Allegro, Cadence, and Concept are registered trademarks of Cadence Design Systems, Inc. AutoCAD and AutoDesk Inventor are registered trademarks of Autodesk, Inc. Baan is a registered trademark of Baan Company. CADAM and CATIA are registered trademarks of Dassault Systemes. COACH is a trademark of CADTRAIN, Inc. DOORS is a registered trademark of Telelogic AB. FLEXlm is a registered trademark of GLOBEtroter Software, Inc. Geomagic is a registered trademark of Raindrop Geomagic, Inc. EVERSYNC, GROOVE, GROOVEFEST, GROOVE.NET, GROOVE NETWORKS, iGROOVE, PEERWARE, and the interlocking circles logo are trademarks of Groove Networks, Inc. Helix is a trademark of Microcadam, Inc. HOOPS is a trademark of Tech Soft America, Inc. HP-UX is a registered trademark and Tru64 is a trademark of the Hewlett-Packard Company. I-DEAS, Metaphase, Parasolid, SHERPA, Solid Edge, and Unigraphics are trademarks or registered trademarks of Electronic Data Systems Corporation (EDS). InstallShield is a registered trademark and service mark of InstallShield Software Corporation in the United States and/or other countries. Intel is a registered trademark of Intel Corporation. IRIX is a registered trademark of Silicon Graphics, Inc. MatrixOne is a trademark of MatrixOne, Inc. Mentor Graphics and Board Station are registered trademarks and 3D Design, AMPLE, and Design Manager are trademarks of Mentor Graphics Corporation. MEDUSA and STHENO are trademarks of CAD Schroer GmbH. Microsoft, Microsoft Project, Windows, the Windows logo, Windows NT, Visual Basic, and the Visual Basic logo are registered trademarks of Microsoft Corporation in the United States and/or other countries. Netscape and the Netscape N and Ship's Wheel logos are registered trademarks of Netscape Communications Corporation in the United States and/or other countries. Oracle is a registered trademark of Oracle Corporation. OrbixWeb is a registered trademark of IONA Technologies PLC. PDGS is a registered trademark of Ford Motor Company. RAND is a trademark of RAND Worldwide. Rational Rose is a registered trademark of Rational Software Corporation. RetrievalWare is a registered trademark of Convera Corporation. RosettaNet is a trademark and Partner Interface Process and PIP are registered trademarks of "RosettaNet," a nonprofit organization. SAP and R/3 are registered trademarks of SAP AG Germany. SolidWorks is a registered trademark of SolidWorks Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and in other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Sun, Sun Microsystems, the Sun logo, Solaris, UltraSPARC, Java and all Java based marks, and "The Network is the Computer" are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and in other countries. VisTools is a trademark of Visual Kinematics, Inc. (VKI). VisualCafé is a trademark of WebGain, Inc. WebEx is a trademark of WebEx Communications, Inc.

**Licensed Third-Party Technology Information**

Certain PTC software products contain licensed third-party technology: Rational Rose 2000E is copyrighted software of Rational Software Corporation. RetrievalWare is copyrighted software of Convera Corporation. VisualCafé is copyrighted software of WebGain, Inc. VisTools library is copyrighted software of Visual Kinematics, Inc. (VKI) containing confidential trade secret information belonging to VKI. HOOPS graphics system is a proprietary software product of, and is copyrighted by, Tech Soft America, Inc. G-POST is copyrighted software and a registered trademark of Intercim. VERICUT is copyrighted software and a registered trademark of CGTech. Pro/PLASTIC ADVISOR is powered by Moldflow technology. Moldflow is a registered trademark of Moldflow Corporation. The JPEG image output in the Pro/Web.Publish module is based in part on the work of the independent JPEG Group. DFORMD.DLL is copyrighted software from Compaq Computer Corporation and may not be distributed. METIS, developed by George Karypis and Vipin Kumar at the University of Minnesota, can be researched at [http://www.cs.umn.edu/~karypis/metis](http://www.cs.umn.edu/~karypis/metis). METIS is © 1997 Regents of the University of Minnesota. LightWork Libraries are copyrighted by LightWork Design 1990-2001. Visual Basic for Applications and Internet Explorer is copyrighted software of Microsoft Corporation. Adobe Acrobat Reader is copyrighted software of Adobe Systems. Parasolid © Electronic Data Systems (EDS). Windchill Info*Engine Server contains IBM XML Parser for Java Edition and the IBM Lotus XSL Edition. Pop-up calendar components Copyright © 1998 Netscape Communications Corporation. All Rights Reserved. TECHNOMATIX is copyrighted software and contains proprietary information of Technomatix Technologies Ltd. Apache Server, Tomcat, Xalan, and Xerces are technologies developed by, and are copyrighted software of, the Apache Software Foundation ([http://www.apache.org/](http://www.apache.org/)) - their use is subject to the terms and limitations at: [http://www.apache.org/LICENSE.txt](http://www.apache.org/LICENSE.txt). UnZip (© 1990-2001 Info-ZIP, All Rights Reserved) is provided "AS IS" and WITHOUT WARRANTY OF ANY KIND. For the complete Info-ZIP license see
Gecko and Mozilla components are subject to the Mozilla Public License Version 1.1 at http://www.mozilla.org/MPL/. Software distributed under the MPL is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the MPL for the specific language governing rights and limitations. Technology "Powered by Groove" is provided by Groove Networks, Inc. Technology "Powered by WebEx" is provided by WebEx Communications, Inc. Acrobat Reader is Copyright © 1998 Adobe Systems Inc. Oracle 8i run-time, Copyright © 2000 Oracle Corporation. The Java™ Telnet Applet (StatusPeer.java, TelnetIO.java, TelnetWrapper.java, TimedOutException.java), Copyright © 1996, 97 Mattias L. Jugel, Marcus Meißner, is redistributed under the GNU General Public License. This license is from the original copyright holder and the Applet is provided WITHOUT WARRANTY OF ANY KIND. You may obtain a copy of the source code for the Applet at http://www.mud.de/se/jta (for a charge of no more than the cost of physically performing the source distribution), by sending e-mail to leo@mud.de or marcus@mud.de-you are allowed to choose either distribution method. The source code is likewise provided under the GNU General Public License. GTK+The GIMP Toolkit are licensed under the GNU LPGL. You may obtain a copy of the source code at http://www.gtk.org/, which is likewise provided under the GNU LPGL. zlib software Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.

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), is 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.

Parametric Technology Corporation, 140 Kendrick Street, Needham, MA 02494 USA
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Record</td>
<td>xix</td>
</tr>
<tr>
<td>About This Guide</td>
<td>xxiii</td>
</tr>
<tr>
<td>Related Documentation</td>
<td>xxiii</td>
</tr>
<tr>
<td>Technical Support</td>
<td>xxiii</td>
</tr>
<tr>
<td>Documentation for PTC Products</td>
<td>xxiv</td>
</tr>
<tr>
<td>Comments</td>
<td>xxiv</td>
</tr>
<tr>
<td>Documentation Conventions</td>
<td>xxiv</td>
</tr>
<tr>
<td>DCA Web Architectural Fundamentals</td>
<td>1-1</td>
</tr>
<tr>
<td>DCA’s Contribution to Windchill Implementations</td>
<td>1-2</td>
</tr>
<tr>
<td>Benefits for Development</td>
<td>1-2</td>
</tr>
<tr>
<td>DCA Architectural Fundamentals</td>
<td>1-3</td>
</tr>
<tr>
<td>DCA Web</td>
<td>1-3</td>
</tr>
<tr>
<td>DCA Solutions</td>
<td>1-4</td>
</tr>
<tr>
<td>Relationship of DCA, Info*Engine, and LWC</td>
<td>1-4</td>
</tr>
<tr>
<td>Pages Generated by DCA are Defined in Configuration XML Files</td>
<td>1-6</td>
</tr>
<tr>
<td>Conceptual Elements</td>
<td>1-7</td>
</tr>
<tr>
<td>Web Implementation</td>
<td>1-7</td>
</tr>
<tr>
<td>Static and Dynamic Elements</td>
<td>1-8</td>
</tr>
<tr>
<td>Element Addressing Scheme</td>
<td>1-8</td>
</tr>
<tr>
<td>Multiple Customization Points - Handlers and Renderers</td>
<td>1-9</td>
</tr>
<tr>
<td>DCA Web Demo</td>
<td>2-1</td>
</tr>
<tr>
<td>Setting up Info*Engine</td>
<td>2-2</td>
</tr>
<tr>
<td>Loading Demo Data</td>
<td>2-2</td>
</tr>
<tr>
<td>Simple Frame Containing a Simple Table</td>
<td>2-3</td>
</tr>
<tr>
<td>Composite Frame Containing a Composite Table</td>
<td>2-4</td>
</tr>
<tr>
<td>Composite Frame Containing a Composite Tree</td>
<td>2-5</td>
</tr>
<tr>
<td>Multiple Components</td>
<td>2-6</td>
</tr>
<tr>
<td>Primitive Elements</td>
<td>2-7</td>
</tr>
<tr>
<td>Wizards</td>
<td>2-8</td>
</tr>
<tr>
<td>Pickers</td>
<td>2-8</td>
</tr>
<tr>
<td>Search</td>
<td>2-9</td>
</tr>
<tr>
<td>Building a Sample Application with DCA</td>
<td>2-10</td>
</tr>
</tbody>
</table>
Referencing a Frame ......................................................................................................... 4-11
Conditional Configuration Expression if.xxx, ifnot.xxx, and if ...................................................4-14
Reloading Configuration Files ..............................................................................................4-14

Properties ....................................................................................................................... 5-1
Property Element ........................................................................................................... 5-2
Property Handler .............................................................................................................. 5-2
Property Renderer ............................................................................................................ 5-3
Required Property ............................................................................................................ 5-4
Current Supported Data Types in DCA Web ........................................................................ 5-4
  java.lang.Boolean ........................................................................................................ 5-4
  java.lang.Date ............................................................................................................... 5-5
  Enumerated Value ......................................................................................................... 5-8
  Hyper Link .................................................................................................................... 5-9
  java.lang.Long ............................................................................................................. 5-9
  java.lang.Double ......................................................................................................... 5-9
  java.lang.String ......................................................................................................... 5-10
Customization that Effects Property Rendering ................................................................... 5-10
  Modify Existing Property Renderer ............................................................................... 5-11
  Extend Existing Property Handler .................................................................................. 5-11
  Create Completely New Property Handler ..................................................................... 5-12
Implement Property Model .................................................................................................. 5-13
Modifying Property Configuration to Use New Renderer or Handler .................................. 5-14
  Writing JSP Renderers .................................................................................................. 5-14

Actions ......................................................................................................................... 6-1
Action Elements ............................................................................................................. 6-2
Action Handlers ............................................................................................................... 6-2
"Simple" Approach for Action Handlers .............................................................................. 6-4
Action Doers vs. Client Doers ............................................................................................. 6-5
Action Handler Types ....................................................................................................... 6-6
  In Place ........................................................................................................................ 6-6
  Go To .............................................................................................................................. 6-6
  New Frame .................................................................................................................... 6-7
  New Temporary Frame ................................................................................................. 6-8
Action List ....................................................................................................................... 6-9
  Active Action List ......................................................................................................... 6-9
  Passive Action List ....................................................................................................... 6-9

Data Acquisition and Manipulation - Doers ...................................................................... 7-1
Useful Glossary Terms ..................................................................................................... 7-2
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Handler</td>
<td>19-4</td>
</tr>
<tr>
<td>Render Handler</td>
<td>19-4</td>
</tr>
<tr>
<td>Element Address</td>
<td>19-5</td>
</tr>
<tr>
<td>Static Element Address</td>
<td>19-5</td>
</tr>
<tr>
<td>Address of a Frame</td>
<td>19-6</td>
</tr>
<tr>
<td>Dynamic Element Address</td>
<td>19-7</td>
</tr>
<tr>
<td>Example of a Long External Form</td>
<td>19-7</td>
</tr>
<tr>
<td>Client Server Communications</td>
<td>19-8</td>
</tr>
<tr>
<td>Rendering</td>
<td>19-8</td>
</tr>
<tr>
<td>Java-Based Renderers</td>
<td>19-9</td>
</tr>
<tr>
<td>Switching Between Java-based and JSP-based Renderers</td>
<td>19-10</td>
</tr>
<tr>
<td>Element Events</td>
<td>19-10</td>
</tr>
<tr>
<td>Conditional Drawing</td>
<td>19-11</td>
</tr>
<tr>
<td>com.ptc.core.ca.co.client.element.CoDrawController</td>
<td>19-11</td>
</tr>
<tr>
<td>Shadow</td>
<td>19-12</td>
</tr>
<tr>
<td>Performance Tuning</td>
<td>19-14</td>
</tr>
<tr>
<td>Understanding Exceptions</td>
<td>19-15</td>
</tr>
<tr>
<td>Utilities</td>
<td>20-1</td>
</tr>
<tr>
<td>Address Utilities</td>
<td>20-2</td>
</tr>
<tr>
<td>Action Utilities</td>
<td>20-2</td>
</tr>
<tr>
<td>Element Utilities</td>
<td>20-3</td>
</tr>
<tr>
<td>visitDown</td>
<td>20-4</td>
</tr>
<tr>
<td>getIeElementOfInstanceHolder</td>
<td>20-7</td>
</tr>
<tr>
<td>getIeElement</td>
<td>20-7</td>
</tr>
<tr>
<td>getIeElementOfInstanceHolder</td>
<td>20-7</td>
</tr>
<tr>
<td>getModel</td>
<td>20-7</td>
</tr>
<tr>
<td>Option Utilities</td>
<td>20-8</td>
</tr>
<tr>
<td>Environment</td>
<td>21-1</td>
</tr>
<tr>
<td>Introduction</td>
<td>21-2</td>
</tr>
<tr>
<td>DCA Web Environment</td>
<td>21-3</td>
</tr>
<tr>
<td>Clipboard</td>
<td>21-4</td>
</tr>
<tr>
<td>Clipboard Mode</td>
<td>21-5</td>
</tr>
<tr>
<td>Clipboard Reason</td>
<td>21-5</td>
</tr>
<tr>
<td>Clipboard Actions</td>
<td>21-6</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Content</td>
<td>22-1</td>
</tr>
<tr>
<td>Listing Content Items</td>
<td>22-2</td>
</tr>
<tr>
<td>Link to Primary Content</td>
<td>22-4</td>
</tr>
<tr>
<td>Property Based</td>
<td>22-4</td>
</tr>
<tr>
<td>Hyperlink Based</td>
<td>22-5</td>
</tr>
<tr>
<td>Download Applet</td>
<td>22-5</td>
</tr>
<tr>
<td>Doers</td>
<td>22-6</td>
</tr>
<tr>
<td>Upload</td>
<td>22-7</td>
</tr>
<tr>
<td>Uploading from a List of Content Items</td>
<td>22-8</td>
</tr>
<tr>
<td>Upload from Create Content Holder Wizard</td>
<td>22-10</td>
</tr>
<tr>
<td>Upload from Edit Properties Wizard</td>
<td>22-13</td>
</tr>
<tr>
<td>Links and Associations</td>
<td>23-1</td>
</tr>
<tr>
<td>Edit Links Functionality</td>
<td>23-2</td>
</tr>
<tr>
<td>How to Configure</td>
<td>23-2</td>
</tr>
<tr>
<td>Referenced Elements</td>
<td>23-3</td>
</tr>
<tr>
<td>Picker Doer</td>
<td>23-4</td>
</tr>
<tr>
<td>Recognizing Link Changes</td>
<td>23-5</td>
</tr>
<tr>
<td>Action on Related Object</td>
<td>23-5</td>
</tr>
<tr>
<td>Hyperlink to Related Object</td>
<td>23-7</td>
</tr>
<tr>
<td>Preferences</td>
<td>24-1</td>
</tr>
<tr>
<td>Internationalization and Localization</td>
<td>25-1</td>
</tr>
<tr>
<td>Internationalization of Configuration Specifications</td>
<td>25-2</td>
</tr>
<tr>
<td>Manual Creation of Entries in .rbInfo Files</td>
<td>25-3</td>
</tr>
<tr>
<td>Step 1</td>
<td>25-3</td>
</tr>
<tr>
<td>Step 2</td>
<td>25-3</td>
</tr>
<tr>
<td>Automatic Creation of Entries in .rbInfo Files</td>
<td>25-4</td>
</tr>
<tr>
<td>Step 1</td>
<td>25-4</td>
</tr>
<tr>
<td>Step 2</td>
<td>25-4</td>
</tr>
<tr>
<td>Step 3</td>
<td>25-5</td>
</tr>
<tr>
<td>Conflict Resolution</td>
<td>25-6</td>
</tr>
<tr>
<td>Comments</td>
<td>25-6</td>
</tr>
<tr>
<td>Configuration Specification Reference</td>
<td>A-1</td>
</tr>
<tr>
<td>Disclaimer</td>
<td>A-2</td>
</tr>
<tr>
<td>Configuration Syntax</td>
<td>A-3</td>
</tr>
<tr>
<td>Use of Identifiers</td>
<td>A-3</td>
</tr>
<tr>
<td>Context</td>
<td>A-3</td>
</tr>
<tr>
<td>Conditional Statements</td>
<td>A-6</td>
</tr>
<tr>
<td>Data Sources Available In Configuration Specifications</td>
<td>A-9</td>
</tr>
<tr>
<td>Configuration Elements</td>
<td>A-35</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Action</td>
<td>A-36</td>
</tr>
<tr>
<td>ActionDoer</td>
<td>A-38</td>
</tr>
<tr>
<td>ActionHandler</td>
<td>A-39</td>
</tr>
<tr>
<td>ActionList</td>
<td>A-40</td>
</tr>
<tr>
<td>Anchor</td>
<td>A-42</td>
</tr>
<tr>
<td>Application</td>
<td>A-43</td>
</tr>
<tr>
<td>Check</td>
<td>A-44</td>
</tr>
<tr>
<td>CheckTaskParam</td>
<td>A-45</td>
</tr>
<tr>
<td>ClientDoer</td>
<td>A-46</td>
</tr>
<tr>
<td>Comparator</td>
<td>A-47</td>
</tr>
<tr>
<td>CompositeColumn</td>
<td>A-48</td>
</tr>
<tr>
<td>CompositeFrame</td>
<td>A-50</td>
</tr>
<tr>
<td>CompositeLayout</td>
<td>A-52</td>
</tr>
<tr>
<td>CompositeNavBar</td>
<td>A-53</td>
</tr>
<tr>
<td>CompositePropertyPanel</td>
<td>A-55</td>
</tr>
<tr>
<td>CompositeTable</td>
<td>A-57</td>
</tr>
<tr>
<td>CompositeTree</td>
<td>A-59</td>
</tr>
<tr>
<td>Connector</td>
<td>A-60</td>
</tr>
<tr>
<td>Context</td>
<td>A-62</td>
</tr>
<tr>
<td>Copy</td>
<td>A-63</td>
</tr>
<tr>
<td>DataDrivenEnumerator</td>
<td>A-65</td>
</tr>
<tr>
<td>DrawHandler</td>
<td>A-68</td>
</tr>
<tr>
<td>Else</td>
<td>A-69</td>
</tr>
<tr>
<td>Elsif</td>
<td>A-70</td>
</tr>
<tr>
<td>EmbeddedURLContent</td>
<td>A-71</td>
</tr>
<tr>
<td>Enumerator</td>
<td>A-73</td>
</tr>
<tr>
<td>Extend</td>
<td>A-75</td>
</tr>
<tr>
<td>ElementGroup</td>
<td>A-78</td>
</tr>
<tr>
<td>Footer</td>
<td>A-79</td>
</tr>
<tr>
<td>For</td>
<td>A-80</td>
</tr>
<tr>
<td>FrameHeader</td>
<td>A-81</td>
</tr>
</tbody>
</table>
SelectAllPrompt ................................................................. A-136
Separator ............................................................................... A-137
Shadow ................................................................................ A-138
SimpleColumn ..................................................................... A-139
SimpleFrame ........................................................................ A-140
SimpleLayout ....................................................................... A-142
SimpleNavBar ....................................................................... A-143
SimplePropertyPanel .......................................................... A-144
SimpleTable .......................................................................... A-145
Synonym ............................................................................... A-147
Task ..................................................................................... A-149
TaskElement ......................................................................... A-151
TaskGroup ........................................................................... A-153
TaskParameter ....................................................................... A-154
TaskOutParameter ................................................................. A-156
Template ............................................................................... A-157
ToolTip ................................................................................ A-158
TopLayout ............................................................................ A-159
Trigger ................................................................................ A-160
Type ..................................................................................... A-161
UpdateLocation ..................................................................... A-162
Validator .............................................................................. A-163
Wizard ................................................................................ A-164
WizardActions ...................................................................... A-165
WizardStepActions ................................................................. A-166
WizardStepElements ............................................................. A-167
Deprecated Configuration Elements ...................................... A-167
Layout (deprecated) ................................................................. A-169
NavBar (deprecated) ............................................................... A-171
PropertyPanel (deprecated) ..................................................... A-173
Deprecated Attributes .......................................................... A-175
frameType of HyperLink ........................................................ A-175
computedValue of TaskParameter ............................................ A-175
Obsolete Configuration Elements .......................................... A-176
AppInstallationStatusDeterminator ....................................... A-176
ApplicationSet ..................................................................... A-177
Obsolete Handlers and Renderers .......................................... A-179
Ratio property handler and renderer ....................................... A-179
Upgrades Requiring Manual Changes in Configuration Specifications .................................. A-180
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>A-180</td>
</tr>
<tr>
<td>Doers</td>
<td>A-183</td>
</tr>
<tr>
<td>Functions</td>
<td>A-188</td>
</tr>
<tr>
<td>Debugging DCA</td>
<td>B-1</td>
</tr>
<tr>
<td>Web-Based Configuration Viewer</td>
<td>B-2</td>
</tr>
<tr>
<td>Command Line-Based Configuration Viewer</td>
<td>B-2</td>
</tr>
<tr>
<td>Top-Level Element Viewer</td>
<td>B-2</td>
</tr>
<tr>
<td>repository.csv</td>
<td>B-3</td>
</tr>
<tr>
<td>repository_xref.csv</td>
<td>B-3</td>
</tr>
<tr>
<td>DCA logging</td>
<td>B-4</td>
</tr>
<tr>
<td>General Logging Functionality</td>
<td>B-4</td>
</tr>
<tr>
<td>Turning logging on and off</td>
<td>B-5</td>
</tr>
<tr>
<td>Selective logging</td>
<td>B-5</td>
</tr>
<tr>
<td>Evaluation of Logging Rules</td>
<td>B-6</td>
</tr>
<tr>
<td>Special instructions</td>
<td>B-7</td>
</tr>
<tr>
<td>Tracing Preparation of Configuration</td>
<td>B-8</td>
</tr>
<tr>
<td>Tracing Reference Resolution</td>
<td>B-8</td>
</tr>
<tr>
<td>Using the HTML Template Processor Bridge</td>
<td>C-1</td>
</tr>
<tr>
<td>Configuring the Bridge</td>
<td>C-2</td>
</tr>
<tr>
<td>Sample Mapping Entries</td>
<td>C-3</td>
</tr>
<tr>
<td>Trail File Support</td>
<td>D-1</td>
</tr>
<tr>
<td>Introduction</td>
<td>D-2</td>
</tr>
<tr>
<td>Use Cases for Trail Support</td>
<td>D-2</td>
</tr>
<tr>
<td>Limitations</td>
<td>D-3</td>
</tr>
<tr>
<td>Controlling Trail File Support</td>
<td>D-3</td>
</tr>
<tr>
<td>ID Structure</td>
<td>D-4</td>
</tr>
<tr>
<td>Address Of The Frame</td>
<td>D-5</td>
</tr>
<tr>
<td>Static Element Address</td>
<td>D-5</td>
</tr>
<tr>
<td>Dynamic Element Address</td>
<td>D-5</td>
</tr>
<tr>
<td>Enabling Trail File Support by a Renderer</td>
<td>D-8</td>
</tr>
</tbody>
</table>
## Change Record

### Table 1 Changes for Windchill 7.0

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
</tr>
</thead>
</table>
| Appendix A, Configuration Specification Reference | The following configuration elements have been deprecated:  
- Layout  
- NavBar  
- PropertyPanel  
Their renderers have also been deprecated unless they are shared by active elements.  
Both deprecated elements and their renderers are fully supported unless explicitly stated otherwise.  
The descriptions of the deprecated elements are included in Deprecated Configuration Elements on page A-167, and have not been changed to reflect deprecations of sub-elements or attributes. |
| Appendix A, Configuration Specification Reference | The following attributes have been deprecated:  
- `frameType` of HyperLink  
  Replaced by `window` attribute.  
- `computedValue` of TaskParameter  
  Replaced by `function` attribute. Requires writing a function. |
### Change

<table>
<thead>
<tr>
<th>Appendix A, Configuration Specification Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following configuration elements are obsolete:</td>
<td>The following configuration elements are obsolete:</td>
</tr>
<tr>
<td>• AppInstallationStatusDeterminator</td>
<td>• AppInstallationStatusDeterminator</td>
</tr>
<tr>
<td>• ApplicationSet</td>
<td>• ApplicationSet</td>
</tr>
<tr>
<td>By making these configuration elements obsolete, no migration problems are expected, or are of severity less critical than the inconvenience of maintaining them.</td>
<td>By making these configuration elements obsolete, no migration problems are expected, or are of severity less critical than the inconvenience of maintaining them.</td>
</tr>
<tr>
<td>The descriptions of the obsolete elements are included in Obsolete Configuration Elements on page A-176.</td>
<td>The descriptions of the obsolete elements are included in Obsolete Configuration Elements on page A-176.</td>
</tr>
</tbody>
</table>

### Change

<table>
<thead>
<tr>
<th>Appendix A, Configuration Specification Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following handler and renderers are obsolete:</td>
<td>The following handler and renderers are obsolete:</td>
</tr>
<tr>
<td>• Property Handler</td>
<td>• Property Handler</td>
</tr>
<tr>
<td>• Java Renderer</td>
<td>• Java Renderer</td>
</tr>
<tr>
<td>com.ptc.core.ca.web.client.property.WebRatioRendererHandler</td>
<td>com.ptc.core.ca.web.client.property.WebRatioRendererHandler</td>
</tr>
<tr>
<td>• JSP Renderer</td>
<td>• JSP Renderer</td>
</tr>
<tr>
<td>/wtcore/jsp/com/ptc/core/ca/web/property/ratio/ratio.jsp</td>
<td>/wtcore/jsp/com/ptc/core/ca/web/property/ratio/ratio.jsp</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Appendix A, Configuration Specification Reference</td>
<td>The following elements, doers, and functions have upgrades requiring manual changes in configuration specifications:</td>
</tr>
<tr>
<td></td>
<td>• ActionList</td>
</tr>
<tr>
<td></td>
<td>• SimpleTable</td>
</tr>
<tr>
<td></td>
<td>• CompositeTable</td>
</tr>
<tr>
<td></td>
<td>• CompositeTree</td>
</tr>
<tr>
<td></td>
<td>• AppInstallation_Status_Determinator</td>
</tr>
<tr>
<td></td>
<td>• ApplicationSet</td>
</tr>
<tr>
<td></td>
<td>• DataDrivenEnumerator</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.clientpicker.Con ullModelDoer</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.client.search.CoPageBackwardActionDoer</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.client.search.CoPageForwardActionDoer</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.client.search.CoSearchActionClientDoer</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.client.search.CoSearchModelTaskDoer</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.client.doer.CoDef aultSimpleActionTaskDoer</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.client.doer.CoDef aultSimpleModelTaskDoer</td>
</tr>
<tr>
<td></td>
<td>• com.ptc.core.ca.co.client.doer.function.impl.CoSimpleFunction_GetUidForNearestInstance_As_Model Param</td>
</tr>
<tr>
<td></td>
<td>These required manual upgrades are described in Upgrades Requiring Manual Changes in Configuration Specifications on page A-180.</td>
</tr>
</tbody>
</table>
About This Guide

The Windchill Client Technology Guide describes the new application development framework, Dynamic Client Architecture (DCA) and how it can be used with the Windchill software suite. This guide provides an overview of functionality, a DCA Web demo, and a Hello World Example. Additionally, there are configuration settings and a configuration specification reference appendix. Developers are expected to be familiar with Info Engine, LWC and XML.

Related Documentation

The following documentation may be helpful:

- *The Windchill Customizer’s Guide*
- *The Windchill Installation and Configuration Guide*
- *The Windchill Application Developer’s Guide*

If books are not installed on your system, see your system administrator.

Technical Support

Contact PTC Technical Support via the PTC Web site, phone, fax, or e-mail if you encounter problems using Windchill.

For complete details, refer to Contacting Technical Support in the *PTC Customer Service Guide* enclosed with your shipment. This guide can also be found under the Support Bulletins section of the PTC Web site at:

[http://www.ptc.com/support/index.htm](http://www.ptc.com/support/index.htm)

The PTC Web site also provides a search facility that allows you to locate Technical Support technical documentation of particular interest. To access this page, use the following link:

[http://www.ptc.com/support/support.htm](http://www.ptc.com/support/support.htm)

You must have a Service Contract Number (SCN) before you can receive technical support. If you do not have an SCN, contact PTC License Management using the instructions found in your *PTC Customer Service Guide* under Contacting License Management.
Documentation for PTC Products

PTC provides documentation in the following forms:

- Help topics
- PDF books

All books are available in HTML and PDF formats, or both, on product CDs. To view HTML books, use your Internet browser. To view and print PDF books, you must have the Adobe Acrobat Reader installed.

All Windchill documentation is included on the CD for the application. In addition, books updated after release (for example, to support a hardware platform certification) are available from the Reference Documents section of the PTC Web site at the following URL:

http://www.ptc.com/cs/doc/reference/

Comments

PTC welcomes your suggestions and comments on its documentation—send comments to the following address:

documentation@ptc.com

Please include the name of the application and its release number with your comments. For online books, provide the book title.

Documentation Conventions

Windchill documentation uses the following conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Item</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Names of elements in the user interface such as buttons, menu paths, and dialog box titles.</td>
<td>Click OK.</td>
</tr>
<tr>
<td></td>
<td>Required elements and keywords or characters in syntax formats.</td>
<td>Select File &gt; Save.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>License File dialog box</td>
</tr>
<tr>
<td></td>
<td></td>
<td>create_&lt;tablename&gt;.sql</td>
</tr>
<tr>
<td><strong>Italic</strong></td>
<td>Variable and user-defined elements in syntax formats. Angle brackets (&lt; and &gt;) enclose individual elements.</td>
<td>create_&lt;tablename&gt;.sql</td>
</tr>
<tr>
<td><strong>Monospace</strong></td>
<td>Examples Messages</td>
<td>JavaGen &quot;wt.doc.*&quot; F true Processing completed.</td>
</tr>
<tr>
<td>&quot;Quotation marks&quot;</td>
<td>Strings</td>
<td>The string &quot;UsrSCM&quot; . . .</td>
</tr>
<tr>
<td>Convention</td>
<td>Item</td>
<td>Example</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><img src="image" alt="CAUTION" /></td>
<td>The CAUTION symbol indicates potentially unsafe situations which may result in minor injury, machine damage or downtime, or corruption or loss of software or data.</td>
<td>When you add a value to an enumerated type (for example, by adding a role in the RolesRB.java resource file), removing that value can result in a serious runtime error. Do not remove a role unless you are certain there is no reference to it within the system.</td>
</tr>
</tbody>
</table>
Dynamic Client Architecture (DCA) is PTC’s new application development framework for use with the Windchill software suite.

DCA combines the rich feature set of Windchill and the integration capabilities of Info*Engine to create a data acquisition layer that can easily combine Windchill data with the customer’s legacy data. This allows customers to fully leverage all data in their environment, and not just the data stored or transferred to Windchill.

On top of that, DCA adds an application description language (ADL), a conceptual layer that processes the ADL, a set of data handlers that bind the ADL to the data acquisition layer, and a set of renderers that produce user interface pages for the application.

While the current release of DCA supports only HTML production, DCA allows for more presentation layers to be added in the future.

**Topic** | **Page**
---|---
DCA’s Contribution to Windchill Implementations | 1-2
DCA Architectural Fundamentals | 1-3
DCA Web | 1-3
DCA Solutions | 1-4
Relationship of DCA, Info*Engine, and LWC | 1-4
Pages Generated by DCA are Defined in Configuration XML Files | 1-6
Conceptual Elements | 1-7
Web Implementation | 1-7
Static and Dynamic Elements | 1-8
Element Addressing Scheme | 1-8
Multiple Customization Points - Handlers and Renderers | 1-9
DCA’s Contribution to Windchill Implementations

DCA brings to Windchill implementations a consistent, usable, and highly customizable application development environment. Instead of every development project having to write new Java code to produce their specific user interfaces, DCA allows them to describe the application behavior in conceptual terms.

The description:

- Includes what objects are available, the attributes that can be accessed on those objects, and the actions that can be performed on those objects.
- Defines what kind of artifacts are needed to produce a user interface, including the menu definitions that wire together all the components into a coherent application.
- Includes information to include or exclude any of these elements based on the application or role of the user within that application.

The ADL is a set of XML files that describe the DCA components to be wired together to create the desired application. The advantages of using XML to describe the application is that, no matter how many table listings you create in the description language, they will all have a consistent user model because there is only one table-generation Java component in the framework. This means that it will be easier to deploy DCA-based applications because of the reduced training and support efforts needed to familiarize the users with the software. The training that is done can focus on the business issues instead of user-interface differences between portions of the application.

Benefits for Development

From a development effort standpoint, it will be easier to create an application by wiring the components together than to create new components for each permutation of objects in the application. Once the ADL has been created for a particular base object, a similar object (whose server behaviors already are accessible through Info*Engine) can have the same base behavior deployed in a matter of hours, sometimes in just a few minutes. This results in considerable savings during the customization phase of the deployment when it is very typical to be adding a few attributes to each of the base PTC objects.

From a customization standpoint, DCA and the ADL can make it possible for a subject matter expert to prototype the application using DCA instead of using paper mock-ups, bringing in programmers to write code, or both.

From an extensibility standpoint, DCA provides several Java component patterns which can be used to extend, replace, or augment the base component set provided by PTC. A common extension is to write a renderer that converts textual status information into an icon for display to the user. Once this new renderer is built (and it can be as little as 50 lines of Java code), a single line change in the ADL...
causes the renderer to be used for every instance of a given attribute throughout the application suite.

The ADL also makes use of a number of inheritance features to allow changes to propagate through several objects as the result of a single change. This makes it easier to upgrade your system from revision to revision because changes that PTC makes to base objects automatically trickle down into all objects inheriting features from the base object definitions.

From a Windchill solution standpoint, DCA is key to integrating or interoperating the various point solutions. As the solutions become DCA-based, it will be much easier to provide a version of Windchill PDMLink with a couple extra pages defined to control Windchill ProjectLink projects or provide a Windchill ProjectLink administrator with a search that allows them to search not only the projects they are working in, but Windchill PDMLink repositories as well.

## DCA Architectural Fundamentals

DCA is an environment neutral application development framework. DCA stands for Dynamic Client Architecture. Its important features are:

- An XML-based UI configuration repository subsystem that defines
  - Definition of tables, trees, forms, wizards and the widgets needed in those definitions
  - Binding to the data access and manipulation subsystem
  - A set of defined patterns for interaction between the presentations elements and the data access system
  - A set of pre-defined handlers for manipulating the attributes
  - A set of pre-defined renderers for producing consistent user model behavior in the HTML UI.

- Multiple customization points that allow the introduction of new behavior to extend the rendering of a particular application.

- A methodology for implementing applications that are tailored to the specific type of user operating it.

## DCA Web

DCA WEB is a web-application development framework based on DCA and JSP architectures. Its important features are:

- A set of pre-defined JSP-based renderers for producing consistent user model behavior in the HTML UI.

- A set of java-based renderers (render handlers) optimized for performance.
DCA Solutions

DCA is designed to alter the customization paradigm for Windchill from one that requires extensive Java programming to one that merely requires configuration. This transition should dramatically reduce the time needed to deploy a Windchill environment. It should also make Windchill customization possible by subject matter experts, not just programmers.

DCA is also designed to significantly improve the user experience with Windchill. Instead of allowing each individual developer or customizer to choose their own widgets and user models for each page developed, DCA provides a set of best practice patterns that result in consistency of user model across the entire application suite. This will reduce the level of training necessary, allowing for faster deployment across a wide variety of users. Users will quickly become used to the user model and will be able to move from module to module and solution to solution with little or no effort. Once they have experienced any Windchill DCA-based solution, they will know how to interact with any of them.

By developing applications using DCA WEB, this facilitates:

- Reuse of core components in a consistent manner by both R&D application developers as well as a more consistent and rapid method for customization of the applications in the field.
- The ability of the configuration system to bind to the definition of new types and attributes defined in the Type Mechanism (LWC) allows rapid customization of the applications at the customer site.
- Rapid customization beyond understanding new types is also supported. Introducing new customer site specific behavior either in the operations available or the how components are displayed in existing applications is also possible.
- A significant reduction in time to produce the user interface on new custom objects, especially those extended from existing objects like Documents and Parts.

Relationship of DCA, Info*Engine, and LWC

DCA is a server-neutral toolkit for producing user interfaces. It does not require Dynamic Typing, but is necessary in order to rapidly produce user interfaces on top of the objects created by Dynamic Typing.

Dynamic Typing allows a Windchill user to create new versions of existing objects by adding attributes to the definition. These objects are available for use by DCA via Info*Engine immediately after their configuration via the Type Manager applet and defining mapping of their logical attributes names. DCA mines the attribute data from those objects (Info*Engine elements) and produces the platform appropriate output to allow the user to interact with the system.
For example, DCA produces a page with fields that the user fills in to specify
search criteria. When the user presses the submit button, those attributes are used
to create a query to pass to Info*Engine. Info*Engine responds with a set of
com.infoengine.object.factory.Element objects that meet the specified
criteria. DCA then extracts the requested set of attributes from each Info*Engine
element and turns them into a table. The table is populated with the commands
specified in the configuration system, tailored to the object type of each row in the
table. The user then selects an action, which gets dispatched to retrieve the object
and produce pages that allow the user to view or modify the object's properties.
All Windchill objects can be manipulated with this new type-aware API and as
such, the Info*Engine elements are used as the primary hand off between the data
acquisition and presentation layers.

DCA defines a clear distinction between the presentation and data acquisition
layers. A DCA form will never know from which server a particular attribute was
retrieved - only the data acquisition layer knows that. DCA always receives its
data in a standard container object called an Element. All DCA ever sees is an
attribute coming from the Element, like a string. The user interface is free to
implement the proper user model for the user to interact with that type of attribute.
The data acquisition layer will handle the fact that it is a string stored in a
WTDocument object in Windchill. The advantage DCA brings is that all strings,
regardless of where they come from, get treated consistently and that once you
have defined an implementation for a string, you can reuse it in any other
application defined in DCA regardless of what server is holding that attribute.
This will bring a high consistency of user model to your applications, making it
easier to deploy them into a very large user population, plus it makes integrations
much easier because the data acquisition layer always produces and consumes the
same container regardless of the true source of the data.
Pages Generated by DCA are Defined in Configuration XML Files

All the HTML pages generated by DCA framework are configured by definitions in XML files. By default the XML files are read from a particular directory tree, currently, $(wt.home)/conf/dca. All XML files found from that point down the directory tree are processed to find DCA configuration information.
**Conceptual Elements**

Packages: com.ptc.core.ca.co.*

The Dynamic Client Architecture framework is designed to support other presentation environments (JFC, ...) in addition to HTML. The conceptual level containing GUI element classes is designed to support this concept. It describes the patterns used in typical applications in such a way as they are neutral to the presentation platform on which the user interface is actually deployed. As an example, it describes the set of attributes that need to be collected in order to successfully create a new part object.

- Conceptual elements are abstract classes that are intended to be presentation environment independent. They encapsulate all the necessary behavior of a UI element.
- They represent all necessary GUI elements to present a UI but don't have the environment specific behavior.
- From an MVC analog, the conceptual elements defined much of the model and control behaviors of an element but not the view portion.
- To support additional presentation environments, this elements are extend to concrete environment specific versions of the element classes.

**Web Implementation**

Package: com.ptc.core.ca.web.*

The Web classes extend the conceptual element classes to create a concrete implementation of a presentation environment - DCA WEB. For example, this layer takes the requests for a name string and number string from the conceptual layer and turns them into an HTML format where the necessary labels and text fields exist to capture the information from the user.

- Concrete implementation to extend Conceptual classes to support Web (HTML) presentation environment.
- Web elements are responsible for the drawing/rendering of elements.
- In Web presentation environment, this means generating correct html for each element.
- This includes drawing sub elements. This is not done in the conceptual level behavior but is the responsibility of the web implementation to determine how to draw its sub elements. Of course, this is normally just telling the sub element to draw/render itself.
- The Web implementation is also responsible for handling input specific to presentation environment. In the case of WEB, the information is transmitted using GET or POST requests. The WEB implementation is responsible of updating the models associated to elements as necessary by extracting the
data from the environment-specific data object. This returns the user supplied input back to the Info*Engine elements for transmission back to the server.

- An additional environment specific behavior of Web element. For example, the WebElement interface defines two additional behaviors specific to the Web environment: Specification of configured jsp renderer and specialized handling of draw errors.

**Static and Dynamic Elements**

- Each page is represented by a hierarchy of elements. The page is correctly generated by traversing this hierarchy and telling the elements to draw themselves.

- Static elements correspond to items defined in configuration, for example: Frames and tables. Static elements are created when a configuration is read and first drawn. As the hierarchy is traversed, static elements are created from the corresponding configuration information.

- Dynamic elements correspond to items acquired from data acquisition. The doer classes acquire the business objects and added dynamic elements to the element hierarchy so the correct display is created. In the case of a table, all the rows are dynamic elements with each row holding a single business object. Dynamic elements implement the CoDynamic interface. An element is static if it does not implement CoDynamic interface.

- Currently implemented dynamic elements are the Row element and TreeNode element.

**Element Addressing Scheme**

Every element has an address. The addressing scheme is crucial to enable different elements to work effectively as a single application.

- The addressing enables an action when it is run to access the correct data object.

- When an action is run on the row of a table (checkout for example), it is the address the allows correct element to be update and the business object associated with the row in the table to accessed.

- The address to a sub element includes the address of its parents. Address defines where element is in the element hierarchy.

- Both static and dynamic elements have addresses.
Multiple Customization Points - Handlers and Renderers

The Handlers in DCA handle the various aspects of an element's behavior.

<table>
<thead>
<tr>
<th>Handler Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw handler</td>
<td>Completely replaces the behavior of the element.</td>
</tr>
<tr>
<td>model handler</td>
<td>Replaces the data acquisition and manipulation behavior of the element.</td>
</tr>
<tr>
<td>render handler</td>
<td>Replaces rendering behavior of the elements. In DCA web the default behavior is to invoke a Java bean-based renderer.</td>
</tr>
<tr>
<td>property handler</td>
<td>Specialized model handler for property elements that insure the proper protocol for data transport from the data access container to the user interface and back again are followed. Additional data types can be created by creating new Property Handler classes and then calling them out in the configuration files.</td>
</tr>
<tr>
<td>action handler</td>
<td>Specialized model handler for action elements with an ability to perform an operation. The action handlers implement the various user model patterns - run a command and refresh the page, execute a form, extract the data and run a command and refresh the page, run a popup and refresh the page when it is dismissed, etc.</td>
</tr>
</tbody>
</table>

Two kinds of handlers are defined in DCA - property handlers and action handlers.

The Renderers in DCA Web are JSP pages themselves. By creating additional pages, you can add new renderers to the environment. So, if you want to implement a widget that combines a text field, a drop-down menu, and a micro-applet for handling drag and drop, you would merely create the new page that responds to the request with the appropriate HTML result.

To improve performance DCA Web provides an ability to specify java-based renderers (render handlers in DCA terminology). DCA Web does not use servlet dispatching mechanism to invoke them thus greatly increasing rendering performance. Developers are expected to initially develop jsp-based renderers due to the ease of jsp-based development with subsequent conversion of most frequently used ones to java-based renderers.
# DCA Web Demo

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up Info*Engine</td>
<td>2-2</td>
</tr>
<tr>
<td>Loading Demo Data</td>
<td>2-2</td>
</tr>
<tr>
<td>Simple Frame Containing a Simple Table</td>
<td>2-3</td>
</tr>
<tr>
<td>Composite Frame Containing a Composite Table</td>
<td>2-4</td>
</tr>
<tr>
<td>Composite Frame Containing a Composite Tree</td>
<td>2-5</td>
</tr>
<tr>
<td>Multiple Components</td>
<td>2-6</td>
</tr>
<tr>
<td>Primitive Elements</td>
<td>2-7</td>
</tr>
<tr>
<td>Wizards</td>
<td>2-8</td>
</tr>
<tr>
<td>Pickers</td>
<td>2-8</td>
</tr>
<tr>
<td>Search</td>
<td>2-9</td>
</tr>
<tr>
<td>Building a Sample Application with DCA</td>
<td>2-10</td>
</tr>
<tr>
<td>Search Patterns</td>
<td>2-45</td>
</tr>
<tr>
<td>Sample XML Configuration File</td>
<td>2-50</td>
</tr>
</tbody>
</table>
Setting up Info*Engine

The DCA Web specific installation process for an application or solution is detailed below.

1. Use PTC.Setup or a similar installation tool to install Info*Engine and Windchill Foundation from the CDs to the file system of the computer on which the product is being installed. This will load class files and Info*Engine task files into the proper file system directories from which they will be executed later.

2. Set up the following property to reflect location of the DCA Web demo configuration files:

   `com.ptc.core.ca.co.common.config.directory=
    codebase/registry/dca`

3. Execute the following commands from Windchill command prompt to load definitions of DCA demo I*E task to LDAP:

   ```
   cd ${wt.home}\codebase\com\ptc\refimpl\co\installation
   install.bat
   ```

Loading Demo Data

To display the DCA Web demo, load the demo files using the following command:

```
java wt.load.LoadFromFile -d <file name>
```

The following is a list of files containing DCA demo data:

- `$(wt.home)/loadFiles/dca/folders.xml` (load this one first)
- `$(wt.home)/loadFiles/dca/docs.xml`
- `$(wt.home)/loadFiles/dca/parts.xml`
- `$(wt.home)/loadFiles/dca/coolparts.xml`

You have to provide an administrator’s user name and password to load demo data.
Simple Frame Containing a Simple Table

Use the following URL and log in as an administrator:


On the screen you see a frame containing:

- a navigation bar
- a property panel displaying the current folder name
- a simple table presenting folder content

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Waiting</th>
<th>Status</th>
<th>Created</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checked Out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>Publications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>Marketing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>ECadSpec1</td>
<td>871164261</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:20:49 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>SCadSpec2</td>
<td>871269762</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:20:50 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>DGadSpecs3</td>
<td>871153759</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:20:51 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>EG2Spec1</td>
<td>872165731</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:20:53 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>SC2Spec2</td>
<td>872265272</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:20:53 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>DG2Spec3</td>
<td>872361948</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:20:54 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>DISTRIBUTOR</td>
<td>76205974621</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:21:14 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>INTAKE, MANIFOLD</td>
<td>46954198822</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:21:14 AM</td>
<td>- Pick an Action</td>
</tr>
<tr>
<td>FLYWHEEL</td>
<td>73132871893</td>
<td>no</td>
<td>Checked in</td>
<td>11-17-03 11:21:14 AM</td>
<td>- Pick an Action</td>
</tr>
</tbody>
</table>
```
Composite Frame Containing a Composite Table

Use the following URL and log in as an administrator:


On the following screen you see a composite frame containing:

- a custom frame header displaying the current folder name
- a navigation bar
- a composite table presenting folder content
Composite Frame Containing a Composite Tree

Use the following URL and log in as an administrator:


On the following screen you see a composite frame containing:

- a navigation bar
- a folder browser tree
Multiple Components

Click on the **Table and Trees** tab. The following screen containing two tables and a tree will appear. You will be able to contract and expand individual components.
Primitive Elements

Click on the Primitives tab. The following screen will present you with a variety of ways labels and hyper links can be rendered.
Wizards

To see the **Create Part** screen below, select from a drop down list of actions on a part object.

Pickers

Click on **Pick Location**. The Folder Picker screen will appear.
Search

Click on the Step 2 tab of the Create Part wizard and you will be able to search for parts to be used by the part you are creating.
Building a Sample Application with DCA

This section will step you through the creation of a small application that will start with the display of a table of objects, then add actions to those forms. Then, using the sample configurations in codebase/registry/dca/com/ptc, we will explore various other features. For this example, we will get a list of all existing LifeCycleTemplate objects in the system to familiarize you with the basics of creating DCA applications.

Create the new package for the definitions

In the $(wt.home)/codebase/registry/dca folder, create a new folder for this package. Let’s assume it will be called "test". Create a new file called lifecycle.xml.

Add the content from the Sample XML Configuration File, found later in this section, to this file and restart the servlet engine. Execute the gateway JSP as follows:

Getting and Displaying a Basic Table

To display a basic table, several configuration objects must be defined. First, there must be a data retrieval task that will pull the object data. Once the data can be obtained, a table that uses some of this data is defined. This table is then wrapped in a frame definition so it can be retrieved from the gateway JSP page.

1. Define the package name.

   To insure that the configuration data is unique, the Repository object is defined with a package name. This particular file can be re-used in its entirety to produce another object list provided the package name is altered.

   `<Repository package="com.ptc.lifecycles"/>

2. Define the synonyms.

   `<Context>
   `  <Synonym id="TARGET_OBJECT"
     value="wt.lifecycle.LifeCycleTemplate"/>

   `</Context>
These synonyms define names that can be re-used multiple times throughout the configuration. You will notice that the full file makes various references to \%[TARGET_OBJECT]. Note that the square brackets around the name are optional – they allow multiple synonyms to be specified in a non-ambiguous manner when necessary. In order to make this file query for other object types, you need only update these synonyms to match your desired object information.

3. Define the properties to be displayed in the user interface and map them to the attributes to be retrieved from the object.

The **Property** element defines how a data values are displayed and manipulated. The **PropertyHandler** defines how the values are moved in and out of the data container provided by an Info*Engine task result. There are a number of standard handlers available to deal with the available data types. Additional handlers can be written and used in **Property** definitions.

The **Renderer** element defines the code used to transform the value into the user interface format. A Renderer can either specify a class or resource attribute. A class attribute specifies a java bean that performs the rendering (fast, but may require Servlet engine restarts in order to have changes take effect). A resource attribute specifies a JSP relative URL (slow, but allows for rapid development and feedback). It is recommended that when building a new renderer, you use the JSP form until you have it working as desired, then convert it to the java bean format for use in high volume pages.

The **Label** defines the property’s label. The value specified here should be looked up in a resource bundle so that the label value produced in the user interface is localized. For this revision, internationalization is not activated, so to localize the application, you must localize the data in the XML files directly.

The **Need** element specifies a logical name of the attribute (from the Info*Engine container) to be used to construct the property’s value.

**Note:** A single property may need more than one single attribute value to operate correctly.
4. Define the table and interface to the Data Retrieval Task

```xml
<Context type="%TARGET_OBJECT">
  <SimpleTable id="table" selectorType="multi">
    <SelectAllPrompt resource="All"/>
    <Type type="%TARGET_OBJECT"/>
    <Label resource="%LABEL"/>
    <SimpleColumn show="name">
      <Label resource="name"/>
    </SimpleColumn>
    <ModelDoer class="com.ptc.core.ca.co.client.doer.
      CoDefaultSimpleModelTaskDoer">
      <Task nextOperation="STDOP|com.ptc.windchill.view"
        taskDelegate="dca-query-objects">
        <TaskParameter name="search_type" value="%TARGET_OBJECT"/>
        <TaskParameter name="where" value="name='*'"/>
      </Task>
    </ModelDoer>
  </SimpleTable>
</Context>
```

The `SimpleTable` element defines the information needed to display a table in the user interface. In the example, the ID is “table”, which will be used later to identify which table is to be included in the output. For now, it is recommended that the ID for each table be unique in the system, as the ID of the table is used for the preference handlers which allow for customization of the columns and sorting behaviors of the table.

The object type to be displayed is specified by the `Type` element (and reuses the `TARGET_OBJECT` synonym setting). Multiple `Type` elements may be specified. When you do so, you tell DCA to produce actions based on the most specific type of object type found for a given row. If you just specify `wt.part.WTPart` as the `Type`, then all parts found, regardless of whether you have just WTParts or 10 different subclasses and subtypes will be treated as WTPart objects. If you specify all 10 types you have defined, each row will be created with the most appropriate actions based on the most specific type found when analyzing the actual data found. The Windchill object hierarchy is used to find the closest match.

The `SimpleColumn` element specifies the name of a drawable element to be rendered as the column. In this case it is the Property we defined earlier called “name”. If we had more properties defined, you could call them out to make more columns display in the table. Any element that can be drawn, even entire other pages, can be used as the target of a `SimpleColumn`.

The `ModelDoer` element defines how data will be retrieved from the data source. There is a single defined `ModelDoer` classes that will help you acquire data. Since DCA is targeting applications which integrate or federate Windchill data to other systems, the primary `ModelDoer` class interfaces to an Info*Engine task. That task is defined in the `Task` element. In our case we are using the Windchill Federated TaskDelegate called `dca-query-objects`. Since this object needs to have a type parameter specified, we include a
A TaskParameter object which indicates that the TARGET_OBJECT synonym be reused as the parameter (search_type) specified to the task. Should you wish to produce a restricted list of the records from the database, you could change the second TaskParameter to produce a where clause, for example:

```xml
<TaskParameter name="where" value="name='a*'"/>
```

This would only return those objects that start with “a”. The parameters required for the task are completely dependent upon the task definition. TaskParameters can also point at Java beans that will produce parameter information. By default, as part of the CoDefaultSimpleModelTaskDoer, any attribute specified in a Need element will be put into the TaskParameter called dca_attribute. This greatly reduces the amount of configuration work needed to create the task interface.

5. Define the frame to hold the table

```xml
<SimpleFrame id="main">
  <Extend ref=":simpleframe.template"/>
  <Label resource="%LABEL"/>
  <ToolTip resource="%TOOLTIP"/>
  <SimpleLayout>
    <LayoutRow id="row5">
      <Insert ref="table" type="%[TARGET_OBJECT]"/>
    </LayoutRow>
  </SimpleLayout>
</SimpleFrame>
```

This Frame is named “main”. Note that it extends from another definition called simpleframe.template. Every page DCA generates needs to have access to CSS style sheets and a set of Javascript functions that assist the user model. This simpleframe.template definition loads the HTML needed by DCA to produce the standard look and feel of the pages.

The Frame then defines a label and tooltip for the page. This information is used in the window caption area in the browser.

The Frame then gets down to the real work by defining a SimpleLayout with a single LayoutRow element that calls for the insertion of the table we defined earlier. LayoutRows must have unique ID attributes so they may be stored properly in the internal DCA data model.

6. Test the Table. When you execute the gateway, it points at the frame “com.ptc.lifecycles:main”.

This points at the table “table”. Since a table requires data, it executes it’s ModelDoer, which executes the Info*Engine task to retrieve all lifecycle templates defined in the system, asking it to return any attributes specified in a Need element anywhere in the table definition (like the one in the Property “name”). Then, for each row returned in the query, a table row is produced. Each row will contain the Property “name”, whose value is retrieved from the attribute container entry called “name”.
Activating the Demo Configuration

The following examples are already defined in the demo packages. The demo packages are defined in the directory codebase/registry/dca. In order to activate this directory as the source of DCA configurations, set the following entry in wt.properties:

```
com.ptc.core.ca.co.common.config.directory=registry\codebase\dca
```

**Note:** On a windows system, be sure to use the double backslash form. Java will accept the forward slash form, but this entry is used to generate pathnames for DCA error messages and so the double backslash form should be used to insure that you can copy / paste the pathnames from the error messages into file / open dialogs in your editor.

Adding Commands to the Table Entries

Run the gateway with alias=com.ptc.folder:first.page.view. You should get a result similar to the figure below.
The use of a drop-down menu in the Actions column to provide actions for each row’s object. The drop-down menus may be of different size because they have different content based on each object type’s definition.

If you open the drop-down menu for “Checked Out”, you will get a list of actions similar to the one shown in the screenshot. Let’s look at how that got there.

The file com/ptc/folder/config_folder.xml defines the SimpleFrame “first.page.view”. We found that by using a file content search tool looking for ‘id="first.page.view"’. “first.page.view” references the ID “firstpage.table” in one of it’s rows. “firstpage.table” is a SimpleTable definition in the same file. It is extended from the ID “table”, also defined in the same file:

```
<SimpleTable id="table" selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Insert ref="browser.actions"/>
  <Type type="%[coolpart:CoolPart]"/>
  <Type type="%[part:WTPart]"/>
  <Type type="%[doc:WTDocument]"/>
  <Type type="%[SubFolder]"/>
  <SimpleColumn show=":name">
    <Label resource="Name"/>
    <For type="%[SubFolder]" show="action.folder.view"/>
  </SimpleColumn>
  <SimpleColumn show=":number">
    <Label resource="Number"/>
  </SimpleColumn>
  <SimpleColumn show=":waitingpromotion">
    <Label resource="Waiting"/>
  </SimpleColumn>
  <SimpleColumn show=":checkoutstatus">
    <Label resource="Status"/>
  </SimpleColumn>
  <SimpleColumn show=":createstamp">
    <Label resource="Created"/>
  </SimpleColumn>
  <SimpleColumn show=":folder.browser.actions" sortable="false">
    <Label resource="Actions"/>
  </SimpleColumn>
  <ModelDoer class="%common:SimpleModelTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-list-folder-members">
      <TaskParameter name="content_type" value="WCTYPE|wt.folder.SubFolder"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.doc.WTDocument"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.part.WTPart"/>
    </Task>
  </ModelDoer>
</SimpleTable>
```
**Note:** The folders do not display any data for several of the columns in the table. This is because the tables can ask a row for data that doesn’t get defined for the object type. In the example, folders don’t get checked out, so the checkout status data is empty. It is requested by the table as “:checkoutstatus”, meaning find checkoutstatus that is appropriate for the current object, but there is not a checkoutstatus property defined for a folder object. Therefore, the table always renders an empty cell.

In this definition, the important section is the one highlighted, which makes a global reference to “folder.browser.actions”. This global reference, combined with the multiple Type entries means that the table will support folders, parts, documents and a subtype of parts called CoolPart. It is expected that “folder.browser.actions” will be defined for each of those contexts. Again, use the search utility looking for ‘id=”folder.browser.actions”’ and you will find several. However, one of them will be defined in com/ptc/folder/config_folder.xml which is a good sign that the folder’s definition for this menu will be from the folder package.

The resulting definition looks like this:

```xml
<ActionList id="folder.browser.actions">
  <Insert ref="common:action.cut"/>
  <Insert ref="common:action.copy"/>
  <Insert ref="action.paste"/>
  <Insert ref="common:action.delete.task"/>
  <Insert ref="action.create"/>
  <Insert ref="part:action.create.nocontext" id="action.create.part"/>
  <Insert ref="doc:action.create.nocontext" id="action.create.doc"/>
  <Insert ref="action.open"/>
  <Insert ref="action.open_in_new_frame"/>
  <Insert ref="action.view.task"/>
  <Insert ref="action.edit.task"/>
  <Insert ref="common:renderer.actionDropDownList"/>
  <Prompt resource="- Pick an Action - &nbsp;&nbsp;"/>
</ActionList>
```

Notice how the actions are inserted into the ActionList instead of being completely defined. This allows you to define an action once and only once, yet use it in many menu definitions throughout your application. Also notice how some of the inserts are for simple local package references (like “action.create”) and some are specific references to other packages (like “common:action.cut”). This is used to gain even more re-use from the definitions. Global actions like cut and copy can be defined in a common package and referenced by every object’s toolbar. However, actions such as “action.create” must be defined for the specific type of object being processed. Notice these two definitions:

```xml
<Insert ref="part:action.create.nocontext" id="action.create.part"/>
<Insert ref="doc:action.create.nocontext" id="action.create.doc"/>
```
They both reference the same ID (action.create.nocontext), but from different packages (part and doc). In order to keep them unique in this particular menu, their ID values are overridden. Otherwise, DCA will cache action.create.nocontext and there will only be one definition used, which would produce two part create actions instead of one part and one doc action.

Finally, the last insert in the ActionList is:

```xml
<Insert ref="common:renderer.actionDropDownList"/>
<Prompt resource="- Pick an Action - &nbsp;&nbsp;"/>
```

This defines to DCA how the ActionList will be rendered. This particular definition provides the menu as a drop-down. Changing this to:

```xml
<Insert ref="common:renderer.horizontalList"/>
```

would produce a toolbar within the Actions column in the table for any folders in the table.

**Actions**

Actions are the handles by which a unit of work is invoked in DCA. They can be defined in several forms:

- **InPlace** – execute a command without further user interaction and update the current page
- **NewTemporaryFrame** – launch a popup window to interact with the user
- **GoTo** – execute a command and display the result in the current browser window.
- **NewFrame** – execute a command that gets displayed in another new browser window.

Here is the sample to launch a create wizard for a folder:

```xml
<Action id="action.newobject">
  <Label resource="Create"/>
  <ToolTip resource="Create"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/newfoldertl.gif"/>
  <Extend ref="com.ptc.windchill:action.newTemporaryFrame"/>
  <Location ref="dialog.create.object"/>
</Action>
```

**com.ptc.windchill:action.newTemporaryFrame** is defined in com/ptc/common/config_common_templates.xml as:

```xml
<Action id="action.newTemporaryFrame.template">
  <Insert ref="common:renderer.action"/>
</Action>
```
**common:renderer.action** is defined in com/ptc/common/config_common_renderers.xml as:

```xml
<Renderer id="renderer.action" 
class="com.ptc.core.ca.web.client.action.WebActionRenderHandler"/>
```

Together, these define an action that will launch a temporary frame using alias=dialog.create.object. It also defines, the label, tooltip and image to be used when rendering the action into another page. The ActionHandler tells it what class to run to activate the action when the user clicks the link. This particular one creates the new URL to be executed, creates the temporary window and launches the URL into it. The final entry is the Renderer – this tells how to draw the action into the user interface, in this case, a hyperlink with an image and a label drawn beneath the image. The particular definition just discussed was used to display the “Create Folder” link in the toolbar on the table as shown in the screenshot.

### Adding a Toolbar to the Table

Referring to the previous screenshot, you will see that there is also a toolbar defined at the top of the table. These actions are either actions that are tied to the selection in the table (since “table” is defined with selectorType="multi", the table is drawn with checkboxes at the front of each row and the one “all” checkbox in the header) or pertain to the table itself (such as the Preferences icon which will launch a wizard to set sort orders and column visibility). So if we look at our “table” element, we will find an Insert referring to “browser.actions”. The definition for browser.actions is as follows:

```xml
<ActionList id="browser.actions" actionListType="toolbar" 
context.insideToolBar="true">
   <Insert ref="common:action.component.expand"/>
   <Insert ref="common:action.component.contract"/>
   <Insert ref="common:action.cut"/>
   <Insert ref="common:action.copy"/>
   <Insert ref="action.paste"/>
   <Insert ref="common:action.delete.task"/>
   <Insert ref="action.create" id="action.create.folder"/>
   <Insert ref=":action.create" id="action.create.part" type="%[part:WTPart]"/>
   <Insert ref=":action.create" id="action.create.doc" type="%[doc:WTDocument]"/>
   <Insert ref="enterprise:action.checkout.task.trusted"/>
   <Insert ref="enterprise:action.undocheckout.task.trusted"/>
   <Insert ref="common:action.table.pref.edit"/>
</ActionList>
```

This ActionList is basically the same as the other list used before except that this list has some items which apply to the table itself. Most of the items are duplicated in both the row-based toolbar and in the table toolbar. This is so the user is free to execute commands one at a time against an object or to execute commands for a selected set of entries in the overall table. Regardless of whether the actions are run from the toolbar or the row, the Action definitions themselves are identical. Each action knows how to get the current selection from the DCA environment. How that is resolved is based on how it is configured into the page. If it is
configured into the table toolbar, the selection state comes from looking at all the checkboxes on the table rows. If from a row’s action column, the selection state comes from the object represented in the row. The DCA developer does not need to know this – it is taken care of for them. They just build Actions, put them in ActionLists and put them into columns or tables or trees.

Another slight difference is that this ActionList defines an actionListType of toolbar and no renderer. The actionListType provides a default rendering for the list to make it look and feel like a toolbar. Either approach is acceptable.

**Navigation Bars**

You will note the way the pages have a set of tabs across the top as part of the header of the page with optional additional rows of links that may be taken. These “navigation bars” are the anchors that keep people in the application you have created for them. For your application, the same navigation bar should be called out for every non-popup page you create (you don’t need / want them on your picker dialogs). We will look at how the navigation bar for the alias com.ptc.folder:first.page.view is constructed.

At the top of the SimpleLayout definition for first.page.view in com/ptc/folder:config_folder.xml, we find the following LayoutRow defined:

```xml
<LayoutRow id="rowNav" ifnot.decoration="portlet">
  <Insert ref="navigation:navbar2" colspan="2"/>
</LayoutRow>
```

This LayoutRow will only be processed if the decoration context setting is not set to “portlet”. This feature exists so that you may re-use any page you define within another page by suppressing it’s header and footer behaviors, leaving only the raw body of the page intact. Try adding “&decoration=portlet” to the URL and see how the navigation bar is suppressed as shown here:

Find the definition of com.ptc.navigation:navbar2. This can be found in com/ptc/navigation/config_nav.xml

```xml
<SimpleNavBar id="navbar2">
  <NavBarRow id="1">
    <NavBarAction id="navbar.item.1">
      <Extend ref="navbar.action.1"/>
    </NavBarAction>
    <NavBarRow id="2">
      <NavBarAction id="navbar.item.1.4">
        <Extend ref="navbar.action.1.4"/>
      </NavBarAction>
      <NavBarAction id="navbar.item.1.5">
        <Extend ref="navbar.action.1.5"/>
      </NavBarAction>
      <NavBarAction id="navbar.item.1.6">
        <Extend ref="navbar.action.1.6"/>
      </NavBarAction>
    </NavBarRow>
    <Insert ref="level2_navbarrow"/>
  </NavBarRow>
  <NavBarAction id="navbar.item.2">
    ...
  </NavBarAction>
</SimpleNavBar>
```

```xml
<NavBarRow id="1">
  <NavBarAction id="navbar.item.1">
    <Extend ref="navbar.action.1"/>
  </NavBarAction>
  <NavBarRow id="2">
    <NavBarAction id="navbar.item.1.4">
      <Extend ref="navbar.action.1.4"/>
    </NavBarAction>
    <NavBarAction id="navbar.item.1.5">
      <Extend ref="navbar.action.1.5"/>
    </NavBarAction>
    <NavBarAction id="navbar.item.1.6">
      <Extend ref="navbar.action.1.6"/>
    </NavBarAction>
  </NavBarRow>
  <Insert ref="level2_navbarrow"/>
</NavBarRow>
```

```xml
<NavBarAction id="navbar.item.2">
  ...
</NavBarAction>
```
Navigation bar elements define a hierarchy of menu objects. Each NavBarRow defines a set of NavBarAction elements, each of which can hold other NavBarRow elements containing more NavBarAction elements, and on and on. Each NavBarRow also needs to specify a renderer. Typically the first level will be rendered as tabs and the successive levels are rendered as simple links, but that is up to the application designer. If you can render it, you can use it in the navbar.

In the sample, there are three top level tabs, each of which have additional lower level actions. In the case of the NavBarRow 3, it also defines a third level of navigation. Each of these NavBarActions defined in the NavBar reuse, via the
Extend element, other predefined Action elements, allowing re-use of various pages already defined. See a few definitions from the same file:

```
<Action id="navbar.action.2">
    <Extend ref="common:navAction.goTo.template"/>
    <Label resource="Primitives"/>
    <ToolTip resource="Primitives"/>
    <Location ref=":junkyard.page.allLabels"/>
    <Insert ref="common:renderer.level1_tab"/>
</Action>

<Action id="navbar.action.2.1">
    <Extend ref="common:navAction.goTo.template"/>
    <Label resource="Labels"/>
    <ToolTip resource="Labels"/>
    <Location ref=":junkyard.page.labels"/>
    <Insert ref="common:renderer.level2_tab"/>
</Action>

<Action id="navbar.action.2.2">
    <Extend ref="common:navAction.goTo.template"/>
    <Label resource="Hyper Links"/>
    <ToolTip resource="Hyper Links"/>
    <Location ref=":junkyard.page.hyperLinks"/>
    <Insert ref="common:renderer.level2_tab"/>
</Action>
```

This creates the overall application by wiring together the useful functions for a given type of user. And since the navbar should be inserted on every non-popup page in the application, it will always allow the user to get back to known anchor points in the defined application. Whenever the user is on a page registered in the navigation bar, the rendered output will highlight the tabs and links which match the current alias. If there is no match, only the first level navigation items are visible, which still allows the user to return to a known point in the application.

From a best practice standpoint, it makes a lot of sense to produce a single referenced global ID defined like this:

```
<ElementGroup id="std.navbar">
    <LayoutRow id="rowNav" ifnot.decoration="portlet">
        <Insert ref="com.hulseware.navbar:navbar" colspan="10" if.appId="beh"/>
        <Insert ref="com.democo:navbar" colspan="10" if.appId=""/>
        <Insert ref="com.democo:navbar" colspan="10" if.appId="app1"/>
        <Insert ref="com.democo:navbar" colspan="10" if.appId="app2"/>
        <Insert ref="com.ptc.programlink.navbar:navbar" colspan="10" if.appId="pgml"/>
        <Insert ref="com.ptc.navigation:navbar" colspan="10" if.appId="sl"/>
        <Insert ref="com.ptc.navigation:navbar" colspan="10" if.appId="SPL"/>
    </LayoutRow>
</ElementGroup>
```
Note: The use of all the different “if” settings in this list – the goal would be to return one and only one navbar definition to any application which references “:std.navbar”. This way, you don’t have to modify any pages at all to add a new application navbar – just add it here.

Creating a Property View Form

Here is the view part popup launched from one of the parts shown in the original screenshot:

This is a simple SimplePropertyPanel implementation which retrieves the desired attributes from the server and displays them plus a toolbar of actions. To find the definition for this page, find the folder.browser.actions definition in the folder com/ptc/part. That definition exists in config_part.xml. The ActionList contains the following reference:

<Insert ref="action.view.task"/>

“action.view.task” is also defined in config_part.xml as:

<Action id="action.view.task" needSelection="true">
  <Extend ref="common:action.newTemporaryFrame.template" />
  <Label resource="View Properties" />
  <ToolTip resource="View Properties: {0} part">
    <Need attribute="name" />
  </ToolTip>
  <Location ref=":dialog.view.task" />
</Action>
There are a few interesting points about this definition. Note the needSelection="true" setting – this indicates that this action can only execute if there is one or more objects which have been selected. Since this page is going to produce a popup window, this action is much better suited for use in an object toolbar than in a table toolbar, since using this action in a multiple selection scenario would result in multiple popup windows being launched, one for each selected object. The second point is the ToolTip definition. It includes a Need element to be able to get the name for the current object. Then, that name element is used to replace the {0} in the resource setting in the ToolTip. This will produce tooltips like “View Properties: ENGINE part”.

The Location element specifies which page gets processed into the temporary frame the action will create. In this case, it is “dialog.view.part”. It is defined as:

```xml
<SimpleFrame id="dialog.view.task" type="%[WTPart]"
   width="750" height="800">
   <Extend ref="common:simpleframe.template"/>
   <Label resource="View Properties"/>
   <Wizard>
      <Label resource="Properties of Part (View)"/>
      <WizardStepElements>
         <Insert ref="layout.view.task"/>
      </WizardStepElements>
      <Insert ref="common:object.getForView.task"/>
      <Insert ref="common:wizard.actions"/>
   </Wizard>
</SimpleFrame>
```

This is the page to be rendered into the new popup. It extends, as always, from common:simpleframe.template to insure the proper look and feel. It sets the caption to “View Properties” via the Label element. It then defines a single step Wizard to display the properties. It will use layout.view.task to define the attribute layout for the one page being produced, which we will come back to shortly. There are two inserts. The first gets the ModelDoer needed to execute the query for our part. It is defined in com/ptc/common/common_config_templates.xml as:

```xml
<ModelDoer id="object.getForView.task"
   class="%SimpleModelTaskDoer">
   <Task nextOperation="STDOP|com.ptc.windchill.view"
      taskDelegate="dca-get-single-object"/>
</ModelDoer>
```

This definition can be used wherever you need to acquire view-only data from an object in a table or tree row.

The second insert (“wizard.actions”) provides the button set for the bottom of the page. It is defined in com/ptc/common/common_config_actions.xml as:

```xml
<WizardActions id="wizard.actions">
   <Insert ref="action.wizard.previous"/>
   <Insert ref="action.wizard.next"/>
   <Insert ref="action.wizard.ok"/>
   <Insert ref="action.wizard.apply"/>
   <Insert ref="action.wizard.cancel"/>
   <Insert ref="common:renderer.horizontalList"/>
</WizardActions>
```
Note: You don’t see five buttons at the bottom of the page – you only see one – the Cancel button. This is because the wizard handlers know when the different buttons are appropriate. In our case, we are running a single step wizard, so the previous and next buttons are not applicable. Also, since we are running in view mode, the OK and Apply buttons are also not applicable. So, only the Cancel button appears.

Now that we have the wrapper for our property sheet, we need to return to the content definition. “layout.view.task” is defined as:

```xml
<SimpleLayout id="layout.view.task" type="%[WTPart]%">
<!-- we want to have ids of layout rows to be generated -->
<LayoutRow id="1">
  <Insert ref="panel.view.actions"/>
</LayoutRow>
<LayoutRow id="2">
  <Insert ref=":number"/>
</LayoutRow>
<LayoutRow id="3">
  <Insert ref=":versionlineage"/>
</LayoutRow>
<LayoutRow id="4">
  <Insert ref=":name"/>
</LayoutRow>
<LayoutRow id="5">
  <Insert ref=":defaultUnit"/>
</LayoutRow>
<LayoutRow id="6">
  <Insert ref=":type"/>
</LayoutRow>
<LayoutRow id="7">
  <Insert ref=":source"/>
</LayoutRow>
<LayoutRow id="8">
  <Insert ref=":view"/>
</LayoutRow>
<LayoutRow id="9">
  <Insert ref=":checkoutstatus"/>
</LayoutRow>
<LayoutRow id="11">
  <Insert ref=":creator"/>
  <Insert ref=":createstamp"/>
</LayoutRow>
<LayoutRow id="12">
  <Insert ref=":modifier"/>
  <Insert ref=":updatestamp"/>
</LayoutRow>
<LayoutRow id="13">
  <Insert ref=":lifecyclename"/>
</LayoutRow>
<LayoutRow id="14">
  <Insert ref=":lifecyclestate"/>
  <Insert ref=":waitingpromotion"/>
</LayoutRow>
```
This creates a 13 x 2 grid for outputting the various Property elements specified. Note that they are all global references with the exception of the toolbar. While this does nothing for our simple example, it is possible to define this set of LayoutRow elements as an ElementGroup. This would allow a re-use pattern to include something like: `view.layout` based on the type of object being worked on.

**Creating a Property Edit Wizard**

Here is the screenshot of the Step 1 page of the property edit popup window:
The following figure is a screenshot of the Step 2 page of the Property edit popup window:

The folder.browser.actions for parts is defined in com/ptc/part/config_part.xml. It is defined as follows:

```xml
<ActionList id="folder.browser.actions">
  <Insert ref="common:action.cut"/>
  <Insert ref="common:action.copy"/>
  <Insert ref="common:action.delete.task"/>
  <Insert ref="enterprise:action.checkout.task"/>
  <Insert ref="enterprise:action.undocheckout.task"/>
  <Insert ref="action.create"/>
  <Insert ref="action.view.task"/>
  <Insert ref="action.view.usedby"/>
  <Insert ref="action.edit.task"/>
  <Insert ref="action.change.id"/>
  <Insert ref=":action.content.list"/>
</ActionList>
```
The definition we want is action.edit.task. It launches the frame dialog.edit.task in a popup window.

<Action id="action.edit.task" needSelection="true">
  <Extend ref="common:action.newTemporaryFrame.template" />
  <Label resource="Edit Properties" />
  <ToolTip resource="Edit Properties: {0} Part">
    <Need attribute="name" />
  </ToolTip>
  <Location ref=":dialog.edit.task" />
</Action>

The definition for dialog.edit.task for WPart objects can be found in com/ptc/part/part_edit_link.xml.

<SimpleFrame id="dialog.edit.task" type="{WPart}" mode="edit" width="750" height="800">
  <Extend ref="common:simpleframe.template" />
  <Label resource="Edit Properties" />
  <Wizard>
    <Label resource="Properties of Part (Edit)"/>
    <WizardStepActions>
      <Insert ref="common:action.wizard.goto">
        <Label resource="Step 1"/>
        <ToolTip resource="Step 1"/>
        <Insert ref="common:renderer.level1_tab"/>
      </Insert>
      <Insert ref="common:action.wizard.goto" id="action.wizard.goto2">
        <Label resource="Step 2"/>
        <ToolTip resource="Step 2"/>
        <Insert ref="common:renderer.level1_tab"/>
      </Insert>
      <Insert ref="common:renderer.wizardTab"/>
    </WizardStepActions>
    <WizardStepElements>
      <Insert ref="layout.edit.task" />
      <Insert ref="layout.editlink.task" />
    </WizardStepElements>
    <ModelDoer class="%common:SimpleModelTaskDoer">
      <Task nextOperation="STDOP|com.ptc.windchill.update" taskDelegate="dca-get-single-object"/>
    </ModelDoer>
    <ActionDoer class="%common:SimpleActionTaskDoer">
      <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-edit-links">
        <TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoSimpleFunction_Action_Param_ObjectsAsGroup" element="table.edit.usage" name="newUsageLinks" selected="false"/>
        <TaskParameter name="output_type" value="relation"/>
      </Task>
    </ActionDoer>
As usual, this frame extends from simpleframe.template to get the standard look and feel. Then, the entire layout is a Wizard element. Multi-step wizards like this one have two sub-elements defined within – WizardStepActions and WizardStepElements. WizardStepActions define the menu set for the individual pages of the wizards and are typically defined as tabs in the rendered result. The number of Elements within the two sections define the number of tabs. If there are two WizardStepActions, there must be two WizardStep Elements.

The WizardStepActions are two inserted copies of common:action.wizard.goto. Note that the Insert is not just the reference to the action.wizard.goto element. It also includes overrides. Note that these two Inserts could also be defined as:

```
<Action id="step1">
  <Extend ref="common:action.wizard.goto"/>
  <Label resource="Step 1"/>
  <ToolTip resource="Step 1"/>
  <Insert ref="common:renderer.level1_tab"/>
</Action>
```

There is no difference in the way the definitions ultimately get processed. It merely demonstrates the flexibility of the configuration system. The decision between the forms would be clarity or ease of re-use. Using Insert obscures the fact that the Action element is really the desired sub-element but the Insert allows using multiple element types. Either form is acceptable; make your decisions based on insuring proper re-use of the elements you define.

That said, the actual definition of action.wizard.goto is in config_common_actions.xml

```
<Action id="action.wizard.goto">
  <ActionHandler class="com.ptc.core.ca.web.client.wizard.
    WebWizardStepActionHandler"/>
  <!-- This is just to show that we can specify a doer
      for a step -->
  <ActionDoer class="com.ptc.core.ca.co.client.wizard.
```
These are necessary for wizards to operate and that it is unlikely that you would ever need to alter these definitions.

The WizardStepElements insert the two page definitions layout.edit.task and layout.editlink.task. “layout.edit.task” is defined in config_part.xml and defines the first page which is a combination of viewing and editing various properties of the part.

```xml
<SimpleLayout id="layout.edit.task" type="%[WPart]">
  <LayoutRow id="1">
    <Insert ref="panel.view.actions" mode="view"/>
  </LayoutRow>
  <LayoutRow id="2">
    <Insert ref=":number" mode="view"/>
  </LayoutRow>
  <LayoutRow id="3">
    <Insert ref=":versionlineage" mode="view"/>
  </LayoutRow>
  <LayoutRow id="4">
    <Insert ref=":name" mode="view"/>
  </LayoutRow>
  <LayoutRow id="5">
    <Insert ref=":defaultUnit" mode="view"/>
  </LayoutRow>
  <LayoutRow id="6">
    <Insert ref=":type"/>
  </LayoutRow>
  <LayoutRow id="7">
    <Insert ref=":source"/>
  </LayoutRow>
  <LayoutRow id="8">
    <Insert ref=":view" mode="view"/>
  </LayoutRow>
  <LayoutRow id="9">
    <Insert ref=":checkoutstatus" mode="view"/>
  </LayoutRow>
  <LayoutRow id="11">
    <Insert ref=":creator" mode="view"/>
  </LayoutRow>
  <LayoutRow id="12">
    <Insert ref=":modifier" mode="view"/>
  </LayoutRow>
  <LayoutRow id="13">
    <Insert ref=":lifecyclename" mode="view"/>
  </LayoutRow>
  <LayoutRow id="14">
    <Insert ref=":lifecyclestate" mode="view"/>
  </LayoutRow>
</SimpleLayout>
```
This definition is very similar to the one used in the view page we reviewed earlier. Note the use of mode="view" to suppress editing of many of the properties.

“layout.editlink.task” is defined in part_edit_link.xml and defines the second page which allows the user to edit the list of child parts associated to the part being edited.

This page defines a table to hold the existing parts which are used by the part we are editing. Note that even complex objects like tables and trees can be placed within these layouts. The issue will be that the wizard will have to have a more complex action to deal with the fact that we will have lists of objects to deal with when it comes time to finish the wizard. This also shows the use of Info*Engine to read Windchill link objects.

Note the references to uses^:number. This says you will find the property number in the uses section of the Info*Engine result. It still uses the standard definition of number supplied for the object (like WTPart), but it will find it in the Info*Engine result as uses^:number (number being the attribute in the Need element from the Property definition for “number”). Note that quantity.amount and quantity.unit will not need this approach because they are actual attributes on the link object.
itself. The uses^:number approach is only needed to find the attributes on the related objects, not those on the links themselves.

Then, back to the main Wizard body. There is a ModelDoer which reads the primary object’s information into the wizard.

```xml
<ModelDoer class="%common:SimpleModelTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.update"
    taskDelegate="dca-get-single-object"/>
</ModelDoer>
```

Finally, after we have loaded data into each of the previous pages, we eventually hit the Finish button on the wizard. The buttons are defined in common:wizard.actions just like on the view wizard. At that point, the ActionDoer takes over

```xml
<ActionDoer class="%common:SimpleActionTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.view"
    taskDelegate="dca-edit-links">
    <TaskParameter function="com.ptc.core.ca.co.client.doer.
      function.impl.CoSimpleFunction_Action_Param_ObjectsAsGroup"
      element="table.edit.usage"
      name="newUsageLinks"
      selected="false"/>
    <TaskParameter name="output_type" value="relation"/>
    <TaskGroup name="linkDescr">
      <TaskElement name="usage">
        <TaskParameter name="type" value="wt.part.WTPartUsageLink"/>
        <TaskParameter name="otherSideAttr" value="uses"/>
        <TaskParameter name="masterOnThisSide" value="false"/>
        <TaskParameter name="masterOnOtherSide" value="true"/>
        <TaskParameter name="newLinksGroupName" value="newUsageLinks"/>
      </TaskElement>
    </TaskGroup>
  </Task>
</ActionDoer>
```

This is a complex updater which expects to find the primary object information and a table containing the link information and the user intent for this operation (adds, deletes, updates). The task definition can be found in tasks/com/ptc/dca/dca-edit-links.xml. This is an example of a complex task being used to cause adds, deletes, and updates to the links between two objects.
Creating a Create Wizard

Here is the screenshot of the Create XXX wizard.

This is another wizard implementation which produces a not yet persisted copy of an object, allows the user to interact with default and sticky values from preferences and finally to persist the new object. To find the definition for this page, find the folder.browser.actions definition in the folder com/ptc/part. That definition exists in config_part.xml. The ActionList contains the following reference:

<Insert ref=":action.create" id="action.create.doc"  
type="%{doc:WTDocument}"/>

“action.create” is defined in com/ptc/doc/config_doc.xml as:

<Action id="action.create">
<Extend ref="common:action.newTemporaryFrame.template" />
<Label resource="Create Document" />
<ToolTip resource="Create Document" />
<Image resource="/wtcore/images/com/ptc/core/foundation/document_create.gif"/>
<Location ref=":dialog.create" />
</Action>
This is just the same as all the other popup launchers we have already seen. This time, we are going to use the frame definition dialog.create for the document context. This definition is found in config_doc.xml and looks like this:

```xml
<SimpleFrame id="dialog.create" mode="create"
    width="600" height="400">
    <Extend ref="common:simpleframe.template"/>
    <Insert ref="common:script.content"/>
    <Label resource="Create Document"/>
    <Wizard>
        <Label resource="Create Document"/>
        <WizardStepActions>
            <Insert id="step.1" ref="common:action.wizard.goto">
                <Label resource="Properties"/>
                <ToolTip resource="Properties"/>
                <Insert ref="common:renderer.level1_tab"/>
            </Insert>
            <Insert id="step.2" ref="common:action.wizard.goto">
                <Label resource="Upload Form"/>
                <ToolTip resource="Upload Form"/>
                <Insert ref="common:renderer.level1_tab"/>
            </Insert>
        </WizardStepActions>
        <WizardStepElements>
            <Insert ref="layout.create"/>
            <Insert ref="common:layout.upload.form"/>
        </WizardStepElements>
    </Wizard>
    <ModelDoer class="com.ptc.core.ca.co.client.doer.
        CoDefaultModelTaskDoer">
        <Task nextOperation="STDOP|com.ptc.windchill.create"
            taskDelegate="dca-create-model">
            <TaskParameter function="com.ptc.core.ca.co.client.doer.
                function.impl.CoSimpleFunction_Model_Param_MainObjectIdentity"/>
            <TaskOutParameter function="com.ptc.core.ca.co.client.
                doer.function.impl.CoSimpleFunction_Model_Result_Default"/>
            <TaskOutParameter function="com.ptc.core.ca.co.client.
                doer.function.impl.CoFunction_Model_Result_CreateNpa"
                name="uploadPath"/>
            <TaskParameter name="target_type"
                value="WCTYPE|wt.doc.WTDocument"/>
            <TaskParameter name="dca_field"
                computedValue="com.ptc.refimpl.co.
                computedvalues.DefaultLocationFieldComputedValue"/>
            <TaskParameter name="dca_field"
                value="default doc name"/>
            <TaskParameter name="dca_field"
                value="default doc number"/>
            <TaskParameter name="dca_field"
                value="a default title"/>
        </Task>
    </ModelDoer>
    <ActionDoer class="com.ptc.core.ca.co.client.doer.
        CoDefaultActionTaskDoer">
        <Task nextOperation="STDOP|com.ptc.windchill.view"
            taskDelegate="dca-create-action">
            <TaskParameter function="com.ptc.core.ca.co.client.doer.
                function.impl.CoFunction_Model_Result_CreateNpa"
                name="uploadPath"/>
            <TaskParameter name="target_type"
                value="WCTYPE|wt.doc.WTDocument"/>
            <TaskParameter name="dca_field"
                computedValue="com.ptc.refimpl.co.
                computedvalues.DefaultLocationFieldComputedValue"/>
            <TaskParameter name="dca_field"
                value="default doc name"/>
            <TaskParameter name="dca_field"
                value="default doc number"/>
            <TaskParameter name="dca_field"
                value="a default title"/>
        </Task>
    </ActionDoer>
</SimpleFrame>
```
This looks similar to the edit wizard with the following exceptions:

- Since we are dealing with a specialized applet for uploading object content, we need to insert some specialized javascript handling configuration information:

  <Insert ref="common:script.content"/>

  which is defined as:

  <Script id="script.content"
      resource="wtcore/js/com/ptc/core/ca/web/misc/content.js">
    <ModelHandler class="com.ptc.core.ca.co.client.frame.
        CoHeaderResourceModelHandler"/>
    <Insert ref="common:renderer.script"/>
  </Script>

- The ModelDoer is much more complex than with the edit wizard. This is because we need to create the unpersisted data model from which to work. Note the use of TaskOutParameter elements – these take the result information from the task and properly push it into the DCA data model for use in the wizard. Note also, the use of TaskParameter and dca_field values to set default values for the various attributes in the object. This is only necessary until a user first executes the wizard. After that, the “sticky” values for every attribute are stored in user preferences and returned on the next create operation.

- The ActionDoer interfaces with a task that is only concerned with creating the object and the links to existing objects. There is no need to process deletes or
updates, since they did not exist at all at the start of the wizard. See the task/com/ptc/dca/dca-create-action.xml.

- The second step has a couple of embedded applets to prepare for content transfer. “common:layout.upload.form” is defined in com/ptc/common/content.xml as:

```xml
<SimpleLayout id="layout.upload.form">
  <LayoutRow id="1">
    <Insert ref=":property.uploadPath" mode="create"/>
    <List>
      <Insert ref="applet.upload.fileChooser"/>
      <Insert ref="applet.upload.dndMicro"/>
    </List>
    <LabelArea labelType="text">
      <Label resource="Drag a file and drop it on the Attach icon." />
    </LabelArea>
  </LayoutRow>
</SimpleLayout>
```

**Note:** the use of mode="create" to insure the renderer and handlers and doers operate as they should for create.

Then, the two applets to produce the applet-based Browse button and the attachment drop target are inserted. These are defined in com/ptc/common/plugin.xml as:

```xml
<Plugin id="applet.upload.fileChooser">
  <Extend ref="applet.template"/>
  <Option theme="attribute" param="name" value="formApplet"/>
  <Option theme="attribute" param="width" value="100"/>
  <Option theme="attribute" param="height" value="30"/>
  <Option theme="parameter" param="java_archive" value="%[SecurityJar]"/>
  <Option theme="parameter" param="java_code" value="%[AppletFileLocator]"/>
  <Option theme="parameter" param="cache_archive" value="%[DocMgrJar]"/>
  <Option theme="parameter" param="mayscript" value="true"/>
</Plugin>

<Plugin id="applet.upload.dndMicro">
  <Extend ref="applet.template"/>
  <Option theme="attribute" param="width" value="16"/>
  <Option theme="attribute" param="height" value="16"/>
  <Option theme="parameter" param="java_code" value="%[AppletDnDMicro]"/>
  <Option theme="parameter" param="cache_archive" value="%[DnDAppletJar]"/>
</Plugin>
```
They are providing the specific arguments needed to drive the applets. They both extend from applet.template, also defined in plugin.xml:

```
<Plugin id="applet.template">
  <Extend ref="plugin.template"/>
  <Option theme="attribute" param="name" value="applet"/>
  <Option theme="attribute" param="width" value="2"/>
  <Option theme="attribute" param="height" value="2"/>
</Plugin>
```

This sets some defaults, but again is extending from another element, plugin.template also defined in plugin.xml:

```
<Plugin id="plugin.template">
  <Option theme="attribute" param="name" value="plugin"/>
  <Option theme="parameter" param="debug" value="false"/>
  <Insert ref=":renderer.plugin"/>
</Plugin>
```

The purpose of this definition is to set the debug setting and to set the renderer (:renderer.plugin) which is defined in the config_common_renderers.xml file in com/ptc/common. This insures consistent use of the Java plug-in for the support of applets in general. This handles generation of the HTML that will support any of the Windchill supported web browsers.

- Finally, note the use of a different set of wizard buttons (common:wizard.upload.actions). These make sure the wizard processes the content upload specified in the second page.
Creating Single Value Pickers

Pickers are a very common user model pattern. They provide a way to access large quantities of objects from which the user must pick one or more. DCA supports two forms of pickers – a single value picker and a multiple value picker. For a single value picker, we’ll look at the definition for the Lifecycle Picker used in the create part screen.
Pressing the “Pick Lifecycle” button results in the following popup dialog:

Picking one of these lifecycles and pressing OK will update the lifecycle field in the original page to have the name of the selected lifecycle. So, to start, let’s look at how the lifecycle field is defined in the create part wizard.

```
<LayoutRow id="4">
    <Property id="new.property.lifecycle">
        <Extend ref="common:property.String.template" />
        <Label resource="Life Cycle:" />
        <Need attribute="lifeCycle"/>
    </Property>
    <Insert ref=":new.property.lifecycle.picker"/>
</LayoutRow>
```

“new.property.lifecycle.picker” is defined in com/ptc/part/lifecycle_pick.xml as follows:

```
<Action id="new.property.lifecycle.picker">
    <Label resource="Pick Lifecycle"/>
    <Location ref=":lifecycle.pick"/>
</Action>
```
Notice the ability to define an Action in the LayoutRow along with other elements. This action will launch “lifecycle.pick”. Note that it also defines an UpdateLocation element to provide a property to accept the picked value. More on that later. The definition of lifecycle.pick can also be found in lifecycle_pick.xml.

```xml
<SimpleFrame id="lifecycle.pick" mode="search" width="200" height="300">
    <Extend ref="common:simpleframe.template"/>
    <Label resource="Lifecycle Picker"/>
    <Wizard>
        <Label resource="Lifecycle Search Picker"/>
        <WizardStepElements>
            <Insert ref="lifecycle.all.layout"/>
        </WizardStepElements>
        <ModelDoer class="com.ptc.core.ca.co.client.doer.CoNoOpModelDoer"/>
        <ClientDoer class="com.ptc.core.ca.co.client.picker.CoPropertyPickerOkActionDoer">
            <Populate from="name"/>
        </ClientDoer>
        <Insert ref="common:picker.actions"/>
    </Wizard>
</SimpleFrame>
```

This is a single-step wizard that uses lifecycle.all.layout to create the body of the picker, then a special ModelDoer that does nothing (the wizard itself is not operating on a single object) and a special ClientDoer that will move an attribute value from the picker to the launching popup based on the setting in the Populate element. In this case, we will return the value of the name element back to the property named in the UpdateLocation of the launching popup (which would be new.property.lifecycle).

The layout looks like this:

```xml
<SimpleLayout id="lifecycle.all.layout">
    <LayoutRow id="rowTable">
        <Insert ref="lifecycle.all.table"/>
    </LayoutRow>
</SimpleLayout>
```

This is a wrapper around lifecycle.all.table to make it usable as a layout. "lifecycle.all.table" is defined as:

```xml
<SimpleTable id="lifecycle.all.table" selectorType="single" mode="view">
    <Type type="wt.lifecycle.LifeCycleTemplate"/>
    <SimpleColumn show=":name">
        <Label resource="Name"/>
    </SimpleColumn>
</SimpleTable>
```
This table simply produces a single selection list of all lifecycles in the system, displaying only the name field. It inserts a pre-defined ModelDoer called lifecycle.all.table.model.doer which is defined as:

```xml
<ModelDoer id="lifecycle.all.table.model.doer"
class="%common:SimpleModelTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.view"
        taskDelegate="dca-query-objects">
    <TaskParameter name="search_type"
                  value="wt.lifecycle.LifeCycleTemplate"/>
    <TaskParameter name="where"
                  value="name=\*\*/">
  </Task>
</ModelDoer>
```

**Note:** This is similar to the ModelDoer definition used in our very first example.

### Creating Multiple Value Pickers

A multi-value picker is used on Step 2 of the Create Part wizard:

The definition for this page creates a table and a toolbar of actions. This table is defined in com/ptc/part/list_picker.xml as:

```xml
<SimpleTable id="table.pick.list.part.target"
             selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Insert ref="pick.action.list.part"/>
  <Type type="wt.part.WTPart"/>
</SimpleTable>
```
<SimpleColumn show=":name">
  <Label resource="Name"/>
</SimpleColumn>
<SimpleColumn show=":number">
  <Label resource="Number"/>
</SimpleColumn>
<SimpleColumn show=":waitingpromotion">
  <Label resource="Waiting"/>
</SimpleColumn>
<SimpleColumn show=":checkoutstatus">
  <Label resource="Status"/>
</SimpleColumn>
<SimpleColumn show=":createstamp">
  <Label resource="Created"/>
</SimpleColumn>
<ModelDoer class="com.ptc.core.ca.co.client.doer.
  CoNoOpModelDoer"/>
</SimpleTable>

**Note:** Since this is a create page, the ModelDoer is a dummy operation that will do nothing. The toolbar definition is in pick.action.list.part and looks like this:

```
<ActionList id="pick.action.list.part"
  actionListType="horizontal">
  <Insert ref="pick.action.add.list.part"/>
  <Insert ref="pick.action.replace.list.part"/>
  <Insert ref="common:action.list.remove"/>
  <Insert ref="common:action.list.removeAll"/>
</ActionList>
```
The two main actions, pick.action.add.list.part and pick.action.replace.list.part, produce another popup dialog that looks like this:

![Popup Dialog Image]

These actions are defined as follows:

```xml
<Action id="pick.action.add.list.part">
  <Extend ref="common:action.pick.add"/>
  <Location ref=":frame.pick.list.part.source.add"/>
  <UpdateLocation ref="table.pick.list.part.target"/>
</Action>

<Action id="pick.action.replace.list.part">
  <Extend ref="common:action.pick.replace"/>
  <Location ref=":frame.pick.list.part.source.replace"/>
  <UpdateLocation ref="table.pick.list.part.target"/>
</Action>
```

They both use different Location values and these are defined as:

```xml
<SimpleFrame id="frame.pick.list.part.source.add"
  mode="search" width="200" height="300">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Part Search Picker"/>
  <Wizard>
    <Extend ref="wizard.pick.list.part"/>
  </Wizard>
</SimpleFrame>
```
<ClientDoer class="com.ptc.core.ca.co.clientpicker.
CoObjectListPickerAddOkActionDoer"/>

</Wizard>
</SimpleFrame>

<SimpleFrame id="frame.pick.list.part.source.replace"
mode="search" width="200" height="300">
<Extend ref="common:simpleframe.template"/>
<Label resource="Part Search Picker"/>
<Wizard>
<Extend ref="wizard.pick.list.part"/>
<ClientDoer class="com.ptc.core.ca.co.clientpicker.
CoObjectListPickerReplaceOkActionDoer"/>
</Wizard>
</SimpleFrame>

**Note:** The only difference between the two is for the name of the ClientDoer class. Actually, these two definitions are prime candidates for refactoring into a single definition that uses some context switch to determine which ClientDoer is used. Regardless, they both extend wizard.pick.list.part which defines a single step search-based picker. It is defined as follows:

<Wizard id="wizard.pick.list.part">
<Label resource="Part Search Picker"/>
<WizardStepElements>
<SimpleLayout>
<LayoutRow id="rowSep1">
<Insert ref="common:separator.horizontal"/>
</LayoutRow>
<LayoutRow id="rowPropertyPanel">
<SimplePropertyPanel id="search.panel.part"
type="%[WTPart]" mode="search">
<Insert ref="search:search.panel.row.property.name"/>
<Insert ref="search:search.panel.row.property.number"/>
<Insert ref="search:search.panel.row.property.source"/>
<LayoutRow id="rowSearchAction">
<Insert ref="search:search.action.alt"
context.tableId="search.table.part"/>
</LayoutRow>
<Insert ref="search.panel.part.model.doer"/>
</SimplePropertyPanel>
</LayoutRow>
<LayoutRow id="rowSep2">
<Insert ref="common:separator.horizontal"/>
</LayoutRow>
<LayoutRow id="rowTable">
<Insert ref="search.table.part" mode="view"/>
</LayoutRow>
</SimpleLayout>
</WizardStepElements>
<ModelDoer class="%common:NullModelDoer"/>
<Insert ref="wizard.pick.action.list"/>
</Wizard>

This in turn, defines the criteria screen for the search (the SimplePropertyPanel search.panel.part). The insert of search:search.action defines the action to process.
The criteria page input and produce the proper filtered query result in the search.table.part definition.

The insert of wizard.pick.action.list at the end of the Wizard definition provides the correct set of buttons to drive the final selection operation and return all selected entries to the UpdateLocation supplied by the launching popup.

**Search Patterns**

This is the result of using the Search tab for the alias com.ptc.folder:first.page.view. This is very similar to what we saw used in the multiple value picker except that this is used in the primary window as a portal from which operations can be launched instead of as merely a selector for another page. We will focus on only two things here: The Type drop-down list plus the Change Type link, which allows us to reconfigure the page for other object definition criteria screens and the last line of the page which controls a paging mechanism. Note that you should explore all of the search pages defined in the samples – config_search.xml, config_search2.xml, config_search3.xml and config_search4.xml. They all have slightly different ways to approach the generation of the drop-down menu and/or the criteria page.
The definition for this page can be found at com/ptc/part/part_search.xml:

```xml
<SimpleFrame id="page.search.part">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Search"/>
  <SimpleLayout>
    <LayoutRow id="rowNav">
      <Insert ref="navigation:navbar2" colspan="2"/>
    </LayoutRow>
    <LayoutRow id="2">
      <Insert ref="search.panel.part"/>
    </LayoutRow>
  </SimpleLayout>
</SimpleFrame>
```
Note: This page is produced in several parts – LayoutRow rowNav calls out the navigation bar; LayoutRow 2 calls out the object type / search criteria panel; LayoutRow 4 calls out the result table; LayoutRow 5 calls out the paging actions for the search result table. We will only inspect the object type / search criteria panel and the paging actions, as we have already covered the other aspects in other sections.

The object type / search criteria panel is defined by search.panel.part which is in the file part_search.xml in com/ptc/part.

This SimplePropertyPanel defines two “chooser” references, depending upon the applid currently running. “search.panel.row.type.chooser.link” produces a drop-down list of specified object types and a link that will redisplay the page using the newly selected object type to be the basis of the search request, which will bring in the appropriate criteria for that new object type.

“search.panel.row.type.chooser.list” produces a drop-down list of specified object types and will activate the redisplay automatically when the drop-down value is changed. Both types have been created to explore the usability issues of each. Some people don’t like to have to press the “change type” link and some people dislike having drop-down lists invoking actions without the intervening “do it” click. Choose the one you like and use it consistently throughout your applications, or, better yet, always use both and then have a context switch that would allow the appropriate model to be used. To switch from one to the other, merely change the context switch setting (like changing applid on the URL).
We will look at `search.panel.row.type.chooser.link` in `config_search.xml` in `com/ptc/search`.

```xml
<LayoutRow id="search.panel.row.type.chooser.link">
    <Insert ref="search.action.type.label"/>
    <Insert ref="search.action.list.type">
        <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/location2.jsp"/>
    </Insert>
    <Insert ref="search.action.type.change"/>
</LayoutRow>
```

The `search.action.list.type` is the definition for the drop-down list of available object types.

```xml
<ActionList id="search.action.list.type">
    <Insert ref="search.action.type.part"/>
    <Insert ref="search.action.type.doc"/>
    <Insert ref="search.action.type.folder"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/location.jsp"/>
</ActionList>
```

This is a fixed list which inserts entries from each of the desired object types (in our case, part, doc and folder). To extend this, merely add more `Insert` elements to new object definitions, each similar in nature to “search.panel.part” above. We’ll look at `search.action.type.part`, also defined in `config_search.xml`.

```xml
<Action id="search.action.type.part">
    <Label resource="Part"/>
    <ToolTip resource="Part"/>
    <Location ref="com.ptc.part:page.search.part"/>
    <ActionHandler class="com.ptc.core.ca.co.client.search.CoSearchTypeActionHandler"/>
</Action>
```

This action points to the specific Location for producing the Part-based search page (`com/ptc/part:page.search.part`) and specifies a specific `ActionHandler` that makes the Location available to the “Change Type” link defined in `search.action.type.change`.

```xml
<Action id="search.action.type.change">
    <Label resource="Change Type"/>
    <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
    <ClientDoer class="com.ptc.core.ca.web.client.search.WebSearchTypeActionClientDoer"/>
    <LocationChooser element="search.action.list.type"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp"/>
</Action>
```

The `LocationChooser` for this action points back at our defined `ActionList` with all the appropriate object type search page actions. The rest of the configuration makes this change the page as indicated by the user in the selected drop-down entry.
The paging mechanism allows for smaller chunks to be returned from a query result, providing for smaller transmissions between the server and browser as well as for providing a way to prevent millions of objects from being returned as a result of a less than appropriate set of criteria. The paging link(s) for this page are defined as “search.page.actions” in config_search.xml.

The four defined actions access some information returned in any Info*Engine task which provides paging information. From this information, DCA constructs the necessary links which will allow the user to move forwards and backwards in the result set.
There are other methods which have been provided. If you use the search links for the part, doc and folders, you will see other types of paging behavior. Again, choose the user model you prefer for your application and implement it consistently throughout your applications with an ElementGroup that inspects the context information for the value of a particular switch.

```xml
<ElementGroup id="std.paging">
  <Insert ref="paging.style.1" if.context.paging="1"/>
  <Insert ref="paging.style.2" if.context.paging="2"/>
  <Insert ref="paging.style.3" if.context.paging="3"/>
</ElementGroup>
```

Then use `<Insert ref=":std.paging"/>` in all your search page definitions.

**Sample XML Configuration File**

**Note:** Some of the strings are broken for readability.

Starting configuration to produce the simple list of lifecycle names.

```xml
<?xml version="1.0" standalone="no"?>
<Repository package="com.ptc.lifecycles">
  <Context>
    <Synonym id="TARGET_OBJECT" value="wt.lifecycle.LifeCycleTemplate"/>
    <Synonym id="LABEL" value="Lifecycles"/>
    <Synonym id="TOOLTIP" value="Windchill Lifecycle Templates"/>
  </Context>

  <Context>
    <SimpleFrame id="simpleframe.template" separatorType="horizontal">
      <Insert ref="link.cssInclude"/>
      <Insert ref="script.javascriptInclude"/>
    </SimpleFrame>

    <Link id="link.cssInclude" resource="wtcore/css/com/ptc/core/ca/web/misc/DCADefault.css">
      <Insert ref="renderer.cssLink"/>
      <ModelHandler class="com.ptc.core.ca.co.client.frame.CoHeaderResourceModelHandler"/>
    </Link>

    <Script id="script.javascriptInclude" resource="wtcore/js/com/ptc/core/ca/web/misc/main.js">
      <Insert ref="renderer.script"/>
      <ModelHandler class="com.ptc.core.ca.co.client.frame.CoHeaderResourceModelHandler"/>
    </Script>

    <Renderer id="renderer.cssLink" class="com.ptc.core.ca.web.client.frame.WebCssLinkRenderHandler"/>
  </Context>
</Repository>
```
<Renderer id="renderer.script"
   class="com.ptc.core.ca.web.client.frame.
   WebScriptRenderHandler"/>
</Context>

<Context type="%TARGET_OBJECT">
  <Property id="name">
    <PropertyHandler class="com.ptc.core.ca.web.client.property.
    string.WebStringPropertyHandler"/>
    <Renderer class="com.ptc.core.ca.web.client.property.
    string.WebStringRenderHandler"/>
    <Label resource="Name:"/>
    <Need attribute="name"/>
  </Property>
</Context>

<Context type="%TARGET_OBJECT">
  <SimpleTable id="table" selectorType="multi">
    <SelectAllPrompt resource="All"/>
    <Type type="%TARGET_OBJECT"/>
    <Label resource="%LABEL"/>
    <SimpleColumn show="name">
      <Label resource="name"/>
    </SimpleColumn>
    <ModelDoer class="com.ptc.core.ca.co.client.doer.
    CoDefaultSimpleModelTaskDoer">
      <Task nextOperation="STDOP|com.ptc.windchill.view"
         taskDelegate="dca-query-objects">
        <TaskParameter name="search_type" value="%TARGET_OBJECT"/>
        <TaskParameter name="where" value="name='*'"/>
      </Task>
    </ModelDoer>
  </SimpleTable>
</Context>

<Context>
  <SimpleFrame id="main">
    <Extend ref="simpleframe.template"/>
    <Label resource="%LABEL"/>
    <ToolTip resource="%TOOLTIP"/>
    <SimpleLayout>
      <LayoutRow id="row5">
        <Insert ref="table" type="%[TARGET_OBJECT]"/>
      </LayoutRow>
    </SimpleLayout>
  </SimpleFrame>
</Context>
</Repository>
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating the Example</td>
<td>3-2</td>
</tr>
<tr>
<td>Page Source of Sample</td>
<td>3-3</td>
</tr>
<tr>
<td>Gateway JSP Page</td>
<td>3-3</td>
</tr>
<tr>
<td>Refactoring the Sample Links Page for Reuse</td>
<td>3-5</td>
</tr>
<tr>
<td>Every DCA WEB Generated HTML Page is a Form</td>
<td>3-6</td>
</tr>
</tbody>
</table>
Creating the Example

To create a simple web page that displays a hyper link to the PTC corporate web page do the following:

1. **Note:** xml is case sensitive. Create the following xml file. It can have any name but must have the xml suffix:

```xml
<Repository package="com.ptc">
   <Import package="com.ptc.common"/>
   <Context>
      <SimpleFrame id="links">
         <Extend ref="common:simpleframe.template"/>
         <Label resource="PTC Links"/>
         <ToolTip resource="Links to PTC pages"/>
         <SimpleLayout>
            <LayoutRow>
               <HyperLink>
                  <HRef resource="http://www.ptc.com/"/>
                  <ToolTip resource="Jump to PTC corporate web page"/>
                  <Image resource="http://www.ptc.com/images/common/front_page_logo.gif"/>
                  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/primitive/hyperLink.jsp"/>
               </HyperLink>
            </LayoutRow>
         </SimpleLayout>
      </SimpleFrame>
   </Context>
</Repository>
```

2. Place the xml file in the $(wt.home)/codebase/registry/dca folder.

3. Restart your JSP engine.

4. Type the following URL:

![Image of PTC Links in Microsoft Internet Explorer](image-url)
Page Source of Sample

<!-- HTML code here -->

Gateway JSP Page

Every DCA WEB URL uses /wtcore/jsp/com/ptc/core/ca/web/gw/gw.jsp. All DCA WEB URLs must go through the gw.jsp and have an alias parameter on the URL. In the example shown, the alias is of the form alias=package:id, where package is com.ptc defined of the <Repository> element and id is links ID of SimpleFrame element.
Alias Passed on gw.jsp does not Resolve to Legal Configuration

If the alias does not resolve to a legal configuration, a page will be generated without any visible elements. Viewing the page source should identify the error. For example, entering the following URL "http://<yourhost>/Windchill/wtcore/jsp/com/ptc/core/ca/web/gw/gw.jsp?alias=doesnotexist" will fail. Viewing the page source will show:

```
<HTML>
<HEAD>
<TITLE>Internal Error</TITLE></HEAD>
<BODY>

<SCRIPT LANGUAGE="JavaScript1.2">
popupErrorDialog("ERROR: Can not find configuration with id=doesnotexist for context=()");
</SCRIPT>
</BODY>
</HTML>
```

How to Show Stack Trace in a Page When an Error Occurs

A more complete error listing can be shown directly in the page by turning debugging on. Add the following line to the wt.properties file and restart the MethodServer and servlet engine.

```
com.ptc.core.ca.web.client.gw.debugMode=true
```

The Alias Must Resolve to a Frame Element Defined in Configuration

Every DCA WEB page must have a frame element (SimpleFrame or CompositeFrame elements) as its root. If a configuration alias is specified that resolves to another element type, the following error will display in the browser when viewing the source:

```
<HTML>
<HEAD>
<TITLE>Internal Error</TITLE></HEAD>
<BODY>

<SCRIPT LANGUAGE="JavaScript1.2">
popupErrorDialog("JSP Exception");
</SCRIPT>
</BODY>
</HTML>
```

With debugging turned on, a CastClassException will display if the element is not a frame:

```
Internal Error
```

JSP Exception
Refactoring the Sample Links Page for Reuse

A better approach would allow reuse of the hyper link element that points to the PTC home page. The following changes to the sample xml file allow the element to be referenced and reused in multiple places. To make an element so it can be referenced by another element, it must be a top-level element in a Context. As a top-level element, it can be referenced using its ID, in this case "home", by using the insert element. The hyper link element with id equals home is now referenced in multiple frames. For elements defined in other packages, the reference ID would need to be "com.ptc:home".

```xml
<?xml version="1.0" standalone="no"?>
<Repository package="com.ptc">
  <Import package="com.ptc.common"/>
  <Context>
    <HyperLink id="home">
      <HRef resource="http://www.ptc.com/"/>
      <ToolTip resource="Jump to PTC corporate web page" />
      <Image resource="http://www.ptc.com/images/common/front_page_logo.gif"/>
      <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/primitive/hyperLink.jsp"/>
    </HyperLink>
    <SimpleFrame id="linksreuse">
      <Extend ref="common:simpleframe.template"/>
      <Label resource="PTC Links"/>
      <ToolTip resource="Links to PTC pages"/>
      <SimpleLayout>
        <LayoutRow>
          <Insert ref="home"/>
        </LayoutRow>
      </SimpleLayout>
    </SimpleFrame>
    <SimpleFrame id="altlinksreuse">
      <Extend ref="common:simpleframe.template"/>
      <Label resource="Alternative PTC Links"/>
      <ToolTip resource="Links to PTC pages"/>
      <SimpleLayout>
        <LayoutRow>
          <Insert ref="home"/>
        </LayoutRow>
      </SimpleLayout>
    </SimpleFrame>
  </Context>
</Repository>
```
Every DCA WEB Generated HTMLPage is a Form

Every DCA WEB generated page is generated as a form. When sub elements are processed recursive down from the frame element, any of the sub elements could add input fields that require that a POST can be run from the page. The frame element cannot anticipate the requirements of the sub-elements so all frames are created as forms with METHOD="POST. The current default frame renderers are:

<table>
<thead>
<tr>
<th>Original URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same page using insert. The generated page source is the same.</td>
</tr>
<tr>
<td>An alternative page that also uses insert to reference &quot;home&quot; hyperlink element</td>
</tr>
</tbody>
</table>

SimpleFrame | /wtcore/jsp/com/ptc/core/ca/web/frame/simpleFrame.jsp
CompositeFrame | /wtcore/jsp/com/ptc/core/ca/web/frame/compositeFrame.jsp
The configuration system provides the capability to completely define an application in a set of XML files that is easy to customize and reuse. Configuration element definitions can be used to completely define all presentation elements and the interaction between the elements.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Overview</td>
<td>4-2</td>
</tr>
<tr>
<td>Configuration Syntax</td>
<td>4-2</td>
</tr>
<tr>
<td>Configuration Repository</td>
<td>4-3</td>
</tr>
<tr>
<td>Configuration Example #1</td>
<td>4-3</td>
</tr>
<tr>
<td>Configuration Example #2</td>
<td>4-4</td>
</tr>
<tr>
<td>Context</td>
<td>4-5</td>
</tr>
<tr>
<td>Attribute Values</td>
<td>4-6</td>
</tr>
<tr>
<td>Reference Resolution Rules</td>
<td>4-7</td>
</tr>
<tr>
<td>Conditional Configuration Expression if.xxx, ifnot.xxx, and if.................</td>
<td>4-14</td>
</tr>
<tr>
<td>Reloading Configuration Files</td>
<td>4-14</td>
</tr>
</tbody>
</table>
Configuration Overview

- A complete list of presentation components (frames, dialogs, tables, and buttons…).
- Actions to configure the interaction of components.
- Doers to bind to the data access and manipulation layer.
- Info*Engine elements functioning as client-server data containers to bind to LWC types and attributes.
- Configuration element definitions that can easily reuse existing configuration elements built in other applications, or reuse core application configuration elements.
- Applications developed with this configuration system can facilitate the following:
  - Reuse in a consistent manner by R&D application developers, as well as a more consistent and rapid method for customization of the applications in the field.
  - The ability of the configuration system to bind to the definition of new types and attributes defined in the Type Mechanism (LWC), which allows rapid customization of the applications at the customer site.
  - Rapid customization, beyond understanding new types, is supported. Introducing new customer site-specific behavior, either in the operations available or the how components are displayed in existing applications, is also possible.

Configuration Syntax

All the UI configuration definitions are specified in XML. A basic understanding of XML and DTD syntax is necessary to understand the XML configuration files and the configuration specification documentation.

For a more complete description of the XML syntax, see *Extensible Markup Language (XML) 1.0 (Second Edition)*.

While no grammar is defined in DTD files, the definition of the legal configuration elements closely resembles DTD syntax.

See “Configuration Specification Reference”, on page A-1, for sample syntax and the definition of configuration elements.

Caution: Element names are case sensitive. Therefore, "insert" is not the same as the correct element name of "Insert". Be careful to correctly specify the case of an element tag. The format is uppercase for the first letter of the word and lowercase for the rest of the word.
Configuration examples will frequently be formatted to increase their readability. This will result in breaking string values when they do not fit into one line of the page thus making XML code snippets technically invalid.

**Configuration Repository**

The configuration repository is the collection of all XML files located under the file folder defined by the "com.ptc.core.ca.co.common.config.directory" property in wt.properties file. If no property exists, the default value for the file folder is $(wt.home)/conf/dca. This folder hierarchy is processed recursively to find all the XML files.

A Repository configuration element must be the root element of each and every file. If not, a warning will be logged and the file will be skipped. All valid XML files will be processed to create the runtime set of legal configuration objects.

**XML Files and Packages**

The separation of the configuration repository into multiple files allows a logical partitioning of the UI configuration information. Each XML file defines "UI configuration" elements that exist in a package. A package is the named collection of these unique identified elements. While each XML file can only specify elements for a single package, the elements visible in the namespace of the package can be separated into multiple files. Every top-level configuration element has a ID in form of "package:id". Children of Context elements are considered to be top-level configuration elements.

**Configuration Example #1:**

[extracted from com\ptc\enterprise\config_revision_controlled.xml]

```xml
<Repository package="com.ptc.enterprise">
  <Import ref="com.ptc.common"/>
  <Context>
    <Synonym id="RevisionControlled" value="wt.enterprise.RevisionControlled"/>
  </Context>

  <Context type="%RevisionControlled">
    <Property id="number">
      <Extend ref="common:property.String.template"/>
      <Label resource="Number:"/>
      <Need attribute="number"/>
    </Property>
  </Context>
</Repository>
```

The **Repository** element must be the root element in configuration file for the file to be valid.
The ID of the property element is "number". The fully qualified ID of the element based on namespace of the package is "com.ptc.enterprise:number".

The Import element allows short names to be used to refer to the elements defined in another package. Thus, instead of specifying "com.ptc.common:property.String.template", we can specify "common:property.String.template".

A Synonym is defined in the context where no dimensions are specified. A property, however, is defined in context with the type dimension set to be equal to wt.enterprise.RevisionControlled.

The Extend element lets us define a property element by extending an existing configuration fragment and overriding some of its features. By defining the property element "property.String.template", this allows all string-based properties to inherit the correct string handler and renderer. It also allows a modified configuration element to inherit behavior and override behavior as well.

Common Configuration Templates in DCA Demo

$(wt.home)/codebase/registry/dca/com/ptc/common/config_common_templates.xml

$WT_HOME/codebase/registry/dca/com/ptc/common/config_common_actions.xml

Configuration Example #2

[Extracted from com\ptc\doc\config_doc.xml]

<Repository package="com.ptc.doc">
  <Import package="com.ptc.common" />
  <Import package="com.ptc.enterprise" />

  <Context>
    <Synonym id="WTDocument" value="wt.doc.WTDocument" />
  </Context>

  <Context type="%WTDocument">
    <SimpleLayout id="layout.view" type="%[WTDocument]">
      <LayoutRow id="1">
        <Insert ref="panel.view.actions" />
      </LayoutRow>
      <LayoutRow id="2">
        <Insert ref=":name" />
        <Insert ref=":number" />
      </LayoutRow>
    </SimpleLayout>
  </Context>
</Repository>

The Insert element allows a configuration element aggregate another configuration element in its definition. In this example we include "name" property defined in the previous example as part of a layout row. The ability to
reuse configuration definition provides all the standard reuse benefit - ability to modify definition of "name" in one place and have all references reflect the change.

Context

Dimensions are name-value pairs that form context of the configuration element. Configuration elements usually inherit context from their parents but can also augment it by specifying additional dimensions.

DCA supports a set of well-known dimensions as well as user-defined dimensions.

Well-known Dimensions

To augment context with well-known dimensions, they can simply be specified as attributes. For instance, in the following examples the layout element will have type dimension set to \texttt{wt.doc.WTDocument}.

```xml
<Repository package="com.ptc.doc">
  <Context type="wt.doc.WTDocument">
    <SimpleLayout id="layout.view">
      ...
    </SimpleLayout>
  </Context>
</Repository>

<Repository package="com.ptc.doc">
  <Context>
    <SimpleLayout id="layout.view" type="wt.doc.WTDocument">
      ...
    </SimpleLayout>
  </Context>
</Repository>
```

DCA recognize the following set of well-known dimensions:

- appId
- decoration
- roleId
- type
- mode

User-defined Dimensions

User-defined dimensions can also be used to augment the context of the element. However, they must be specified as attributes with the \texttt{context} prefix. For instance:

```xml
<Repository package="com.ptc.doc">
  <Context color="blue">
    <SimpleLayout id="layout.view">
      ...
    </SimpleLayout>
  </Context>
</Repository>
```
The following two statements are equivalent because any well-known dimension can also be referenced with the user-defined syntax; hence, "type" and "context.type" are the same dimension.

```
<SimpleLayout id="layout.view" context.type="blue">
<SimpleLayout id="layout.view" type="blue">
```

### Attribute Values

Attribute values of configuration elements can be specified explicitly as string constants. For instance,

```
<SimpleLayout id="layout.view" type="wt.doc.WTDocument">
```

In this example the value of the type attribute is `wt.doc.WTDocument`.

When the same string is frequently used, a synonym to it can be used instead. Thus, the change in the synonym will result in the change in many places.

```
<Synonym id="WTDocument" value="wt.doc.WTDocument" />
...
<SimpleLayout id="layout.view" type="%[WTDocument]">
```

or

```
<SimpleLayout id="layout.view" type="%WTDocument">
```

In this example, the value of the type attribute is still `wt.doc.WTDocument`.

Not only synonyms can be used to initialize attribute values. A variety of sources can be used, too. In the following example,

```
<SimpleLayout id="layout.view" type="%[context#a].%[frame#b]">
```

the value of the type attribute is still `wt.doc.WTDocument` when context dimension a has a value of `wt.doc`, and frame parameter b has a value of `WTDocument`. See “Configuration Specification Reference”, on page A-1, for more details.
Attribute Sources

Attribute values can come from the following sources:

- synonym
- dimension
- session (conceptual session)
- request (conceptual request)
- frame (conceptual frame)
- element value (value of the enumerator element)
- attribute value of Info*Engine element

For instance,

```xml
<SimpleLayout id="layout.view" bgcolor="%[session#color]"
  fgcolor="%[frame#color]">
</SimpleLayout>
```


Reference Resolution Rules

To construct a configuration definition for a given element, the configuration system needs to resolve references to other configuration elements. To do that, DCA takes into account scoping and context rules.

Scoping rules are evaluated before context rules, and limit the set of the configuration specifications to look for a fragment. Context rules are applied after the scope has been determined.

Scoping Rules

Scoping rules define where to look for the configuration fragment describing sub-elements.

Same Package Scope

To reference an element in the same package, the element’s ID must be specified without the package information. For instance

```xml
<LayoutRow id="2">
  <Insert ref="name" />
</LayoutRow>
```
Package Scope

To reference an element in a specified package, the element’s ID must be specified with the package information. For instance

```xml
<LayoutRow id="2">
  <Insert ref="com.ptc.part:name" />
</LayoutRow>
```

In this example com.ptc.part is a package name and name is the element ID in the package.

Global Scope

Besides the "package:id" and "id" forms of identifier referencing, a third form of referencing exists in the format of ":id". It informs the configuration system that it should look for the element in all packages.

```xml
<LayoutRow id="2">
  <Insert ref=":com.ptc.part:name" />
</LayoutRow>
```

The ":id" form should be used only when "id" represents some general entity for which dimensions are required to completely resolve the reference.

Context Rules

DCA takes context rules into account after applying scoping rules. The purpose of the context rules are to find the best configuration fragment in the given scope. For instance, we can define an element with ID dialog.view in many packages for many different object types:

- in package "com.ptc.folder" for type "com.ptc.folder.Folder"
- in package "com.ptc.part" for type "com.ptc.part.WTPart"

Cases exist where we will only know the object type at runtime (row in a heterogeneous table). In these cases we need to use the type of the object and the ID of the configuration element to retrieve correct configuration element definition. This allows a set of configuration elements (all elements with ID dialog.view) to be specified with the correct concrete configuration found based on the dimensions.

In the example above, the context dimension is the object type. The view property page of the folder object is defined with the type context folder. When the folder is displayed as a row in a table and an action runs to display its property page, the type of the object and its ID are used to find the correct element in configuration describing UI.
The sample below shows the "name" property for folder defined in the context with type dimension.

[Extracted from com\ptc\folder\config_part.xml]

```xml
<context type="%[Folder]"/>
<property id="name">
    <extend ref="common:property.name.template"/>
</property>
</context>
```

The following is how this property can be used to present names of parts and documents:

[Extracted from com\ptc\folder\config_folder.xml]

```xml
<simpleTable id="table" selectorType="multi">
    <selectAllPrompt resource="All"/>
    <insert ref="browser.actions"/>

    <type type="%[part:WTPart]"/>
    <type type="%[doc:WTDocument]"/>
    <type type="%[SubFolder]"/>

    <simpleColumn show=":name">
        <label resource="Name"/>
    </simpleColumn>

    ...
```

**Best Configuration Fragment**

The configuration system always resolves a reference to a configuration element in some context. Let’s call it dimension set A. The referenced configuration element is defined in some context, too. Let’s call it dimension set B. Only dimensions specified on the `Context` element are included in it. A configuration fragment is considered to be the best when B is a sub-set of A.

*type* dimension is an exception in the sense that the type from A does not have to be exact to the type from B. The fragment is considered to be the best when type from B is a sub-type from A.

For every correct reference there must be a single best fragment for DCA to produce an exception.

1.

```xml
<context type="wt.part.WTPart" color="yellow">
    <layoutRow id="sample">
        ...
    </layoutRow>
</context>
```

B1 = {(type, wt.part.WTPart), (color, yellow)}

2.

```xml
<context type="wt.part.WTPart" color="white">
    <layoutRow id="sample">
        ...
    </layoutRow>
</context>
```
B2 = \{(\text{type}, \text{wt.part.WTPart}), (\text{color}, \text{white})\}

3. 

The following results in the exception, as more than one best fragment (1 and 2) are found.

\[
\text{A1} = \{(\text{type}, \text{wt.part.WTPart})\}
\]

To correct the problem, the \text{color} dimension must be specified.

\[
\text{A2} = \{(\text{type}, \text{wt.part.WTPart}), (\text{color}, \text{white})\}
\]

This makes A2 equal to B2.

The following works perfectly as there is a single match (fragment 3).

\[
\text{A3} = \{(\text{type}, \text{wt.doc.WTDocument}), (\text{color}, \text{white})\}
\]

This makes A3 equal to B3.

The following will not work as A3 is a sub-set of B3.

\[
\text{A3} = \{(\text{type}, \text{wt.doc.WTDocument})\}
\]

But the next example will work as B3 is a sub-set of A4.

\[
\text{A4} = \{(\text{type}, \text{wt.doc.WTDocument}), (\text{color}, \text{white}), (\text{size}, \text{large})\}
\]
When `com.xxx.SpecialDocument` is a sub-type of the `wt.doc.WTDocument` the following will work too:

```xml
<SimpleLayout type="com.xxx.SpecialDocument"
  context.color="white">
  <Insert ref="sample"/>
</SimpleLayout>
```

A5 = \{(type,com.xxx.SpecialDocument), (color,white)\}

There are cases when a dimension is specified on a top-level element - a child of `Context` element.

```xml
<Context type="wt.part.WTPart" color="yellow">
  <LayoutRow id="sample" context.size="large">
  ...
  </LayoutRow>
</Context>
```

In such a case, this dimension is not part of the set B.

B4 = \{(type,wt.part.WTPart), (color, yellow)\}

**Referencing a Frame**

Frame elements can be referenced similarly to the other elements. The following will refer to a frame in the same package:

```xml
<Action id="action.open_in_new_frame" needSelection="true">
  <Extend ref="action.newFrame.template"/>
  <Label id="Open in new window"/>
  <Location ref="page.view"/>
</Action>
```

The frame can also be referenced in the different package:

```xml
<Action id="action.open_in_new_frame" needSelection="true">
  <Extend ref="action.newFrame.template"/>
  <Label id="Open in new window"/>
  <Location ref="com.ptc.part:page.view"/>
</Action>
```

or in any package:

```xml
<Action id="action.open_in_new_frame" needSelection="true">
  <Extend ref="action.newFrame.template"/>
  <Label id="Open in new window"/>
  <Location ref=":page.view"/>
</Action>
```

However, there is a difference in the set of dimensions that can be used to locate a frame. The permanent frame can only be specified in the context containing the following well-known dimensions:

- appId
- decoration
- roleId
All other dimensions will be ignored. The temporary frame, however, can be defined in any context.

The following example configures an action that creates a new frame with decoration dimension set to no.

```xml
<Action id="action.open_in_new_frame" needSelection="true">
  <Extend ref="action.newFrame.template"/>
  <Label id="Open in new window"/>
  <Location ref="page.view" decoration="no"/>
</Action>
```

Frame Configuration

DCA needs to know the context to find frame configuration. To find it, DCA creates new context and populates it with dimensions of the Location element that references the frame.

For security reasons, DCA treats roleId differently than other dimensions. The roleId dimension is never externalized as part of an address of DCA elements. As a result, DCA Web will ignore the roleId dimension value of the Location element.

DCA employs the following algorithm to find a configuration of a frame element based on its address:

1. Find context of the frame.
2. Create temporary context using appId and typeId dimensions from frame context.
3. Using the temporary context, find an appropriate so-called role determinator.
4. Find a role using role determinator.
5. Populate frame context with roleId dimension.
6. Using frame context, find a frame configuration.

Role Determinator

Role determinator is a class that implements the CoRoleDeterminator interface defined in com.ptc.core.ca.co.client.misc package.
Role determinator needs to implement the `getRole` method that returns a string to be used as a value of a `roleId` dimension by DCA.

```java
public String getRole( CoContext context,
            CoConfigFeature config,
            CoEnvData envData )
    throws WTException;
```

Here, `context` object contains the values of the appId and typeId dimensions, `config` object describes the RoleDeterminator configuration element, and `envData` provides access to the DCA environment.

DCA Web provides an abstract implementation of the `CoRoleDeterminator` interface to give access to web-specific implementation of the DCA environment.

Caution: Do not use `WebRoleDeterminator` unless you need access to servlet request or other servlet or jsp specific objects. Use `CoRoleDeterminator` instead.

**Role Determinator Configuration**

Multiple role determinators can be specified to enable different role resolutions for different applications. The typical way to do it would be one of the following:

```xml
<Context appId="myApp1">
    <RoleDeterminator id="RoleDeterminator"
        class="MyRoleDeterminatorClass1"/>
</Context>

<Context appId="myApp2">
    <RoleDeterminator id="RoleDeterminator"
        class="MyRoleDeterminatorClass2"/>
</Context>
```

or

```xml
<Context>
    <RoleDeterminator id="RoleDeterminator"
        class="MyRoleDeterminatorClass1"
        appId="myApp1"/>

    <RoleDeterminator id="RoleDeterminator"
        class="MyRoleDeterminatorClass2"
        appId="myApp2"/>
</Context>
```

typeId dimension can also be used.
Caution: ID of the RoleDeterminator configuration element must always be RoleDeterminator or it will be ignored.

Conditional Configuration Expression if.xxx, ifnot.xxx, and if

The if.xxx, ifnot.xxx, and if are attributes, not element definitions. These attribute definitions allow the conditional construction of configuration elements based on dimension values at runtime.

For instance, one might want to use a header and footer for primary pages but will not want them for temporary pages. The following is a solution to this problem:

```xml
<SimpleLayout>
  <LayoutRow id="row1" if.decoration="yes">
    <Insert ref="navbar"/>
  </LayoutRow>
  ...
</SimpleLayout>
```

In DCA Web, if a URL does not have a decoration=yes parameter, row1 will not be included in the configuration definition of the layout.

Reloading Configuration Files

DCA can automatically reload configuration files when they are changed. It is implemented for the convenience of developers to avoid frequent restarting of a jsp engine. The feature can also be used in production system but there is a significant performance penalty when it is enabled.

To enable automatic reloading of configuration files, set the following property in wt.properties file:

```
com.ptc.core.ca.co.common.config.dynamicUpdate=true
```

When DCA is asked to draw a frame, it first tries to find it in the cache. When found, DCA checks whether the frame was updated before the time when configuration files changed. When the frame was updated after the last change in configuration files, DCA discards it and creates a new one.

Caution: Modifications of configuration files in the middle of complex operations like wizard operations can cause exceptions. It is recommended that developers refresh their frames after making changes in configuration files.
This chapter discusses property elements, handlers, and renderers; and how they are used in DCA.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property Element</td>
<td>5-2</td>
</tr>
<tr>
<td>Property Handler</td>
<td>5-2</td>
</tr>
<tr>
<td>Property Renderer</td>
<td>5-3</td>
</tr>
<tr>
<td>Required Property</td>
<td>5-4</td>
</tr>
<tr>
<td>Current Supported Data Types in DCA Web</td>
<td>5-4</td>
</tr>
<tr>
<td>Customization that Effects Property Rendering</td>
<td>5-10</td>
</tr>
</tbody>
</table>
**Property Element**

In DCA, a Property element presents attributes of an Info*Engine element. Depending on the mode, a Property element can allow the user to change attributes of the Info*Engine element. The typical use of a Property element would be inside a property panel, a wizard, a table cell, or a tree node.

```xml
<SimpleFrame id="part.view" type="%[WTPart]" mode="view">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="View Properties"/>
  <Wizard>
    <Label resource="Properties of Part (View)"/>
    <WizardStepElements>
      <SimpleLayout>
        <LayoutRow id="1">
          <Property>
            <Label resource="Name:"/>
            <Need attribute="name"/>
            <PropertyHandler class="com.ptc.core.ca.web.client.
                          property.string.WebStringPropertyHandler"/>
            <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
                          property/string/string.jsp"/>
          </Property>
        </LayoutRow>
        <Insert ref=":number"/>
      </SimpleLayout>
    </WizardStepElements>
    <Insert ref="common:object.getForView.task"/>
    <Insert ref="common:wizard.actions"/>
  </Wizard>
</SimpleFrame>
```

A property element delegates most of its functions to a property handler.

**Property Handler**

One role of a property handler (PropertyHandler element) is to prepare a property model for a subsequent rendering. In DCA Web, a property handler is also responsible for mining an HTTP request for a user’s input to populate an appropriate Info*Engine element with new attribute values.

DCA Web provides a variety of property handler classes implemented to handle a different data types.

While DCA supports a variety of property handlers, only handlers extending com.ptc.core.ca.co.client.property.CoSimplePropertyHandler are implemented for this release of DCA Web. CoSimplePropertyHandler is responsible for the preparation of a model based on a single attribute, and for the attribute update in corresponding Info*Engine element.

To locate an Info*Engine element, CoSimplePropertyHandler uses a "simple" approach of searching for the nearest instance holder ancestor. Thus, a property
will be able to behave correctly when placed inside a property panel, wizard, table row, or a tree node. The property does not have to be a child of an instance holder ancestor. In fact, in many cases it is impossible. For instance, a property cannot be a child of a property panel, but it can be a child of a LayoutRow element that is a child of a property panel. When CoSimplePropertyHandler conducts a search for a nearest instance holder ancestor, it skips all Layout, LayoutRow, and List positioning elements until it finds an element holding an Info*Engine element.

It is believed that this "simple" approach covers a majority of the use cases. When this is not the case, alternative property handlers can be developed.

**Property Renderer**

In DCA Web, the role of the property renderer (Renderer element) is to output html required to display the property using the data prepared by the property handler. Any valid web-application resource can be used as a renderer, including jsp pages, servlets, and beans.

DCA Web provides a variety of both JSP and bean-based renderers implemented to work for a variety of different data types. Note that PTC recommends the use of JSP-based renderers only during the development cycle due to the extreme servlet engine overhead of invoking JSP pages as includes.

**Note:** Although the example directly references a particular resource, there is a global setting that indicates whether jsp or java-based renderers should be used. This allows a single switch to change the rendering approach. In order to effectively use this, the following pattern for specifying Renderer elements and Property elements should be consistently used.

**In a centrally located package:**

```xml
<Context>
    <Renderer id="renderer.action" resource="/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp"/>
</Context>

<Context rendererType="java">
    <Renderer id="renderer.action" class="com.ptc.core.ca.web.client.action.WebActionRenderHandler"/>
</Context>
```

**In your application description:**

```xml
<Property id="myprop">
    ...
    <Insert ref=":renderer.action"/>
</Property>
```
This way, you will use the renderer specified in the common package. Should you wish to, you can globally change all uses of a particular renderer by changing the common definition.

Also, if you wish to force a certain renderer to always be JSP or java-based, merely change the Renderer entry in both contexts in the common file to be identical. This way, regardless of the property setting, you will get the same rendering behavior.

**Required Property**

Simple property (all existing properties are simple) can be configured to indicate that it requires a value in create, update, and search mode.

```
<Property id="myprop" required="true"/>
```

By default, properties do not require value specification from a user. It is recommended, however, that you do not specify the attribute on the property definition. Instead, specify it when you include the property in the definition of some other element. This way you can reuse property definition for inclusion in element in different mode.

```
<Property id="myprop">
  ...
</Property>
```

```
<LayoutRow>
  <Insert ref=":myprop" required="true"/>
  ...
</LayoutRow>
```

**Current Supported Data Types in DCA Web**

The data types most commonly used to represent attribute values of I*E elements are supported in the DCA framework. The supported data types are rendered using tightly coupled renderer/handler pairs. The following list contains these pairs and what data type they render. Each handler can only accept input from the renderers specified. However, for viewing, you can use any renderer you like. This allows you to create graphical renderers that substitute graphical information for textual information (for example, whether or not an object is checked in or out of the vault).

**java.lang.Boolean**

*Property Handler*

```
com.ptc.core.ca.web.client.property.binary.WebBinaryPropertyHandle
```

*Java Renderers*

```
com.ptc.core.ca.web.client.property.binary.WebCheckBoxRenderHandle
```
com.ptc.core.ca.web.client.property.binary.WebDropDownListRenderHandler

com.ptc.core.ca.web.client.property.binary.WebRadioButtonsRenderHandler

**JSP Renderers**

/wtcore/jsp/com/ptc/core/ca/web/property(binary/checkbox.jsp
/wtcore/jsp/com/ptc/core/ca/web/property(binary/dropdownlist.jsp
/wtcore/jsp/com/ptc/core/ca/web/property(binary/radiobuttons.jsp

WebDropDownListRenderHandler and dropdownlist.jsp requires the specification of labels for true and false values through Option elements.

Examples:

```xml
< Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property/binary/checkbox.jsp"/>
< Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property/binary/dropdownlist.jsp">
  < Option theme="label" param="true" value="yes" />
  < Option theme="label" param="false" value="no" />
</Renderer>
< Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property/binary/dropdownlist.jsp">
  < Option theme="label" param="true" value="yes" />
  < Option theme="label" param="false" value="no" />
</Renderer>
```

**java.lang.Date**

**Property Handler**

com.ptc.core.ca.web.client.property.date.WebDateTimePropertyHandler

**Java Renderer**

com.ptc.core.ca.web.client.property.date.WebDateTimeRenderHandler

**JSP Renderer**

/wtcore/jsp/com/ptc/core/ca/web/property/date/datetime.jsp

WebDateTimeRenderHandler and datetime.jsp requires the specification of formats for date and time values via Option elements. Examples:

```xml
< Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property/date/datetime.jsp">
  < Option theme="dateFormat" param="date.style.pattern" value="dd-MM-yy" />
  < Option theme="dateFormat" param="time.style.pattern" value="hh:mm:ss a" />
</Renderer>
```
In addition to the specification of formats for data and time values, a date renderer can be configured to present a prompt for a user in create, edit, or search mode.

```xml
<Option theme="dateFormat" param="date.style.pattern" value="dd-MM-yy" />
<Option theme="dateFormat" param="date.style.prompt" value="mm-dd-yy" />
<Option theme="dateFormat" param="dateOnly" value="true" />
</Renderer>
```

DCA demo provides a variety of formats to choose from.

**Date formats:**

```xml
<ElementGroup id="dateFormat.default.template">
  <Option theme="dateFormat" param="date.style.pattern" value="MM-dd-yy" />
  <Option theme="dateFormat" param="date.style.prompt" value="mm-dd-yy" />
</ElementGroup>
<ElementGroup id="dateFormat.short.template">
  <Option theme="dateFormat" param="date.style.pattern" value="MM/dd/yy" />
  <Option theme="dateFormat" param="date.style.prompt" value="mm/dd/yy" />
</ElementGroup>
<ElementGroup id="dateFormat.medium.template">
  <Option theme="dateFormat" param="date.style.pattern" value="MM-dd-yy" />
  <Option theme="dateFormat" param="date.style.prompt" value="mm-dd-yy" />
</ElementGroup>
```

The options below are not suitable to be used in edit mode, so they do not define prompts:

```xml
<Option id="dateFormat.long.template" theme="dateFormat" param="date.style.pattern" value="MMM dd', yyyy" />
<Option id="dateFormat.full.template" param="date.style.pattern" value="MMMM dd', yyyy" />
```

```xml
<Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property/ date/datetime.jsp">
  <Option theme="dateFormat" param="date.style.pattern" value="dd-MM-yy" />
  <Option theme="dateFormat" param="dateOnly" value="true" />
</Renderer>
```
Time formats:

<ElementGroup id="timeFormat.default.template">
    <Option theme="timeFormat" param="time.style.pattern" value="hh:mm:ss a"/>
    <Option theme="timeFormat" param="time.style.prompt" value="hh:mm:ss AM/PM"/>
</ElementGroup>

<ElementGroup id="timeFormat.short.template">
    <Option theme="timeFormat" param="time.style.pattern" value="hh:mm a"/>
    <Option theme="timeFormat" param="time.style.prompt" value="hh:mm AM/PM"/>
</ElementGroup>

<ElementGroup id="timeFormat.medium.template">
    <Option theme="timeFormat" param="time.style.pattern" value="hh:mm:ss a"/>
    <Option theme="timeFormat" param="time.style.prompt" value="hh:mm:ss AM/PM"/>
</ElementGroup>

The options below are not suitable to be used in edit mode, so they do not define prompt:

<Option id="timeFormat.long.template" theme="timeFormat" param="time.style.pattern" value="hh:mm:ss a z"/>
<Option id="timeFormat.full.template" theme="timeFormat" param="time.style.pattern" value="hh:mm:ss 'o''clock' a zzz"/>

Internationalization

Since each locale expects an appropriate formatting for date and time, DCA recommends using internationalization capabilities when specifying formats. Instead of specifying formats themselves, DCA developers are expected to specify keys to external resources specified in .rbInfo files. See “Internationalization and Localization”, on page 25-1, for more details.

To internationalize date format param attribute needs to contain a key.

<ElementGroup id="dateFormat.default.template">
    <Option theme="dateFormat" param="date.style.pattern" value="MM-dd-yyyy[#pattern#]com.ptc.refimpl.co.generic.common.res.dcaDataTimeFormats:1"/>
    <Option theme="dateFormat" param="date.style.prompt" value="mm-dd-yyyy[#prompt#]com.ptc.refimpl.co.generic.common.res.dcaDataTimeFormats:2"/>
</ElementGroup>

In this example the keys refer to the values in the file.
A version of the file for a default locale can contain formats for default locales

1. value=MM-dd-yy
   1. comment=pattern

2. value=mm-dd-yy
   2. comment=prompt

while a version for Italy would contain locale specific formats:

1. value=dd-MM-yy
   1. comment=pattern

2. value=dd-mm-aa
   2. comment=prompt

Locale specific format must not be translated. The value of the key must remain in English and must be one of those that java understands.

Local specific prompt, on the other hand, must be translated. In the example above it’s mm-dd-yy for the default locale and dd-mm-aa for Italy.

### Enumerated Value

**Property Handler**

com.ptc.core.ca.web.client.property.enumerated.WebEnumeratedPropertyHandler

**Java Renderers**

com.ptc.core.ca.web.client.property.enumerated.WebEnumeratedDropDownListRenderHandler
com.ptc.core.ca.web.client.property.enumerated.WebEnumeratedDropDownList2RenderHandler
com.ptc.core.ca.web.client.property.enumerated.WebEnumeratedRadioButtonsRenderHandler

**JSP Renderers**

/ wtcore/jsp/com/ptc/core/ca/web/property/enumerated/dropdownlist.jsp
/ wtcore/jsp/com/ptc/core/ca/web/property/enumerated/dropdownlist2.jsp
/ wtcore/jsp/com/ptc/core/ca/web/property/enumerated/radiobuttons.jsp

dropdownlist2.jsp **renderer provides "No Selection" value when does not.**
**Hyper Link**

**Property Handler**

com.ptc.core.ca.web.client.property.hyperlink.
   WebHyperlinkPropertyHandler

**Java Renderers**

com.ptc.core.ca.web.client.property.hyperlink.
   WebHyperlinkRenderHandler
com.ptc.core.ca.web.client.property.hyperlink.
   WebContentUrlRenderHandler

**JSP Renderers**

/wtcore/jsp/com/ptc/core/ca/web/property/hyperlink/hyperlink.jsp
/wtcore/jsp/com/ptc/core/ca/web/property/hyperlink/contentUrl.jsp

**java.lang.Long**

**Property Handler**

com.ptc.core.ca.web.client.property.integer.
   WebIntegerPropertyHandler

**Java Renderer**

com.ptc.core.ca.web.client.property.integer.
   WebIntegerRenderHandler

**JSP Renderer**

/wtcore/jsp/com/ptc/core/ca/web/property/integer/integer.jsp

**java.lang.Double**

**Property Handler**

com.ptc.core.ca.web.client.property.real.WebRealPropertyHandler

**Java Renderers**

com.ptc.core.ca.web.client.property.real.
   WebRealRenderHandler
com.ptc.core.ca.web.client.property.real.
   WebRealWithUnitsRenderHandler

**JSP Renderers**

/wtcore/jsp/com/ptc/core/ca/web/property/real/real.jsp
/wtcore/jsp/com/ptc/core/ca/web/property/real/realwithunits.jsp
**java.lang.String**

Property Handler

```java
com.ptc.core.ca.web.client.property.string.
WebStringPropertyHandler
```

**Java Renderers**

```java
com.ptc.core.ca.web.client.property.string.
WebStringRenderHandler
com.ptc.core.ca.web.client.property.string.
WebTextAreaRenderHandler
```

**JSP Renderers**

```java
/wtcore/jsp/com/ptc/core/ca/web/property/string/string.jsp
/wtcore/jsp/com/ptc/core/ca/web/property/string/textarea.jsp
```

WebStringRenderHandler and string.jsp may have the following options:

```xml
<Option theme="input" param="size" value="<integer>"/>
<Option theme="input" param="maxlength" value="<integer>"/>
```

```xml
<Property>
  <Need attribute="name"/>
  <Option theme="input" param="size" value="50"/>
  <Option theme="input" param="maxlength" value="5"/>
  <Renderer class="com.ptc.core.ca.web.client.property.string.
WebStringRenderHandler"/>
</Property>
```

WebTextAreaRenderHandler and textarea.jsp need width and height that can be specified through Option mechanism

```xml
<Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property/
string/textarea.jsp"/>
<Option theme="size" param="width"  value="30"/>
<Option theme="size" param="height" value="10"/>
</Renderer>
```

**Customization that Effects Property Rendering**

You may need to customize some or all parts of an implementation to render a property. You may decide to do one or more of the following:

1. Modify supported implementation of a property renderer
2. Implement your own property renderer
3. Implement your own property handler
4. Implement your own property model
**Modify Existing Property Renderer**

**When to modify**

If there is a business need to render every property of a certain type differently than the supported renderer for that type, you can modify the existing renderer. This change will affect every property that has been configured to use the renderer.

**How to modify**

Modify how and what the renderer generates for html as long as the changes comply with the property handler submit rules which are usually:

- Keep the same number of input fields or select options
- Use the same addresses for the input fields or select options

**Extend Existing Property Handler**

**When to extend**

If there is a business need to render a particular property differently than the existing renderers, you need to create a new renderer. This requires the creation of a new property handler that you may decide to extend from an existing one.

**How to extend**

If the renderer produces output comparable with an existing renderer, it can be used with an existing handler. Remember, the handler mines requests for property data. If you wish to use an existing handler you have to:

1. Implement the renderer following the guidelines and using the template.
2. Extend the handler and override `canHandleInput` as follows:

```java
protected boolean canHandleInput( CoProperty property,
       WebEnvData web_envData )
    throws WTException {

    // Check to see if supported renderer of super class
    // is configured.
    boolean canHandle = super.canHandleInput(property, web_envData);

    if (!canHandle) {
        String renderer = WebElementUtils.getRenderer(property,
                CoConfigFeatureType.RENDERER, web_envData);
        canHandle = compare renderer resource with your resource
    }

    return canHandler;
```
3. Modify properties renderer and handler configuration.

If the renderer cannot be used with an existing handler you will need to perform the three steps above and override the `updateProperty` method of a Property Handler:

**How to override `updateProperty` method:**

Extend the existing Web handler and override `updateProperty`. Here is an example of what you would have to do to read a string value:

```java
public void updateProperty( CoProperty property,
WebEnvData web_envData )
throws WTException
{
    String[] prop_param_values =
        WebPropertyUtils.getPropertyParamValues(property,
        web_envData);
    String attr_value = null;
    attr_value = WebUtils.convertToUTF8(prop_param_values[0],
        web_envData);
    updateInstance(property, attr_value, web_envData);
}
```

**Create Completely New Property Handler**

**When to create:**

Usually you will create a completely new property handler when you need to support a new data type. You can also choose to create it when extending existing handlers does not satisfy your business needs.

If you have implemented a property model you will have to implement a new property handler as described in the *Implement Property Model* section.

**How to create:**

In DCA, to create a completely new property handler you can implement a class that extends one of the following classes:

```java
com.ptc.core.ca.co.client.property.CoPropertyHandler
com.ptc.core.ca.co.client.property.CoAbstractPropertyHandler
com.ptc.core.ca.co.client.property.CoSimplePropertyHandler
```

Most of the time you will extend `CoSimplePropertyHandler` class. Your handler may implement some methods to be called from a renderer. This is to avoid putting too much java code in renderers, making them complex and difficult to maintain. Use of static methods is recommended but regular methods can also be used.
Implement the following methods:

```java
protected void updateModel( CoProperty property,
    Info*Engine elementElement element,
    CoEnvData envData )
    throws WTException;
```

```java
protected CoPropertyModel newModel() throws WTException
```

```java
protected boolean canHandleInput( CoProperty property, WebEnvData
    web_envData )
    throws WTException
```

```java
public void updateInstance( CoProperty property,
    Object value,
    CoEnvData envData )
    throws WTException
```

In DCA Web, extend the newly create handler and implement the
com.ptc.core.ca.web.client.property.WebPropertyHandler interface. You
will need to write an updateProperty method.

```java
public void updateProperty( CoProperty property,
    WebEnvData web_envData )
    throws WTException
```

**Implement Property Model**

**When to implement:**

If the model (com.ptc.core.ca.co.client.property.CoSimplePropertyModel) used
by a supported renderer and handler pair does not provide enough functionality
for your business situation, you need to implement your own property model.

**How to implement:**

To implement a specialized model, you can implement a class that implements
one of the following two interfaces:

- com.ptc.core.ca.co.client.property.CoPropertyModel
- com.ptc.core.ca.co.client.property.CoSimplePropertyModel

Or you can implement a class that extends:

- com.ptc.core.ca.co.client.property.CoAbstractSimplePropertyModel.
How to enable the implemented model:

To enable of the new implemented model return its instance in:

```java
protected CoPropertyModel newModel() throws WTException
```

### Modifying Property Configuration to Use New Renderer or Handler

To change the property configuration to change the renderer, find the property to change in the DCA configuration. Then change the Property element's Renderer element resource attribute to equal the new renderer’s web application resource.

To change the property configuration to change the handler, find the property to change in the DCA registry. Then change the Property element's PropertyHandler element class attribute to equal the new handler full class name.

Sample (strings are broken for readability):

```xml
<Property id="myprop">
    <Label id="My Prop:"/>
    <Need attribute="myattribute"/>
    <PropertyHandler class="com.ptc.core.ca.web.client.property.
    myprop.WebMyPropPropertyHandler"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property
    /myprop/myprop.jsp"/>
</Property>
```

### Writing JSP Renderers

#### Guidelines

- The property value should be encapsulated in the model.
- To prevent excess white space in the generated html add an empty jsp code section at the end a jsp line. Add
  ```jsp
  <%//
  ```
  after the ending and begin the next jsp code section with
  ```jsp
  %>
  ```
  See the template for concrete examples.
- Ensure that `<SELECT>` and `<INPUT>` are closed off with `</SELECT>` and `</INPUT>`.
- Keep jsp code set small. Encapsulate detailed implementation in handler.
- Needs to support the four modes: create, view, edit, search
- Should catch exceptions and NOT re-throw.
try {
    CoProperty property = (CoProperty)element;
    CoElementMode mode = property.getMode();
    boolean editable = !mode.equals(CoElementMode.VIEW);
    CoSimplePropertyModel model = (CoSimplePropertyModel)property.getModel();

    if (!editable) {
        // Generate read only html
        //
    } else {
        // Based on mode generate html
        // Set need_input_end_tag or need_select_end_tag to true right after
        // the <INPUT> or <SELECT> tag. Then Set need_input_end_tag or
        // need_select_end_tag to false right after the </INPUT> or </SELECT> tag
        //
    }
} catch (Throwable t) {
    envData.getErrorInfo().setThrowable(t);
This chapter discusses action elements and handlers, and how they are used in DCA.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Elements</td>
<td>6-2</td>
</tr>
<tr>
<td>Action Handlers</td>
<td>6-2</td>
</tr>
<tr>
<td>&quot;Simple&quot; Approach for Action Handlers</td>
<td>6-4</td>
</tr>
<tr>
<td>Action Doers vs. Client Doers</td>
<td>6-5</td>
</tr>
<tr>
<td>Action Handler Types</td>
<td>6-6</td>
</tr>
<tr>
<td>Action List</td>
<td>6-9</td>
</tr>
</tbody>
</table>
Action Elements

In DCA, action elements are used in configuration specifications when there is a need to provide means for the user to trigger an operation, open a new window or launch a wizard or a picker. An action can be presented in many different ways and its visibility and accessibility can be controlled programmatically.

Here is a typical configuration of an action.

[formatted for readability]

```xml
<Action id="action.checkout.task" needSelection="true">
  <Image resource="/wtcore/images/checkout.gif"/>
  <Label resource="Checkout"/>
  <ToolTip resource="Check Out"/>
  <ActionHandler class="com.ptc.core.ca.web.client.
                   action.WebSimpleInPlaceActionHandler"/>
  <ActionDoer class="%common:SimpleActionTaskDoer">
    <Need attribute="checkoutInfo.state" type="%[RevisionControlled]"/>
    <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-checkout"/>
    <Validator id="com.ptc.wip.checkout.validator"
               class="com.ptc.refimpl.co.
                   validators.SampleCheckoutValidator"/>
  </ActionDoer>
</Action>
```

Action elements delegate most of their functions to action handlers.

Action Handlers

The Action handler updates a model of an Action element. When an ActionDoer is specified, the handler invokes methods on the doer to set up the visibility and accessibility statuses of the Action element:

```java
boolean isRelevant(CoActionHandlerData data)
boolean canExecute(CoActionHandlerData data)
```

The Action handler is also responsible for the implementation of the method that gets executed when an in-place action is triggered.

```java
void performOperation( CoAction action, CoEnvData envData )
```

DCA provides a variety of action handlers including those capable of triggering the execution of Info*Engine tasks. For that, handlers may require configured action or client doers. For more details on doers, refer to “Data Acquisition and Manipulation - Doers” on page 7-1.
The basic types of handlers are used in situations where the action to perform does not require business objects or the action holder. Action handlers of the basic type extend com.ptc.core.ca.co.client.action.CoActionHandler class. The handler creates an instance of com.ptc.core.ca.co.client.action.CoActionHandlerData to pass it to a doer.

Note: Handlers of the basic type know only about an action that was triggered and about environment.
"Simple" Approach for Action Handlers

The more robust "simple" handlers are used in situations where the operation to perform requires access to the business object or objects and action holder. The simple action handlers all identified by "Simple" in their class names. An example is com.ptc.core.ca.web.client.action.WebSimpleInPlaceActionHandler.

A simple action handler passes an instance of com.ptc.core.ca.co.client.simple.CoSimpleActionHandlerData object to the doer's methods. This means that the doers configured for "simple" actions are required to accept this type of handler data. All doers having "simple" in their names and a variety of wizard, tree, and picker doers meet this requirement.
The following table presents the meaning of the attributes in the CoSimpleActionHandlerClass.

<table>
<thead>
<tr>
<th>Action location</th>
<th>actionHolder</th>
<th>instanceHolders</th>
<th>actionHolderRelated Instance</th>
<th>actionHolderRelated type and UFID</th>
</tr>
</thead>
<tbody>
<tr>
<td>in a row</td>
<td>row</td>
<td>row</td>
<td>instance holder ancestor of a table</td>
<td>type and UFID of action holder related instance or a frame</td>
</tr>
<tr>
<td>in a node</td>
<td>node</td>
<td>node</td>
<td>Not available</td>
<td></td>
</tr>
<tr>
<td>in a property panel</td>
<td>property panel</td>
<td>property panel</td>
<td>instance holder ancestor of property panel's parent</td>
<td>type and UFID of action holder related instance or a frame</td>
</tr>
<tr>
<td>in toolbar of a table</td>
<td>table</td>
<td>selected rows</td>
<td>instance holder ancestor of table's parent</td>
<td>type and UFID of action holder related instance or a frame</td>
</tr>
<tr>
<td>in a toolbar of a tree</td>
<td>tree</td>
<td>selected nodes</td>
<td>instance holder ancestor of tree's parent</td>
<td>type and UFID of action holder related instance or a frame</td>
</tr>
<tr>
<td>in a wizard</td>
<td>wizard</td>
<td>wizard</td>
<td>instance holder in a launching frame</td>
<td>instance holder in a launching frame</td>
</tr>
<tr>
<td>in a frame</td>
<td>frame</td>
<td>Not available</td>
<td>Not available</td>
<td>type and UFID of a frame</td>
</tr>
</tbody>
</table>

**Action Doers vs. Client Doers**

Action doers perform Info*Engine tasks while client doers do not.

Action handlers that require doers obtain the class name of the doer from the class attribute of the ActionDoer or ClientDoer element specified for the Action element.
### Action Handler Types

DCA package com.ptc.core.ca.co.client.action.

DCA Web package com.ptc.core.ca.web.client.action.

#### In Place

"In Place" action handlers do not change location of the current frame and do not create new frames. An example of such an action would be "Delete" action in a toolbar of a table. Use "In Place" action handlers for tasks that require no additional input from the user other then the selection of the action item (button, menu item, tool bar item).

"In Place" action handlers call the doer to determine action visibility, accessibility, and to perform an operation.

DCA provides the following "In Place" action handlers:

- com.ptc.core.ca.co.client.action.CoSimpleInPlaceActionHandler
- com.ptc.core.ca.co.client.action.CoInPlaceActionHandler

They execute the doer and then refresh the current DCA frame in the current window.

DCA Web extends DCA action handlers and provides the following handlers to be used in configuration specifications:

- com.ptc.core.ca.web.client.action.WebSimpleInPlaceActionHandler
- com.ptc.core.ca.web.client.action.WebInPlaceActionHandler

The handler executes the doer. The browser window gets refreshed automatically due to the use of a POST request in a browser that causes a redraw of DCA frame element in the current window.

#### Go To

"Go To" action handlers help with changing the location of a current DCA frame. These handlers look for the Location configuration element. Use "Go To" action handler to navigate between DCA frames in the same window.

While a doer can be specified, it can only be used to determine the visibility and accessibility of an action; doer’s "execute" method will not be called.

DCA provides the following "Go To" action handlers:

- com.ptc.core.ca.co.client.action.CoSimpleGoToActionHandler
- com.ptc.core.ca.co.client.action.CoGoToActionHandler

Both handlers take into account appId, roleId, and decoration dimensions of a location element that can be specified explicitly, or inherited from ancestor elements.

<Location ref="goto.myApp.location" appId="myApplication"/>
CoSimpleGoToActionHandler attempts to locate an instance holder using "simple" approach to get the type and UFID of its Info*Engine element. When not found, a type and UFID of the frame will be used.

DCA Web extends DCA action handlers and provides the following handlers to be used in configuration specifications:

com.ptc.core.ca.web.client.action.WebSimpleGoToActionHandler
com.ptc.core.ca.web.client.action.WebGoToActionHandler

When activated, a new page will be loaded in a browser window presenting a DCA frame element for a specified location.

**New Frame**

"New Frame" action handlers help with the creation of a new window and loading DCA frame with the appropriate location in it. These handlers look for the Location configuration element. Use "New Frame" action handler to create new permanent windows.

While a doer can be specified, it can only be used to determine the visibility and accessibility of an action; doer’s "execute" method will not be called.

DCA provides the following "New Frame" action handlers:

com.ptc.core.ca.co.client.action.CoSimpleNewFrameActionHandler
com.ptc.core.ca.co.client.action.CoNewFrameActionHandler

Both handlers take into account appId, roleId, and decoration dimensions of a location element that can be specified explicitly, or inherited from ancestor elements.

<Location ref="newFrame.myApp.location" appId="myApplication"/>

Both handlers take into account width and height attributes of a frame referenced by ref attribute.

CoSimpleNewFrameActionHandler attempts to locate an instance holder using "simple" approach to get the type and UFID of its Info*Engine element. When not found, a type and UFID of the frame will be used.

DCA Web extends DCA action handlers and provides the following handlers to be used in configuration specifications:

com.ptc.core.ca.web.client.action.WebNewFrameActionHandler
com.ptc.core.ca.web.client.action.WebNewFrameActionHandler

When activated, a new browser window will be created and a new page will be loaded in it window presenting a DCA frame element for a specified location.
"New Temporary Frame" action handlers help with the creation of a new temporary window and loading DCA frame with the appropriate location in it. These handlers look for the Location configuration element. Use "New Temporary Frame" action handler to create dialogs, wizards, and pickers.

While a doer can be specified, it can only be used to determine the visibility and accessibility of an action; doer's "execute" method will not be called.

DCA provides the following "New Temporary Frame" action handlers:

- `com.ptc.core.ca.co.client.action.CoNewTemporaryFrameActionHandler`

Both handlers take into account appId, roleld, and decoration dimensions of a location element that can be specified explicitly, or inherited from ancestor elements.

```xml
<Location ref="newTmpFrame.myApp.location" appId="myApplication"/>
```

Both handlers take into account width and height attributes of a frame referenced by ref attribute.

`CoSimpleNewFrameActionHandler` attempts to locate an instance holder using "simple" approach to get the type and UFID of its Info*Engine element. When not found, a type and UFID of the frame will be used.

When `CoSimpleNewFrameActionHandler` is used, at any time during the drawing of a temporary frame the following code can find an action from the launching frame.

```java
CoFrame frame = element.getFrame();
String oaa = frame.getOpenerActionAddress();
CoElement opener_action =
    CoElementUtils.giveElement(oaa, envData);
```

DCA Web extends DCA action handlers and provides the following handlers to be used in configuration specifications:

- `com.ptc.core.ca.web.client.action.WebNewTemporaryFrameActionHandler`

When activated, a new browser window will be created and a new page will be loaded in it window presenting a DCA frame element for a specified location.
**Action List**

**Active Action List**

Active action list will trigger an operation when an action is selected from a list of actions.

The following configures the active action list:

```xml
<ActionList id="folder.browser.actions">
  <Insert ref="common:action.cut"/>
  <Insert ref="common:action.copy"/>
  <Insert ref="action.paste"/>
  <Insert ref="common:action.delete.task"/>
  <Insert ref="action.create.task"/>
  <Insert ref="action.open"/>
  <Insert ref="action.open_in_new_frame"/>
  <Insert ref="action.view.task"/>
  <Insert ref="action.edit.task"/>
  <Insert ref="common:renderer.actionDropDownList"/>
  <Prompt resource="- Pick an Action - 
  &nbsp;&nbsp;"/>
</ActionList>
```

**Passive Action List**

Passive action list will trigger an operation when an action is selected and trigger is activated.

The following configures the passive action list:

```xml
<ActionList id="folder.browser.actions">
  <Insert ref="common:action.cut"/>
  <Insert ref="common:action.copy"/>
  <Insert ref="action.paste"/>
  <Insert ref="common:action.delete.task"/>
  <Insert ref="action.create.task"/>
  <Insert ref="action.open"/>
  <Insert ref="action.open_in_new_frame"/>
  <Insert ref="action.view.task"/>
  <Insert ref="action.edit.task"/>
  <Insert ref="common:actionlist.trigger"/>
  <Insert ref="common:renderer.passiveActionDropDownList"/>
  <Prompt resource="- Pick an Action - &nbsp;&nbsp;"/>
</ActionList>
```
Data Acquisition and Manipulation - Doers

This chapter discusses the concepts behind integrating server-side functionality with DCA clients.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useful Glossary Terms</td>
<td>7-2</td>
</tr>
<tr>
<td>Doers</td>
<td>7-3</td>
</tr>
<tr>
<td>Configuration</td>
<td>7-5</td>
</tr>
<tr>
<td>Doer Implementation</td>
<td>7-7</td>
</tr>
<tr>
<td>Outer Task</td>
<td>7-26</td>
</tr>
<tr>
<td>Progress Monitoring</td>
<td>7-26</td>
</tr>
</tbody>
</table>
Useful Glossary Terms

**Info*Engine element**

A `com.infoengine.object.factory.Element` class that constitutes a unit of Information.

**Doer**

A class that does work on behalf of the framework. A doer could be pure client-side functionality, in which case it is a ClientDoer, or it could perform an operation on a remote repository. If the Doer is doing work to get the model for a DCA element it is called a Model Doer, if it is performing an action on one or more Info*Engine elements it is called an Action Doer.

In DCA, certain elements can have models. Those elements implement the CoModelDriven interface. Their models implement the CoModel interface. A model-driven element can reference one or more Info*Engine elements by implementing either the CoInstanceHolder or CoInstanceListHolder interface.
Doers

During element creation/update of such an elements, doer is called that always implements CoModelDoer interface. Triggering an action results in the execution of an operation. The operation is implemented via a class implementing CoActionDoer.

An action doer can be either client-side or server-side. A client-side doer does not invoke in Info*Engine task while the server-side does. An example of a client-side doer would be an implementation of a Copy operation:

```java
public void execute( CoSimpleActionHandlerData execData )
throws WTException {
    CoClipboard clipboard = execData.getEnvData().getSessionData().getClipboard();
    clipboard.setReason(CoClippingReason.COPY);
    super.execute(execData);
}
```

CoClipboardCopyDoer does not perform any client-server communications, it simply remembers a set of selected objects in the clipboard. Client doers are outside of the scope of this section.

Server doers come in two general types, an Action Doer that performs a server-side operation on one or more Info*Engine elements objects, and a Model Doer that retrieves the data for an element.

The primary goal of this framework is to allow developers with a server-side focus to retain their focus. In a nutshell all a doer does is invoke their server-side functionality and post results to the client framework. The client developers decide what to do with this data. And since we cannot anticipate every possible scenario so numerous customization opportunities have been made available.
DCA uses a factory to find an appropriate doer. It then creates and initializes an instance of an appropriate handler data object and uses it to invoke execute method on a doer. This execute method will run an Info*Engine task.

**Model Doer**

During the process of drawing an element, a DCA element needs to update its model. A typical updateModel method looks like the following:

```java
public void updateModel( CoEnvData envData )
    throws WTException {
    boolean is_executing_doer = CoElementUtils.isExecutingDoer(this);
    CoElementUtils.updateModel(this, envData);

    CoTableModel table_model = (CoTableModel)model;
    if (table_model.isExpanded() &&
        (is_executing_doer ||
         !table_model.isWasExpanded())) {
        CoModelDoer doer = (CoModelDoer)envData.getSessionData().
            getDoerFactory().getModelDoer(this, envData);

        if (doer == null) {
            CoConfigReportUtil.reportError((CoElement)this,
                "A doer is required.", envData);
        }

        CoSimpleModelHandlerData data =
            new CoDefaultSimpleModelHandlerData();
        data.init(this, envData);

        LogHelper.traceDoer (doer, data, envData);
        doer.execute(data);
    }
}
```

Lines in bold emphasize the important topics of this section.

**Cleanable Elements**

Some of the DCA elements, like instance holders, table, tree, and frame, implement the com.ptc.core.ca.co.client.element.CoCleanable interface.

```java
<<Interface>>
CoCleanable

dirty : boolean = true
```

As such, the elements can be marked dirty. When the element is marked dirty and is in view mode, it executes its data acquisition operation while updating its model by invoking a model doer.

DCA elements are typically marked dirty by invoking the following method:

CoElementUtils.markDirty(CoElement element, CoEnvData envData)
The method checks to see if there are any ancestor elements in non-view mode before attempting to set dirty flag on the cleanable. When such an ancestor element is found, the flag will not be set. This, for instance, prevents DCA from marking dirty cleanable elements contained by a wizard in create, search, or edit mode.

**Action Doer**

When an "in-place" action is triggered, an ActionHandler is called. The body of a typical in-place action handler looks like the following:

```java
public void performOperation( CoAction action, CoEnvData envData )
    throws WTException {
    CoActionHandlerData data = newActionHandlerData();
    data.init(action, envData);
    CoActionDoer doer = (CoActionDoer)envData.getSessionData().getDoerFactory().getActionDoer(action, envData);
    doer.execute(data);
}
```

Lines in bold emphasize the important topics of this section.

**Configuration**

To hook up server-side functionality to a DCA client, it is necessary to associate the correct types of doers to the configuration element. This section discusses how to do that. When working out your Action/Model doers, much work can be saved by the use of synonyms.

**ModelDoer**

A model doer retrieves the model for a DCA element. The configuration is almost identical to an ActionDoer.

```xml
<SimpleTable id="lifecycle.all.table" selectorType="single" mode="view">
    <Type type="wt.lifecycle.LifeCycleTemplate"/>
    <SimpleColumn show=":name">
        <Label resource="Name"/>
    </SimpleColumn>
    <ModelDoer class="%common:SimpleModelTaskDoer">
        <Task nextOperation="STDOP|com.ptc.windchill.view"
            taskDelegate="dca-query-objects">
            <TaskParameter name="search_type" value="wt.lifecycle.LifeCycleTemplate"/>
            <TaskParameter name="where" value="name=**"/>
        </Task>
    </ModelDoer>
</SimpleTable>
```

**Note:** In the example above we have a couple of static task parameters.
**ActionDoer**

The ActionDoer can appear in two places: inside of an Action element, and inside of a Wizard element. The semantics are slightly different for the two scenarios. When it is in the context of an Action element, the operation is executed when this action is triggered. When it is in the Wizard element, the operation is executed when an Ok or Apply action of the wizard is triggered.

The following is a typical action configuration:

```xml
<Action id="action.undocheckout.task" needSelection="true">
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/uncheckout.gif"/>
  <Label resource="Undo Checkout"/>
  <ToolTip resource="Undo Checkout"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebSimpleInPlaceActionHandler" />
  <ActionDoer class="com.ptc.core.ca.co.client.doer.CoDefaultSimpleActionTaskDoer">
    <Need attribute="checkoutInfo.state" type="%[RevisionControlled]" />
    <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-undo-checkout" />
    <Validator class="com.ptc.refimpl.co.validators.SampleUndoCheckoutValidator" />
  </ActionDoer>
</Action>
```

The action doer requires an attribute `checkoutInfo.state` to perform an operation. In this particular example the attribute will be needed by the validator.

You must specify a `taskDelegate`. Failure to do so will cause an exception to be thrown. The value here corresponds to the name of an Info*Engine task.

Validator class simply evaluates the environment information to determine whether an action should be made available for execution. The class attribute must refer to a class implementing

```
com.ptc.core.ca.co.client.doer.CoActionValidator
```

It has two methods: one to determine whether the action can even be considered, and a second that says whether it is allowed to be executed.
Doer Implementation

Task Doers

The job of a Task doer is to:

- take a composite combination of
  - static information from configuration system
  - dynamic information from DCA elements and Info*Engine elements they refer to
- prepare Info*Engine task parameters
- invoke Info*Engine task
- populate models of DCA elements with Info*Engine elements for model doers
- populate models of DCA elements with Info*Engine elements
- invalidate cache for action doers

The reader is assumed to be familiar with the basics of Info*Engine Task Delegate and Webject concepts. The following diagram presents the base class for all task doers - CoAbstractTaskDoer:
Calls other methods to prepare task data, name, parameters, feedback, execute task, process result, etc.

```java
prepareTaskParameters(taskData : CoTaskData)
```

Prepares task parameters by preparing default task parameters and configurable task parameters.

```java
addDefaultTaskParameters(taskData : CoTaskData)
```

Abstract method to delegate the preparation of the default task parameters to sub classes. Default task parameters have `dca_` prefix in their names.

```java
addConfigurableTaskParameters(taskData : CoTaskData)
```

Prepares task parameters based on the configuration specification for the doer including static parameters, computed values (deprecated), and functions.

```java
executeTask (execData : CoHandlerData)
```

Execute task via federation helper.

```java
processTaskResults(taskData : CoTaskData)
```

Process output task parameters by delegating the processing of default and configurable parameters.

```java
processDefaultTaskResults(taskData : CoTaskData)
```

Abstract method to delegate processing of default output task parameters to subclasses.

```java
processConfigurableTaskResults(taskData : CoTaskData)
```

Process configurable output task parameters via functions.

### Task Parameters

Task parameters can be specified via the `TaskParameter` element with a single explicit value or function.

```xml
<TaskParameter name="color" value="green"/>
```

### Functions

A Function is a Java class implementing code that is executed by a doer before task invocation to prepare input parameters and after task invocation to process output task parameters.

The underlying mode of operation for all Task-based doers is to prepare the information into an argument array for `FederationHelper.doAction` which takes an `Object[][]` as its arguments. The `Object[][]` consists of a series of name value pairs. This form of customization involves the framework feeding all of the data to the customized computed value function. This is actually quite powerful in an Info*Engine task context, since any argument that can be fed into Info*Engine can be appended to the default attributes.
Function specification in configuration file generally looks like:

```xml
<TaskParameter function="function-class-name"/>
<TaskOutParameter function="function-class-name"/>
```

where `function-class-name` is a fully qualified class name.

Functions can be configurable via additional parameters from attributes and sub-elements of its task parameter. Example:

```xml
<TaskParameter function="CoSimpleFunction_Action_Param_ObjectsAsGroup"
    element="table.pick.list.part.target" name="GROUP_USES"/>
```

Here the function receives two additional attributes to parameterize its behavior: "element" and "name".

Functions must implement one of four interfaces from `com.ptc.core.ca.co.client.doer.function` package:

- CoFunction_Action_Param
- CoFunction_Action_Result
- CoFunction_Model_Param
- CoFunction_Model_Result

Every function takes `taskData` parameter of class `CoTaskData` created by task doer:

```java
class CoTaskData {
    String taskName;
    java.util.ArrayList taskParameters;
    com.ptc.core.util.feedback.common.FeedbackSpec taskFeedBack;
    com.infoengine.object.factory.Group taskResult;
    CoHandlerData execData;
    CoConfigFeatureType doerType;

    CoTaskData(CoHandlerData execData, CoConfigFeatureType doerType)
    getEnvData()
    getTaskParameters()
    getRawTaskParameters()
    addTaskParameter(name, value)
    toString()
}
```

The function also takes `execData` parameter of class `CoActionHandlerData` or `CoModelHandlerData` created by the `updateModel` method of the element or potentially by model handler. The meaning of `CoActionHandlerData` is described in the action handler section.
Basic model handler data only knows about an environment and an element it executes a doer for. The more robust "simple" approach employed by DCA elements provides more information:
<table>
<thead>
<tr>
<th>element</th>
<th>Typeld &amp; UFID from element</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>instance holder ancestor or frame</td>
</tr>
<tr>
<td>tree</td>
<td>instance holder ancestor or frame</td>
</tr>
<tr>
<td>property panel</td>
<td>instance holder ancestor or frame</td>
</tr>
<tr>
<td>wizard</td>
<td>frame</td>
</tr>
</tbody>
</table>

**CoFunction**

CoFunction**_Action_Param_**  
```
public void addInputParameters (CoTaskData taskData,
                               CoConfigFeature parameterConfig,
                               CoActionHandlerData execData)

    throws WTException;
```

CoFunction**_Action_Result_**  
```
public void processResult (CoTaskData taskData,
                           CoConfigFeature parameterConfig,
                           CoActionHandlerData execData)

    throws WTException;
```

CoFunction**_Model_Param_**  
```
public void addInputParameters (CoTaskData taskData,
                                CoConfigFeature parameterConfig,
                                CoModelHandlerData execData)

    throws WTException;
```

CoFunction**_Model_Result_**  
```
public void processResult (CoTaskData taskData,
                          CoConfigFeature parameterConfig,
                          CoModelHandlerData execData)

    throws WTException;
```

**Functions - Implementation**

For the purpose of this section, we will call xml attributes used in a declaration of a function-based task parameter as **parameters of the function**.

```xml
<TaskParameter function="func"
param1="value1"
...  
paramN="valueN"/>
```

The following is a list of functions DCA implements out of the box in the package com.ptc.core.ca.co.cleint.doer.function.impl.
CoFunction_Model_Result_CreateNPA

The function creates non-persistent attribute (NPA) on the elements in the task output group. The function can have the following parameters:

name = text [CS]

The name of the attribute (NPA) to create. Required.

class = text [CS]

The type of the value of the created attribute. Must be a fully qualified class name (C). NPA’s value is expected to be of that class (C). Not required.

value = text [CS]

The value to be used as a value of a newly created attribute. Not required.

The function uses `valueOf(value)` method on the class C when found. The function uses C(value) constructor when found. Otherwise the function will fail with an exception.

useAsTaskParameter = (true | false)

Indicates whether the attribute should be included in the list of task parameter by doers utilizing "simple" approach to the creation of task parameters. Not required. Default value is true.

Example:

```xml
<TaskOutParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoFunction_Model_Result_CreateNpa"
                 name="sampleBooleanAttribute"
                 class="java.lang.Boolean"
                 value="true"
                 useAsTaskParameter="false"/>
```

CoFunction_Action_Result_CreateNPA

The function creates non-persistent attribute (NPA) on the elements in the task output group. The function can be configured exactly the same as CoFunction_Model_Result_CreateNPA.

Simple Functions

To take advantage of a so-called 'simple' approach, functions can extends one of following four abstract classes from com.ptc.core.ca.co.client.doer.function package:

- CoSimpleFunction_Action_Param
- CoSimpleFunction_Action_Result
- CoSimpleFunction_Model_Param
- CoSimpleFunction_Model_Result

CoSimpleFunction_Action_Param

```java
public void addInputParameters (CoTaskData taskData,
                                CoConfigFeature parameterConfig,
                                CoSimpleActionHandlerData execData)
    throws WTException;
```
CoSimpleFunction_Action_Result
    public void processResult (CoTaskData taskData,
                                CoConfigFeature parameterConfig,
                                CoSimpleActionHandlerData execData)
    throws WTException;

CoSimpleFunction_Model_Param
    public void addInputParameters (CoTaskData taskData,
                                     CoConfigFeature parameterConfig,
                                     CoSimpleModelHandlerData execData)
    throws WTException;

CoSimpleFunction_Model_Result
    public void processResult (CoTaskData taskData,
                                CoConfigFeature parameterConfig,
                                CoSimpleModelHandlerData execData)
    throws WTException;

Simple Functions - Implementation

    The following is a list of "simple" functions DCA implements out of the box in
    the package com.ptc.core.ca.co.cleint.doer.function.impl.

CoSimpleFunction_Model_Param_AttrSpecFromConfig
    The function adds the following task parameters:

        dca_attribute
            Contains names of the attributes based on the Need statements used by
            configuration specification.

        dca_includes_args
            Parameter value is always equal to true

        dca_includes_constraints
            Parameter value is always equal to true

        dca_includes_descriptions
            Parameter value is always equal to true

CoSimpleFunction_Model_Param_MainObjectIdentity
    The function adds task parameters describing "main" object's identity:

        dca_object_type
            Type of the I*E element.

        dca_object_ref
            UFID of the I*E element.

        element
            I*E element.
**CoSimpleFunction_Model_Result_Default**

The function does typical processing of a task result for a simple model doer.

**CoSimpleFunction_Action_Param_ObjectsInfo**

The function adds parameters describing objects that are target of this action. For every object the following parameters are added:

- **dca_attribute**
  One or more string parameters with values containing logical names of attributes to construct a filter.

- **dca_field**
  One or more string parameters with values that look like `<name>=<value>`, where name is the name of an attribute and value is its value.

- **dca_object_type**
  Type of the I*E element.

- **dca_object_ref**
  UFID of the I*E element.

- **element**
  I*E element.

**CoSimpleFunction_Action_Result_RefreshAll**

Marks appropriate DCA elements as dirty. This function usually should be invoked after tasks updates anything in a database. The function employs the following algorithm:

1. iterates through a set of all instance holders in session notifying them about the potential change in their instances. Instance holders, in turn, attempt to mark themselves as dirty by invoking

   ```java
   CoElementUtils.markDirty(CoElement element, CoEnvData envData)
   ```

   See the explanation of this method above.

   Some of the instance holders like row and tree node attempt to mark their table or a tree respectively using the same method.

2. when invoked in a temporary frame, attempts to find an opener action in the launching frame. When found, attempts to mark a cleanable ancestor element as dirty by invoking

   ```java
   CoElementUtils.markDirty(CoElement element, CoEnvData envData)
   ```
**CoSimpleFunction_Action_Param_ObjectsAsUfids**

Prepares task parameters containing UFIDs of Info*Engine elements. Typically used to specify additional task parameters like a list of used WTParts for Create Part action.

The function does the following:

1. locates DCA element containing Info*Engine elements using parameter `element`. The element path must be unique in a frame definition. When no `element` parameter is specified, an action holder is used.
2. locates Info*Engine elements in a found DCA element
3. for every Info*Engine element adds task parameter with name specified in configuration via `name` parameter and value containing UFID of the element.

The function can use the following parameters:

- **name** = `text[CS]`
  Task parameters will have this name.

- **element** = `text[CS]`
  Contains path of the DCA element. The element will be located and used to get Info*Engine elements from.

- **selected** = `(true | false)`
  Indicates whether to take Info*Engine elements from either selected DCA elements or all DCA elements when an element identified by path is a selector (table or tree). The default value is `false`.

- **emptyGroupAllowed** = `(true | false)`
  Indicates whether the function can create an empty group when elements cannot be found. The default value is `false`.

**CoSimpleFunction_Action_Param_ObjectsAsGroup**

Prepares a task parameter containing an Info*Engine Group. Typically used to specify additional task parameters like a list of used WTParts for Create Part action.

The function does the following:

1. locates DCA element containing Info*Engine elements using parameter `element`. The element path must be unique in a frame definition. When no `element` parameter is specified, an action holder is used.
2. locates Info*Engine elements in a found DCA element
3. creates an Info*Engine Group consisting of Info*Engine elements
4. adds task parameter with name specified in configuration via `name` attribute using newly created group as a value
The function can use the following parameters:

```
name = text
```
Contains name of the task parameter to create.

```
element = text
```
Contains path of the DCA element. The element will be located and used to get Info*Engine elements from.

```
selected = (true | false)
```
Indicates whether to take Info*Engine elements from selected DCA elements or all DCA elements when an element is a selector (table or tree).

```
emptyGroupAllowed = (true | false)
```
Indicates whether the function can create an empty group when elements cannot be found. The default value is false.

Adding a Group as a Task Parameter

A group can be added to the list of task parameters either in a function (recommended) or by extending a doer. To add a group into the list of task parameters by extending a doer use the following method:

```java
Group my_group = ...
CoDoerHelper.addGroupIntoTaskParameters (my_group,"my_group_name", taskData);
```

There is no need to create an instance of CoDoerHelper class because all its methods are static.

In the task you can get the group as follows

```java
Group my_group = getGroup ("my_group_name");
```

Caution: If you don’t use the above mentioned method to specify a group as a task parameter, be aware of the following feature of I*E, which is not quite obvious. The Group task parameter must be specified not in the form you probably would expect:

```
("my_group_name", my_group)
```

but in the form:

```
("$GROUP", my_group)
```

The information about the name of the group (i.e. "my_group_name") is specified in the name attribute of the group and can be accessed from within a task as follows

```java
Group my_group = getGroup ("my_group_name");
String my_group_name = my_group.getName());
```
Removing Task Parameters

Sometimes there is a need to remove already generated task parameters. DCA provides two similar functions to be used with model and action task doers in com.ptc.core.ca.co.client.doer.function.impl package.

CoFunction_Action_Param_RemoveParameters
CoFunction_Model_Param_RemoveParameters

Both functions act the same way and use the following attributes:

- name = text
  The name of task parameter to be removed.
- value = text
  The value of task parameter to be removed.

The following removes a parameter with a name dca_attribute and a value uploadPath.

```xml
<TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoFunction_Action_Param_RemoveParameters"
               name="dca_attribute"
               value="uploadPath"/>
```

Regular expressions can be used as a value for both of the attributes. The developer can specify, for example,

```xml
<TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoFunction_Action_Param_RemoveParameters"
               name="dca*."
               value="upload*."/>
```

Both attributes are optional. Not specifying an attribute is the same as specifying its value as "*". When both attributes are not specified, the function will remove all previously generated task parameters.
Model Task Doer

The Doer hierarchy in `com.ptc.core.ca.co.cleint.doer` package for a ModelTask doer is shown below.
CoDefaultModelTaskDoer

CoDefaultModelTaskDoer does the following:
1. prepares static task parameters
2. invokes specified functions to prepare dynamic task parameters. (It also takes care of computed values [deprecated]).
3. executes the task
4. invokes specified functions to process task parameters

CoDefaultSimpleModelTaskDoer

CoDefaultSimpleModelTaskDoer does everything that CoDefaultModelTaskDoer does. Additionally it invokes the following functions:

CoSimpleFunction_Model_Param_AttrSpecFromConfig
CoSimpleFunction_Model_Param_MainObjectIdentity
CoSimpleFunction_Model_Result_Default

Debugging Task Doers

To debug DCA Info*Engine tasks modify your wt.properties as follows:
1. To log tasks execution in a method server add the standard Info*Engine property
   
   wt.adapter.verboseWebject=true

2. To log DCA Info*Engine task invocation from servlet engine add:

   com.ptc.core.ca.co.verbose.task=true

Note: The Log will be available in Windchill\logs\DCA.log

Action Task Doer

An Action Task Doer has a similar class hierarchy to a Model Task Doer.
CoDoer

<<Interface>>

CoAbstractTaskDoer

execute(execData : CoHandlerData)

initTaskData(execData : CoHandlerData) : CoTaskData

prepareTaskName(taskData : CoTaskData)

prepareTaskParameters(taskData : CoTaskData)

prepareTaskFeedback(taskData : CoTaskData)

executeTask(taskData : CoTaskData)

processTaskResult(taskData : CoTaskData)

<<Abstract>>

addDefaultTaskParameters(taskData : CoTaskData)

addConfigurableTaskParameters(taskData : CoTaskData)

processDefaultTaskResult(taskData : CoTaskData)

getDoerFeatureType() : CoConfigFeatureType

<<Interface>>

CoActionDoer

execute()

canExecute()

isRelevant()

CoDefaultActionTaskDoer

addDefaultTaskParameters()

processDefaultTaskResult()
CoDefaultActionTaskDoer

CoDefaultActionTaskDoer does the following:

1. prepares static task parameters
2. invokes specified functions to prepare dynamic task parameters. (It also takes care of computed values [deprecated]).
3. executes the task
4. invokes specified functions to process task parameters

CoDefaultSimpleActionTaskDoer

CoDefaultSimpleActionTaskDoer does everything that CoDefaultActionTaskDoer does. Additionally it invokes the following functions:

CoSimpleFunction_Action_Param_ObjectsInfo
CoSimpleFunction_Action_Result_RefreshAll

Action Validators

The framework decides whether an action should be enabled by calling methods on a doer:

boolean isRelevant(CoActionHandlerData data)
boolean canExecute(CoActionHandlerData data)

Developers are free to implement CoActionDoer interface or extend one of the doer classes and implement this methods.

Action task doer dispatches those calls to action validators when specified. By default the validator returns true. The doer will call two methods from this class, isRelevant and canExecute. The isRelevant method should be over-ridden to see, for instance, if there is a need to check the license status of the current principal before continuing. canExecute should be over-ridden if there are conditions that need to be satisfied in order for the action to be enabled. For example undo-checkout should be disabled for an object that is currently checked in.

<Action id="action.undocheckout.task" needSelection="true">
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/unchecout.gif"/>
  <Label resource="Undo Checkout"/>
  <ToolTip resource="Undo Checkout"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebSimpleInPlaceActionHandler"/>
  <ActionDoer class="%common:SimpleActionTaskDoer">
    <Need attribute="checkoutInfo.state" type="%[RevisionControlled]"/>
    <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-undo-checkout"/>
  </ActionDoer>
</Action>
The default implementation returns true. You may create your own subclass of this method that takes other parameters. The default implementation returns true.

The DCA framework comes with some sample validators. These are in $(wt.home)/src/com/ptc/refimpl/validators.

DCA provides a special action doer for the actions that do not require an execution of an operation when triggered. This are usually actions of go-to, new frame, and new temporary frame types. The implements `execute` method as empty and dispatches to the validator when needed.

```xml
<Action id="action.go_to">
  ...
  <ActionHandler class="com.ptc.core.ca.web.client.action. WebSimpleGoToActionHandler"/>
  <ActionDoer class="com.ptc.core.ca.co.client.doer. CoValidatingActionDoer">
    <Validator class="specify full class name here"/>
  </ActionDoer>
</Action>
```
Task Validator

Task Validators are used in the case when it is impractical to disable the action, but a task should not be executed under certain conditions. For instance, it may not be possible, or can be detrimental, to performance to validate each and every action when no more than a single action can be triggered.

All task doers that extend com.ptc.core.ca.co.doer.CoAbstractTaskDoer may use task validation functionality. Task validation works in the same way as action validation, no new configuration elements are introduced.

The task validators has to implement the interface with a single method:

\[
\text{com.ptc.core.ca.co.doer.CoTaskValidator}
\]

When a method returns false, the task will not be executed as well as the result processing methods. Otherwise, the task will be executed as usual. The following is an example of a configuration fragment:

\[
\text{<Task>}
\quad \text{<Validator class="task-validator-class">}
\quad \ldots
\quad \text{/Validator>}
\quad \ldots
\quad \text{/Task>}
\]

There is an example of task validator usage is in the Edit Properties dialog for a document. The dialog allows a user to change some attributes on the selected document, and upload primary content. Changing attributes is performed by an Info*Engine task (dca-update-single-object). The content upload is performed using an applet. Both operations are performed inside the same action (OK or Apply). If a user needs to upload a file only, there is no need to execute the task. In that case, a task validator can be used to check if there are any attribute changes:

\[
\text{<ActionDoer class="%[common:SimpleActionTaskDoer]">}
\quad \text{<Task nextOperation="STDOP|com.ptc.windchill.view"
\quad \quad \quad \text{taskDelegate="dca-update-single-object">}
\quad \text{<Validator class="com.ptc.core.ca.co.client.doer.
\quad \quad \quad \text{CoTaskValidatorHasParameter"}>
\quad \quad \text{<CheckTaskParam name="dca_field"/>}
\quad \quad \text{/Validator>}
\quad \text{/Task>}
\quad \text{/ActionDoer>}
\]

The CoTaskValidatorHasParameter contains a list of strings and checks if they are provided as task parameters. If some of them are missing, the task will not be
executed. In this particular case, if there is no task parameter field, the deupdate-single-object task will not be executed since no attributes are changed.

Another example of task validation that can be used is Checkout/Open action. If an object is already checked out, the application should open it without checkout task execution.

**Using the computedValues [deprecated]**

The simplest way to customize a Task-based doer is to customize the computedValue attribute of the TaskParameter tag.

**Computed Values Class Diagram**

The class diagram involved in the ComputedValues customization approach is shown below

![Class Diagram](image)

If a computedValue taskParameter attribute is in the context of an ActionDoer tag, the class that is specified should implement the CoActionTaskComputedValue interface. If the computedValue attribute is in the context of a ModelDoer tag, then the class specified in the class argument should implement the CoModelTaskComputedValue interface.

**ComputedValue Interface contract**

The contract for the computed values is that the customizer is expected to return an ArrayList of objects corresponding to the parameter argument

**CoActionTaskComputedValue**

Below is the computeValues interface.

```java
public List computeValues( CoConfigFeature taskparameter,  
Element ieElement,  
CcElement elem,  
CoConfigFeature actionDoerConfig,  
CoSimpleActionHandlerData execData )  
throws WTException;
```
The significance of each of the arguments is as follows:

**taskParameter**
- is the configuration element describing TaskParameter.

**isElement**
- whether the Info*Engine element that the action is going to be performed on.

**elem**
- DCA element that is an action holder e.g. a row in a table or a node in a tree.

**actionDoerConfig**
- the configuration element describing the ActionDoer configuration.
  CoConfigFeature provides a getParent method that allows you to navigate up in your UI hierarchy which allows you to pull all static information available in the context of the call.

**execData**
- provides data for a simple approach and all environment data.

The framework will call the specified implementation and expect you to return null, or a sub-class of List. For example, suppose you wish to populate an Info*Engine task parameter called "foo". You run your computation and determine that the parameter values for foo should be "bar"; You would populate the list with "bar". The framework already knows you want to supply values for the "foo" parameter.

**CoModelTaskComputedValue**

Below is the prototype for the computeValues method for a CoModelTaskComputedValue object.

```java
public List computeValues( CoConfigFeature taskParameter,
                        CoConfigFeature modelDoerConfig,
                        CoSimpleModelHandlerData execData )
    throws WTException;
```

The significance of each of these arguments is as follows

**taskParameter**
- is the configuration element describing TaskParameter.

**modelDoerConfig**
- the configuration element describing the ModelDoer configuration.
  CoConfigFeature provides a getParent method that allows you to navigate up in your UI hierarchy which allows you to pull all static information available in the context of the call.

**execData**
- provides data for a simple approach and all environment data.

The framework will call the specified implementation and expect you to return null, or a sub-class of List. For example, suppose you wish to populate an
Info*Engine task parameter called "foo". You run your computation and determine that the parameter values for foo should be {"bar", "barfic", "barferific"}; You would populate the list with "bar", "barfic", and "barferific". The framework already knows you want to supply values for the "foo" parameter.

**Outer Task**

A task specified for a doer (both ModelDoer and ActionDoer) can be executed by DCA from within another so-called *outer task*. You may decide to specify outer task to achieve a transactional behavior or to externalized commonly used operations like setting up task context.

To specify that a task needs to be executed from within outer task OuterTask element must be used. This outer task gets all the parameters of the regular task. It will get also an additional parameter with name dca_target_task. Its value is the name of the task specified in Task element by taskDelegate attribute.

Example:

```xml
<ActionDoer class="%common:SimpleActionTaskDoer">
  <Task taskDelegate="dca-create-part-action"
        nextOperation="STDOP\com.ptc.windchill.view">
    <OuterTask taskDelegate="dca-transaction"/>
    <TaskParameter function="com.ptc.core.ca.co.client.doer.
                       function.impl.CoSimpleFunction_Action_Param_ObjectsAsUfids"
                    element="table.pick.list.part.target" name="uses"
                    selected="false"/>
    <TaskParameter name="target_type" value="WCTYPE\wt.part.WTPart"/>
    <TaskParameter name="link_type" value="WCTYPE\wt.part.WTPartUsageLink"/>
  </Task>
</ActionDoer>
```

This example demonstrates how to configure a task to be executed from within other task. The task dca-transaction provides transactional behavior for the included task.

**Progress Monitoring**

DCA provides a mechanism for monitoring the progress of long running tasks. The typical scenario is a retrieval of a large set of Info*Engine elements by a search result table. The mechanism, however, is general and can be employed to support other use cases.

The current implementation is based on the wt.util.WTThread capability to notify registered listeners about changes in thread’s properties. DCA employs four such properties: state, status, progressCount, and progressPercent. A task that runs inside from the WTThread is responsible for the update of the properties.
In order for the progress monitoring to take place, a task to be monitored has to be configured to run in WTThread. An action that triggers the task has to be configured to launch a temporary frame to present thread properties. Lastly, the frame must be refreshed on a regular basis to present the latest thread properties.

DCA demo present an example of configuring process monitoring using search functionality for the WTPart objects.

**Configuring a Task to Enable Progress Monitoring**

To configure a task to enable progress monitoring, add the attribute progressMonitor with a true value to the Task configuration element.

```xml
[com/ptc/part/part_search.xml]
<ModelDoer class="%common:SimpleModelTaskDoer">
  <Task id="search.table.doer.model.task"
    nextOperation="STDOP|com.ptc.windchill.view"
    taskDelegate="dca-query-objects-with-sort-and-paging"
    progressMonitor="%[context#progressMonitor]">
    <TaskParameter name="search_type" value="%[WTPart]"/>
    ... 
  </Task>
</ModelDoer>
```

**Caution:** A frame can contain no more than one instance of a task definition with the progress monitor enabled.

**Configuring an Action to Trigger an Execution of a Task**

A specially configured in-place action must be used to trigger an execution of a task to be monitored. It must use a special action handler that extends in-place action handler but is capable of the initialization of the monitoring process and launching a temporary frame containing status information.

```xml
[com/ptc/part/part_search.xml]
<Insert ref="search:search.action.alt"
  context.tableId="search.table.part"/>
```

In the example above, the action uses the regular search action

```xml
<Action id="search.action.alt">
  <Label resource="Search"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.
                     WebSimpleInPlaceActionHandler"/>
  <ActionDoer    class="com.ptc.core.ca.co.client.search.
                        CoSearchActionDoer"/>
  <On event="executeActionDoer" at="end">
    <Copy  from="%[request#where_clause]"
           to="%[frame#%[context#tableId].where_clause]"/>
    <Copy  from="true" to="%bean#%[context#tableId]#Dirty]"/>
    <Remove to="%[frame#%[context#tableId].paging_session_id]"/>
  </On>
</Action>
```
In the demo in order to enable progress monitoring functionality the action and the model doer of a search result table must have progressMonitor context dimension.

<SimpleFrame id="page.search.part" context.progressMonitor="true">
...
</SimpleFrame>

**Note:** None of the handlers or doers requires progressMonitor dimension that is used only for demo purposes to present an example of configuring search functionality that works with or without progress monitoring.

The **Location** element indicates the location of the progress monitor frame containing the status information. The frame contains a single-step wizard that holds an instance whose attributes represent WTThread properties. The frame periodically refreshes itself to present new property values. When the monitored thread is done executing a task or interrupted, the frame closes itself on the next refresh.

The wizard contains two actions -

- **Close** to stops monitoring and close the frame
- **Interrupt** to interrupt the monitored thread and close the frame. When the thread is interrupted, an empty group is returned as a task result.

The following is a frame configuration:

[com/ptc/common/progress_monitor.xml]

```
<SimpleFrame id="frame.progress.monitor" mode="edit" width="300" height="200">
  <Extend ref=":simpleframe.template" />
  <Label resource="Progress Monitor" />
  <Wizard>
    <Label resource="Progress Monitor" />
    <WizardStepElements>
      <Layout mode="view">
        <LayoutRow id="lr.state">
          <Property>
```


The refreshPeriod attribute defines the refreshing period in milliseconds. The default value is 3000. The refreshOpener attribute set to false indicates that the opener frame shouldn't be refreshed after the frame closing instruction.

The update of the frame’s model is handled by WebProgressMonitorFrameModelHandler. It updates the frame model and prepares the appropriate java script either to refresh the frame or to close it. The decision to refresh or close itself is made by the CoDoerHelper utility class static method call:

CoDoerHelper.isTaskThreadInProgress(openerFrame, envData);

The wizard's model doer CoProgressMonitorModelDoer gets the instance that holds WTThread properties and plugs it in the wizard element (instance holder).
The instance is obtained by calling a static method of the CoDoerHelper utility class:

```java
CoInstance instance =
    CoDoerHelper.getProgressMonitorInstance(openerFrame, envData);
```

The action doer CoProgressMonitorInterruptActionDoer sends a request to interrupt a monitored WTThread once the Interrupt action is triggered. CoDoerHelper utility class has a static method to perform the thread interruption:

```java
CoDoerHelper.interruptTaskThread(openerFrame, envData);
```

The wizard actions are based on the regular Ok and Cancel actions. The Interrupt action's handler has an attribute `refreshOpener` set to false to avoid refreshing the opener frame after it executes.

```xml
<WizardActions id="progress.actions">
    <Insert ref=":action.wizard.ok">
        <Label resource="Interrupt"/>
        <ToolTip resource="Interrupt"/>
        <ActionHandler class="com.ptc.core.ca.web.client.wizard.WebOkActionHandler" refreshOpener="false"/>
    </Insert>
    <Insert ref=":action.wizard.cancel">
        <Label resource="Close"/>
        <ToolTip resource="Close"/>
    </Insert>
    <Insert ref="renderer.horizontalList"/>
</WizardActions>
```

The property elements used in the wizard are related to the WTThread properties mentioned above. The wizard's instance has four NPAs of String type for every of them. Whenever a WTThread fire a property change event, those attributes get updated. At the moment the implementation of search task maintains only two of properties (state and progressCount), so the other two (status and progressPercent) don't have to be present in the configuration.
This chapter discusses the use of frame parameters in DCA.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Parameters</td>
<td>8-2</td>
</tr>
</tbody>
</table>
Frame Parameters

A frame element can have frame parameters that are name-value pairs.

**Caution:** When the frame element is recycled, its parameters are recycled with it.

**Caution:** Use packaging names to avoid accidentally overwriting parameters created by other developers. For instance, use "com.ptc.search.type" instead of "type".

To assign initial values to frame parameters, specify the appropriate number of SetParam elements.

```
<SimpleFrame id="search3.all">
  <Extend ref="common:simpleframe.template"/>
  <On event="executeModelDoer" at="end">
    <Copy from="part" to="%[frame#com.ptc.search.type_param]"/>
  </On>
</SimpleFrame>
```

To manipulate frame parameters, use the com.ptc.core.ca.co.client.frame.FrameUtils class.

```java
<<Abstract>>
CoFrameUtils

<<Static>> getFrameData(element : CoElement, envData : CoEnvData) : java.util.HashMap
<<Static>> getFrameParamValue(element : CoElement, paramName : String, envData : CoEnvData) : Serializable
<<Static>> setFrameParamValue(element : CoElement, paramName : String, paramValue : Serializable, envData : CoEnvData)
```
This chapter discusses the two types of tables supported by DCA: simple and composite. Both tables present tabular data using the same data acquisition mechanism and drawing algorithm.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Table</td>
<td>9-2</td>
</tr>
<tr>
<td>Composite Table</td>
<td>9-3</td>
</tr>
<tr>
<td>Removing Unwelcome Objects</td>
<td>9-5</td>
</tr>
<tr>
<td>Removing Rows</td>
<td>9-5</td>
</tr>
<tr>
<td>Moving Rows</td>
<td>9-7</td>
</tr>
<tr>
<td>Table Preferences</td>
<td>9-9</td>
</tr>
</tbody>
</table>
A simple table can only have an action list in its toolbar and as such, customization is quite limited. However, a simple table is much easier to configure than composite table. You will typically use simple tables when configuring pickers.

The following is an example of the configuration of a lifecycle picker for the part creation wizard located in part/lifecycle_pick.xml:

```xml
<SimpleTable id="lifecycle.all.table" selectorType="single" mode="view">
  <Type type="wt.lifecycle.LifeCycleTemplate"/>
  <SimpleColumn show=":name">
    <Label resource="Name"/>
  </SimpleColumn>
  <ModelDoer class="%common:SimpleModelTaskDoer">
    <Task id="search.table.doer.model.task" nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-query-objects">
      <TaskParameter name="search_type" value="wt.lifecycle.LifeCycleTemplate"/>
      <TaskParameter name="where" value="name=***"/>
    </Task>
  </ModelDoer>
</SimpleTable>
```

**Explanation**

Simple table in view mode configured for single select:

```xml
<SimpleTable id="lifecycle.all.table" selectorType="single" mode="view">
  <Type type="wt.lifecycle.LifeCycleTemplate"/>
  <SimpleColumn show=":name">
    <Label resource="Name"/>
  </SimpleColumn>
  <ModelDoer class="%common:SimpleModelTaskDoer">
    <Task id="search.table.doer.model.task" nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-query-objects">
      <TaskParameter name="search_type" value="wt.lifecycle.LifeCycleTemplate"/>
      <TaskParameter name="where" value="name=***"/>
    </Task>
  </ModelDoer>
</SimpleTable>
```

Type is the type of objects the table will present. There is no need to specify sub-types unless they require special treatment using a `For` statement.

```xml
<SimpleColumn show=":name">
  <Label resource="Name"/>
</SimpleColumn>
```

Present element with ID ":name" in the column with label "Name".

```xml
<ModelDoer class="%common:SimpleModelTaskDoer">
```

Invoke model task doer for data acquisition.

```xml
<Task id="search.table.doer.model.task" nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-query-objects">
```

Invoke a task by model task doer to perform search.
Search for lifecycle objects with arbitrary names.

**Composite Table**

A composite table can have a header layout and a top layout. A composite table can be contracted so that only header layout remains visible. This is achieved by placing component expand and contract actions in its header.

Both header and top layouts are real layouts and can contain any elements that can be descendents of a layout element including action lists, actions, property panels, etc.

The following is an example from `folder/config_folder_alt.xml`:

```xml
<CompositeTable id="alttable" selectorType="multi" >
    <SelectAllPrompt resource="All"/>
    <Insert ref="altpage.table.headerLayout"/>
    <Insert ref="altfirstpage.table.topLayout"/>

    <Type type="%[coolpart:CoolPart]"/>
    <Type type="%[part:WTPart]"/>
    <Type type="%[doc:WTDocument]"/>
    <Type type="%[SubFolder]"/>

    <CompositeColumn show=":name">
        <Label resource="Name"/>
        <For type="%[SubFolder]" show="altaction.folder.view"/>
    </CompositeColumn>

    <CompositeColumn show=":number">
        <Label resource="Number"/>
    </CompositeColumn>

    <ModelDoer class="%common:SimpleModelTaskDoer">
        <Task nextOperation="STDOP|com.ptc.windchill.view"
            taskDelegate="dca-list-folder-contents">
            <TaskParameter name="content_type" value="WCTYPE|wt.folder.SubFolder"/>
            <TaskParameter name="content_type" value="WCTYPE|wt.doc.WTDocument"/>
            <TaskParameter name="content_type" value="WCTYPE|wt.part.WTPart"/>
        </Task>
    </ModelDoer>
</CompositeTable>
```
Explaination

<CompositeTable id="alttable" selectorType="multi" >

Define composite table:

  <SelectAllPrompt resource="All"/>

Specifies a propmpt for embedded "Select All" action.

  <Insert ref="altpage.table.headerLayout"/>
  <Insert ref="altfirstpage.table.topLayout"/>

Use existing definitions of header and top layout:

  <Type type="%[coolpart:CoolPart]"/>
  <Type type="%[part:WTPart]"/>
  <Type type="%[doc:WTDocument]"/>
  <Type type="%[SubFolder]"/>

Present object of the specified types including sub-types (simple table supports this feature too):

  <CompositeColumn show=":name">
    <Label resource="Name"/>
    <For type="%[SubFolder]" show="altaction.folder.view"/>
  </CompositeColumn>

Present column with a label "Name" containing element with ID " :name" for all objects of all types but sub-folder. Present an element with ID "altaction.folder.view" for sub-folders. (simple table supports this feature too):

  <CompositeColumn show=":number">
    <Label resource="Number"/>
  </CompositeColumn>

Present element with ID " :number" for objects of all types. Some of the object like sub-folders will not have number and will have nothing in the cells of this column. (simple table supports this feature too):

  <ModelDoer class="%common:SimpleModelTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
      taskDelegate="dca-list-folder-contents">
      <TaskParameter name="content_type" value="WCTYPE|wt.folder.SubFolder"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.doc.WTDocument"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.part.WTPart"/>
    </Task>
  </ModelDoer>

Invoke model task doer to find folder content of designated types.
Removing Unwelcome Objects

Generally, table configuration should list all expected types using Type tags for a table to be able to present objects of those types and sub-types. However, when a rogue task returns objects of unexpected types, those objects need to be removed from a result set or DCA will throw an exception. To address this issue, DCA provides a function that removes the objects of unwelcome types - the types that are neither listed in table configuration or sub-types of those types that are listed.

The following configures a table to present part, documents, and sub-folders using hypothetical task returning all folder members.

```xml
<SimpleTable id="table">
  <Type type="%[part:WTPart]"/>
  <Type type="%[doc:WTDocument]"/>
  <Type type="%[SubFolder]"/>
  <ModelDoer class="%common:SimpleModelTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
          taskDelegate="dca-list-all-folder-members">
      <TaskParameter name="folder_path" value="/Administrator"/>
      <TaskOutParameter function="com.ptc.core.ca.co.client.doer.
        function.impl.
          CoFunction_Model_Result_Table_RemoveUnwelcomeTypes"/>
    </Task>
  </ModelDoer>
</SimpleTable>
```

Removing Rows

To remove selected rows or all rows from the table, configure actions using the following doers from com.ptc.core.ca.co.client.table package

CoTableRemoveSelectedActionDoer.java
CoTableRemoveAllActionDoer.java
Both doers can have a parameter specified as an attribute to indicates a path to the table:

```
  element= text[CS]
```

The path to the table to remove elements from. See the description of the attribute #element_path#attr_name for more details on the element path syntax.

When a parameter is not specified, the action holder of the action the doer is configured for is used.

**Actions in the Toolbar of a Table**

Configure actions and put them in a toolbar of a table.

```xml
<Action id="action.list.removeAll">
  <Extend ref="action.inPlace.template"/>
  <ToolTip resource="Delete All"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/deleteAll.gif"/>
  <Label resource="Delete All"/>
  <ClientDoer class="com.ptc.core.ca.co.client.table.
                  CoTableRemoveAllActionDoer"/>
</Action>
```
<Action id="action.list.remove" needSelection="true">
   <Extend ref="action.inPlace.template"/>
   <ToolTip resource="Delete"/>
   <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/delete.gif"/>
   <Label resource="Delete"/>
   <ClientDoer class="com.ptc.core.ca.co.client.table.CoTableRemoveSelectedActionDoer"/>
</Action>

Actions Outside a Table

You need to specify the ID of the table via the element attribute. The ID must be unique for a frame.

<Action id="action.list.removeAll">
   <Extend ref="action.inPlace.template"/>
   <ToolTip resource="Delete All"/>
   <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/deleteAll.gif"/>
   <Label resource="Delete All"/>
   <ClientDoer class="com.ptc.core.ca.co.client.table.CoTableRemoveAllActionDoer" element="source_table_id"/>
</Action>

Action in a Cell

To configure an action to remove one row, just place it in the cell of a table.

<Action id="action.row.remove">
   <Extend ref="action.inPlace.template"/>
   <ToolTip resource="Delete"/>
   <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/delete.gif"/>
   <Label resource="Delete"/>
   <ClientDoer class="com.ptc.core.ca.co.client.table.CoTableRemoveSelectedActionDoer"/>
</Action>

Moving Rows

To move selected rows or all rows from the table to another table, configure actions using the following doers from com.ptc.core.ca.co.client.table package

CoTableMoveSelectedActionDoer
CoTableMoveAllActionDoer

Both doers require specification of the ID of the other table via target_element attribute. The ID must be unique for a frame.
Actions in the Toolbar of a Table.

Configure actions and put them in a toolbar of a table.

```xml
<Action id="moveDown">
  <Extend ref="common:action.list.moveSelectedDown"/>
  <ClientDoer class="com.ptc.core.ca.co.client.table.
    CoTableMoveSelectedActionDoer"
    target_element="target_table_id"/>
</Action>

<Action id="moveAllDown">
  <Extend ref="common:action.list.moveAllDown"/>
  <ClientDoer class="com.ptc.core.ca.co.client.table.
    CoTableMoveAllActionDoer"/>
</Action>
```
Actions Outside a Table

You need to specify the ID of the table via element attribute. The ID must be unique for a frame.

```xml
<Action id="moveDown">
  <Extend ref="common:action.list.moveSelectedDown"/>
  <ClientDoer class="com.ptc.core.ca.co.client.table.CoTableMoveSelectedActionDoer"
    element="source_table_id"
    target_element="target_table_id"/>
</Action>
```

Action in a Cell

To configure an action to remove one row, just place it in the cell of a table.

```xml
<Action id="moveAllDown">
  <Extend ref="common:action.list.moveAllDown"/>
  <ClientDoer class="com.ptc.core.ca.co.client.table.CoTableMoveAllActionDoer"
    target_element="target_table_id"/>
</Action>
```

Table Preferences

DCA provides a mechanism for table customization to allow modifications of column visibility, column order, and column sorting. The same mechanism can also be used for a tree, but changes are restricted to column visibility and column order.

DCA stores all the changes in Windchill preference mechanism for a given user.

Caution: Current implementation supports sorting by column values only when column cells contain simple (single-attribute based) properties.

Table Preference Wizard

The DCA demo provides an example of a table preference wizard to configure column visibility, column order, and column sorting. The user is expected to use the wizard when more than a single change needs to be applied to a table at once.

In the DCA demo the following action is included in the toolbar of a table to enable access to a preference wizard

```xml
<Action id="action.table.pref.edit">
  <Extend ref="common:action.newTemporaryFrame.template"/>
  <Label resource="Preferences"/>
</Action>
```
A preference dialog has one tab to set up sorting criteria, and another one for column visibility and order.

To enable column sorting, select the name of the column and sort mode:

1. Click **Update**. The next row will appear where you will be able to select another column.

2. To disable column sorting click **Clear**.

On a tab controlling column visibility and column order, you will be able to move a column to the next or previous position. For your convenience, you will be able to move a column by more than one position using **jump** actions.
A configuration of a preference wizard uses a set of doers written specifically to support preference mechanism. They are generic and will handle any DCA table. See com.ptc.preferences package for more details.

Column jump is implemented by the doers

```
com.ptc.core.ca.co.client.table.pref.CoColumnJumpUpActionDoer
com.ptc.core.ca.co.client.table.pref.CoColumnJumpDownActionDoer
```

By default these doers moves a column by 5 positions. The number of positions per jump can be configured by specifying a value attribute. For example

```
<ActionDoer class="com.ptc.core.ca.co.client.table.pref.
  CoColumnJumpUpActionDoer" value="10" />
```

instructs a doer to jump 10 positions a time.

**In-Place Table Preferences**

Frequently users like to move a single column or sort using its values. DCA provides an ability to customize a table by placing special in-place actions in table column headers.

**Caution:** When a user triggers one of the sorting actions in a column header of a table, DCA removes all existing sorting preferences for this table.
All the doers used to configure in-place actions are generic. The following is an example of an action to sort by column values in descending order.

```xml
<Action id="action.column.sort.descending">
  <ToolTip resource="Sort Descending"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/sortup.gif"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.table.pref.CoColumnSortDescendingActionDoer"/>
  <Insert ref="common:renderer.action"/>
</Action>
```
This chapter discusses the composite tree, the only tree supported by DCA.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layouts</td>
<td>10-2</td>
</tr>
<tr>
<td>Explanation</td>
<td>10-3</td>
</tr>
<tr>
<td>Root Node Visibility</td>
<td>10-4</td>
</tr>
</tbody>
</table>
The composite tree can have both a header and a top layout. The composite tree can be contracted so that only the header layout remains visible. This is achieved by placing expand and contract actions in its header.

Both header and top layouts are real layouts and can contain any elements that can be descendents of a layout element; including action lists, actions, property panels, etc.

The following is an example from part/part_usedby.xml

```xml
<Action id="tree.node.action">
  <ActionHandler class="com.ptc.core.ca.web.client.tree.WebTreeNodeActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.tree.CoDefaultTreeNodeActionDoer"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/tree/treeNodeAction.jsp">
    <Option id="image" param="contract1" value="/wtcore/images/com/ptc/core/ca/web/tree/contract1.gif"/>
    <Option id="image" param="contract2" value="/wtcore/images/com/ptc/core/ca/web/tree/contract2.gif"/>
    ...
  </Renderer>
</Action>

<List id="tree.node.usedby.list">
  <Insert ref="tree.node.action"/>
  <Insert ref=":name"/>
</List>

<CompositeTree id="view.usedby.tree" selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Type type="%[WTPart]"/>
  <CompositeColumn show="tree.node.usedby.list">
    <Label resource="Name"/>
  </CompositeColumn>
  <CompositeColumn show=":number">
    <Label resource="Number"/>
  </CompositeColumn>
  <ModelDoer class="com.ptc.core.ca.co.client.tree.CoDefaultTreeModelTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
      taskDelegate="dca-default-tree">
      <TaskParameter name="target_type" value="wt.part.WTPartUsageLink"/>
      <TaskParameter name="direction" value="usedBy"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.part.WTPart"/>
  </ModelDoer>
</CompositeTree>
```
Explanation

<CompositeTree id="view.usedby.tree" selectorType="multi" >

Composite tree in view mode supporting multi select.

<SelectAllPrompt resource="All"/>

Specifies a prompt for "Select All" embedded action.

<Type type="\[%[WTPart]\]"/>

Presents parts as its nodes:

<CompositeColumn show="tree.node.usedby.list">
    <Label resource="Name"/>
</CompositeColumn>

Presents a list of elements in the first column:

<List id="tree.node.usedby.list">
    <Insert ref="tree.node.action"/>
    <Insert ref=":name"/>
</List>

The list includes tree node action and name property. The action is specially configured to render itself using an appropriate image with an offset calculated based on the location of the node in a tree. All the images are configurable via options.

A list can include other properties to present the state of the object (e.g. expander folder vs. contracted one), actions, etc.

<CompositeColumn show=":number">
    <Label resource="Number"/>
</CompositeColumn>

Presents an element with ID ":number".

<ModelDoer class="com.ptc.core.ca.co.client.tree.
    CoDefaultTreeModelTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view" 
        taskDelegate="dca-default-tree">
        <TaskParameter name="target_type" 
            value="wt.part.WTPartUsageLink"/>
        <TaskParameter name="direction" 
            value="usedBy"/>
        <TaskParameter name="content_type" 
            value="WCTYPE\|wt.part.WTPart"/>
    </Task>
</ModelDoer>
Execute model doer for a tree. A doer will invoke a task for nodes when needed. A task will navigate $\textit{WTPartUsageLink}$ in the $\textit{usedBy}$ direction looking for $\textit{WTParts}$ returning the attributes for viewing.

When expand action on a tree node is triggered, the same task needs to be executed to return information about child nodes. An action should be configured to have special action handler and action doer.

```xml
<Action id="tree.node.action">
  <ActionHandler class="com.ptc.core.ca.web.client.tree.WebTreeNodeActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.tree.CoDefaultTreeNodeActionDoer"/>
</Action>
```

$\textit{WebTreeNodeActionHandler}$ updates the model associated with the tree node. The model’s action image and action indent icon list is modified based on the type (leaf or node) and state (first node, middle node, expanded or contracted) of the tree node. The handler uses options to locate resources. The handler executes the configured doer.

The tree node action doer will locate model doer configuration for a tree and invoke its task to perform data acquisition for the node an action was triggered for.

### Root Node Visibility

The visibility of the root node is controlled by $\textit{showRootNode}$ attribute. To make it invisible set its value to false:

```xml
<CompositeTree id="tree" showRootNode="false">
  ...
</CompositeTree>
```
This chapter discusses the Enumerator, an element that presents the user with a list of available choices.

**Note:** The Enumerator is not related to the enumerated property, which presents a list of attribute values to the user even though it may look and function exactly the same.

The Enumerator has a list of available string values, with one of values selected as the current value.

**Topic**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statically-Defined Enumerator</td>
<td>11-2</td>
</tr>
<tr>
<td>Data-Driven Enumerator</td>
<td>11-2</td>
</tr>
</tbody>
</table>
**Statically-Defined Enumerator**

Enumerator can be statically defined in the configuration specification.

1. Configure labels for an enumerator.

```xml
<LabelArea id="search3.enumerator.type.part" labelType="text">
  <Label resource="Part"/>
  <ToolTip resource="Part"/>
</LabelArea>
<LabelArea id="search3.enumerator.type.doc" labelType="text">
  <Label resource="Document"/>
  <ToolTip resource="Document"/>
</LabelArea>
<LabelArea id="search3.enumerator.type.folder" labelType="text">
  <Label resource="Folder"/>
  <ToolTip resource="Folder"/>
</LabelArea>
```

2. Configure an enumerator to have labels and values. The first value is a default value.

```xml
<Enumerator id="search3.enumerator.type">
  <Insert ref="search3.enumerator.type.part" value="part"/>
  <Insert ref="search3.enumerator.type.doc" value="doc"/>
  <Insert ref="search3.enumerator.type.folder" value="folder"/>
</Enumerator>
```

3. You have the option of setting an initial value from a frame parameter value.

```xml
<Enumerator id="search3.enumerator.type">
  <Insert ref="search3.enumerator.type.part" value="part"/>
  <Insert ref="search3.enumerator.type.doc" value="doc"/>
  <Insert ref="search3.enumerator.type.folder" value="folder"/>
  <On event="initModel" at="end">
    <Copy from="%[frame#com.ptc.search.type_param]" to="%[enum#.]"/>
  </On>
</Enumerator>
```

**Data-Driven Enumerator**

The DataDrivenEnumerator element has been specifically designed to present a value of an attribute of an Info*Engine element and a list of potential values (list of options) to choose from. To find the Info*Engine element, the data-driven enumerator searches for the nearest instance holder ancestor. A list of options can be configured statically, dynamically, or using mixed approach.

Both Need and NeedNpa configuration elements can be used to bind the data-driven enumerator to the attributes of Info*Engine element. The only difference is in the way Need and NeedNpa elements are treated by the doers. The doers provided by DCA ignore the NeedNpa configuration element while taking into account the Need configuration elements when preparing task parameters for a task to retrieve Info*Engine elements. The difference might be important when, for instance, a data-driven enumerator is included in the property panel when we
want the task to retrieve the value but don’t want to ask for options as data sources do not store them in a format DCA understands.

**Static-List Options Based**

A list of options can be configured using `LabelArea` elements similarly to how it’s done for the `Enumerator` element. In this following the current value of the enumerator is bound to a value of an attribute `enum_value`. The list of `LabelArea` elements is used to construct the list of options.

```xml
<DataDrivenEnumerator id="enum">
  <LabelArea id="1" labelType="text" value="1" default="true">
    <Label resource="Option 1"/>
  </LabelArea>
  <LabelArea id="2" labelType="text" value="2">
    <Label resource="Option 2"/>
  </LabelArea>
  <Need attribute="enum_value" as="value"/>
</DataDrivenEnumerator>
```

**Dynamic-List Options Based**

A list of options for `DataDrivenEnumerator` can be defined dynamically.

**Task-Based List of Options**

A list of options can be task-based. In this case the data-driven enumerator must be configured to execute a model doer that, in turn, is configured to execute a task. The task is responsible for returning a group of Info*Engine elements. The doer is responsible for extracting attribute values of those element for a specified attribute to be used as labels and UFIDs to be used as values. The following is an example:

```xml
<DataDrivenEnumerator id="dde.enumElement">
  <Need attribute="dde.defaultValue" as="value"/>
  <ModelDoer class="com.ptc.core.ca.co.client.primitive.CoDataDrivenEnumeratorModelTaskDoer" labelAttribute="name">
    <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-list-folder-members">
      <TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoSimpleFunction_Model_Param_MainObjectIdentity"/>
      <TaskParameter name="folder_path" computedValue="com.ptc.refimpl.co.computedvalues.PersonalFolderComputedValue"/>
      <TaskParameter name="content_type" value="%[doc:WTDocument]%"/>
    </Task>
  </ModelDoer>
</DataDrivenEnumerator>
```
In this example the list of options will be created using names of the documents in the personal folder as labels and their UFIDs as values.

Sometimes it’s desirable to use a value of an attribute as an option value instead of using element’s UFID. In this case the name of the attribute can be specified via an additional `valueAttribute` attribute.

```xml
<ModelDoer class="com.ptc.core.ca.co.client.primitive.
            CoDataDrivenEnumeratorModelTaskDoer"
    lableAttribute="name"
    valueAttribute="name"/>
```

**Caution:** The doer used in the example above does not follow "simple" approach. As such it does not provide task parameter containing UFID of any elements. As you might know, DCA uses the task dispatching mechanism as the data acquisition layer. This task dispatching uses the Dispatch-Tasks webject. This webject is documented in the *Windchill Info*Engine Users Guide (it's a management webject!). As you can see in its documentation the dispatch-tasks webject takes as input a group with elements with UFID (obid with repository information) & fti (generalized type/class information) and an action argument. This triple is used to find the right Info*Engine task to execute from the LDAP entries of the task delegate. The Repository from the UFID gives the repository type and supporting adapter. This repository type and the fti & action leads to the right task implementation. This mechanism takes into account the type hierarchy that was provided by the function specified by the CLIMBER parameter in the Dispatch-Tasks webject. This behavior has as consequence that you always have to provide an input element (with UFID & fti) to execute any tasks from DCA, because otherwise the task dispatcher is not able to find the right task.

When no element is available, add the following task parameter

```xml
<TaskParameter function="com.ptc.core.ca.co.client.doer.
    function.impl.CoFunction_Model_Param_FudgeElement"/>
```
Advanced Task-Based List of Options

Sometimes it’s desirable to specify a few options statically and the rest of the options dynamically. The following is an example:

```xml
<DataDrivenEnumerator id="dde.enumElement">
  <NeedNpa attribute="dde.defaultValue" as="value"/>
  <LabelArea labelType="text" value="" default="true">
    <Label resource="-- Select a document --"/>
  </LabelArea>
  ...
  <ModelDoer class="com.ptc.core.ca.co.client.primitive.CoDataDrivenEnumeratorModelTaskDoer" attribute="name">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
      taskDelegate="dca-list-folder-members">
      <TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoSimpleFunction_Model_Param_MainObjectIdentity"/>
      <TaskParameter name="folder_path"
        computedValue="com.ptc.refimpl.co.computedvalues.PersonalFolderComputedValue"/>
      <TaskParameter name="content_type"
        value="%[doc:WTDocument]"/>
    </Task>
  </ModelDoer>
</DataDrivenEnumerator>
```

In this example, the list of options will consist of options specified by LabelArea elements and options created using the Info*Engine elements from a task result. DCA will use values of name attribute of Info*Engine elements as labels to construct options and their UFIDs as values.
Windchill Enumeration Type-Based List of Options

A list of options can be based on the Windchill enumeration. In this case the data-driven enumerator must be configured to execute a model doer that creates its options based on the specified Windchill enumeration class.

Any number of LabelArea elements can be specified. When no LabelArea elements are specified, or none of them is marked as default, default value specified for the enumeration type is used. When the enumeration type does not have a default value, the first value is used.

Attribute-Based List of Options

DataDrivenEnumerator can be bound to two attributes of an Info*Engine element. One of the attribute must contain its value, and the other one must contain a list of options.

Any number of LabelArea elements can be specified. When no LabelArea elements are specified, or none of them is marked as default, default value specified for the enumeration type is used. When the enumeration type does not have a default value, the first value is used.
**Programmatically Populated List of Options**

The list of options can be created programmatically. The following is an example:

```java
Object red_option =
    new CoEnumeratorUtils.SimpleEnumeratorElement("1","Red", "r");
Object blue_option =
    new CoEnumeratorUtils.SimpleEnumeratorElement("2","Blue", "b");
Object green_option =
    new CoEnumeratorUtils.SimpleEnumeratorElement("3","Green","g");
ArrayList color_options = new ArrayList();
color_list.add(red_option);
color_list.add(blue_option);
color_list.add(green_option);
ArrayList color_option_list =
    CoEnumeratorUtils.createEnumeratorSubElementConfigurations(     
        color_options, null);
// setting the current value
CoPropertyUtils.setAttributeValue(ie_element, "color_value", "b");
// setting the list of available values
CoPropertyUtils.setAttributeValue(ie_element, "color_options",  
    color_option_list);
```

**Windchill-Enumerated Type-Based List of Options**

A list of options can be configured based on the standard Windchill-enumeredated type wt.fc.EnumeratedType. DCA provides a class that can be used to configure task output parameter to create a needed attribute on an Info*Engine element of the instance holder ancestor.

```xml
<Task ...
   ...  
   <TaskOutParameter function=",com.ptc.core.ca.co.client.windchill.  
   CoFunction_Model_Result_CreateAttributesForDataDrivenEnumerator"  
   class="wt.lifecycle.State"/>
   ...
</Task>
```

Configured as in the example above, the function will create two attributes with the names State_options and State_value containing values to represent available choices and a current value respectively. To do that, it finds the list of all legitimate enumerated values for the specified class, builds the label configuration list, and assigns it to State_options attribute. The function takes the specified class name and uses it a prefix to construct attribute names as

```xml
<prefix>_options
<prefix>_value
```

For the example above the configuration of the data-driven enumerator should look as follows:

```xml
<DataDrivenEnumerator id="lifecyclestate">  
    <NeedNpa attribute="State_options"/>
    <Need attribute="State_value"/>
</DataDrivenEnumerator>
```
Sometimes it’s undesirable to use the default naming convention for attribute names. In such a case a prefix can be specified via configuration specification.

When the function discovers a prefix attribute, it will use its value to construct attribute names that are `partState_options` and `partState_value` in the example above.

### Sorting Enumerated Entries

DCA provides an ability to specify a comparator to enable sorting of entries of the enumerator. The following is an example of its configuration:

```xml
<Comparator id="comparator.template" class="com.ptc.core.ca.co.client.primitive.CoDefaultDataDrivenEnumeratorComparator" direction="ascending"/>
<DataDrivenEnumerator id="enum">
  <Need attribute="lifeCycle.id^name" as="value"/>
  <LabelArea id="la.0" labelType="text" value="" default="true">
    <Label resource="-- No Selection --"/>
  </LabelArea>
  <Insert ref="common:comparator.template" direction="descending"/>
  <ModelDoer class="%[common:DDEModelTaskDoer]" labelAttribute="name" valueAttribute="name">
    <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-query-objects">
      <TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoFunction_Model_Param_FudgeElement"/>
      <TaskParameter name="search_type" value="wt.lifecycle.LifeCycleTemplate"/>
      <TaskParameter name="where" value="name=**"/>
    </Task>
  </ModelDoer>
</DataDrivenEnumerator>
```

In this example the enumerated entries will be sorted in the descending order but the entry with a label --No Selection-- will remain at the top as it’s value is an empty string. The comparator must implement the `com.ptc.core.ca.co.client.primitive.CoDataDrivenEnumeratorComparator` interface.
This section describes how to implement "Search" functionality similar to the Windchill local search, using the DCA framework. An existing example in the DCA demo implements search by object type and two attributes (name, number) used to construct a search criterion. Those attributes are not applicable for some object types (Folder).

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>12-2</td>
</tr>
<tr>
<td>Action List-Based Implementation</td>
<td>12-2</td>
</tr>
<tr>
<td>Enumerator-Based Implementation</td>
<td>12-10</td>
</tr>
<tr>
<td>Frame Property-Based Implementation</td>
<td>12-13</td>
</tr>
<tr>
<td>Connector-Based Implementation (recommended)</td>
<td>12-16</td>
</tr>
<tr>
<td>Search With Complex Criteria</td>
<td>12-20</td>
</tr>
<tr>
<td>Search Term Producer</td>
<td>12-24</td>
</tr>
<tr>
<td>Search With IBAs</td>
<td>12-28</td>
</tr>
<tr>
<td>Search With Paging</td>
<td>12-31</td>
</tr>
<tr>
<td>Multiple Searches</td>
<td>12-37</td>
</tr>
</tbody>
</table>
Introduction

Typically, to let the user to search for objects the system must present a mechanism enabling a user to choose a search criteria. When DCA analyzes the search criteria, it constructs a so-called where-clause that it passes later on to Info*Engine task.

For the purpose of this section, a where-clause consists of search terms joined by the '& relation.

\[
\text{where-clause} := \text{search-term}_1 \& \ldots \& \text{search-term}_N
\]

A search-term consists of a combination of an attribute, its value, and a relation.

\[
\text{search-term} := \text{attr-name} \ \text{relation} \ \text{attr-value}
\]

Typically, a property panel in search mode is used as a mechanism to enable the user to enter a search criteria. DCA automatically constructs a where-clause based on the structure of the property panel. DCA would look for following elements in search mode to determine which attributes need to participate in the construction of a where-clause:

- properties
- data driven enumerators bound as value to attributes using Need configuration elements

Since search property panel can be different for different object types, different set of look up attributes can be used for different types without writing Java code.

The rules for the construction of a where-clause are:

- Search term is either based on an attribute the property needs, or an attribute the value of a data driven enumerator is bound to.
- When a property input field is empty, the attribute is not used to create a where clause.
- Only '=' (equal) relation is supported for a search term unless complex criterion can be used (see below). As a result, a search-term looks as follows:

\[
\text{search-term} := \text{attr-name} = \text{attr-value}
\]

Action List-Based Implementation

The Demo Search function is implemented as a simple frame with a property panel in search mode and a table element in view mode. The property panel holds the property elements presenting available search criteria (attribute values), as well as the "in-place" search action.

To access the demo search functionality use the following URL:

The table displays the list of objects that meet specified search criterion. The choice of search properties, and table columns is hardcoded in the configuration file.

**Configuration**

The demo *Search* is configured in:

```xml
<config-base>/com/ptc/search/config_search.xml
<config-base>/com/ptc/part/part_search.xml
<config-base>/com/ptc/doc/doc_search.xml
<config-base>/com/ptc/folder/folder_search.xml
```

**Executing Search**

The *Search* action is an 'in-place' action. When triggered, it creates a `where` clause and communicates it to the table element. The Table element runs a search task using the supplied `where` clause and displays the search result. The table can be configured to display different set of columns without any code writing.

**Example**

The following is an example of a demo configuration:

**Frame Element**

```xml
<SimpleFrame id="page.search">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Search"/>
  <SimpleLayout>
    <LayoutRow id="rowSep1">
      <Insert ref="common:separator.horizontal"/>
    </LayoutRow>
    <LayoutRow id="rowSearchPanel">
      <Insert ref="search.panel"/>
    </LayoutRow>
    <LayoutRow id="rowSep2">
      <Insert ref="common:separator.horizontal"/>
    </LayoutRow>
    <LayoutRow id="rowTable">
      <Insert ref="search.table"/>
    </LayoutRow>
  </SimpleLayout>
</SimpleFrame>
```

The frame contains a property panel `search.panel` and a table `search.table`. The elements `common:separator.horizontal` display the horizontal lines before and after the property panel. They do not participate in the search process and are used for cosmetic reasons only.
Property Panel:

```xml
<SimplePropertyPanel id="search.panel" type="%[part:WTPart]"
mode="search">
  <LayoutRow id="rowAttr1">
    <Insert ref=":name"/>
  </LayoutRow>
  <LayoutRow id="rowAttr2">
    <Insert ref=":number"/>
  </LayoutRow>
  <LayoutRow id="rowSearchAction">
    <Insert ref="search:search.action.alt"
context.tableId="target.table.id"/>
  </LayoutRow>
</SimplePropertyPanel>
```

The property panel element contains one property element for every attribute in
the search criteria. A model task doer is specified to execute the specified task that
creates an Info*Engine element and populates its attributes with initial values.
Use task parameters to set those values through configuration specifications.

In the demo, `name` and `number` attributes are initialized with empty strings
(value="name=" , value="number="). To use other initial values, value attributes need to be modified, like:

```xml
  <TaskParameter name="field" value="name=A*"/>
  <TaskParameter name="field" value="number=*"/>
```

In this case, the Search page will be displayed with A* in the name field, and * in
the number field.

Search Action

The `search.action.alt` element is configured to execute a search specific
doer:

```xml
<Action id="search.action.alt">
  <Label resource="Search"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.
WebSimpleInPlaceActionHandler"/>
  <ActionDoer class="com.ptc.core.ca.co.client.search.
CoSearchActionDoer"/>
```

The action uses CoSearchActionDoer to save the user input to pass it to the table, in the form of a where clause string. Also, it uses either WebSimpleInPlaceActionHandler or WebProgressMonitorInPlaceActionHandler depending on the configured context, so the appropriate progress monitoring mode can be switched.

**Note:** None of the handlers or doers requires progressMonitor dimension that is used only for demo purposes to present an example of configuring search functionality that works with or without progress monitoring.

**Search Result Table**

```
<SimpleTable id="search.table.template" selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Type type="%[WTPart]%"/>
  <SimpleColumn show=":name">
    <Label resource="Name"/>
  </SimpleColumn>
  <SimpleColumn show=":number">
    <Label resource="Number"/>
  </SimpleColumn>
  <SimpleColumn show=":waitingpromotion">
    <Label resource="Waiting"/>
  </SimpleColumn>
  <SimpleColumn show=":checkoutstatus">
    <Label resource="Status"/>
  </SimpleColumn>
  <SimpleColumn show=":createstamp">
    <Label resource="Created"/>
  </SimpleColumn>
  <SimpleColumn show=":folder.browser.actions" sortable="false">
    <Label resource="Actions"/>
  </SimpleColumn>
</SimpleTable>

<SimpleTable id="search.table.part" clientSideSortable="false">
  <Extend ref="search.table.template"/>
</SimpleTable>
```
The table element is configured to execute the search task and to present search result. The CoDefaultSimpleModelTaskDoer executes a task via the task delegate dca-query-objects that performs searching using object type, where clause, and sorting and paging parameters prepared by the task parameter functions.

Also, the task will execute only if there is either where or paging_session_id exists in the task parameters. The configured task validator is responsible for this:

The TaskParameter element is responsible for the type of the objects, which is wt.part.WTPart in this case:
Type Selector

A type selector is implemented in such a way that selection of a type results in an appropriate frame being loaded in the same window. This way the search frame for one type can be configured differently than a frame for another type. There are two ways to achieve this behavior.

Active Type Selector

The Type selector is configured as a drop-down list of "go-to" actions. When the user triggers an action, the window loads an appropriate frame.

DCA demo supports the search of parts, documents, and folders. For every type, there is a basic set of attributes that can used to construct search criteria:

- Part (name, number, source)
- Document (name, number)
- Folder (name)

Use the following URLs to access search functionality:


To implement active type selector configure and include an action list in the search property panel definition.

```xml
<SimplePropertyPanel id="search.panel.part" type="%[WTPart]" mode="search">
  <Insert ref="search:search.panel.row.type.chooser.list"/>
  <Insert ref="search:search.panel.row.property.name"/>
  <Insert ref="search:search.panel.row.property.number"/>
  <Insert ref="search:search.panel.row.property.source"/>
  <LayoutRow id="rowSearchAction">
    <Insert ref="search:search.action.alt" context.tableId="search.table.part"/>
  </LayoutRow>
  <Insert ref="search.panel.model.doer"/>
</SimplePropertyPanel>
```

Configure a selector by defining an action list with a label element:

```xml
<LayoutRow id="search.panel.row.type.chooser.list">
  <Insert ref="search.action.type.label"/>
  <Insert ref="search.action.list.type"/>
</LayoutRow>
```

```xml
<LabelArea id="search.action.type.label" labelType="text">
  <Label resource="Type:"/>
  <ToolTip resource="Type"/>
</LabelArea>
```
Define a list of "go-to" actions:

<ActionList id="search.action.list.type">
  <Insert ref="search.action.type.part"/>
  <Insert ref="search.action.type.doc"/>
  <Insert ref="search.action.type.folder"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/location.jsp"/>
</ActionList>

Define actions themselves:

<Action id="search.action.type.part">
  <Label resource="Part"/>
  <ToolTip resource="Part"/>
  <Location ref="com.ptc.part:page.search.part"/>
  <ActionHandler class="com.ptc.core.ca.co.client.search.CoSearchTypeActionHandler"/>
</Action>

<Action id="search.action.type.doc">
  <Label resource="Document"/>
  <ToolTip resource="Document"/>
  <Location ref="com.ptc.doc:page.search.doc"/>
  <ActionHandler class="com.ptc.core.ca.co.client.search.CoSearchTypeActionHandler"/>
</Action>

<Action id="search.action.type.folder">
  <Label resource="Folder"/>
  <ToolTip resource="Folder"/>
  <Location ref="com.ptc.folder:page.search.folder"/>
  <ActionHandler class="com.ptc.core.ca.co.client.search.CoSearchTypeActionHandler"/>
</Action>

**Passive Type Selector**

The Type selector is configured as a drop-down list of disabled "go-to" actions. It essentially acts similar to enumeration property as no actions can be triggered but a location of a selected one is used. A specially configured action "Change Type" is used to load an appropriate frame in the current window when the user triggers it.

The DCA demo supports the search of parts, documents, and folders. For every type, there is a basic set of attributes that can used to construct search criteria:

- **Part** (name, number, source)
- **Document** (name, number)
- **Folder** (name)

Use the following URLs to access search functionality:

http://<yourhost>/Windchill/wtcore/jsp/com/ptc/core/ca/web/gw/gw.jsp?alias=com.ptc.part:page.search.part&amp;appId=sp
To implement passive type selector configure and include an action list and an action in the search property panel definition.

```xml
<SimplePropertyPanel id="search.panel.part" type="%[WTPart]"
    mode="search">
    <Insert ref="search:search.panel.row.type.chooser.link"/>
    <Insert ref="search:search.panel.row.property.name"/>
    <Insert ref="search:search.panel.row.property.number"/>
    <Insert ref="search:search.panel.row.property.source"/>
    <LayoutRow id="rowSearchAction">
        <Insert ref="search:search.action.alt" context.tableId="search.table.part"/>
    </LayoutRow>
    <Insert ref="search.panel.model.doer"/>
</SimplePropertyPanel>
```

Configure a selector by defining an action list with a label element:

```xml
<LayoutRow id="search.panel.row.type.chooser.link">
    <Insert ref="search.action.type.label"/>
    <Insert ref="search.action.list.type">
        <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/location2.jsp"/>
    </Insert>
    <Insert ref="search.action.type.change"/>
</LayoutRow>
```

A special renderer, location2.jsp, must be used to present an action list with actions in a disabled state. Here we’ve decided to reuse the definition of an action list configured in the example of an active type selector by overriding its renderer. It could have been created as a separate action list if needed.

Here is the configuration for the type changing action:

```xml
<Action id="search.action.type.change">
    <Label resource="Change Type"/>
    <ActionHandler class="com.ptc.core.ca.co.client.search.
        CoSearchActionHandler"/>
    <ClientDoer class="com.ptc.core.ca.web.client.search.
        WebSearchTypeActionClientDoer"/>
    <LocationChooser id="search.action.list.type"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/
        action.jsp"/>
</Action>
```

It is crucial to use the correct handler, renderer, and location chooser element pointing to an action list.
Enumerator-Based Implementation

It is possible to define multiple search property panels with corresponding search result tables and conditionally draw them based on the values of one or more enumerator elements.

The following example presents how to implement a search for parts, documents, and folders.

1. Configure a group of elements containing a label, enumerator, and an action.

```xml
<Context>
  <Synonym id="Folder" value="wt.folder.Folder"/>

  <ElementGroup id="search2.type_chooser">
    <List>
      <Insert ref="search2.enumerator.type.label"/>
    </List>
  </ElementGroup>
</Context>
```
2. Configure enumerator to have one of three values. Default value set to part.

```xml
<Enumerator id="search2.enumerator.type">
  <Insert ref="search2.enumerator.type.part" value="part" default="true"/>
  <Insert ref="search2.enumerator.type.doc" value="doc"/>
  <Insert ref="search2.enumerator.type.folder" value="folder"/>
</Enumerator>
```

3. Configure labels for enumerator’s values

```xml
<LabelArea id="search2.enumerator.type.part" labelType="text">
  <Label resource="Part"/>
  <ToolTip resource="Part"/>
</LabelArea>

<LabelArea id="search2.enumerator.type.doc" labelType="text">
  <Label resource="Document"/>
  <ToolTip resource="Document"/>
</LabelArea>

<LabelArea id="search2.enumerator.type.folder" labelType="text">
  <Label resource="Folder"/>
  <ToolTip resource="Folder"/>
</LabelArea>
```

4. Configure in-place action the user has to trigger when value of the enumerator changes.

```xml
<Action id="search2.enumerator.type.change">
  <Label resource="Change Type"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.doer.CoNoOpActionDoer"/>
</Action>
```

In DCA Web, Action does not have to execute any operations. DCA Web mines a request to generically initialize values of enumerator objects.

5. Configure a table the user will be able to populate with final results. It must have no-op doer.

```xml
<SimpleTable id="search2.table2">
  <Extend ref="folder:table"/>
  <ModelDoer class="com.ptc.core.ca.co.client.doer.CoNoOpModelDoer"/>
</SimpleTable>
```
6. Configure a frame and configure its content to conditionally include one of the search layouts.

```xml
<SimpleFrame id="search2.all">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Search"/>
  <SimpleLayout>
    <LayoutRow id="tc">
      <Insert ref="search2.type_chooser"/>
    </LayoutRow>
    <LayoutRow id="s">
      <Insert ref="common:separator.horizontal"/>
    </LayoutRow>
    <LayoutRow id="p">
      <Insert ref="part:search2.layout.part"/>
      <DrawHandler class="com.ptc.core.ca.co.client.element.CoDrawController">
        <Check value="%[enum#search2.enumerator.type]==part"/>
      </DrawHandler>
    </LayoutRow>
    <LayoutRow id="d">
      <Insert ref="doc:search2.layout.doc"/>
      <DrawHandler class="com.ptc.core.ca.co.client.element.CoDrawController">
        <Check value="%[enum#search2.enumerator.type]==doc"/>
      </DrawHandler>
    </LayoutRow>
    <LayoutRow id="f">
      <Insert ref="folder:search2.layout.folder"/>
      <DrawHandler class="com.ptc.core.ca.co.client.element.CoDrawController">
        <Check value="%[enum#search2.enumerator.type]==folder"/>
      </DrawHandler>
    </LayoutRow>
    <LayoutRow id="t">
      <Insert id="search2.table2" ref="com.ptc.folder:table" type="%[Folder]">
        <ModelDoer class="com.ptc.core.ca.co.client.doer.CoNoOpModelDoer"/>
      </Insert>
    </LayoutRow>
  </SimpleLayout>
</SimpleFrame>
</Context>
```

7. Configure search layouts for every type:

```xml
<SimpleLayout id="search2.layout.part">
  <LayoutRow id="p1">
    <Insert ref="search2.panel"/>
  </LayoutRow>
  <LayoutRow id="p2">
    <Insert ref="common:separator.horizontal"/>
  </LayoutRow>
</SimpleLayout>
```
8. Configure search panel for every layout:

```xml
<SimplePropertyPanel id="search2.panel" type="%[WTPart]"
                  mode="search">
  <Insert ref="search:search.panel.row.property.name"/>
  <Insert ref="search:search.panel.row.property.number"/>
  <Insert ref="search:search.panel.row.property.source"/>
  <LayoutRow id="rowSearchAction">
    <Insert ref="search:search.action.alt"
            context.tableId="search.table.part"/>
  </LayoutRow>
</SimplePropertyPanel>
```

9. Configure search result table (not shown)

10. Configure action lists to let user select and move items to the final list

```xml
<ActionList id="search2.move" actionListType="horizontal">
  <Action id="moveDown">
    <Extend ref="common:action.list.moveSelectedDown"/>
    <ClientDoer class="%common:CoTableMoveSelectedDoer"
                element="search.table.part"
                targetElement="search2.table2"/>
  </Action>
  <Action id="moveAllDown">
    <Extend ref="common:action.list.moveAllDown"/>
    <ClientDoer class="%common:CoTableMoveAllDoer"
                element="search.table.part"
                targetElement="search2.table2"/>
  </Action>
</ActionList>
```

## Frame Property-Based Implementation

It is possible to define multiple search property panels with corresponding search result tables and conditionally draw them based on the values of one or more frame properties.

1. Configure a group of elements containing a label, enumerator, and an action.

```xml
<Context>
  <ElementGroup id="search3.type_chooser">
    <List>
      <Insert ref="search3.enumerator.type.label"/>
      <Insert ref="search3.enumerator.type"/>
      <Insert ref="search3.enumerator.type.change"/>
    </List>
  </ElementGroup>
</Context>
```
2. Configure enumerator to have one of three values.

```xml
<Enumerator id="search3.enumerator.type"
   <Insert ref="search3.enumerator.type.part" value="part"/>
   <Insert ref="search3.enumerator.type.doc" value="doc"/>
   <Insert ref="search3.enumerator.type.folder" value="folder"/>
   <On event="executeModelDoer" at="end">
      <Copy from="%[frame#com.ptc.search.type_param]"
             to="%[enum#.]"/>
   </On>
</Enumerator>
```

3. Configure labels for enumerator’s values

```xml
<LabelArea id="search3.enumerator.type.part" labelType="text">
   <Label resource="Part"/>
   <ToolTip resource="Part"/>
</LabelArea>

<LabelArea id="search3.enumerator.type.doc" labelType="text">
   <Label resource="Document"/>
   <ToolTip resource="Document"/>
</LabelArea>

<LabelArea id="search3.enumerator.type.folder" labelType="text">
   <Label resource="Folder"/>
   <ToolTip resource="Folder"/>
</LabelArea>
```

4. Configure in-place action the user has to trigger when value of the enumerator changes.

```xml
<Action id="search3.enumerator.type.change">
   <Label resource="Change Type"/>
   <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
   <ClientDoer class="com.ptc.core.ca.co.client.doer.CoNoOpActionDoer"/>
   <On event="executeActionDoer" at="end">
      <Copy from="%[enum#search3.enumerator.type]"
             to="%[frame#com.ptc.search.type_param]"/>
   </On>
</Action>
```

When the specified doer is executed, it finds an enumerator element specified by the element attribute and assigns its current value to a frame parameter with the name specified via name attribute.

5. Configure a table the user will be able to populate with final results. It must have no-op doer.
6. Configure a frame and configure its content to conditionally include one of the search layouts. Make sure you specify initial value for the frame property.

```xml
<SimpleFrame id="search3.all">
    <Extend ref="common:simpleframe.template"/>
    <On event="executeModelDoer" at="end">
        <Copy to="%[frame#com.ptc.search.type_param]" from="part"/>
    </On>
    <Label resource="Search"/>
    <SimpleLayout>
        <LayoutRow id="tc">
            <Insert ref="search3.type_chooser"/>
        </LayoutRow>
        <LayoutRow id="s">
            <Insert ref="common:separator.horizontal"/>
        </LayoutRow>
        <LayoutRow id="p">
            <Insert ref="part:search3.layout.part"/>
            <DrawHandler class="com.ptc.core.ca.co.client.element.CoDrawController">
                <Check value="%[frame#com.ptc.search.type_param]==part"/>
            </DrawHandler>
        </LayoutRow>
        <LayoutRow id="d">
            <Insert ref="doc:search3.layout.doc"/>
            <DrawHandler class="com.ptc.core.ca.co.client.element.CoDrawController">
                <Check value="%[frame#com.ptc.search.type_param]==doc"/>
            </DrawHandler>
        </LayoutRow>
        <LayoutRow id="f">
            <Insert ref="folder:search3.layout.folder"/>
            <DrawHandler class="com.ptc.core.ca.co.client.element.CoDrawController">
                <Check value="%[frame#com.ptc.search.type_param]==folder"/>
            </DrawHandler>
        </LayoutRow>
        <LayoutRow id="t">
            <Insert id="search3.table2" ref="com.ptc.folder:table" type="%[Folder]">
                <ModelDoer class="com.ptc.core.ca.co.client.doer.CoNoOpModelDoer"/>
            </Insert>
        </LayoutRow>
    </SimpleLayout>
</SimpleFrame>
```
7. Configure search layouts for every type

```xml
<SimpleLayout id="search3.layout.part">
  <LayoutRow id="p1">
    <Insert ref="search3.panel"/>
  </LayoutRow>
  <LayoutRow id="p2">
    <Insert ref="common:separator.horizontal"/>
  </LayoutRow>
  <LayoutRow id="p3">
    <Insert ref="search.table.part"/>
  </LayoutRow>
  <LayoutRow id="p4">
    <Insert ref="search3.move"/>
  </LayoutRow>
</SimpleLayout>
```

8. Configure search panel for every layout (see previous example)

9. Configure search result table (see previous example)

10. Configure action lists to let user select and move items to the final list (not shown)

**Connector-Based Implementation (recommended)**

Connector-based implementation (Search V in the demo) is based on the ability to dynamically compute context based on the value of the type selector enumerator and to locate and include the appropriate search panel and search result table.

1. Configure a group of elements containing a label, enumerator, and an action.

   [config_search5.xml]

   ```xml
   <Context>
     <ElementGroup id="search5.type_chooser">
       <List>
         <Insert ref="search5.enumerator.type.label"/>
         <Insert ref="search5.enumerator.type"/>
         <Insert ref="search5.enumerator.type.change"/>
       </List>
     </ElementGroup>
   </Context>
   <LabelArea id="search5.enumerator.type.label" labelType="text">
     <Label resource="Searching for:"/>
     <ToolTip resource="Type"/>
   </LabelArea>
   ```

2. Configure enumerator to have one of three values. The values must object types.

   ```xml
   <Enumerator id="search5.enumerator.type">
     <Insert ref="search5.enumerator.type.part" value="%[doc:WTDocument]"/>
     <Insert ref="search5.enumerator.type.doc"/>
   </Enumerator>
   ```
3. Configure labels for enumerator’s values

```xml
<LabelArea id="search3.enumerator.type.part" labelType="text">
  <Label resource="Part"/>
  <ToolTip resource="Part"/>
</LabelArea>
```

```xml
<LabelArea id="search3.enumerator.type.doc" labelType="text">
  <Label resource="Document"/>
  <ToolTip resource="Document"/>
</LabelArea>
```

```xml
<LabelArea id="search3.enumerator.type.folder" labelType="text">
  <Label resource="Folder"/>
  <ToolTip resource="Folder"/>
</LabelArea>
```

4. Configure in-place action the user has to trigger when value of the enumerator changes.

```xml
<Action id="search5.enumerator.type.change">
  <Label resource="Change Type"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.
  WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.doer.
  CoNoOpActionDoer"/>
</Action>
```

5. Configure a frame with a layout to include the type selector and a connector. The connector will include the search configuration fragment containing search panel and search result table located in the context with the type dimension corresponding to the current value of the enumerator.

```xml
<SimpleFrame id="search5.all">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Search"/>
  <Layout>
    <LayoutRow id="rowNav">
      <Insert ref="navigation:navbar2" colspan="2"/>
    </LayoutRow>
    <LayoutRow id="tc">
      <Insert ref="search5.type_chooser"/>
    </LayoutRow>
    <LayoutRow id="s">
      <Insert ref="common:separator.horizontal"/>
    </LayoutRow>
    <LayoutRow id="p">
      <Connector ref=":search5.layout">
        <NewContext>
          <Copy from="[%enum#search5.enumerator.type]" to="[%context.type]"/>
        </NewContext>
      </Connector>
    </LayoutRow>
  </Layout>
</SimpleFrame>
```
6. Defile configuration fragments containing search panels and search result tables in the appropriate contexts

[part_search2.xml]

<Context type="%[WTPart]">
  <Layout id="search5.layout">
    ...
  </Layout>
</Context>

[doc_search2.xml]

<Context type="%[WTDocument]">
  <Layout id="search5.layout">
    ...
  </Layout>
</Context>

7. Sometimes there is a need to reset input elements to their default values. For instance, it might be required to provide a **Clear** button to be able to reset search criteria to their default values. The simplest way to accomplish the task is to discard a connected element so that the system will re-create it from scratch during subsequent drawing.

DCA provides a doer that navigates an ancestor hierarchy starting from an action it’s registered for, locates a connector element, and disconnects and discards an element connected by it. To trigger an execution of the doer, an in-place action needs to be configured.

<Action id="action.clearConnected">
  <Extend ref="common:action.inPlace.template"/>
  <ToolTip resource="Clear"/>
  <Label resource="Clear"/>
  <ActionDoer class="com.ptc.core.ca.co.client.primitive.
  CoRemoveConnectedActionDoer"/>
</Action>

When the action is used as part of the configuration of a search property panel, upon the execution of a doer it needs to discard a set of frame parameters responsible for where clause and paging.

<PropertyPanel id="search2.panel" type="%[WTDocument]"
  mode="search">
  <Insert ref="search:search.panel.row.property.name"/>
  <Insert ref="search:search.panel.row.property.number"/>
Search With Complex Criteria

You can configure search functionality to let the user specify various search criteria like less than, more than, for example. The DCA demo provides the Search IV page that demonstrates this functionality. This page is available from the first page’s navigation bar, as shown in the following figure.

The Search IV example is based on the Search II example. This search implementation offers the flexibility of choosing the binary relation between an attribute value and the input value. The earlier implementations assumes '=' (equal) as the implied relation only.

1. Configure frame that includes an enumerator for a type selector and conditionally drawn search panels and tables.

[config_search4.xml]

```xml
<SimpleFrame id="search4.all">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Search"/>
  <SimpleLayout>
    <LayoutRow id="tc">
      <Insert ref="search2.type_chooser"/>
    </LayoutRow>
  </SimpleLayout>
</SimpleFrame>
```
2. Configure a layout that includes a search panel and a result table:

```xml
<SimpleLayout id="search4.layout.doc">
  <LayoutRow id="p1">
    <Insert ref="search4.panel.doc"/>
  </LayoutRow>
  <LayoutRow id="p2">
    <Insert ref="common:separator.horizontal"/>
  </LayoutRow>
  <LayoutRow id="p3">
    <Insert ref="doc:search.table.doc"/>
  </LayoutRow>
  <LayoutRow id="p4">
    <Insert ref=":search2.move"/>
  </LayoutRow>
</SimpleLayout>
```

3. Configure a layout that includes search criteria:

```xml
<SimplePropertyPanel id="search4.panel.doc" type="%[doc:WTDocument]" mode="search">
</SimplePropertyPanel>
```
4. Configure search criteria with the relation enumerator. To do that include a label, a relation enumerator, and a property. The property should not have a label. The following is a configuration for the name property of a part.

```xml
<LayoutRow id="search4.panel.row.property.name">
  <LabelArea labelType="text">
    <Label resource="Name"/>
  </LabelArea>
  <Insert ref="search4.enumerator.operation"/>
  <Property>
    <Extend ref=":property.String.template"/>
    <Need attribute="name"/>
  </Property>
</LayoutRow>
```

5. Configure enumerators:

```xml
<Enumerator id="search4.enumerator.operation">
  <Insert ref="search4.enumerator.operation.eq" value="=" default="true"/>
  <Insert ref="search4.enumerator.operation.ne" value="!="/>
  <Insert ref="search4.enumerator.operation.lt" value="&lt;"/>
  <Insert ref="search4.enumerator.operation.le" value="&lt;="/>
  <Insert ref="search4.enumerator.operation.gt" value="&gt;"/>
  <Insert ref="search4.enumerator.operation.ge" value="&gt;=""/>
</Enumerator>
```

6. Configure enumerator’s labels:

```xml
<LabelArea id="search4.enumerator.operation.eq" labelType="text">
  <Label resource="="/>
  <ToolTip resource="EQUAL operation"/>
</LabelArea>
<LabelArea id="search4.enumerator.operation.ne" labelType="text">
  <Label resource="!="/>
  <ToolTip resource="NON EQUAL operation"/>
</LabelArea>
...
```

For the demo, the document and part property panel use Create Stamp attribute. For search purposes, this datetime property is shown in dateOnly, short form:

```xml
<LayoutRow id="search4.panel.row.property.createStamp">
  <LabelArea labelType="text">
    <Label resource="Create Stamp"/>
  </LabelArea>
</LayoutRow>
```
When a datetime property is shown in the dateOnly form, the time part is set to 00:00:00.0 AM. That means that using equals and not equal relation might not give the result that a user expects. Even though the visible date parts are identical, the properties might differ in the invisible time part. The good practice is not to use those two relations when a property is in the dateOnly form.

**Note:** The immediate label before the field is not needed in this case. The value attribute of theme="label" options are set to "" (empty string), since the Option tags are mandatory in the current datetime property implementation.

The current Info*Engine Query-Objects webject implementation supports more complex attribute forms inside where clause task parameter. The Creator property demonstrates that feature (see Need tag):

```
<LayoutRow id="search4.panel.row.property.creator">
  <LabelArea labelType="text">
    <Label resource="Creator"/>
  </LabelArea>
  <Insert ref="search4.enumerator.operation"/>
  <Property>
    <Extend ref="common:property.String.template"/>
    <Need attribute="iterationInfo.creator\name"/>
  </Property>
</LayoutRow>
```

**Caution:** The relation enumerator element must be a descendent of the parent of the property element. When no enumerator element is found, the equal (=) relation is assumed. Also, when there are more then one enumerator elements, the search behavior is unpredictable.

**Caution:** The solution developers are responsible to use relations supported by the Info*Engine query webjects, since there is no internal validation of the configured relation enumerator values.
Search Term Producer

Sometimes more than one attribute needs to participate in the creation of a single search term, a single attribute participates in the creation of multiple search terms, or multiple attributes produce a set of search terms using the rules different from the default ones.

To address such cases, DCA introduced the mechanism to contribute to the creation of a where clause via so-called search term producer. The producer is a class that implements a single method of returning a string containing a single term or a combination of multiple terms.

When a producer is specified for a DCA element, it will be used to contribute to the creation of a where clause no matter what mode the element is in. Otherwise, the default mechanism described in the introduction will be utilized.

DCA demo contains an example of the use of a search term producer. The example demonstrates how to configure a user interface to allow the user to search for objects created on a certain date, during the specified period of time, or within the last day, week, month, and etc.

The following presents a configured fragment:

```xml
[elementGroup id="search.panel.row.group.createStamp"]
  [layoutRow id="lr.0"]
    [labelArea labelType="text" colspan="3"]
      [label resource="Created:"]
    [/labelArea]
    [insert ref="createStamp.option"]
    [insert ref="createStamp.value.onDate"]
[/elementGroup]
```

The following presents a configured fragment:

```xml
[elementGroup id="search.panel.row.group.createStamp"]
  [layoutRow id="lr.0"]
    [labelArea labelType="text" colspan="3"]
      [label resource="Created:"]
    [/labelArea]
    [insert ref="createStamp.option"]
    [insert ref="createStamp.value.onDate"]
[/elementGroup]
```
All property elements included above participate in the creation of a user interface providing that their search term provides lets them do so.

The configuration above demonstrates how to configure a search for an object created on a specified date. The configuration below demonstrates how to configure search for an object created within the specified period of time.
In both examples above the search term producer utilizes the value of a relation attribute to properly construct a search term. The legitimate values are EQ (equals), NE (non equal), LT (less then), LE (less then or equal), GT (greater then), and GE (greater then or equal). The default value is EQ.

The next example demonstrates the use of a search term producer with an enumerator. An enumerator does not participate in a creation of a where clause unless it has a search term producer.

In all of the examples above NPAs must be created by the functions specified as the task output parameters.
Since only a single option to specify date-related criteria should be available for the user at a time, an enumerator needs to be configured to present a list of choices:

```xml
<Enumerator id="createStamp.option">
    <LabelArea id="la.0" labelType="text" value="On Date" default="true">
        <Label resource="On Date"/>
    </LabelArea>
    <LabelArea id="la.1" labelType="text" value="Range">
        <Label resource="Range"/>
    </LabelArea>
    <LabelArea id="la.2" labelType="text" value="Since">
        <Label resource="Since"/>
    </LabelArea>
    <Renderer class="com.ptc.core.ca.web.client.primitive.WebEnumeratorRenderHandler">
        <Option theme="widget" param="type" value="radiobutton"/>
        <Option theme="widget" param="vertical" value="true"/>
        <Option theme="widget" param="labelFirst" value="true"/>
        <Option theme="widget" param="shadow" value="true"/>
    </Renderer>
</Enumerator>
```

Finally, the group of elements related to date search criteria need to be included in the property panel responsible for the creation of a where clause.

```xml
<PropertyPanel id="search.panel.doc" type="%[WTDocument]" mode="search">
    <Insert ref="search:search.panel.row.type.chooser.link" />
    <Insert ref="search:search.panel.row.property.name"/>
    <Insert ref="search:search.panel.row.property.number"/>
    <Insert ref="search:search.panel.row.dde.lifecycle"/>
    <Insert ref="search:search.panel.row.group.createStamp"/>
    <LayoutRow id="rowSearchAction">
        <Insert ref="search:search.action.alt" context.tableId="search.table.doc"/>
    </LayoutRow>
</PropertyPanel>
```
Search With IBAs

A search functionality can be configured to let the user to specify IBA values to form a search criteria (where clause). DCA demo provides Search VI page that demonstrates this capabilities. The page is available from under Search tab.

Rationale

Every object type in Windchill can have a unique set of IBAs associated with it via Type Manager. Thus to enable the use of IBAs as part of a search criteria, the object type needs to be selected first. For the selected object type, DCA can retrieve the description of a set of IBAs applicable to it.

Since IBAs can be defined via Type Manager at any time, DCA suggests that for a given object type a search criteria property panel contains a set of configured properties or data driven enumerators in search mode and a table containing a list of IBAs. The first part of the panel needs to be explicitly configured by an application developer to present an intuitive and easy to use user interface. The table contained in the second part of the panel presents the rest of IBAs that are not part of the first part. Thus the user can benefit from using a manually configured (intuitive and easy to use) portion of the search criteria property panel and automatically generated second portion of it.

Type Chooser

DCA demo presents an example of a data-driven enumerator-based type chooser. The enumerator uses the model doer to execute a task that in turn executed a "Query-Type-List" webject. This webject returns a list of all subtypes for a given type in a form of a list of Info*Engine elements where every element represents a single type.

```xml
<DataDrivenEnumerator id="search6.enumerator.type">
  <NeedNpa attribute="class" as="value"/>
  <ModelDoer class="%[common:DDEModelTaskDoer]"
    labelAttribute="type"
    valueAttribute="class">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
      taskDelegate="dca-query-type-list">
      <TaskParameter name="root_type"
        value="theType=wt.part.WTPart:type=
        Part:class=wt.part.WTPart:obid="/>
      <TaskParameter name="root_type"
        value="theType=wt.doc.WTDocument:type=
        Document:class=wt.doc.WTDocument:obid="/>
      <TaskParameter function="com.ptc.core.ca.co.client.doer.
        function.impl.CoFunction_Model_Param_FudgeElement"/>
    </Task>
  </ModelDoer>
</DataDrivenEnumerator>
```
After choosing the desired type, the user triggers an action that remembers this type as a frame parameter and marks a table of IBAs as dirty.

```xml
<Action id="search6.enumerator.type.change">
  <Label resource="Change Type"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.doer.CoNoOpActionDoer"/>
  <On event="executeActionDoer" at="end">
    <Copy from="%[attribute#search6.enumerator.type.panel#class]" to="%frame#com.ptc.search.type_param"/>
    <Copy from="true" to="%bean#tableForAnyAttr#Dirty"/>
  </On>
</Action>
```

**Search Criteria Property Panel**

The search criteria property panel can contain properties in search mode and data-driven enumerators positioned using the standard layout capabilities. It also contains a table of IBAs. It is recommended to use `Connector` to include type specific search criteria property panel into the overall layout of a search frame.

```xml
<LayoutRow id="tt2">
  <Connector ref="search6.query.parameters.panel">
    <NewContext>
      <Copy from="%[enum#search6.enumerator.type]" to="%context.type"/>
    </NewContext>
  </Connector>
</LayoutRow>
```

** Explicitly Configured Attributes**

Each property or data-driven enumerator contained by the property panel is bound to an Info*Engine element of the property panel via `Need` statement that specifies an attribute’s name.

```xml
<Need attribute="xxx"/>
```

We will call a collection of such attributes as a set of explicitly configured attributes.

**Table of IBAs**

The table of IBAs is capable of presenting all attributes for a given object type. However, to avoid duplication the table of IBAs must not include explicitly configured attributes. To achieve that a special model doer must be used.

The table of IBAs is configured to have two columns `name` and `value`. Column `name` presents the name of an attribute. Column `value` presents the value of the
attribute in search mode to allow the user to specify a search criteria for this attribute.

It is recommended that a single definition of a table containing IBAs is shared across all applicable use cases. In the future, DCA is planning to provide different implementations of the functionality that queries Windchill about the attributes related to a particular type of an object. This is to better support type-specific, classification-based, and other IBAs.

The following is the configuration for the table containing IBAs.

```xml
<SimpleTable id="tableForAnyAttr" mode="search">
  <Type type="DCA|CLASSIFICATION|Property|String"/>
  <Type type="DCA|CLASSIFICATION|Property|Boolean"/>
  <Type type="DCA|CLASSIFICATION|Property|Integer"/>
  <Type type="DCA|CLASSIFICATION|Property|Float"/>
  <Type type="DCA|CLASSIFICATION|Property|Enumerated"/>

  <SimpleColumn show="#name">
    <Label resource="Name"/>
  </SimpleColumn>
  <SimpleColumn show="#value">
    <Label resource="Value"/>
  </SimpleColumn>
</SimpleTable>
```

The model doer `CoTableAllAttributesForGivenTypeModelDoer` does the following:

1. Reads the type from the frame parameter `com.ptc.search.type_param`. Using Info*Engine query schema functionality requests a list (A) of all attributes for the given type.

2. Computes the list (B) of explicitly configured attributes using configuration for the property panel.

3. Removes content of the list B from the list A. For every attribute left in the list A creates Info*Engine element that represents a single attribute. Every such an element will have two attributes - `name` and `value`.

   Attribute `name` has a type `java.lang.String`. It is presented via a string property bound to it.

   Attribute `value` has an unusual looking type as follows:

   ```xml
   <Type type="DCA|CLASSIFICATION|Property|String"/>
   ```

   Remember, DCA gets the information about the attributes using Info*Engine query functionality. It is entirely possible that two attributes that have their values of the same `java.lang.String` are actually very different attributes from the point of view of classification and must be presented
differently. A hypothetical example would be an attribute contain a value "true" as a string. It can be classified as a boolean attribute or a string attribute.

The unusual looking types represent DCA classification attributes. This classification provides the means to bind a class of attributes to a property. For every type defined by the classification there is a correspondent Property element configured to present an attribute’s value.

```xml
<Context type="DCA|CLASSIFICATION|Property|String">
  <Property id="value">
    <Extend ref="common:property.String.template" />
    <Label resource="Value:"/>
    <Need attribute="value"/>
  </Property>
</Context>

<Context type="DCA|CLASSIFICATION|Property|Boolean">
  <Property id="value">
    <Label resource="Value:"/>
    <Need attribute="value"/>
    <PropertyHandler class="com.ptc.core.ca.web.client.property.binary.WebBinaryPropertyHandler"/>
    <Insert ref="common:renderer.binaryDropdownList">
      <Option theme="label" param="false" value="false"/>
      <Option theme="label" param="true" value="true"/>
    </Insert>
  </Property>
</Context>

<Context type="DCA|CLASSIFICATION|Property|TimeStamp">
  <Property id="value">
    <Extend ref="common:property.persistableInfoTimestamp.template"/>
    <Label resource="Value:"/>
    <Need attribute="value"/>
  </Property>
</Context>
```

**Search With Paging**

DCA lets application developers configure search in such a way that only a fraction of a result set is presented at a time. This allows the applications to present a large result set in chunks providing significant performance and usability benefits.

**Model doer with paging**

Page-mode operation is initiated by calling the Query-Objects webject with a PAGE_OFFSET and a PAGE_COUNT parameter in addition to the usual TYPE and WHERE parameters. Upon the first query of the database a paging session will be created. The unique identifier for this paging session will be returned as a metadata item of the output group. This paging session identifier can then be specified as the value of a PAGING_SESSION_ID parameter in subsequent calls.
to the webect. This will cause the webect to apply the PAGE_OFFSET and PAGE_COUNT parameters to the previously created paging session. As the result the set of objects identified by PAGE_OFFSET and PAGE_COUNT are retrieved from the result set that was previously produced and associated with the paging session.

DCA provides a set of task parameter functions that take care of paging related parameters. The following is a typical example of their configuration:

```xml
<ModelDoer class="%[common:SimpleModelTaskDoer]">
  <Task
    nextOperation="STDOP|com.ptc.windchill.view"
    taskDelegate="dca-query-objects-with-sort-and-paging">
    ...
    <Insert ref="search:eg.task.param.sort-n-page"/>
  </Task>
</ModelDoer>

<ElementGroup id="eg.task.param.sort-n-page">
  <TaskParameter function="%[ModelTaskParamWhereClause]"/>
  <TaskParameter function="%[ModelTaskParamPageCount]"/>
  <TaskParameter function="%[ModelTaskParamPageOffset]"/>
  <TaskParameter function="%[ModelTaskParamPagingSessionId]"/>
  <TaskParameter function="%[ModelTaskParamSortBy]"/>
  <TaskParameter function="%[ModelTaskParamSorted]"/>
  <TaskOutParameter function="%[ModelTaskResultPagingParameters]"/>
</ElementGroup>
```

Paging and sorting parameters could be modified via user preferences.

When paging involves sorting, it's recommended to disable client-side sorting that a table employs by default. Otherwise, a table attempts to sort every individual page, which is completely unnecessary.

```xml
<CompositeTable clientSideSortable="false"/>
```

### Paging actions

DCA demo provides a set of actions to support paging. Those actions are configured to be descendents of a table footer.

#### Example 1

<table>
<thead>
<tr>
<th>City</th>
<th>ID</th>
<th>Date/Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cleveland</td>
<td>15</td>
<td>12-25-02 05:37:14 PM</td>
</tr>
<tr>
<td>SunspotCA</td>
<td>14</td>
<td>c/i</td>
</tr>
<tr>
<td>TornadoCA</td>
<td>15</td>
<td>c/i</td>
</tr>
</tbody>
</table>

<< First  < Prev  Page 1 of 7  Next >>
The layout above is used as a composite table footer in the com.ptc.doc:page.search.doc frame:

```xml
<Layout id="search.page.actions.alt">
  <Insert ref="search.firstPage"/>
  <Insert ref="search.prevPage.alt"/>
  <Insert ref="space"/>
  <Action id="page">
    <Label resource="Page "/>
    <ActionHandler class="%[PageInPlaceActionHandler]"/>
    <ClientDoer class="%[PageActionClientDoer]"/>
  </Action>
  <Action id="page_number">
    <Label resource="\{0\}">
      <NeedNpa attribute="page_number"/>
    </Label>
    <ActionHandler class="%[PageInPlaceActionHandler]"/>
    <ClientDoer class="com.ptc.core.ca.web.client.search.
                     WebPageActionClientDoer"/>
    <Renderer class="com.ptc.core.ca.web.client.action.
                     WebTextFieldActionRenderHandler"/>
  </Action>
  <Action id="pages">
    <Label resource="of \{0\}"/>
    <NeedNpa attribute="total_pages"/>
    <Label>
      <ActionHandler class="%[PageInPlaceActionHandler]"/>
      <ClientDoer class="%[PageActionClientDoer]"/>
    </Label>
    <Insert ref="space"/>
    <Insert ref="search.nextPage.alt"/>
    <Insert ref="search.lastPage"/>
  </Action>
</Layout>
```

The actions page and pages use dynamic labels. Their visibility and displayed values depend on the number of table items, page size and the current page index. We had to use Action elements instead of LabelArea elements as the latter does not provide a means to control visibility.

Paging actions that navigate some number of pages from the current page should use the doer com.ptc.core.ca.co.client.search.CoPageActionDoer, or its subclasses. The doer configuration requires the attribute pageOffset that
defines a relative offset of the next page index from the current page index. The configuration of the most common paging actions that navigates to the previous and next page is shown below:

[com/ptc/search/config_search.xml]

<Action id="search.prevPage">
  <Label resource="&lt; Prev {0} ">
    <NeedNpa attribute="prev_count"/>
  </Label>
  <ActionHandler class="com.ptc.core.ca.co.client.search.
    CoPageInPlaceActionHandler"/>
  <ActionDoer class="%[PageActionDoer]" pageOffset="-1"/>
</Action>

<Action id="search.nextPage">
  <Label resource="Next {0} &gt;">
    <NeedNpa attribute="next_count"/>
  </Label>
  <ActionHandler class="com.ptc.core.ca.co.client.search.
    CoPageInPlaceActionHandler"/>
  <ActionDoer class="%[PageActionDoer]" pageOffset="1"/>
</Action>

**Note:** Paging action doers for navigating to the previous and next page that were developed for 6.2.6 are deprecated and can be replaced by the configuration shown above

<Action id="search.prevPage.alt">
  <Label resource="&lt; Prev"/>
  <ActionHandler class="%[PageInPlaceActionHandler]"/>
  <ClientDoer class="com.ptc.core.ca.co.client.search.
    CoPageBackwardActionDoer"/>
</Action>

<Action id="search.nextPage.alt">
  <Label resource="Next"/>
  <ActionHandler class="%[PageInPlaceActionHandler]"/>
  <ClientDoer class="com.ptc.core.ca.co.client.search.
    CoPageForwardActionDoer"/>
</Action>

Even though the pageOffset attribute expects an integer value, there are two special values it can take. The value first makes the action navigate to the first page, and the value last makes the action navigate to the last page:

<Action id="search.firstPage">
  <Label resource="&lt;&lt; First"/>
  <ActionHandler class="%[PageInPlaceActionHandler]"/>
  <ActionDoer class="%[PageActionDoer]" pageOffset="first"/>
</Action>

<Action id="search.lastPage">
  <Label resource="Last &gt;&gt;"/>
  <ActionHandler class="%[PageInPlaceActionHandler]"/>
  <ActionDoer class="%[PageActionDoer]" pageOffset="last"/>
</Action>
Example 2

This layout is used as a simple table footer in the com.ptc.part:page.search.part frame

[com/ptc/part/part_search.xml]

<Layout id="page.actions.footer">
  <LayoutRow id="0">
    <Insert ref="search:search.result.range"/>
    <List>
      <Insert ref="search:search.firstPage">
        <Location ref="common:frame.progress.monitor"/>
        <ActionHandler class="com.ptc.core.ca.co.client.search.
          CoPageInPlaceActionHandler"
          progressMonitor="true"/>
      </Insert>
      <Insert ref="search:search.prevPage">
        <Location ref="common:frame.progress.monitor"/>
        <ActionHandler class="com.ptc.core.ca.co.client.search.
          CoPageInPlaceActionHandler"
          progressMonitor="true"/>
      </Insert>
      <Insert ref="search:field.separator"/>
      <Insert ref="search:search.nextPage">
        <Location ref="common:frame.progress.monitor"/>
        <ActionHandler class="com.ptc.core.ca.co.client.search.
          CoPageInPlaceActionHandler"
          progressMonitor="true"/>
      </Insert>
      <Insert ref="search:search.lastPage">
        <Location ref="common:frame.progress.monitor"/>
        <ActionHandler class="com.ptc.core.ca.co.client.search.
          CoPageInPlaceActionHandler"
          progressMonitor="true"/>
      </Insert>
    </List>
  </LayoutRow>
</Layout>

Two of the actions in the previous footer are used instead of labels, since they can be disabled or enabled depending on the table status, and the displayed value is dynamically changed depending on the total table row number, page size and so on. If the table is empty, no element is rendered in the footer.

[com/ptc/search/config_search.xml]

<Action id="search.result.range">
  <Label resource="Results: {0} - {1} of {2}"/>
Example 3

The last example is the paging actions used as a simple table footer in the com.ptc.folder:page.search.folder frame:

`<com/ptc/search/config_search.xml>`

```xml
<Layout id="search.page.select.actions.layout">
  <LayoutRow id="r1">
    <Insert ref="search.prevPage.alt"/>
    <LabelArea>
      <Label resource="Pages:"/>
    </LabelArea>
    <Insert ref="search.page.select.actions"/>
    <Insert ref="search.nextPage.alt"/>
  </LayoutRow>
</Layout>

<ActionList id="search.page.select.actions" actionListType="horizontal">
  <Action id="1">
    <Label resource="1"/>
    <Extend ref="search.selectPage"/>
  </Action>
  <Action id="2">
    <Label resource="2"/>
    <Extend ref="search.selectPage"/>
  </Action>
  <Action id="3">
    <Label resource="3"/>
    <Extend ref="search.selectPage"/>
  </Action>
</ActionList>

<Action id="search.selectPage">
  <ActionHandler class="com.ptc.core.ca.co.client.search.
```
The action list displays three actions that dynamically adjust themselves to display and navigate to some page range around the current page. The action list can contain any number of actions.

**Multiple Searches**

Occasionally there is a need to have multiple search property panels and search result tables in the same frame. It can be easily accomplished with a small re-configuration of the existing search functionality.

Each property panel and search result table uses frame parameters with predefined names. In order to have more than one search in a frame, search actions must be configured to save parameters under unique names, and search result tables must be configured to retrieve parameters using those names.

**Note:** In order to have multiple search result tables each table must have unique IDs.

DCA demo provides an example of having multiple searches in the same frame in `com.ptc.doc: page.search.doc`.

```xml
<CompositeTable id="search.alttable.doc.1">
    <Extend ref="search.alttable.doc"/>
</CompositeTable>
<CompositeTable id="search.alttable.doc.2">
    <Extend ref="search.alttable.doc"/>
</CompositeTable>
```

Search actions must now be configured to prepare frame parameters in a way search tables expect them. The following two search actions extend the regular demo Search action:

```xml
<Insert id="search.action.1" ref="search:search.action.alt"
    context.tableId="search.alttable.doc.1">
    <Label resource="Search 1"/>
</Insert>
<Insert id="search.action.2" ref="search:search.action.alt"
    context.tableId="search.alttable.doc.2">
    <Label resource="Search 2"/>
</Insert>
```

Notice that each action copies a set of relevant frame parameter values from the set of parameters with standard names to the set of parameters with unique names. It then removes parameters with the standard names to avoid affecting other searches.
Each new unique parameter name starts from the table ID.

Search and paging related functionality first look up for the frame parameters with the prefix equals to the ID of the table. Only when such parameters can not be found, it looks up for the parameters with generic names. That allows performing the searching and paging actions, as well as the table sorting and column manipulation on multiple tables.
In DCA, Pickers are components that let the user pick one or more instance holders containing Info*Engine elements from a selector component. A selector component can be either a table or a tree. The Picker framework does not differentiate between multiple ways a selector component can be populated. Thus, the selector component can be browser-based, search-based, or populated some other way. When asked to locate a selector component, picker framework will traverse descendant elements starting from a tree until finds one.

**Caution:** It is important that the picker frame has a single selector component.

**Topic** | **Page**
---|---
Property Picker | 13-2
Object List Picker | 13-5
Property Retriever | 13-10
Property Picker

We will call an Info*Engine element of the selected instance holder a *picked instance*. We will call an Info*Engine element of the instance holder of a property a *target instance*.

Property picker is responsible for the selection of a single instance holder from a selector component. It will use properties of a picked instance to populate one or more attributes of the target instance.

The following is an example of a location property picker:

In this example target and picked instances both represent folders. The purpose of the picker is to initialize location property and present to a user. The following is a configuration snippet from folder/config_folder.xml:

```xml
<SimpleLayout id="folder.layout.create.task" type="%[SubFolder]">
  <LayoutRow id="1">
    <Insert ref="new.property.location"/>
    <Insert ref="new.property.location.picker"/>
  </LayoutRow>
  <LayoutRow id="2">
    <Insert ref="new.property.name"/>
  </LayoutRow>
</SimpleLayout>
```
The property presents the value of an attribute "folder".

<Property id="new.property.location">
  <Extend ref="property.String.template"/>
  <Label resource="Location:"/>
  <Need attribute="folder"/>
  <Label resource="Folder Location"/>
</Property>

We will call the action that brings up a picker dialog a picker action. Here is its configuration:

<Action id="new.property.location.picker">
  <Label resource="Pick Location"/>
  <Location ref=":folder.pick.browse"/>
  <UpdateLocation ref="new.property.location"/>
  <ActionHandler class="com.ptc.core.ca.web.client_picker.WebPickerActionHandler"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/button.jsp"/>
</Action>

The Location parameter of the picker action is the address of a picker frame. The Update location contains ID of the property to update when picker frame is closed.

WebPickerActionHandler opens a new "popup" picker window displaying the picker page specified by the configured location value. The property element with ID specified by UpdateLocation will be the action holder of a picker action.

There are multiple ways of constructing a picker frame. In the example below the frame contains a one step wizard with a model doer that does nothing.

<SimpleFrame id="folder.pick.browse" mode="search"
  width="200" height="300">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Folder Picker"/>
  <Wizard>
    <Label resource="Folder Picker"/>
    <WizardStepElements>
      <Insert ref="folder.pick.browse.table"/>
    </WizardStepElements>
    <ModelDoer class="%common:NullModelDoer"/>
    <ClientDoer class="%common:FolderPropertyPickerOkActionDoer">
      <Populate from="name"/>
    </ClientDoer>
  </Wizard>
</SimpleFrame>

The table implements a rudimentary folder browser.

<SimpleTable id="folder.pick.browse.table" mode="view"
  selectorType="single">
  <Type type="@[Folder]"/>
  <SimpleColumn show="picker.action.folder.view">
    <Label resource="Name"/>
  </SimpleColumn>
</SimpleTable>
In this example, an initial directory is a user cabinet. The function could have been written to take into account other information if needed. While in the picker frame, the user can browse folders using actions provided to navigate to sub-folders.

When the user selects a folder and clicks **Ok**, `FolderPropertyPickerOkActionDoer` is invoked.

```xml
<ClientDoer class="%common:FolderPropertyPickerOkActionDoer">
    <Populate from="name"/>
</ClientDoer>
```

The doer is configured by a short `Populate` statement to populate the first attribute needed by the property specified by `UpdateLocation` in the picker action. More than one `Populate` statement can be specified. In the example above, the `location` attribute of the target instance will be populated with a value of the `name` attribute of the picked instance.

An alternative way of specifying how the attribute values are populated would be to use the long version of the `Populate` statement.

```xml
<Populate attribute="location" from="name"/>
```

Short and long `Populate` statements can be used in the configuration of the same doer. The doer must be:

`com.ptc.core.ca.co.client.picker.CoPropertyPickerOkActionDoer`

or its sub-class as in the example above. A short `Populate` statement will populate an appropriate attribute. Thus short statement #2 will populate the second attribute needed by the property.

A sub-class of `CoPropertyPickerOkActionDoer` can perform additional manipulations with attribute values. The following is an example of computing a location attribute:

```java
public class CoFolderPropertyPickerOkActionDoer extends CoPropertyPickerOkActionDoer {
    public void execute(CoActionHandlerData execData) throws WTException {
        CoPropertyPickerOkActionHandlerData data =
```
{CoPropertyPickerOkActionHandlerData}execData;

ArrayList pairList = getAttrPairList(data);
String[] pair = (String[])pairList.get(0);

Object picked_value = getPickedAttributeValue(pair[1], data);
Object current_value = getTargetAttributeValue(pair[0], data);
setTargetAttributeValue(pair[0],
    current_value + "/" + picked_value,
    data);
addPropertyToIgnore(data.getTargetProperty());
}
}

In this example execute method does not call super.execute to avoid copying of the attribute value but it’s not a requirement.

CoPropertyPickerOkActionDoer provides a service to gain access to attribute values of a target and picked instance.

Another way to implement a picker is to configure picker frame to execute a search to populate a selector in single select mode. This is no different than configuring regular search functionality. CoPropertyPickerOkActionDoer will be able to locate a selector and use selected instance.

**Object List Picker**

We will call Info*Engine elements of the selected instance holders a *picked instances*.

Object list picker is responsible for the selection of a multiple instance holders from a selector component. It will use picked instances to populate the model of a list instance holder CoInstanceListHolder target element. A table is the only element that can hold a list of Info*Engine elements at the moment.

The following is an example of an object list picker
The following is a configuration snippet from part/list_picker.xml:

```xml
<SimpleTable id="table.pick.list.part.target"
    selectorType="multi">
    <SelectAllPrompt resource="All"/>
    <Insert ref="pick.action.list.part"/>
</SimpleTable>
```
The table is using CoNoOpModelDoer as no data acquisition is needed.

Three different actions are configured for the demo. Their purpose is to demonstrate how picked instances can be used to replace existing list, add to it, or merge into it.

Every action has UpdateLocation element containing ID of the list holder element.

When an action is triggered, a new frame is created. It can be configured to contain search or browsing functionality. In the demo it is configured to execute search for parts.
CoObjectListPickerAddOkActionDoer gets executed on Ok and it populates a list of Info*Engine element in table.pick.list.part.target table in the launching frame.

Two concrete doers are provided in com.ptc.core.ca.co.client-picker package:

CoObjectListPickerAddOkActionDoer
CoObjectListPickerReplaceOkActionDoer
Caution: A picker frame must be configured in such a way that every picked instance has enough attributes to be properly presented in an instance list holder in a launching frame.

To remove picked instances Remove Selected or Remove All actions can be configured for the toolbar of a table.

```
<Action id="pick.action.removeAll.list.part">
   <Extend ref="common:action.inPlace.template"/>
   <ToolTip resource="Delete All"/>
   <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/delete.gif"/>
   <Label resource="Delete All"/>
   <ClientDoer class="com.ptc.core.ca.co.client.table.
               CoTableRemoveAllActionDoer"/>
</Action>
<Action id="pick.action.remove.list.part" needSelection="true">
   <Extend ref="common:action.inPlace.template"/>
   <ToolTip resource="Delete"/>
   <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/delete.gif"/>
   <Label resource="Delete"/>
   <ClientDoer class="com.ptc.core.ca.co.client.table.
               CoTableRemoveSelectedActionDoer"/>
</Action>
```
Property Retriever

A property retriever is responsible for the population of an Info*Engine element of a target instance holder by attributes of a retrieved instance. While the retriever frame can be configured to associate an Info*Engine element with the wizard using a variety of different ways, the functionality responsible for the population of a target instance is essentially the same that of a property picker.

DCA demo presents an example of a property retriever implementing folder-creation functionality launched from within a document creation wizard. The goal of the retriever is to associate a newly created folder with the folder creation wizard (retriever frame) and to copy over folder path to the document’s Info*Engine element.

The following is a fragment from the configuration of the document creation wizard:

```
<Layout id="layout.create" type="%[WTDocument]">
  <LayoutRow id="1">
    <Property id="new.property.location">
      <Extend ref="common:property.location.template"/>
      <Label resource="Location"/>
    </Property>
    <Insert ref="folder:new.property.location.picker"/>
    <Insert ref="folder:new.property.location.create"/>
  </LayoutRow>
  <LayoutRow id="2">
    <Insert ref=":number" required="true"/>
    <Insert ref=":name" required="true"/>
  </LayoutRow>
</Layout>
```

The wizard contains an action to launch folder creation (property retriever) frame.

```
<Action id="location-picker.template">
  <UpdateLocation ref="new.property.location"/>
  <ActionHandler class="com.ptc.core.ca.web.client-picker.WebPickerActionHandler"/>
  <Insert ref="common:renderer.actionButton"/>
</Action>

<Action id="new.property.location.create">
  <Extend ref="location-picker.template"/>
  <Label resource="Create Folder"/>
  <ToolTip resource="Create Folder"/>
  <Location ref=":dialog.create-picker"/>
</Action>
```
The following is a configuration of the folder creation (property retriever) frame. The configuration of action doer is of a particular interest.

```xml
<SimpleFrame id="dialog.create.picker" mode="create" width="600" height="300">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Create Folder"/>
  <Wizard>
    <Label resource="Folder Create Wizard"/>
    <WizardStepElements>
      <Insert ref="folder.layout.create.task"/>
    </WizardStepElements>
    <ModelDoer class="%common:SimpleModelTaskDoer">
      <Task nextOperation="STDOP|com.ptc.windchill.create"
            taskDelegate="dca-create-model">
        <TaskParameter name="target_type" value="wt.folder.SubFolder"/>
        <TaskParameter name="dca_field" computedValue="com.ptc.refimpl.co.computedvalues.DefaultLocationFieldComputedValue"/>
        <TaskParameter name="dca_field" value="name=default folder name"/>
      </Task>
    </ModelDoer>
    <ActionDoer class="com.ptc.core.ca.co.client.picker.
                     CoDefaultSimplePropertyPickerOkActionTaskDoer">
      <Task nextOperation="STDOP|com.ptc.windchill.view"
            taskDelegate="dca-create-folder-with-path-attribute">
        <TaskOutParameter function="com.ptc.core.ca.co.client.doer.function.impl.
                                CoFunction_Action_Result_SetInstanceToInstanceHolder"/>
      </Task>
      <Validator class="com.ptc.core.ca.co.client.wizard.
                         CoWizardDialogActionValidator"/>
      <Populate from="path"/>
    </ActionDoer>
  </Wizard>
</SimpleFrame>
```

This action doer is configured to execute a task that creates a folder. Then using the task output parameter function, it assigns the Info*Engine element representing folder to the wizard. Finally, the doer uses Populate statements to copy over path attribute of the folder element.

The dca-create-folder-with-path-attribute task creates a folder and adds the folder's path as the value of an NPA (non-persistent attribute) path to the folder Info*Engine element.

The list of wizard actions needs to be configured to include Ok that triggers the invocation of a doer.
CoPropertyRetrieverOkActionHandler action handler is responsible for the preparation of the date about the source instance in form of an instance of CoPropertyRetrieverOkActionHandlerData class that it passes to the action doer. For that it used the instance of the wizard element.

The default attribute name for the folder's path is path. When another attribute name is required for storing the path, the task parameter dca_path_attribute_name can be used to indicate the task that the attribute with the name equal to the value of the task parameter needs to be use instead. Of course, the attribute name has to match the value of from attribute of the Populate element. The example is shown below:

```xml
<ActionDoer ...>
  <Task ...>
    ...
    <TaskParameter name="dca_path_attribute_name" value="some_attribute_name"/>
    ...
  </Task>
  ...
  <Populate from="some_attribute_name"/>
  ...
</ActionDoer>
```
This section describes rationale behind the placement and layout of the DCA navigation bar.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>14-2</td>
</tr>
<tr>
<td>Rationale</td>
<td>14-3</td>
</tr>
<tr>
<td>Configuration</td>
<td>14-4</td>
</tr>
<tr>
<td>Left-Hand Navigation</td>
<td>14-5</td>
</tr>
</tbody>
</table>
Location

DCA introduces the idea of a logical location in DCA framework. Every DCA frame has a location that is a static part of its address. Location includes the following parameters:

- ID of the configuration feature describing the frame
- type of the Info*Engine element (optional)
- a value of appId dimension (optional)
- a value of a decoration dimension (optional)
- a value of a roleId dimension (optional)

Example

<!--Context-->
<CompositeFrame id="altfirst.page.view" type="%[Folder]">
  ...
</CompositeFrame>

Location of the frame element described by this configuration specification will include ID equal to altfirst.page.view and type equal to %[Folder].

Typically, application developers using DCA are expected to begin from a storyboard. Once the storyboard is completed, every screen will be assigned a logical location. Next, application developers are expected to begin configuring logical locations to present proper information according to the storyboard.

![Diagram showing the relationship between DCA and Building Applications from storyboard to configuration specification.](image-url)
Rationale

The Navigation bar provides an ability to specify a tree of logical locations.

\[
\begin{array}{ccc}
1 & 2 \\
/ & \ \ \\
1.1 & 1.2 & 1.3 & 2.1 & 2.2
\end{array}
\]

Here we have locations 1 and 2 with three and two sub-locations respectively. When a location of a current frame is 1, the user can navigate to 2, 1.1, 1.2, and 1.3. A navigation bar is expected to present two rows of actions:

1
2
1.1 1.2 1.3

[1] 2
Action [1] is disabled as 1 is a current location.

When a location of a current frame is 2, the user can navigate to 1, 2.1, and 2.2. A navigation bar is expected to present two rows of actions:

1 2
2.1 2.2


When a location of a current frame is 1.2, the user can navigate to 1, 2, 1.1, and 1.3. A navigation bar is expected to present two rows of actions:

1 2
1.1 [1.2] 1.3

Action [1.2] is disabled.

The following describes an algorithm the navigation bar is using to present itself:

- find a current node with location equal to a current location in a tree of locations
- find a path from a root node to the current node
- for every node that belongs to the path prepare a list of siblings. A sibling is any child of a node’s parent.
- Display the lists of siblings, one list in a row
Configuration

The Navigation bar is a specially configured layout containing go-to actions to help user navigate between logical locations. See navigation/config_nav.xml

<Action id="navbar.action.1">
  <Extend ref="common:navAction.goTo.template"/>
  <Label resource="Tables and Trees"/>
  <ToolTip resource="Tables and Trees"/>
  <Location ref=":junkyard.page.allTables"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/tab.jsp"/>
</Action>

<Action id="navbar.action.1.1">
  <Extend ref="common:navAction.goTo.template"/>
  <Label resource="Simple Table"/>
  <ToolTip resource="Simple Table"/>
  <Location ref="folder:first.page.view"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/subtab.jsp"/>
</Action>

<SimpleNavBar id="navbar">
  <NavBarRow id="1">
    <NavBarAction id="navbar.item.1">
      <Extend ref="navbar.action.1"/>
      <NavBarRow id="2">
        <NavBarAction id="navbar.item.1.1">
          <Extend ref="navbar.action.1.1"/>
        </NavBarAction>
        <NavBarAction id="navbar.item.1.2">
          <Extend ref="navbar.action.1.2"/>
        </NavBarAction>
        <NavBarAction id="navbar.item.1.3">
          <Extend ref="navbar.action.1.3"/>
        </NavBarAction>
        <Insert ref="panel.search"/>
      </NavBarRow>
    </NavBarAction>
    <NavBarAction id="navbar.item.2">
      <Extend ref="navbar.action.2"/>
      <NavBarRow id="3">
        <NavBarAction id="navbar.item.2.1">
          <Extend ref="navbar.action.2.1"/>
        </NavBarAction>
        <NavBarAction id="navbar.item.2.2">
          <Extend ref="navbar.action.2.2"/>
        </NavBarAction>
      </NavBarRow>
    </NavBarAction>
  </NavBarRow>
</SimpleNavBar>

<Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/menu/tabrow.jsp"/>
<Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/layout/header.jsp"/>
Here the `NavBarRow` element is used to group multiple `NavBarAction` elements to make them siblings. Navigation bar will use `Location` parameters of each `NavBarAction` element to build a tree of available logical location.

To enable navigation bar include it in the definition of a frame.

```xml
<SimpleFrame id="first.page.view" type="%[Folder]">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Folder Explorer"/>
  <ToolTip resource="Folder Explorer"/>
  <SimpleLayout>
    <LayoutRow id="rowNav" ifnot.decoration="portlet">
      <Insert ref="navigation:navbar"/>
    </LayoutRow>
    ...
  </SimpleLayout>
</SimpleFrame>
```

It is permissible to define multiple navigation bars to include in different definitions of different frames.

**Caution:** When the definition of a navigation bar does not contain a current location, it will not be able to present itself as it will not be able to complete the algorithm described above. Make sure there is a configured frame for every location specified in the definition of a navigation bar.

**Caution:** The definition of a navigation bar must not contain multiple instances of actions with the same location. When it does, it will not be able to correctly determine the path to the current node in a tree.

**Caution:** For this release of DCA Web only, the ID of the element is taken into account when comparing locations. Specifying multiple locations with the same ID and different set of dimensions will lead to unpredictable results.

### Left-Hand Navigation

The navigation bar organizes actions in a tree structure. By default, it groups available actions to provide multiple rows of actions. A section of a navigation bar can be configured to present actions in a column. This way a navigation bar can be placed on the left (or right) side of a frame. To view the demo, use the following URL:

```
```

The following example presents how to implement a three-level navigation bar with the top two levels presented as horizontal bars and the third level as a vertical navigation bar on the left side of the frame.
Caution: You will have to place one instance of a navigation bar in the top portion of your frame layout, and another instance on the left portion of it. As the result two instances of navigation bar element will be created to achieve desired effect.

[navigation/config_nav.xml]

<SimpleNavBar id="navbar1">
  <NavBarRow id="1">
    <NavBarAction id="navbar.item.1">
      <Extend ref="navbar.action.1"/>
    </NavBarAction>
    <NavBarRow id="2">
      <NavBarAction id="navbar.item.1">
        <Extend ref="navbar.action.1"/>
      </NavBarAction>
      <NavBarRow id="11">
        <NavBarAction id="navbar.item.1.0">
          <Extend ref="navbar.action.1.0"/>
        </NavBarAction>
        <NavBarAction id="navbar.item.1.0.1">
          <Extend ref="navbar.action.1.0.1"/>
        </NavBarAction>
        <NavBarAction id="navbar.item.1.0.2">
          <Extend ref="navbar.action.1.0.2"/>
        </NavBarAction>
        <Insert ref="level3_navbarrow"/>
      </NavBarRow>
      <Insert ref="level1_navbarrow"/>
    </NavBarAction>
    <NavBarRow id="1">
      <NavBarAction id="navbar.item.1.1">
        <Extend ref="navbar.action.1.1"/>
      </NavBarAction>
      <NavBarAction id="navbar.item.1.2">
        <Extend ref="navbar.action.1.2"/>
      </NavBarAction>
      <Insert ref="level2_navbarrow"/>
    </NavBarRow>
    <Insert ref="header"/>
  </NavBarRow>
</SimpleNavBar>
Here we have a three-level action tree.

```
   1            2
   /  |     \  /  \  
 1.0 1.1  1.2  2.1  2.2  
 /   |   \    
1.0.1 1.0.2 1.0.3
```

In the example above for each level (NavBarRow) we specified a renderer:

```
level1_navbarrow
level2_navbarrow
level3_navbarrow
```

Depending on the context, the actual renderer may be different. The following configuration provides two sets of NavBarRow renderers for two contexts.

```
<Context>
  <Renderer id="level1_navbarrow"
    resource="/wtcore/jsp/com/ptc/core/ca/web/menu/level1_navbarrow.jsp"/>
  <Renderer id="level2_navbarrow"
    resource="/wtcore/jsp/com/ptc/core/ca/web/menu/level2_navbarrow.jsp"/>
  <Renderer id="level3_navbarrow"
    resource="/wtcore/jsp/com/ptc/core/ca/web/menu/empty_navbarrow.jsp"/>
  <Renderer id="header"
    resource="/wtcore/jsp/com/ptc/core/ca/web/layout/header.jsp"/>
</Context>

<Context navbarlocation="left">
  <Renderer id="level1_navbarrow"
    resource="/wtcore/jsp/com/ptc/core/ca/web/menu/empty_navbarrow.jsp"/>
  <Renderer id="level2_navbarrow"
    resource="/wtcore/jsp/com/ptc/core/ca/web/menu/empty_navbarrow.jsp"/>
  <Renderer id="level3_navbarrow"
    resource="/wtcore/jsp/com/ptc/core/ca/web/menu/vertical_navbarrow.jsp"/>
  <Renderer id="header"
    resource="/wtcore/jsp/com/ptc/core/ca/web/layout/vertical_header.jsp"/>
</Context>
```

In the context of navbarlocation="left" the top two levels of actions are not displayed as empty_navbarrow.jsp does not produce any output. The third level of action list is rendered vertically using the vertical_navbarrow.jsp renderer.

In the default only the first two levels are rendered and third level actions are not.

The following configuration shows how this navigation bar configuration is used. The navigation bar is included twice; the first time in a default context, and second time in the context where navbarlocation is equal to left.
Using dimensions and empty renderers you can choose to render any of the levels in the tree either as a vertically or horizontally.
The Wizard Framework is an extension of basic DCA capabilities, where the Wizard element is constructed out of existing elements with little customization.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>15-2</td>
</tr>
<tr>
<td>Wizard Action Handlers</td>
<td>15-7</td>
</tr>
<tr>
<td>Configuring Doers for Wizard</td>
<td>15-10</td>
</tr>
</tbody>
</table>
Introduction

In DCA, the Wizard Framework plays multiple roles. The Wizard Framework:

- maintains an instance of an Info*Engine element
- presents properties of an Info*Engine element
- maintains a list of step elements, (usually layouts) presenting one of them at a time
- maintains a list of step actions to navigate between steps and control their visibility and accessibility
- maintains a list of wizard actions (Prev, Next, Ok, Apply, Cancel) and controls their visibility and accessibility
- refreshes the opener window on a successful Ok or Apply.

The image below is a typical snapshot of a part creation wizard.

The wizard has two steps, with step #1 selected as current. "Prev" action is grayed out as there is no previous step. Step action for Step #1 (on the tab) is also grayed out, as step #1 is the current step.

The following is a configuration fragment for the wizard above:

```xml
<SimpleFrame id="dialog.create"
    width="600" height="400" mode="create">
    <Extend ref="common:simpleframe.template"/>
    <Label resource="Create Part"/>
    <Wizard>...</Wizard>
</SimpleFrame>
```
<Label resource="Create Part"/>

<WizardStepActions>
  <Insert ref="common:action.wizard.goto" id="1">
    <Label resource="Step 1"/>
    <ToolTip resource="Step 1"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/ action/tab.jsp"/>
  </Insert>
  <Insert ref="common:action.wizard.goto" id="2">
    <Label resource="Step 2"/>
    <ToolTip resource="Step 2"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/ action/tab.jsp"/>
  </Insert>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/wizard/ wizardtab.jsp"/>
</WizardStepActions>

<WizardStepElements>
  <Insert ref="layout.create"/>
  <Insert ref="layout.table.uses" mode="view"/>
</WizardStepElements>

<ModelDoer class="%common:SimpleModelTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.create"
    taskDelegate="dca-create-model">
    <TaskParameter name="target_type" value="WCTYPE|wt.part.WTPart"/>
    <TaskParameter name="dca_field" computedValue="com.ptc.refimpl.co. computedvalues.DefaultLocationFieldComputedValue"/>
    <TaskParameter name="dca_field" value="name=default part name"/>
    <TaskParameter name="dca_field" value="number=default part number"/>
    <TaskParameter name="dca_field" value="lifeCycle=Default"/>
  </Task>
</ModelDoer>

<ActionDoer class="%common:SimpleActionTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.view"
    taskDelegate="dca-create-part-action">
    <TaskParameter function="com.ptc.core.ca.co.client. doer.function.impl.CoSimpleFunction_Action_Param_ObjectsAsGroup"
      name="uses" element="table.pick.list.part.target" selected="false"/>
    <TaskParameter name="target_type" value="WCTYPE|wt.part.WTPart"/>
    <TaskParameter name="link_type" value="WCTYPE|wt.part.WTPartUsageLink"/>
  </Task>
</ActionDoer>

<Insert ref="common:wizard.actions"/>
</Wizard>
</SimpleFrame>
WizardStepActions

The wizard is built of the following elements:

```xml
<WizardStepActions>
  <Insert ref="common:action.wizard.goto" id="1">
    <Label resource="Step 1"/>
    <ToolTip resource="Step 1"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
                action/tab.jsp"/>
  </Insert>
  <Insert ref="common:action.wizard.goto" id="2">
    <Label resource="Step 2"/>
    <ToolTip resource="Step 2"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
                action/tab.jsp"/>
  </Insert>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/wizard/
               wizardtab.jsp"/>
</WizardStepActions>
```

Step actions are just regular action elements with a wizard-specific action handler. They can have action or client doer if required. The handler keeps track of a step index in the data model of a wizard element.

```xml
<Action id="action.wizard.goto">
  <ActionHandler class="com.ptc.core.ca.web.client.wizard.
                  WebWizardStepActionHandler"/>
  <!-- This is just to show that we can specify a doer for a step -->
  <ActionDoer class="com.ptc.core.ca.co.client.wizard.
                 CoWizardStepActionDoer"/>
</Action>
```

WizardStepElements

```xml
<WizardStepElements>
  <Insert ref="layout.create"/>
  <Insert ref="layout.table.uses" mode="view"/>
</WizardStepElements>
```

This is just a sequence of configuration elements - one per each step of the wizard. Depending on the current step, an appropriate configuration element will be used to display the content of the step.

A step will usually contain a layout containing properties, tables, or trees. The same property can be presented on different steps, as DCA will create different DCA elements for each occurrence. They will, however, share the same set of attributes stored in an Info*Engine element controlled by the Wizard.

ModelDoer

ModelDoer is just a regular model doer.
**ActionDoer**

ActionDoer is just a regular action doer.

**WizardActions**

WizardActions are a collection of pre-configured actions.

```xml
<WizardActions id="wizard.actions">
    <Insert ref="action.wizard.previous"/>
    <Insert ref="action.wizard.next"/>
    <Insert ref="action.wizard.ok"/>
    <Insert ref="action.wizard.apply"/>
    <Insert ref="action.wizard.cancel"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/horizontallist.jsp"/>
</WizardActions>

<Action id="action.wizard.previous">
    <Label resource="Prev"/>
    <ToolTip resource="Prev"/>
    <ActionHandler class="com.ptc.core.ca.web.client.wizard.WebPreviousStepActionHandler"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/wizard/wizard_next_prev.jsp"/>
</Action>

<Action id="action.wizard.next">
    <Label resource="Next"/>
    <ToolTip resource="Next"/>
    <ActionHandler class="com.ptc.core.ca.web.client.wizard.WebNextStepActionHandler"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/wizard/wizard_next_prev.jsp"/>
</Action>

<Synonym id="KeyEnter" value="\u000D"/>

<Action id="action.wizard.ok" activationKey=%[KeyEnter]">
    <Label resource="Ok"/>
    <ToolTip resource="Ok"/>
    <ActionHandler class="com.ptc.core.ca.web.client.wizard.WebOkActionHandler"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/wizard/wizard_ok_apply.jsp"/>
</Action>

<Action id="action.wizard.apply">
    <Label resource="Apply"/>
    <ToolTip resource="Apply"/>
    <ActionHandler class="com.ptc.core.ca.web.client.wizard.WebApplyActionHandler"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/wizard/wizard_ok_apply.jsp"/>
</Action>
```
<Action id="action.wizard.cancel">
  <Label resource="Cancel"/>
  <ToolTip resource="Cancel"/>
  <ActionHandler class="com.ptc.core.ca.web.client.wizard.
WebCancelActionHandler"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
wizard/wizard_cancel.jsp"/>
</Action>

When a Wizard has only a single step, Prev and Next actions will not display themselves. Otherwise, they will control their accessibility based on the step index.

Triggering Ok or Apply will cause the invocation of ActionDoer. When successful, the opener window will be refreshed.
Wizard Action Handlers

Package com.ptc.core.ca.web.client.wizard

Wizard action handlers are used in configuring actions for the Wizard element. There are two types of wizard action handlers. Wizard step action handlers and wizard dialog action handlers.

The wizard action handlers are all com.ptc.core.ca.co.client.action.CoSimpleInPlaceAction Handlers. This a wizard element will be an action holder and the only instance holder.
**Wizard Step Action Handlers**

These action handlers are used when a true "wizard" is required to step the user through sets of dialog panels before applying the user changes.

<table>
<thead>
<tr>
<th>Description</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WebWizardStepActionHandler</strong></td>
<td>Executes the configured action doer. Keeps track of the previous and next wizard step actions. Disables itself if current step action. This is to be used as a wizard step action handler.</td>
</tr>
<tr>
<td><strong>WebNextStepActionHandler</strong></td>
<td>Executes the configured step action that is specified after the current wizard step action. Sets the visible state to false and disables the action if none or one step action is configured. This is to be used when standard wizard next behavior is desired.</td>
</tr>
<tr>
<td><strong>WebPreviousStepActionHandler</strong></td>
<td>Executes the configured step action that is specified before the current wizard step action. Sets the visible state to false and disables the action if none or one step action is configured. This is to be used when standard wizard previous behavior is desired.</td>
</tr>
</tbody>
</table>
**Wizard Dialog Action Handlers**

These action handlers are used to apply or cancel user changes in a dialog box.

<table>
<thead>
<tr>
<th><strong>WebOkActionHandler</strong></th>
<th><strong>Description</strong></th>
<th><strong>Use</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Executes ClientDoer or ActionDoer of a Wizard element. When the doer successfully finishes the execution, the wizard frame is closed. Enables the action based on configured doer enabled status. In view mode the action's visible state is set as false.</td>
<td>This is to be used in the configuration of an &quot;Ok&quot; Action element.</td>
</tr>
<tr>
<td><strong>WebApplyActionHandler</strong></td>
<td>Executes ClientDoer or ActionDoer of a Wizard element. When the doer finishes execution the wizard frame is refreshed. Enables the action based on configured doer enabled status. In view and search modes the action's visible state is set as false.</td>
<td>This is to be used in the configuration of an &quot;Apply&quot; Action element.</td>
</tr>
<tr>
<td><strong>WebCancelActionHandler</strong></td>
<td>Closes the wizard frame.</td>
<td>This is to be used in the configuration of a &quot;Cancel&quot; Action element.</td>
</tr>
</tbody>
</table>
Configuring Doers for Wizard

Configuring a Model Doer for Wizard

As the Wizard is an instance holder, its model doer is responsible for the preparation of an Info*Engine element for it.

```xml
<ModelDoer class="%common:SimpleModelTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.create"
    taskDelegate="dca-create-model">
    <TaskParameter name="target_type" value="WCTYPE|wt.part.WTPart"/>
    <TaskParameter name="dca_field" computedValue="com.ptc.refimpl.co.
       computedvalues.DefaultLocationFieldComputedValue"/>
    <TaskParameter name="dca_field" value="name=default part name"/>
    <TaskParameter name="dca_field" value="number=default part number"/>
    <TaskParameter name="dca_field" value="lifeCycle=Default"/>
  </Task>
</ModelDoer>
```

All the properties that have a wizard element as their instance holder ancestor, will present attributes of wizard’s Info*Engine element.

Table, tree, and property panel descendent elements of a wizard will be responsible for their own data acquisition. Here is a typical configuration for the table of part usage links:

```xml
<SimpleLayout id="layout.table.uses">
  <LayoutRow>
    <Insert ref="table.pick.list.part.target"/>
  </LayoutRow>
</SimpleLayout>

<SimpleTable id="table.pick.list.part.target" selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Insert ref="pick.action.list.part"/>
  <Type type="wt.part.WTPart"/>
  <SimpleColumn show=":name">
    <Label resource="Name"/>
  </SimpleColumn>
  <SimpleColumn show=":number">
    <Label resource="Number"/>
  </SimpleColumn>
  <SimpleColumn show=":waitingpromotion">
    <Label resource="Waiting"/>
  </SimpleColumn>
  <SimpleColumn show=":checkoutstatus">
    <Label resource="Status"/>
  </SimpleColumn>
  <SimpleColumn show=":createstamp">
    <Label resource="Created"/>
  </SimpleColumn>
  <ModelDoer class="com.ptc.core.ca.co.client.doer.
      CoNoOpModelDoer"/>
</SimpleTable>
```
The model doer for the table actually does not do any data acquisition and the table is going to be populated by picking parts.

Initialization from Opener

Frequently there is a need to initialize the Info*Engine element belonging to the wizard with data from the opener action holder. For instance, when you launch a part creation wizard using an action in a row that presents a folder, you may want to use that folder as a default location.

We will call an action holder element of a launching action a source element. We will call its instance a source Info*Engine element.

DCA provides mechanisms to copy attributes values from a source instance to a wizard’s instance via an event handler or a function. We will call the attributes involved as source attribute and target attribute respectively.

Using Event Handler (Recommended)

An event handler can be specified, for instance, at the end of the executeModelDoer event. In the example below a value of a source attribute with name source_attr_name will be copied to the target attribute with the name target_attr_name. The source attribute belongs to the instance holder of the opener action in the frame the wizard was launched from. The target attribute belongs to the current element - wizard.

```
<Wizard>
  ...
  <On event="executeModelDoer" at="end">
    <Copy from="%[attribute#opener::..#source_attr_name]" to="%[attribute#.#target_attr_name]"/>
  </On>
</Wizard>
```

When a target attribute does not exist, it will be created.

**Caution**: The newly created attribute can be made available to the task as NPA. For NPA with a logical name `xxx` you must have the following statement for the appropriate data type in `LogicalAttributes.xml` when a webject used by a task complains that it can not understand what `xxx` is.

```
<Property>
  <LogicalForm>xxx</LogicalForm>
  <ExternalForm>NPA|xxx</ExternalForm>
</Property>
```

**CoWizardCopyAttributeFunction**

DCA implements `CoWizardCopyAttributeFunction` in `com.ptc.core.ca.co.client.wizard` package. The function

- locates source instance
- finds Populate statements and iterates through them
– if a statement contains `npa_attribute` creates a non persistent attribute (NPA)
– copies an attribute value from a source Info*Engine element to a wizard’s Info*Engine element.

```
<ModelDoer class="%common:SimpleModelTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.create"
    taskDelegate="dca-create-model">
    <TaskParameter name="target_type" value="WCTYPE|wt.part.WTPart"/>
    <TaskParameter name="dca_field" computedValue="com.ptc.refimpl.co.
      computedvalues.DefaultLocationFieldComputedValue"/>
    <TaskParameter name="dca_field" value="name=default part name"/>
    <TaskParameter name="dca_field" value="number=default part number"/>
    <TaskParameter name="dca_field" value="lifeCycle=Default"/>
    <TaskOutParameter function="com.ptc.core.ca.co.client.wizard.
      CoWizardCopyAttributeFunction">
      <Populate npa_attribute="xxx" from="name"/>
    </TaskOutParameter>
  </Task>
</ModelDoer>
```

**Caution:** Function must be used with `TaskOutParameter` as only output parameters have access to wizard’s Info*Engine element.

It is not necessary for a task parameter using `CoWizardCopyAttributeFunction` to have `Populate` sub-elements so you can extend `CoWizardCopyAttributeFunction` to use convenient functions to have access to source and target attributes. However, specifying
CoWizardCopyAttributeFunction without Populate sub-elements is the same as not specifying it at all.

Populate statement must have either attribute or npa_attribute. Its value defines the attribute in wizard’s Info*Engine element to copy attribute value from source Info*Engine element.

Attribute attribute specifies existing attribute. When it does not exist the exception will be thrown.

Attribute npa_attribute specifies not persistent attribute. When it does not exist in wizard's Info*Engine element, it will be created.

**Caution:** The newly created attribute can be made available to the task as NPA. For NPA with a logical name *xxx* you must have the following statement for the appropriate data type in LogicalAttributes.xml when a webject used by a task complains that it can not understand what *xxx* is.

```xml
<Property>
  <LogicalForm>xxx</LogicalForm>
  <ExternalForm>NPA|xxx</ExternalForm>
</Property>
```

The from attribute is optional. It specifies the name of the attribute in the wizard’s Info*Engine element. When not specified, an attribute specified by attribute or npa_attribute will be used in the source Info*Engine element.

### Configuring Action Doer for Wizard

The action doer for the wizard is responsible for the preparation of task parameters to perform an operation. In the example of the "Create Part" wizard, those parameters include a list of links. Here an action doer uses a function to prepare parameters including a list of usage links for a task.

```xml
<ActionDoer class="%common:SimpleActionTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-create-part-action">
    <TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoSimpleFunction_Action_Param_ObjectsAsGroup" name="uses" element="table.pick.list.part.target" selected="false"/>
    <TaskParameter name="target_type" value="WCTYPE|wt.part.WTPart"/>
    <TaskParameter name="link_type" value="WCTYPE|wt.part.WTPartUsageLink"/>
  </Task>
</ActionDoer>
```
Here is a code snippet of the demo task `dca-create-part-action` that creates a part with links:

```xml
<?xml version="1.0" standalone="yes"?>
<%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<%
String usesMasterRef = "";
int size;
%

<!DOCTYPE PROCESS>

<PROCESS name="Create Object">
<description>
Create a Windchill WTPart and add WTPartUsageLinks
</description>

<ie:webject name="Create-Objects" type="ACT">
<ie:param name="instance" data="${@FORM\[\]supporting-adapter\[\]*]"
    valueSeparator=";" delim=";"
    default="windchill"/>
<ie:param name="attribute" data="${@FORM\[\]dca_attribute\[\]*]"
    delim=","/>
<ie:param name="format" data="${@FORM\[\]format\[\]*]"
    default="false"/>
<ie:param name="next_op" data="${@FORM\[\]dca_next_operation\[\]*]"/>
<ie:param name="include_descriptors" data="${@FORM\[\]dca_include_descriptors\[\]*]"
    default="true"/>
<ie:param name="include_constraints" data="${@FORM\[\]dca_include_constraints\[\]*]"
    default="true"/>
<ie:param name="include_args" data="${@FORM\[\]dca_include_args\[\]*]"
    default="false"/>
<ie:param name="type" data="${@FORM\[\]target_type\[\]*]"/>
<ie:param name="field" data="${@FORM\[\]dca_field\[\]*]" delim=","/>
</ie:webject>

<%
Vector uses = getParams ("uses");
size = uses.size();
for (int i=0; i<size; i++)
{
    String uses_i = (String)uses.elementAt(i);
    String where = "(obid='"+uses_i+'');
%
<ie:webject name="Query-Objects" type="OBJ">
<ie:param name="INSTANCE" data="${@FORM\[\]supporting-adapter\[\]*]"
    valueSeparator=";" delim=";"
    default="windchill"/>
<ie:param name="TYPE" data="wt.part.WTPart"/>
<ie:param name="ATTRIBUTE" data="masterReference"/>
<ie:param name="WHERE" data=""%{where.toString()}%"></ie:param>
<ie:param name="GROUP_OUT" data="childPart"/>
</ie:webject>

<%
Group g = getGroup ("childPart");
%>
```
if ( g != null && g.getElementCount () > 0 )
{
    Enumeration elements = g.getElements ();
    while ( elements.hasMoreElements () )
    {
        Element element = (Element)elements.nextElement ();
        String masterReference = (String)element.getValue ("masterReference");
        if (masterReference!=null)
        {
            if (usesMasterRef.length() > 0) usesMasterRef += ";";
            usesMasterRef += masterReference;
        }
    }
} //for

<%>

if  (size > 0)
{
%

<ie:webject name="Create-Links" type="ACT" resumable="true" >
    <ie:param name="instance" data="${@FORM[]}supporting-adapter[*]"
        valueSeparator=";"
        delim=";" default="windchill"/>
    <ie:param name="format" data="${@FORM[]}format[]" default="false"/>
    <ie:param name="type" data="${@FORM[]}link_type[]"/>
    <ie:param name="from_object_ref" data="${newPart[0].obid[0]}"/>
    <ie:param name="TO_OBJECT_REF" data="<%=usesMasterRef.toString()%>" delims=";"/>
    <ie:param name="FIELD" data="quantity.amount=${@FORM[]}quantity[0]"
        default="1.0"/>
    <ie:param name="FIELD" data="quantityUnit=${@FORM[]}quantityunit[0]"
        default="ea"/>
    <ie:param name="GROUP_OUT" data="${@FORM[]}group_out[0]"
        default="partusagelink"/>

    </ie:webject>

%

</PROCESS>
Validating User Input

Wizard can be configured to ensure that required parameters are provided by the user. DCA, however, does not verify that parameter values are correct and meet the constraints imposed by servers connected via Info*Engine adapters.

To configure wizard to ensure that all required parameters are provided specify the following:

1. Put required attribute on the appropriate properties

   ```xml
   <Insert ref="new.property.name" required="true"/>
   ```

2. Add a validator for an action doer.

   ```xml
   <ActionDoer class="%common:SimpleActionTaskDoer">
     <Task nextOperation="STDO|com.ptc.windchill.view" taskDelegate="dca-create-action">
       <TaskParameter name="target_type" value="WCTYPEwt.folder.SubFolder"/>
     </Task>
     <Validator class="com.ptc.core.ca.co.client.wizard.CoWizardDialogActionValidator"/>
   </ActionDoer>
   ```

Modifying Values In The Opener Window

Values in various sources can be modified in the opener window. An event handler can be specified, for instance, at the end of the executeActionDoer event. In the example below a value of an attribute with name `attr_name` will be used to create or modify a frame parameter `param_name` in the opener frame.

```xml
<wizard>
  ...
  <on event="executeActionDoer" at="end">
    <copy from="%[attribute#opener:..#attr_name]" to="%[frame#opener:#param_name]"/>
  </on>
</wizard>
```
The Plugin element is used to embed an applet or Active X controls in DCA containers.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>16-2</td>
</tr>
<tr>
<td>Configuration</td>
<td>16-2</td>
</tr>
<tr>
<td>Renderers</td>
<td>16-4</td>
</tr>
</tbody>
</table>
**Background**

DCA Web produces HTML output to be rendered by a browser. Due to the difference between Internet Explorer and Netscape Navigator, DCA Web needs to use both `Object` and `Embed` tags. The `Applet` tag is deprecated starting from HTML 4 in favor of `Object` tag. See the HTML 4 specification for more details.

The `Object` tag can have a list of attributes, as well as a list of parameters, The `Embed` tag can only have attributes.

Example:

```html
<OBJECT attr-1="value-1" attr-2="value-2" …>
    <PARAM name="param-1" value="value-1">
    <PARAM name="param-2" value="value-2">
    …
</OBJECT>
```

The `Object` tag can have a more complex syntax, but for this release, DCA supports the basic form only.

**Configuration**

DCA provides the `Plugin` element to configure applet or ActiveX control. Use the options mechanism to configure attributes and parameters for plugin.

Example:

```xml
[<config-base>/com/ptc/common/plugin.xml]

<Synonym id="TypeApplet"
    value="application/x-java-applet;version=1.3"/>
<Synonym id="AppletTestCode"
    value="com/ptc/refimpl/co/client/plugin/TestApplet"/>

<Plugin id="plugin.template">
    <Option theme="attribute" param="name"   value="plugin"/>
    <Option theme="parameter" param="debug"  value="false"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
        plugin/windchillPlugin.jsp"/>
</Plugin>

<Plugin id="applet.template">
    <Extend ref="plugin.template"/>
    <Option theme="attribute" param="name"   value="applet"/>
    <Option theme="attribute" param="width"   value="2"/>
    <Option theme="attribute" param="height"  value="2"/>
    <Option theme="attribute" param="type"    value="%TypeApplet"/>
</Plugin>

<Plugin id="applet.test">
    <Extend ref="applet.template"/>
    <Option theme="attribute" param="width"   value="400"/>
    <Option theme="attribute" param="height"  value="50"/>
    <Option theme="attribute" param="code"    value="%AppletTestCode"/>
</Plugin>
```
Insert the plugin element into some other element, for example, frame:

```xml
<SimpleFrame id="dialog.applet.test" width="750" height="400">
  <Extend ref=":simpleframe.template"/>
  <Label resource="Download"/>
  <SimpleLayout>
    <LayoutRow>
      <Insert ref="applet.test"/>
    </LayoutRow>
  </SimpleLayout>
</SimpleFrame>
```

Use the following URL to see a demo applet:

```
```

The frame with the demo applet should look like the following figure:

![Demo Applet Frame](image)

**Static Parameters**

Static parameters are specified through the Option mechanism using the `attribute` or `parameter` theme.

```xml
<Option theme="attribute" param="name" value="plugin"/>
<Option theme="parameter" param="debug" value="false"/>
```

**Caution:** In DCA Web when the Embed tag is used, both themes will result in a list of attributes as the Embed tag does not recognize parameters.
Dynamic Parameters

Dynamic parameters are those that depend on the attributes of Info* Engine elements. They are available at the runtime only and you need to define and configure a model doer to set them. The doer will have access to the Plugin element and its model by the doer data parameter of its execute method. The Plugin model holds two maps, one for attribute, and another for parameter key-value pairs. The model doer is able to modify the maps in order to update all runtime values.

The ModelDoer is an optional tag, so the test applet doesn't use it. There is no default implementation of such a doer, since an applet has no clear runtime properties, generally.

Renderers

The default renderer for a Plugin element is windchillPlugin.jsp. It renders a Plugin element using Object and Embed tags to ensure that plugin works in both IE and Netscape browsers.
The Legacy Bridge is available to link DCA frames to external legacy resources, like template processor-based pages and ProjectLink-based pages, for example, and to link back to DCA frames.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linking to a Template Processor-Based Resource</td>
<td>17-2</td>
</tr>
<tr>
<td>Linking to a Windchill Resource</td>
<td>17-3</td>
</tr>
<tr>
<td>Linking to a NetMarket Resource</td>
<td>17-4</td>
</tr>
<tr>
<td>Linking with Dynamically Generated Parameters</td>
<td>17-5</td>
</tr>
</tbody>
</table>
Linking to a Template Processor-Based Resource

Linking to a Template Processor-Based Resource Without an Object Reference

To link to a template processor-based resource without an object reference, configure the HyperLink element using a special model handler as follows:

```xml
<HyperLink id="h0" labelType="text" drawType="text">
  <Label resource="Link to Windchill TP"/>
  <ToolTip resource="Link to Windchill TP"/>
  <ModelHandler class="com.ptc.core.ca.co.client.windchill.
                   CoWindchillTPHyperLinkProducer">
    <Template anonymous="false">
      <Option theme="param" name="a" value="b"/>
    </Template>
  </ModelHandler>
</HyperLink>
```

Using the configuration in the example above, DCA will produce this URL:

http://host.ptc.com/Windchill/servlet/WindchillAuthGW?a=b

Attribute `anonymous` specifies whether to generate a link using anonymous or authenticated Windchill gateway. The default value is false.

Template processor class and method can be specified as attributes to `Template` element.

```xml
<Template class="myclass" method="mymethod"/>
```

You can specify as many parameters as you want with `theme="param"`. Parameters must follow the convention for the template processor resource you configure a link to.

Linking to a Template Processor-Based Resource With Object Reference

To link to a template processor-based resource with object reference, configure the HyperLink element using a special model handler that utilizes "simple" approach:

```xml
<HyperLink id="details" labelType="image" drawType="image">
  <ToolTip resource="Details"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/favicon.ico"/>
  <ModelHandler class="com.ptc.core.ca.co.client.windchill.
                   CoSimpleWindchillTPHyperLinkProducer">
    <Template class="wt.enterprise.URLProcessor" method="URLTemplateAction" anonymous="false">
      <Option theme="param" name="action" value="ObjProps"/>
    </Template>
  </ModelHandler>
</HyperLink>
```
Using the configuration in the example above, DCA will produce this URL:

http://host.ptc.com/Windchill/servlet/WindchillAuthGW/wt.enterprise.URLProcessor/URLTemplateAction?
oid=VR%3Awt.doc.WTDocument%3A6200&action=ObjProps&u8=1

Attribute anonymous specifies whether to generate a link using anonymous or authenticated Windchill gateway. The default value is false.

Template processor class and method must be specified as attributes to Template element.

```
<Template class="myclass" method="mymethod"/>
```

You can specify as many parameters as you want with theme="param". Parameters must follow the convention for the template processor resource you configure a link to.

**Linking to a Windchill Resource**

**Linking to a Windchill Resource Without Object Reference**

To link to a Windchill resource without object reference, configure the HyperLink element using a special model handler as follows:

```
<HyperLink id="h2" labelType="text" drawType="text">
  <Label resource="Link to Windchill" />
  <ToolTip resource="Link to Windchill" />
  <HRef resource="wtcore/jsp/wt/portal/index.jsp"/>
  <ModelHandler class="com.ptc.core.ca.co.client.windchill.CoWindchillHyperLinkProducer">
    <Option theme="param" name="a" value="b"/>
  </ModelHandler>
</HyperLink>
```

Using the configuration in the example above, DCA will produce this URL:

http://host.ptc.com/Windchill/wtcore/jsp/wt/portal/index.jsp?a=b

You can specify as many parameters as you want with theme="param". Parameters must follow the convention for the resource you configure a link to.

**Linking To a Windchill Resource With Object Reference**

To link to a Windchill resource with object reference, configure HyperLink element using a special model handler that utilizes "simple" approach:

```
<HyperLink id="h2" labelType="text" drawType="text">
  <Label resource="Link to Windchill" />
  <ToolTip resource="Link to Windchill" />
  <HRef resource="wtcore/jsp/wt/portal/index.jsp" />
  <ModelHandler class="com.ptc.core.ca.co.client.windchill.CoSimpleWindchillHyperLinkProducer">
    <Option theme="param" name="a" value="b"/>
  </ModelHandler>
</HyperLink>
```
Using the configuration in example above DCA will produce this URL

http://host.ptc.com/Windchill/wtcore/jsp/wt/portal/index.jsp?a=b&o id=OR%3Awv.doc.WTDocument%3A6200

You can specify as many parameters as you want with theme="param". Parameters must follow the convention for the resource you configure a link to.

### Linking to a NetMarket Resource

NetMarket framework is a foundation of Project Link and Dynamic Design Link solutions.

### Linking to a NetMarket Resource Without Object Reference

To link to a NetMarket resource without object reference, configure the HyperLink element similarly to how you configure it to link to a Windchill resource:

```xml
<HyperLink id="h2" labelType="text" drawType="text">
  <Label resource="Link to Windchill"/>
  <ToolTip resource="Link to Windchill"/>
  <HRef resource="/netmarkets/jsp/netmarkets/view.jsp"/>
  <ModelHandler class="com.ptc.core.ca.co.client.windchill.CoWindchillHyperLinkProducer">
    <Option theme="param" name="a" value="b"/>
  </ModelHandler>
</HyperLink>
```

Using the configuration in the example above, DCA will produce this URL:

http://host.ptc.com/Windchill/netmarket/jsp/netmarkets/view.jsp?a=b

You can specify as many parameters as you want with theme="param". Parameters must follow the convention for the resource you configure a link to.

### Linking To a NetMarket Resource With Object Reference

To link to a Windchill resource with object reference, configure HyperLink element using a special model handler that utilizes "simple" approach and specify an appropriate nm type:

```xml
<HyperLink id="h2" labelType="text" drawType="text">
  <Label resource="Link to Windchill"/>
  <ToolTip resource="Link to Windchill"/>
  <HRef resource="/netmarkets/jsp/netmarkets/view.jsp"/>
  <ModelHandler class="com.ptc.core.ca.co.client.windchill.CoSimpleWindchillNmHyperLinkProducer" nmType="doc">
    <Option theme="param" name="a" value="b"/>
  </ModelHandler>
</HyperLink>
```
Using the configuration in example above DCA will produce this URL

http://host.ptc.com/Windchill/wtcore/jsp/wt/portal/index.jsp?a=b&o id=doc%7AEwt.doc.WTDocument%3A62000

You can specify as many parameters as you want with theme="param". Parameters must follow the convention for the resource you configure a link to.

**Linking with Dynamically Generated Parameters**

Sometimes there is a need to add dynamically generated parameters to hyper links. This can not be achieved through the use of an Option element. Application developers will need to extend the appropriate hyper link producer class and overridegetParam method.

```java
class MyProducer extends CoSimpleWindchillTPHyperLinkProducer {
    public HashMap getParams( CoElement element, CoEnvData envData )
        throws WTException {
        HashMap map = super.getParams(element, envData);
        map.put ("myparam", "myvalue");
        return map;
    }
}
```

A hyper link needs to be configured to use a new hyper link producer:

```xml
<HyperLink id="h2" labelType="text" drawType="text">
    <Label resource="Link to Windchill"/>
    <ToolTip resource="Link to Windchill"/>
    <HRef resource="wtcore/jsp/wt/portal/index.jsp"/>
    <ModelHandler class="MyProducer">
        <Option theme="param" name="a" value="b"/>
    </ModelHandler>
</HyperLink>
```

Here is a resultant URL:

http://host.ptc.com/Windchill/wtcore/jsp/wt/portal/index.jsp?a=b&myparam=myvalue&o id=VR%3Awt.doc.WTDocument%3A62000
This chapter will reference the coolpart soft type and its sample XML file:

<table>
<thead>
<tr>
<th>XML file</th>
<th>S$(wt.home)/codebase/registry/dca/com/ptc/coolpart/config_coolpart.xml</th>
</tr>
</thead>
<tbody>
<tr>
<td>To load sample soft type coolpart and create sample coolpart objects.</td>
<td>S$(wt.home)/loadfiles/dca/coolpart.csv</td>
</tr>
</tbody>
</table>

**Topic**  
- Modifying Configuration Files to Support a Newly Created Soft Type .......... 18-2  
- Creating and Modifying a Property Sheet to Support a Soft Type ................. 18-3  
- Modifying a Table Definition to Support a Soft Type ........................................ 18-4
Modifying Configuration Files to Support a Newly Created Soft Type

1. Create an xml file and package for a new soft type.
   <Repository package="com.ptc.coolpart">

2. Import package of extended type in new package:
   <Import package = "com.ptc.part"/>

3. Define SYNONYM for object type if needed:
   <Context>
   <Synonym id="CoolPart" value="CoolPart"/>
   </Context>

Create the following entries in logicalIdentifiers.properties

CoolInt = IBA|CoolInt
CoolURL = IBA|CoolURL
CoolFloat = IBA|CoolFloat
CoolUnit = IBA|CoolUnit
CoolRatio = IBA|CoolRatio
CoolTimestamp = IBA|CoolTimestamp
CoolBoolean = IBA|CoolBoolean
CoolString = IBA|CoolString
CoolTextArea=IBA|CoolTextArea
CoolPart=WCTYPE|wt.part.WTPart|CoolPart

4. Create property elements for the attributes:

   <Context type="%[CoolPart]">
   <Property id="CoolURL">
   <Label resource="Cool URL:"/>
   <Need attribute="CoolURL"/>
   <PropertyHandler class="com.ptc.core.ca.web.client.property.
   hyperlink.WebHyperlinkPropertyHandler"/>
   <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
   property/hyperlink/hyperlink.jsp"/>
   </Property>
   <!-- Do this for all the properties using the correct data
   type handlers and renderers for the attribute -->
   </Context>
Creating and Modifying a Property Sheet to Support a Soft Type

Define a new property sheet by extending the existing property panel element for the extended type, using the context of the CoolPart. Add the additional properties after the extend statement.

```xml
<Context type="%[CoolPart]%" />
<SimpleLayout id="layout.view.task" type="%[CoolPart]%">
        <Extend ref=":layout.view" type="%[part:WTPart]%"/>
        <LayoutRow id="15">
                <Insert ref="CoolURL"/>
        </LayoutRow>
        <LayoutRow id="16">
                <Insert ref="CoolInt"/>
        </LayoutRow>
        <LayoutRow id="17">
                <Insert ref="CoolFloat"/>
        </LayoutRow>
        <LayoutRow id="18">
                <Insert ref="CoolRatio"/>
        </LayoutRow>
        <LayoutRow id="19">
                <Insert ref="CoolTimestamp"/>
        </LayoutRow>
        <LayoutRow id="20">
                <Insert ref="CoolBoolean"/>
        </LayoutRow>
        <LayoutRow id="21">
                <Insert ref="CoolString"/>
        </LayoutRow>
        <LayoutRow id="22">
                <Insert ref="CoolUnit"/>
        </LayoutRow>
        <LayoutRow id="23">
                <Insert ref="CoolTextArea"/>
        </LayoutRow>
</SimpleLayout>
</Context>
```
Modifying a Table Definition to Support a Soft Type

1. There is no need to add a new type to the existing table configuration when there is already an entry for a super type. Otherwise add a new Type sub element to the table configuration. CoolPart will now show as rows in the table if the Doer query criteria will find them.

   `<Type type="${coolpart:CoolPart}"/>

2. To add a new column to the table to show the a new attribute, define a SimpleColumn or CompositeColumn with the show attribute specified to the ID of the property defined for the CoolPart. Use the ":" scoping rule so the attribute is found based on the context. In this case, the type of the object for the row which is CoolPart.

   `<SimpleColumn show=":CoolURL">
     <Label resource="Cool URL"/>
   </SimpleColumn>`
This chapter explains how elements are used and managed within DCA.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Cache</td>
<td>19-2</td>
</tr>
<tr>
<td>Element Tree</td>
<td>19-2</td>
</tr>
<tr>
<td>Draw Handler</td>
<td>19-3</td>
</tr>
<tr>
<td>Model Handler</td>
<td>19-4</td>
</tr>
<tr>
<td>Render Handler</td>
<td>19-4</td>
</tr>
<tr>
<td>Element Address</td>
<td>19-5</td>
</tr>
<tr>
<td>Client Server Communications</td>
<td>19-8</td>
</tr>
<tr>
<td>Rendering</td>
<td>19-8</td>
</tr>
<tr>
<td>Element Events</td>
<td>19-10</td>
</tr>
<tr>
<td>Conditional Drawing</td>
<td>19-11</td>
</tr>
<tr>
<td>Shadow</td>
<td>19-12</td>
</tr>
<tr>
<td>Performance Tuning</td>
<td>19-14</td>
</tr>
<tr>
<td>Understanding Exceptions</td>
<td>19-15</td>
</tr>
</tbody>
</table>
Frame Cache

The DCA framework maintains a cache of DCA frame elements. Every frame element has a unique address and time stamp of the last draw. When a new frame element needs to be created and there is not enough space in the cache, any older frames are discarded. DCA Web provides a property to control the maximum number of frames in the DCA cache. The default value is 10.

```
com.ptc.core.ca.web.client.element.factory.maxFrameCount=<count>
```

When DCA Web receives a request to perform an operation resulting from triggering an action, it tries to locate an appropriate frame in the cache that the action belongs to. If a frame is not in the cache, a new frame is created and drawn. In debug mode, DCA Web will throw an exception; in user mode, the user will be prompted to repeat an operation.

Element Tree

The DCA frame maintains a tree of DCA elements. During the process of drawing a frame, descendent elements will be drawn recursively. The following illustrations describe what is involved in the drawing process:
Every DCA element implements a default drawing algorithm that involves initialization of an element and its rendering. It is possible to provide a so-called *draw handler* to replace this algorithm.

```
element.default_init()
```

```
element.default_render()
```

```
frame.draw()
```

```
element.draw()
```

```
Configuration subsystem
```

```
Data Access and Manipulation
```

```
Draw Handler
```

```
drawHandler.draw()
```

```
Has draw handler?
```

```
What do I look like?
```

```
What do I do?
```

```
Give me the data I need
```

```
What do I do?
```

```
Draw Handler
```

```
What do I look like?
```

```
Give me the data I need
```

```
UI Framework
```

```
Sub Elements
```

```
frame.draw()
```

```
draw()
```

```
element.draw()
```

```
Has draw handler?
```

```
element.default_init()
```

```
element.default_render()
```
**Model Handler**

Default initialization algorithm updates a model of an element first and then makes sure the element has correct children.

```
  element.default_init()
  Has model handler?
```

```
  modelHandler.updateModel()
  container.updateChildren()
```

```
  element.updateModel()
```

When a default algorithm is not sufficient, a so-called *model handler* can be provided.

**Render Handler**

In DCA Web, the rendering of an element invokes a Java bean-based renderer.

```
  element.default_render()
  has render handler?
```

```
  renderHandler.render()
```

```
  element.render()
```

When so-called render handler is provided, it is asked to render an element. Typically render handlers are used to improve performance of a production system as the cost of its invocation is substantially less than that of a jsp page.
Element Address

Every frame has a unique address. Every descendent element has a unique address in the address space of its parent element. Thus it’s possible to construct a unique address (long address) of the element in DCA by concatenating addresses.

An address object can be externalized to a string. Thus long addresses can be externalized to a string by joining external forms of addresses.

There are three types of element addresses:

- static element address
- address of a frame
- dynamic element address

Static Element Address

The address of an element that is statically defined by configuration information, for example, a Label or Table.

External Form

\[
<\text{element id in session}!1!\text{<config feature id}>\]

Example

\[402!1!\text{actions}\]

where:

- 402 - unique element id in session
- 1 - indicates it’s a static element address
- actions - id of the configuration feature
Address of a Frame

The address of a frame element.

External Form

<element id in session>!3!<config feature id>! 
<frameId>!<context>!<type>~~<ufid>

Example

344!3!folder:browser!901098!SL:*:visitor! 
wt.folder.SubFolder~~OR:wt.folder.SubFolder:60813:706477804- 
1014927869984-8362568-80-96-253-132@myhost.ptc.com

where

344 - unique element ID in session

3 - indicates it's an address of a frame

folder:browser - ID of the configuration feature describing the frame

SL:*:visitor - context of the frame where

SL - a value of appId dimension

* - a value of a decoration dimension (not set if *)

visitor - a value of a roleId dimension

wt.folder.SubFolder - type of the object

OR:wt.folder.SubFolder:60813:706477804- 
1014927869984-8362568-80-96-253-132@myhost.ptc.com

- UFID of the Info*Engine element
**Dynamic Element Address**

The address of a DCA element that is not statically defined by the configuration information, for example, a Row or TreeNode.

**External Form**

```<element id in session>!2!<type>~~<ufid>```

**Example**

```
440!2!wt.doc.WTDocument~~
VR:wt.doc.WTDocument:65320:996734169-1015437860265-
7370093-80-96-253-132@yourhost.ptc.com
```

where

- **440** - unique element ID in session
- **2** - indicates it’s a dynamic element address
- **wt.doc.WTDocument** - type of the object
- **VR:wt.doc.WTDocument:65320:996734169-1015437860265-
  7370093-80-96-253-132@yourhost.ptc.com** - UFID of the Info*Engine element

**Example of a Long External Form**

[broken down to pieces for readability]

```
0!3!com.ptc.folder:first.page.tree.view!*!*:*:*!*$
1!1!SimpleLayout$
6!1!row2$
19!1!altfirstpage.tree$
23!1!firstpage.tree.topLayout$
38!1!$50!1!treebrowser.actions.select$
54!1!action.paste:
```
Client Server Communications

The DCA client communicates with the DCA server through name value pairs. In DCA Web, an HTTP request will contain name value pairs where some of the pairs will contain the long external form of element addresses as either name or value.

For instance, triggering the “Paste” action sends name value pair with name `in_place_action_address` and value containing an address of the “Paste” action element. The address gets resolved into a reference to an action element and an operation is executed according to the configuration specification.

Another example is when the user selects a set of elements and clicks "Delete". DCA Web will send name value pairs for each selected element with name `selected` and value containing external form of an address of a selected element. Addresses get resolved into references to elements and the delete operation is executed according to the configuration specification.

Rendering

DCA Web uses the following algorithm to render a DCA element.

1. Check to see if renderer is specified via configuration specification. This can be either java-based renderer or jsp-based renderer. Use it when found.

2. Check to see if an element has a default java-based renderer. Use it when found.
3. Check to see if an element has a default jsp-based renderer. Use it when found.

4. Raise an exception otherwise.

Property element uses a different algorithm as it does not have a default renderer of any type. A property renderer must be provided via configuration.

**Java-Based Renderers**

In DCA Web Java-based renderers must extend `com.ptc.core.ca.web.client.element.WebRenderHandler` class. Typically java-based renderers are used to improve performance of a production system as the cost of its invocation is substantially less than that of a jsp page.

Java-based renderer writes to the output stream provided as a parameter to the method it must implement:

```java
protected void render( CoElement element,
                       JspWriter out,
                       WebEnvData web_envData )
    throws WTException {
```
Switching Between Java-based and JSP-based Renderers

You can change the rendering algorithm to give preference to jsp-based renderers by adding the following property to wt.properties file

```
com.ptc.core.ca.web.client.frame.context.dimensions.rendererType=jsp
```

This will also establish the value of the `rendererType` dimension you can check for any element in the configuration specification.

Example:

```
<Context>
  <Renderer id="renderer.action" resource="/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp"/>
</Context>

<Context rendererType="java">
  <Renderer id="renderer.action" class="com.ptc.core.ca.web.client.action.WebActionRenderHandler"/>
</Context>

<Action>
  ...
  <Insert ref="#renderer.action"/>
</Action>
```

When a property is set to `jsp`, jsp renderer will be chosen.

Element Events

During the process of drawing a DCA element, the element goes through the following stages

- drawing
- initialization of a model
- update of a model
- execution of a model Doer
- update of a set of children
- rendering

Not all the elements go through all the stages. For instance, some of the elements, like a separator, cannot have doers. An action element can additionally go through the process of an execution of an action doer.

Based on the stages of DCA elements, DCA provides a mechanism to fire an event before an element gets in a particular stage and after it leaves it. It is possible to associate handlers for a particular element event via configuration specification. For instance, the following instructs DCA to initialize the value of the enumerator at the end of updating its model.
A new element On is introduced. The following is an example of the specification of an event handler to set up default value of an enumerator.

```xml
<Enumerator id="search5.enumerator.type">
  <Insert ref="search5.enumerator.type.part" value="%[doc:WTDocument]"/>
  <Insert ref="search5.enumerator.type.doc" value="%[part:WTPart]"/>
  <Insert ref="search5.enumerator.type.folder" value="%[folder:SubFolder]"/>
  <On event="initModel" at="end">
    <Copy from="%[part:WTPart]" to="%[enum#.]"/>
  </On>
</Enumerator>
```

**Conditional Drawing**

To implement conditional drawing, specify a draw handler that conditionally delegates drawing to the element’s default implementation.

```java
public void draw( CoElement element, CoEnvData envData )
  throws WTException {
    if (condition met) {
      element.default_draw(envData);
    } else {
      remove subelements
    }
}
```

In DCA Web, it is important to remove subelements when they expect user input to avoid situation as there is no input since they were not drawn.

**com.ptc.core.ca.co.client.element.CoDrawController**

Draw controller draws a layout or a layout row element when the values of correspondent Enumerator elements and frame properties match target values.

```xml
<LayoutRow id="p">
  <Insert ref="part:search2.layout.part"/>
  <DrawHandler class="com.ptc.core.ca.co.client.element.
CoDrawController">
    <Check value="%[enum#search2.enumerator.type]==part"/>
  </DrawHandler>
</LayoutRow>
```

In the example above draw controller will allow drawing of the layout row when an enumerator with id `search2.enumerator.type` has a value equal to `part`.

**Caution**: Enumerator element has no relationship with a property of enumerated type.

The next example presents how to configure to draw conditionally based on the value of the frame property with the name `com.ptc.search.type_param`
It is possible to specify multiple conditions to check enumerator and frame property values for a single controller. All of them must be met for a controller to allow the drawing of the element.

**Caution:** CoDrawController can control drawing of a layout or a layout row element only.

### Shadow

With the exception of a frame, every DCA element is always a child of a parent element. When a parent element is rendered, it asks its children to draw themselves. However, it does it only once. Most of the time this is the desired behavior, but there are situations when it’s not. Consider an example below when an enumerator controls which input field needs to be used as search criteria.

In this example, we have a layout with three layout rows containing properties for inputting search criteria. Certainly, we can configure the enumerator to draw itself vertically and to occupy three cells using rowspan equal to 3. Functionality wise, it will work perfectly. The problem, however, is in the imperfections of the resultant HTML layout-enumerated options will not be aligned with the layout rows.

To address the problem, a Shadow configuration element is introduced. The Shadow element itself references other DCA element and draws a shadow of the
referenced element. The Shadow element itself does not know what the shadow looks like. Instead, the referenced element (we’ll call it the original element) is responsible for providing all the data the shadow needs to display itself at the time of rendering the original element. The following is the configuration of the enumerator in the example above:

```xml
[com/ptc/search/date_search.xml]
<Enumerator id="createStamp.option">
  <LabelArea id="la.0" labelType="text" value="On Date"
    default="true">
    <Label resource="On Date"/>
  </LabelArea>
  <LabelArea id="la.1" labelType="text" value="Range">
    <Label resource="Range"/>
  </LabelArea>
  <LabelArea id="la.2" labelType="text" value="Since">
    <Label resource="Since"/>
  </LabelArea>
  <Renderer class="com.ptc.core.ca.web.client.primitive.WebEnumeratorRenderHandler">
    <Option theme="widget" param="type" value="radiobutton"/>
    <Option theme="widget" param="vertical" value="true"/>
    <Option theme="widget" param="labelFirst" value="true"/>
    <Option theme="widget" param="shadow" value="true"/>
  </Renderer>
</Enumerator>
```

**Note:** Not every element has an ability to have shadows. At the moment only Enumerator or DataDrivenEnumerator with radio button renderer can have shadows.

The following is the configuration of the search fragment for the example above.

```xml
[com/ptc/search/date_search.xml]
<ElementGroup id="search.panel.row.group.createStamp">
  <LayoutRow id="lr.0">
    <LabelArea labelType="text" rowspan="3">
      <Label resource="Created:"/>
    </LabelArea>
    <Insert ref="createStamp.option"/>
    <Insert ref="createStamp.value.onDate"/>
  </LayoutRow>
  <LayoutRow id="lr.1">
    <Shadow ref="createStamp.option"/>
    <Insert ref="createStamp.value.fromDate"/>
    <Insert ref="createStamp.value.toDate"/>
  </LayoutRow>
  <LayoutRow id="lr.2">
    <Shadow ref="createStamp.option"/>
    <LabelArea labelType="text">
      <Label resource=""/>
    </LabelArea>
    <Insert ref="createStamp.value.since"/>
  </LayoutRow>
</ElementGroup>
```
It is a responsibility of the original element to provide an output that the shadow element can present. The original element does it repeatedly using the following method in DCA Web implementation to place an output in a special buffer:

```java
public class WebPrimitiveUtils {
    static public void putShadowContent(CoElement element,
                                        Object content,
                                        CoEnvData envData)
        throws WTException;
}
```

The original element must know the expected number of shadows to call the method the appropriate number of times.

The shadow element is a DCA element without a model. It simply call the following method to retrieve an object that can be used to present a shadow.

```java
public class WebPrimitiveUtils {
    static public Object getShadowContent(CoElement element,
                                          CoEnvData envData)
        throws WTException;
}
```

The method retrieves and removes data from the buffer to ensure the next shadow element gets the appropriate data.

**Performance Tuning**

To log time spend for task executions vs. rendering and other DCA activities, set the following property in `wt.properties`:

```properties
com.ptc.core.ca.co.common.util.dcaTimerMode=true
```

DCA will write log into `$(wt.home)/logs/DCAtiming.log` file in the following format:

```
DCA Time=380  Task Time=1072  JSP Rendering Time=0
DCA Time=290  Task Time=962   JSP Rendering Time=0
```

To find what jsp-based renderers are used and how often, set the following property in `wt.properties`

```properties
com.ptc.core.ca.co.common.util.renderLogMode=true
```

DCA will write log into `$(wt.home)/logs/rendererStat.log` file.
Understanding Exceptions

To control exception reporting, DCA Web provides the ability to specify a property in wt.properties.

```
com.ptc.core.ca.web.client.gw.debugMode=<true|false>
```

The default value is false. When set to true, DCA Web will produce information containing request parameters and stacktrace when an exception occurs. You will see all name value pairs the DCA Web client (browser) sent to DCA Web server (jps engine). Some of the pairs will contain long external forms of addresses of DCA elements as name or value.
DCA provides a variety of utility classes with static methods.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Utilities</td>
<td>20-2</td>
</tr>
<tr>
<td>Action Utilities</td>
<td>20-2</td>
</tr>
<tr>
<td>Element Utilities</td>
<td>20-3</td>
</tr>
<tr>
<td>Option Utilities</td>
<td>20-8</td>
</tr>
</tbody>
</table>
Address Utilities

DCA provides address utilities in the
com.ptc.core.ca.co.client.address package.

<table>
<thead>
<tr>
<th>AddressUtils</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;Abstract&gt;&gt;</td>
</tr>
<tr>
<td>CoAddressUtils</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; fromLongExternalForm(longAddressStr : String, envData : CoEnvData) : java.util.ArrayList</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; fromLongExternalFormQuick(longAddressStr : String, envData : CoEnvData) : CoAddress</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; fromShortExternalFormQuick(addressStr : String, envData : CoEnvData) : CoAddress</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; toLongExternalForm(address : CoAddress, envData : CoEnvData) : String</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; toLongExternalForm(address : CoAddress, buf : StringBuffer, envData : CoEnvData)</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; getAddressType(addressStr : String, envData : CoEnvData) : String</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; getStringTokenizerForInit(addressStr : String, envData : CoEnvData) : java.util.StringTokenizer</td>
</tr>
</tbody>
</table>

The purpose of most of the methods is to convert an address to its external form and vice versa.

Action Utilities

DCA provides action utilities in the
com.ptc.core.ca.co.client.action package.

<table>
<thead>
<tr>
<th>ActionUtils</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;Abstract&gt;&gt;</td>
</tr>
<tr>
<td>CoActionUtils</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; getActionHolder(action : CoAction, envData : CoEnvData) : CoElement</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; getActionHolderAddress(action : CoAction, envData : CoEnvData) : CoAddress</td>
</tr>
<tr>
<td>&lt;&lt;Static&gt;&gt; updateGoToAddress(data : CoSimpleActionHandlerData)</td>
</tr>
</tbody>
</table>

Action utilities are usually used to locate a so-called *action holder* element for a given action. An action holder is the element that logically owns an action.
Element Utilities

DCA provides action utilities in the
com.ptc.core.ca.co.client.element package.

The following methods are of the most interest to application developers.
visitDown

public static CoElementVisitStatus visitDown(
    CoElement element,
    CoElementVisitor visitor,
    CoElementVisitType type,
    CoEnvData envData)
throws WTException

Use it when you need to traverse breadth first, a tree of elements and invoke a
visitor for each node.

Example of a method to locate an element by ID:

GET_ELEMENT
public static final CoElementVisitType VISIT_TYPE_GET_ELEMENT_BY_ID =
    new CoElementVisitType("GetElementById");
private static final CoElementVisitor VISITOR = new Visitor();

public static final String GET_ELEMENT_BY_ID_IN_ID = "GetElementById.In";
public static final String GET_ELEMENT_BY_ID_OUT = "GetElementById.Out";

public static Element findDescendentById(CoElement element,
    String id,
    CoEnvData envData)
throws WTException {
    Map anyData = envData.getRequestData().getAnyData();
    anyData.put(GET_ELEMENT_BY_ID_IN_ID, id);

    CoElementIUtils.visitDown(element, VISITOR,
        VISIT_TYPE_GET_ELEMENT_BY_ID, envData);

    Element element = (Element)anyData.get(GET_ELEMENT_BY_ID_OUT);
    anyData.remove(GET_ELEMENT_BY_ID_IN_ID);
    if (element != null) {
        anyData.remove(GET_ELEMENT_BY_ID_OUT);
    }
    return element;
}

private static class Visitor implements CoElementVisitor {
    public CoElementVisitStatus visit( CoElement element,
        CoElementVisitType type,
        CoEnvData envData )
    throws WTException {
        CoElementVisitStatus status = CoElementVisitStatus.CONTINUE;

        if (type == VISIT_TYPE_GET_ELEMENT_BY_ID) {
            CoAddress address = element.getAddress();
            if (address instanceof CoStaticElementAddress) {
                CoStaticElementAddress static_address = (CoStaticElementAddress)address;
                String alias = static_address.getAlias();

                Map anyData = envData.getRequestData().getAnyData();
                String id = anyData.get(GET_ELEMENT_BY_ID_IN_ID).toString();
if (id.equals(alias)) {
    anyData.put(GET_ELEMENT_BY_ID_OUT, element);
    status = CoElementVisitStatus.STOP;
}
return status;
}

visitDown

public static CoElementVisitStatus visitDownDepthFirst(
    CoElement element,
    CoElementVisitor visitor,
    CoElementVisitType type,
    CoEnvData envData)
    throws WTException

Use it when you need to traverse depth first, a tree of elements and invoke a visitor for each node.

getInstance

public static CoInstance getInstance(
    CoInstanceHolder instanceHolder,
    CoEnvData envData)
    throws WTException {

Returns a so-called coinstance of instance holder.

Example of a method to get an Info*Engine element from an instance holder:

public static Element getIeElement(
    CoInstanceHolder instanceHolder,
    CoEnvData envData)
    throws WTException {
    CoInstance instance = getInstance(instanceHolder, envData);
    Element ie_element = null;
    if (instance != null) {
        ie_element = instance.getIeElement();
    }
    return ie_element;
}

setInstance

Sets a so-called coistance to an instance holder.
**getInstanceList**

```java
public static ArrayList getInstanceList(
    CoInstanceHolder instanceListHolder,
    CoEnvData envData)
throws WTException {

Returns a list of co-instances of instance list holder.
```

**setInstanceList**

```java
public void setInstanceList(
    CoInstanceHolderList instanceListHolder,
    ArrayList instances,
    CoEnvData envData)
throws WTException {

Sets a list of co-instances to an instance list holder.
```

**Example of a doer that populates a table with selected elements:**

```java
public void execute( CoSimpleActionHandlerData execData )
throws WTException {
    CoAction action = execData.getAction();
    CoFrame frame = action.getFrame();
    CoEnvData envData = execData.getEnvData();

    ArrayList elements =
        CoElementUtils.findDescendentsByIId(frame, "table_id",
            true, false, envData);
    CoTable table = (CoTable) elements.get(0);
    ArrayList selected = execData.getInstanceHolders();

    ArrayList content = CoElementUtils.getInstanceList(target,
            envData);

    //merge "selected" into "content"
    for (int i=0; i<selected.size();i++){
        CoInstanceHolder selected_element =
            (CoInstanceHolder)selected.get(i);
        Element selected_ie_element =
            CoElementUtils.getIeElement(selected_element, envData);
        CoInstance new_instance = execData.getEnvData().
            getSessionData().getInstanceCache().
            getInstance (selected_ie_element, null);
        content.add(i, new_instance);
    }

    CoElementUtils.setInstanceList(target, content , envData);
}
```
**getIeElement**

```java
public static Element getIeElement(
    CoInstanceHolder instanceHolder,
    CoEnvData envData)
throws WTException {
    Returns an Info*Engine element of an instance holder.
}
```

**getIeElementOfInstanceHolder**

```java
public static Element getIeElementOfInstanceHolder(
    CoElement element,
    CoEnvData envData)
throws WTException {
    When an element is an instance holder, it returns its Info*Engine element. Otherwise, it finds an instance holder ancestor of an element and returns its Info*Engine element.
}
```

**getModel**

```java
public static CoModel getModel(
    CoModelDriven modelDriven,
    CoEnvData envData)
throws WTException {
    Returns a model of a model driven element.
}```
Option Utilities

DCA provides option utilities in the com.ptc.core.ca.co.client.element package.

Use option utilities when you need to query option values. The following is an example of getting a height of a property:

```xml
<Property id="property.TextArea.template">
    <PropertyHandler class="com.ptc.core.ca.web.client.property.
        string.WebTextAreaPropertyHandler"/>
    <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
        property/string/textarea.jsp">
        <Option theme="size" param="height" value="5"/>
        <Option theme="size" param="width" value="40"/>
    </Renderer>
</Property>
```

```java
public Integer getHeight(
    CoProperty property,
    CoEnvData envData)
    throws WTException {
    CoConfigFeature configFeature = property.getConfiguration(envData);
    Integer height = CoOptionUtils.
        getRenderingOptionIntegerValue(configFeature,
            "size",
            "height",
            envData);
    return height;
}
```
The family of methods `getOptionXxxValue` returns a value attribute of an Option element that is a child of an element described by the configuration feature.

The family of methods `getRenderingOptionXxxValue` locates a description of a renderer or render handler of an element described by the configuration feature. Then it returns a value attribute of an Option element that is a child of located element. See example above.
DCA provides an environment-independent DCA environment. DCA-based applications are expected to access the DCA environment and avoid accessing HTTP, Java, or Windows-specific objects. DCA provides an environment-independent notion of a session and a request.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>21-2</td>
</tr>
<tr>
<td>DCA Web Environment</td>
<td>21-3</td>
</tr>
<tr>
<td>Clipboard</td>
<td>21-4</td>
</tr>
</tbody>
</table>
Introduction

DCA session objects provide access to the DCA clipboard, the session scope hash map (\textit{anyData}) for storing arbitrary data, and various factories.

The DCA request object provides access to a current frame, current element, and locale; and request scope has a map (\textit{anyData}) for storing arbitrary data.

Use hash map with a request scope for inter-component communication when components are in the same frame. Otherwise, use a session scope.

\textbf{Caution}: Do not store references to DCA elements in hash maps unless you are familiar with DCA implementation. Unpredictable results can occur when hash maps are accidently misused.
DCA Web Environment

The DCA Web environment implements session objects through a JSP session scope bean, and request objects through a request scope bean.
**Caution:** Do not use environment-specific portions of session and request objects unless absolutely necessary.

**Clipboard**

DCA provides an environment-independent clipboard.

Clipboard stores clipboard items. Clipboard items remember addresses of DCA elements placed on the clipboard. This allows the removal of DCA elements when their frames are aged out by the frame caching algorithm.

You will need to resolve addresses into references when you need to access DCA elements from a clipboard.

```java
public void processClippedElements (CoEnvData envData) throws WTException {
    CoClipboard clipboard = envData.getSessionData.getClipboard();
    clipboard.resolveReferences(envData);
    ArrayList items = clipboard.getItems();
}
```
Iterator item_iter = items.iterator();

while (item_iter.hasNext()) {
    CoClipboardItem item = (CoClipboardItem)item_iter.next();
    CoInstanceHolder instance_holder =
        (CoInstanceHolder)item.getInstanceHolder();
    ...
}

clipboard.releaseReferences(envData);

**Clipboard Mode**

Clipboard can be in either clipboard or shopping cart mode. In clipboard mode, placing the next set of elements on a clipboard removes its content first. In shopping cart mode it adds a new set to the existing clipboard content.

---

**CoClipboardMode**

- mode : String
  - CLIPBOARD : CoClipboardMode = new CoClipboardMode("Clipboard")
  - SHOPPING_CART : CoClipboardMode = new CoClipboardMode("ShoppingCart")

---

**Clipboard Reason**

It might be important to distinguish between an intent to copy and an intent to cut. DCA provides so-called clipping reason.

---

**CoClippingReason**

- reason : String
  - COPY : CoClippingReason = new CoClippingReason("Copy")
  - CUT : CoClippingReason = new CoClippingReason("Cut")

---
Clipboard Actions

Cut and Copy Actions

DCA demo provides examples of Cut and Copy actions

```xml
<Action id="action.cut" needSelection="true">
  <Label resource="Cut"/>
  <ToolTip resource="Cut"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/cut.gif"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebSimpleInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.selection.CoClipboardCutDoer"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp"/>
</Action>

<Action id="action.copy" needSelection="true">
  <Label resource="Copy"/>
  <ToolTip resource="Copy"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/copy.gif"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebSimpleInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.selection.CoClipboardCopyDoer"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp"/>
</Action>

Note: While both clipboard copy and cut doers can be used with com.ptc.core.ca.co.client.selection.CoClipboardActionHandler, they can avoid the extra overhead of resolving references by using regular in-place action handler.

Paste Action

Paste action can be configured to execute a doer.

Non-task Doer

```xml
<Action id="action.paste">
  <Label resource="Paste"/>
  <ToolTip resource="Paste"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/paste.gif"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
  <ActionDoer class="doer class"/>
</Action>
```
The action can use an in-place action handler and the doer must extend `com.ptc.core.ca.co.client.selection.CoClipboardPasteDoer` and override `execute(CoSimpleActionHandlerData execData)` method. It must also call `super.execute` at the end.

```java
public void execute (CoSimpleActionHandlerData execData)
    throws WTException {
    CoEnvData envData   = execData.getEnvData();
    CoClipboard clipboard = envData.getSessionData.getClipboard();
    clipboard.resolveReferences(envData);
    ArrayList items     = clipboard.getItems();
    Iterator  item_iter = items.iterator();
    while (item_iter.hasNext()) {
        CoClipboardItem item = (CoClipboardItem)item_iter.next();
        CoInstanceHolder instance_holder =
            (CoInstanceHolder)item.getInstanceHolder();
        ...
    }
    clipboard.releaseReferences(envData);
    super.execute();
}
```

The action can also use `com.ptc.core.ca.co.client.selection.CoClipboardActionHandler` action handler. In this case the doer still has to extend `com.ptc.core.ca.co.client.selection.CoClipboardPasteDoer`, override `execute(CoSimpleActionHandlerData execData)` method, and call `super.execute` at the end. It does not, however, have to resolve references and release them as the handler does it.

```java
public void execute (CoSimpleActionHandlerData execData)
    throws WTException {
    CoEnvData envData   = execData.getEnvData();
    CoClipboard clipboard = envData.getSessionData.getClipboard();
    ArrayList items     = clipboard.getItems();
    Iterator  item_iter = items.iterator();
    while (item_iter.hasNext()) {
        CoClipboardItem item = (CoClipboardItem)item_iter.next();
        CoInstanceHolder instance_holder =
            (CoInstanceHolder)item.getInstanceHolder();
        ...
    }
    super.execute();
}
```
Task Doer

Paste actions can be configured to execute a doer that can trigger the execution of a task.

```xml
<Action id="action.paste">
  <Label resource="Paste"/>
  <ToolTip resource="Paste"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/paste.gif"/>
  <ActionHandler class="com.ptc.core.ca.web.client.selection.WebClipboardActionHandler"/>
  <ActionDoer class="%common:SimpleActionTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="task-delegate_name">
      <Task outParameter function="com.ptc.core.ca.co.client.selection.CoClipboardFunction_Action_Result_CleanClipboard"/>
      <TaskParameter function="com.ptc.core.ca.co.client.selection.CoClipboardFunction_Action_Param_AddClippedAsUfids" name="clipped"/>  
    </Task>
  </ActionDoer>
</Action>
```

An action configured for the execution of a task on Paste must;

- Utilize `com.ptc.core.ca.co.client.selection.CoClipboardActionHandler`,
- Use task parameter function to create parameter containing UFIDs of clipboard objects,
- Use output task parameter function to clean up clipboard by releasing references from it.

Clipboard Functions

The following is a list of clipboard functions DCA implements out of the box in the package `com.ptc.core.ca.co.client.doer.selection`.

**CoClipboardFunction_Action_Param_AddClippedAsUfids**

Prepares a task parameter containing UFIDs of Info*Engine elements that belong to DCA elements placed on the clipboard.

The function does the following:

1. Gets an Info*Engine element from every DCA element on a clipboard
2. For every Info*Engine element adds task parameter with name specified in configuration via name parameter and value containing UFID of the element

The function can use the following parameters:

**name** = `text[CS]`

Contains name of the task parameter to create.
**CoClipboardFunction_Action_Param_AddClippedAsGroup**

Prepares a task parameter containing an Info*Engine Group. The group contains Info*Engine elements that belong to DCA elements placed on the clipboard.

The function does the following:

1. Gets an Info*Engine element from every DCA element on a clipboard,
2. Creates Info*Engine Group consisting of Info*Engine elements,
3. Adds task parameter with name specified in configuration via `name` attribute using newly created group as a value.

The function can use the following parameters:

- **name = text**
  - Contains name of the task parameter to create.

**CoClipboardFunction_Action_Result_CleanClipboard**

Releases references to DCA elements from the clipboard.
The DCA demo provides a mechanism to display the content items of content holder elements, and to expose content URLs for viewing and downloading.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing Content Items</td>
<td>22-2</td>
</tr>
<tr>
<td>Link to Primary Content</td>
<td>22-4</td>
</tr>
<tr>
<td>Download Applet</td>
<td>22-5</td>
</tr>
<tr>
<td>Upload</td>
<td>22-7</td>
</tr>
</tbody>
</table>
Listing Content Items

The following is an example of an action, *List Content Items*. It opens a wizard-based frame to display a table of content items of the related content holder elements.

[common/content.xml]

```xml
<Action id="action.content.list" needSelection="true">
  <Extend ref=":action.newTemporaryFrame.template"/>
  <Label resource="List Content Items" />
  <ToolTip resource="List Content Items: {0} document">
    <Need attribute="name"/>
  </ToolTip>
  <Location ref=":dialog.content.list"/>
</Action>
```

The target frame is a single step wizard containing a table of content items:

[common/content.xml]

```xml
<SimpleFrame id="dialog.content.list" width="750" height="400">
  <Extend ref=":simpleframe.template"/>
  <Label resource="List Content Items"/>
  <Wizard>
    <Label resource="List Content Items"/>
    <WizardStepElements>
      <Insert ref="layout.content.list"/>
    </WizardStepElements>
    <ModelDoer class="com.ptc.core.ca.co.client-picker.CoNullModelDoer"/>
    <Insert ref=":wizard.actions"/>
  </Wizard>
</SimpleFrame>

<SimpleLayout id="layout.content.list">
  <LayoutRow id="1">
    <Insert ref="table.content.list"/>
  </LayoutRow>
</SimpleLayout>

<SimpleTable id="table.content.list" selectorType="multi" mode="view">
  <SelectAllPrompt resource="All"/>
  <Type type="wt.content.ApplicationData"/>
  <Type type="wt.content.URLData"/>
  <Type type="wt.content.Aggregate"/>
  <SimpleColumn show=":property.content.fileName">
    <Label resource="File Name"/>
  </SimpleColumn>
  <SimpleColumn show=":property.content.fileSize">
    <Label resource="File Size"/>
  </SimpleColumn>
  <SimpleColumn show=":property.content.url">
    <Label resource="URL"/>
  </SimpleColumn>
</SimpleTable>
```
The content items can be instances of those three types:

- `wt.content.ApplicationData`
- `wt.content.URLData`
- `wt.content.Aggregate`

The following model doer initializes the table:

```xml
<ModelDoer id="content.list.model.doer"
  class="com.ptc.core.ca.co.client.doer.CoDefaultSimpleModelTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.view"
        taskDelegate="dca-list-content-items">
    <TaskParameter name="attribute" value="fileName"/>
    <TaskParameter name="attribute" value="fileSize"/>
  </Task>
</ModelDoer>
```

The configuration file contains several properties that are applicable for the content item types, like `fileName`, `role`, `isPrimary`, etc. Those properties are rendered as standard property types. The only exception is `property.content.url` property:

```xml
<Property id="property.contentUrl.template" window="new">
  <Label resource="URL:"/>
  <PropertyHandler class="com.ptc.core.ca.web.client.property.string.WebStringPropertyHandler"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/property/hyperlink/contentUrl.jsp"/>
</Property>
```

```xml
<Property id="property.content.url">
  <Extend ref="property.contentUrl.template"/>
  <Need attribute="urlLocation"/>
</Property>
```

This is a String type of property, but since it represents a URL that the content can be reached from, the `contentUrl.jsp` renderer is used to allow viewing and downloading functionality. The renderer creates a link to the content item URL, using the file name of the content item as a label. Clicking on the link will open the file, for MIME types that the browser understands. The file can be downloaded like any hyperlink content (click the right mouse button and select Save Target As...). The `window` attribute of the `Property` tag provides the configuration of the target frame for a content item to be loaded in. The value, `new`, opens the content in the new window, while other values including default opens it in the same window.
Link to Primary Content

The DCA Demo provides an example of a table configuration when the link to primary content is presented in the table row containing a content holder element.

```
<SimpleTable id="table.contentItem" selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Insert ref="browser.actions"/>
  <Type type="%[part:WTPart]"/>
  <Type type="%[doc:WTDocument]"/>
  <Type type="%[SubFolder]"/>
  <SimpleColumn show=":name">
    <Label resource="Name"/>
    <For type="%[SubFolder]" show="action.folder.view"/>
  </SimpleColumn>
  <SimpleColumn show=":number">
    <Label resource="Number"/>
  </SimpleColumn>
  <SimpleColumn show=":contentItemUrl">
    <Label resource="Content"/>
  </SimpleColumn>
  ...
  <ModelDoer resultSetType="multiple"
    class="%common:SimpleModelTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
      taskDelegate="dca-list-folder-members-with-content">
      <TaskParameter name="content_type" value="WCTYPE|wt.folder.SubFolder"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.doc.WTDocument"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.part.WTPart"/>
    </Task>
  </ModelDoer>
</SimpleTable>
```

Content items are Info*Engine elements. They are not attributes of content holders. In order to display a content link inside the same table row of its content holder, the table model doer executes a `dca-list-folder-members-with-content` task.

This task creates an additional attribute `contentItemUrl` to every content holder element, and sets the first content item URL as attribute's value. Due to the performance issue, this task should be used only if a content item and a content holder are required to be in the same table.
**Property Based**

A property element can be configured to present a link to content.

```xml
<Property id="contentItemUrl">
  <Extend ref="common:property.contentUrl.template"/>
  <Label resource="Content:"/>
  <Need attribute="contentItemUrl"/>
</Property>
```

This property is handled and rendered in the same manner as the property.content.url property of a content item described above. The window attribute of the Property tag provides the configuration of the target frame for a content item to be loaded in. The value new opens the content in the new window, while other values including default opens it in the same window.

**Hyperlink Based**

A hyperlink element can be configured instead of the property.

```xml
<HyperLink id="contentItemUrl" labelType="text" drawType="text">
  <Label resource="{0}"
    drawType="text">
    <Need attribute="name" />
  </Label>
  <Need attribute="contentItemUrl" />
  <HRef resource="%{attribute#..#contentItemUrl}" />
</HyperLink>
```

**Download Applet**

DCA Web can be configured to download content using an applet through the Plugin element. Plugin needs to be configured to load the standard wt.clients.util.http.DownloadApplet applet that is used by core Windchill.

Example:

```xml
[com/ptc/common/plugin.xml]

<Plugin id="applet.download">
  <Extend ref="applet.template"/>
  <Option theme="attribute" param="name" value="formApplet"/>
  <Option theme="parameter" param="java_code" value="%AppletDownload"/>
  <Option theme="parameter" param="cache_archive" value="%CasheArchive"/>
  <ModelDoer class="com.ptc.core.ca.web.client.content.WebContentDownloadModelDoer"/>
</Plugin>
```
In the DCA demo, downloading content using the applet is available from the Content Items dialog. The Download action is an item in the content item's action list:

[com/ptc/common/content.xml]

```
<ActionList id="content.actionList">
  <Insert ref="action.download"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/dropdownlist.jsp"/>
  <Prompt resource="- Pick an Action - &nbsp;&nbsp;"/>
</ActionList>

<Action id="action.download" needSelection="true">
  <Label resource="Download Content"/>
  <ToolTip resource="Download Content"/>
  <Location ref=":dialog.download"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
  <ActionDoer class="com.ptc.core.ca.co.client.plugin.CoPluginGoToActionDoer"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp"/>
</Action>
```

This is an in-place action. However, the doer issues an instruction to DCA to load the frame specified by the Location tag. That frame contains the applet:

[com/ptc/common/content.xml]

```
<SimpleFrame id="dialog.download" width="750" height="400">
  <Extend ref=":simpleframe.template"/>
  <Label resource="Download"/>
  <SimpleLayout>
    <LayoutRow>
      <Insert ref="applet.download"/>
    </LayoutRow>
  </SimpleLayout>
</SimpleFrame>
```

After the download, the original frame containing the download action is reloaded in the same window.

**Doers**

- `com.ptc.core.ca.co.client.plugin.CoPluginGoToActionDoer`
  
  This doer is responsible for the loading of a frame with the address specified through the Location element. The action holder of an action using this doer will be an opener action holder for the new frame.

- `com.ptc.core.ca.web.client.content.WebContentDownloadModelDoer`

  This doer is responsible for the initialization of parameters for the `wt.clients.util.http.DownloadApplet` applet.
The DCA demo provides upload content item functionality through the wt.clients.util.http.UploadApplet applet. See a document describing the applet for the description of its values. The following is a demo configuration:

```
[com/ptc/common/plugin.xml]:
<Plugin id="applet.upload">
  <Extend ref="applet.template"/>
  <Option theme="attribute" param="name" value="formApplet"/>
  <Option theme="parameter" param="java_code" value="\%[AppletUpload]\""/>
  <Option theme="parameter" param="java_archive" value="\%[SecurityJar]\""/>
  <Option theme="parameter" param="cache_archive" value="\%[DocMgrJar]\""/>
  <Option theme="parameter" param="callingAction" value="create"/>
  <Option theme="parameter" param="MAYSCRIPT" value="true"/>
  <Option theme="parameter" param="SCRIPTABLE" value="true"/>
  <Option theme="parameter" param="uploadIfFileChanged" value="true"/>
  <Option theme="parameter" param="continueIfFileNotFound" value="true"/>
  <Option theme="parameter" param="continueIfFileUnchanged" value="true"/>
  <Option theme="parameter" param="removable" value="true"/>
  <Option theme="parameter" param="checksum" value=""/>
  <ModelDoer class="\%[ContentUploadModelDoer]\">
    <Task nextOperation="STDOP|com.ptc.windchill.create"
      taskDelegate="dca-get-upload-url">
      <TaskParameter function="com.ptc.core.ca.co.client.doer.
        function.impl.CoFunction_Model_Param_FudgeElement"/>
    </Task>
  </ModelDoer>
</Plugin>
```

The model doer of a plug-in element executes the dca-get-upload-url task in order to get the upload URL of the related content holder, which is a required applet parameter. It also sets all of the dynamic applet parameters.
Uploading from a List of Content Items

In DCA Demo, content items can be uploaded from the Content Items dialog. The Upload Form wizard tab holds the file chooser panel and the Upload action. Its layout configuration is shown below:

```xml
<SimpleLayout id="layout.upload.form">
  <LayoutRow id="1">
    <Insert ref=":property.uploadPath" mode="create"/>
    <Insert ref="applet.upload.fileChooser"/>
    <Insert ref="applet.upload.dndMicro"/>
  </LayoutRow>
  <LayoutRow id="2" ifnot.mode="create">
    <Insert ref="action.upload"/>
  </LayoutRow>
</SimpleLayout>
```

The uploadPath property uses NPA, created to handle the local path parameter input:

```xml
<context type="wt.content.ContentHolder">
  <property id="property.uploadPath">
    <extend ref="property.uploadPath.template"/>
  </property>
</context>
```

```xml
<property id="property.uploadPath.template">
  <label resource="Upload path:"/>
  <need attribute="uploadPath"/>
  <propertyHandler class="%[ContentUploadPathPropertyHandler]"/>
  <renderer resource="/wtcore/jsp/com/ptc/core/ca/web/content/uploadPath.jsp"/>
</property>
```

The user chooses a file by browsing the local file system and selecting a file or by dragging and dropping a file into the DnD micro applet. The applet is embedded in the layout:

```xml
<plugin id="applet.upload.fileChooser">
  <extend ref="applet.template"/>
  <option theme="attribute" param="classid" value="%[ClassID]%"/>
  <option theme="attribute" param="name" value="formApplet"/>
  <option theme="attribute" param="width" value="100"/>
  <option theme="attribute" param="height" value="30"/>
  <option theme="attribute" param="codebase" value="%[URLGetPlugin]%"/>
  <option theme="parameter" param="java_archive" value="%[SecurityJar]%"/>
  <option theme="parameter" param="java_code" value="%[AppletFileLocator]%"/>
</plugin>
```
The Upload action is an in-place action that redirects the request to the page specified by the Location tag. That page holds the applet that performs the file uploading:

```
<com/ptc/common/content.xml>
<Action id="action.upload" needSelection="true">
  <Label resource="Upload Content"/>
  <ToolTip resource="Upload Content"/>
  <Location ref=":dialog.upload"/>
  <ActionHandler class="%[InPlaceActionHandler]"/>
  <ActionDoer class="%[PluginGoToActionDoer]"/>
  <Insert ref="common:renderer.action"/>
</Action>
```

The page configuration is:

```
[com/ptc/common/content.xml]
<SimpleFrame id="dialog.upload" width="750" height="400">
  <Extend ref=":simpleframe.template"/>
</SimpleFrame>
```
When finished, the control goes back to the Content Items dialog. The action is currently implemented as in-place, in order to easier set the URL parameter of the applet to the location of the wizard the user triggers upload from. After successful completion, the Content Items page is displayed again.

Since the applet’s (not DCA’s) specification is not final yet, it is recommended that the user use the parameter and attribute values as is unless the user is familiar with the meaning of them.

**Upload from Create Content Holder Wizard**

In DCA Demo, the implementation of Create Document wizard lets a user specify and upload the primary content at the time a document is created. The following is a configuration fragment for the wizard:

```xml
[com/ptc/common/config_doc.xml]

<SimpleFrame id="dialog.create" mode="create"
    width="600" height="400">
    <Extend ref="common:simpleframe.template"/>
    <Insert ref="common:script.content"/>
    <Label resource="Create Document"/>
    <Wizard>
        <Label resource="Create Document"/>
        <WizardStepActions>
            <Insert id="step.1" ref="common:action.wizard.goto">
                <Label resource="Properties"/>
                <ToolTip resource="Properties"/>
                <Insert ref="common:renderer.level1_tab"/>
            </Insert>
            <Insert id="step.2" ref="common:action.wizard.goto">
                <Label resource="Upload Form"/>
                <ToolTip resource="Upload Form"/>
                <Insert ref="common:renderer.level1_tab"/>
            </Insert>
            <Insert ref="common:renderer.wizardTab"/>
        </WizardStepActions>
        <WizardStepElements>
            <Insert ref="layout.create"/>
            <Insert ref="common:layout.upload.form"/>
        </WizardStepElements>

        <ModelDoer class="com.ptc.core.ca.co.client.doer.
            CoDefaultModelTaskDoer">
            <Task nextOperation="STDOP|com.ptc.windchill.create"
                taskDelegate="dca-create-model">
                <TaskParameter function="com.ptc.core.ca.co.client.doer.
                    function.impl.CoSimpleFunction_Model_Param_MainObjectIdentity"/>
```
The Upload functionality is configured as the second wizard tab. Creation of the document requires task execution. It is then followed by the redirection to the applet page. This complex action prevents standard wizard actions from being used in the list of wizard actions. New actions need to be configured instead.

```
<WizardActions id="wizard.upload.actions">
  <Insert ref="action.wizard.previous"/>
  <Insert ref="action.wizard.next"/>
  <Insert ref="action.wizard.upload.ok"/>
</WizardActions>
```
<Insert ref="action.wizard.upload.apply"/>
<Insert ref="action.wizard.cancel"/>
<Insert ref="renderer.horizontalList"/>
</WizardActions>

<Action id="action.wizard.upload.template">
  <Location ref="dialog.upload.close"/>
  <ActionHandler class="com.ptc.core.ca.web.client.wizard.
  WebApplyAndGoActionHandler" emptyFrame="false"/>
  <Insert ref="renderer.wizard_ok_apply"/>
  <On event="executeActionDoer" at="beginning">
    <Copy from="%[attribute#Wizard#uploadPath]" to="%[frame#uploadPath]/">
  </On>
  <On event="executeActionDoer" at="end">
    <Copy from="%[frame#uploadPath]" to="%[attribute#Wizard#uploadPath]/">
  </On>
</Action>

<Action id="action.wizard.upload.ok">
  <Extend ref="action.wizard.upload.template"/>
  <Label resource="Ok"/>
  <ToolTip resource="Ok"/>
  <Location ref="dialog.upload.close"/>
</Action>

<Action id="action.wizard.upload.apply">
  <Extend ref="action.wizard.upload.template"/>
  <Label resource="Apply"/>
  <ToolTip resource="Apply"/>
  <Location ref="dialog.upload"/>
</Action>

Now both OK and Apply actions use WebApplyAndGoActionHandler. This action handler does everything that the regular WebApplyActionHandler does, but after the execution of the operation it issues an instruction to redirect a window to the location containing the upload applet. For the applet, the OK action sets the URL to the upload frame that closes itself after the applet execution, while the Apply action sets the URL to the upload frame that goes to the initial wizard page after the applet execution.

To support the correct applet initialization, the action doer needs to perform some additional work after the task is executed. It is achieved by the task output function.

<TaskOutParameter function="com.ptc.core.ca.co.client.doer.
  function.impl.CoFunction_Action_Result_SetInstanceToInstanceHolder" />

The function makes sure that the task output element is set as an instance to the actions instance holder ancestor (wizard). The upload applet considers this element as a content holder.
The following function creates an NPA with the configured name to the task output element. This attribute will hold the path of the file to be uploaded as the primary content item.

```xml
<TaskOutParameter function="com.ptc.core.ca.co.client.doer. function.impl.CoFunction_Action_Result_CreateNpa"
name="uploadPath"/>
```

Frames `dialog.upload` and `dialog.upload.close` are both configured to include the plugin element. The plugin on the former frame uploads a file and changes location back to the original wizard. The plugin on the later frame changes location to the frame that closes the dialog window.

```xml
<Plugin id="applet.upload.close">
  <Extend ref="applet.upload"/>
  <ModelDoer class="%[ContentUploadModelDoer]">
    <Location status="success" ref="com.ptc.common:dialog.close"/>
    <Location status="failure" ref="com.ptc.common:dialog.close"/>
    <Task nextOperation="STDOP|com.ptc.windchill.create" taskDelegate="dca-get-upload-url">
      <TaskParameter
        function="com.ptc.core.ca.co.client.doer.function. impl.CoFunction_Model_Param_FudgeElement"/>
    </Task>
  </ModelDoer>
</Plugin>
```

**Upload from Edit Properties Wizard**

The current implementation of the Edit Properties wizard dialog allows the user to set a primary content in addition to some document property changes. The configuration fragment for the wizard dialog is:

```
[com/ptc/common/config_doc.xml]
```

```xml
<SimpleFrame id="dialog.edit.task" mode="edit"
width="750" height="400">
  <Extend ref="common:simpleframe.template"/>
  <Insert ref="common:script.content"/>
  <Label resource="Edit Properties"/>
  <Wizard>
    <Label resource="Document Edit Wizard"/>
    <WizardStepActions>
      <Insert id="step.1" ref="common:action.wizard.goto">
        <Label resource="Properties"/>
        <ToolTip resource="Properties"/>
      </Insert>
```

```
```
The upload form is a part of the first wizard tab, while the content items table is in the second wizard tab. This wizard doesn't differ from the Create Document wizard in the context of content upload. But it uses a task validator since there is no need to execute the update task if no attributes are changed.
Links and Associations

Link objects connect two business objects by keeping their references, thus establishing a link between them. On the other hand, the association between two business objects is usually done via a foreign key.

For the purpose of this chapter, we will sometimes ignore the difference between a link and an association, and will use the term relationship for both unless stated otherwise. We will also call a linked or associated object the related object.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit Links Functionality</td>
<td>23-2</td>
</tr>
<tr>
<td>Action on Related Object</td>
<td>23-5</td>
</tr>
<tr>
<td>Hyperlink to Related Object</td>
<td>23-7</td>
</tr>
</tbody>
</table>
Edit Links Functionality

DCA provides support for the typical task of editing links. The DCA demo demonstrates this in the example, editing the uses links for WTPart objects located in com\ptc\part\part_edit_link.xml. The example presents a list of \texttt{uses} links as a table on one of the steps of the WTPart's "Edit Property" wizard. Each link’s attributes, \texttt{quantity\_unit} and \texttt{quantity\_amount}, are shown as well as the attributes of the other side objects, like \texttt{name} and \texttt{number}.

In the demo, the user can update a list of links. Specifically, the user can do the following:

1. Update a link's attributes
2. Remove a link from the list of the links
3. Add a new link to the list of the links

Modifications performed in the link table are sent to the data sources only when the user clicks the \textbf{OK} or \textbf{Apply} buttons.

How to Configure

To implement the edit link functionality configure the following components:

1. A table to contain link objects. There is nothing special about configuring it for editing links. The only difference is that it uses the reference to the link properties in a form \texttt{uses\^:name}. See the detailed description below for reference.

2. A picker to allow the user to pick several WTPart objects. This picker has a special doer for \textbf{OK} button,\texttt{com.ptc.core.ca.co.client-picker.CoObjectListForLinkPickerAddOkActionDoer}, that creates the Info*Engine element representing WTPartUsageLink object for every picked WTPart object. It also adds these new elements to the table of existing links.

3. A task (\texttt{dca-edit-links.xml} in the demo) that gets a \textit{current} list of links as an input parameter. This task performs changes in the database to make the links in the database consistent with the link table updated by the user. To find out which links should be created, updated, and deleted the task uses the method \texttt{prepareLinkInfoForOperationsCreateUpdateDelete} from \texttt{com.ptc.core.ca.co.client.doer.CoDoerHelper} class.
Referenced Elements

Frequently, link tables include properties and other elements of the other side object. Use the following syntax to refer to them:

```
reference_attribute^referenced_element
```

For instance, to refer to a `name` property of the used part, you need to specify:

```
uses^name
```

When a property is configured in a different package, use a colon (`:`) or a full name to indicate that as follows:

```
uses^:name
```

The syntax described above was specifically introduced to enable the sharing of configuration specifications for properties of referenced elements. For instance, the following would refer to the same property on the other side object for a different link:

```
usedBy^:name
```

While any reference can be specified in this form, only property references are currently supported. For property reference, `uses^:name` means the reference to the property and `:name` of the object, referenced by reference attribute `uses` of a given object. The following is an example of how such a reference is used in the Edit Property wizard for WTPart:

```xml
<SimpleTable id="table.edit.usage" selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Insert ref="pick.action.list.part.for.edit"/>
  <Type type="wt.part.WTPartUsageLink"/>
  <SimpleColumn show="uses^:number">
    <Label resource="Number"/>
  </SimpleColumn>
  <SimpleColumn show="uses^:name">
    <Label resource="Name"/>
  </SimpleColumn>
  <SimpleColumn show=":quantity.amount">
    <Label resource="Amount"/>
  </SimpleColumn>
  <SimpleColumn show=":quantity.unit">
    <Label resource="Unit"/>
  </SimpleColumn>
</SimpleTable>
```

The `ref` attribute of the `Insert` or `Extend` elements, as well as the `show` attribute of the `SimpleColumn` element, specify the reference to a configuration element. This reference can be specified, for example, in a form like `name`, `:name` or `com.ptc.part:name`. All the forms can benefit from the syntax described above.

For any reference attribute used in reference like `uses^:name` (uses attribute of WTPartUsageLink) DCA requires you to specify a type of the referenced object. DCA needs it to locate the configuration for the specified property (name
property) of the referenced object. The type should be specified in the following form:

```xml
<Context type="wt.part.WTPartUsageLink">
  <ReferenceAttribute id="uses" type="wt.part.WTPart"/>
</Context>
```

**Caution:** DCA cannot now use the real type of the referenced object to locate a correct property definition for the exact type. For instance, for the `uses` attribute of WTPartUsageLink the type of a referenced object is defined as WTPart. However, the referenced object can be of type CoolPart that is a subtype of WTPart. Even when `name` property is configured differently for CoolPart, it will be presented using the definition of a property for WTPart.

### Picker Doer

DCA provides a client doer for the Ok button in the picker window that adds new link objects to the list of links. The doer is implemented by the class.

```java
com.ptc.core.ca.co.client.picker.CoObjectListForLinkPickerAddOkActionDoer
```

The doer does the following:

- iterates through a list of picked objects (of WTPart type in the demo)
- for every object creates an Info*Engine element representing a link object (WTPartUsageLink in the demo) pointing to the object
- adds newly created link object to the table of links

The doer requires the following parameters:

1. The `otherSideAttribute` attribute to specify a name of the reference attribute (`uses` in the demo).

   ```xml
   <ClientDoer class="com.ptc.core.ca.co.client.picker.CoObjectListForLinkPickerAddOkActionDoer" otherSideAttribute="uses"/>
   ```

2. A task specification to configure a task to be executed to create an Info*Engine element representing a new link object. The following is from the demo:

   ```xml
   <Task nextOperation="STDOP|com.ptc.windchill.create"
      taskDelegate="dca-create-model">
     <TaskParameter name="dca_include_args" value="true"/>
     <TaskParameter name="dca_include_constraints" value="true"/>
     <TaskParameter name="dca_include_descriptors" value="true"/>
     <TaskParameter name="target_type" value="WCTYPE|wt.part.WTPartUsageLink"/>
     <TaskParameter name="dca_field" value="quantity.unit=ea"/>
     <TaskParameter name="dca_field" value="quantity.amount=2.0"/>
     <TaskParameter name="dca_attribute" value="quantity.amount"/>
     <TaskParameter name="dca_attribute" value="quantity.unit"/>
   </Task>
   ```
Recognizing Link Changes

DCA provides a method, `prepareLinkInfoForOperationsCreateUpdateDelete`, in the `com.ptc.core.ca.co.client.doer.CoDoerHelper` class, to propagate changes in the link table to update the data source.

The method compares a list of links currently existing in a data source links with a list of links updated by user and prepares three sets - a set of links to be created, a set of links to be deleted, and a set of links to be updated.

```java
public static Group prepareLinkInfoForOperationsCreateUpdateDelete(Group new_links, Group old_links)

• new_links
  a set of links from the table of links after a user’s manipulations.

• old_links
  a set of links retrieved from the database.
```

After the execution of the method

• `new_link` will contain a set of links to create in the data source.

• `old_links` will contain a set of links to delete in the data source.

• return Group will contain a set of links to update in the data source.

Use the method in a task that updates links. See `dca-edit-links` task in the DCA demo.

Action on Related Object

An address of the frame usually includes type and UFID of the object the frame represents. When an action uses a simple approach, the type and UFID of an appropriate instance holder or frame are used to create an address of the target frame. Sometimes it is not desirable when, for instance, we’d like to configure an action to view the properties of a related object. It would certainly be desirable to have type and UFID of that object be in the address of the frame displaying its properties.

DCA provides an ability to configure an action to use type and UFID of the related object when simple approach is employed. The configuration must indicate the attributes to be used as type and UFID via `as` attribute. The following is an example of configuring an action to present the properties of a lifecycle of a document.

```xml
<config-base>/com/ptc/doc/config_doc.xml

<Action id="action.lifecycle.view">
  <Extend ref="common:action.newTemporaryFrame.template"/>
  <Label resource="Info"/>
```
The Need configuration elements indicate that the drawing of the action requires two attributes in the Info*Engine element that represent type and UFID of the related object. In the example above `state.lifeCycleId` defines a link and `obid` and `class` indicate attribute names of the attributes of the related object.

**Caution:** Due to current limitation in the command layer, actions can be configured to present object on the object related by an association only. Links are not yet supported using this syntax.

As usual, a simple task doer include all of the attributes defined by the Need configuration elements as parameters of its I*E task.

Sometimes an Info*Engine task returns the attribute not under the name it was requested. This is due to way Info*Engine communicates with the command subsystem. For instance, when DCA asks for an attribute `folder.id` of the `wt.folder.Folder`, it comes back under the name `folder`. To get around this problem, an additional property is introduced for the Need configuration element. The property `accessibleByName` is strictly optional.

When the attribute to be used as type of UFID is NPA (non-persistent attribute), it should not be included into task parameters. Since this precludes Need element from being used, NeedNpa configuration element is introduced to be able to specify the NPA to be used as type or UFID.

The following defines a go-to action to navigates to the parent folder in the folder browser. As due to the current limitation, `folder.id^class` cannot be retrieved from the task result, the attribute is created by a function.

```xml
<Action id="action.parentFolder.view">
  <Extend ref="common:action.goTo.template"/>
  <Label resource="Up"/>
  <ToolTip resource="Open parent folder"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/up.gif"/>
  <Location ref=":page.view">
    <NewContext>
      <Copy from="%[attribute#$folder.id^class]" to="%[context#type]"/>
    </NewContext>
  </Location>
  <Need attribute="folder.id" accessibleByName="folder" as="ufid"/>
</Action>
```
The following fragment shows how the creation of an NPA is configured:

```
<config-base>/com/ptc/folder/config_folder.xml

<PropertyPanel id="panel.view.task" type="%[SubFolder]">
  <LayoutRow id="1">
    <Insert ref="action.parentFolder.view"/>
  </LayoutRow>
  <LayoutRow id="2">
    <Insert ref="currentName"/>
  </LayoutRow>
  <ModelDoer class="%common:SimpleModelTaskDoer">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
          taskDelegate="dca-get-single-object">
      <TaskParameter name="format" value="true"/>
      <TaskOutParameter function="com.ptc.core.ca.co.client.windchill.CoFunction_Model_Result_CreateTypeAttributeUsingUfid"
                         from="folder" attribute="folder.id^class"/>
    </Task>
  </ModelDoer>
  <Insert ref="common:renderer.propertyPanel"/>
</PropertyPanel>
```

The specified function creates an NPA and initializes its value extracting type information from the UFID of a Windchill object. This makes the function purely Windchill specific.

The `from` parameter holds the name of the attribute that contains the UFID of related object, parent folder in the example above. The string representing the object’s type is extracted from UFID and is assigned to the newly created NPA whose name is defined by `attribute` parameter.

### Hyperlink to Related Object

Hyperlinks to related Windchill objects can be configured similarly for actions to related objects when hyperlink producers utilizing simple approach are in use. Use `Need` or `NeedNpa` configuration elements to indicate the attribute containing UFID. An attribute containing type is not required as Windchill OID includes type information.
DCA uses Windchill preference mechanism to store preferences like page size, column order, etc. At the moment DCA works with user preferences only. They are stored using preference node `/com/ptc/core/ca/co/DCAType.Preference` with a unique key.

DCA preferences are accessible via configuration specifications. To optimize performance, DCA uses a pass-through cache of preference values. DCA Demo provides a simple preference cache viewer that displays the cache content in a table. The viewer's frame can be accessed via the alias `com.ptc.preferences:preference.cache`. 

Preferences

24
The configuration of the preference viewer’s frame is shown below:

[com/ptc/preferences/config_pref.xml]

```xml
<Synonym id="DCAPref" value="DCAType.Preference"/>

<SimpleFrame id="preference.cache"
  type="%[DCAPref]"
  mode="view"
  width="750" height="600">
  <Extend ref="common:simpleframe.template"/>
  <Label resource="Cache Preferences"/>
  <Insert ref="preference.cache.layout"/>
</SimpleFrame>

<Layout id="preference.cache.layout">
  <LayoutRow id="lr.0">
    <Insert ref="preference.cache.table"/>
  </LayoutRow>
</Layout>
```
In order to populate the table, a special table model doer was created that iterates the preference cache and for every entry creates an I*E element with two attributes. The attributes store a key and a value of a preference entry:

```xml
<SimpleTable id="preference.cache.table">
  <Type type="%[DCAPref]"/>
  <Insert ref="table.preference.cache.actionList"/>
  <SimpleColumn show="preference.key">
    <Label resource="Preference Key"/>
  </SimpleColumn>
  <SimpleColumn show="preference.value">
    <Label resource="Preference Value"/>
  </SimpleColumn>
  <ModelDoer class="com.ptc.core.ca.co.common.prefs.CoTableCachePreferencesModelDoer"/>
</SimpleTable>

<Property id="preference.key">
  <Extend ref="common:property.String.template"/>
  <Label resource="Preference Key"/>
  <NeedNpa attribute="preference.key"/>
</Property>

<Property id="preference.value">
  <Extend ref="common:property.String.template"/>
  <Label resource="Preference Value"/>
  <NeedNpa attribute="preference.value"/>
</Property>

A user can clear the preference cache by triggering *Clear* action that is located in the table header:

```xml
<Action id="preference.cache.action.clear">
  <Label resource="Clear"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/clear.gif"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
  <ActionDoer class="com.ptc.core.ca.co.common.prefs.CoClearCachePreferenceActionDoer"/>
</Action>
```

```xml
<ActionList id="table.preference.cache.actionList" actionListType="horizontal">
  <Insert ref="preference.cache.action.clear"/>
</ActionList>
```
DCA provides a mechanism to internationalize DCA-based applications via the internationalization of configuration specifications. DCA also provides an internationalization utility that can be used to automate the creation of entries in .rbInfo files.

**Caution:** At the moment, manually created .rbInfo files cannot be automatically regenerated.

**Caution:** Keys in automatically generated .rbInfo files must not be modified or added manually.

DCA employs standard Windchill localization process.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internationalization of Configuration Specifications</td>
<td>25-2</td>
</tr>
<tr>
<td>Manual Creation of Entries in .rbInfo Files</td>
<td>25-3</td>
</tr>
<tr>
<td>Automatic Creation of Entries in .rbInfo Files</td>
<td>25-4</td>
</tr>
</tbody>
</table>
Internationalization of Configuration Specifications

Any attribute in the configuration specification can be internationalized. While mostly resource attributes will be internationalized, this feature is not limited to them.

**Caution:** While any attribute can be internationalized, entries in .rbInfo files for attributes that are expected to be translated must be kept separate from the other entries. This is to prevent an accidental translation of attribute values, like a resource of a Renderer when different renderers are specified for different locales.

To internationalize an attribute it’s value must be of the following syntax:

```
attribute_value::="<value>[#<comment>#]<rbInfoFile>:<key>
```

- **value = text** [CS]
  The value of the key for the default .rbInfo file. Must be specified by an application developer only when an entry in the .rbInfo file is generated automatically.

- **comment = text** [CS]
  The value of the comment for the default .rbInfo file. Can be specified by an application developer only when an entry in the .rbInfo file is generated automatically.

- **rbInfoFile = text** [CS]
  The full class name of the resource bandle class generated from .rbInfo file. For instance `refimpl.co.generic.common.res.dca`. Must be specified by an application developer only when an entry in .rbInfo file is created manually.

- **key = text** [CS]
  The value of the key for the .rbInfo file. Must be specified by an application developer only when an entry in .rbInfo file is created manually.

The following are examples of some attribute values.

- The value ready for the automatic generation of .rbInfo file:
  ```xml
  <ToolTip resource="View Properties:(0)document[#ToolTip for an action to view properties of a document#]"/>
  ```

- The value after the automatic generation of .rbInfo file:
  ```xml
  <ToolTip resource="View Properties:(0)document[#ToolTip for an action to view properties of a document#]com.ptc.refimpl.co.generic.common.res.dca:2"/>
  ```

- The value for the entry in .rbInfo file created manually:
  ```xml
  <ToolTip resource="[##]om.ptc.refimpl.co.generic.common.res.dca:VIEW_PROPS"/>
  ```
Manual Creation of Entries in .rbInfo Files

To manually create an entry in .rbInfo file perform the following two steps.

Step 1

Locate internationalizable string in the configuration specification and change it to be as follows:

```attribute_value::="[##]<rbInfoFile>:<key>
```

For instance, the following configuration fragments;

```<ToolTip resource="View Properties:{0}document"/>
```

becomes;

```<ToolTip
    resource="[##]om.ptc.refimpl.co.generic.common.res.dca:
    VIEW_PROPS"/>
```

While a value and a comment can be legally specified in the configuration specification, DCA will ignore them unless resource bundle can not be found or has problems returning a localized string. In such a case DCA will use the value of the attribute from the configuration specification.

```[##]om.ptc.refimpl.co.generic.common.res.dca:VIEW_PROPS```

Step 2

Add an entry to the specified .rbInfo file:

```<key>.Value=<localized value>
<key>.Comment=<Comment for the entire value>
<key>.ArgComment0=<Comment for argument 0>
<key>.ArgComment1=<Comment for argument 1>
...
```

For instance,

```VIEW_PROPS.Value=View Properties:{0} document
VIEW_PROPS.Comment=Tooltip for an action to view a document
VIEW_PROPS.ArgComment0=Name of the document to view```

**Note:** Different entries in the same configuration file can specify keys in different .rbInfo files. Thus a set of configuration files can share the same .rbInfo file and a single configuration file can use a set of .rbInfo files.
Automatic Creation of Entries in .rbInfo Files

To automatically create an entry in .rbInfo file, perform the following three steps.

Step 1

Indicate a location of .rbInfo file in the configuration specification by adding the `rbInfo` attribute to the `Repository` element

```xml
<Repository package="com.ptc.folder"
           rbInfo="com.ptc.refimpl.co.generic.common.res.dca"/>
```

Step 2

Mark strings as internationalizable by adding four trailing characters `##` to them.

```xml
attribute_value::="<value>[##]"/>
```

For instance, the following configuration fragments;

```xml
<Label   resource="Number"/>
<ToolTip resource="View Properties: {0} document">
```

become;

```xml
<Label   resource="Number[##]"/>
<ToolTip resource="View Properties: {0} document[##]">
```

**Note:** Not only resources, but any strings in configuration specifications can be internationalized.

**Note:** When the same localizable string is used more than once in the configuration specification the use of synonyms is recommended to avoid code duplication.

```xml
<Synonym id="step1" value="Step One[##]"/>
<Label   resource="%[synonym#step1]"/>
<ToolTip resource="%[synonym#step1]"/>
```

**Note:** It is perfectly safe to mark more than one identical string as internationalizable, as DCA utility will create a unique entry in .rbInfo file for every occurrence.
Step 3

Once configuration files are updated to include internationalization information, DCA internationalization utility can create and update .rbInfo files. It’s syntax is as follows:

```
java com.ptc.core.ca.co.common.config.localization.Localization Helper
<xml-files-root-directory> <resource-file-root-directory>
```

where,

**xml-files-root-directory**
the root directory of the configuration repository

**resource-file-root-directory**
the root directory of the repository containing resource bundles

For example:

```
java com.ptc.core.ca.co.common.config.localization.Localization Helper
S:\Windchill\codebase\registry\dca   S:\Windchill\src
```

This utility will find all strings marked as internationalizable in all XML configuration files. For every marked string, it will;

1. Create an entry in the .rbInfo file in a location specified via rbInfo attribute with a unique key. This may result in a creation or update of an .rbInfo file.

2. Update the internationalizable string in the configuration specification to reflect an entry in .rbInfo file by adding an information about .rbInfo file and a key. The information will be added at the end of the string.

For the example above, the following entries will be created in the dca.rbInfo file located in S:\Windchill\wrc\com\ptc\refimpl\co\generic\common\res:\

```
1. value=Number
2. value=View Properties: {0} document
```

The configuration file will be updated to include references to resource bundles:

```
<Label resource="Number[##]com.ptc.refimpl.co.generic.common.res.dca:1"/>
<ToolTip resource="View Properties:{0}document[##]com.ptc.refimpl.co.generic.common.res.dca:2"/>
```

The highlighted numbers are the unique keys that the utility generated by comparing resource bundles with configuration specification.
Note: When values are updated in the configuration specifications, running internationalization utility propagates updates to .rblInfo files automatically. For instance, if label changes from Number to Part:

```xml
<Label resource="Part Number[##]com.ptc.refimpl.co.generic.common.res.dca:1"/>
```

the entry in the default rblInfo file changes to:

1.value=Part Number

Caution: Never modify automatically generated keys.

Conflict Resolution

When multiple developers work on the same configuration files, there is always a possibility of a conflict. For instance, a configuration file can have two entries with the same key:

```xml
<Label resource="Number[##]com.ptc.refimpl.co.generic.common.res.dca:1"/>
<Label resource="Something[##]com.ptc.refimpl.co.generic.common.res.dca:1"/>
```

and the rblInfo file can have corresponding entries;

1.value=Number
1.value=Something

Generally, DCA utility resolves conflicts automatically. The resulting configuration file should be as follows:

```xml
<Label resource="Number[##]com.ptc.refimpl.co.generic.common.res.dca:1"/>
<Label resource="Something[##]com.ptc.refimpl.co.generic.common.res.dca:2"/>
```

and the rblInfo file can have corresponding entries;

1.value=Number
2.value=Something

Comments

DCA utility supports a comment for a localizable message. The following is how it gets specified in the original configuration specification

```xml
<Comment text="The best beer in the world."/>
<Label resource="Carlsberg[##]"/>
```

Running utility produces the following:

```xml
<Label resource="Carlsberg[#The best beer in the world.]"/>
```
Configuration Specification Reference

This section serves as a reference for valid DCA configuration elements. This also includes an explanation of syntax and descriptions of deprecated and obsolete elements and attributes.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disclaimer</td>
<td>A-2</td>
</tr>
<tr>
<td>Configuration Syntax</td>
<td>A-3</td>
</tr>
<tr>
<td>Configuration Elements</td>
<td>A-35</td>
</tr>
<tr>
<td>Deprecated Configuration Elements</td>
<td>A-167</td>
</tr>
<tr>
<td>Deprecated Attributes</td>
<td>A-175</td>
</tr>
<tr>
<td>Obsolete Configuration Elements</td>
<td>A-176</td>
</tr>
<tr>
<td>Obsolete Handlers and Renderers</td>
<td>A-179</td>
</tr>
<tr>
<td>Upgrades Requiring Manual Changes in Configuration Specifications</td>
<td>A-180</td>
</tr>
</tbody>
</table>
Disclaimer

Formal DTD syntax cannot be used for the definitions of configuration elements. This document extends DTD syntax by employing additional simple and intuitive constructs that are not part of DTD syntax, thus making declarations invalid for use by validating XML editors.

Extend, Insert elements are legal sub-elements of every configuration element. All element definition has this implied definition in its legal entity list.

```xml
<!ELEMENT ENTITY
    (Insert*, Extend*) >
```

Due to the use of Extend and Insert elements, DTD fragments define what kind of sub-elements and attributes an element can have after inheritance and aggregation statements are taken into account.
Configuration Syntax

Use of Identifiers

```xml
<!ATTLIST ENTITY
id CDATA                     #IMPLIED
>
```

Top-level Elements

It is always necessary to specify identifiers for top-level elements - children of `Context` element.

Descendents of Top-level Elements

It is not necessary (while legal) to specify IDs for the descendents of top-level elements. Only specify IDs when

- configuration element has a corresponding conceptual element
- a tag of the configuration element is not unique in the namespace of its parent configuration element

The most common case will be a set of `LayoutRow` elements in layout, property panel, or action elements in the `ActionList`. Each of them must have unique ID. `TaskParameter`, on the contrary, does not have to have unique ID as it does not have a correspondent conceptual element.

Context

Dimensions are name-value pairs that form the context of the configuration element. DCA supports the following set of well-known dimensions as well as user defined dimensions.

Well-known Dimensions

**appId**

```text
appId = text[CS]
```

Application dimension. Use it to distinguish between application specific configuration fragments.

**decoration**

```text
decoration = text[CS]
```

Decoration dimension allows to turn element off and on using the value in the conditional configuration attributes.

**mode**

```text
mode = (view | create | edit | search)
```


Mode dimension defines the behavior of the conceptual element that is associated with the configuration element. Mode can significantly change how the element is rendered. A property element in view mode would just show up as read only text, but in edit mode an editable text field would show up in the page to allow the property value to be edited. For example:

- **View** creates a read only version of the user interface element.
- **Edit** creates an editable version of the user interface element with the current (if any) data value used form the associated business object.
- **Create** creates an editable version of the user interface element with an empty or default value as appropriate.
- **Search** creates a search version of the user interface element that displays a more appropriate widget to gather input to search for the value.

**roleId**

```
roleId = text[CS]
```

Role dimension reflects the role of the user. Use it to distinguish between role specific configuration fragments.

**type**

```
type = text[CS]
```

Type dimension is used to distinguish between business object type specific configuration fragments. A property panel, row in a table or node in a tree will set this attribute in context based on the type of the object they represent.

**User-defined Dimensions**

```
xxx = text[CS]
```

User defined dimension is a name value pair where the xxx name must follow the rules of an attribute name syntax in XML. The value can be any text.

**Context Element**

Context element specifies the required set of dimensions for the top level configuration elements. Top level elements are those that are children of a Context element. To resolve a reference to a top level configuration elements all required dimensions must be set.

```
<Context type="wt.part.WTPart">
    <Action id="action">...
    </Action>
</Context>

<Context type="wt.doc.WTDocument">
    ...
</Context>
```
Augmenting Context via %dimensions

Configuration element usually inherits context from its parent but can also augment it by specifying additional dimensions. The application developer is allowed to modify context of an element by specifying dimensions as attributes of the element.

<ENTITY %dimensions
appId        CDATA    #IMPLIED
decoration   CDATA    #IMPLIED
mode         (view | create | edit | search) #IMPLIED
roleId       CDATA    #IMPLIED
type         CDATA    #IMPLIED
<context.xxx CDATA    #IMPLIED)*
>

The following specifies a user-defined dimension when you want to add it to the context. More than one dimension can be added using this mechanism.

context.xxx = text[CS]

Example:

<Action mode="view" context.dim1="one">
  ...
</Action>

In this example, an Action element inherits context from its parent and has mode dimension set to "view" and dim1 user-defined dimension set to "one".

<Action mode="view" context.dim1="one">
  ...
  <Renderer resource="myresource.jsp" context.dim1="two"/>
</Action>

In this example, a Renderer element inherits context from its parent and additionally has mode dimension set to "view" and dim1 user-defined dimension set to "two".

Reusing %dimensions When Augmenting Context

Sometimes context needs to be augmented in a few places with the same set of dimensions. The following mechanism can be used to avoid code duplications. The mechanism relies on DCA abilities to understand configuration elements with arbitrary tags as long as they do not exist in the resolved configuration.

Thus the following can be introduced:

<DimensionSet id="janes_dimensions" context.bgcolor="white"/>
The effect is the same as to define

```
<SimpleLayout context.bgcolor="white" context.fgcolor="black">
...
</SimpleLayout>
```

**Conditional Statements**

**Conditional Statements via %conditions**

Application developers are allowed to conditionally include or exclude configuration elements using conditional attributes.

```
<!ENTITY %conditions
  if.appId             CDATA    #IMPLIED
ifnot.appId          CDATA    #IMPLIED
if.type              CDATA    #IMPLIED
ifnot.type           CDATA    #IMPLIED
if.mode              (view | create | edit | search)  #IMPLIED
ifnot.mode           (view | create | edit | search)  #IMPLIED
if.decoration        CDATA    #IMPLIED
ifnot.decoration     CDATA    #IMPLIED
if.roleId            CDATA    #IMPLIED
ifnot.roleId         CDATA    #IMPLIED
(if.xxx               CDATA    #IMPLIED)*
(ifnot.xxx            CDATA    #IMPLIED)*
if                CDATA    #IMPLIED
>
```

where xxx is a name of the user defined dimension.

See %dimensions for the definition of attributes participating in %conditions.

**Caution:** Only string constants, synonyms, and dimensions can participate in conditional statements.

**Example #1**

```
<Action id="myAction" mode="view">
...
<Renderer resource="myresource1.jsp" if.context.dim1="one"/>
<Renderer resource="myresource2.jsp" if.context.dim1="two"/>
</Action>
```

<ActionList>
Inserted Action element will use the renderer myresource1.jsp as user-defined dimension dim1 is set to "one".

Example #2

<Action id="myAction" mode="view">
  ...
  <Renderer resource="myresource1.jsp" if="%[context.dim1]==one"/>
  <Renderer resource="myresource2.jsp" if="%[context.dim1]==two"/>
</Action>

<ActionList>
  <insert ref="myAction" context.dim1="one"/>
  ...
</ActionList>

As in the previous example inserted Action element will still use the renderer myresource1.jsp as user-defined dimension dim1 is set to "one".

Example #3

<Synonym id="ONE" value="one"/>

_Context context.two="two">
  <Action id="myAction" mode="view">
    ...
    <Renderer resource="myresource1.jsp"
      if="%[context#dim1]==%[synonym#ONE]"/>
    <Renderer resource="myresource2.jsp"
      if="%[context#dim1]==%[context#two]"/>
  </Action>
</Context>

<ActionList>
  <insert ref="myAction" context.dim1="one"/>
  ...
</ActionList>
</Context>

As in the two previous examples inserted Action element will still use the renderer myresource1.jsp as user-defined dimension dim1 is set to "one".

Conditional Statements In Event Handlers

Conditional statements inside event handlers do not have restrictions on the availability of values from data sources used to construct them. Regular restrictions apply for setting values.

The syntax of conditional statements resembles that of the Java language.

<On event="executeModelDoer" at="end">
  <if condition="%[request#x1]==1">
    <Copy from="one" to="%[request#message]"/>
  </if>
</On>
<elsif condition="%[request#x1]==2"/>
   <Copy from="two" to="%[request#message]"/>
<else/>
   <Copy from="three" to="%[request#message]"/>
</if>
</On>

It is possible to specify nested conditional statements

<On event="executeModelDoer" at="end">
   <if condition="%[request#x1]==1">
      <if condition="%[request#x2]==1">
         <Copy from="one-one" to="%[request#message]"/>
      </if>
      <else/>
      <Copy from="one-something" to="%[request#message]"/>
   </if>
   <elsif condition="%[request#x1]==2"/>
      <Copy from="two" to="%[request#message]"/>
   <else/>
      <Copy from="three" to="%[request#message]"/>
   </if>
</On>

Conditional statements are case insensitive.
Data Sources Available In Configuration Specifications

Configuration system provides access to a variety of data sources. The data from those sources can be used to construct attribute values of configuration elements and to manipulate values in the same or other data sources as well.

Most of the data sources can only provide string values but some can provide values of other types.

String Constants

This is the simplest data source and it’s read only.

String constants have always string values.

Syntax

attribute_value

Parameter Definition

attribute_value = text [CSI]

A value of a string constant.

How to Set

n/a

How to Get

The following example assigns the value of a string constant to a dimension. The value of the type dimension is wt.doc.WTDocument.

<SimpleLayout id="view" type="wt.doc.WTDocument"/>

String constant can be used as attributes value for any attribute in the configuration specification.
Synonyms

Instead of frequently using the string constants with the same value, DCA configuration system allows the specification of a synonym. A single change to the synonym’s value results in the changes everywhere it’s used.

Synonyms have always string values.

How to Set

Synonyms must be defined in the context like any other top level configuration element. Any combination of string constants and already defined dimensions can be used to compute their values.

The following defines the synonym using a string constant

```
<Context type="wt.doc.WTDocument">
  <Synonym id="WTDocument" value="wt.doc.WTDocument"/>
</Context>
```

The following defines a synonym using already defined dimension

```
<Context type="wt.doc.WTDocument">
  <Synonym id="WTDocument" value="\[context#type\]"/>
</Context>
```

How to Get

Syntax

```
attribute = %synonym_name | %\[synonym_name\] | %\[synonym\#synonym_name\]
```

Parameter Definition

```
synonym_name = text [CS]
```

Name of the synonym.

Examples

In all of the following examples the value of the type dimension is set to be wt.doc.WTDocument.

```
<SimpleLayout id="view" type="%WTDocument"/>
<SimpleLayout id="view" type="%\[WTDocument\]"/>
<SimpleLayout id="view" type="%\[synonym\#WTDocument\]"/>
```

Since a synonym is a regular configuration element it is possible to apply scoping rules to the resolution of the synonym. Thus the following uses the value of the synonym from com.ptc.doc package

```
<SimpleLayout id="view" type="%\[com.ptc.doc:WTDocument\]"/>
<SimpleLayout id="view" type="%\[synonym\#com.ptc.doc:WTDocument\]"/>
```

It is also possible to specify a reference to a synonym with the global scope.
The normal context resolution rules apply to the resolution of a reference to a synonym no matter what usage syntax is employed.

Dimensions

Dimensions are name-value pairs that form context of configuration elements. Configuration elements usually inherit context from their parents, but can also augment it by specifying additional dimensions.

Dimensions can only have string values.

DCA supports a set of well-known dimensions, as well as user-defined ones. The following are well-known dimensions:

- appId
- decoration
- roleId
- type
- mode

How to Set

Dimensions can be defined using Context element as XML attributes using string constants as values.

```xml
<Context type="wt.part.WTPart" context.color="blue">
...
</Context>
```

Dimensions can also be defined to augment context of any configuration element. Any combination of string constants and already defined synonyms and dimensions can be used to compute their values.

```xml
<Context context.width="20">
...
</Context>
```

```xml
<SimpleLayout type="wt.part.WTPart"
   context.color="blue"
   context.height="@[context#width]">
...
</SimpleLayout>
```

Dimensions can also be defined as part of a definition of a go-to, new frame, or new temporary frame action to form a context that is going to be used to find a definition of a target frame. While any combination of any attribute values can be used to compute the values of dimensions defined that way, not all dimension can be used to form context for a permanent frame. Only the following sub-set of well-known dimensions will be taken into account:

- appId
- decoration
There are no restrictions on dimensions that form context to be used to find a definition of a target temporary frame.

In this example the target frame will have color dimension set to blue.

```
<Action>
  ...
  <Location ref=":view" context.color="blue"/>
  ...
</Action>
```

The NewContext element is permitted as a child of a Location element to create a new context for the target frame referenced by the Location element.

```
<Action>
  ...
  <Location ref=":view">
    <NewContext initWithCurrentContext="true">
      <Copy from="%{synonym#WTPart}" to="%{context#type}"/>
      <Copy from="%{context#color}" to="%{context#bgcolor}"/>
      <Copy from="white" to="%{context#fgcolor}"/>
      <Remove to="%{context#decoration}"/>
    </NewContext>
  ...
</Action>
```

In this example the frame with alias view will have the set of dimensions copied from the set of dimensions of the Location configuration element with

- **type** dimension set with the value of the synonym WTPart in the package where the action is defined

- **bgcolor** dimension set with the value of the dimension color of the Location element

- **bgcolor** dimension set with the value white

- **decoration** dimension removed from the set of dimensions

**Caution:** It is possible to use parameters and attributes as the source for context dimensions of the target frame. However, since DCA caches frame configurations, it’s not beneficial to have too many frame configurations with different contexts for a given location (frame ID).

**Caution:** HTTP request parameters should be used with caution to construct context for temporary frames. DCA evaluates context at the moment when a temporary frame is about to be created. In DCA Web it happens after processing of the HTTP request that resulted in a drawing of an action, so httpRequest value source being used to construct context is different from the httpRequest value source that was used during the drawing of the action.
Assuming the during the drawing of the following action abc request parameter was defined with a value X, the context will have dimension abc with no value.

```
<Action>
  ...
  <Location ref=":view">
    <NewContext initWithCurrentContext="true">
      <Copy from="%[request#abc]" to="%[context#abc]/" />
      <Copy from="%[httpRequest#def]" to="%[context#abc]/" />
    </NewContext>
  ...
</Location>
```

**Caution:** Due to exactly the reason described above request value source should not be used to construct new context for temporary frames.

**How to Get**

**Syntax**

```
attribute = %[context#dimension_name]
```

**Parameter Definition**

```
dimension_name = text [CS]
```

Name of the dimension.

**Examples**

The following example creates user defined bgcolor dimension using a value of user defined color dimension.

```
<SimpleLayout id="view" context.bgcolor="%[context#color]">
```

It is possible to access dimensions for a given configuration element programmatically.

```
CoConfigFeature config_feature = ...
CoEnvData envData = ...
CoContext context = config_feature.getValue(
  CoConfigFeatureType.CONTEXT, envData);
String my_dim_val = (String)context.get("my_dim");
```

**Caution:** Do not add, change, or remove dimensions programmatically.
Preferences

DCA uses Windchill preference mechanism to store preferences like page size, column order, etc.

Preferences have always string values.

Syntax

```
%[preference#user#preference_key]
```

Parameter Definition

```
preference_key = text [CS]
```

The value of a preference key.

How to Set

Preferences can be set using any combination of string constants, synonyms, dimensions, and any already defined parameters from any scope and attribute values.

```
<Frame id="sample">
    <On event="initModel" at="end">
        <Copy from="%[context#height]"
            to="%[preference#user#/com/ptc/core/ca/co/DCAType.Preference/someKey]"/>
    </On>
    ...
</Frame>
```

How to Get

The following example initializes task parameter with a value of a preference result_size.

```
<TaskParameter name="result_size" value="%[preference#user#/com/ptc/core/ca/co/DCAType.Preference/result_size]"/>
```
Properties

DCA permits accessing values from various property files from a file system. The access is read only.

Properties have always string values.

Syntax

```text
%(property#file_path#property_name} | 
%(property#property_name}
```

Parameter Definition

- **file_path = text [CI]**
  - File name of the property file including path information. Can be specified relative to the codebase directory.

- **property_name = text} [CS]**
  - Name of the property from the property file.

How to Set

n/a

How to Get

The following example initializes task parameter with a value of a property `result_size` from a file `search.properties`.

```xml
<TaskParameter name="result_size" value="%(property#search.properties#result_size)"/>
```

`wt.properties` is considered to be a default property file. Thus the following statements are equivalent:

```xml
<TaskParameter name="result_size" value="%(property#wt.properties#result_size)"/>

<TaskParameter name="result_size" value="%(property#result_size)"/>
```

When a property file is located in a subdirectory, it is possible to specify a path relative to the codebase directory

```xml
<TaskParameter name="result_size" value="%(property#search/search.properties#result_size)"/>
```
Session Parameters

Conceptual session has a hash map for storing arbitrary name-value pairs - session parameters. Session parameters can be used to establish communication between DCA elements across request and frame boundaries.

Having a reference to a conceptual environment it is possible to obtain access to them in java code

```java
CoEnvData envData = ...;
HashMap session_map = envData.getSessionData().getAnyData();
```

An object of any type can be used as a key or a value. However, when a key and a value are of `java.lang.String` type, configuration system provides an access to the parameter via configuration file.

**Caution:** Session parameters are parameters of the conceptual session and have no relationship to the servlet session.

Session parameters can have values of any types as long as they can be serialized and deserialized.

Syntax

```
%[session#parameter_name]
```

Parameter Definition

```
parameter_name = text
```

Name of the session parameter.

How to Set

Session parameter can be set using any combination of string constants, synonyms, dimensions, and any *already defined* parameters from any scope and attribute values.

```xml
<Frame id="sample">
  <On event="initModel" at="end">
    <Copy from="white" to="%[session#fgcolor]"/>
    <Copy from="%[synonym#width]" to="%[session#width]"/>
    <Copy from="%[context#height]" to="%[session#height]"/>
    <Copy from="%[session#color]" to="%[session#bgcolor]"/>
  </On>
  ...
</Frame>
```

How to Get

The following example initializes task parameter with a value of the session parameter *result_size*.

```
<TaskParameter name="result_size" value="%[session#result_size]"/>
```
Request Parameters

Conceptual request has a hash map for storing arbitrary name-value pairs - request parameters. Request parameters can be used to establish communication between DCA elements during the processing of a single request.

Having a reference to a conceptual environment, it is possible to obtain access to them in java code:

```java
CoEnvData envData = ... 
HashMap request_map = envData.getRequestData().getAnyData();
```

An object of any type can be used as a key or a value. However, when a key and a value are of `java.lang.String` type, configuration system provides an access to the parameter via configuration file.

**Caution:** Request parameters are parameters of the conceptual request and have no relationship to the HTTP request parameters.

Request parameters can have values of any types.

Syntax

```%
[request#parameter_name]
```

Parameter Definition

`parameter_name = text] [CS]`

Name of the request parameter.

How to Set

Request parameter can be set using any combination of string constants, synonyms, dimensions, and any already defined parameters from any scope and attribute values.

```xml
<Frame id="sample">
<On event="initModel" at="end">
  <Copy from="white" to="%[request#fgcolor]"/>
  <Copy from="%[synonym#width]" to="%[request#width]"/>
  <Copy from="%[context#height]" to="%[request#height]"/>
  <Copy from="%[session#color]" to="%[request#bgcolor]"/>
</On>
...
</Frame>
```

How to Get

The following example initializes task parameter with a value of a request parameter `result_size`.

```xml
<TaskParameter name="result_size" value="%[request#result_size]"/>
```
Http Request Parameters

Having a reference to conceptual environment, it’s possible to access parameters from http request in java code

```java
CoEnvData envData = ...;
HttpServletRequest request = ((WebEnvData)envData).getGatewayJspData().getRequest();
```

Configuration system can also be used to provided read-only access to http request parameters.

HTTP request parameters have always string values.

**Syntax**

```
{%[httpRequest#parameter_name]}
```

**Parameter Definition**

```
parameter_name = text] [CS]
```

Name of the HTTP request parameter.

**How to Set**

n/a

**How to Get**

The following example initializes task parameter with a value of the HTTP request parameter color.

```xml
<TaskParameter name="color" value="{%[httpRequest#color]}"/>
```
Frame Parameters

Conceptual frame has a hash map for storing arbitrary name-value pairs - frame parameters. Frame parameters can be used to establish communication between DCA elements across the request boundaries. The elements must all be descendents of the same frame.

Having a reference to a conceptual element or environment it is possible to obtain access to them in java code

```java
CoElement element = ...
CoFrame frame = element.getFrame();
HashMap request_map = frame.getAnyData();
```

or

```java
CoEnvData envData = ...
CoElement element = envData.getRequestData().getCurrentElement();
CoFrame frame = element.getFrame();
HashMap request_map = frame.getAnyData();
```

An object of any type can be used as a key or a value. However, when a key and a value are of `java.lang.String` type, configuration system provides an access to the parameter via configuration file.

**Caution:** Frame parameters are parameters of the conceptual request and have no relationship to the HTML frames.

Frame parameters can have values of any types as long as they can be serialized and deserialized.

**Syntax**

```
%[frame#parameter_name]
```

**Parameter Definition**

`parameter_path = text` [CS]

Path to the frame parameter.

**Syntax:**

```
parameter_name |
opener::parameter_name
```

`parameter_name = text` [CS]

Specifies a frame parameter in the frame the current DCA element belongs to.

`opener::parameter_name`

Specifies a frame parameter in the opener frame for the frame the current DCA element belongs to.
How to Set

Frame parameter can be set using any combination of string constants, synonyms, dimensions, and any *already defined* parameters from any scope and attribute values.

```xml
<Frame id="sample">
  <On event="initModel" at="end">
    <Copy from="white" to="%[frame#fgcolor]"/>
    <Copy from="%[synonym#lmargin]" to="%[frame#lmargin]"/>
    <Copy from="%[context#rmargin]" to="%[frame#rmargin]"/>
    <Copy from="%[session#color]" to="%[frame#bgcolor]"/>
  </On>
  ...
</Frame>
```

How to Get

The following example initializes task parameter with a value of a frame parameter `result_size`.

```xml
<TaskParameter name="result_size" value="%[frame#result_size]"/>
```
Values of Enumerator Elements

Enumerators have values. The value of the enumerator element is the value chosen from the list of available values.

Having a reference to an enumerator element it is possible to obtain its value programmatically.

```java
CoEnumerator enum = ...;
CoEnumeratorModel enum_model = (CoEnumeratorModel)enum.getModel();
int chosen_index = enum_model.getChosenIndex();
ArrayList values = enum_model.getValues();
String chosen_value = (String)values.get(chosen_index);
```

It is also possible to change what value is chosen by changing `chosen_index` in the model.

Configuration system provides access to the chosen value of the enumerator element.

**Caution:** Enumerator element has no relationship to the enumerated property.

Values of enumerator elements have always string values.

**Syntax**

`%[enum#enum_name]`

**Parameter Definition**

`enum_name = text` [CS]

Id of the enumerator. A dot (.) instead of ID indicates this element.

**How to Set**

The value of the enumerator can be set using any combination of string constants, synonyms, dimensions, and any already defined parameters from any scope and attribute values as long as it produces a legal value. Only values from the list of available values are considered legal.

```xml
<LabelArea id="choice_part" labelType="text">
    <Label resource="Part"/>
    <ToolTip resource="Part"/>
</LabelArea>

<Enumerator id="choice">
    <Insert ref="choice_part" value="part"/>
    <Insert ref="choice_doc" value="doc"/>
    <Insert ref="choice_folder" value="folder"/>
    <On event="initModel" at="end">
        <Copy from="%[frame#choice]" to="%[enum#]"/>
    </On>
</Enumerator>
```
In this example the value of the same enumerator is initialized from the frame parameter choice.

How to Get

The following example initializes task parameter with a value of an enumerator with ID sample that must be unique in the current frame.

<!-- TaskParameter name="choice" value="%[enum#sample]" / -->

Attribute Values of Info*Engine Elements

Instance holder elements maintain Info*Engine elements. The element is an instance holder when it implements CoInstanceHolder interface. At the moment DCA supports the following instance holders:

- SimplePropertyPanel, CompositePropertyPanel
- Wizard
- Row in a table or a tree

Having a reference to an element it is possible to obtain the attribute value of the Info*Engine element programmatically.

CoElement element = ...
Element ie_element = CoElementUtils.getIeElementOfInstanceHolder(element, envData);
Object attr_value = ie_element.getValue(attribute_name);

It is also possible to set value programmatically

CoPropertyUtils.setAttributeValue(ie_element, attribute_name, attr_value);

Attribute values of Info*Engine elements can be of any types as long as they can be serialized and deserialized.

Syntax

`[%[attribute#element_path#attribute_name]`
opener::..

**element_id = text** [CS]

Specifies a DCA element by ID in the frame the current element belongs to.

Specifies the current DCA element

Specifies an instance holder of the current DCA element.

**opener::element_id**

Specifies a DCA element by ID in the opener frame of the frame the current element belongs to. Such specification can only be used in the configuration of a temporary frame.

**opener::..**

Specifies an instance holder of the opener action in the opener frame of the frame the current element belongs to. Such specification can only be used in the configuration of a temporary frame.

**attribute_name = text** [CS]

Name of the attribute of the Info*Engine element.

**How to Set**

The value of the attribute of the Info*Engine element can be set using any combination of string constants, synonyms, dimensions, and any already defined parameters from any scope and attribute values as long as it produces a legal value.

```xml
<Enumerator id="type">
  <Insert ref="choice_part" value="wt.part.WTPart"/>
  <Insert ref="choice_doc" value="wt.doc.WTDocument"/>
  <Insert ref="choice_folder" value="wt.folder.WTFolder"/>
</Enumerator>

<Action id="action.type.change">
  <Label resource="Change Type"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.
  WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.element.
  CoNoOpActonDoer"/>
  <On event="executeActionDoer" at="end">
    <Copy from="%[enum#type]" to="%[attribute#..#obj_type]"/>
  </On>
</Action>
```

Configuration Specification Reference  A-23
In this example the value of the enumerator with ID `type` is used to set the value of the attribute `obj_type` of the Info*Engine element. To find the element DCA will look for the nearest instance holder ancestor of the action.

The following example sets the initial value of the attribute `obj_type` of the Info*Engine element of the property panel.

```xml
<SimplePropertyPanel>
    ...
    <ModelDoer class=...>
        <On event="executeModelDoer" at="end">
            <Copy from="%[enum#type]" to="%[attribute#.obj_type]"/>
        </On>
    </ModelDoer>
</SimplePropertyPanel>
```

The following example sets the value of the attribute `obj_type` of the Info*Engine element. To find the element DCA will look for the instance holder with ID `sample` in the current frame. ID must be unique among the descendents of the frame.

```xml
<Action id="action.type.change">
    <Label resource="Change Type"/>
    <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
    <ClientDoer class="com.ptc.core.ca.co.client.element.CoNoOpActionDoer"/>
    <On event="executeActionDoer" at="end">
        <Copy from="%[enum#type]" to="%[attribute#sample#obj_type]"/>
    </On>
</Action>
```

**How to Get**

The following example initializes task parameter with a value of the attribute `ufid` of the Info*Engine element that belongs to the instance holder ancestor of the current element.

```xml
<TaskParameter name="parent_ufid" value="%[attribute#.ufid]"/>
```
Bean Properties

DCA element and their models are beans. As such, their properties can be accessed via getters and setters methods.

DCA configuration specification provides a syntax to access bean properties of the following types:

- java.util.String
- int
- long
- float
- double
- boolean

Syntax

```
[%[bean#element_id#bean_property]]
```

Parameter Definition

- **element_id** = text] [CS]
  
  Id of the DCA element.

- **bean_property** = text] [CS]
  
  Path to the bean property of DCA element.

How to Set

The following examples marks a property panel as dirty

```
<Action id="action.markDirty">
  <Label resource="Mark Dirty"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.
    WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.element.
    CoNoOpActoonDoer"/>
  <On event="executeActionDoer" at="end">
    <Copy from="true" to="[%[bean#sample#Dirty]]"/>
  </On>
</Action>
```

To access the value of a label attribute of the model using the following syntax

```
<Copy from="My Label" to="[%[bean#sample#Model.Label]]"/>
```

This effectively executes `getModel().getLabel()`

How to Get

The following example initializes task parameter with a value of the dirty parameter of the element with ID sample.

```
<TaskParameter name="dirty" value="[%[bean#sample#Dirty]]"/>
```
Installation Status

Sometimes there is a need to draw a DCA element when certain applications and modules are installed or not installed. DCA provides a mechanism to check the installation status of a set of applications and modules.

Installation statuses have always string values.

How to Set

It is only possible to set an installation status of an application or a module by specifying an installation status determinator class using the following syntax

```xml
<Context>
  <Init id="Init">
    <InstallationStatusDeterminator class = "xxx"/>
  </Init>
</Context>
```

Caution: The ID of the Init element must be "Init" and the element must be defined in the default context.

The installation status determinator class must implement com.ptc.core.ca.co.client.misc.CoInstallationStatusDeterminator interface. DCA provides two default implementations of the interface to allow the specification of installed applications and modules along with their versions via configuration specification.

```xml
<Context>
  <Init id="Init">
    <InstallationStatusDeterminator class="com.ptc.core.ca.co.
     client.misc.CoDefaultAppInstallationStatusDeterminator"
     appId="Applicaton A" version="6.2.6"/>
    <InstallationStatusDeterminator class="com.ptc.core.ca.co.
     client.misc.CoDefaultAppInstallationStatusDeterminator"
     appId="Applicaton B" version="5.1.3"/>
    <InstallationStatusDeterminator class="com.ptc.core.ca.co.
     client.misc.CoDefaultModuleInstallationStatusDeterminator"
     moduleId="Module C" version="6.0.2"/>
  </Init>
</Context>
```

When DCA reads configuration specifications it will set up the installation status as follows:

Applicaton A version 6.2.6
Applicaton B version 5.1.3
Module      C version 6.0.2

How to Get

To determine an installation status a set of applications and modules needs to be specified first in the appropriate context. The usual context resolution and scoping rules will be later applied to find the set.

```xml
<Context>
```
<ApplicationSet id="set1">
  <Application appId="Application A" from="6.0.2" to="6.2.8"/>
  <Application appId="Application B" from="4.2"/>
  <Module moduleId="Module C" to="7.0"/>
</ApplicationSet>

Syntax

%
[ifInstalled#condition#installation_set_id]

Parameter Definition

class = (any | all)

Indicates whether all or any of the conditions in the installation set need to be
satisfy to return true.

installation_set_id = text] [CS]

Id of the installation set.

Examples

The example above defines a set of two applications and a module with version-
based conditions. The following example determines whether any of the
applications and modules in the set above are installed.

<Insert ref="something" if="%[ifInstalled#any#set1]==true">

The following example conditionally draws if all of the applications and modules
are installed.

<LayoutRow id="2">
  <LabelArea>
    <Label resource="Set 1"/>
  </LabelArea>
  <DrawHandler class="%[common:DrawController]">
    <Check value="%[ifInstalled#all#set1]==true"/>
  </DrawHandler>
</LayoutRow>

Note the use of a colon in the specification of an element ID. The usual context
resolution and scoping rules apply to the selection of the appropriate set.
Attribute Value Substitution

It is possible to substitute a portion of the attribute value of the configuration element by the value from a data source providing that both of the following conditions are met:

1. the value if of type string
2. the value must be available at the moment of evaluation

The following is a list of examples of substitutions.

```xml
<TaskParameter name="sample"
          value="%[session#%[context#d1]%[frame#f1]]="/>
```

When

context#d1 is A
frame#f1 is F

the statement is equivalent to

```xml
<TaskParameter name="sample" value="%[session#AF]"/>
```

In the next example when

request#r1 is R
context#d2 is session

the following statement

```xml
<TaskParameter name="%[synonym#%[request#r1]]"
          value="%[context#d2]%[frame#f1]="/>
```

is equivalent to

```xml
<TaskParameter name="synonym#R" value="%[session#F]="/>
```

Attribute Value Evaluation

To properly use attributes in the configuration system it is crucial to understand when their values are evaluated. The evaluation process happens during one of the following three stages configuration system goes through:

1. loading configuration specifications
2. resolving configuration specification
3. using configuration specification
Loading Configuration Specifications

This is the initial state when the configuration system reads configuration specifications from the designated directory and loads them in memory. During this stage the system evaluates the following:

- attributes of all Context elements
- ID attributes of all top-level configuration elements

Attributes of Context elements can be specified using any combination of string constants and synonyms. IDs of the top-level configuration elements can be specified using string constants only.

**Caution:** Avoid circular dependencies.

Resolving Configuration Specification

Once all of the configuration specifications are loaded, it is possible to ask configuration system for the configuration describing particular frame. This is exactly what DCA does attempting to draw the frame. DCA supplies ID of the frame and frame context as parameters when making a call to configuration sub-system. To process the call configuration sub-system needs to evaluate the following:

- ref attributes
- if, if.xxx, and ifnot.xxx attributes
- dimension values for both well-known and user-defined dimensions

The values of the attributes listed above can be specified using any combination of string constants and already defined synonyms and dimensions.

**Caution:** Avoid circular dependencies.

Using Configuration Specifications

Once all of the configuration specifications are loaded and a configuration describing a particular frame is produced, DCA can use it to create frame element and its descendents subsequently drawing them. The values of all configuration attributes not mentioned in the previous steps are evaluated at the moment when they are needed by DCA. Any combination of string constants, synonyms, dimensions, and any already defined parameters from any scope and attribute values can be used to define a value.

It is possible to specify values that are not a string constant for ID, class, and resource attributes of configuration elements.

```xml
<Context>
  <SimpleTable id="table">
    <Type type="%[context#t_type]"/>
    <SimpleColumn show=":name"/>
    <ModelDoer class="%common:SimpleModelTaskDoer">
```

Configuration Specification Reference
In this example we’ve configured a generic table capable of presenting objects of a particular type as a content of a folder. The type is defined via user-defined dimension \texttt{t\_type}.

\textbf{Caution:} Avoid circular dependencies.

\textbf{Caution:} Avoid ambiguous parameterization. Since attribute values are evaluated when requested, they may depend on the other values when substitution is used. As the result values may not be the same next time.

\textbf{Copying Values}

It is possible to generically copy values between different sources. For instance the following will copy the value of \texttt{type} dimension is equal to the value of the frame parameter \texttt{type}.

\texttt{<Copy from="\%[context\#type]" to="\%[frame\#type]"/>}

The usual substitution rules apply.

\texttt{<Copy from="\%[context\%[frame\#dim\_name]]" to="\%[frame\#type]"/>}

Copying operation can be triggered programmatically. The following code triggers copy operation for all \texttt{Copy} sub-element for a given configuration feature.

\texttt{CoConfigFeature config\_feature = ... CoConfigHelper.processCopyElements (CoConfigurationImpl config, CoEnvData envData)}

DCA supports a set of places where Copy operations are initiated by DCA.

\textbf{Caution:} It is not possible to change the value of a string constant, synonym, or dimension using \texttt{Copy}. 

\texttt{<Task nextOperation="STDOP|com.ptc.windchill.view" taskDelegate="dca-list-folder-members">}
\texttt{<TaskParameter name="content\_type" value="\%[context\#t\_type]"/>}
\texttt{</Task>}
\texttt{</ModelDoer>}
\texttt{</SimpleTable>}

\texttt{<LayoutRow id="parts" context.t\_type="WCTYPE|wt.part.WTPart">}
\texttt{<Insert id="\%[context\#t\_type]\_table" ref="table"/>}
\texttt{</LayoutRow>}
\texttt{<LayoutRow is="docs" context.t\_type="WCTYPE|wt.part.WTDocument">}
\texttt{<Insert id="\%[context\#t\_type]\_table" ref="table"/>}
\texttt{</LayoutRow>
Prior to the Rendering of a Conceptual Element

The default drawing algorithm updates the model of a DCA element, creates a set of its children, if any, and renders the element. When DCA detects Copy sub-element of the configuration element, it invokes copy operation before it renders the element.

Thus the following will set the value of obj_type frame parameter to be part.

```xml
<SimpleFrame id="sample">
  <On event="executeModelDoer" at="end">
    <Copy from="part" to="%[frame#obj_type]"/>
  </On>
</SimpleFrame>
```

The following will set the value of the enumerator element from obj_type frame parameter.

```xml
<Enumerator id="choice">
  <Insert ref="choice_part" value="part"/>
  <Insert ref="choice_doc" value="doc"/>
  <Insert ref="choice_folder" value="folder"/>
  <On event="executeModelDoer" at="end">
    <Copy from="%[frame#obj_type]" to="%[enum#.]"/>
  </On>
</Enumerator>
```

After the Execution of a Doer

The following action is configured to execute a doer that does nothing but let’s specify a Copy sub-element.

```xml
<Action id="action.type.change">
  <Label resource="Change Type"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.element.CoNoOpActoonDoer"/>
  <On event="executeModelDoer" at="end">
    <Copy from="%[enum#choice]" to="%[frame#obj_type]"/>
  </On>
</Action>
```

When the action is triggered, the value of the enumerator from the previous example will be used as a new value of the obj_type frame parameter.

Checking Values

It is possible to generically check values from different sources. For instance the following will check whether the value of type dimension is equal to the value of the synonym WTPart.

```xml
<Check value="%[context#type]==%[WTPart]"/>
```

You can also check for inequality
The usual substitution rules apply.

The result of the check can only be accessed programmatically. The following code evaluates all Check sub-element for a given configuration feature. The result is true when all of the conditions are met and false otherwise.

```java
CoConfigFeature config_feature = ...
Boolean result = config_feature.getBoolean (CoConfigFeatureType.CHECK, envData);
```

DCA provides draw controller that takes advantage of Check feature.

**Connector**

Connector element allows drawing of a DCA element created based on a configuration fragment located in dynamically calculated context. This is fundamentally different from the Insert element that can only use statically calculated context. The following is an example of the configuration utilizing Connector.

```xml
<LayoutRow>
  <Connector ref=":mypanel">
    <NewContext>
      <Copy from="%[enum#typeSelector]" to="%[context#type]"/>
      <Copy from="%[attribute#..#activityType]" to="%[context#activityType]"/>
    </NewContext>
  </Connector>
</LayoutRow>
```

In this example layout row needs to include the element with ID mypanel. The element must be defined as a top level element in any package. Since it's a top level element, it can be defined in multiple contexts. For instance

```xml
<Context type="wt.part.WTPart" activityType="Review">
  <SimplPropertyPanel id="mypanel">
    ...
  </SimplPropertyPanel>
</Context>

<Context type="wt.doc.WTDocument" activityType="Approve">
  <SimplPropertyPanel id="mypanel">
    ...
  </SimplPropertyPanel>
</Context>
```

To include the correct configuration fragment in the layout row, Connector needs to compute context first. For that it will evaluate all Copy statements to come up with a set of name value pairs - dimensions that form context. The context then will be used to locate a fragment with the specified ID mypanel.
There are many use cases when Connector element can be very useful. The most important are the following:

- an ability to find correct search panel and search result table based on the value of the enumerator representing an object type
- an ability to find correct property panel for a workflow activity

Limitations

Since Connector element computes context dynamically, it’s not possible to know the attributes the connected fragment needs to be correctly drawn. Thus attempting to connect fragments that depend on the attributes of Info*Engine elements, but do not perform the data acquisition, will most likely fail with an exception at run time. Property panels, table, trees and positioning elements like layouts and lists containing them can, on the other hand, be connected safely.

Connector cannot refer to a Label, Tooltip, ElementGroup and etc. as such configuration elements do not have correspondent DCA elements. Only configuration elements that have correspondent DCA elements can be referred to by the Connector.

Handling Element Events

DCA provides a mechanism to specify an event handler for a DCA element via On configuration element. The following is an example of the initialization of a default value for an enumerator.

```xml
<Enumerator id="search5.enumerator.type">
  <Insert ref="search5.enumerator.type.part" value="%[doc:WTDocument]"/>
  <Insert ref="search5.enumerator.type.doc" value="%[part:WTPart]"/>
  <Insert ref="search5.enumerator.type.folder" value="%[folder:SubFolder]"/>
  <On event="initModel" at="end">
    <Copy from="%[part:WTPart]" to="%[enum#.]\"/>
  </On>
</ Enumerator>
```

The following events are supported

<table>
<thead>
<tr>
<th>Event</th>
<th>Description of the stage the element goes through</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw</td>
<td>drawing of the element</td>
</tr>
<tr>
<td>initModel</td>
<td>initialization of the model of the element</td>
</tr>
<tr>
<td>updateModel</td>
<td>update of the model of the element</td>
</tr>
<tr>
<td>executeModelDoer</td>
<td>execution of the model doer</td>
</tr>
<tr>
<td>executeActionDoer</td>
<td>execution of the action doer</td>
</tr>
</tbody>
</table>
It is possible to handle events at different times related to the stage the element goes through.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description of the stage the element goes through</th>
</tr>
</thead>
<tbody>
<tr>
<td>updateChildrenSet</td>
<td>update of the set of children</td>
</tr>
<tr>
<td>render</td>
<td>render the element</td>
</tr>
</tbody>
</table>

It is possible to specify a condition using `when` attribute. The syntax for a condition is exactly the same as the syntax for `if` except other values can be used from any sources. For instance,

```xml
<On event="initModel" at="end"
    when="%[attribute#.name=="MyPart"]">
    ...
</On>
```

The absence of a condition is the same as if the condition is evaluated to true.

### Spanning Multiple Cells in a Layout

To indicate that an element occupies more than one cell in a layout use the following attributes. Their meaning is the same as the `COLSPAN` and `ROWSPAN` attributes of a Table element in HTML.

```xml
<!ENTITY  %span
colspan CDATA          #IMPLIED
rowspan CDATA          #IMPLIED>
```

**colspan = number[CN]**

The number of columns spanned by the element in a layout. The default value for this attribute is one ("1").

**rowspan = number[CN]**

The number of rows spanned by the element in a layout. The default value for this attribute is one ("1").
Configuration Elements

The following is the meaning of the features of the configuration elements:

**Concrete Web Element**

A concrete WebElement exists for this configuration element. Many configuration elements don't have a concrete WebElement but contribute to the behavior of a parent element, usually in the construction of the model for the element. Concrete WebElement will have its draw method called, to draw the element on the screen. Elements that map to concrete WebElement can have ModelHandler, RenderHandler, and DrawHandler specified to override the default behavior of the class.

**Web Container Element**

The element is a container and has concrete web element children that need to be drawn when this element is drawn. The container element has the responsibility to make sure the children are drawn. If overriding the drawHandler, the drawHandler must take responsibility for drawing the children.

**Java Renderer**

Java-based renderers that exist to render an element.

**JSP Renderers**

JSP-based renderers that exist to render an element.
**Action**

**Description**

The Action element configures application operations available to the user. Its behavior is controlled by its configured sub elements. Generally:

- ActionHandler controls how user operation behaves. For example, load a frame in the same or a new window, execute an operation in the same frame.
- Doer controls what an operation does. It also controls action visibility and accessibility.
- Location is a configuration ID of a frame element to load in the same or new window.
- Renderer controls how the action is drawn to the screen.

**Syntax**

```xml
<!ELEMENT  Action
    (Label?, Image?, ToolTip?, Location?, UpdateLocation?, On*,
     (DrawHandler |
     (ActionHandler, (ActionDoer | ClientDoer)?,
     (RenderHandler | Renderer)?
    )
  )>
<!ATTLIST Action
     id CDATA                     #IMPLIED
     needSelection  (true | false)  "false"
     activationKey #IMPLIED
     %dimensions; #IMPLIED
     %conditions; #IMPLIED
     %span; #IMPLIED>
```

**Attribute definitions**

- **id = [text]** [CS]
  The ID of the element.

- **needSelection = (true | false)**
  Defines whether the action requires a business object as part of its operation. This attribute is intended to be used for actions that are in selector components like a table or a tree. If set to true, a set of objects needs to be selected in a selector in order for an operation to be executed.

- **activationKey = [text]** [CI]
  Contains unicode value of the key that activates this action when depressed. Typically equal to "\u000D" which is an Enter key. Only a single action in a frame can specify a particular activation key.
Features

Concrete Web Element
com.ptc.core.ca.web.client.action.WebAction

Container Element
no

Java Renderers
com.ptc.core.ca.web.client.action.WebActionRenderHandler
com.ptc.core.ca.web.client.action.WebActionButtonRenderHandler

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp
/wtcore/jsp/com/ptc/core/ca/web/action/button.jsp
/wtcore/jsp/com/ptc/core/ca/web/action/help.jsp
**ActionDoer**

**Description**

The ActionDoer specifies the class that implements an operation to run when an action is triggered. It also implements methods to control action visibility and accessibility. For the list of legal ActionDoer classes (and when to use them), see the section on Data Acquisition and Manipulation in this chapter.

**Syntax**

```xml
<!ELEMENT  ActionDoer
   (Need*, Task?, Validator?, Copy*) >
<!ATTLIST ActionDoer
   id CDATA                     #IMPLIED
   class CDATA            #REQUIRED
   %dimensions;           #IMPLIED
   %conditions;           #IMPLIED
   >
```

**Attribute definitions**

- **id** = `text` [CS]
  The ID of the element.

- **class** = `text` [CS]
  The fully qualified name of the Java class that implements a an action doer. The doer must implement `com.ptc.core.ca.co.client.doer.CoActionDoer` interface.

**Features**

- **Concrete Web Element**
  no

- **Container Element**
  no

- **Java Renderers**
  Not available

- **JSP Renderers**
  Not available
**ActionHandler**

**Description**

The ActionHandler element configures how an action behaves. Open a new window, perform an operation on the selected object, open a wizard on the selected object, … For a list of configurable ActionHandler classes, see the section on *Actions and Action Handlers* in this chapter.

**Syntax**

```
<!ELEMENT  ActionHandler EMPTY
>
<!ATTLIST ActionHandler
  id CDATA                   #IMPLIED
  class CDATA            #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

**Attribute definitions**

- **id** =  *text* [CS]
  
  The ID of the element.

- **class** =  *text* [CS]
  
  The fully qualified name of the Java class that implements an action handler. The handler must implement `com.ptc.core.ca.co.client.action.CoActionHandler` interface

**Features**

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available

- **JSP Renderers**
  
  Not available
**ActionList**

**Description**

The ActionList defines a set of actions that are drawn as a group. An ActionList is comprised of Actions, ActionLists and Separators.

**Syntax**

```xml
<!ELEMENT  ActionList
    (Prompt?, Action*, ActionList*, HyperLink*,
     Label*, ToolTip*, Trigger?, Image*,
     Separator*, On*,
     (DrawHandler | ModelHandler?,
      (RenderHandler | Renderer)? )
    )>

<!ATTLIST ActionList
    id CDATA                     #IMPLIED
    actionListType  (vertical | horizontal | dropDown | toolbar)
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
    %span; #IMPLIED
>
```

**Note:** Drop down action lists must have `Prompt` subelement specified. The rest of the action list types ignore it.

**Attribute definitions**

- **id** = text[CS]
  
  The ID of the element.

- **actionListType** = (vertical | horizontal | dropDown | toolbar )
  
  Environment independent direction to render the action list. When specified in DCA Web, a renderer need not be specified and the default renderer for a type will be used. Otherwise do not specify action list type but rather specify an appropriate renderer.

- **Caution:** `Trigger` sub-element has effect only on passive drop down list renderer as it uses it to render an action trigger.

**Features**

**Concrete Web Element**

com.ptc.core.ca.web.client.action.WebActionList

**Container Element**

yes
**Java Renderers**

com.ptc.core.ca.web.client.action.WebActionDropDownListRenderHandler  
com.ptc.core.ca.web.client.action.WebPassiveActionDropDownListRenderHandler  
com.ptc.core.ca.web.client.action.WebActionHorizontalListRenderHandler  
com.ptc.core.ca.web.client.action.WebActionLevel1TabRenderHandler  
com.ptc.core.ca.web.client.action.WebActionLevel2FooterRenderHandler  
com.ptc.core.ca.web.client.action.WebActionLevel2TabRenderHandler  
com.ptc.core.ca.web.client.action.WebActionVerticalListRenderHandler  

**JSP Renderers**

/wtcore/jsp/com/ptc/core/ca/web/action/dropdownlist.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/passivedropdownlist.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/horizontallist.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/level1_tab.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/level2_footer.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/level2_tab.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/subtab.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/tab.jsp  
/htcore/jsp/com/ptc/core/ca/web/action/verticallist.jsp
Anchor

Description

The Anchor element is used by Hyperlink element to configure an anchor.

Syntax

```xml
<!ELEMENT Anchor EMPTY >
<!ATTLIST Anchor
    id CDATA #IMPLIED
    resource CDATA #REQUIRED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  
  ID of the element.

- **resource** = text[CS]
  
  Name of the anchor.

Examples

To configure:

```xml
<HyperLink labelType="text" drawType="text">
    <Anchor resource="my_anchor"/>
</HyperLink>
```

To create a link to an anchor:

```xml
<HyperLink labelType="text" drawType="text">
    <HRef resource="#my_anchor"/>
    <Label resource="My Anchor"/>
</HyperLink>
```

Features

**Concrete Web Element**

- no

**Container Element**

- no

**Java Renderers**

- Not available

**JSP Renderers**

- Not available
Application

Description

The Application element defines a condition that an application must satisfy to belong to a set of applications and modules.

Syntax

```xml
<!ELEMENT Application EMPTY>
<!ATTLIST Application
    id CDATA                    #IMPLIED
    %dimensions;              #IMPLIED
    %conditions;              #IMPLIED>
```

Attribute definitions

`id = text[CS]`

The ID of the element.

Examples

```xml
<Context>
  <InstallationSet id="set1">
    <Application appId="Applicaton A" from="6.0.2" to="6.2.8"/>
    <Application appId="Applicaton B" from="4.2"/>
    <Module moduleId="Module C" to="7.0"/>
  </InstallationSet>
</Context>
```

Features

**Concrete Web Element**

no

**Container Element**

no

**Java Renderers**

Not available

**JSP Renderers**

Not available
Check

Description

The Check element specifies conditions to check. A parent of the Check element usually specifies validator or controller class that is responsible for the interpreting the results of the check. DCA provides com.ptc.core.ca.co.client.element.CoDrawController to conditionally draw based on the conditions specified by the Check elements. When values match, controller draws the element it is specified for.

Syntax

```xml
<!ELEMENT  Check EMPTY>
<!ATTLIST Check
  id CDATA                      #IMPLIED
  value CDATA                   #REQUIRED
  %dimensions;                  #IMPLIED
  %conditions;                  #IMPLIED
>
```

Attribute definitions

- **id** =  `text[CS]`
  
  ID of the element.

- **value** =  `text[CS]`
  
  Condition to check.

Features

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available

- **JSP Renderers**
  
  Not available
CheckTaskParam

Description
The CheckTaskParam element requires com.ptc.core.ca.co.client.doer.CoTaskValidatorHasParameter task validator to verify that the task is supplied with a specified parameter. When it is, the task will be executed, otherwise it will not.

Syntax

<!ELEMENT CheckTaskParam EMPTY >
<!ATTLIST CheckTaskParam
  id CDATA #IMPLIED
  name CDATA #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED>

Attribute definitions

id = text[CS]
  ID of the element.

name = text[CS]
  Name of the task parameter.

Features

Concrete Web Element
  no

Container Element
  no

Java Renderers
  Not available

JSP Renderers
  Not available
ClientDoer

Description

The ClientDoer element defines the operation intended only to run in the client. Client doers are operations that don't modify business data or acquire data from the database. For example, reordering the columns in a table does not retrieve any data but just reorders the data.

Syntax

```xml
<!ELEMENT ClientDoer (On*)>
<!ATTLIST ClientDoer
  id CDATA              #IMPLIED
  class CDATA            #REQUIRED
  %dimensions;           #IMPLIED
  %conditions;           #IMPLIED
>
```

Attribute definitions

id =  text[CS]

The ID of the element.

class = text[CS]

The fully qualified name of the Java class that implements a client doer. The doer must implement `com.ptc.core.ca.co.client.doer.CoActionDoer` interface.

Features

Concrete Web Element

no

Container Element

no

Java Renderers

Not available

JSP Renderers

Not available
Comparator

Description

The Comparator element can be used to sort entries of a data driven enumerator.

Syntax

```xml
<!ELEMENT  Comparator EMPTY
    >
<!ATTLIST Comparator
    id CDATA #IMPLIED
    class CDATA #REQUIRED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
    %span; #IMPLIED
    >
```

Attribute definitions

- **id** = text [CS]
  - ID of the element.
- **class** = text [CS]
  - The fully qualified class name of the class that implements a sorting algorithm.

Features

- **Concrete Web Element**
  - no
- **Container Element**
  - no
- **Java Renderers**
  - Not available
- **JSP Renderers**
  - Not available
CompositeColumn

Description

The CompositeColumn element allows the creation of a complex column header. Using the SimpleColumn element creates the column header using a standard faceplate renderer based on the label attribute. The CompositeColumn allows many elements to be included in the column header since the header is simple layout-based. This allows specifying any number of LayoutRows in it. The "For" element allows the creation of a cell content element to be substituted depending on the type of the Info*Engine element a row represents.

Syntax

```xml
<!ELEMENT  CompositeColumn
  (For*, Label, LayoutRow*, On*,
   (DrawHandler | ModelHandler?,
    (RenderHandler | Renderer)?)
  )>
<!ATTLIST CompositeColumn
  id CDATA                 #IMPLIED
  visible (true | false)  "true"
  show CDATA        #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

- **id** =  **text**
  - The ID of the element.

- **visible** =  **true | false**
  - Determine if column (or maybe the column header) is visible. The default value is true. This was not used in service pack 3.

- **show** =  **text**
  - The ID of the element to draw in the cell for this column in each row of the table. Generally this is the ID of a Property, an ActionList or an Action. However, there are very few restrictions on the type of element that can be drawn in the cell.

Features

- **Concrete Web Element**
  - com.ptc.core.ca.web.client.layout.WebLayout

- **Container Element**
  - yes
Java Renderers
  see SimpleLayout element

JSP Renderers
  see SimpleLayout element
CompositeFrame

Description

The CompositeFrame is one of the legal root window elements (SimpleFrame is the other). In addition to single layout element, it allows specification of a FrameHeader element. This allows rendering of a custom FrameHeader. In DCA Web an alias parameter specified on the URL must by ID of an existing frame element. ID must be unique unless it is qualified by context dimensions. The view property frames for objects could have the same IDs but are found because they have different type context.

Syntax

```xml
<!ELEMENT  CompositeFrame
  (FrameHeader, On*,
   (Connector |
    CompositeLayout | SimpleLayout |
    CompositePropertyPanel | SimplePropertyPanel |
    CompositeTable | SimpleTable |
    CompositeTree |
    EmbeddedURLContent |
    Wizard),
   (DrawHandler |
    ModelHandler?,
    (RenderHandler | Renderer)?
  }
 )>
<!ATTLIST CompositeFrame
  id CDATA          #REQUIRED
  width CDATA          #IMPLIED
  height CDATA         #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  >
```

Attribute definitions

- **id** = text[CS]
  
  ID of the element.

- **width** = length[CN]
  
  Width of the frame in pixels.

- **height** = length[CN]
  
  Height of the frame in pixels

  **Caution:** roleId attribute must not be specified fpr a frame element.

Options

```xml
<Option theme="base" param="target"`
```
value="_blank | _self | _parent | _top |
<window name/>

The option sets the target of every action or hyperlink unless overridden. It can only be used in DCA Web.

Examples

<SimpleFrame id="test">
  <Option theme="base" param="target" value="_parent"/>
</SimpleFrame>

Features

Concrete Web Element
com.ptc.core.ca.web.client.frame.WebCompositeFrame

Container Element
yes

Java Renderers
com.ptc.core.ca.web.client.frame.WebCompositeFrameRenderHandler

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/frame/compositeFrame.jsp
**CompositeLayout**

**Description**

The CompositeLayout element allows placement of various elements using a layout mechanism conceptually similar to `<Table>` HTML tag. The CompositeLayout element will place an element in the correct location on the screen based on the information configured in each LayoutRow element.

The CompositeLayout allows more complex faceplate behavior by supporting composite component which allows generic rendering of any simple layout sub-elements as children of optional HeaderLayout and TopLayout elements. SimpleLayout allows only standard faceplate rendering.

**Syntax**

```xml
<!ELEMENT CompositeLayout
  (HeaderLayout?, TopLayout?, LayoutRow+, On*,
   (DrawHandler | (ModelHandler?,
     (RenderHandler | Renderer)?
   )
 )
 )>
<!ATTLIST CompositeLayout
 id CDATA               #IMPLIED
 %dimensions; #IMPLIED
 %conditions; #IMPLIED
 %span; #IMPLIED
 >
```

**Attribute definitions**

- **id** = text
  - The ID of the element.

**Features**

- **Concrete Web Element**
  - com.ptc.core.ca.web.client.layout.WebCompositeLayout

- **Container Element**
  - yes

- **Java Renderers**
  - com.ptc.core.ca.web.client.layout.WebCompositeLayoutRenderHandler

- **JSP Renderers**
  - /wtcore/jsp/com/ptc/core/ca/web/layout/compositeLayout.jsp
CompositeNavBar

Description

The CompositeNavBar element represents a menu or a navigation bar. Conceptually, it contains a tree of NavBarAction elements grouped together using NavBarRow elements. At run time it determines "current" action by comparing its location to the location of its frame. Once current action is determined, a list of NavBarRow elements containing ancestor actions is constructed and drawn. This gives a visual appearance of a context sensitive navigation bar.

In terms of rendering the rows defined in a CompositeNavBar, the default behavior is the same as the CompositeLayout. The difference between using a CompositeNavBar element and a CompositeLayout element is that the CompositeNavBar element expects to have actions and is controlled by a set of business objects that are used to determine availability of the actions in the navigation bar.

The CompositeNavBar allows more complex faceplate behavior by supporting composite component which allows generic rendering of any simple layout sub-elements as children of optional HeaderLayout and TopLayout elements. SimpleNavBar allows only standard faceplate rendering.

Syntax

```xml
<!ELEMENT  CompositeNavBar
  (HeaderLayout?, TopLayout?, NavBarRow+, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
  )
 )>

<!ATTLIST CompositeNavBar
  id CDATA     #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span;       #IMPLIED
>
```

Attribute definitions

- **id** = [text][CS]
  - The ID of the element.

Features

- **Concrete Web Element**
  - com.ptc.core.ca.web.client.menu.WebCompositeNavBar

- **Container Element**
  - yes
**Java Renderers**

com.ptc.core.ca.web.client.layout.WebCompositeNavBarRenderHandler.java

**JSP Renderers**

/ wtcore/jsp/com/ptc/core/ca/web/menu/compositeNavBar.jsp
**CompositePropertyPanel**

**Description**

The CompositePropertyPanel is a so-called instance holder element as it has associated Info*Engine element. It also extends a CompositeLayout element and this allows placement of various elements using a general layout mechanism.

Generally, CompositePropertyPanel element is used to create a property sheet for a business object. By using a CompositePropertyPanel, the Property elements children of the LayoutRow elements can access the attribute business data from the CompositePropertyPanel element.

The CompositePropertyPanel allows more complex faceplate behavior by supporting composite component which allows generic rendering of any simple layout sub-elements as children of optional HeaderLayout and TopLayout elements. SimplePropertyPanel only standard faceplate rendering.

**Syntax**

```xml
<!ELEMENT  CompositePropertyPanel
  (HeaderLayout?, TopLayout?, LayoutRow*, On*,
   (DrawHandler |
    ((ModelHandler | ModelDoer)?,
    (RenderHandler | Renderer)?
   )
  )
)>
<!ATTLIST CompositePropertyPanel
  id CDATA                     #IMPLIED
  %dimensions;                 #IMPLIED
  %conditions;                 #IMPLIED
  %span;                       #IMPLIED
  >
```

**Attribute definitions**

`id = text[CS]`

The ID of the element.

**Features**

**Concrete Web Element**

com.ptc.core.ca.web.client.propertyPanel.WebCompositePropertyPanel

**Container Element**

yes

**Java Renderers**

com.ptc.core.ca.web.client.propertyPanel.WebCompositePropertyPanelRendererHandler
JSP Renderers

/wtcore/jsp/com/ptc/core/ca/web/propertyPanel/compositePropertyPanel.jsp
CompositeTable

Description

The CompositeTable element defines the configuration of a table. It allows specification of the type of objects to support, columns to display in the table, and a ModelDoer to initialize the table when it is invoked. The CompositeColumn elements determine how the column header and cell of a column of each row in the table is rendered. The CompositeColumn element allows a more complex rendering of a column header than the SimpleColumn element.

The CompositeTable allows more complex faceplate behavior by supporting composite component which allows generic rendering of any simple layout sub-elements as children of optional HeaderLayout and TopLayout elements. SimpleTable allows only standard faceplate rendering.

Syntax

```xml
<!ELEMENT  CompositeTable
  (HeaderLayout?, TopLayout?, Footer?,
   SelectAllPrompt, NoItemsPrompt?,
   Type*, CompositeColumn+, On*,
   (DrawHandler |
    (ModelHandler | ModelDoer),
   (RenderHandler | Renderer)?
  )
>
<!ATTLIST CompositeTable
  id CDATA                      #IMPLIED
  selectorType  (single | multi | none)  "none"
  clientSideSortable (true | false)  "true"
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

Attribute definitions

id =  text[CS]

The ID of the element.

selectorType = (single | multi | none)

This attribute specifies the selection behavior of the table. In DCA Web when none if specified, no column will be rendered that supports selection.

clientSideSortable = (true | false)

This attribute specifies whether the table should sort data on the client side (servlet engine in DCA Web) or the server side (Oracle). The default value is "true" for backward compatibility reasons.
Features

**Concrete Web Element**
com.ptc.core.ca.web.client.table.WebCompositeTable

**Container Element**
yes

**Java Renderers**
com.ptc.core.ca.web.client.table.WebCompositeTableRenderHandler

**JSP Renderers**
/wtcore/jsp/com/ptc/core/ca/web/table/compositeTable.jsp
**CompositeTree**

**Description**

The CompositeTree element defines the configuration of a tree shown as a tree-table. It has all the same behavior as described for a CompositeTable.

**Syntax**

```xml
<!ELEMENT  CompositeTable (
    HeaderLayout?,
    TopLayout?,
    Footer?,
    SelectAllPrompt, NoItemsPrompt?,
    Type*,
    CompositeColumn+,
    On*,
    (DrawHandler |
        ((ModelHandler | ModelDoer),
        Renderer?)
    )
)>
```

```xml
<!ATTLIST CompositeTable
    id CDATA                      #IMPLIED
    selectorType  (single | multi | none)  "none"
    showRootNode  (true | false)  "false"
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
    %span; #IMPLIED
>
```

**Attribute definitions**

- **id** = text[CS]
  
  ID of the element.

- **selectorType** = (single | multi | none)
  
  Defines the selection behavior of the table. In DCA Web when none is specified, no column will be rendered that supports selection.

- **showRootNode** = (true | false)
  
  Defines whether to show root node. Default value is true.

**Features**

- **Concrete Web Element**
  
  com.ptc.core.ca.web.client.tree.WebCompositeTree

- **Container Element**
  
  yes

- **Java Renderers**
  
  com.ptc.core.ca.web.client.table.WebCompositeTreeRenderHandler

- **JSP Renderers**
  
  /wtcore/jsp/com/ptc/core/ca/web/table/compositeTree.jsp
**Connector**

**Description**

The Connector element allows drawing of a DCA element created based on a configuration fragment located in dynamically calculated context. This is fundamentally different from the Insert element that can only use statically calculated context.

Since the Connector element computes context dynamically, it’s not possible to know the attributes the connected fragment needs to be correctly drawn. Thus attempting to connect fragments that depend on the attributes of Info*Engine elements, but do not perform the data acquisition, will most likely fail with an exception at run time. Property panels, table, trees and positioning elements like layouts and lists containing them can, on the other hand, be connected safely.

Connector cannot refer to a Label, Tooltip, ElementGroup and etc. as such configuration elements do not have correspondent DCA elements. Only configuration elements that have correspondent DCA elements can be referred to by the Connector.

**Syntax**

```xml
<!ELEMENT  Connector (NewContext)>
<!ATTLIST Connector
    id CDATA #IMPLIED
    ref CDATA #REQUIRED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
    %span; #IMPLIED
>
```

**Attribute definitions**

- **id** = text[CS]
  
  ID of the element. The element must define a configuration of a DCA element.

- **ref** = text[CS]
  
  The ID of the configuration element to be connected.

**Features**

- **Concrete Web Element**
  
  com.ptc.core.ca.web.client.tree.WebConnector

- **Container Element**
  
  yes

- **Java Renderers**
  
  com.ptc.core.ca.web.client.primitive.WebConnectorRenderHandler
JSP Renderers

/WTcore/jsp/com/ptc/core/ca/web/primitive/connector.jsp
Context

Description
The Context element defines the context for a group of configuration elements. The context will apply for all the descendent elements unless they override it. See %dimensions for attribute description.

Syntax
<!ELEMENT Context ANY>
<!ATTLIST Context
%dimensions;
>

Context element cannot be a child of another context element.

Features

Concrete Web Element
no

Container Element
no

Java Renderers
Not available

JSP Renderers
Not available
Copy

Description

The Copy element allows the copying values from the source to the target.

Syntax

```xml
<!ELEMENT Copy EMPTY>
<!ATTLIST Copy
  id CDATA #IMPLIED
  from CDATA #REQUIRED
  to CDATA #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  ID of the element.
- **from** = text[CS]
  The source value.
- **to** = text[CS]
  The target.

Examples

```xml
<Action id="search3.enumerator.type.change">
  <Label resource="Change Type"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.
    WebInPlaceActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.doer.
    CoNoOpActionDoer">
    <On event="executeModelDoer" at="end">
      <Copy from="%[enum#search3.enumerator.type]"
        to="%[frame#com.ptc.search.type_param]"/>
    </On>
  </ClientDoer>
</Action>
```

Features

- **Concrete Web Element**
  no
- **Container Element**
  no
- **Java Renderers**
  Not available
JSP Renderers
Not available
**DataDrivenEnumerator**

**Description**

The DataDrivenEnumerator element allows the configuration of a list of options where a single option can be chosen as current one. The element is always bound to the Info*Engine element of the nearest instance holder ancestor via an attribute containing the current value.

The list of options can be defined in a variety of ways:

1. Similar to how it’s done for Enumerator element via LabelArea elements.
2. As a value of an attribute of an Info*Engine element specified via NeedNpa configuration element as `options`. The value must contain a list of available options represented by CoConfigFeature objects.
3. As a result of a task execution.

**Syntax**

```xml
<!ELEMENT  DataDrivenEnumerator
  ( (Need | NeedNpa), LabelArea*, (Need | NeedNpa)?,
    Comparator?, On*,
    (DrawHandler |
    ((ModelHandler | ModelDoer)?,
      (RenderHandler | Renderer)?
    )
  )
>)
<!ATTLIST DataDrivenEnumerator
  id CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

**Attribute definitions**

`id = text` [CS]

ID of the element.

**Examples**

Example #1:

```xml
<DataDrivenEnumerator id="enum">
  <LabelArea id="1" labelType="text"
    value="1"
    default="true">
    <Label resource="Option 1"/>
  </LabelArea>
  <LabelArea id="2" labelType="text"
    value="2">
  </LabelArea>
</DataDrivenEnumerator>
```
In this example the current value of the enumerator is bound to a value of the attribute `enum_value`. The list of LabelArea element is used to construct the list of possible values.

Example #2:

```xml
<DataDrivenEnumerator id="enum">
  <NeedNpa attribute="enum_options" as="options" />
  <Need attribute="enum_value" as="value" />
</DataDrivenEnumerator>
```

In this example, the value of the attribute `enum_options` is a `java.util.ArrayList` object that in turn contains a list of `CoConfigFeature` objects describing enumerator options (LabelAreas). The options are not defined in the configuration specification.

The current value of the enumerator is bound to the value of the attribute `enum_value`.

Example #3:

```xml
<DataDrivenEnumerator id="enum">
  <LabelArea id="1" labelType="text"
    value="1"
    default="true">
    <Label resource="-- Choose One --"/>
  </LabelArea>
  <LabelArea id="2" labelType="text"
    value="2">
    <Label resource="-- Use default --"/>
  </LabelArea>
  <ModelDoer class="com.ptc.core.ca.co.client.primitive.
    CoDataDrivenEnumeratorModelTaskDoer"
    labelAttribute="name">
    <Task nextOperation="STDOP|com.ptc.windchill.view"
      taskDelegate="dca-get-single-object">
      <TaskParameter name="search_type" value="wt.folder.Cabinet"/>
      <TaskParameter name="where" computedValue="com.ptc.refimpl.co.computedvalues.
        PersonalFolderWhereComputedValue"/>
      <TaskParameter name="content_type" value="WCTYPE|wt.doc.WTDocument"/>
    </Task>
  </ModelDoer>
  <NeedNpa attribute="enum_options" as="options" />
  <Need attribute="enum_value" as="value" />
</DataDrivenEnumerator>
```

In this example, the list of options will consist of two options specified by LabelArea elements and options created using the Info*Engine elements from a
task result. DCA will use values of name attribute of Info*Engine elements as labels to construct options and UFIDs of the elements as values.

Sometimes it’s desirable to use a value of an attribute as an option value instead of using element’s UFID. In this case the name of the attribute can be specified via an additional valueAttribute attribute.

```xml
<ModelDoer class="com.ptc.core.ca.co.client.primitive.CoDataDrivenEnumeratorModelTaskDoer"
    labelAttribute="name"
    valueAttribute="name">
```

Features

**Concrete Web Element**

com.ptc.core.ca.web.client.primitive.WebDataDrivenEnumerator

**Container Element**

yes

**Java Renderers**

com.ptc.core.ca.web.client.primitive.WebEnumeratorRenderHandler

**JSP Renderers**

/WT/core/jsp/com/ptc/core/ca/web/primitive/enumerator.jsp
DrawHandler

Description

This DrawHandler element specifies a Java class to be responsible for drawing the element. It can be specified for any element that maps to a concrete WebElement class. The implemented Java class must implement the CoDrawHandler interface. If a DrawHandler is configured, it assumes ALL responsibility for drawing the element. This includes creating children components and determining how children are drawn (generally by telling them to draw themselves).

Syntax

```xml
<!ELEMENT  DrawHandler EMPTY >
<!ATTLIST DrawHandler
  id CDATA                      #IMPLIED
  class CDATA                   #REQUIRED
  %dimensions;                  #IMPLIED
  %conditions;                  #IMPLIED
  >
```

Attribute definitions

- **id** =  text[CS]
  The ID of the element.

- **class** =  text[CS]
  The fully qualified name of the Java class that implements a draw handler. The handler must implement `com.ptc.core.ca.co.client.element.CoDrawHandler` interface.

Features

- **Concrete Web Element**
  no

- **Container Element**
  no

- **Java Renderers**
  Not available

- **JSP Renderers**
  Not available
Else

Description
The Else element defines a default condition. The element can only be used as a child of If element. The element tag is case insensitive.

Syntax
```xml
<!ELEMENT Else ((Copy | Remove)+)
>
<!ATTLIST Else
  id CDATA                    #IMPLIED
  %dimensions;               #IMPLIED
  %conditions;              #IMPLIED
>
```

Attribute definitions
id = text[CS]
The ID of the element.

Examples
```xml
<On event="executeModelDoer" at="end">
  <if condition="%request#x1]==1">
    <Copy from="one" to="%request#message"/>
  </if>
  <elsif condition="%request#x1]==2">
    <Copy from="two" to="%request#message"/>
  </elsif>
  <else/>
    <Copy from="three" to="%request#message"/>
  </else>
</On>
```

Features
Concrete Web Element
no

Container Element
no

Java Renderers
Not available

JSP Renderers
Not available
Elsif

Description
The Elsif element defines an alternative condition. The element can only be used as a child of If element. The element tag is case insensitive.

Syntax
<!ELEMENT Elsif ((Copy | Remove)+)>
<!ATTLIST Elsif
   id CDATA                    #IMPLIED
   condition CDATA   #REQUIRED
   %dimensions; #IMPLIED
   %conditions; #IMPLIED
>

Attribute definitions
- **id** = text
  - The ID of the element.
- **condition** = text
  - Alternative condition..

Examples
<On event="executeModelDoer" at="end">
  <if condition="%[request#x1]==1">
    <Copy from="one" to="%[request#message]"/>
    <elsif condition="%[request#x1]==2"/>
    <Copy from="two" to="%[request#message]"/>
    <else/>
    <Copy from="three" to="%[request#message]"/>
  </if>
</On>

Features
- **Concrete Web Element**
  - no
- **Container Element**
  - no
- **Java Renderers**
  - Not available
- **JSP Renderers**
  - Not available
**EmbeddedURLContent**

**Description**

The EmbeddedURLContent element displays the content referenced by the specified URL. It configuration is similar to HyperLink element.

**Syntax**

```xml
<!ELEMENT  EmbeddedURLContent
  (HRef, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?)
   )
  )>

<!ATTLIST EmbeddedURLContent
  id CDATA  #IMPLIED
  width CDATA          #IMPLIED
  height CDATA          #IMPLIED
  border CDATA          #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

**Attribute definitions**

- **id** = text[CS]
  
  The ID of the element.

- **width** = length[CN]
  
  The width of the rectangle containing URL content.

- **height** = length[CN]
  
  The height of the rectangle containing URL content.

- **border** = number[CN]
  
  The thickness of the border enclosing URL content.

**Example**

The following example include the content of $(base_url)/wtcore/jsp/portal/index.jsp?a=b into a rectangular area of the specified size. The renderer is implemented using IFrame HTML element in DCA Web.

```xml
<EmbeddedURLContent width="100%" height="100%" border="3">
  <HRef    resource="/wtcore/jsp/portal/index.jsp"/>
  <ModelHandler class="com.ptc.core.ca.co.client.windchill.
    CoWindchillHyperLinkProducer">
    <Option theme="param" name="a" value="b"/>
  </ModelHandler>
</EmbeddedURLContent>
```
Features

Concrete Web Element
com.ptc.core.ca.web.client.primitive.WebEmbeddedURLContent

Container Element
no

Java Renderers
com.ptc.core.ca.web.client.primitive.WebIFrameRenderHandler

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/primitive/iframe.jsp
**Enumerator**

**Description**

The Enumerator element defines the configuration for a set of values and lets the user to choose one of them. Each LabelArea sub-elements must have a value specified via value attribute and only one of them can be a default one.

**Syntax**

```
<!ELEMENT  Enumerator
  (LabelArea+, On*,
   (DrawHandler |
   (ModelHandler, Renderer? )
  )
 )>
<!ATTLIST Enumerator
  id CDATA                      #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

**Attribute definitions**

- **id** = text[CS]
  - ID of the element.

**Examples**

```xml
<Enumerator id="search2.enumerator.type">
  <LabelArea id="search2.enumerator.type.part" labelType="text"
    value="part"
    default="true">
    <Label resource="Part"/>
    <ToolTip resource="Part"/>
  </LabelArea>
  <Insert ref="search2.enumerator.type.doc" value="doc"/>
  <Insert ref="search2.enumerator.type.folder" value="folder"/>
</Enumerator>
```

**Features**

- **Concrete Web Element**
  - com.ptc.core.ca.web.client.primitive.WebEnumerator

- **Container Element**
  - yes

- **Java Renderers**
  - com.ptc.core.ca.web.client.primitive.WebEnumeratorRenderHandler
JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/primitive/enumerator.jsp
Extend

Description
The Extend element allows a configuration to inherit and override the definition of another configuration element thus allowing existing element definitions to be reused. The new element will be a copy of the extended element with the ID, attributes and sub elements overridden by the definitions in the inheriting element.

Syntax
<!ELEMENT Extend EMPTY>
<!ATTLIST Extend
  ref CDATA    #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>

Attribute definitions
Ref = text[CS]
This attribute is the ID of the configuration element to extend.

Examples
Example #1
<Repository package="com.ptc.folder">
  <Import package = "com.ptc.common/> 
  <Import package = "com.ptc.enterprise"/>
  ...
  <Property id="property.name">
    <Extend ref="common:property.name.template"/>
    <Tooltip resource="Part name"/>
  </Property>
  <Property id="property.number">
    <Extend ref="common:property.number.template">
      <Renderer = resource="/wcore/jsp/...myrenderer.jsp"/>
    </Extend>
  </Property>
</Repository>

Example #2
[Extracted from com\ptc\common\config_common_templates.xml]
<Property id="property.String.template">
  <PropertyHandler class="com.ptc.core.ca.web.client.property.
    string.WebStringPropertyHandler"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
    property/string/string.jsp"/>
</Property>
The element named "property.name.template" evaluates to:

```
<Property id="property.name.template">
  <PropertyHandler class="com.ptc.core.ca.web.client.property.
    string.WebStringPropertyHandler"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
    property/string/string.jsp"/>
  <Label resource="The name:"/>
  <Need attribute=":attribute.name"/>
</Property>
```

The reference to the extend element is lost.

An element can have multiple extend statements and they can be placed anywhere in the element. The position of the extend tag controls the order of the elements when the final elements are created from merging the hierarchy of the elements. For example:

```
<SimplePropertyPanel id="newPanel">
  <Property id="newFirstProperty"/>
  <Extend ref="OldPanel_1"/>
  <Property id="newMiddleProperty"/>
  <Extend ref="OldPanel_2"/>
  <Property id="newLastProperty"/>
</SimplePropertyPanel>
```

The SimplePropertyPanel display property in the order the properties are defined in the panel. The above sample configuration would create a "newPanel" SimplePropertyPanel that display

"newFirstProperty" property
the merged properties from "OldPanel_1"
"newMiddleProperty" property
the merged properties from "OldPanel_2"
"newLastProperty".

Extend does not have to extend a similar element, you are free to introduce elements with new element tags to be used as base elements in Extend statements as long as the resultant element is valid.

**Features**

**Concrete Web Element**

no

**Container Element**

no
Java Renderers
Not available

JSP Renderers
Not available
ElementGroup

Description

The ElementGroup element allows you to group multiple elements together to reuse their definitions without cloning (cut/paste). When inserted, members of the element group will be children of the element that contains the group.

Syntax

```xml
<!ELEMENT  ElementGroup  (Any+)>
<!ATTLIST ElementGroup
  ref CDATA    #REQUIRED
  %dimensions;  #IMPLIED
  %conditions;  #IMPLIED
>
```

Attribute definitions

Ref = text[CS]

This attribute is the ID of the configuration element to extend.

Examples

```xml
<ElementGroup id="folder.browser.actions">
  <Insert ref="action.cut"/>
  <Insert ref="action.copy"/>
  <Insert ref="action.paste"/>
</ElementGroup>
...

<ActionList id="folder.actions">
  <Insert ref="folder.browser.actions"/>
</ActionList>
```

Features

Concrete Web Element
no

Container Element
no

Java Renderers
Not available

JSP Renderers
Not available
Footer

Description

The Footer element is a standard SimpleLayout Element used to configure a footer of a table or a tree element. The Footer is displayed below the element it belongs to.

Syntax

```xml
<!ELEMENT Footer
  (LayoutRow+, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
   )
  )>
<!ATTLIST Footer
  id CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  
  The ID of the element.

Features

- **Concrete Web Element**
  
  com.ptc.core.ca.web.client.layout.WebSimpleLayout

- **Container Element**
  
  yes

- **Java Renderers**
  
  com.ptc.core.ca.web.client.layout.WebSimpleLayoutRenderHandler

- **JSP Renderers**
  
  /wtcore/jsp/com/ptc/core/ca/web/layout/simpleLayout.jsp
For

Description

The For element allows the creation of a cell content element to be substituted depending on the type of the Info*Engine element a row represents.

Syntax

```xml
<!ELEMENT  For EMPTY
>
<!ATTLIST CompositeColumn
    type CDATA                 #REQUIRED
    show CDATA        #REQUIRED
    %dimensions;       #IMPLIED
    %conditions;       #IMPLIED
>
```

Attribute definitions

- **type** =  `text[CS]`
  Type of the business object (Info*Engine element).

- **show** =  `text[CS]`
  The ID of the element to draw in the cell for this column in each row of the table for a specified type. Generally this is the ID of a Property, an ActionList, or an Action. However, there are very few restrictions on the type of element that can be drawn in the cell.

Features

- **Concrete Web Element**
  no

- **Container Element**
  no

- **Java Renderers**
  Not available

- **JSP Renderers**
  Not available
FrameHeader

Description

The FrameHeader element allows configuring a header layout for a CompositeFrame element in addition to the standard frame child element supported by both simple and composite frame elements. FrameHeader behaves identically to the SimplePropertyPanel. It allows any number of LayoutRows to be specified and is only allowed as a sub element of CompositeFrame. The concrete class of the FrameHeader is a WebSimplePropertyPanel so it can also support an associated business object.

CompositeFrame supports similar behavior for the Link and Script element using LayoutRow in the FrameHeader. See element "subfolder.frame.header" for an example of how to define FrameHeader.

Syntax

```
<!ELEMENT  FrameHeader
    (LayoutRow+, On*,
    (DrawHandler |
    ((ModelHandler  | ModelDoer),
    (RenderHandler | Renderer)?
    )
    )
    )>

<!ATTLIST FrameHeader
    id CDATA #IMPLIED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
```

Attribute definitions

id =  text[CS]

The ID of the element.

Features

Concrete Web Element
com.ptc.core.ca.web.client.propertyPanel.WebSimplePropertyPanel

Container Element
yes

Java Renderers
com.ptc.core.ca.web.client.frame.WebFrameHeaderRenderHandler

JSP Renderers
/ wtcore/jsp/com/ptc/core/ca/web/frame/frameHeader.jsp
HeaderLayout

Description

The HeaderLayout element is a standard SimpleLayout Element used to configure layout component in composite elements (CompositeXxx) or a Wizard. In a composite element the HeaderLayout is displayed above the TopLayout.

Syntax

```xml
<!ELEMENT  HeaderLayout
  (LayoutRow+, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
  )>
<!ATTLIST HeaderLayout
  id CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  >
```

Attribute definitions

- **id** =  **text[CS]**
  The ID of the element.

Features

- **Concrete Web Element**
  com.ptc.core.ca.web.client.layout.WebSimpleLayout

- **Container Element**
  yes

- **Java Renderers**
  com.ptc.core.ca.web.client.layout.WebSimpleLayoutRenderHandler

- **JSP Renderers**
  /wtcore/jsp/com/ptc/core/ca/web/layout/simpleLayout.jsp
HRef

Description

The HRef element (capital "H" and capital "R") defines a resource for a hyper link element.

Syntax

```xml
<!ELEMENT  HRef EMPTY
    >
<!ATTLIST HRef
    id CDATA #IMPLIED
    resource %URI #REQUIRED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
    >
```

Attribute definitions

- **id** = text
  
  The ID of the element

- **resource** = uri
  
  URI of hyper link. It can be
  
  - absolute e.g. "http://www.ptc.com"
  
  - relative e.g "view.html"
  
  - reference to an anchor e.g. "#anchor_name"
  
  - javascript call using "javascript:...".

Features

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available

- **JSP Renderers**
  
  Not available
**HyperLink**

**Description**

The HyperLink element displays a hyper link.

**Syntax**

```
<!ELEMENT HyperLink
    ( (HRef | Anchor), Label?, Image?, ToolTip?, On*,
    (DrawHandler | (ModelHandler?,
    (RenderHandler | Renderer)?
    )
    )
    )>
```

```
<!ATTLIST HyperLink
    id CDATA #IMPLIED
    labelType (image|text|textImage,imageText) "imageText"
    drawType (both |image |text) "both"
    frameType (same | new ) "same" -- DEPRECATED --
    window (same | new| parent | top) "same"
    width CDATA          #IMPLIED
    height CDATA          #IMPLIED
    %dimensions;          #IMPLIED
    %conditions;          #IMPLIED
    %span;                #IMPLIED
    >
```

**Attribute definitions**

**id = text[CS]**  
The ID of the element.

**labelType= (image|text|textImage,imageText)**  
Defines how to draw a hyper link. The default is "imageText" - image on the left, text on the right.

**drawType = (both | image | text)**  
Defines what can be clicked to activate a hyper link. The default is "both" - image and label. Can be specified when labelType is either textImage or imageText.

**frameType= (same | new ) -- DEPRECATED --**  
[Deprecated. Replace by window.] Defines how to navigate to a resource when user activate hyper link. The default is "same" - navigate in the same window.

**window= (same | new | parent | top)**  
Defines how to navigate to a resource when user activate hyper link. The default is "same" - navigate in the same window.
width = number
The width of the new window in pixels. Used only when frame type is new.

height = number
The height of the new window in pixels. Used only when frame type is new.

Features

Concrete Web Element
com.ptc.core.ca.web.client.primitive.WebHyperlink

Container Element
no

Java Renderers
com.ptc.core.ca.web.client.primitive.WebHyperLinkRenderHandler
com.ptc.core.ca.web.client.primitive.WebHyperLinkLabelRenderHandler
com.ptc.core.ca.web.client.primitive.WebHyperLinkRenderHandler

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/primitive/hyperLink.jsp
/wtcore/jsp/com/ptc/core/ca/web/primitive/hyperLinkImage.jsp
/wtcore/jsp/com/ptc/core/ca/web/primitive/hyperLinkLabel.jsp
If

Description

The If element defines the beginning of a conditional statement. The element can only be used inside an event handler. Conditional statements can be nested. The element tag is case insensitive.

Syntax

```
<!ELEMENT If ((Copy | Remove)+, Elsif*, (Elsif, Else)?
   >
<!ATTLIST If
   id CDATA                    #IMPLIED
   condition CDATA               #REQUIRED
   %dimensions; #IMPLIED
   %conditions; #IMPLIED
   >
```

Attribute definitions

- **id** =  **text**
  
  The ID of the element.

- **condition** =  **text**
  
  Condition.

Examples

```
<On event="executeModelDoer" at="end">
  <if condition="%[request#x1]==1">
    <Copy from="one" to="%[request#message]"/>
  </if>
  <elsif condition="%[request#x1]==2">
    <Copy from="two" to="%[request#message]"/>
  </elsif>
  <else/>
    <Copy from="three" to="%[request#message]"/>
  </else>
</On>
```

Features

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available

- **JSP Renderers**
  
  Not available
Image

Description

The Image element defines a URI to an image.

Syntax

```xml
<!ELEMENT  Image EMPTY
>
<!ATTLIST Image
   id CDATA                    #IMPLIED
   resource  %URI              #REQUIRED
   %dimensions; #IMPLIED
   %conditions; #IMPLIED
>
```

Attribute definitions

```xml
id =  text[CS]
   The ID of the element.

resource =  uri[CT]
   URI of Image to be displayed.
```

Features

Concrete Web Element

no

Container Element

no

Java Renderers

Not available

JSP Renderers

Not available
**Import**

**Description**

The Import element allows referencing identifiers defined in another package in the current package. It is conceptually similar to an import statement in Java. Using import allows using just the "package name:id" to reference a named configuration element instead of the "full package name:id" reference. Using multiple IMPORT elements allows multiple packages to be import into the current package.

**Syntax**

```xml
<!ELEMENT Import EMPTY>
<!ATTLIST Import
    package CDATA #REQUIRED>
```

**Attribute definitions**

`package = text[CS]`

The fully qualified name of the package imported into the current package.

**Examples**

```xml
<Repository package="com.ptc.folder">
    <Import package = "com.ptc.common"/>
    <Import package = "com.ptc.enterprise"/>
    ...
    <Property id="property.name">
        <Extend ref="common:property.name.template"/>
    </Property>
</Repository>
```

**Features**

**Concrete Web Element**

no

**Container Element**

no

**Java Renderers**

Not available

**JSP Renderers**

Not available
**Init**

**Description**

The Init element defines a set of operations that need to be executed when the configuration system is loaded on startup.

**Syntax**

```xml
<!ELEMENT Init (AppInstallationStatusDeterminator*)>
<!ATTLIST Init
   id CDATA #IMPLIED>
```

**Attribute definitions**

- **id** = text
  - The ID of the element. Must always be equal to "Init"

**Examples**

```xml
<Context>
  <Init id="Init">
    <AppInstallationStatusDeterminator class="com.ptc.core.ca.co.
  client.misc.CoDefaultAppInstallationStatusDeterminator"
       appId="Applicaton A" version="6.2.6/>
    <AppInstallationStatusDeterminator class="com.ptc.core.ca.co.
  client.misc.CoDefaultAppInstallationStatusDeterminator"
       appId="Applicaton B" version="5.1.3"/>
  </Init>
</Context>
```

When DCA reads configuration specifications it will set up the installation status as follows:

Applicaton A version 6.2.6
Applicaton B version 5.1.3

**Features**

- **Concrete Web Element**
  - no
- **Container Element**
  - no
- **Java Renderers**
  - Not available
- **JSP Renderers**
  - Not available
Insert

Description

The Insert element allows a configuration element to aggregate another configuration element in its definition. This allows existing configuration element definitions to be reused without cloning (cut/paste) of the configuration element. The insert element can have any legal sub-element or attributes that are valid for the element that is inserted. The insert elements can be used as many times as necessary to aggregate the correct sub elements.

Syntax

```xml
<!ELEMENT  Insert
             (ANY*) >
<!ATTLIST Insert
             ref CDATA    #REQUIRED
```

Attribute definitions

`ref = text[CS]`

The ID of the configuration element to be inserted.

Examples

[Extracted from com\ptc\common\config_common_templates.xml]

```xml
<WizardActions id="picker.actions">
  <Insert ref="action.picker.ok"/>
  <Insert ref="action.wizard.cancel"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
               action/horizontallist.jsp"/>
</WizardActions>

<Action id="action.wizard.ok">
  <Label resource="Ok"/>
  <ToolTip resource="Ok"/>
  <ActionHandler class="com.ptc.core.ca.web.client.
                   wizard.WebOkActionHandler"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
               wizard/wizard_ok_apply.jsp"/>
</Action>

<Action id="action.wizard.cancel">
  <Label resource="Cancel"/>
  <ToolTip resource="Cancel"/>
  <ActionHandler class="com.ptc.core.ca.web.client.
                   wizard.WebCancelActionHandler"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/
               /wizard/wizard_cancel.jsp"/>
</Action>
```

An equivalent configuration definition would be:
The Insert allows attributes as well as sub-elements of the inserted element to be overridden. The following is an example.

[Extracted from com\ptc\common\config_common_templates.xml]

```xml
<Action id="action.goTo.template">
    <ActionHandler class="com.ptc.core.ca.web.client.action.WebSimpleGoToActionHandler"/>
    <Renderer jsp="/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp"/>
</Action>
```

The next example creates an ActionList using an action defined in a previous example and overriding sub-elements and attributes. The id attribute is overridden with a new value. While new elements are added to the new create action element, none of the sub-elements are overridden because none where defined in parent element.

```xml
<ActionList>
    <Insert ref="action.goTo.template" id="action.folder.view" needSelection="true">
        <Label id="new label"/>
        <ToolTip id="new tooltip"/>
        <Location id="page.view"/>
    </Insert>
</ActionList>
```

The resultant element is:

```xml
<Action id="action.folder.view" needSelection="true">
    <ActionHandler class="com.ptc.core.ca.web.client.action.WebSimpleGoToActionHandler"/>
```
Example:

```xml
<ActionList id="folder.browser.actions">
  <Insert ref="common:action.cut"/>
  <Insert ref="common:action.copy"/>
</ActionList>
...
<LayoutRow>
  <Insert ref="folder.browser.actions"/>
  <Insert ref="action.paste"/>
</Insert>
</LayoutRow>
```

An equivalent configuration definition would be:

```xml
<LayoutRow>
  <Insert ref="common:action.cut"/>
  <Insert ref="common:action.copy"/>
  <Insert ref="action.paste"/>
</LayoutRow>
```

Features

**Concrete Web Element**

No

**Container Element**

No

**Java Renderers**

Not available

**JSP Renderers**

Not available
InstallationStatusDeterminator

Description

The InstallationStatusDeterminator element specifies a class that will be used to set up the installation status of an application.

Syntax

```xml
<!ELEMENT InstallationStatusDeterminator EMPTY>
<!ATTLIST InstallationStatusDeterminator
  id CDATA #IMPLIED
  class CDATA #REQUIRED>
```

Attribute definitions

- `id = text[CS]`
  The ID of the element.
- `class = text[CS]`
  The fully qualified name of the class that implements an installation status determinator. The class must implement `com.ptc.core.ca.co.client.misc.CoInstallationStatusDeterminator` interface.

Examples

```xml
<Context>
  <Init id="Init">
    <InstallationStatusDeterminator class="com.ptc.core.ca.co.client.misc.CoDefaultAppInstallationStatusDeterminator"
      appId="Applicaton A" version="6.2.6"/>
    <InstallationStatusDeterminator class="com.ptc.core.ca.co.client.misc.CoDefaultModuleInstallationStatusDeterminator"
      appId="Module B" version="5.1.3"/>
  </Init>
</Context>
```

When DCA reads configuration specifications it will set up the installation status as follows:

Applicaton A version 6.2.6
Module B version 5.1.3

Features

- **Concrete Web Element**
  no
- **Container Element**
  no
Java Renderers
   Not available
JSP Renderers
   Not available
InstallationSet

Description

The InstallationSet element defines a set of applications and modules that can be used to check for their installation status.

Syntax

<!ELEMENT  InstallationSet (Application*, Module*)>
<!ATTLIST InstallationSet
  id CDATA                    #IMPLIED
  %dimensions;                #IMPLIED
  %conditions;                #IMPLIED
>

Attribute definitions

id =  text[CS]

The ID of the element.

Examples

<Context>
  <InstallationSet id="set1">
    <Application appId="Applicaton A" from="6.0.2" to="6.2.8"/>
    <Application appId="Applicaton B" from="4.2"/>
    <Module moduleId="Module C" to="7.0"/>
  </InstallationSet>
</Context>

Features

Concrete Web Element
  no

Container Element
  no

Java Renderers
  Not available

JSP Renderers
  Not available
Label

Description
The Label element defines the text to display. This can be used differently depending on the parent element using the label. It usually depends on the faceplate behavior of the parent component to determine where the text is displayed. Label can perform substitution based on the Need sub element.

Syntax
<!ELEMENT  Label
  (Need?)>
<!ATTLIST Label
  id CDATA #IMPLIED
  resource CDATA #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>

Attribute definitions
id =  text[CS]
The ID of the element.

resource =  text/CS/
String to display. If the value is of the form "aaa {0} bbb", this assumes that a Need sub element is specified. The value for an attribute specified in the need element will be substituted for the "{0}" portion of the resource. When more than one attribute value is needed for substitution use "aaa {0} bbb {1} ccc" and specify the appropriate number of Need statements.

Features

Concrete Web Element
no

Container Element
no

Java Renderers
Not available

JSP Renderers
Not available
LabelArea

Description

The LabelArea element displays text, image, or both.

Syntax

```xml
<!ELEMENT LabelArea
  (Label?, Image?, ToolTip?, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
   )
 )
 )>
<!ATTLIST LabelArea
  id CDATA                     #IMPLIED
  labelType  (text |image | imageText | textImage) #IMPLIED
  value CDATA                   #IMPLIED
  %dimensions;                   #IMPLIED
  %conditions;                   #IMPLIED
  %span;                           #IMPLIED
">
```

Attribute definitions

- **id = text[CS]**
  The ID of the element.

- **labelType = (text|image|imageText | textImage)**
  Defines how to draw a label area element. The default is "imageText" - image on the left, text on the right.

- **value = text[CS]**
  The value associated with the label area. This attribute is only used when configuring enumerators.

Options

```xml
<Option theme="font" param="style"
  value="(italic | normal)"/>
<Option theme="font" param="weight"
  value="(bold | normal)"/>
<Option theme="font" param="color"
  value="( <hex value> | <color name> )"/>
<Option theme="font" param="size"
  value="(larger | smaller | xx-small | x-small |
   small | medium | large | x-large | xx-large)"/>
<Option theme="font" param="wrap"
  value="(nowrap | normal)"/>
<Option theme="css" param="class"
  value="(<any valid css class name>)"/>
<Option theme="size" param="height | width"
```
Examples

<LabelArea id="example" labelType="imageText">
  <Label resource="some words"/>
  <Image resource="/wtcore/images/com/ptc/core/ca/web/misc/cut.gif"/>
  <Option theme="font" param="style" value="italic"/>
</LabelArea>

Features

Concrete Web Element
com.ptc.core.ca.web.client.primitive.WebLabel

Container Element
no

Java Renderers
com.ptc.core.ca.web.client.primitive.WebLabelRenderHandler
com.ptc.core.ca.web.client.primitive.WebLabelImageRenderHandler
com.ptc.core.ca.web.client.primitive.WebImageLabelRenderHandler
com.ptc.core.ca.web.client.primitive.WebImageRenderHandler

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/primitive/image.jsp
/wtcore/jsp/com/ptc/core/ca/web/primitive/labelImage.jsp
/wtcore/jsp/com/ptc/core/ca/web/primitive/imageLabel.jsp
/wtcore/jsp/com/ptc/core/ca/web/primitive/image.jsp
LayoutRow

Description

The LayoutRow is a sub-element of a layout container responsible for displaying a row of elements. It is conceptually similar to <TR> HTML tag. It can have any number of child elements.

If only a single LayoutRow element is defined in a Layout container, its ID can be omitted. If multiple LayoutRow elements are defined, IDs are required so correct addresses are generated for sub-elements in the component hierarchy. The reason IDs are required if multiple rows are specified is the address scheme used to find elements in the hierarchy. If only a single layout row is used, the tag of element itself can be used in generating the address of the element. With multiple rows, the ID must be specified to get a unique address.

Syntax

```xml
<!ELEMENT  LayoutRow
  (Action*, ActionList*, HyperLink*, LabelArea*,
   List*, Property*, Separator*,
   Connector*, On*,
   SimpleLayout*, CompositeLayout*,
   SimpleNavBar*, CompositeNavBar*,
   SimplePropertyPanel*, CompositePropertyPanel*,
   SimpleTable*, CompositeTable*,
   CompositeTree*,
   Wizard?,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
 )>

<!ATTLIST LayoutRow
   id CDATA               #IMPLIED
   %dimensions; #IMPLIED
   %conditions; #IMPLIED
>
```

Attribute definitions

- `id = text[CS]`
  
  The ID of the element.

Options

- `<Option theme="align" param="vertical" value="(top | middle | bottom)"/>
- `<Option theme="align" param="horizontal" value="(left | right | center | justify)"/>
- `<Option theme="size" param="height | width" value="(<length> | <percentage> | auto | inherit)"/>
- `<Option theme="font" param="style "

Configuration Specification Reference
value="(italic | normal)"/>
<Option theme="font" param="weight "
value="(bold | normal)"/>
<Option theme="font" param="color"
value="( <hex value> | <color name> )"/>
<Option theme="font" param="size"
value="(larger | smaller | xx-small | x-small | small |
medium | large | x-large | xx-large)"/>
<Option theme="font" param="wrap"
value="(nowrap | normal)"/>
<Option theme="css" param="class"
value="(<any valid css class name>)"/>

Example

<LayoutRow id="layout.row">
  <Option theme="align" param="vertical" value="top"/>
  ...
</LayoutRow>

Features

Concrete Web Element
com.ptc.core.ca.web.client.layout.WebLayoutRow

Container Element
yes

Java Renderers
com.ptc.core.ca.web.client.layout.WebLayoutRowRenderHandler

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/layout/layoutrow.jsp
Link

Description

The Link element renders a link in the HTML header. For example, a link to a CSS file.

Syntax

```xml
<!ELEMENT Link
   (DrawHandler |
    (ModelHandler,
     (RenderHandler | Renderer)
    )
 )>
<!ATTLIST Link
   id CDATA #IMPLIED
   resource CDATA #REQUIRED
   %dimensions; #IMPLIED
   %conditions; #IMPLIED
 >
```

Link element must use the following ModelHandler to handle its model

`com.ptc.core.ca.co.client.frame.CoHeaderResourceModelHandler`

Attribute definitions

- `id = text[CS]`
  
  The ID of the element.

- `resource = text[CS]`
  
  Resource is the href of the link.

Options

```xml
<Option theme="css"
   param="useragent | title | rel | type"
   value="<string>"/>
```

Examples

```xml
<Link id="frame.header.link.css.default.gecko"
   resource="wtcore/css/com/ptc/core/ca/web/misc/gecko.css">
   <Option theme="css" param="title" value="Default"/>
   <Option theme="css" param="rel" value="stylesheet"/>
   <Option theme="css" param="type" value="text/css"/>
   <Option theme="css" param="useragent" value="gecko"/>
   <ModelHandler class="com.ptc.core.ca.co.client.frame.
   CoHeaderResourceModelHandler"/>
   <Insert ref="common:renderer.cssLink"/>
</Link>
```
Features

**Concrete Web Element**
com.ptc.core.ca.web.client.primitive.WebLabel

**Container Element**
no

**Java Renderers**
com.ptc.core.ca.web.client.frame.WebCssLinkRenderHandler (not built in)

**JSP Renderers**
/wtcore/jsp/com/ptc/core/ca/web/frame/cssLink.jsp (not built in)
**List**

**Description**

The List element allows any arbitrary list of elements to be displayed.

**Syntax**

```xml
<!ELEMENT  List
    (Action*, ActionList*, HyperLink*, LabelArea*,
     List*, Property*, Separator*,
     Connector*, On*,
     SimpleLayout*, CompositeLayout*,
     SimpleNavBar*, CompositeNavBar*,
     SimplePropertyPanel*, CompositePropertyPanel*,
     SimpleTable*, CompositeTable*,
     CompositeTree*,
     (DrawHandler |
      (ModelHandler,
       (RenderHandler | Renderer))? )
    )>

<!ATTLIST List
    id CDATA %dimensions; #IMPLIED
    %conditions; #IMPLIED
    %span; #IMPLIED
>
```

**Attribute definitions**

```xml
id =  text[CS]
```

The ID of the element.

**Features**

**Concrete Web Element**

com.ptc.core.ca.web.client.list.WebList

**Container Element**

yes

**Java Renderers**

com.ptc.core.ca.web.client.list.WebListRenderHandler

**JSP Renderers**

`/wtcore/jsp/com/ptc/core/ca/web/list/list.jsp`
Location

Description
The Location element defines an address of the target frame for "go-to", "new frame", and "new temporary frame" actions. The action will use the ID of a target frame along with a set of dimensions (context) to form the address of the target frame. The configuration system dynamically finds configuration for the target frame based on the ID and context. So at runtime, different frames will be presented based on the context of the target frame. Sub-element NewContext helps to augment the context of the target frame.

Syntax

```xml
<!ELEMENT Location (NewContext?)>
<!ATTLIST Location
  id CDATA #IMPLIED
  ref CDATA #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED>
```

Attribute definitions

- **id** = text
  The ID of the element.

- **ref** = text
  This attribute is the ID of the frame configuration element. The value can be specified as "abc" when it refers to an element in the same package, "def:abc" when it refers to an element in "def" package and the package is imported, or in ":abc" form when dynamic reference resolution is needed.

Features

- **Concrete Web Element**
  no

- **Container Element**
  no

- **Java Renderers**
  Not available

- **JSP Renderers**
  Not available
LocationChooser

Description

The LocationChooser element specifies an action list element to be used as a selector for an object type when configuring search functionality. See "Configuring Search" chapter for more details.

Syntax

```xml
<!ELEMENT LocationChooser EMPTY>
<!ATTLIST Location
  id CDATA #IMPLIED
  ref CDATA #REQUIRED
  %conditions; #IMPLIED
  %dimensions; #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  The ID of the element.
- **element** = text[CS]
  This attribute value is the ID of the action list configuration element.

Features

- **Concrete Web Element**
  no
- **Container Element**
  no
- **Java Renderers**
  Not available
- **JSP Renderers**
  Not available
ModelDoer

Description

The ModeDoer defines the operation to run to populate a model of an element. For the list of legal ModelDoer classes (and when to use them), see section 6.

Syntax

```xml
<!ELEMENT  ModelDoer
          (Need*, Task) >
<!ATTLIST ModelDoer
    id CDATA                      #IMPLIED.
    class CDATA                   #REQUIRED
    %dimensions;                  #IMPLIED
    %conditions;                  #IMPLIED
>
```

Attribute definitions

- **id** = text
  - The ID of the element.
- **class** = text
  - The fully qualified name of the class that implements a model doer. The doer must implement com.ptc.core.ca.co.client.doer.ModelDoer interface.

Features

- **Concrete Web Element**
  - no
- **Container Element**
  - no
- **Java Renderers**
  - Not available
- **JSP Renderers**
  - Not available
**ModelHandler**

**Description**

The ModelHandler element defines the Java class used to create and update the model for an element.

**Syntax**

```
<!ELEMENT ModelHandler EMPTY>
<!ATTLIST ModelHandler
  id CDATA               #IMPLIED
  class CDATA            #REQUIRED
  %dimensions;           #IMPLIED
  %conditions;           #IMPLIED
>
```

**Attribute definitions**

- **id =** [text][CS]
  - The ID of the element.
- **class =** [text][CS]
  - The fully qualified name of the class that implements a model handler. The handler must implement com.ptc.core.ca.co.client.element.CoModelHandler interface.

**Features**

- **Concrete Web Element**
  - no
- **Container Element**
  - no
- **Java Renderers**
  - Not available
- **JSP Renderers**
  - Not available
Module

Description

The Module element defines a condition that an module must satisfy to belong to a set of applications and modules.

Syntax

```xml
<!ELEMENT Module EMPTY>
<!ATTLIST Module
  id CDATA                    #IMPLIED
  %dimensions;               #IMPLIED
  %conditions;               #IMPLIED
  >
```

Attribute definitions

- **id** = text
  
  The ID of the element.

Examples

```xml
<Context>
  <InstallationSet id="set1">
    <Module moduleId="Module A" from="6.0.2" to="6.2.8"/>
    <Module moduleId="Applicaton B" from="4.2"/>
    <Application appId="Applicaton C" to="7.0"/>
  </InstallationSet>
</Context>
```

Features

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available

- **JSP Renderers**
  
  Not available
**NavBarAction**

**Description**

The NavBarAction element defines a go-to action element to be used in navigation bar as a child of NavBarRow element.

**Syntax**

```xml
<!ELEMENT NavBarAction (Label?, ToolTip?, Location?, NavBarRow?, On*,
  (DrawHandler | (ActionHandler, (ActionDoer | ClientDoer),
  (RenderHandler | Renderer)?)
  )
 )

<!ATTLIST Action
  id CDATA                   #IMPLIED
  colspan CDATA          #IMPLIED
  rowspan CDATA          #IMPLIED
  %dimensions;          #IMPLIED
  %conditions;          #IMPLIED
  %span;                  #IMPLIED

 NavBarAction usually uses the following action handler:
 com.ptc.core.ca.web.client.menu.WebNavBarActionHandler
```

**Attribute definitions**

- **id** = text[CS]
  The ID of the element.

- **colspan** = number[CN]
  The number of columns spanned by the element in a layout. The default value for this attribute is one ("1").

- **rowspan** = number[CN]
  The number of rows spanned by the element in a layout. The default value for this attribute is one ("1").

**Features**

- **Concrete Web Element**
  com.ptc.core.ca.web.client.action.WebAction

- **Container Element**
  no

- **Java Renderers**
  com.ptc.core.ca.web.client.action.WebActionRenderHandler
com.ptc.core.ca.web.client.action.WebActionButtonRenderHandler

**JSP Renderers**

/WTcore/jsp/com/ptc/core/ca/web/action/action.jsp
/WTcore/jsp/com/ptc/core/ca/web/action/button.jsp
/WTcore/jsp/com/ptc/core/ca/web/action/help.jsp
**NavBarRow**

**Description**

The NavBarRow element is used to group sub-elements in a tree of NavBarActions belonging to a NavBar element. NavBarRow element extends LayoutRow and can contain any elements that can be children of LayoutRow. NavBarRow element determines its "current" action.

**Syntax**

```xml
<!ELEMENT  NavBarRow
 (NavBarAction+,
  Action*, ActionList*, HyperLink*, LabelArea*,
  List*, Property*, Separator*,
  Connector*, On*,
  SimpleLayout*, CompositeLayout*,
  SimpleNavBar*, CompositeNavBar*,
  SimplePropertyPanel*, CompositePropertyPanel*,
  SimpleTable*, CompositeTable*,
  CompositeTree*,
  (DrawHandler |
   (ModelHandler?,
   (RenderHandler | Renderer)?
  )
 )
 )>
<!ATTLIST NavBarRow
 id CDATA                     #IMPLIED
 %dimensions; #IMPLIED
 %conditions; #IMPLIED
">
```

**Attribute definitions**

`id` = text

The ID of the element.

**Features**

**Concrete Web Element**

com.ptc.core.ca.web.client.menu.WebNavBarRow

**Container Element**

yes

**Java Renderers**

com.ptc.core.ca.web.client.menu.WebLevel1NavBarRowRenderHandler
com.ptc.core.ca.web.client.menu.WebLevel2NavBarRowRenderHandler
com.ptc.core.ca.web.client.menu.WebLevel1FooterRowRenderHandler
com.ptc.core.ca.web.client.menu.WebLevel2FooterRowRenderHandler
com.ptc.core.ca.web.client.menu.WebVerticalNavBarRowRenderHandler
com.ptc.core.ca.web.client.menu.WebEmptyNavBarRowRenderHandler

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/menu/level1_navbarrow.jsp
/wtcore/jsp/com/ptc/core/ca/web/menu/level2_navbarrow.jsp
/wtcore/jsp/com/ptc/core/ca/web/menu/level1_footerrow.jsp
/wtcore/jsp/com/ptc/core/ca/web/menu/level2_footerrow.jsp
/wtcore/jsp/com/ptc/core/ca/web/menu/vertical_navbarrow.jsp
/wtcore/jsp/com/ptc/core/ca/web/menu/empty_navbarrow.jsp
Need

Description

The Need element allows the specification of the necessary attribute for another element, usually a Property or a Doer. A single element can have more than one Need sub-element.

Syntax

```xml
<!ELEMENT Need EMPTY>
<!ATTLIST Need
  id CDATA #IMPLIED
  attribute CDATA #REQUIRED
  accessibleByName CDATA #IMPLIED
  as (type | ufid) #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

- **id** = `text[CS]`
  The ID of the element.
- **attribute** = `text[CS]`
  The name of Info*Engine attribute.
- **accessibleByName** = `text[CS]`
  The name of Info*Engine attribute to access the requested attribute under. Use it only when a task returns a requested attribute under a different name.
- **as** = (type | ufid)
  To indicate that a particular attribute contains a type or UFID of an object.

Features

- **Concrete Web Element**
  no
- **Container Element**
  no
- **Java Renderers**
  Not available
- **JSP Renderers**
  Not available
NeedNpa

Description

The NeedNpa element allows the specification of the necessary non-persistent attribute for another element, usually a Property, an Action, or a Hyperlink. A single element can have more than one Need sub-element.

Syntax

```xml
<!ELEMENT  NeedNpa EMPTY
        >
<!ATTLIST NeedNpa
      id CDATA                     #IMPLIED
      attribute CDATA           #REQUIRED
      as (type | ufid)           #IMPLIED
      %dimensions;              #IMPLIED
      %conditions;              #IMPLIED
        >
```

Attribute definitions

- **id** = text
  - The ID of the element.
- **attribute** = text
  - The name of Info*Engine non-persistent attribute.
- **as** = (type | ufid)
  - To indicate that a particular attribute contains a type or UFID of an object.

Features

- **Concrete Web Element**
  - no
- **Container Element**
  - no
- **Java Renderers**
  - Not available
- **JSP Renderers**
  - Not available
NewContext

Description

The NewContext element is permitted as a child of a Location element to create a new context for the target frame referenced by the Location element. It can also be used to define the context for an element connected by Connector element.

Syntax

```xml
<!ELEMENT  NewContext (Copy*, Remove*)>
<!ATTLIST NewContext
  id CDATA                     #IMPLIED
  initWithCurrentContext (true | false)  #IMPLIED
  %dimensions;                 #IMPLIED
  %conditions;                 #IMPLIED
>

Caution: Frame parameters must not be used to construct context for temporary frames.

Attribute definitions

id =  text[CS]
   ID of the element.

initWithCurrentContext =  (true | false)
   Specifies whether to initialize new context with context of this element. The default value is true.

Examples

```xml
<Action>
  ...
  <Location ref=":view">
    <NewContext initWithCurrentContext="true">
      <Copy from="%[synonym#WTPart]" to="%[context#type]"/>
      <Copy from="%[context#color]"  to="%[context#bgcolor]"/>
      <Copy from="white"             to="%[context#fgcolor]"/>
      <Remove                        to="%[context#decoration]"/>
    </NewContext>
    ...
  </Location>
  ...
</Action>

The following is illegal

```xml
<Action>
  ...
  <Location ref=":view">
    <NewContext initWithCurrentContext="true">
      <Copy from="%[frame#abc]" to="%[context#abc]"/>
  </Location>
  ...
</Action>
```
Features

Concrete Web Element
no

Container Element
no

Java Renderers
Not available

JSP Renderers
Not available
NoItemsPrompt

Description

The NoItemsPrompt element defines the prompt text to be used by a composite table or a tree when it does not contain any items.

Syntax

```xml
<!ELEMENT  NoItemsPrompt (Need?)>
<!ATTLIST NoItemsPrompt
    id CDATA #IMPLIED
    resource CDATA #REQUIRED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
>
```

Attribute definitions

- **id** = `text[CS]`
  The ID of the element.

- **resource** = `text[CS]`
  String to display. If the value is of the form "aaa {0} bbb", this assumes that a Need sub element is specified. The value for an attribute specified in the need element will be substituted for the "{0}" portion of the resource. When more than one attribute value is needed for substitution use "aaa {0} bbb {1} ccc" and specify the appropriate number of Need statements.

Features

- **Concrete Web Element**
  no

- **Container Element**
  no

- **Java Renderers**
  Not available

- **JSP Renderers**
  Not available
On

Description

The On element allows the specification of a set of event element handlers.

Syntax

```xml
<!ELEMENT On (Copy+)
>  
<!ATTLIST On
    id CDATA               #REQUIRED
    event (draw|initModel|updateModel|executeModelDoer|executeActionDoer|updateChildrenSet|render)  #REQUIRED
    at (beginning | end)          #REQUIRED
    when CDATA                    #IMPLIED
    %dimensions;                    #IMPLIED
    %conditions;                   #IMPLIED
>  
```

Attribute definitions

- **id = text[CS]**
  ID of the element.
- **event = text[CS]**
  An event a handler is provided for. The following events are supported

<table>
<thead>
<tr>
<th>Event</th>
<th>Description of the stage the element goes through</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw</td>
<td>drawing of the element</td>
</tr>
<tr>
<td>initModel</td>
<td>initialization of the model of the element</td>
</tr>
<tr>
<td>updateModel</td>
<td>update of the model of the element</td>
</tr>
<tr>
<td>executeModelDoer</td>
<td>execution of the model doer</td>
</tr>
<tr>
<td>executeActionDoer</td>
<td>execution of the action doer</td>
</tr>
<tr>
<td>updateChildrenSet</td>
<td>update of the set of children</td>
</tr>
<tr>
<td>render</td>
<td>render the element</td>
</tr>
</tbody>
</table>
at = (beginning | end)
Time relative to a stage the element goes through when the handler is invoked. It is possible to handle event at different times related to the stage the element goes through.

<table>
<thead>
<tr>
<th>Handling time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginning</td>
<td>at the beginning of the</td>
</tr>
<tr>
<td>end</td>
<td>initialization of the model of the element</td>
</tr>
</tbody>
</table>

when = text[CS]
Condition that needs to be met for a handler to be involved. The syntax for a condition is exactly the same as the syntax for if element other than values can be used from any sources. For instance,

```xml
<On event="initModel" at="end"
      when="%[attribute#.name=="MyPart"]"/>
```

Features

**Concrete Web Element**
no

**Container Element**
no

**Java Renderers**
Not available

**JSP Renderers**
Not available
Option

Description

The Option element allows the specification of name value pairs. A renderer or any of the element handler can get access to options via com.ptc.core.ca.co.client.element.CoOptionUtils helper class.

Syntax

```
<!ELEMENT  Option EMPTY >
<!ATTLIST Option
    id CDATA               #REQUIRED
    theme CDATA           #REQUIRED
    option CDATA           #REQUIRED
    value CDATA            #REQUIRED
    %dimensions;         #IMPLIED
    %conditions;         #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  ID of the element.
- **theme** = text[CS]
  Theme of the option.
- **option** = text[CS]
  Name of the option.
- **value** = text[CS]
  Value of the option.

Features

- **Concrete Web Element**
  no
- **Container Element**
  no
- **Java Renderers**
  Not available
- **JSP Renderers**
  Not available
OuterTask

Description

The OuterTask element specifies the task to be executed by the doer. The task will invoke the execution of the regular task specified by Task element. This outer task gets all the parameters of the regular task. It will get also an additional parameter with name dca_target_task. Its value is the name of the task specified in Task element by taskDelegate attribute.

Syntax

```xml
<!ELEMENT OuterTask EMPTY>
<!ATTLIST OuterTask
  id CDATA #IMPLIED
  taskDelegate CDATA #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED>
```

Attribute definitions

- **id** = text[CS]
  
  ID of the element.

- **taskDelegate** = text[CS]
  
  Task delegate for the outer task.

Examples

```xml
<ActionDoer class="%common:SimpleActionTaskDoer">
  <Task taskDelegate="regular-task"
    nextOperation="STDOP|com.ptc.windchill.view">
    <OuterTask taskDelegate="dca-transaction"/>
    <TaskParameter name="type" value="WCTYPE|wt.part.WTPart"/>
  </Task>
</ActionDoer>
```

In this example the task dca-transaction provides transactional behavior for the included task.

Features

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available
JSP Renderers
Not available
Plugin

Description

The Plugin element defines the configuration of an applet or an ActiveX control.

Syntax

```xml
<!ELEMENT  Plugin
    (Option*, On*,
    (DrawHandler | ((ModelHandler | ModelDoer),
    Renderer?)
    )
    )>

<!ATTLIST CompositeTable
    id CDATA                      #IMPLIED
    %dimensions;         #IMPLIED
    %conditions;        #IMPLIED
    %span;             #IMPLIED
    >
```

Attribute definitions

```xml
id =  text[CS]
```

ID of the element.

Features

**Concrete Web Element**

com.ptc.core.ca.web.client.plugin.WebPlugin

**Container Element**

no

**Java Renderers**

com.ptc.core.ca.web.client.plugin.WebWindchillPluginRenderHandler

**JSP Renderers**

/WTcore/jsp/com/ptc/core/ca/web/plugin/windchillPlugin.jsp
Populate

Description
The Populate element is intended to provide a mechanism to set up a mapping between attributes from a picked and targeted instances. See the section on Pickers for more details.

Syntax

```xml
<!ELEMENT  Populate EMPTY
    >
<!ATTLIST Need
    id CDATA               #IMPLIED
    attribute CDATA        #IMPLIED
    from CDATA             #REQUIRED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
    >
```

Attribute definitions

- `id` = text[CS]
  The ID of the element.

- `attribute` = text[CS]
  The name of the attribute in the target instance to populate with a value.

  When no "attribute" attribute is specified, Need attributes of the target property (specified by UpdateLocation element) are evaluated. For every Populate element a Need element with the same index is found. Then its attribute is used as a target attribute to populate its value from a "from" attribute.

- `from` = text[CS]
  The name of the attribute in the picked instance to take a value of.

Features

- **Concrete Web Element**
  - no

- **Container Element**
  - no

- **Java Renderers**
  - Not available

- **JSP Renderers**
  - Not available
Prompt

Description
The Prompt element defines the prompt text to display. It can be used differently depending on the parent element using the prompt.

Syntax
```xml
<!ELEMENT Prompt (Need?)>
<!ATTLIST Prompt
   id CDATA #IMPLIED
   resource CDATA #REQUIRED
   %dimensions; #IMPLIED
   %conditions; #IMPLIED>
```

Attribute definitions
- id = text[CS]
  The ID of the element.
- resource = text[CS]
  String to display. If the value is of the form "aaa {0} bbb", this assumes that a Need sub element is specified. The value for an attribute specified in the need element will be substituted for the "{0}" portion of the resource. When more than one attribute value is needed for substitution use "aaa {0} bbb {1} ccc" and specify the appropriate number of Need statements.

Features
- Concrete Web Element
  no
- Container Element
  no
- Java Renderers
  Not available
- JSP Renderers
  Not available
Property

Description

The Property element presents an attribute or attributes of an Info*Engine element. The standard data types have predefined templates elements that define the default PropertyHandler and Renderer for each data type. The Need sub-elements are required to get correct mapping to the Info*Engine attributes. For the Property element to work correctly, it should be specified as a descendent of one of the following elements in the configuration specification:

- SimplePropertyPanel, CompositePropertyPanel
- CompositeColumn, SimpleColumn
- Wizard

The Property is able to obtain its attribute values from the Info*Engine element associated with ancestor PropertyPanel, table row, or tree node element.

Syntax

```xml
<!ELEMENT  Property
    (Need, Label?, On*,
     (DrawHandler |
     (ModelHandler | PropertyHandler)?,
     Renderer
     )
    )>
<!ATTLIST Property
  id CDATA                     #REQUIRED
  required (true | false) "false"
  sticky (true | false) "true"
  frameTyp (new  | same) "same"
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  
  ID of the element.

- **required** = text[CN]
  
  Used for simple property to indicate that it requires a value in create, update, and search mode.

- **sticky** = text[CN]
  
  Used for simple property to indicate that it can specify previously entered value as a default value in create or search mode.
frameType = text[CS]
Available only for the property presenting a hurl to a content item. Provides the configuration of the target frame for a content item to be loaded in. The value new opens the content in the new window, while any other values opens it in the same window. Default is same.

Features

Concrete Web Element
com.ptc.core.ca.web.client.property.WebProperty

Container Element
no

Java Renderers
depend on the data type

JSP Renderers
depend on the data type
PropertyHandler

Description

The PropertyHandler element configures the Property element by specifying the Java class that handles the specific data type. The Property handler class works together with the property Renderer. The PropertyHandler class provides the data to the property Renderer. It is also responsible for extracting the data from the POST request and updating the proper Info*Engine element.

Syntax

```xml
<!ELEMENT  PropertyHandler EMPTY
>
<!ATTLIST PropertyHandler
   id CDATA                 #IMPLIED
   class CDATA            #REQUIRED
   %dimensions; #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  - The ID of the element.

- **class** = text[CS]
  - The Java class name of property handler class extending com.ptc.core.ca.co.client.property.CoPropertyHandler.

Features

- **Concrete Web Element**
  - no

- **Container Element**
  - no

- **Java Renderers**
  - Not available

- **JSP Renderers**
  - Not available
Remove

Description

The Remove element allows the removal of a dimension from context.

Syntax

```xml
<!ELEMENTRemove EMPTY>
<!ATTLIST Remove
  id CDATA                #IMPLIED
  to CDATA               #REQUIRED
  %dimensions;           #IMPLIED
  %conditions;           #IMPLIED
>
```

Attribute definitions

- **id** = text [CS]
  
  ID of the element.

- **to** = text [CS]
  
  The dimension to remove from context.

Examples

```xml
<Action>
  ...
  <Location ref=":view">
    ...
    <NewContext initWithCurrentContext="true">
      <Copy from="%[synonym#WTPart]" to="%[context#type]"/>
      <Copy from="%[context#color]" to="%[context#bgcolor]"/>
      <Copy from="white" to="%[context#fgcolor]"/>
      <Remove to="%[context#decoration]"/>
    </NewContext>
    ...
  </Location>
</Action>
```

Features

**Concrete Web Element**

- no

**Container Element**

- no

**JSP Renderers**

- Not available

**JSP Renderers**

- Not available
Renderer

Description

The Renderer element defines the resource the parent element uses to generate html or any other type of structured information.

Syntax

```
<!ELEMENT  Renderer EMPTY>
<!ATTLIST Renderer
   id CDATA             #IMPLIED
   resource CDATA       #IMPLIED
   class CDATA       #IMPLIED
   %dimensions; #IMPLIED
   %conditions; #IMPLIED>
```

Attribute definitions

id =  text

ID of the element.

resource = text

This attribute defines the resource to render the element. It can be any valid web application resource. Usually it will be either a jsp page or a servlet.

class = text

This attribute defines the class to render the element. In DCA it must implement com.ptc.core.ca.co.client.element.CoRenderHandler interface. In DCA Web it must extend com.ptc.core.ca.web.client.element.WebRenderHandler class.

Features

Concrete Web Element

no

Container Element

no

Java Renderers

Not available

JSP Renderers

Not available
Repository

Description

The Repository element must be the root element for a configuration file to be valid.

Syntax

```xml
<!ELEMENT Repository
   (Import*, Context*) >
<!ATTLIST Repository
   package CDATA #IMPLIED
   rbInfo CDATA #IMPLIED
>
```

Attribute definitions

- **package** = text[CS]
  
  Defines the namespace (package) for all the elements defined in this file. See the Reference Resolution section for the rules governing the resolution of references.

- **rbInfo** = text[CS]
  
  Defines a file name for the resource bundles. The resource bundle file will be created with .rbInfo extension.

Examples

- common.xml
  
  ```xml
  <Repository package="com.ptc.common" rbInfo="com.ptc.common">
    <Property id="property.name.template">
      ...
    </Property>
  </Repository>
  ```

- folder.xml
  
  ```xml
  <Repository package="com.ptc.folder">
    <Import package = "com.ptc.common"/>
    ...
    <Property id="property.name">
      <Extend ref="common:property.name.template"/>
    </Property>
  </Repository>
  ```

Features

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no
Java Renderers
Not available

JSP Renderers
Not available
RoleDeterminator

Description

The RoleDeterminator element specifies the class to be used to determine the value of roleId dimension. The class must implement com.ptc.core.ca.co.client.misc.CoRoleDeterminator interface.

Syntax

<!ELEMENT  RoleDeterminator EMPTY >
<!ATTLIST  RoleDeterminator   id CDATA       #REQUIRED
   class CDATA #REQUIRED
   if.appId CDATA; #IMPLIED
   if.typeId CDATA; #IMPLIED
   ifnot.appId CDATA; #IMPLIED
   ifnot.typeId CDATA; #IMPLIED>

Attribute definitions

id =  text[CS]
ID of the element. ID of the RoleDeterminator configuration element must always be RoleDeterminator or it will be ignored.

class =  text[CS]
The fully qualified name of the class that implements a role determinator. The role determinator must implement com.ptc.core.ca.co.client.misc.CoRoleDeterminator interface.

id =  text[CS]
ID of the element.

Examples

<Context>
<RoleDeterminator id="RoleDeterminator"
class="com.ptc.refimpl.co.client.misc.CoTestRoleDeterminator"/>
</Context>

Caution: ID of the RoleDeterminator configuration element must always be RoleDeterminator or it will be ignored.

Features

Concrete Web Element
no
Container Element
   no
Java Renderers
   Not available
JSP Renderers
   Not available
The Script element renders HTML script tag. It is only used to provide links to the necessary JavaScript files.

**Syntax**

```xml
<!ELEMENT Script
   (ModelHandler, Renderer, RenderHandler?, DrawHandler?) >
<!ATTLIST Script
   id CDATA                     #IMPLIED
   resource CDATA               #IMPLIED
   %dimensions;                 #IMPLIED
   %conditions;                 #IMPLIED
   >
```

**Attribute definitions**

- **id** = `text[CS]`
  The ID of the element.

- **resource** = `text[CS]`
  The Web application resource pointing to javascript file.

**Features**

- **Concrete Web Element**
  com.ptc.core.ca.web.client.primitive.WebLabel

- **Container Element**
  no

- **Java Renderers**
  com.ptc.core.ca.web.client.frame.WebScriptRenderHandler

- **JSP Renderers**
  /wtcore/jsp/com/ptc/core/ca/web/frame/script.jsp
**SelectAllPrompt**

**Description**

The SelectAllPrompt element defines the prompt text for the "Select All" embedded action of a selector. It can be used differently depending on the parent element using the prompt.

**Syntax**

```xml
<!ELEMENT  SelectAllPrompt
    (Need?)>
<!ATTLIST SelectAllPrompt
  id CDATA #IMPLIED
  resource CDATA #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

**Attribute definitions**

- **id** = text[CS]
  
  The ID of the element.

- **resource** = text/[CS]
  
  String to display. If the value is of the form "aaa {0} bbb", this assumes that a Need sub element is specified. The value for an attribute specified in the need element will be substituted for the "{0}" portion of the resource. When more than one attribute value is needed for substitution use "aaa {0} bbb {1} ccc" and specify the appropriate number of Need statements.

**Features**

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available

- **JSP Renderers**
  
  Not available
Separator

Description

The Separator is used to place a vertical or horizontal separator. It is usually placed as a child element of a LayoutRow with the appropriate colspan or rowspan attributes.

Syntax

```xml
<!ELEMENT Separator
  (On*, DrawHandler | 
   (RenderHandler | Renderer)?)>
<!ATTLIST Separator
  id CDATA                   #IMPLIED
  separatorType (vertical | horizontal) #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>```

Attribute definitions

- **id** = text[CS]
  The ID of the element.

- **separatorType** = (vertical | horizontal)
  The Direction to render the separator. This is implied for an action list and required otherwise.

Features

- **Concrete Web Element**
  com.ptc.core.ca.web.client.primitive.WebSeparator

- **Container Element**
  no

- **Java Renderers**
  com.ptc.core.ca.web.client.primitive.WebHorizontalSeparatorRenderHandler
  com.ptc.core.ca.web.client.primitive.WebVerticalSeparatorRenderHandler

- **JSP Renderers**
  /wtcore/jsp/com/ptc/core/ca/web/primitive/horizontalSeparator.jsp
  /wtcore/jsp/com/ptc/core/ca/web/primitive/verticalSeparator.jsp
**Shadow**

**Description**

The Shadow element references another DCA element and draws a shadow of it. The Shadow element itself does not know what the shadow looks like. Instead, the referenced element (will call it the original element) is responsible for providing all the data the shadow needs to display itself at the time of rendering the original element.

**Syntax**

```xml
<!ELEMENT  Shadow
  (On*, DrawHandler |
   (RenderHandler | Renderer)?
 )>
<!ATTLIST Shadow
  id CDATA                   #IMPLIED
  ref CDATA                   #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

**Attribute definitions**

- **id** =  `text[CS]`
  The ID of the element.

- **ref** =  `text[CS]`
  The reference to the element the shadow is created for.

**Features**

- **Concrete Web Element**
  com.ptc.core.ca.web.client.primitive.WebShadow

- **Container Element**
  no

- **Java Renderers**
  com.ptc.core.ca.web.client.primitive.WebShadowRenderHandler

- **JSP Renderers**
  /wtcore/jsp/com/ptc/core/ca/web/primitve/shadow.jsp
**SimpleColumn**

**Description**

The SimpleColumn elements specifies the column to place in a table. The column header is rendered using the faceplate renderer based on the label attribute. The cell element is created based on the configuration specification referred to by the value of the "show" attribute. The "For" element allows the rendering of the cell to be substituted depending on the context.

**Syntax**

```xml
<!ELEMENT SimpleColumn (Label?, For*, On*, (DrawHandler | (ModelHandler?, (RenderHandler | Renderer)? )))>
<!ATTLIST SimpleColumn
id CDATA #IMPLIED
show CDATA #IMPLIED %dimensions; #IMPLIED
%conditions; #IMPLIED>
```

**Attribute definitions**

- **id** = text[CS]
  - The ID of the element.

- **show** = text[CS]
  - The ID of the element to create in the cell for this column in each row of the table. Generally this is the name of a Property (to display the attribute value of a business object for the row), an ActionList (to provide actions to run against the row) or an Action (to go to the property page of the object). However, there are no restrictions on the type of element that can be created in the cell.

**Features**

- **Concrete Web Element**
  - com.ptc.core.ca.web.client.table.WebSimpleColumn

- **Container Element**
  - yes

- **Java Renderers**
  - com.ptc.core.ca.web.client.table.WebSimpleColumnRenderHandler

- **JSP Renderers**
  - /wtcore/jsp/com/ptc/core/ca/web/table/simplecolumn.jsp
SimpleFrame

Description

The SimpleFrame is one of the legal root window elements (CompositeFrame is the other). While Link and Script element are legal, these are usually obtained by the SimpleFrame definition extending the SimpleFrame template. All aliases specified to the gw.jsp must identify a frame element.

Syntax

```xml
<!ELEMENT  SimpleFrame
  (Link?, Script?, On*,
   (Connector |
    SimpleLayout | CompositeLayout |
    SimplePropertyPanel | CompositePropertyPanel |
    SimpleTable | CompositeTable |
    CompositeTree |
    Wizard),
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
   )
  )>
<!ATTLIST SimpleFrame
  id CDATA                #REQUIRED
  width CDATA          #IMPLIED
  height CDATA         #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

- **id** =  text[CS]
  
  ID of the element.

- **width** = length[CN]
  
  Width of the frame in pixels.

- **height** = length[CN]
  
  Height of the frame in pixels

**Caution:** roleId attribute must not be specified for a frame element.

Features

**Concrete Web Element**

com.ptc.core.ca.web.client.frame.WebSimpleFrame

**Container Element**

yes
**Java Renderers**
com.ptc.core.ca.web.client.frame.WebSimpleFrameRenderHandler

**JSP Renderers**
/wtcore/jsp/com/ptc/core/ca/web/frame/simpleFrame.jsp
SimpleLayout

Description

The SimpleLayout element allows placement of various elements using a layout mechanism conceptually similar to <Table> HTML tag. The SimpleLayout element will place an element in the correct location on the screen based on the information configured in each LayoutRow element.

Syntax

```xml
<!ELEMENT  SimpleLayout
  (LayoutRow+, On*,
   (DrawHandler | 
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
 )>

<!ATTLIST SimpleLayout
  id CDATA               #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
 >
```

Attribute definitions

```plaintext
id =  text[CS]
```

The ID of the element.

Features

**Concrete Web Element**

com.ptc.core.ca.web.client.layout.WebSimpleLayout

**Container Element**

yes

**Java Renderers**

com.ptc.core.ca.web.client.layout.WebSimpleLayoutRenderHandler

**JSP Renderers**

/wtcore/jsp/com/ptc/core/ca/web/layout/simpleLayout.jsp
**SimpleNavBar**

**Description**

The SimpleNavBar element represents a menu or a navigation bar. Conceptually, it contains a tree of NavBarAction elements grouped together using NavBarRow elements. At run time it determines "current" action by comparing its location to the location of its frame. Once current action is determined, a list of NavBarRow elements containing ancestor actions is constructed and drawn. This gives a visual appearance of a context sensitive navigation bar.

In terms of rendering the rows defined in a SimpleNavBar, the default behavior is the same as the SimpleLayout. The difference between using a SimpleNavBar element and a SimpleLayout element is that the SimpleNavBar element expects to have actions and is controlled by a set of business objects that are used to determine availability of the actions in the navigation bar.

**Syntax**

```xml
<!ELEMENT  SimpleNavBar
  (NavBarRow+, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
 )>
<!ATTLIST SimpleNavBar
  id CDATA     #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span;       #IMPLIED

Attribute definitions

id = text[CS]
   The ID of the element.

Features

**Concrete Web Element**
   com.ptc.core.ca.web.client.menu.WebSimpleNavBar

**Container Element**
   yes

**Java Renderers**
   com.ptc.core.ca.web.client.layout.WebSimpleNavBarRenderHandler.java

**JSP Renderers**
   /wtcore/jsp/com/ptc/core/ca/web/menu/simpleNavBar.jsp```
**SimplePropertyPanel**

**Description**

The SimplePropertyPanel is a so-called instance holder element as it has associated Info*Engine element. It also extends a SimpleLayout element and this allows placement of various elements using a general layout mechanism.

Generally, SimplePropertyPanel element is used to create a property sheet for a business object. By using a SimplePropertyPanel, the Property elements children of the LayoutRow elements can access the attribute business data from the SimplePropertyPanel element.

**Syntax**

```xml
<!ELEMENT SimplePropertyPanel
(LayoutRow*, On*,
(DrawHandler |
  ((ModelHandler | ModelDoer)?,
  (RenderHandler | Renderer)?
  )
  )
  )>

<!ATTLIST SimplePropertyPanel
  id CDATA                     #IMPLIED
  %dimensions;               #IMPLIED
  %conditions;              #IMPLIED
  %span;                    #IMPLIED
>
```

**Attribute definitions**

`id` = text

The ID of the element.

**Features**

**Concrete Web Element**

com.ptc.core.ca.web.client.propertyPanel.WebSimplePropertyPanel

**Container Element**

yes

**Java Renderers**

com.ptc.core.ca.web.client.propertyPanel.WebSimplePropertyPanelRenderHandler

**JSP Renderers**

/wtcore/jsp/com/ptc/core/ca/web/propertyPanel/simplePropertyPanel.jsp
**SimpleTable**

**Description**

The SimpleTable element defines the configuration of a table. It allows specification of the type of objects to support, columns to display in the table, and a ModelDoer to initialize the table when it is asked to draw itself. The SimpleColumn elements determine how the column header and cell of a column of each row in the table is rendered. If Label, ToolTip or Image elements are specified, they are used in rendering a faceplate for the Table.

**Syntax**

```xml
<!ELEMENT SimpleTable
  (Type*, SimpleColumn*, ActionList?,
   Label?, ToolTip?, Image?, Footer?,
   SelectAllPrompt, On*,
   (DrawHandler |
    ((ModelHandler | ModelDoer),
     (RenderHandler | Renderer)?
    )
   )
 )>

<!ATTLIST SimpleTable
  id CDATA                     #IMPLIED
  selectorType  (single | multi | none)  "none"
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

**Attribute definitions**

- **id** = **text**
  - The ID of the element.

- **selectorType** = **(single | multi | none)**
  - This specifies the selection behavior of the table. In DCA Web when none are specified, no column will be rendered that supports selection.

**Features**

- **Concrete Web Element**
  - com.ptc.core.ca.web.client.table.WebSimpleTable

- **Container Element**
  - yes

- **Java Renderers**
  - com.ptc.core.ca.web.client.table.WebSimpleTableRenderHandler
JSP Renderers
wtcore/jsp/com/ptc/core/ca/web/table/simpletable.jsp
Synonym

Description

The Synonym element allows the specification of a symbolic substitute for another value. Synonyms are primarily used to create convenience references to other names to simplify configuration definition and binding to Info*Engine. Multiple synonyms with the same id can be defined in different contexts. The usual reference resolution rules will apply when a value of a synonym is needed.

Syntax

```xml
<!ELEMENT  Synonym EMPTY
>
<!ATTLIST Synonym
    id CDATA       #REQUIRED
    value CDATA    #REQUIRED
>
```

Attribute definitions

`id` = `text[CS]`

This attribute is the name of the synonym. The synonym is only visible to the configuration system. When the runtime version of the configuration object is accessed, the configuration system has already replaced the synonym with the actual value.

`value` = `text[CS]`

The value is the actual value of the attribute. When the configuration object is processed, the configuration system replaces references to the synonym ID with the value of this attribute. The "%[xxx]" or "%xxx" syntax must be for to the synonym ID for the substitution to occur. For example, if the synonym ID is "Folder" then a reference to the synonym is "%[Folder]" or "%Folder".

Examples

```xml
<Synonym id="Folder" value="wt.folder.Folder">
...
<TaskParameter name="type" value="%[Folder]"/>
<TaskParameter name="type" value="%Folder"/>
```

Features

Concrete Web Element

no

Container Element

no

Java Renderers

Not available
JSP Renderers
Not available
Task

Description

The Task element defines the configuration for an Info*Engine task.

Syntax

```xml
<!ELEMENT Task (TaskParameter*, TaskGroup*, Validator?) >
<!ATTLIST Task
  id CDATA                     #IMPLIED
  nextOperation CDATA #IMPLIED
  taskDelegate CDATA #REQUIRED
  taskType CDATA #IMPLIED
  taskRepository CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

- id = text[CS]
  The ID of the element.
- nextOperation = text[CS]
  The syntax of the nextOperation should be the External form of a valid OperationIdentifier. See LWC documentation for the explanation of this parameter. The typical values will be
  STDOP|com.ptc.windchill.create
  STDOP|com.ptc.windchill.view
  STDOP|com.ptc.windchill.update
  STDOP|com.ptc.windchill.search
- taskDelegate = text[CS]
  The name of the Info*Engine task delegate.
- taskType = text[CS]
  Specifies a data type for a task dispatching mechanism. DCA takes this attribute into account only when it generates a fudge element. When DCA generates a fudge element and the attribute is not specified, a default fudge element with type wt.fc.WTObject is generated.
- taskRepository = text[CS]
  Specifies a repository name for a task dispatching mechanism.

Features

Concrete Web Element

no
Container Element
  no

Java Renderers
  Not available

JSP Renderers
  Not available
TaskElement

Description

The TaskElement element defines the element for the TaskGroup.

Syntax

```xml
<!ELEMENT TaskElement (TaskParameter*) >
<!ATTLIST TaskElement
  id  CDATA                     #IMPLIED
  name CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Examples

```xml
<Task nextOperation="STDOP|com.ptc.windchill.view"
       taskDelegate="my_delegate">
  <TaskGroup name="my_group">
    <TaskElement name="my_element">
      <TaskParameter name="my_param" value="my_param_val"/>
    </TaskElement>
  </TaskGroup>
</Task>
```

Attribute definitions

- **id** = text[CS]
  - The ID of the element.

- **name** = text[CS]
  - The name of the element. Typically the element can be accessed in the task as follows
  ```java
  <%
  Group       group     = getGroup("my_group");
  Enumeration elem_enum = linkDescr.getElements();

  while (elem_enum.hasMoreElements()) {
    Element element = (Element)elem_enum.nextElement();

    if (element.getName().equals("my_element")) {
      String my_param_val=(String)element.getValue("my_param");
      ...
  ```

Features

- **Concrete Web Element**
  - no

- **Container Element**
  - no

- **Java Renderers**
  - Not available
JSP Renderers
Not available
TaskGroup

Description

The TaskGroup element defines the group as a parameter to an Info*Engine task.

Syntax

```xml
<!ELEMENT TaskGroup (TaskElement+) >
<!ATTLIST TaskGroup
  id CDATA #IMPLIED
  name CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Examples

```xml
<Task nextOperation="STDOP|com.ptc.windchill.view"
  taskDelegate="my_delegate">
  <TaskGroup name="my_group">
    <TaskElement name="my_element">
      <TaskParameter name="my_param" value="my_param_val"/>
    </TaskElement>
  </TaskGroup>
</Task>
```

Attribute definitions

- **id** = text[CS]
  - The ID of the element.

- **name** = text[CS]/
  - The name of the group. Typically the group can be accessed in the task as follows
    ```csharp
    Group       group     = getGroup("my_group");
    ...
    ```

Features

- **Concrete Web Element**
  - no

- **Container Element**
  - no

- **Java Renderers**
  - Not available

- **JSP Renderers**
  - Not available
TaskParameter

Description

The TaskParameter element is used to specify information for an Info*Engine task. Possible uses are for field values.

Syntax

<!ELEMENT TaskParameter (TaskParameter*) >
<!ATTLIST TaskParameter
  id CDATA #IMPLIED
  name CDATA #IMPLIED
  value CDATA #IMPLIED
  computedValue CDATA #IMPLIED --DEPRECATED--
  function CDATA #IMPLIED
  element CDATA #IMPLIED
  selected CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>

Attribute definitions

id = text[CS]
  The ID of the element.

name = text[CS]
  The name of the task parameter.

value = text[CS]
  The value of the static task parameter.

computedValue = text[CS] -- DEPRECATED --
  [Deprecated] A fully qualified name of the class to compute a value for the task parameter. If the value is specified, computedValue is ignored. The class must implement one of the following two interfaces
  com.ptc.core.ca.co.client.doer.CoModelTaskComputedValue
  com.ptc.core.ca.co.client.doer.CoActionTaskComputedValue

Can not be used when TaskParameter is a child of TaskElement.

function = text[CS]
  A fully qualified name of the class that implements one of the following interfaces
  com.ptc.core.ca.co.client.doer.function.CoFunction_Action_Param
  com.ptc.core.ca.co.client.doer.function.CoFunction_Model_Param

  or extends one of the following classes:
  com.ptc.core.ca.co.client.doer.function.CoSimpleFunction_Action_Param
  com.ptc.core.ca.co.client.doer.function.CoSimpleFunction_Model_Param
This provides a mechanism of computing dynamic task parameters which cannot be provided through static task parameters.

Can not be used when TaskParameter is a child of TaskElement.

\text{element} = \text{text[CS]}

The ID of the configuration element is a function that will locate, to prepare task parameter. The ID must be unique in the frame.

Can not be used when TaskParameter is a child of TaskElement.

Features

\textbf{Concrete Web Element}  
no

\textbf{Container Element}  
no

\textbf{Java Renderers}  
Not available

\textbf{JSP Renderers}  
Not available
**TaskOutParameter**

**Description**

The TaskOutParameter element defines the configuration of an output parameter for an Info*Engine task.

**Syntax**

```xml
<!ELEMENT TaskParameter (TaskParameter*) >
<!ATTLIST TaskParameter
  id CDATA #IMPLIED
  function CDATA #IMPLIED
%dimensions; #IMPLIED
%conditions; #IMPLIED
>
```

**Attribute definitions**

- **id** = text[CS]
  
  The ID of the element.

- **function** = text[CS]
  
  The fully qualified name of the class that implements one of the following interfaces
  
  com.ptc.core.ca.co.client.doer.function.CoFunction_Action_Result
  com.ptc.core.ca.co.client.doer.function.CoFunction_Model_Result

  or extends one of the following classes:

  com.ptc.core.ca.co.client.doer.function.CoSimpleFunction_Action_Result
  com.ptc.core.ca.co.client.doer.function.CoSimpleFunction_Model_Result

**Features**

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**

  Not available

- **JSP Renderers**

  Not available
**Template**

**Description**

The Template element defines the configuration of a specially configured HyperLink element to provide a link to a template processor-based resource.

**Syntax**

```xml
 <!ELEMENT  Template    (Option+) >
<!ATTLIST Template
  id CDATA             #IMPLIED
  class CDATA         #IMPLIED
  method CDATA        #IMPLIED
  anonymous (true | false) "false"
  %dimensions;        #IMPLIED
  %conditions;        #IMPLIED
 >
```

**Attribute definitions**

**id** = text[CS]

ID of the element.

**class** = text[CS]

The fully qualified name of the template processor class.

**method** = text[CS]

Template processor method.

**anonymous** = text[CS]

Weather to construct a link to an anonymous or authenticated resource. Default value is false.

**Caution:** class and method attributes must be provided when this element is used with "simple" model handler.

**Features**

**Concrete Web Element**

no

**Container Element**

no

**Java Renderers**

Not available

**JSP Renderers**

Not available
ToolTip

Description
The ToolTip element defines the tooltip to display. This can be used differently depending on the parent element using the tooltip. Tooltips can perform substitution based on the Need sub-elements.

Syntax

```
<!ELEMENT ToolTip (Need?)>
<!ATTLIST Image
  id CDATA #IMPLIED
  resource CDATA #REQUIRED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

- **id** = text[CS]
  - The ID of the element.

- **resource** = text[CS]
  - The string resource to display. If the value is of the form "\{0\}", this assumes that a Need sub element is specified. The value for attribute specified in the need element will be substituted for the "\{0\}" portion of the resource.

Features

- **Concrete Web Element**
  - no

- **Container Element**
  - no

- **Java Renderers**
  - Not available

- **JSP Renderers**
  - Not available
**TopLayout**

**Description**

The TopLayout element is a standard layout element used to configure layout component in a composite element (CompositeTable, CompositeTree, etc.). In a standard composite component, the TopLayout is displayed below the HeaderLayout and above the component. When a component is contracted, the TopLayout disappears but the HeaderLayout stays visible.

**Syntax**

```
<!ELEMENT  TopLayout
    (LayoutRow+, On*,
     (DrawHandler |
      (ModelHandler?,
       (RenderHandler | Renderer)?
      )
     )
    )>
<!ATTLIST TopLayout
    id CDATA               #IMPLIED
    %dimensions;         #IMPLIED
    %conditions;         #IMPLIED
  >
```

**Attribute definitions**

- `id` = `text[CS]`
  The ID of the element.

**Features**

- **Concrete Web Element**
  com.ptc.core.ca.web.client.layout.WebSimpleLayout

- **Container Element**
  yes

- **Java Renderers**
  com.ptc.core.ca.web.client.layout.WebSimpleLayoutRenderHandler

- **JSP Renderers**
  /wtcore/jsp/com/ptc/core/ca/web/layout/simpleLayout.jsp
Trigger

Description

The Trigger element configures an action that triggers an execution of an operation selected in the list of actions.

Syntax

```xml
<!ELEMENT Trigger
  (Label?, Image?, ToolTip?, On*,
   (DrawHandler |
    (ActionHandler, (ActionDoer | ClientDoer)?,
     (RenderHandler | Renderer)?
   )
  )
>
<!ATTLIST Trigger
  id CDATA                     #IMPLIED
  %dimensions;                 #IMPLIED
  %conditions;                 #IMPLIED
  %span;                       #IMPLIED
>
```

Attribute definitions

- `id = text[CS]`
  - The ID of the element.

**Caution:** In DCA Web out of the box `passivedropdownlist.jsp` action list renderer will not delegate drawing to a trigger but rather render it itself. It will only require the specification of a label but will take into account an image and a tooltip when provided.

Features

- **Concrete Web Element**
  - `com.ptc.core.ca.web.client.action.WebAction`

- **Container Element**
  - `no`

- **Java Renderers**
  - `com.ptc.core.ca.web.client.action.WebActionRenderHandler`
  - `com.ptc.core.ca.web.client.action.WebActionButtonRenderHandler`

- **JSP Renderers**
  - `/wtcore/jsp/com/ptc/core/ca/web/action/action.jsp`
  - `/wtcore/jsp/com/ptc/core/ca/web/action/button.jsp`
**Type**

**Description**

The Type element specifies the types of business objects to use for the list data containers - tree or table. It is used multiple times to specify all the objects types the container will handle. The values are used by the data access layer to obtain the correct type of business objects.

**Syntax**

```xml
<!ELEMENT  Type EMPTY

<!ATTLIST Type
    type CDATA                    #REQUIRED
    %dimensions;                 #IMPLIED
    %conditions;                 #IMPLIED
>
```

**Attribute definitions**

- **type = text[CS]**
  
The name of object type. The value is usually referenced to the synonym that defines the type Info*Engine understands.

**Features**

- **Concrete Web Element**
  
  no

- **Container Element**
  
  no

- **Java Renderers**
  
  Not available

- **JSP Renderers**
  
  Not available
UpdateLocation

Description
The UpdateLocation element defines the ID of a property or a table to populate by a picker. The ID will be used to update the correct element after selection is made and Ok button is hit.

Syntax
<!ELEMENT  UpdateLocation EMPTY
  >
<!ATTLIST Location
    ref CDATA    #REQUIRED
    %dimensions; #IMPLIED
    %conditions; #IMPLIED
  >

Attribute definitions
ref = text[CS]
Contains the ID of the Property or Table configuration element.

Features
Concrete Web Element
no
Container Element
no
Java Renderers
Not available
JSP Renderers
Not available
Validator

Description

The Validator element configures specifies a validator class for an action or a task. When specified for an action, it answers the question whether the action is relevant for a current situation, and if relevant, whether it can be triggered. When specified for a task, it determines whether the task should be executed.

Syntax

```
<!ELEMENT Validator EMPTY

<!ATTLIST Validator
  id CDATA                   #IMPLIED
  class CDATA            #REQUIRED
  %dimensions;            #IMPLIED
  %conditions;            #IMPLIED
>
```

Attribute definitions

- **id** = `text` [CS]
  - The ID of the element.
- **class** = `text` [CS]
  - The fully qualified name of the Java class that implements a validator. When specified for an action, the class must implement `com.ptc.core.ca.co.client.doer.CoActionValidator` interface. When specified for the task, the class must implement `com.ptc.core.ca.co.client.doer.CoTaskValidator` interface.

Features

- **Concrete Web Element**
  - no
- **Container Element**
  - no
- **Java Renderers**
  - Not available
- **JSP Renderers**
  - Not available
Wizard

Description
The Wizard element is the root element of the wizard framework. The wizard can also be used to define dialogs by configuring a single step wizard. A wizard consists of WizardStepActions, WizardStepElements for each page of the wizard, and WizardActions that are the standard wizard buttons on each wizard page.

Syntax
```xml
<!ELEMENT  Wizard
  (((Label?, ToolTip?, Image?) | HeaderLayout)
     WizardStepActions, WizardStepElements, WizardActions, On*,
     (DrawHandler |
      ((ModelHandler | ModelDoer),
       ActionDoer,
       (RenderHandler | Renderer)?
      )
     )
  )>
<!ATTLIST Wizard
  id CDATA                      #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

Attribute definitions

```
id =  text[CS]
The ID of the element.
```

Features

**Concrete Web Element**
com.ptc.core.ca.web.client.wizard.WebWizard

**Container Element**
yes

**Java Renderers**
com.ptc.core.ca.web.client.wizard.WebWizardRenderHandler

**JSP Renderers**
/wtcore/jsp/com/ptc/core/ca/web/wizard/wizard.jsp
WizardActions

Description

The WizardActions defines the set of buttons on each page per a standard wizard implementation. A common template of wizard actions is predefined using the name "wizard.actions". It defines the "Prev", "Next", "Ok", "Apply" and "Cancel" buttons for the wizard. This predefined element using the horizontal action list renderer.

Syntax

```xml
<!ELEMENT  WizardActions
  (Action+, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
  )>
<!ATTLIST WizardActions
  id CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
">
```

Attribute definitions

- **id** =  text[CS]
  - The ID of the element.

Features

- **Concrete Web Element**
  - com.ptc.core.ca.web.client.action.WebActionList

- **Container Element**
  - yes

- **Java Renderers**
  - com.ptc.core.ca.web.client.action.WebActionDropDownListRenderHandler
  - com.ptc.core.ca.web.client.action.WebActionHorizontalListRenderHandler

- **JSP Renderers**
  - /wtcore/jsp/com/ptc/core/ca/web/action/dropdownlist.jsp
  - /wtcore/jsp/com/ptc/core/ca/web/action/horizontallist.jsp
**WizardStepActions**

**Description**

The WizardStepsActions defines the set of actions to navigate between the steps of the wizard. Triggering an action will cause a step to become current. The visibility and accessibility status of the actions and the number of steps will control the visibility and accessibility of Next and Previous action in the list of wizard actions.

**Syntax**

```xml
<!ELEMENT  WizardStepActions
  (Action*, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
  )>
<!ATTLIST WizardStepActions
  id CDATA #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
>
```

**Attribute definitions**

- `id = text[CS]`
  
  The ID of the element.

**Features**

**Concrete Web Element**

- com.ptc.core.ca.web.client.action.WebActionList

**Container Element**

- yes

**Java Renderers**

- com.ptc.core.ca.web.client.action.WebActionHorizontalListRenderHandler
- com.ptc.core.ca.web.client.action.WebActionVerticalListRenderHandler

**JSP Renderers**

- /wtcore/jsp/com/ptc/core/ca/web/wizard/horizontallist.jsp
- /wtcore/jsp/com/ptc/core/ca/web/wizard/verticallist.jsp
**WizardStepElements**

**Description**

The `WizardStepElements` defines the set of steps to navigate through in the wizard. Triggering a step action with will cause a step with a correspondent index to draw.

**Syntax**

```xml
<!ELEMENT  WizardStepElements
  (Action*, ActionList*, List*, Property*,
   Connector*, On*,
   SimpleLayout*, CompositeLayout*,
   SimplePropertyPanel*, CompositePropertyPanel*,
   SimpleTable*, CompositeTable*,
   CompositeTree*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
 )>
<!ATTLIST WizardStepActions
  id CDATA #IMPLIED
  %dimensions;  #IMPLIED
  %conditions;  #IMPLIED
 >
```

**Attribute definitions**

- `id`: text\[CS\]
  - The nameID of the element.

**Features**

- **Concrete Web Element**
  - `com.ptc.core.ca.web.client.action.WebWizardStepList`
- **Container Element**
  - yes
- **Java Renderers**
  - `com.ptc.core.ca.web.client.action.WebWizardStepList`
- **JSP Renderers**
  - `/wtcore.jsp/com/ptc/core/ca/web/wizard/wizardsteplist.jsp`

**Deprecated Configuration Elements**

The configuration elements described by this section have been deprecated. Their renderers have been deprecated too unless they are shared by other ‘active’
elements. Both deprecated elements and their renderers a fully supported unless otherwise is explicitly stated.

Description of deprecated elements have not been changed to reflect deprecations of sub-elements or attributes.
Layout (deprecated)

Upgrade Instructions

1. Replace Layout by SimpleLayout element.

2. Replace
   com.ptc.core.ca.web.client.layout.WebLayoutRenderHandler
   by
   com.ptc.core.ca.web.client.layout.WebSimpleLayoutRenderHandler

3. Replace
   /wtcore/jsp/com/ptc/core/ca/web/layout/layout.jsp
   by
   /wtcore/jsp/com/ptc/core/ca/web/layout/simpleLayout.jsp

Description

The Layout element allows placement of various elements using a layout mechanism conceptually similar to <Table> HTML tag. The Layout element will place an element in the correct location on the screen based on the information configured in each LayoutRow element.

Syntax

```
<!ELEMENT  Layout
  (LayoutRow+, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
   )
  )
>)
<!ATTLIST Layout
  id CDATA               #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

Attribute definitions

```
id =  text [CS]
```

The ID of the element.

Features

Concrete Web Element
com.ptc.core.ca.web.client.layout.WebLayout

Container Element
yes
Java Renderers
   com.ptc.core.ca.web.client.layout.WebLayoutRenderHandler

JSP Renderers
   /wtcore/jsp/com/ptc/core/ca/web/layout/layout.jsp
**NavBar (deprecated)**

**Upgrade Instructions**

1. Replace NavBar by SimpleNavBar element.

2. Replace
   com.ptc.core.ca.web.client.layout.WebLayoutRenderHandler
   by
   com.ptc.core.ca.web.client.layout.WebSimpleNavBarRenderHandler

3. Replace
   /wtcore/jsp/com/ptc/core/ca/web/layout/navbar.jsp
   by
   /wtcore/jsp/com/ptc/core/ca/web/layout/simpleNavBar.jsp

**Description**

The NavBar element represents a menu or a navigation bar. Conceptually, it contains a tree of NavBarAction elements grouped together using NavBarRow elements. At run time it determines "current" action by comparing its location to the location of its frame. Once current action is determined, a list of NavBarRow elements containing ancestor actions is constructed and drawn. This gives a visual appearance of a context sensitive navigation bar.

In terms of rendering the rows defined in a NavBar, the default behavior is the same as the standard Layout Container. The difference between using a NavBar element and a Layout element is the NavBar element expects to have actions and is controlled by a set of business objects that are used to determine availability of the actions in the NavBar.

**Syntax**

```xml
<!ELEMENT NavBar
   (NavBarRow+, On*,
   (DrawHandler |
    (ModelHandler?,
     (RenderHandler | Renderer)?
    )
   )
   )>

<!ATTLIST NavBar
   id CDATA     #IMPLIED
   %dimensions;     #IMPLIED
   %conditions;    #IMPLIED
   %span;          #IMPLIED

>```

**Attribute definitions**

`id = text[CS]`

The ID of the element.
Features

Concrete Web Element
com.ptc.core.ca.web.client.menu.WebNavBar

Container Element
yes

Java Renderers
com.ptc.core.ca.web.client.layout.WebLayoutRenderHandler.java

JSP Renderers
/wtcore/jsp/com/ptc/core/ca/web/menu/navbar.jsp
**PropertyPanel (deprecated)**

**Upgrade Instructions**

1. Replace Property by SimplePropertyPanel element.

2. Replace
   
   com.ptc.core.ca.web.client.propertyPanel.WebPropertyPanelRenderHandler
   by
   
   com.ptc.core.ca.web.client.propertyPanel.WebSimplePropertyPanelRenderHandler

3. Replace
   
   /wtcore/jsp/com/ptc/core/ca/web/propertyPanel/propertyPanel.jsp
   by
   
   /wtcore/jsp/com/ptc/core/ca/web/propertyPanel/propertyPanel.jsp

**Description**

The PropertyPanel is a so-called instance holder element as it has associated Info*Engine element. It also extends a Layout element and this allows placement of various elements using a general layout mechanism.

Generally, PropertyPanel element is used to create a property sheet for a business object. By using a PropertyPanel, the Property elements children of the LayoutRow elements can access the attribute business data from the PropertyPanel element.

**Syntax**

```xml
<!ELEMENT  PropertyPanel
    (LayoutRow*, On*,
     (DrawHandler |
      ((ModelHandler | ModelDoer)?,
       (RenderHandler | Renderer)?
      )
     )
    )>

<!ATTLIST PropertyPanel
  id CDATA                     #IMPLIED
  %dimensions; #IMPLIED
  %conditions; #IMPLIED
  %span; #IMPLIED
>
```

**Attribute definitions**

`id = text[CS]`

The ID of the element.
Features

**Concrete Web Element**
com.ptc.core.ca.web.client.propertyPanel.WebPropertyPanel

**Container Element**
yes

**Java Renderers**
com.ptc.core.ca.web.client.propertyPanel.WebPropertyPanelRenderHandler

**JSP Renderers**
/wtcore/jsp/com/ptc/core/ca/web/propertyPanel/propertyPanel.jsp
Deprecation Attributes

frameType of HyperLink

Replace by window attribute.

computedValue of TaskParameter

Replace by function attribute. Requires writing a function.
Obsolete Configuration Elements

The configuration elements described by this section have been obsolete. Typically DCA deprecate configuration elements instead of obsoleting them to avoid the introduction of migration problems. In case of elements in this sections either no migration problems are expected or the severity of them is expected to be less critical than the inconvenience for the user of maintaining old configuration elements.

AppInstallationStatusDeterminator

Upgrade Instructions

Replace by InstallationStatusDeterminator.

Description

The AppInstallationStatusDeterminator element specifies a class that will be used to set up the installation status of an application.

Syntax

```xml
<!ELEMENT AppInstallationStatusDeterminator EMPTY>
<!ATTLIST AppInstallationStatusDeterminator
  id CDATA #IMPLIED>
```

Attribute definitions

`id = text[CS]`

The ID of the element.

Examples

```xml
<Context>
  <Init id="Init">
    <AppInstallationStatusDeterminator
      class="com.ptc.core.ca.co.
       client.misc.CoDefaultAppInstallationStatusDeterminator"
      appId="Applicaton A" version="6.2.6"/>
    <AppInstallationStatusDeterminator
      class="com.ptc.core.ca.co.
       client.misc.CoDefaultAppInstallationStatusDeterminator"
      appId="Applicaton B" version="5.1.3"/>
  </Init>
</Context>
```

When DCA reads configuration specifications it will set up the installation status as follows:

Applicaton A version 6.2.6
Applicaton B version 5.1.3
Features

Concrete Web Element
   no

Container Element
   no

Java Renderers
   Not available

JSP Renderers
   Not available

ApplicationSet

Upgrade Instructions
   Replace by InstallationSet.

Description
   The ApplicationSet element defines a set of applications that can be used to check their installation status.

Syntax

   <!ELEMENT ApplicationSet (Application*)>
   <!ATTLIST ApplicationSet
      id CDATA #IMPLIED
      %dimensions; #IMPLIED
      %conditions; #IMPLIED
   >

Attribute definitions

   id =  text[CS]
      The ID of the element.

Examples

   <Context>
      <ApplicationSet id="set1">
         <Application appId="Applicaton A" from="6.0.2" to="6.2.8"/>
         <Application appId="Applicaton B" from="4.2" />
         <Application appId="Applicaton C" to="7.0" />
      </ApplicationSet>
   </Context>
Features

Concrete Web Element
   no

Container Element
   no

Java Renderers
   Not available

JSP Renderers
   Not available
Obsolete Handlers and Renderers

Ratio property handler and renderer

LWC no longer support ratio attribute types. The following handler and renderers are obsolete.

Property Handler

com.ptc.core.ca.web.client.property.ration.WebRatioPropertyHandler

Java Renderer

com.ptc.core.ca.web.client.property.ratio.WebRatioRenderHandler

JSP Renderer

/wtcore/jsp/com/ptc/core/ca/web/property/ratio/ratio.jsp
Upgrades Requiring Manual Changes in Configuration Specifications

Elements

ActionList

Action list with a drop down renderer that has an additional entry (prompt) must specify it via Prompt element.

Old Syntax

```xml
<ActionList id="content.actionList">
  <Insert ref="action.download"/>
  <Insert ref="action.checkoutOpen"/>
  <Renderer class="com.ptc.core.ca.web.client.action.WebActionDropDownListRenderHandler"/>
</ActionList>
```

New Syntax

```xml
<ActionList id="content.actionList">
  <Prompt resource="&lt;span style='font:13px Arial,sans-serif';&gt;&lt;font color='grey';&gt;&lt;strong&gt;Pick an Action -&lt;br/&gt;&lt;/strong&gt;&lt;/font&gt;&lt;/span&gt;"/>
  <Insert ref="action.download"/>
  <Insert ref="action.checkoutOpen"/>
  <Renderer class="com.ptc.core.ca.web.client.action.WebActionDropDownListRenderHandler"/>
</ActionList>
```

SimpleTable
CompositeTable
CompositeTree

Tables and Trees in multiSelect mode must now specify a prompt for the Select All action via SelectAllPrompt element.

Old Syntax

```xml
<SimpleTable id="search.table.template" selectorType="multi">
  <Type type="%[context#type]"/>
</SimpleTable>
```

New Syntax

```xml
<SimpleTable id="search.table.template" selectorType="multi">
  <SelectAllPrompt resource="All"/>
  <Type type="%[context#type]"/>
</SimpleTable>
```
AppInstallationStatusDeterminator

InstallationStatusDeterminator should be used instead of this element. Installation status determinator can now set installation statuses of both applications and modules.

Old Syntax

```xml
<Context>
  <Init id="Init">
    <AppInstallationStatusDeterminator class="com.ptc.core.ca.co.client.misc.CoDefaultAppInstallationStatusDeterminator"
         appId="Application A" version="6.2.6"/>
  </Init>
</Context>
```

New Syntax

```xml
<Context>
  <Init id="Init">
    <InstallationStatusDeterminator class="com.ptc.core.ca.co.client.misc.CoDefaultAppInstallationStatusDeterminator"
         appId="Application A" version="6.2.6"/>
  </Init>
</Context>
```

ApplicationSet

InstallationSet should be used instead of this element. Installation set can now include both applications and modules.

Old Syntax

```xml
<Context>
  <ApplicationSet id="set1">
    <Application appId="Application A" from="6.0.2" to="6.2.8"/>
  </ApplicationSet>
</Context>
```

New Syntax

```xml
<Context>
  <InstallationSet id="set1">
    <Application appId="Application A" from="6.0.2" to="6.2.8"/>
  </InstallationSet>
</Context>
```
DataDrivenEnumerator

An attribute needed by a data driven enumerator must now be indicated as an option or a value. The order of the Need or NeedNpa sub-elements is not important.

It is recommended that you use `com.ptc.core.ca.co.client.windchill.CoWindchillEnumeratorDDEModelDoer` to create an enumerator based on the Windchill enumeration. See *Enumerator* chapter.

**Old Syntax**

```xml
<DataDrivenEnumerator>
  <Need attribute="dde.configList"/>
  <Need attribute="dde.defaultValue"/>
</DataDrivenEnumerator>
```

**New Syntax**

```xml
<DataDrivenEnumerator>
  <Need attribute="dde.configList" as="options"/>
  <Need attribute="dde.defaultValue" as="value"/>
</DataDrivenEnumerator>
```

Attribute marked as `options` are only needed for attribute-based enumerators. See *Enumerator* chapter.
Doers

com.ptc.core.ca.co.client_picker.CoNullModelDoer

Description
A model doer that does not perform any operations or changes in DCA objects.

Old Syntax

<SimpleTable>
...<ModelDoer class="com.ptc.core.ca.co.client_picker.CoNullModelDoer"/>

New Syntax

<SimpleTable>
...<ModelDoer class="com.ptc.core.ca.co.client.doer.CoNoOpModelDoer"/>

com.ptc.core.ca.co.client.search.CoPageBackwardActionDoer

Description
Paging action doer for navigating to the previous page that was developed prior to 6.2.6 is deprecated and can be replaced by the configuration shown below.

Old Syntax

<Action id="search.prevPage.alt">
      <Label resource="&lt; Prev"/>
      <ActionHandler class="%[PageInPlaceActionHandler]"/>
      <ClientDoer class="com.ptc.core.ca.co.client.search.CoPageBackwardActionDoer"/>
    </Action>

New Syntax

<Action id="search.prevPage">
      <Label resource="&lt; Prev {0}">
          <NeedNpa attribute="prev_count"/>
      </Label>
      <ActionHandler class="com.ptc.core.ca.co.client.search.CoPageInPlaceActionHandler"/>
      <ActionDoer class="com.ptc.core.ca.co.client.search.CoPageActionDoer"
      pageOffset="-1"/>
    </Action>
com.ptc.core.ca.co.client.search.CoPageForwardActionDoer

Description

Paging action doer for navigating to the next page that was developed prior 6.2.6 is deprecated and can be replaced by the configuration shown below.

Old Syntax

```xml
<Action id="search.nextPage.alt">
  <Label resource="&gt; Next"/>
  <ActionHandler class="%\[PageInPlaceActionHandler\]"/>
  <ClientDoer class="com.ptc.core.ca.co.client.search.
               CoPageForwardActionDoer"/>
</Action>
```

New Syntax

```xml
<Action id="search.prevPage">
  <Label resource="&gt; Next {0}"/>
  <NeedNpa attribute="next_count"/>
  </Label>
  <ActionHandler class="com.ptc.core.ca.co.client.search.
                  CoPageInPlaceActionHandler"/>
  <ActionDoer class="com.ptc.core.ca.co.client.search.
                CoPageActionDoer"
              pageOffset="1"/>
</Action>
```

com.ptc.core.ca.co.client.search.CoSearchActionClientDoer

Description

CoSearchActionClientDoer was developed to perform the following activities:

- compute where clause for the search operation
- reset paging parameters
- make the current frame as dirty in order to force an execution of a doer of a search result table

The last activity in particular presents a problem as it forces data acquisition for the search property panel itself. This results in a loss of values entered by the user.

The doer is deprecated and should be replaced by CoSearchActionDoer with some additional configuration elements, as shown below.
Old Syntax

```xml
>Action id="search.action">
  <Label resource="Search"/>
  <ActionHandler class="com.ptc.core.ca.co.client.search.
  CoSearchActionHandler"/>
  <ClientDoer class="com.ptc.core.ca.co.client.search.
  CoSearchActionClientDoer"/>
  <Renderer resource="/wtcore/jsp/com/ptc/core/ca/web/action/
  action.jsp"/>
</Action>
```

New Syntax

```xml
<Action id="search.action.alt">
  <Label resource="Search"/>
  <ActionHandler class="com.ptc.core.ca.web.client.action.
  WebSimpleInPlaceActionHandler"/>
  <ActionDoer class="com.ptc.core.ca.co.client.search.
  CoSearchActionDoer"/>
  <On event="executeActionDoer" at="end">
    <Copy from="%\[request#where_clause\]"
          to="%\[frame\#%context#tableId\].where_clause\]"/>
    <Copy from="true" to="%\[bean\#%context#tableId\]#Dirty"/>
    <Remove to="%\[frame\#%context#tableId\].paging_session_id\]"/>
    <Remove to="%\[frame\#%context#tableId\].page_offset\]"/>
  </On>
  <Insert ref="common:renderer.action"/>
  <Insert ref="eg.action.handler.progressMonitor
                if="%\[context#progressMonitor\]==true"/>
</Action>

<ElementGroup id="eg.action.handler.progressMonitor">
  <ActionHandler class="com.ptc.core.ca.web.client.action.
  WebProgressMonitorInPlaceActionHandler"/>
  <Location ref="common:frame.progress.monitor"/>
</ElementGroup>
```

**Note:** None of the handlers or doers requires `progressMonitor` dimension that is used only for demo purposes to present an example of configuring search functionality that works with or without progress monitoring.
CoSearchModelTaskDoer was developed to conditionally execute search task based on where clause is present and to handle some parameters related to search functionality. Since its introduction DCA developed an ability to specify task validators and task parameter functions. The doer is deprecated and can be replaced by CoDefaultSimpleModelTaskDoer with small additional configuration changes as described below.

Additionally, the does does not provide dca_type task parameter anymore and provides dca_object_type parameter instead.

Old Syntax

```xml
<ModelDoer class="com.ptc.core.ca.co.client.search.CoSearchModelTaskDoer">
  <Task nextOperation="STDOP|com.ptc.windchill.view"
    taskDelegate="dca-query-objects-with-sort-and-paging">
    <TaskParameter name="dca_type" value="%[WTDocument]"/>
    <TaskParameter name="format" value="false"/>
    <TaskParameter name="case_ignore" value="true"/>
    <Insert ref="search:eg.task.param.sort-n-page"/>
  </Task>
</ModelDoer>

<ElementGroup id="eg.task.param.sort-n-page">
  <TaskParameter function="%[ModelTaskParamWhereClause]"/>
  <TaskParameter function="%[ModelTaskParamPageCount]"/>
  <TaskParameter function="%[ModelTaskParamPageOffset]"/>
  <TaskParameter function="%[ModelTaskParamPagingSessionId]"/>
  <TaskParameter function="%[ModelTaskParamSortBy]"/>
  <TaskParameter function="%[ModelTaskParamSorted]"/>
</ElementGroup>
```

New Syntax

```xml
<ModelDoer class="%[common:SimpleModelTaskDoer]">
  <Task nextOperation="STDOP|com.ptc.windchill.view"
    taskDelegate="dca-query-objects-with-sort-and-paging">
    <Insert ref="search:validator.task.query"/>
    <TaskParameter name="search_type" value="%[WTDocument]"/>
    <TaskParameter name="format" value="false"/>
    <TaskParameter name="case_ignore" value="true"/>
    <Insert ref="search:eg.task.param.sort-n-page"/>
  </Task>
</ModelDoer>

<ElementGroup id="eg.task.param.sort-n-page">
  <TaskParameter function="%[ModelTaskParamWhereClause]"/>
  <TaskParameter function="%[ModelTaskParamPageCount]"/>
  <TaskParameter function="%[ModelTaskParamPageOffset]"/>
  <TaskParameter function="%[ModelTaskParamPagingSessionId]"/>
  <TaskParameter function="%[ModelTaskParamSortBy]"/>
  <TaskParameter function="%[ModelTaskParamSorted]"/>
</ElementGroup>
```
Both doers do not provide `dca_type` task parameter anymore. Instead they provide `dca_object_type` task parameter. DCA demo tasks have been modified accordingly.
Functions

com.ptc.core.ca.co.client.doer.function.impl.CoSimpleFunction_GetUfidForNearestInstance_As_ModelParam

Description
A function that locates an instance holder ancestor and uses its UFID as a task parameter.

Old Syntax
<TaskParameter function="com.ptc.core.ca.co.client.doer.function.impl.CoSimpleFunction_GetUfidForNearestInstance_As_ModelParam name="my_object"/>

New Syntax
<TaskParameter name="my_object" value="[%attribute#..#ufid]/">
DCA employs the inheritance and aggregation mechanisms to simplify application development. A few debugging tools exist to help application developers see the resultant DCA configuration after all Insert and Extend statements have resolved.

### Topic

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web-Based Configuration Viewer</td>
<td>B-2</td>
</tr>
<tr>
<td>Command Line-Based Configuration Viewer</td>
<td>B-2</td>
</tr>
<tr>
<td>Top-Level Element Viewer</td>
<td>B-2</td>
</tr>
<tr>
<td>DCA logging</td>
<td>B-4</td>
</tr>
<tr>
<td>Tracing Preparation of Configuration</td>
<td>B-8</td>
</tr>
<tr>
<td>Tracing Reference Resolution</td>
<td>B-8</td>
</tr>
</tbody>
</table>
Web-Based Configuration Viewer

To access the configuration viewer use the following URL:

http://<host>/Windchill/wtcore/jsp/com/ptc/core/ca/co/config/viewer/input.jsp

You will have an ability to specify the element ID and context dimensions:

<table>
<thead>
<tr>
<th>Element Id</th>
<th>Application Id</th>
<th>Type</th>
<th>Decoration</th>
<th>Other Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>:first.page.view</td>
<td></td>
<td>wt.lifecycle.LifeCycleTemplate</td>
<td>portlet</td>
<td>mode=edit;abc=def;ghi=jkl</td>
</tr>
</tbody>
</table>

For example:

The "Element Id" field is required; other fields are optional. Name value pairs in the list of "other attributes" must be separated by semicolons.

When submitted, the viewer will present configuration specification where Insert, Extend, and references (ref=":abc") are resolved:

Command Line-Based Configuration Viewer

Use the following syntax:

```
java com.ptc.core.ca.co.common.config.viewer.Viewer
    element_id
    [--d name=value]... [-f filename]
```

Where `element_id` is the id of the configuration element to view and `name` is the name of a dimension parameter with `value` and `filename` is the name of the file, XML output will be written into. The default location will be when no file name is specified:

<current directory>/Viewer.xml

Top-Level Element Viewer

Top-level element viewer helps with understanding where the top-level elements are defined and used. Upon execution the viewer creates two comma-separated files in the working directory.

To execute the viewer use the following syntax:

```
java com.ptc.core.ca.co.common.config.viewer.PrintRepository
```
**repository.csv**

*repository.csv* describes where the elements are defined. It contains a list of IDs of top-level configuration elements along with their context information, tag, package and file names. Every line in the file describes a single configuration element in the following format:

```
{id},tag,context,package,file
```

where,

- `<id>`
  - ID of the top-level configuration element
- `<tag>`
  - tag of the top-level configuration element
- `<context>`
  - context the top-level configuration element defined into
- `<package>`
  - package the top-level configuration element defined into
- `<file>`
  - file the top-level configuration element defined into

Below is an example of a typical output:

```
action.setSortingParam,Action,(),com.ptc.preferences,S:\Windchill\codebase\registry\dca\com\ptc\preferences\config_pref.xml
dialog.edit.task,Wizard,(type=CoolPart ),com.ptc.coolpart,S:\Windchill\codebase\registry\dca\com\ptc\coolpart\config_coolpart.xml
wizard.pref.actions,ActionList,(),com.ptc.preferences,S:\Windchill\codebase\registry\dca\com\ptc\preferences\config_pref.xml
```

**repository_xref.csv**

*repository_xref.csv* describes where the elements are used. The file contents a list of ids of top-level configuration elements along with the information about the elements that reference them using `ref` attributes. Every line in the file describes a single configuration element in the following format:

```
{id},package,ref by id,ref by tag,ref by context,ref by package
```

where,

- `<id>`
  - ID of the top-level configuration element
The following is an example of a typical output:

```
action.wizard.ok, com.ptc.common, wizard.pref.actions, WizardActions, (), com.ptc.preferences, S:\Windchill\codebase\registry\dca\com\ptc\preferences\config_pref.xml
renderer.enumeratedDropdownList, com.ptc.common, type, Property, (type=CoolPart), com.ptc.coolpart, S:\Windchill\codebase\registry\dca\com\ptc\coolpart\config_coolpart.xml
```

A special care is taken to present information about the elements references in global scope via \texttt{ref=":id"}. In this case configuration viewer presents no information about a package of the top-level element by placing \texttt{ANY} in its field. For instance:

```
source, ANY, layout.view.task, SimpleLayout, (type=CoolPart), com.ptc.coolpart, S:\Windchill\codebase\registry\dca\com\ptc\coolpart\config_coolpart.xml
```

**DCA logging**

**General Logging Functionality**

General logging functionality of DCA enables developers request DCA logging of various information in XML format. When logging is enabled, DCA generates a separate log file for every HTTP request. To store log files, DCA creates a new directory for every servlet engine instance in \texttt{Windchill/log} directory. The name of the directory has the following format:

```
DCA_<current_date_time>_session_<number>
```
**Debugging DCA B-5**

### Turning logging on and off

To turn general logging functionality on specify the following property in the `wt.properties`:

```
com.ptc.core.ca.co.log.rules.file=*
```

This will provide the most detailed log information for all logging themes (see below).

Sometimes developers are not interested in the information about how configuration elements are resolved. This is information is provided as part of the most detailed log and is quite voluminous. To disable the logging of such information provide the following logging rule (see below).

```
theme=element.build,%%remove=true
```

**Note:** The information about configuration element resolution is generated only during the first request to DCA to draw a frame with a particular address. It is not generated during subsequent requests for the same address.

### Selective logging

It is possible to selectively log debug information by specifying a file containing logging rules. The name of this file should be specified via the following property

```
com.ptc.core.ca.co.log.rules.file=<log file path>
```

For instance:

```
com.ptc.core.ca.co.log.rules.file=codebase\logrules.properties
```

The file containing logging rules is a regular property file where every property defines one logging rule. The following is an example of a typical set of rules:

```
log.rule.1=tag=*,id=*,%%remove=true
log.rule.2=tag=*,id=action.component.expand
log.rule.3=theme=config.*,%%remove=true
log.rule.5=theme=doer.*
log.rule.6=id=*first.page.view*,%%subtree=true
log.rule.7=id=com.ptc.folder:first.page.folder.frame.header,%%subtree=true
log.rule.8=*first.page*,theme=CONFIG.*,%%subtree=true
log.rule.9=theme=config.get,id=:first.page.folder,stack=true
```

Every rule is a comma separated sequence of name-value pairs. For instance, rule #2 contains two name-value pairs:

```
pair1 name: tag value: *
pair2 name: id value: action.component.expand
```
The rule #2 instructs DCA to log information about elements with any tags when their ids are equal to action.component.expand.

Usually the name corresponds to the name of an attribute of XML element in the log file. The easiest way to get familiar with all the names is to instruct DCA to produce the most detailed log file.

The value in the name-value pair can be any regular expression (there are some limitations at the moment as only Java 1.4 has predefined support for regular expression parsing).

There are a few special names that do not correspond to attribute names of XML elements in the log file. The name tag corresponds to the tag of XML element rather than to the attribute name. The name theme corresponds to the area in DCA of a particular interest. DCA, however, will not include information about the theme into log file unless the following property is specified in wt.properties:

```
com.ptc.core.ca.co.log.theme=true
```

Of course, the file will contain log for the themes specified in the log rules.

All possible themes are specified in source code of the class

```
com.ptc.core.co.co.common.log.CoLogHelper
```

DCA periodically checks the log rules file's time stamp and reload rules when needed. That gives developers a way to tailor logging rules not having to restart DCA.

**Evaluation of Logging Rules**

The easiest way to understand logging rules is to view them as a filter that is applied to the most detailed log. The most detailed log contains all possible XML elements describing a processing of a single request. An XML element passes the filter when its attributes match a rule. An attribute match a rule when there is a name-value pair in the rule that matches attribute name-attribute value pair.

Consider the following fragment from the most detailed log file:

```
<Request>
  <Request.Info>
    <Url url="/Windchill/wtcore/jsp/com/ptc/core/ca/web/gw/gw.jsp"/>
    <Parameters>
      <Parameter u8="1"/>
      <Parameter alias=":altpage.view"/>
      <Parameter ctype="wt.folder.SubFolder"/>
    </Parameters>
  </Request.Info>
</Request>
```

When the rule file contains a single rule:

```
log.rule.1=tag=Parameter
```
the resultant log file will contain only information about request parameters;

```xml
<Request>
  <Parameter u8="1"/>
  <Parameter alias=":altpage.view"/>
  <Parameter type="wt.folder.SubFolder"/>
  <Parameter ctype="wt.folder.SubFolder"/>
</Request>
```

To leave only information about the UFID, specify a more stringent rule:

```xml
log.rule.1=tag=Parameter,ufid=*  
```

This produces the following:

```xml
<Request>
</Request>
```

**Special instructions**

Every log rule can additionally include special instructions. By default, when XML log element matches the rule, it is included in the log file. An instruction `%remove=true` instructs DCA to filter out elements that match the rule containing it.

For instance, the following rule:

```xml
log.rule.120=theme=config.feature,%remove=true
```

filters out all elements for the `config.feature` theme.

All special instructions have format name=value. The name of a special instruction starts with special string `%%`. The value can be `true` or `false`. DCA supports the following instructions:

- `%remove`  
- `%subtree`  
- `%stack`

An instruction `%subtree=true` instructs DCA to pass through the filter not only the matching element, but also all of its subelements. For instance, to log information about the configuration element with `first.page.view` id, specify the following rule:

```xml
log.rule.121=theme=config*,id=first.page.view,%subtree=true
```

An instruction `%stack=true` instructs DCA to output stack trace during logging the matching element. Use this feature get better understanding of DCA workings. For example, the rule below will produce a stack trace when DCA configuration subsystem is requested the value of the configuration feature `width`:

```xml
log.rule.122=theme=config.feature,width=*,%stack=true
```
Tracing Preparation of Configuration

To trace the process of preparing configuration information set the following property to true in wt.properties file:

```
com.ptc.core.ca.co.common.config.reference.resolution.debug
```

The debug information will be written to the output stream of a servlet engine and to the $(wt.home)/logs/DCA.log file. Generally the file will contain one or more sections separated by special delimiters like the following:

```
=============== get configuration STARTED ===============
section log goes here
=============== get configuration FINISHED ===============
```

One section describes the process of preparing configuration information for a single request for configuration information. This usually a request to prepare configuration information for a new frame.

The section log contains information in XML format. It can be extracted and viewed using any available XML viewer.

Tracing Reference Resolution

To trace the process of resolving references set the following property to true in wt.properties file:

```
com.ptc.core.ca.co.common.config.reference.context.resolution.debug
```

The debug information will be written according to the rules described above. It will contain information about available options and decisions the system made. It will also include names of the files containing configuration fragments.
Currently, the majority of Windchill’s HTML pages are produced through Windchill’s HTML template processor mechanism. This mechanism predates Java Servlet Pages (JSP) and will be replaced over time by Windchill’s JSP-based Dynamic Client Architecture (DCA) and JSP pages in general. Until this replacement is complete, however, it is clear that a means must be provided to easily and efficiently migrate portions of the user interface based on the HTML template processor to DCA or JSP-based interfaces.

The Windchill HTML Template Processor Bridge makes such a migration possible by allowing requests for HTML template processor-based pages to be selectively re-mapped to JSP or DCA-based request handlers. This facility is intended for use by those customizing Windchill, both field consultants and customers, as well as by PTC development itself.

When used to re-map to DCA-based request handlers, the filter passes over to DCA the values of appId and decoration dimensions specified as request parameters.

The HTML Template Processor Bridge essentially acts as a filter on all requests received by the Windchill HTTP gateway servlets (HTTPGatewayServlet, HTTPAuthGatewayServlet, and WTServlet). It maps only HTTP GET requests which target wt.enterprise.URLProcessor (i.e., whose path-info begins with “/wt.enterprise.URLProcessor/”) which have a request handler registered for the given request action and target object type. All other requests are passed through to the pre-existing request processing logic.

**Topic**

Configuring the Bridge.......................................................................................................................... C-2
Configuring the Bridge

To activate the bridge itself, the wt.properties file must contain a wt.httpgw.filter.list entry.

This entry consists of a comma-delimited list of filter classes. One of the filter classes in this list must be:

```
com.ptc.core.ca.web.windchill.HTTPGatewayBridgeFilter
```

Thus the minimal entry would be:

```
wt.httpgw.filter.list=com.ptc.core.ca.web.windchill.HTTPGatewayBridgeFilter
```

In addition to enabling the bridge itself, the mapping files for the bridge must be specified. This is done through the wt.services.applicationcontext.httpGatewayBridgeFilter.defaultPropertyFiles and wt.services.applicationcontext.httpGatewayBridgeFilter.customPropertyFiles properties. The former property is intended for use by PTC to specify mapping files used as part of the product itself. The latter property is intended for use by customizers and customers.

**Note:** Like other wt.services.applicationcontent.*.defaultPropertyFiles properties, wt.services.applicationcontext.httpGatewayBridgeFilter.defaultPropertyFiles will default to reading from service.properties. This property should not be allowed to default in this manner, however, due to the initialization algorithm used by the bridge filter, else initialization time will be much greater than it should be.

The mapping files specified for the bridge should consist of entries similar to those found in service.properties and having the following form:

```
wt.services/rsc/default/HTTPGatewayBridgeFilter[.dca|.jsp]/{action}/{type}/0={destination}
```

where, the square brackets ([]) indicate an optional portion of the request and the | indicates an exclusive or. Thus an entry will start in one of the following fashions:

```
wt.services/rsc/default/HTTPGatewayBridgeFilter/…
wt.services/rsc/default/HTTPGatewayBridgeFilter.dca/…
wt.services/rsc/default/HTTPGatewayBridgeFilter.jsp/…
```

*action* is the value of the HTTP request parameter named “action” to be mapped (e.g. “ObjProps”).

*type* is the class or type name to be mapped (e.g. “wt.part.WTPart” or “wt.doc.WTDocument[SoftType|SubSoftType]”). For requests containing a “class” HTTP request parameter, the class or type name specified by this parameter will be compared against the *type* in the mapping entry. For requests containing an “oid” HTTP request parameter, the type of the object referenced
by this parameter will be compared against the type specified in the mapping entry instead.

destination is the destination the request is to be mapped to and will be formatted as follows:

- If “.dca” is specified (after “HTTPGatewayBridgeFilter”), then destination should be the desired value of the HTTP “alias” parameter optionally followed by ‘!’ and then the desired value of the DCA “ctype” parameter.

- If “.jsp” is specified (after “HTTPGatewayBridgeFilter”), then destination should the JSP page the request is to be handled by in class notation (e.g. “com.acme.MyJSPPage”).

- Otherwise, the destination should be a web-app relative path where the request is to be sent (e.g. “/wtcore/jsp/SomeJSPPage.jsp”).

Note: if destination is “.”, then the request is explicitly not mapped. In other words, requests best matching such an entry will get be allowed to pass through to the pre-existing request processing logic.

It should be noted that type can be specified as a base-type or modeled base class or interface and match sub-types or sub-classes (except where a more specific mapping has been established for a given sub-class or sub-type). Thus specifying type as “wt.doc.WTDocument” will establish a mapping for the given action for all document sub-classes and sub-types. Similarly, specifying type as “wt.doc.WTDocument|SpecDoc” will establish a mapping for the given action for the SpecDoc type and all of its sub-types. Note that specifying a non-modeled class or interface, e.g. “java.io.Serializable” for type will not apply to any sub-types of the specified type, with the notable exception of specifying “java.lang.Object” for type which will establish a default mapping for all classes and types for the given action.

Sample Mapping Entries

Some sample mapping entries follow, along with a description as to their effect. Note that the destinations of these entries are fictitious with the exceptions of the third and fifth mappings.

```plaintext
# following line maps request for list of cabinets to (Windchill web-app)/com/acme/jsp/cabinetListPage.jsp
wt.services/rsc/default/HTTPGatewayBridgeFilter/Cabinets/java.lang.Object/0=/com/acme/jsp/cabinetListPage.jsp

# following line maps request for cabinet properties page (top-level contents list) to ../cabinetPropPage.jsp
wt.services/rsc/default/HTTPGatewayBridgeFilter/ObjProps/wt.folder.Cabinet/0=/com/acme/jsp/cabinetPropPage.jsp
```
# following line maps request for change issue properties to a DCA details page for a change issue

```
wt.services/rsc/default/HTTPGatewayBridgeFilter.dca/ObjProps/wt.change2.WTChangeIssue/0=com.ptc.windchill.enterprise.change2.problemReports:simpleFrame.problemReportsDetailPage
```

# following line sends change items of type SpecialType (or sub-type thereof) to a special, separate DCA details page

```
wt.services/rsc/default/HTTPGatewayBridgeFilter.dca/ObjProps/wt.change2.WTChangeIssue|SpecialType/0=com.ptc.windchill.enterprise.change2.problemReports:simpleFrame.problemReportsSpecialDetailPage
```

# following line maps request for object properties on a ContentItem (no such links are currently generated) to a simple JSP page that determines if the user is allowed to see the given item and generates a direct link to the item

# note that this mapping applies to ApplicationData and URLData subclasses of ContentItem

```
wt.services/rsc/default/HTTPGatewayBridgeFilter/ObjProps/wt.content.ContentItem/0=/wtcore/jsp/report/bom/contentItemViewer.jsp
```
DCA Web provides a mechanism to enable trail file support when used with a test browser (browser_test) developed as part of Wildfire project, or a browser embedded in Pro/Engineer Wildfire. Both browser versions are capable of writing the information about user actions in a trail file, and of subsequently replaying user actions by reading from the file.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>D-2</td>
</tr>
<tr>
<td>Controlling Trail File Support</td>
<td>D-3</td>
</tr>
<tr>
<td>ID Structure</td>
<td>D-4</td>
</tr>
</tbody>
</table>
Introduction

When trail recording capabilities are enabled, every HTML element capable of accepting user input has a unique identifier (trail file identifier). Among such elements are anchors, select lists, and input fields.

The following is an example of a unique identifier:

```html
<INPUT TYPE="checkbox" name="selectAll"
ID="scom_ptc_folder:first_page_view!*:*:*!*$Layout$Row3$Layout$Row5$firstpage_table~~selectAll"
value="a441"
onClick="selectDeselectAll(this)";>
```

The following is a typical code snippet from a trail file.

```plaintext
~ Update `main_dlg_cur` `EMBED_BROWSER URL` `\`
~ Trail `main_dlg_cur` `bwTheBrowser` `\`
~ Load Complete`
~ Trail `main_dlg_cur` `bwTheBrowser` `\`
~ Click ID:`scom_ptc_folder:first_page_view!*:*:*!$Layout$Row3$Layout$Row5$firstpage_table~~selectAll`
```

In this example the trail mechanism recorded loading of a DCA demo page with an alias `com.ptc.folder:first.page.view`. It also recorded a subsequent click on the action with ID `selectAll`.

Use Cases for Trail Support

Trail file support has been developed primarily to support automated regression tests of DCA functionality in PDM Link for Pro/E Wildfire. The idea is to be able to record user actions in a trail file for a subsequent replay. However, more use cases have been observed.

Automated Regression Testing

The standalone browser can be used to automate regression testing of any DCA-based application or functionality. With trail file-based automated regression tests, the tricky thing is to decide on what to write into trail files. DCA can write the information that does not depend on persistent IDs of Windchill objects or persistent IDs of objects from other repositories. This gives the following benefits

- while playing a trail file, the standalone browser records a new trail file that can be compared to the original one. The files can be compared automatically to give an indication if something went wrong.
- a trail file can be replayed with future builds or even future versions of DCA-based applications (functionality) unless user visible functionality substantially changes.
- cosmetic changes (css and etc.) have no effect on trail files
• additions of new features like a property that is not required or a new action have to effect on trail file replay

• there is also no need to recreate a database to ensure that all persistent IDs match those recorded into a trail file prior to playing a file back as DCA does not depend on those IDs.

• extra parts and documents in the database don't affect replaying trail files as DCA does not record absolute or relative positions of the objects in tables or trees

Tech Support

DCA trail file recording mechanism can be turned on demand. This gives tech support an opportunity to ask an end-user to turn it on, perform some actions, and send a trail file as part of a problem report. Of course, this feature can be used for internal purposes too. For instance, to better communicate problems between developers.

Localization

A trail file recorded in US locale can be replayed on other locales. This gives localizers the opportunity to see the functionality they don't have to be familiar with to check for locale specific errors.

Training

Playback of previously recorded trail files can be used as a training tool.

Limitations

When trail file support is enabled, DCA turns off the mechanism to provide default values for input elements based on the last entered value (sticky properties). This is done to enable an ability to replay trail files independently on what the last entered values were. Otherwise, when the user accept the default, DCA trail mechanism does not record it and if the next time the default changes, trail can go out of sequence upon replaying.

Controlling Trail File Support

Trail file support can be permanently enabled by placing the following into wt.property

\[ \text{com.ptc.core.ca.web.client.gw.trailMode=true} \]

The default value of the property is false. When the property is not set to true, DCA Web will look for a User-Agent - a special part of the HTTP header. When Pro/Engineer Wildfire runs in a trail recording mode, it sets User-Agent to Wildfire WithTrail. When such a value is detected, DCA turns trail file support on for a current HTTP request.
Since the test browser allows to set User-Agent via the option in Action menu, it is possible to use the option to enable trail file support.

Finally trail file support can be dynamically enabled or disabled for a current session via HTTP request parameter trailMode by setting its value to true or false. For instance, the following URL turns trail file support on.


To replay trail file with the test browser, launch the browser with the following command:

browser_test -uitrail <trail file name>

**Caution:** At the moment DCA does not provide trail file support for actions triggered from applets. Only triggering of HTML-based actions is supported.

**ID Structure**

Trail file ID of HTML element is constructed using a full address of the correspondent DCA element. The following is an example of a typical ID:

`scom_ptc_folder:first_page_view!*:*:*!*Layout$row3$Layout$row5$firstpage_table~~selectAll`

This ID consists of a list of addresses of descendent DCA elements and a DCA element itself separated by $. The addresses are in so-called persistent form. More on it later. The following is a breakdown structure of the address above

**Address of the frame**

`scom_ptc_folder:first_page_view!*:*:*!*Layout$row3$Layout$row5$firstpage_table~~selectAll`

`com.ptc.folder:first.page.view` is the alias of the frame. No dimensions are specified.

**Static element address**

`Layout` - implicit ID of the layout. Configuration specification does not specify explicit ID.

**Static element address**

`row3` - explicit ID of the layout row specified in the configuration specification.

**Static element address**

`Layout` - implicit ID of the layout. Configuration specification does not specify explicit ID.

**Static element address**

`row5` - explicit ID of the layout row specified in the configuration specification.
Static element address

```
firstpage_table- explicit ID of the table specified in the configuration specification.
```

Static element address

```
selectAll - a name of the HTML element (select all box) within a table.
```

**Caution:** To satisfy requirements of the browsers, all dots (.) in the IDs are replaced by underscore (_) symbols.

Address Of The Frame

In its persistent form, an address of the frame contains an ID of the configuration frame element and a set of dimension values. It can also contain the type of the I*E element this frame is related to, but it never contains its UFID.

```
address of frame:=
  s<frame alias>!<app id>:<decoration>:<context type>[!<type>][!*]
```

Example:

```
scom_ptc_folder:first_page_view!SPL:*:wt_part_WTPart!wt_folder WTF older!*```

Static Element Address

In its persistent form a static element address contains an ID of the corresponding configuration element.

```
static element address:=<id of the configuration element>
```

When configuration element does not have an ID explicitly specified in the configuration specification, its tag is used as an implicit ID.

Example:

```
row5```

Dynamic Element Address

Dynamic element address presents the most challenge as the address corresponds to the DCA element that holds an I*E element. Since non-persistent form of the address contains a type and a UFID of the I*E element, the easiest solution would be to record them as part of a persistent form. However, this will render a trail support mechanism dependent on UFIDs that in turn depend on the data sources and their persistence identifiers. For instance, the UFID of the Windchill object includes its persistent ID.

To ensure that UFIDs of the I*E elements remain the same during recording and replaying, a special actions need to be taken. In the majority of the cases before replaying the data sources need to be restored precisely to the state they were prior
recording. While restoring data sources is possible, it is highly undesirable for many reasons.

The other significant limitation of the approach based on recording UFIDs is that trail files can not be used with different versions or builds of the software during development since UFIDs of the "same" objects can change.

DCA took a different approach in constructing a persistent form of an addresses of a dynamic element. Instead of recording a UFID, DCA records a combination of attributes of an Info*Engine element. What exactly needs to be recorded is configured via configuration specification. In its persistent form a dynamic element address contains a type of the Info*Engine element and a set of attribute values.

\[
\text{dynamic element address:} = \langle \text{type of Info*Engine element} \rangle \^\^ \\
\langle \text{attribute value} \rangle [\^\langle \text{attribute value} \rangle]
\]

Example:

\[
\text{wt\_folder\_WTFolder} \^\^ \text{Design}
\]

or

\[
\text{wt\_part\_WTPart} \^\^ \text{Wheel} \^ \text{versionA}
\]

**Trail ID Provider**

To configure the how and what gets recorded in a trail file instead of UFID, DCA introduced a concept of a Trail ID Provider.

```
<Interface>
GTailIdProvider

getId(element : GElement, uid : String, config : GConfigFeature, envData : GEnvData) : String
</Interface>
```

The trail ID provider is responsible for returning a string that gets recorded instead of UFID. Different providers can be specified in the configuration specification in different contexts as follows:

```
<Context>
  <TrailIdProvider id="TrailIdProvider" class="class A">
  ...
  </TrailIdProvider>
</Context>

<Context type="wt.part.WTPart">
  <TrailIdProvider id="TrailIdProvider" class="class B">
```
...  
</TrailIdProvider>  
</Context>  

Thus in the example above an ID provider implemented by class A is specified in the default context while another provided implemented by class B is specified in the context of a part.  

A trail ID provider can only be specified as a child of Context element.  

DCA Web provides a generic trail ID provider com.ptc.core.ca.web.client.misc.WebTrailIdProvider capable of using values of attributes specified via configuration specification. The following is an example of its configuration.  

<Context>  
  <TrailIdProvider id="TrailIdProvider"  
                    class="com.ptc.core.ca.web.client.misc.WebTrailIdProvider">  
    <Need attribute="name"/>  
  </TrailIdProvider>  
</Context>  

In this example the value of the attribute name will be used to create an ID.  

Example:  

wt_folder_WTFolder^^Design  

When more than one attribute value needs to be recorded, more Need statements can be specified. The following will record name and number of a document.  

<Context type=wt.doc.WTDocument>  
  <TrailIdProvider id="TrailIdProvider"  
                   class="com.ptc.core.ca.web.client.misc.WebTrailIdProvider">  
    <Need attribute="name"/>  
    <Need attribute="number"/>  
  </TrailIdProvider>  
</Context>  

wt_doc_WTDocument^^Gadget Doc^6934  

Situation Sensitive Recording  

While usually it’s desirable to record only a part name, sometimes more than one Info*Engine element of the same type exists with the same name. For instance, the table containing part versions will contain more than one entry with the same name. In such cases trail ID provider can be specified in multiple contexts for the same object type.  

<Context type=wt.part.WTPart>  
  <TrailIdProvider id="TrailIdProvider"  
                   class="com.ptc.core.ca.web.client.misc.WebTrailIdProvider">  
    <Need attribute="name"/>  
  </TrailIdProvider>  
</Context>
The second trail ID provider is specified in the context with additional user defined dimension versioned.

A table presenting part versions then can be configured as follows

```xml
<Context type=wt.part.WTPart versioned="true">
  <TrailIdProvider id="TrailIdProvider"
      class="com.ptc.core.ca.web.client.misc.WebTrailIdProvider">
    <Need attribute="name"/>
    <Need attribute="version"/>
  </TrailIdProvider>
</Context>
```

```xml
<LayoutRow id="a">
  <CompositeTable id="t" context.versioned="true">
    ...
  </CompositeTable>
</LayoutRow>
```

or can be inserted as follows

```xml
<LayoutRow id="a">
  <Insert ref="parttable" context.versioned="true"/>
</LayoutRow>
```

In both cases correctly configured trail ID providers will be used.

**Enabling Trail File Support by a Renderer**

All renderers provided as part of DCA distribution are trail file enabled. Custom written renderers, however, may have to undergo a change. To enable proper trail file support, all HTML input elements must have IDs that conform to DCA standards. For the purpose of this chapter an HTML element is input HTML element when it can be changed or triggered by the user.

The renderer can have zero, one, or more input elements.

**Zero Input Elements**

No changes required.

**One Input Element**

DCA provides a class
`com.ptc.core.ca.web.client.address.WebAddressUtils` to help with generation of IDs. The following is an excerpt from row.jsp

```xml
<INPUT><!--
WebAddressUtils.writeUniquePersistentId(element, out, envData);
%> TYPE="<%= input_type %>"><%
%> name="<%=WebAddressUtils.FIELD_SELECTED%>"><%
%> value="<%=addr_var_name%>>
```
More Than One Input Element

When one of the HTML input elements can be designated as primary, its ID can be generated as in the example above. For the rest of input HTML element unique string qualifiers must be assigned.

```html
<INPUT><!--
  WebAddressUtils.writeUniquePersistentId(property, "numerator",
  out, envData);
%
  TYPE="text" NAME="<%=numerator_param_name%>"
  ...

<INPUT><!--
  WebAddressUtils.writeUniquePersistentId(property, "denominator",
  out, envData);
%
  TYPE="text" NAME="<%=denominator_param_name%>"
  ...
```
A
Action Doer, 7-5
Action Doers vs. Client Doers, 6-5
Action Handlers, 6-2
Action Validators, 7-7, 7-21
Activating the Demo Configuration, 2-15

C
Composite Frame Containing a Composite Tree, 2-5
Creating and Modifying a Property Sheet to Support a Soft Type, 18-3

D
Data Acquisition and Manipulation - Doers, 6-8, 7-1
DCA and Soft Types, 15-1
DCA Architectural Fundamentals, 1-3
DCA Solutions, 1-4
DCA Web Architectural Fundamentals, 1-1, 2-1
DCA Web Architectural Fundamentals, 1-1, 2-1
Doers and the DCA/Web, 7-4

E
Element Addressing Scheme, 1-8
Every DCA WEB generated html page is a form, 3-6

G
Generating a Page From a URL Request, 1-9
Global Scope, 4-8

H
Handler Definitions, 6-6

J
JSP, 5-14

M
Modifying a Table Definition to Support a Soft Type, 18-4
Modifying Configuration Files to Support a Newly Created Soft Type, 18-1
Modifying Property Configuration to Use New Renderer or Handler, 5-14
Multiple customization points - handlers and renderers, 1-9

P
Pages Generated by DCA WEB are Defined in Configuration xml Files, 1-7

R
Refactoring sample links page for reuse, 3-5
Role of Action Handlers, 5-16, 6-1

S
Search Patterns, 2-45
Setting up Info*Engine to use DAC/Web, 1-7, 2-2
Simple Frame Containing a Simple Table, 2-3
Static and Dynamic Elements, 1-8

W
Web Implementation, 1-7
Writing JSP Renderers, 5-14

X
XML Files and Packages, 4-3