facet Control Options

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcCoordSysExport Instructions::GetFacetControlOptions</td>
<td>Provides read and write access to the flags that control the facet export options.</td>
</tr>
<tr>
<td>pfcCoordSysExport Instructions::SetFacetControlOptions</td>
<td></td>
</tr>
<tr>
<td>pfcTriangulation Instructions::GetFacetControlOptions</td>
<td></td>
</tr>
<tr>
<td>pfcTriangulation Instructions::SetFacetControlOptions</td>
<td></td>
</tr>
</tbody>
</table>
Import Feature

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWSolid::ImportAsFeat</td>
<td>Creates a new import feature in the solid and takes the profile path as one of its input arguments.</td>
</tr>
</tbody>
</table>

Import Model

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWSolid::ImportAsModel</td>
<td>Imports a model in the solid.</td>
</tr>
</tbody>
</table>

Interactive Curve Collection

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWSolid::CollectCurves</td>
<td>Interactively creates a collection of curves.</td>
</tr>
</tbody>
</table>

Layer Items

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcLayer::CountUnsupportedItems</td>
<td>Returns the count of item types not supported as a pfcModelItem object in the specified layer.</td>
</tr>
</tbody>
</table>

Models

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcModel::CleanupDependencies</td>
<td>Cleans the dependencies for an object in the Creo Parametric workspace.</td>
</tr>
</tbody>
</table>

Multiple Threads Mode

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWSession::EnableMultiThreadMode</td>
<td>Runs Creo Parametric in the multithread-enabled mode.</td>
</tr>
</tbody>
</table>
## Post Regeneration Relations

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcModel::RegeneratePost RegenerationRelations</td>
<td>Regenerates and deletes the post regeneration relations in a model.</td>
</tr>
<tr>
<td>pfcModel::DeletePost RegenerationRelations</td>
<td></td>
</tr>
</tbody>
</table>

## Quick Drawing Options

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcQuickPrintGeneralView Instructions::Create</td>
<td>Provides read and write access to the instructions to print a solid model.</td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::GetGeneral ViewLocation</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::SetGeneral ViewLocation</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::GetScale</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::SetScale</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::GetViewDisplayStyle</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::SetViewDisplayStyle</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::GetViewName</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintGeneralView Instructions::SetViewName</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::Create</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetDrawingTemplate</td>
<td></td>
</tr>
<tr>
<td>New Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetDrawingTemplate</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetGeneralViewInstructions</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetGeneralViewInstructions</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetLayoutType</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetLayoutType</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetManualLayoutType</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetManualLayoutType</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetOrientation</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetOrientation</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetPaperSize</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetPaperSize</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetPrintFlatToScreen</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetPrintFlatToScreen</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::GetProjectionViewLocations</td>
<td></td>
</tr>
<tr>
<td>wfcQuickPrintInstructions::SetProjectionViewLocations</td>
<td></td>
</tr>
</tbody>
</table>
# Sections

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcSection::SetName</td>
<td>Creates and manipulates 2D and 3D sections.</td>
</tr>
<tr>
<td>wfcSection::SetIntent ManagerMode</td>
<td></td>
</tr>
<tr>
<td>wfcSection::SetIntent ManagerMode</td>
<td></td>
</tr>
<tr>
<td>wfcWSession::GetActive Section</td>
<td></td>
</tr>
<tr>
<td>wfcSection::SetActive</td>
<td></td>
</tr>
<tr>
<td>wfcSection::GetEpsilon</td>
<td></td>
</tr>
<tr>
<td>wfcSection::SetEpsilon</td>
<td></td>
</tr>
<tr>
<td>wfcSection::ListSection Entities</td>
<td></td>
</tr>
<tr>
<td>wfcWFeature::GetSections</td>
<td></td>
</tr>
<tr>
<td>wfcSection::GetLocation</td>
<td></td>
</tr>
<tr>
<td>wfcWSession::CreateSection 2D</td>
<td></td>
</tr>
<tr>
<td>wfcSection::AddEntity</td>
<td></td>
</tr>
<tr>
<td>wfcSection::DeleteEntity</td>
<td></td>
</tr>
<tr>
<td>wfcSection::GetEntity</td>
<td></td>
</tr>
<tr>
<td>wfcSection::GetEntityIds</td>
<td></td>
</tr>
<tr>
<td>wfcSectionEntity::Get SectionEntityType</td>
<td></td>
</tr>
</tbody>
</table>

The following classes allow you to create and modify various types of section entities:

- wfcSectionEntity
- wfcSectionEntityArc
- wfcSectionEntityBlendVertex
- wfcSectionEntityCSys
- wfcSectionEntityCenterLine
Session Objects

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcCreateMatrix3D</td>
<td>Initialize the matrix, outline, point and vector objects.</td>
</tr>
<tr>
<td>wfcCreatePoint2D</td>
<td></td>
</tr>
<tr>
<td>wfcCreatePoint3D</td>
<td></td>
</tr>
<tr>
<td>wfcCreateOutline2D</td>
<td></td>
</tr>
<tr>
<td>wfcCreateOutline3D</td>
<td></td>
</tr>
<tr>
<td>wfcCreateVector2D</td>
<td></td>
</tr>
<tr>
<td>wfcCreateVector3D</td>
<td></td>
</tr>
</tbody>
</table>

Simplified Representation in Views

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcView2D::GetSimpRep</td>
<td>Returns the simplified representation for the specified drawing view.</td>
</tr>
</tbody>
</table>

Sketcher Element Data

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcSpecialValue::GetSectionValue</td>
<td>Provides read and write access to the value of the element PRO_E_SKETCHER for the specified feature.</td>
</tr>
</tbody>
</table>
# Step Size for Exported Files

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcCoordSysExport Instructions::GetStepSize</td>
<td>Provides read and write access to the step size of the exported files.</td>
</tr>
<tr>
<td>pfcCoordSysExport Instructions::SetStepSize</td>
<td></td>
</tr>
<tr>
<td>pfcTriangulation Instructions::GetStepSize</td>
<td></td>
</tr>
<tr>
<td>pfcTriangulation Instructions::SetStepSize</td>
<td></td>
</tr>
</tbody>
</table>

# Threaded Hole Properties

<table>
<thead>
<tr>
<th>New Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfcWHoleFeature::GetHoleProperties</td>
<td>Gets and sets the type of thread for the hole and for the screw size.</td>
</tr>
<tr>
<td>wfcWHoleFeature::SetHoleProperties</td>
<td></td>
</tr>
<tr>
<td>wfcHoleProperties::Create</td>
<td></td>
</tr>
<tr>
<td>wfcHoleProperties::GetThreadSeries</td>
<td></td>
</tr>
<tr>
<td>wfcHoleProperties::SetThreadSeries</td>
<td></td>
</tr>
<tr>
<td>wfcHoleProperties::GetScrewSize</td>
<td></td>
</tr>
<tr>
<td>wfcHoleProperties::SetScrewSize</td>
<td></td>
</tr>
</tbody>
</table>

# Superseded Functions

This section describes the superseded functions for Object TOOLKIT C++ for Creo Parametric 2.0.
**Drawing Sheet Information**

<table>
<thead>
<tr>
<th>Superseded Function</th>
<th>New Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcSheetOwner::GetSheet Data</td>
<td>pfcSheetOwner::GetSheet Info</td>
</tr>
</tbody>
</table>

**Layer Items**

<table>
<thead>
<tr>
<th>Superseded Function</th>
<th>New Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcLayer::Has UnsupportedItems</td>
<td>pfcLayer::Count UnsupportedItems</td>
</tr>
</tbody>
</table>

**Miscellaneous Technical Changes**

The following changes in Creo Parametric 2.0 can affect the functional behavior of Object TOOLKIT C++. PTC does not anticipate that these changes cause critical issues with existing Object TOOLKIT C++ applications.

**Import Feature from Inventor, IBS, and PTS File**

You can create the import feature from Inventor assembly and part files, IBS, and PTL file types.
This appendix describes the variety of libraries available in an Object TOOLKIT C++ installation.
Standard Libraries

Object TOOLKIT C++ applications will be linked with the following libraries:

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>protoolkit.lib otk_cpp.lib</td>
<td>Spawn mode library</td>
</tr>
<tr>
<td>pt_asynchronous.lib protoolkit.lib otk_cpp_async.lib</td>
<td>Asynchronous mode library</td>
</tr>
<tr>
<td>protk_dll.lib otk_cpp.lib</td>
<td>DLL mode library</td>
</tr>
</tbody>
</table>

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><platform><obj>.
This appendix describes the implications of DRM on Object TOOLKIT C++ applications.
Introduction

Digital Rights Management (DRM) helps to control access to your intellectual property. Intellectual property could be sensitive design and engineering information that you have stored within Creo Parametric parts, assemblies, or drawings. You can control access by applying policies to these Creo Parametric objects. Such objects remain protected by the policies even after they are distributed or downloaded. Creo Parametric objects for which you have applied policies are called DRM-protected objects. For more information on the use of DRM in Creo Parametric Creo Parametric 4.0, refer to the DRM online help.

The following sections describe how Object TOOLKIT C++ applications deal with DRM-protected objects.

Implications of DRM on Object TOOLKIT C++

Any Object TOOLKIT C++ application accessing DRM-protected objects can run only in interactive Creo Parametric sessions having COPY permissions. As Object TOOLKIT C++ applications can extract content from models into an unprotected format, Object TOOLKIT C++ applications will not run in a Creo Parametric session lacking COPY permission.

If the user tries to open a model lacking the COPY permission into a session with a Object TOOLKIT C++ application running, Creo Parametric prompts the user to spawn a new session. Also, new Object TOOLKIT C++ applications will not be permitted to start when the Creo Parametric session lacks COPY permission.

If a Object TOOLKIT C++ application tries to open a model lacking COPY permission from an interactive Creo Parametric session, the application throws the pfcXToolkitNoPermission exception.

When a Object TOOLKIT C++ application tries to open a protected model from a non-interactive or batch mode application, the session cannot prompt for DRM authentication, instead the application throws the pfcXToolkitAuthenticationFailure exception.

Exception Types

Some Object TOOLKIT C++ methods require specific permissions in order to operate on a DRM-protected object. If these methods cannot proceed due to DRM restrictions, the following exceptions are thrown:
• `pfcXToolkitNoPermission`—Thrown if the method cannot proceed due to lack of needed permissions.

• `pfcXToolkitAuthenticationFailure`—Thrown if the object cannot be opened because the policy server could not be contacted or if the user was unable to interactively login to the server.

• `pfcXToolkitUserAbort`—Thrown if the object cannot be operated upon because the user cancelled the action at some point.

The following table lists the methods along with the permission required and implications of operating on DRM-protected objects.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Permission Required</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pfcBaseSession::RetrieveAssem SimpRep</code></td>
<td>OPEN</td>
<td>If file has OPEN and COPY permissions, model opens after authentication.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Throws the <code>pfcXToolkitNoPermission</code> exception otherwise.</td>
</tr>
<tr>
<td><code>pfcBaseSession::CreateDrawing FromTemplate</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcBaseSession::RetrieveGraphics SimpRep</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcBaseSession::RetrieveGeomSimpRep</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcBaseSession::RetrieveModel</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcBaseSession::RetrieveModel WithOpts</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcBaseSession::RetrievePartSimpRep</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcBaseSession::RetrieveSymbolic SimpRep</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcModel::Rename</code></td>
<td>OPEN</td>
<td>File is saved with the current policy to disk if it has COPY permission.</td>
</tr>
<tr>
<td><code>pfcModel::Backup</code></td>
<td>SAVE</td>
<td>File is saved with the current policy to disk if it has SAVE and COPY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>permissions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Throws the <code>pfcXToolkitNoPermission</code> exception if model has COPY permission, but lacks SAVE permission.</td>
</tr>
<tr>
<td><code>pfcModel::Copy</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pfcModel::Save</code></td>
<td>SAVE</td>
<td>File is saved with the current policy to disk if it has SAVE and COPY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>permissions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Throws the <code>pfcXToolkitNoPermission</code> exception if model has COPY permission, but lacks SAVE permission.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Throws the <code>pfcXToolkitNoPermission</code> exception if the assembly file has models with COPY permission, but lacking SAVE permission.</td>
</tr>
</tbody>
</table>
Methods & Permissions

<table>
<thead>
<tr>
<th>Method</th>
<th>Permission Required</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfcModel::Export for pfcPlotInstructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfcModel::Export for pfcProductView-ExportInstructions</td>
<td>PRINT</td>
<td>Drawing file is printed if it has PRINT permission.</td>
</tr>
<tr>
<td>(only if the input model is a drawing)</td>
<td></td>
<td>Throws the pfcXToolkitNoPermission exception if drawing file lacks PRINT permission.</td>
</tr>
<tr>
<td>pfcBaseSession::ExportCurrent RasterImage</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Copy Permission to Interactively Open Models**

When the user tries to open protected content lacking COPY permission through the Creo Parametric user interface with a Object TOOLKIT C++ application running in the same session:

1. Creo Parametric checks for the authentication credentials through the user interface, if they are not already established.
2. If the user has permission to open the file, Creo Parametric checks if the permission level includes COPY. If the level includes COPY, Creo Parametric opens the file.
3. If COPY permission is not included, the following message is displayed:

   ![Confirmation Message]

   You do not have COPY permission for this DRM protected content, and therefore you cannot access Pro/TOLKIT applications. A new session of Pro/ENGINEER will be started without the Pro/TOLKIT applications. Proceed?

   OK Cancel

4. If the user clicks **Cancel**, the file is not opened in the current Creo Parametric session and no new session is spawned.
5. If the user clicks **OK**, an additional session of Creo Parametric is spawned which does not permit any Object TOOLKIT C++ application. Object TOOLKIT C++ applications set to automatically start by Creo Parametric will not be started. Asynchronous applications will be unable to connect to this session.
6. The new session of Creo Parametric is automatically authenticated with the same session credentials as were used in the previous session.
7. The model that Creo Parametric was trying to load in the previous session is loaded in this session.
8. Other models already open in the previous session will not be loaded in the new session.

9. Session settings from the previous session will not be carried into the new session.

10. The new session will be granted the licenses currently used by the previous session. This means that the next time the user tries to do something in the previous session, Creo Parametric must obtain a new license from the license server. If the license is not available, the action will be blocked with an appropriate message.

**Additional DRM Implications**

- The method `pfcModel::CheckIsSaveAllowed` returns `false` if prevented from save by DRM restrictions.
- The method `pfcBaseSession::CopyFileToWS` is designed to fail and throw the `pfcXToolkitNoPermission` exception if passed a DRM-protected Creo Parametric model file.
- The method `pfcBaseSession::ImportToCurrentWS` reports a conflict in its output text file and does not copy a DRM-protected Creo Parametric model file to the Workspace.
- While erasing an active Creo Parametric model with DRM restrictions, the methods `pfcModel::Erase` and `pfcModel::EraseWithDependencies` do not clear the data in the memory until the control returns to Creo Parametric from the Object TOOLKIT C++ application. Thus, the Creo Parametric session permissions may also not be cleared immediately after these methods return.
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.
Geometry Representations

Surface Parameterization ................................................................. 444
Plane .................................................................................................. 444
Cylinder ......................................................................................... 445
Cone .................................................................................................. 446
Torus .................................................................................................. 446
General Surface of Revolution .......................................................... 447
Ruled Surface .................................................................................. 448
Tabulated Cylinder ......................................................................... 448
Coons Patch ...................................................................................... 449
Fillet Surface .................................................................................... 449
Spline Surface ................................................................................... 450
NURBS Surface ............................................................................... 451
Cylindrical Spline Surface ............................................................... 452
Edge and Curve Parameterization .................................................... 453
Line .................................................................................................. 454
Arc ................................................................................................... 454
Spline ............................................................................................... 454
NURBS .............................................................................................. 455

This appendix describes the geometry representations of the data used by
Object TOOLKIT C++. 
Surface Parameterization

A surface in Creo Parametric 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:

- Cone on page 446
- Coons Patch on page 449
- Cylinder on page 445
- Cylindrical Spline Surface on page 452
- Fillet Surface on page 449
- General Surface of Revolution on page 447
- NURBS on page 455
- Plane on page 444
- Ruled Surface on page 448
- Spline Surface on page 450
- Tabulated Cylinder on page 448
- Torus on page 446

Plane

![Plane Diagram]

The plane entity consists of two perpendicular unit vectors ($e_1$ and $e_2$), the normal to the plane ($e_3$), and the origin of the plane.

Data Format:
Parameterization:

\[(x, y, z) = u \cdot e1 + v \cdot e2 + \text{origin} \]

**Cylinder**

![Cylinder diagram](image)

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:

<table>
<thead>
<tr>
<th>e1[3]</th>
<th>Unit vector, in the u direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2[3]</td>
<td>Unit vector, in the v direction</td>
</tr>
<tr>
<td>e3[3]</td>
<td>Normal to the plane</td>
</tr>
<tr>
<td>origin[3]</td>
<td>Origin of the plane</td>
</tr>
<tr>
<td>radius</td>
<td>Radius of the cylinder</td>
</tr>
</tbody>
</table>

Parameterization:

\[
(x, y, z) = \text{radius} \cdot (\cos(u) \cdot e1 + \sin(u) \cdot e2) + v \cdot e3 + \text{origin}
\]

**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 \((e1, e2, \text{ and } e3)\) 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 \( v \times \tan(\alpha) \).

Data Format:
- \( e1[3] \): Unit vector, in the u direction
- \( e2[3] \): Unit vector, in the v direction
- \( e3[3] \): Normal to the plane
- \( \text{origin}[3] \): Origin of the cone
- \( \alpha \): Angle between the axis of the cone and the generating line

Parameterization:
\[
(x, y, z) = v \times \tan(\alpha) \times (\cos(u) \times e1 + \sin(u) \times e2) + v \times e3 + \text{origin}
\]

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 \( R1 + R2 \times \cos(v) \), and the height of the point along the axis of revolution is \( R2 \times \sin(v) \).

Data Format:
- \( e1[3] \): Unit vector, in the u direction
- \( e2[3] \): Unit vector, in the v direction
- \( e3[3] \): Normal to the plane
- \( \text{origin}[3] \): 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:

\[(x, y, z) = (R1 + R2 \cdot \cos(v)) \cdot [\cos(u) \cdot e1 + \sin(u) \cdot e2] + R2 \cdot \sin(v) \cdot e3 + \text{origin}\]

**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
- \(\text{origin}[3]\)  Origin of the surface of revolution
- \(\text{curve}\)  Generating curve

**Parameterization:**

\[
(x, y, z) = [c1 \cdot \cos(u) - c2 \cdot \sin(u)] \cdot e1 + [c1 \cdot \sin(u) + c2 \cdot \cos(u)] \cdot e2 + c3 \cdot e3 + \text{origin}
\]
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
- $curve_1$ 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  
uvnder_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:

\[
R(v) = P(v) - C(v) \\
(x,y,z) = C(v) + R(v) \cdot \cos(u) + T(v) \times R(v) \times \sin(u)
\]

**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 \( Nu \times Nv \)
- **u_tan_arr[][][3]**: Array of \( u \) tangent vectors at interpolant points, of size \( Nu \times Nv \)
- **v_tan_arr[][][3]**: Array of \( v \) tangent vectors at interpolant points, of size \( Nu \times Nv \)
- **uvder_arr[][][3]**: Array of mixed derivatives at interpolant points, of size \( Nu \times Nv \)
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.

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 \(v\)
- wgths[] Array of weights for rational NURBS, otherwise NULL
- c_point_arr[][3] Array of control points

Definition:
\[ R(u, v) = \sum_{i=0}^{N1} \sum_{j=0}^{N2} \frac{N1 \times k \times B_{i,k}(u) \times B_{j,l}(v)}{N1 \times N2} \]

\[ \sum_{i=0}^{N1} \sum_{j=0}^{N2} w_{i,j} \times B_{i,k}(u) \times B_{j,l}(v) \]

\( k \) = 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 \((x,y,z)*wij\)

**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 \(S1\) was replaced with a cone \((r_1 = r_2, r_3 = r_4, \text{ and } r_1 \neq r_3)\).

If a replacement cannot be done (such as for the surface \(S0\) in the figure \((r_a \neq rb \text{ or } rc \neq rd)\)), 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 454
• Line on page 454
• NURBS on page 455
• Spline on page 454

Line

Data Format:

end1[3]  Starting point of the line
end2[3]  Ending point of the line

Parameterization:

\[(x, y, z) = (1 - t) \times \text{end1} + t \times \text{end2}\]

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:

\[t' \text{ (the unnormalized parameter) is}\]
\[(l - t) \times \text{start_angle} + t \times \text{end_angle}\]
\[(x, y, z) = \text{radius} \times [\cos(t') \times \text{vector1} +\sin(t') \times \text{vector2}] + \text{origin}\]

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 th spline segment such that:
par_arr[i] < t' < par_arr[i+1]

(If t < 0 or t > p_max, 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.

---

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
- \( w_i \) = weights
- \( C_i \) = control points \((x, y, z) \times w_i\)
- \( B_{i,k} \) = basis functions

By this equation, the number of control points equals \( N+1 \).

References:


Index

2-D
  sections
    allocating, 241

A
Accuracy
  getting and setting, 156
ActionListener
  creating, 304
events, 304
feature-level, 309
session-level, 305
solid-level, 308
types, 304
UI command, 306
ActionListener classes, 24
ActionSource interface, 304
definition, 24
Activate
  window, 189
Add
  items to a layer, 200
section entities, 242
Allocate
  2-D sections, 241
simplified representations, 359
APIWizard
  browsing
    objects, 40
    user’s guide, 41
defined, 36
display frame, 38
find (search mechanism), 43
interface defined, 37
navigating the tree, 40
searching for a string, 43
  supported web browsers, 37
topic/object selection frame, 37
tree updating, 37
Applications
  creating, 28
hierarchy, 28
Arcs
  description, 253
representation, 454
Area
  surface, 258
Array classes, 22
Arrays, 22
Arrows, 253
Assemblies
  coordinate systems, 193
creating, 156
hierarchy, 282
structure of, 282
asynchronous mode, 365
Attributes
  array classes, 23
compact data classes, 19
Creo Parametric-related objects, 17
sequence classes, 21
Axes, 260
evaluating, 260

B
B-splines
  description, 253
Browsing
  Creo Parametric TOOLKIT
    user’s guide with APIWizard, 41
objects with APIWizard, 40
Cells accessing, 299
Children, 214
Circles, 253
Classes types, 16
Clear window, 189
Close window, 189
Collection curve collection, 179
interactive for curves, 177
introduction, 176
programmatic access, 179
surface collection, 182
Colors, 51
Commands designating, 83
Composite curves description, 253
Cones
class representation, 257
genre presentation, 446
Configuration options, 50
Contours evaluating, 256
traversing, 252
Contours, locating in a model, 256
Coons patches geometry representation, 449
Coordinate systems, 260
assemblies, 193
datum, 193
drawing, 193
Drawing View, 193
evaluating, 260
screen, 192
section, 194
solid, 192
window, 193
Coordinate transformations, 191
Copy models, 95
Copying sections, 241
Create
2-D sections, 240
action listeners, 304
applications, 28
assembly, 156
buttons, 80
family table columns, 300
family table instance, 298
layer, 200
local group, 219
material, 168
menus, 80
parameters, 265
part, 156
section models, 240
simplified representations, 359
UDFs, 221
window, 188
Create Interactively Defined UDFs, 222
Creating UDFs, 221
Creo Parametric accessing, 52
Curves, 253
data structures, 453
determining the type, 253
t parameter, 253
types, 253
reserved, 253
Cylinders, 257
genre presentation, 445
spline surfaces, 452
tabulated, 257
genre presentation, 448

Data types
enums, 23
Delete
  feature pattern, 219
  models, 95
  section entities, 242
  simplified representations, 359
Depth
  selection, 65
Descriptors
  model, 90
Designating
  command, 84
  commands, 83
  icon, 84
Designating commands, 83
Detail Entity interface
  description, 131
Dictionaries, 23
Dimension2D.Dimension2D
  interface
    description, 118
Dimensions, 272
  information, 272
  tolerances, 273
Display
  model in window, 188
  models, 95
  selection, 66
Display status
  of layers, 200
Documentation
  see APIWizard, 36
Drawing
  transformations, 196

E
Edges, 253
  determining the type, 253
  evaluating, 254
  t parameter, 253
  traversing, 252
  types, 253
  reserved, 253
Element
  diagnostics, 211
Ellipses, 253
Entities
  adding to sections, 242
Enumerated Classes, 23
Enumerated types, 23
Epsilon
  specifying, 242
Erase
  models, 95
Evaluation
  axes, 260
  contour, 256
  coordinate system, 260
  edge, 254
  point, 260
  surface, 258
Event handling
  try-catch-finally blocks, 29
Examples
  ActionListener classes, 25
  creating a 2D section, 241
  creating a sweep section, 247
  manipulating a 3D section, 248
  normalizing a coordinate transformation matrix, 196
  of sequences, 21
  of utilities, 26
Exceptions
  ActionListener, 25
  array classes, 23
  compact data classes, 19
  Creo Parametric-related objects, 18
  handling in code, 29
  sequence classes, 21
  utility, 26
Export
  files, 312
F

Faces
  traversing, 252
Family tables, 298
cells, 299
  columns
    accessing, 299
instances
  accessing, 298
symbols, 299
Features
  accessing, 214
creating, 216
element paths, 207
element special values, 206
element tree, 207
element values, 206
failed, 214
groups, 214, 219
identifiers, 214
information, 215
operations, 216
parents, 214
patterns, 219
read-only, 215
redefine, 210
resuming, 216
suppressing, 216
user-defined, 220
WCreate, 208
Fields
  ActionListener classes, 24
of utilities, 26
Files
  exporting, 312
message, 52
  contents, 53
naming restrictions, 52
Fillet surfaces
  geometry representation, 449
Find
  APIWizard search mechanism, 43
Frames
  display frame in APIWizard, 38
topic/object selection, 37

G

General surface of revolution, 447
Generic model
  getting, 298
Geometry
  solid edge, 255
terms, 252
traversal, 252
Groups, 219220

H

Hierarchy
  application, 28
Highlight
  selections, 66

I

Information
  Drawing, 107
Inheritance
  ActionListener classes, 25
  compact data classes, 19
  Creo Parametric-related objects, 18
  of arrays, 23
  of utilities, 26
  sequence classes, 21
Initialize
  ActionListener classes, 24
  array classes, 23
  compact data classes, 19
  Creo Parametric-related objects, 17
  sequence classes, 21
  utilities, 26
Initializing objects, 51
Installation, 14
Interactive selection, 64
Interactively Defined UDFs
create, 222

K
Keywords
instanceof
using, 253

L
Layers, 200
operations, 200
Libraries
standard, 432
Lines
description, 253
representation, 454
styles, 51
Lists
of children, 214
of current windows, 188
of layer items, 200
of materials, 168
of ModelItems, 198
of pattern members, 214, 219
of rows in a family table, 298
of subitems, 198
of views, 190
of windows, 188
Local groups
creating, 219
Locks, 298

M
Macros, 50
Mass properties, 166
Materials, 168
Matrix
code example, 196
Matrix3D object, 196
Memory management
Creo Parametric-related objects, 17
Message files, 52
contents, 53
restrictions, 52
Message window
reading from, 54
writing to, 54
Methods
ActionListener classes, 24
array classes, 23
compact data classes, 19
Creo Parametric-related objects, 18
of utilities, 26
sequence classes, 21
ModelItems
duplicating, 199
evaluating, 260
getting, 198
information, 199
types, 198
Models
descriptors, 90
Drawing
Obtaining, 106
exporting, 312
getting, 90
operations, 95
retrieving, 91
section, 240
Modify
simplified representations, 360

N
Normalize
matrix, 196
NURBS
representation, 455
surface, 451

O
Object TOOLKIT C++
class types, 16
enumerated types, 23
installation, 14
registry file, 14
setting up, 14
Objects
browsing with APIWizard, 40
Open
file, 91
Operations
Drawing, 107
feature, 216
layer, 200
model, 95
solid, 156, 354
view, 191
window, 189, 356
Outlines
contour, 256

P
Parameters, 265
information, 268
ParamValue objects, 264
Parents, 214
Parts, 168
creating, 156
Pattern leaders, 214, 219
Patterns, 219
create, 210
pfcModel::Models
information, 92
pfcXToolkitDrawingCreateErrors
description, 105
Planes, 257
graph geometry representation, 444
Points, 260
evaluating, 260
Polygons, 253
Popup Menu
Adding to the Graphics
Window, 86
Using Trail files to determine
names, 86
Popup menus
Adding, 87
Popup Menus, 85
Accessing, 87
Principal curve, 258

Q
Quick drawing instructions, 350

R
Refresh
window, 189, 356
Regenerate
events, 305
solids, 156, 354
Registry file, 14
Remove
items from a layer, 200
Rename
models, 95
Repaint
events, 305
window, 189, 356
Reset
view, 191
Restrictions
on text message files, 52
Retrieve
2-D sections, 244
graph geometry of a simplified
representation, 359
graphics of a simplified
representation, 359
material, 168
simplified representations, 359
view, 190
Revolved surfaces, 257
Rotate
view, 191
Ruled surfaces, 257
graph geometry representation, 448
Save models, 95
view, 191
Screen coordinate system, 192
Search
  APIWizard search mechanism, 43
  using the APIWizard, 43
Sections
  allocating, 241
  copying, 241
  creating
    2-D, 240
    models, 240
  definition, 239
  entities, 242
  examples
    creating a 2D section, 241
    creating a sweep section, 247
    manipulating a 3D section, 248
  mode, 240
  retrieving, 244
Selection, 64
  accessing data, 65
  controlling display, 66
Sequence Classes, 20
Sequences, 20
  sample class, 21
Session objects
  getting, 48
Setting Up, 14
Sheets
  Drawing, 108
Simplified representations
  adding items, 361
  creating, 359
  deleting, 359
  items, 361
  extracting information from, 360
  modifying, 360
  retrieving
    geometry, 359
    graphics, 359
    utilities, 362
Sketched features
  create, 246
  create with 2D sections, 247
  creating features with 3D sections, 247
  overview, 246
Smart pointers
  Creo Parametric-related objects, 17
Solids
  accuracy, 156
  coordinate system, 192
  geometry traversal, 198
  getting a solid object, 156
  information, 156
  mass properties, 166
  operations, 156, 354
Splines
  cylindrical spline surface, 452
  description, 253
  representation, 454
  surface, 450
Status
  feature, 215
  layer, 200
Surfaces, 256
  cylindrical spline, 452
  data structures, 444
  evaluating, 258
    area, 258
    evaluating parameters, 258
  fillet
    geometry representation, 449
  general surface of revolution, 447
  NURBS
    geometry representation, 451
  revolved, 257
  ruled, 257, 448
  spline, 450
  traversing, 252
  types, 257
UV parameterization, 257

T

t parameter
description, 253
Table interface
description, 124
Tabulated cylinders, 257
glometry representation, 448
Text, 253
message files, 52
Tolerance, 273
Torii, 257
Torus, 446
Transformations, 191, 194
solid to coordinate system
datum coordinates, 195
solid to screen coordinates, 195
in a drawing, 196
Traversal
of a solid block, 252
of geometry, 252
try-catch-finally block
description, 32

U

UDFs, 220
creating, 221
Union classes, 20
Unions, 20
User’s Guide
browsing with APIWizard, 41
documentation
online, 36
Utilities
sample class, 26
simplified representations, 362
Utilities classes, 26

V

Values

ParamValue, 265
View2D.View2D interface
description, 111
Views
display information, 115
Drawing, 111
getting a view object, 190
list of, 190
operations, 191
retrieving, 190
saving, 191
Visibility, 215
Visit
simplified representations, 360

W

Window coordinate system, 193
Windows, 188
activating, 189
clearing, 189
closing, 189
creating, 188
operations, 189, 356
repainting, 189
Write
to the message window, 54