



Windchill Application Developer's Guide[®]

Windchill[®] 7.0

December 2003

Copyright © 2003 Parametric Technology Corporation. All Rights Reserved.

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

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

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC. UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION.

Registered Trademarks of Parametric Technology Corporation or a Subsidiary

Advanced Surface Design, Behavioral Modeling, CADD5, Computervision, EPD, EPD.Connect, Expert Machinist, Flexible Engineering, HARNESSDESIGN, Info*Engine, InPart, MECHANICA, Optegra, Parametric Technology, Parametric Technology Corporation, PHOTORENDER, Pro/DESKTOP, Pro/E, Pro/ENGINEER, Pro/HELP, Pro/INTRALINK, Pro/MECHANICA, Pro/TOOLKIT, PTC, PT/Products, Shaping Innovation, and Windchill.

Trademarks of Parametric Technology Corporation or a Subsidiary

3DPAINT, Associative Topology Bus, AutobuildZ, CDRS, CounterPart, Create Collaborate Control, CV, CVact, CVaec, CVdesign, CV-DORS, CVMAC, CVNC, CVToolmaker, DataDoctor, DesignSuite, DIMENSION III, DIVISION, e/ENGINEER, eNC Explorer, Expert MoldBase, Expert Toolmaker, GRANITE, ISSM, KDiP, Knowledge Discipline in Practice, Knowledge System Driver, ModelCHECK, MoldShop, NC Builder, PartSpeak, Pro/ANIMATE, Pro/ASSEMBLY, Pro/CABLING, Pro/CASTING, Pro/CDT, Pro/CMM, Pro/COLLABORATE, Pro/COMPOSITE, Pro/CONCEPT, Pro/CONVERT, Pro/DATA for PDGS, Pro/DESIGNER, Pro/DETAIL, Pro/DIAGRAM, Pro/DIEFACE, Pro/DRAW, Pro/ECAD, Pro/ENGINE, Pro/FEATURE, Pro/FEM-POST, Pro/FICIENCY, Pro/FLY-THROUGH, Pro/HARNESS, Pro/INTERFACE, Pro/LANGUAGE, Pro/LEGACY, Pro/LIBRARYACCESS, Pro/MESH, Pro/Model.View, Pro/MOLDESIGN, Pro/NC-ADVANCED, Pro/NC-CHECK, Pro/NC-MILL, Pro/NCPOST, Pro/NC-SHEETMETAL, Pro/NC-TURN, Pro/NC-WEDM, Pro/NC-Wire EDM, Pro/NETWORK ANIMATOR, Pro/NOTEBOOK, Pro/PDM, Pro/PHOTORENDER, Pro/PIPING, Pro/PLASTIC ADVISOR, Pro/PLOT, Pro/POWER DESIGN, Pro/PROCESS, Pro/REPORT, Pro/REVIEW, Pro/SCAN-TOOLS, Pro/SHEETMETAL, Pro/SURFACE, Pro/VERIFY, Pro/Web.Link, Pro/Web.Publish, Pro/WELDING, Product Development Means Business, Product First, ProductView, PTC Precision, Shrinkwrap, Simple Powerful Connected, The Product Development Company, The Way to Product First, Wildfire, Windchill DynamicDesignLink, Windchill PartsLink, Windchill PDMLink, Windchill ProjectLink, and Windchill SupplyLink.

Third-Party Trademarks

Adobe is a registered trademark of Adobe Systems. Advanced ClusterProven, ClusterProven, and the ClusterProven design are trademarks or registered trademarks of International Business Machines Corporation in the United States and other countries and are used under license. IBM Corporation does not warrant and is not responsible for the operation of this software product. AIX is a registered trademark of IBM Corporation. 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 GLOBETrotter Software, Inc. Geomagic is a registered trademark of Raindrop Geomagic, Inc. EVERSINC, 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 U.S. and 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>. 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/>) - their use is subject to the terms and limitations at: <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

<ftp://ftp.info-zip.org/pub/infozip/license.html>. 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. 040103

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

Content

About This Guide.....	vii
The Windchill Development Environment.....	1-1
Directory Structure.....	1-2
The codebase Directory.....	1-4
The src Directory	1-5
Artifact Management	1-6
Environment Variables	1-7
Class path.....	1-7
Path	1-7
SQL path.....	1-7
Rational Rose virtual path map.....	1-7
Property Files	1-8
wt.properties file.....	1-8
service.properties file.....	1-9
tools.properties file.....	1-10
db.properties file	1-10
Getting Started With Windchill.....	2-1
An Overview of the Windchill Development Process.....	2-2
Verify The Development Environment	2-2
Model the Object in Rose	2-5
Generate Java Classes From Rose.....	2-6
Create DatabaseTables	2-7
Initialize the Object	2-8
Design the GUI Layout	2-8
Code the GUI.....	2-9
Run the Applet in Netscape or Internet Explorer	2-10
Modeling Business Objects.....	3-1
Rational Rose and Windchill.....	3-2
Windchill Modeling Heuristics.....	3-4
Windchill Foundation Abstractions	3-8
Windchill Foundation Interfaces.....	3-8
Windchill Foundation Classes.....	3-10
The Command Layer	4-1

Command Beans	4-3
Command Delegates	4-9
Attribute Handlers	4-16
The Enterprise Layer	5-1
Enterprise Abstractions	5-2
Simple Business Class	5-2
Folder Resident Business Class	5-4
Managed Business Class	5-5
Revision Controlled Business Class	5-6
Iterated Folder Resident Business Class	5-8
Cabinet Managed Business Cclass	5-9
Document Abstractions	5-10
Part Abstractions	5-13
Design Overview	5-13
Change Abstractions	5-20
Change Item Classes	5-22
Associations with Product Information	5-25
Change Item Modeling Approach	5-28
Change Items Classes	5-29
Windchill Services	6-1
Windchill Packages	6-2
access package — Access Control Service	6-4
Design Overview	6-4
External Interface	6-6
Business Rules	6-6
Event Processing	6-7
admin package — Domain Administration Service	6-7
Design Overview	6-7
External Interface	6-8
Business Rules	6-8
Event Processing	6-8
content package — Content Handling Service	6-8
Design Overview	6-9
Working with ContentItems	6-12
Business Rules	6-14
Event Processing	6-14
content replication — Content Replication Service	6-14
fv.master — Master Service	6-14
fv.replica — Replica Service	6-14

intersvrcom — InterSrvCom Service	6-15
wrmf.delivery — Shipping Service	6-15
wrmf.delivery — Receiver Service	6-15
wrmf.transport — Generic Transport Service	6-16
Business Rules — content replication	6-16
effectivity package — Effectivity Service	6-16
Design Overview	6-16
External Interface	6-20
Business Rules	6-20
Event Processing	6-20
federation package — Federation Service	6-21
Design Overview	6-21
External Interface	6-22
Business Rules	6-23
Event Processing	6-23
folder package — Foldering Service	6-24
Design Overview	6-25
External Interface	6-26
Business Rules	6-27
fv package — File Vault Service	6-27
Design Overview	6-28
External Interface	6-29
Event Processing	6-29
identity package — Identity Service	6-29
Object Model	6-30
Implementing New Display Identification Delegates	6-33
Import and Export Package	6-33
index package — Indexing Service	6-34
Design Overview	6-34
External Interface	6-36
Business Rules	6-36
Event Processing	6-36
lifecycle package — Life Cycle Management Service	6-36
Design Overview	6-37
External Interface	6-38
Business Rules	6-39
Event Processing	6-40
locks package — Locking Service	6-41
Design Overview	6-41

External Interface	6-42
Business Rules.....	6-42
Event Processing	6-43
notify package — Notification Service	6-43
Design Overview	6-43
External Interface	6-45
Event Processing	6-45
org package — Organization Service	6-46
ownership package — Ownership service.....	6-47
Design Overview	6-47
External Interface	6-48
Business Rules.....	6-48
Event Processing	6-48
project package — Project Management Service	6-48
Design Overview	6-49
External Interface	6-50
Event Processing	6-50
queue package — Background Queuing Service	6-50
External Interface	6-52
Business Rules.....	6-52
Event Processing	6-52
router package — Routing Service	6-52
Design Overview	6-53
Example	6-55
scheduler package — Scheduling Service	6-56
External Interface	6-57
Event Processing	6-57
session package — Session Management Service.....	6-57
External Interface	6-58
Business Rules.....	6-58
Event Processing	6-58
Standard Federation Service doAction() and sendFeedback()	6-58
Design Overview	6-59
External Interface	6-60
Event Processing	6-60
vc package — Version Control Service	6-64
Design Overview	6-65
External Interface	6-69
Business Rules.....	6-70

Event Processing	6-70
vc package — Baseline Service	6-70
Design Overview	6-70
External Interface	6-71
Business Rules	6-71
Event Processing	6-73
vc package — Configuration Specification Service	6-73
vc package — Version Structuring Service	6-80
vc package — Version Viewing Service	6-82
vc package — Work in Progress Service	6-84
Design Overview	6-85
External Interface	6-86
Business Rules	6-86
Event Processing	6-87
workflow package — Workflow Service	6-87
Design Overview	6-87
External Interface	6-93
Business Rules	6-93
Engineering Factor Services	6-93
Handling CAD Models	6-94
Handling Family Instances	6-96
Handling Model Structure	6-98
The Relationship Between CAD Models and WTParts	6-100
Persistence Management	7-1
Persistence Manager	7-2
Store	7-4
Modify	7-5
Save	7-5
Delete	7-6
Refresh	7-7
Find	7-8
Navigate	7-9
Get LOB	7-10
Get Next Sequence	7-11
Prepare for Modification	7-11
Prepare for View	7-12
Search	7-12
Query	7-12
QuerySpec	7-13

SearchCondition.....	7-14
QueryResult	7-15
Multiple Class Queries	7-15
Transaction	7-16
Paging	7-17
Developing Server Logic.....	8-1
Service Management	8-2
Automatic Service Startup.....	8-2
Service Startup and Shutdown.....	8-3
Service Event Management.....	8-3
Service Event Registration	8-3
Service Event Subscription	8-4
Service Event Notification	8-6
Service Event Exception Handling	8-6
Service Event Conventions	8-7
Implementing Business Data Types	8-7
Initializing Business Attributes.....	8-7
Business Attribute Accessors.....	8-8
Overriding Accessor Methods	8-9
Validating Business Attributes.....	8-9
Implementing the checkAttribute Method.....	8-10
Business Attribute Aggregation	8-11
Business Attribute Persistence.....	8-11
Business Attribute Derivation	8-12
Implementing Business Services	8-12
Initializing Business Services	8-13
Business Service Operations	8-14
Vetoing Business Service Events.....	8-15
Business Service Rules.....	8-15
Business Service Communication.....	8-16
Lightweight Services	8-16
The Modeling Implementation	8-17
The Inner Class Implementation	8-18
Updating a Master Through an Iteration	8-22
Introduction.....	8-22
Read-only and Queryable Master Attributes on the Iteration	8-22
Updating Master Attributes via the Iteration	8-23
Updating the Master and Iteration Separately.....	8-23
System Generation	9-1

Overview of System Generation	9-2
How Rose UML Maps to Java Classes	9-2
Mapping Modeled Classes to Java	9-3
Mapping Operations to Java	9-8
Mapping Attributes to Java	9-10
Mapping Associations to Java	9-15
Implicit Persistable Associations Stored with Foreign ID References	9-18
Implementing Interfaces	9-24
Implementing the NetFactor Interface	9-25
Implementing the ObjectMappable interface	9-27
Implementing the Externalizable interface	9-28
Extending the EnumeratedType class	9-31
Stereotyping an interface as remote	9-32
How Rose UML Maps to Info Objects	9-33
How Rose UML Maps to Database Schema	9-34
How Rose UML Maps to Localizable Resource Info Files.....	9-38
Header.....	9-39
Resource Entry Format	9-39
Resource Entry Contents.....	9-40
Building Runtime Resources	9-40
Using the Windchill System Generation Tool	9-41
Registry Files	9-41
Generating mData Files	9-42
Generating Java Code.....	9-42
Generating Info Files	9-42
Generating SQL Files	9-43
Using Windchill System Generation in a Build Environment	9-44
Management of the mData File	9-45
Build Sequence.....	9-45
Command Line Utilities	9-46
Developing Client Logic.....	10-1
The Client Programming Model.....	10-2
Presentation Logic.....	10-3
Integrated Development Environments (IDEs)	10-3
Task Logic	10-3
Interacting with Business Objects.....	10-4
Client-Side Validation	10-5
Invoking Server Methods	10-5
Windchill Java Beans	10-6

Importing Windchill Beans to Visual Cafe	10-6
Package Dependencies	10-7
WTContentHolder Bean	10-7
WTExplorer Bean	10-11
PartAttributesPanel Bean	10-18
WTQuery Bean.....	10-21
WTChooser bean	10-23
EnumeratedChoice Bean	10-24
Life Cycle Beans	10-27
Spinner bean	10-28
WTMultiList bean.....	10-28
AssociationsPanel bean	10-29
EffectivityPanel Bean	10-33
FolderPanel Bean.....	10-35
AttributesForm Bean	10-37
ViewChoice Bean	10-43
PrincipalSelectionPanel Bean	10-46
PrincipalSelectionBrowser Bean	10-51
HTTPUploadDownload Panel Bean.....	10-57
Overview	10-57
API.....	10-58
Sample Ccode.....	10-59
Clipboard Support	10-61
Copying a WXObject to the Windchill Clipboard	10-61
Retrieving a WXObject From the Windchill Clipboard	10-61
Accessing the System Clipboard.....	10-62
Programming Outside the Sandbox Using security.jar	10-62
Threading.....	10-64
Refreshing Client Data	10-66
Registering as a Listener for the Event	10-66
Listening for the Event.....	10-67
Broadcasting the Event	10-67
Unregistering as a Listener to Events	10-68
Online Help	10-68
Preferences.....	10-71
The Preference Registry	10-72
Using Batch Containers	10-73
Design Overview	10-73
External Interface	10-74

The URLFactory	10-74
Writing a Mapping File	10-76
URLFactory in the JSP Environment	10-77
Setting the JSP Reference Point	10-78
Generating Links on the Page	10-80
Internationalizing JSP Pages in Windchill.....	10-81
The Rbinfo File Format.....	10-85
Internationalization and Localization	11-1
Background	11-2
The Windchill Approach.....	11-2
Localizing Text Visible to the User	11-4
Resource Info (.rbInfo) Files	11-7
Building Runtime Resource Bundles for Resource Info Files	11-9
Import Export Framework.....	12-1
Overview.....	12-2
Windchill User Authentication	A-1
User Authentication Framework	A-2
wt.method.MethodAuthenticator	A-2
wt.auth.AuthenticationHandler.....	A-3
wt.auth.Authenticator	A-4
Reference Implementation (HTTP authentication)	A-5
Null Authentication.....	A-7
Windchill Design Patterns	B-1
The Object Reference Design Pattern.....	B-2
The Business Service Design Pattern	B-2
The Master-iteration Design Pattern.....	B-6
Advanced Query Capabilities.....	C-1
QuerySpec.....	C-1
Descendant Query Attribute	C-1
Single Column Expression in SELECT Clause.....	C-1
Table Expression in FROM Clause	C-3
Expression in WHERE Clause.....	C-4
Bind Pparameters	C-7
Query Limit	C-7
SearchCondition	C-8
Compound Query	C-9
Access Control Consideration	C-11
Sorting	C-11
Join Support.....	C-13

Evolvable Classes	D-1
Background Information	D-2
General Externalization Guidelines.....	D-3
Hand-coded Externalization Guidelines	D-3
Migration Guidelines for Classes with Hand-coded Externalization.....	D-4
Examples of Generated Externalization Code for Evolvable Classes	D-4
Example of Generated Constants	D-4
Example of a writeExternal Method.....	D-4
Example of a readVersion Method.....	D-5
Example of a readOldVersion Method	D-6
GUI Design Process	E-1
Overview of the Design Process	E-2
Gather User Requirements	E-4
Define User Profiles	E-4
Define User Requirements	E-5
Analyze Tasks.....	E-6
Define Use Cases	E-7
Model Task Structure and Sequence.....	E-8
Develop Task Scenarios	E-10
Develop Storyboards.....	E-12
Design the GUI	E-13
Model Task Objects.....	E-13
Design the GUI.....	E-14
Prototype the GUI.....	E-15
Evaluate the GUI	E-17

Change Record

This table describes any major content changes or reorganizations since the last release.

Table 1 Changes for Release 7.0

Change	Description
Chapter 10, Developing Client Logic	Removed the out of date Client Side Access Control section. The available APIs are documented via Javadoc for the wt.access package
Chapter 12, Import Export Framework	This is a new chapter since Release 6.2.
Chapter 13, Xconfmanager Utility	This is a new chapter since Release 6.2. It details the xconfmanager utility and the Windchill command.

About This Guide

The Windchill Application Developer's Guide describes how to create applications using the Windchill product and how to customize existing Windchill applications. You are assumed to be familiar with the third party components used with Windchill: Java, Rational Rose, Oracle, and the IDE (integrated development environment) of your choice. Examples or screen shots use Visual Cafe as an example of a supported IDE, but Windchill supports any Java IDE customers choose to use. This manual does not address how to use these products in general, but how to use them as they relate to Windchill.

Related Documentation

The following documentation may be helpful:

- *The Windchill Customizer's Guide*
- *The Windchill Installation and Configuration Guide*
- *The Windchill Client Technology 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 email 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>

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>

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

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 the 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:

Convention	Item	Example
Bold	Names of elements in the user interface such as buttons, menu paths, and dialog box titles. Required elements and keywords or characters in syntax formats.	Click OK . Select File > Save . License File dialog box create_<tablename>.sql
<i>Italic</i>	Variable and user-defined elements in syntax formats. Angle brackets (< and >) enclose individual elements.	create_<tablename>.sql
Monospace	Examples Messages	JavaGen "wt.doc.*" F true Processing completed.

Convention	Item	Example
"Quotation marks"	Strings	The string "UsrSCM" . . .
	<p>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.</p>	<p>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.</p>

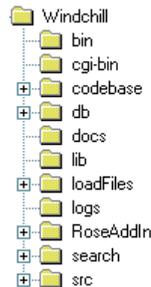
1

The Windchill Development Environment

Topic	Page
Directory Structure	1-2
Artifact Management.....	1-6
Environment Variables.....	1-7
Property Files	1-8

Directory Structure

The image below shows the Windchill directory structure after you install all available components. If Windchill was installed as recommended to follow this structure, go to the home directory where Windchill is installed and navigate through this structure as it is described here.



Windchill Directory Structure

bin

Contains various batch scripts, such as ToolsSetup.bat.

cgi-bin

Contains the Windchill common gateway interface wrappers.

codebase

Contains the runtime environment files.

db

Contains the database properties file and SQL scripts.

docs

Contains PDF versions of the Windchill manuals.

lib

Contains the wtbeans.jar file, which holds the Java bean components that have been developed for use with Windchill clients. For more information, see Chapter 10, Developing Client Logic.

loadFiles

Contains files used to load initial data.

logs

Default location for trace logs when logging is turned on.

RoseAddIn

Contains the Windchill extensions to Rational Rose (customized menus, property, and script files).

search

Contains Windchill extensions to Verity Search 97.

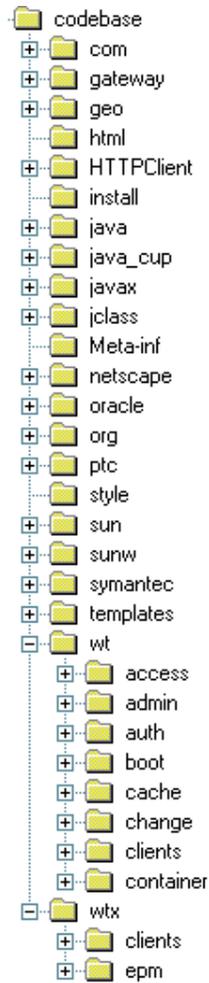
src

Contains development environment files.

The codebase directory and src directory are described in more detail in the following subsections.

The codebase Directory

In general, the codebase directory contains executable class files, property files, and other files that contain information needed by the runtime environment for Java and Web processing. The codebase directory can be expanded to show the following directories.¹



The codebase Directory

Most of these directories contain third party product class files.

1. To allow presentation in manual format, many subdirectories in the wt directory are not included in this figure.

The html directory contains templates that are used to generate HTML dynamically.

The wt directory contains the executable code for the packages supplied by Windchill (only a subset of which are shown in this illustration) and files required for localization, such as resource bundles and HTML files.

Within these packages are executable class files compiled from corresponding Java files of the same name in the src\wt directory. This set of Java source or .java files is created by the Windchill code generation tool for every business object modeled in Rose.

Files in the form *<object>* **.ClassInfo.ser** are also code-generated and contain metadata needed by the runtime environment. They are all described in more detail in chapter [9, System Generation](#).

Each package also contains resource bundles, that is, files that contain localizable information and are in the form:

*<packageName >***Resource.class**

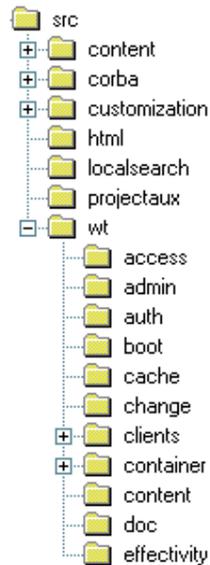
*<packageName >***ModelRB.RB.ser**

*<EnumeratedTypeClassName >***RB.RB.ser**

The wtx directory contains EPM-related files and directories.

The src Directory

In general, the src directory contains the source files needed for application development. It holds the files necessary to create new models in Rose and generate code. The src directory can be expanded to show the following directories.



The src Directory

The customization directory contains programming examples of customizations, which are described in more detail in either the *Windchill Customizer's Guide* or the readme file that is in the directory.

The wt directory contains the source files for the packages supplied by Windchill (only a subset of which are shown in this illustration).

In the wt directory and its package subdirectories, you will see the following kinds of files:

- Rose model components: model files with the suffix `.mdl` and package files with the suffix `.cat`. (The `.cat` file is described later in this chapter.)
- Code generation files: files with the suffix `mData`, which are intermediate files created by the code generation tool and used to create Java classes, Info classes, and SQL files (for more information, see chapter [9, System Generation](#)).
- Localizable ResourceInfo (`.rbInfo`) files, which contain customizable display text for EnumeratedTypes and modeled elements.

Artifact Management

In Rational Rose, you can optionally separate a model into controlled units as a way of sharing code. Controlled units are source files with a .cat extension that can be checked in and out of your source code control system.

Control units are created at the package level. In Rational Rose, right-click on a package in the Browser window, select **Units**, select **Control...; <package>**, select the filename for the controlled unit. The menu item is only selectable for packages that are not controlled.

This action produces a file with the extension .cat. You can then check in and check out these .cat files to whatever source code management system you use. The directory structure used in the source code management system must match the package structure used in Rose.

You can change a control unit that is read-only to write by selecting the WriteEnable<*package*> item. This is useful if you want to manipulate the model in some way (for example, temporarily changing its appearance before printing it). Be aware that this action does write enable the file at the file system level, so if you don't want it left writeable, select the Write Protect <*package*> item. These are menu items are both available from the same **Units** menu item mentioned above.

Environment Variables

When you install the Windchill Information Modeler, the settings required for business modeling and code generation are set automatically. You should have to do nothing more. But, if you have problems, the following sections give information about required settings.

Class path

We recommend that you not make any modifications to your CLASSPATH after installation. In particular, do not add the codebase directory. Doing so can cause Netscape Navigator to incorrectly assume a security violation. (See the *Windchill Installation Guide* for further information.)

Path

Your PATH variable should contain % WT_HOME% \bin and JDK\bin. % WT_HOME% contains batch scripts, such as ToolsSetup.bat, required to use the Information Modeler. (See the *Windchill Installation and Configuration Guide* for further information.)

SQL path

You must create (or add to) your SQLPATH variable and set it to the location where the SQL scripts that create the necessary tables will be written. When you are in a SQL*Plus session, SQL*Plus searches these directories for SQL scripts. By default, this value is c:\ptc\windchill\db\sql.

Rational Rose virtual path map

Rose provides a virtual path map mechanism to support parallel development by teams. The virtual path map enables Rose to create model files whose embedded path names are relative to a user-defined symbol. Thus, Rose can work with models moved or copied among workspaces and archives by redefining the actual directory associated with the user-defined symbol.

The virtual path map contains a list of entries, each of which represents a mapping between a virtual path symbol and an actual path name. To use this capability with Windchill, include the following virtual path map entries:

- WT_EXTENSIONS, which specifies the directory that contains the Windchill extensions to Rational Rose.
- WT_STD_PACKAGES, which specifies the directory where the packages for Java and JGL reside.
- WT_WORK, which specifies the top level directory where .mData files and .cat files are located. Every wt package modeled in Rose has a corresponding .cat file in the c:\ptc\windchill\src\wt directory.

- Module specific path maps will also be created automatically, based on information stored in `moduleRegistry.properties` and `moduleDir.properties`. For example, `wnc/CommonCore` is the path map for the `CommonCore` module of the `wnc` assembly.

Note: See the *Windchill Installation and Configuration Guide* for further information.

Property Files

Windchill uses standard Java property files to determine runtime configuration properties. The codebase directory contains:

- `wt.properties`
Contains properties used for general Java system configuration and Windchill system configuration.
- `service.properties`
Contains properties used by the Windchill service delegate mechanism.
- `debug.properties`
Contains properties used by Windchill code to control debug info capturing. (See Javadoc `wt.util` package for further information.)
- `user.properties`
Contains user overrides used by Rational Rose and the Windchill code generation tools.
- `moduleRegistry.properties`
Contains list of registered modules.
- `moduleDir.properties`
Contains home directory for each registered module.

The `db` directory contains:

- `db.properties`
Contains properties used by Windchill's database connection layer to access the database.

The System Generation jars (`SystemGeneration.jar`, `WindchillUtil.jar` & `CommonCore.jar`) contain:

- `tools.properties`
Contains properties used by Rational Rose and the Windchill code generation tools.

- debug.properties
Contains properties used by Windchill code to control debug info capturing.
- service.properties
 - Contains properties used by the Windchill service delegate mechanism, for the System Generation tools.
- wt.properties

This is an abbreviated form of the file that is in codebase.

(Care must be taken when using a manually created classpath that includes both codebase and System Generation jars, since properties files will be loaded based on the order of the classpath components.)

The following sections discuss only a subset that you as a developer are most likely to be interested in. A complete set of properties and descriptions for the wt.properties, tools.properties, and db.properties files can be found in the *Windchill Administrator's Guide* and the properties.html file in the codebase directory.

To change properties, you can edit the file directly or use the System Configurator GUI from the Windchill application home page. The System Configurator allows you to modify property files; start, stop, and restart the server manager and all method servers; and launch other Windchill applications.

wt.properties file

To use Windchill, the following properties must be set in the wt.properties file (this is usually done at installation). Note that you must use double back slashes to specify path names in the wt.properties file. This is necessary because the string is read by a Java program.

- wt.home, which specifies the top level of the class directory structure where Windchill is installed. The default value is c:\\windchill.
- wt.server.codebase, which is used by client applications (not applets). It specifies a URL from which client applications can download server resources such as property files. Client applets use their own codebase URL as specified in their APPLET tags. Server applications may use this property when writing dynamically generated HTML to be returned to a client browser. It is used to build URLs for static resources such as images or HTML files that reside under the server's codebase directory.
- java.rmi.server.hostname, which specifies a host name used to identify the server host. It is used by the Java RMI runtime for clients to look up the IP address of the server. It can be specified as a symbolic name, such as a fully-qualified Internet domain name, or numerically in dot notation (for example, 127.0.0.1). If not specified, the RMI runtime will use the name returned by InetAddress.getLocalHost() method, which may return a name that is not

known to remote clients. We recommend that this property be set to the fully-qualified Internet domain name of the server host.

You may also want to set the following properties:

- `wt.access.enforce`

This property enforces access control. By default, it is true. However, if you are debugging and want to bypass access control temporarily, you can set it to false.

- `wt.logs.enabled`

This property enables and disables logging in applications that support it, such as the Windchill Server Manager and Method Server applications. By default, it is false. To write debugging messages to a log file, you must set it to true.

- `wt.method.verboseClient` and `wt.method.verboseServer`

These properties cause trace messages to be printed from the client-side and server-side, respectively, of the method server remote interfaces. By default, it is false. Turning on these properties causes trace information to be written to logs for debugging purposes.

Similar properties are available for the server manager:

`wt.manager.verboseClient` and `wt.manager.verboseServer`.

In looking through the properties, you will see many service names followed by the word "verbose" (for example, `wt.access.verboseExecution` and `wt.access.verbosePolicy`). In general, these properties allow you to turn on debug tracing.

service.properties file

The `service.properties` file contains properties used by the Windchill service delegate mechanism. This mechanism is a general facility for adding delegate classes to an existing service to implement new, customized behavior. In this context, service can mean any sort of Java mechanism that provides functionality to other classes.

For example, assume a copy service exists that can make copies of certain classes of objects. The service knows how to make copies only for objects of certain classes: the classes for which copy service delegates have been created. Each delegate implements an interface, defined as part of the copy service, that contains the methods needed by the service. Once the delegate is created and put in the codebase, the copy service is notified of its existence by adding an entry to the `service.properties` file.

Generally, each service is accessed using a factory. The factory either returns instances of delegates to perform services on particular classes of objects, or it performs the operations on the objects itself by instantiating the necessary delegates internally.

If a Windchill service supports customization by adding delegates, the description of how to do the customization is described elsewhere in the documentation.

tools.properties file

The tools.properties file contains properties that are used by the System Generation tools. The following properties within wt.tools.properties are of interest:

- wt.generation.bin.dir, which specifies where .ClassInfo.ser files (serialized info objects) will be generated, following the package structure.
- wt.generation.source.dir, which specifies where .mData files are expected to be found and where .java files will be generated, following the package structure.

Note: Because the source.dir entry informs the code generator of the location of mData files and, in Rose, the WT_WORK variable informs the model information export tools of the location of mData files, they must point to the same location.

- wt.generation.sql.dir, which specifies where SQL scripts will be generated.
- wt.generation.sql.xxxTablesSize, which sets default sizes for tables.
- wt.classRegistry.search.path, which specifies the path to search for files representing classes to be registered.
- wt.classRegistry.search.pattern, which specifies the file pattern to consider as representing classes to be registered.

Note: Because tools.properties is contained within the SystemGeneration.jar, user overrides to these properties are placed in codebase\user.properties.

user.properties file

The user.properties file contains user overrides that are used by the System Generation tools.

Note: Configuration overrides for System Generation should be configured in user.properties using the xconfmanager utility. See the Windchill SystemAdministrator's Guide for information on the xconfmanager utility.

db.properties file

The db.properties file contains properties that are used by Windchill's persistence layer to access the database. They can be set in the wt.properties file but are usually kept in a separate file identified by the wt.pom.properties entry. Because a password is contained in this file, you should maintain it in a secure location. The values in the separate file override the values in the wt.properties file.

In the db.properties file, you must set the following properties:

- wt.pom.dbUser, which specifies the Oracle user name you or your Oracle administrator defined for you. This user is the owner of Windchill tables and stored procedures. There is no default; it must be set.
- wt.pom.dbPassword, which specifies the Oracle password you or your Oracle administrator defined for you. There is no default; it must be set.
- wt.pom.serviceName, which is the service name you or your Oracle administrator created using Oracle Net 8. There is no default; it must be set.

2

Getting Started With Windchill

The remainder of this manual describes how to create applications using the Windchill product and how to customize existing Windchill applications. You are assumed to be familiar with the third party components used with Windchill: Java, Rational Rose, Oracle, and the IDE (integrated development environment) of your choice. Examples or screen shots use Visual Cafe as an example of a supported IDE, but Windchill supports any Java IDE customers choose to use. This manual does not address how to use these products in general, but how to use them as they relate to Windchill.

This chapter gives you a brief overview of the Windchill development process and shows how to create a simple application using Windchill. By following this example, you will perform the major steps in developing an application and verify that the development environment (as described in chapter [1, The Windchill Development Environment](#)) is set up correctly.

Topic	Page
An Overview of the Windchill Development Process	2-2
Verify The Development Environment.....	2-2

An Overview of the Windchill Development Process

The process of developing Windchill applications is iterative and model-driven. You start with Rational Rose, an object-oriented analysis and design tool. Rose gives you the capability to create a graphic representation — that is, a model — of an application, its components, their interfaces, and their relationships. Windchill provides foundation classes that can be loaded into Rose so you can include in the applications any of the functionality already developed (for example, version control and the ability to lock objects). In Rose, you can create your own classes and extend those provided by Windchill.

When you finish the initial modeling phase, you use the Windchill code generation tool to create Java code from the Rose model. This generated code is already optimized to take advantage of Windchill's architecture. You can then implement in this code the server business logic and the client task logic. A Java IDE, such as Jbuilder, NetBeans, Eclipse or Visual Café (the IDE we have used), is useful for building the client-side presentation portions of applications that include HTML and Java applets. Examples or screen shots use Visual Cafe as an example of a supported IDE, but Windchill supports any Java IDE customers chooses to use.

Following testing, you can return to the model, make modifications, and repeat the process without losing work you have already done. This iterative, model-driven process allows you to create or customize an application, get portions of it running as quickly as possible, and continually improve it.

Verify The Development Environment

This section identifies the environment you need to run this example. It discusses only a subset of the information in chapter 1, The Windchill development environment. See that chapter for more detailed information on setting up the development environment.

We assume that Windchill and all required third party products are installed. If you did not install Windchill yourself, you will need to know where it is installed. These steps assume installation in the c:\ptc\windchill directory. You will also need to know your Oracle user name, password, and host name.

1. Verify the following environment variables:

- PATH

Ensure that the PATH variable includes the Oracle path. An example follows:

```
% [ ]SystemRoot% [ ]\system32;% [ ]SystemRoot% [ ];  
C:\ORANT\BIN;c:\jdk1.1.6\bin
```

- SQLPATH

The SQLPATH specifies the directory in which sql scripts are generated. It must match the value specified in the tools.properties file entry named

wt.generation.sql.dir. By default, this value is (wt.home)\db\sql. For example,

```
c:\ptc\windchill\db\sql
```

2. Verify the contents of the following property files:

- wt.properties

Set the following entry in the windchill\codebase\wt\wt.properties file:

```
java.rmi.server.hostname =<hostname >
```

For example,

```
java.rmi.server.hostname=Smith.windchill.com
```

- db.properties

In the windchill\db\db.properties file, ensure your Oracle user name, password, and service name are entered in this file.

```
wt.pom.dbUser =<username >
```

```
wt.pom.dbPassword =<password >
```

3. Start the Windchill servers. Open a new console window.

- Start the server manager and a method server by entering the following command:

```
java wt.manager.ServerLauncher
```

A window will be started for each but they will be minimized.

- Initialize the administrator tables by entering the following command in a console window:

```
java wt.load.Demo
```

This command loads the database with administrative information. Respond "yes" to all prompts. When prompted for a user name/password, enter the same user name and password you use to access your computer.

4. Establish your Rose model directory.

In the c:\ptc\windchill\src directory, create the subdirectory helloWorld. This directory will contain files for the helloWorld package.

Model the Object in Rose

1. Start Rational Rose and check the virtual path map. From the File menu, select Edit Path Map and ensure the following values are set:

```
WT_WORK = c:\ptc\windchill\src
```

```
WT_EXTENSIONS = c:\ptc\windchill\RoseAddIn
```

WT_STD_PACKAGES = \$WT_EXTENSIONS\Standard Packages

2. Establish the initial Rose model by performing the following steps:
 - a. From the File menu, select Open, browse to c:\ptc\windchill\src\wt, and load the model WTDesigner.mdl.
 - b. When asked whether to load subunits, press the Yes button.
 - c. Save the model as c:\ptc\windchill\src\helloWorld\HelloWorld.mdl.
 - d. When asked whether to save subunits, press the No button.
3. Model the person class by performing the following steps:
 - a. In the Logical View/Main class diagram, drop in a Package icon and label it helloWorld.
 - b. Use the dependency tool to draw dependencies from helloWorld to the wt and java packages.
 - c. Go to the Main diagram of the helloWorld package.
 - d. Drop on a class icon and give the class the name Item (the parent for Person). Attributes and operations for Item automatically appear. Change the diagram to suppress attributes and operations of Item. Ensure that the Show Visibility option is on for Item (so you can see in the diagram that it comes from the fc package).
 - e. Drop on another class icon and give it the name Person.
 - f. Make Person a subclass of Item. (Use the generalization icon/tool to draw a line from Person to Item.)
 - g. Insert the attributes name, age, title, and id. Name, title, and id should be strings (String) and age should be an integer (int). Use lowercase or a mix of upper- and lowercase letters for these attributes; do not use all uppercase letters.¹ Right click to start the specification dialog. Make all the attributes public and change the Windchill property of each to constrain=false. Click the Apply button for each change and, when you are done, click the OK button.
 - h. Select the menu option Browse > Units. Select the HelloWorld package and press the Control button. Save the unit to c:\ptc\windchill\src\helloWorld\helloWorld.cat. With the helloWorld package selected, press the Save button.

-
1. For every attribute modeled, a corresponding constant is generated using all uppercase letters. For example, if you model an attribute of employeeName, a constant of EMPLOYEE_NAME is generated. Modeling an attribute of EMPLOYEE_NAME would cause the generated constant to conflict with the attribute name. Therefore, unless the attribute being modeled is itself a constant, do not name modeled attributes using all uppercase letters.

- i. Save the Rose model file. When asked whether to save subunits, click the No button.

Tip:

From this point on, saving the model is a two-step process:

1. Select the menu option Browse > Units to initiate the Units dialog. In the Units dialog box, select the package you defined earlier and select Save. When asked whether to save subunits, click the **Yes** button. Close the Units dialog box.
2. Select the menu option File > Save. When asked whether to save subunits, click the No button.

Generate Java Classes From Rose

1. Go to the parent package of the Person class (by selecting the Logical View/Main diagram from the class browser, then selecting the helloWorld package).
2. From the Tools menu, select Windchill > System Generation.
3. From the popup window, select Java Source Code, WT Introspector Support, and Database Support, then click the OK button. The code generator creates several files:
 - An mData file (helloWorld.mData) and a Java file (Person.java), which are put in the directory c:\ptc\windchill\src\helloWorld that you created earlier in this example. (The generated mData file and java file go to the named package in the source directory based on the value specified in the tools.properties file entry named wt.generation.source.dir which, in this case, is c:\ptc\windchill\src.)
 - A ClassInfo file (Person.ClassInfo.ser), which is put in the directory c:\ptc\windchill\codebase\helloWorld (this directory is created by the code generator). (The location of this file is based on the value specified in the wt.generation.bin.dir entry which, in this case, is c:\ptc\windchill\codebase.)
 - SQL scripts that create the tables which store the Person class, and a directory named helloWorld in c:\ptc\windchill\db\sql which contains all the scripts.

You can verify this step by finding these files in the directories.

Create Database Tables

In Oracle SQL*Plus, create the tables by running the following command:

```
@helloWorld\make_helloWorld
```

This script creates a Person table and a PersonPk package. You may see a message the first time you run this script because the Person table does not yet exist; you can ignore this message.

Note: Establish the project using the IDE of your choice.

Initialize the Object

This step overrides a default initialize() method in order to set the person's id attribute.

1. Edit the Person.java file to add the following code for the initialize() method at the end of the file in the user.operations block:

```
protected void initialize() throws WTEException
{
    Date today = new Date();

    super.initialize();
    System.out.println("Person - initialize executing!");
    String s = String.valueOf(today.toLocaleString());
    setId(s);
}
```

2. Add the following import statement in the user imports block:

```
import java.util.Date;
```

3. From the File menu, select Save All and, from the Project menu, select Rebuild All.

Design the GUI Layout

1. From the Project menu, select the Object tab and double click on CreatePerson class.
2. Using the FormDesigner, add labels for Name:, Title:, Age:, and Id:. Use the Text property of the Label to set the label value.
3. Add TextFields for the three fields you just added, plus a TextField for the message area. The TextFields for Id and MessageArea should be set to enabled=false and editable=false. Name the fields nameField, titleField, ageField, idField, and messageField. messageField should span the width of the applet.
4. Add a button between the idField and the messageField. The button name should be saveButton, and its label text should be Save.
5. Save all your work.

Code the GUI

To make the applet function, you must associate an action event with the `saveButton`. The method that executes will construct a new `Person` object, set its values based on the input fields, and invoke the `PersistenceHelper` to save it in the database. After the object is saved, the method will update the `id` field on the GUI and put a status message in the `messageField`.

1. Edit the source of `CreatePerson.java`.
2. From the form designer, click the right mouse button on the Save button and select Bind Event. Select the `actionPerformed` event and press the Edit button.
3. Change the contents of the method to the following:

```
void saveButton_ActionPerformed(java.awt.event.ActionEvent event)
{
    Person p;
    int age;
    try
    {
        age = Integer.parseInt(ageField.getText());
    }
    catch (NumberFormatException nfe)
    {
        messageField.setText("Must supply number for age");
        return;
    }
    try
    {
        p = Person.newPerson();
        p.setName(nameField.getText());
        p.setTitle(titleField.getText());
        p.setAge(age);
        p = (Person) PersistenceHelper.manager.save(p);
    }
    catch (Exception wte)
    {
        wte.printStackTrace();
        messageField.setText("Exception: " + wte.toString() );
        return;
    }
    idField.setText( p.getId());
    messageField.setText("HelloWorld!");
    return;
}
```

4. Add the following import statements:

```
import wt.util.WTException;
import wt.util.WTContext;
import wt.fc.PersistenceHelper;
import java.lang.Integer;
import java.lang.NumberFormatException;
```

5. Insert the following statement as the first line of the `init()` method in the `CreatePerson.java` file:

```
WTContext.init(this);
```

Also delete the automatically generated line:

```
symantec.itools.lang.Context.setApplet(this);
```

6. Compile the whole applet by selecting Execute from the Project menu. You can now execute the project in Visual Café using the Applet Viewer or run it outside of Visual Café in a browser.
7. If you encounter an error that states the CreatePerson.class file cannot be found, close Visual Café. Copy the contents of c:\ptc\windchill\codebase\helloWorld to c:\ptc\windchill\src\helloWorld (these are the class files). Start Visual Café and execute the applet. (Be sure the method server is running.)

Run the Applet in Netscape or Internet Explorer

1. Create an HTML file with the following tags and save it as c:\ptc\windchill\codebase\helloWorld\CreatePerson.html. Replace <hostname> with the name of your Windchill system.

```
<HTML>
<BODY>
<APPLET code="helloWorld/CreatePerson.class"
        WIDTH=590 HEIGHT=600 codebase=http://<hostname>/Windchill>
</APPLET>
</BODY>
</HTML>
```

2. Open the CreatePerson.html file in Netscape or Internet Explorer.
3. At this point, confirm your Windchill method server is still running. Refer to the instructions given earlier under Verifying your development environment if it is not.
4. Enter the name, title, and age (where age must be a number) and press the Save button.
5. The method server should be started up automatically, if it is not already started. You should see in the output your message about the Person initialize() method.
6. In your applet, you should see the generated ID (that is, the system time) and the message "HelloWorld" in the messageField.

3

Modeling Business Objects

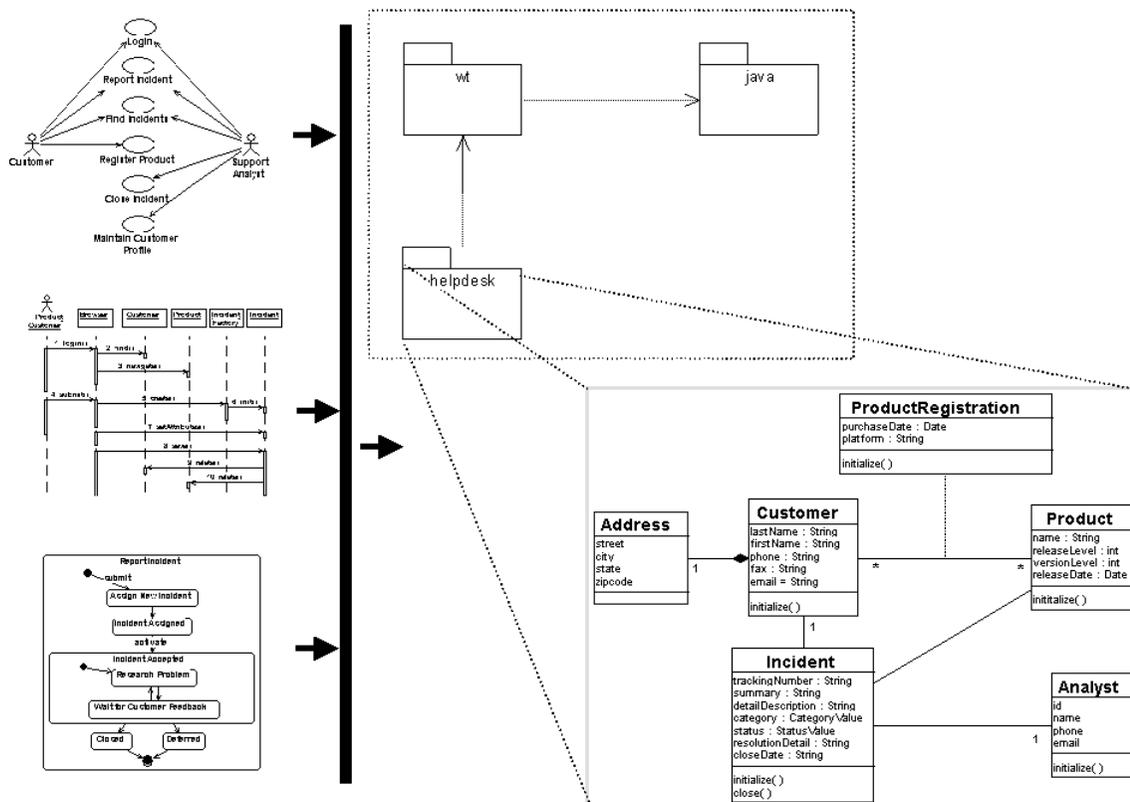
Topic	Page
Rational Rose and Windchill.....	3-2
Windchill Modeling Heuristics	3-4
Windchill Foundation Abstractions.....	3-8

Rational Rose and Windchill

The Windchill development environment uses the Rational Rose design and analysis tool to model business objects. Rose allows you to create a graphical representation of an application, its components, their interfaces, and their relationships. Windchill then uses this model to generate code that is used in server and client development.

You are assumed to be familiar with Rose or learning to use it. This section does not describe how to use Rose, but how Windchill interacts with it.

Rose offers several kinds of diagrams (as shown in the following figure) intended to help you analyze your model.

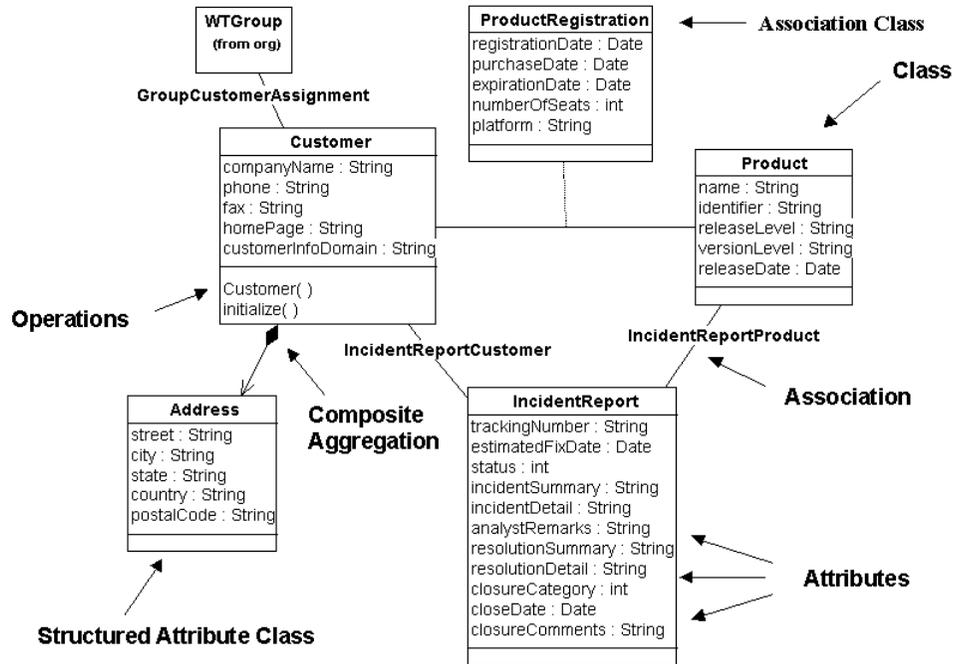


Rose Analysis Diagrams

On the left side are some of the various analysis diagrams available in Rose. From top to bottom, they are: a use case diagram, a sequence diagram, and a state transition diagram. The right side shows two class diagrams.

The class diagram is the only type of diagram used by Windchill for code generation. Although we recommend that you use analysis diagrams, that choice is up to you. Windchill requires only the class diagrams plus some system specifications that you set within Rose to generate code.

The following figure shows a class diagram for sample business information objects (Customer, Product, and IncidentReport) and the relationships between them. The annotation in bold points out UML (Unified Modeling Language) concepts and notation.



Sample Class Diagram

Address is a structured attribute class associated with the Customer class. The composite aggregation notation indicates that Address is considered a part of the Customer class. The Address object depends on the existence of the Customer object. For example, if the Customer class is deleted, the Address class is also deleted.

An association class allows you to add attributes, operations, and other features to associations. Because customers register for products, an attributed association is modeled between Customer and Product. IncidentReportCustomer and IncidentReportProduct are also link classes, but they have no attributes. You need not give them a class box.

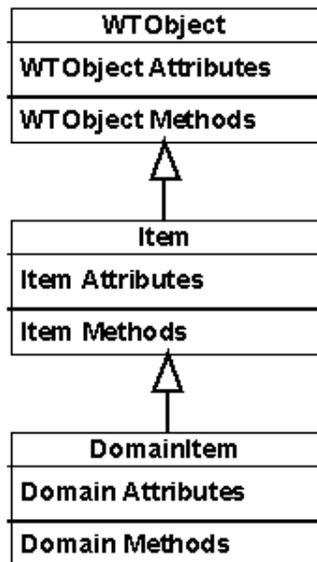
If any of these concepts or notations are unfamiliar, you should learn more about UML and Rose before proceeding. Many of the Windchill concepts are explained using class diagrams.

Windchill Modeling Heuristics

This section is intended to give you some background on the design of the Windchill class architecture and the approach we have used in modeling. The Windchill class architecture was designed with the following objectives in mind:

- To promote the development of business objects that reflect the characteristics of a three-tier architecture; that is, presentation for the client layer, business logic for the server layer, and persistence for the database layer.
- To ensure a model from which optimized code can be generated. The code that is generated should provide for managing the state of objects, transporting objects between the client and the server, and manipulating objects in the database.
- To provide an environment that enables value-added development. You must be able to extend the model and put new capabilities in existing objects.

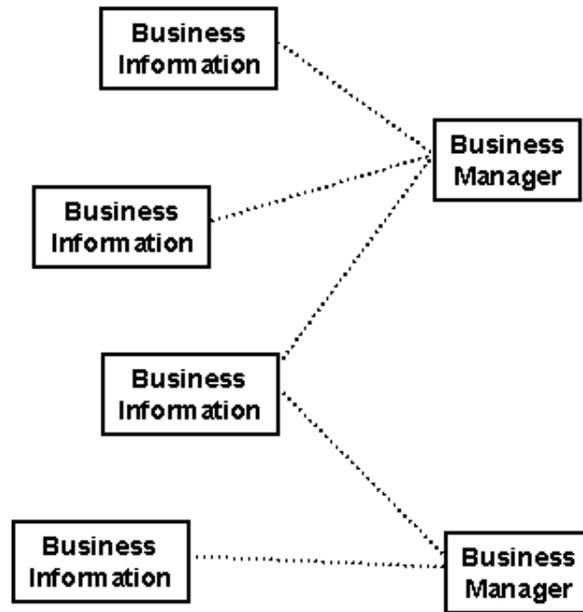
One approach to achieving these objectives is to inherit functionality.



Functionality through Inheritance

As you add subclasses to a parent class, in this case extending class WObject with Item and then DomainItem, the attributes and methods of each preceding class are inherited by the subclasses. This approach works in many circumstances. But sometimes you need only a small percentage of the functionality that is being inherited. In that case, either you have much more than you actually need in your object, or you copy just the code you want and add it to your own object, creating redundancy and potential maintenance problems.

Another approach, which we have implemented in Windchill, is to partition functionality, as in the figure below, into objects that are responsible primarily for maintaining business information (also called knowers) versus objects responsible primarily for performing business operations (also called doers).



Functionality through Partitioning

Using this approach, business models have two major kinds of classes: business information classes and business manager classes.

Business information classes represent the business information and associations you want to manage and maintain in a database. These classes extend the foundation classes provided in Windchill. They may implement one or more business manager interfaces. These classes go back and forth between the client and the server with data.

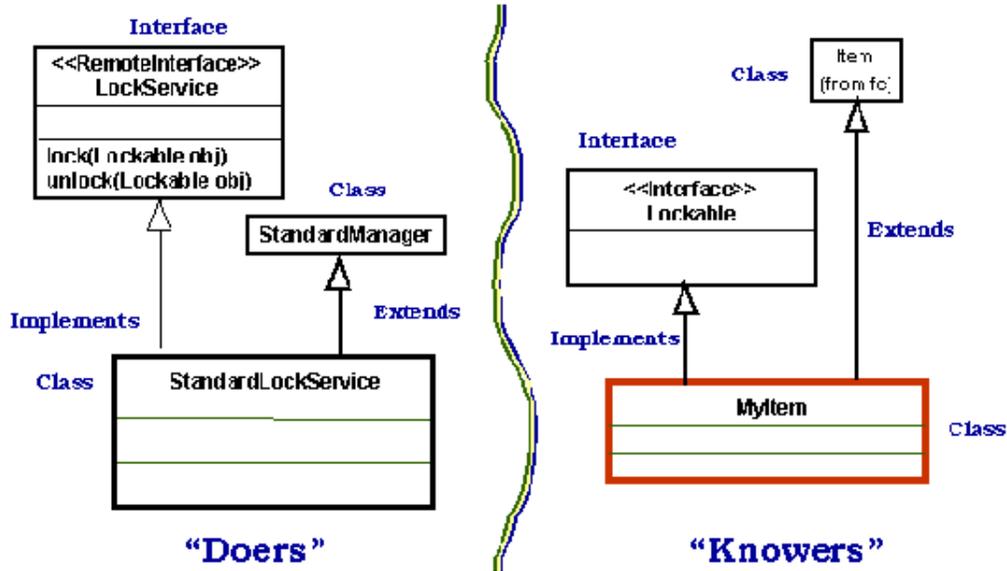
Business manager classes represent the business rules that are applied to the business information objects. These classes extend the Windchill StandardManager class. Business managers implement an interface class that provides a client-side API to the business manager methods. The code for business manager classes is located in the server.

Business managers are intended to implement small portions of functionality. The knower/doer separation approach allows you to choose which business managers, the doers, to implement for your business information, the knowers.

Windchill provides you with a number of managers (described in more detail in the chapter on server development) from which you can choose as much

functionality as you want, or design your own managers to meet your specific business needs.

The following is an example of one of the managers Windchill provides, the LockService. (In naming classes, managers are sometimes also called services.)



LockService Example

In this example, the knower is MyItem, which extends Item (a foundation class provided by Windchill). Thus, MyItem inherits all the attributes and behavior of Item. In addition, MyItem implements the Lockable interface.

The notation <<Interface>> is a stereotype. It is a cue to the code generator to add an Interface modifier to your generated class. This indicates that you must provide the code for this method in an implementation class. (In Java, an interface is a collection of operations and final fields without implementation methods which form an abstract type. A Java class conforms to an abstract type when it implements an interface. When a Java class implements a Java interface, the class or one of its subclasses must provide an implementation method for each operation in the interface.)

Besides the attributes and methods it inherited from Item, MyItem also has the functionality defined in the Lockable interface.

The left side of the figure shows how to model the interface for a server-side service on a client. The doer is StandardLockService and it runs on the server. It inherits from the Windchill StandardManager, which provides standard, basic operations for a typical manager, such as starting and shutting down. (When you write your own managers, they also can inherit from StandardManager.)

StandardLockService is the actual implementation of the lock service functionality and it implements the LockService remote interface.

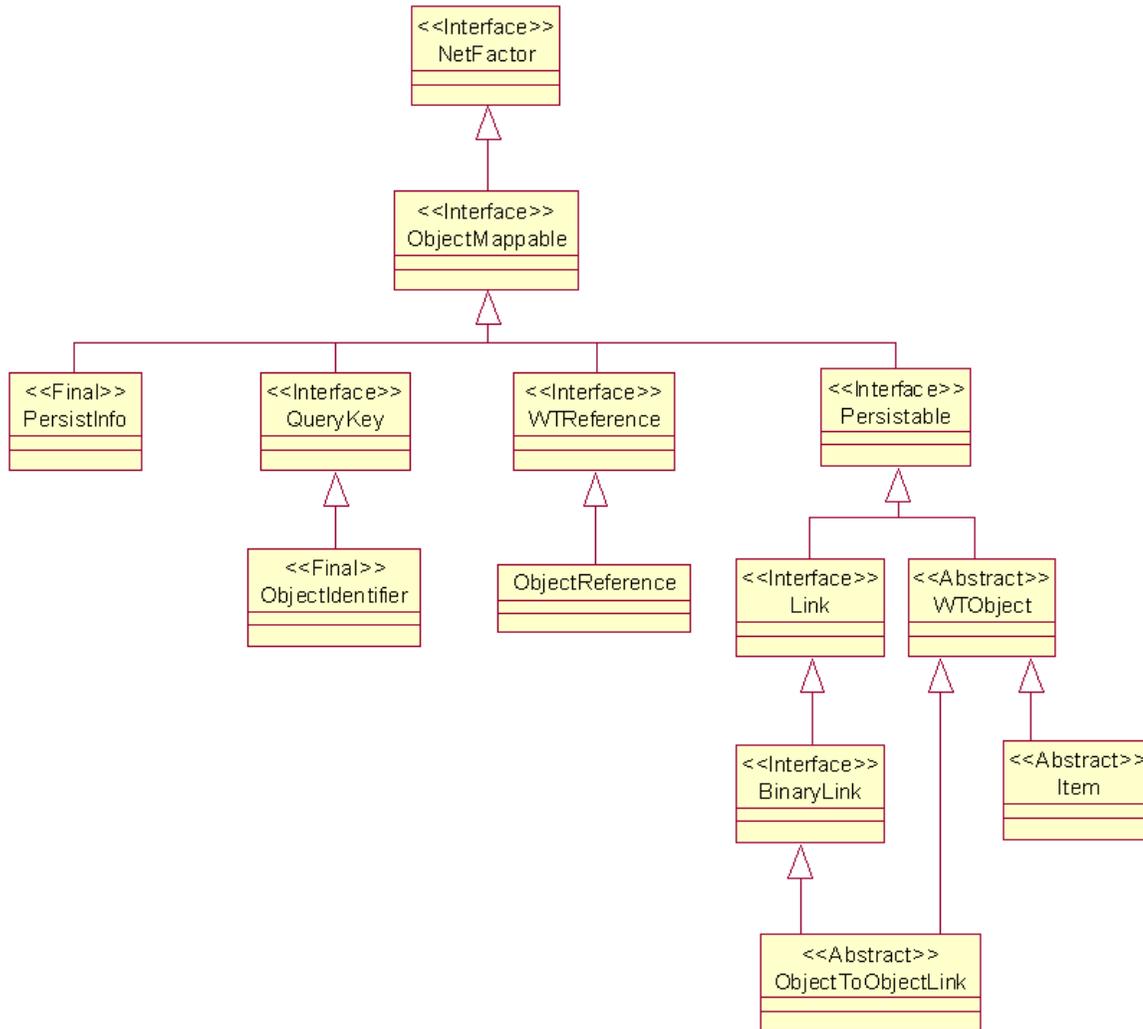
<<RemoteInterface>> is another stereotype. It is a cue to the code generator to create a forwarding class (described later in the chapter on code generation). The remote interface must be available on the client.

The LockService interface describes the lock services that are available on the client, lock and unlock. These services are invoked remotely from the client to the server. StandardLockService on the server actually contains all the code to support the lock and unlock methods.

StandardLockService expects a lockable object. It can accept MyItem because MyItem implemented the Lockable interface. Likewise, it can accept any other business information class that implements the Lockable interface. To get access to the lock service functionality, a class must implement the Lockable interface.

Windchill Foundation Abstractions

At an infrastructure layer of Windchill's architecture are foundational abstractions to be used by services and applications. These abstractions, shown in the following figure, represent the fundamental types that are commonly used by others as a part of a Windchill system.



Foundation Hierarchy

Windchill Foundation Interfaces

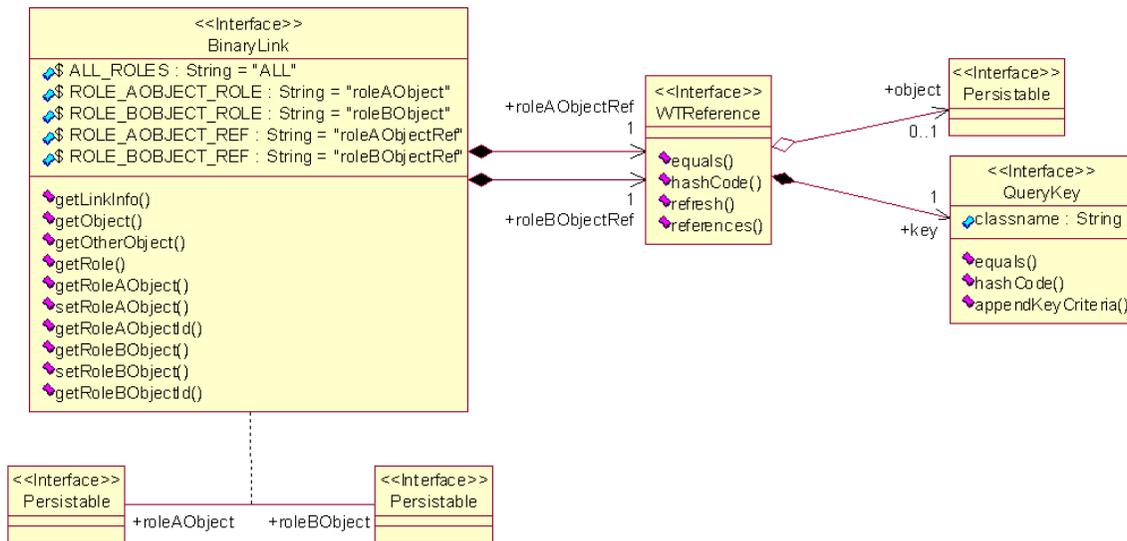
At the root of the hierarchy is the NetFactor interface. If an abstraction asserts itself as being an object of type NetFactor, it is classified as belonging to a Windchill system. One side effect of being a NetFactor type is that modeled constructors are generated as factories (for example, newMyItem), along with supporting initialization methods if none matching the same signature are found in the classes ancestry.

Classes that are asserted as being ObjectMappable can be written into and read from the database. All remaining abstractions in the foundation hierarchy are a kind of ObjectMappable abstraction. All subtypes of ObjectMappable are Serializable, which gives an object the ability to use RMI for travel between the client and server. Also, every abstract or concrete descendent must implement the externalizable methods writeExternal and readExternal specified in ObjectMappable. These methods provide custom serialization code to decompose and recompose objects to and from a stream.

When ObjectMappable is implemented, the code generator generates a readExternal and a writeExternal method. The writeExternal method takes the attributes of an object and writes them into the database. The readExternal method takes a result set from the database and turns it into attributes in an object. For an attribute to have a column in the database, it must implement ObjectMappable.

The PersistInfo interface contains information for each object that is stored in the database. PersistInfo does not implement Persistable; it is a structured attribute. It does, however, implement ObjectMappable. This means createStamp, modifyStamp, updateStamp, and updateCount will all be included in readExternal and writeExternal operations.

Links, object references, and query keys are generalized as interfaces as shown in the following figure.

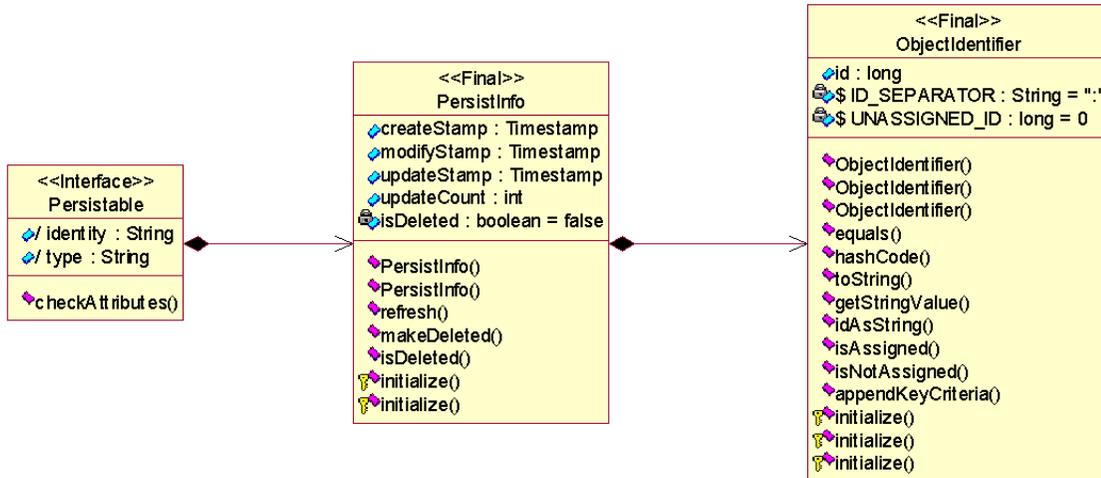


Binary Links

The QueryKey interface specifies a qualification for a persistable object in the database. It can be used as a primary key, secondary key, or a foreign key. The WTRReference interface specifies an indirect addressing mechanism in which it persists one key that, when used in a query, results in finding an object. If none or more than one object is found, this results in an exception. The object itself is transient and thus not persisted.

The Link interface specifies the concept of a container of roles and, in particular, the BinaryLink interface, a kind of Link, is an abstraction of an attributed member of an association between two persistable objects. The actual containment of the objects is done by aggregation of references for each role.

The Persistable interface gives an object a primary key (that is, the object identifier) as shown in the following figure, and a table in the database.



Persistable Objects

First class objects implement the Persistable interface. As a result, a database table is generated for each class in which their objects will be stored. The structured attributes are stored in the database table of their associated first class object. All persistable objects, plus any structured attributes that must be written into or read from the database, must implement the ObjectMappable interface.

Windchill Foundation Classes

Windchill provides three base classes with some basic functionality for business information objects: WObject, Item, and ObjectToObjectLink. Many business information objects provided by Windchill, and probably many that you create yourself, extend these foundation classes and, therefore, inherit attributes and methods from these classes. (We recommend that if you extend Windchill-supplied classes, you use those described in chapter 5, [The Enterprise Layer](#), which were designed to be used for customization.)

WObject

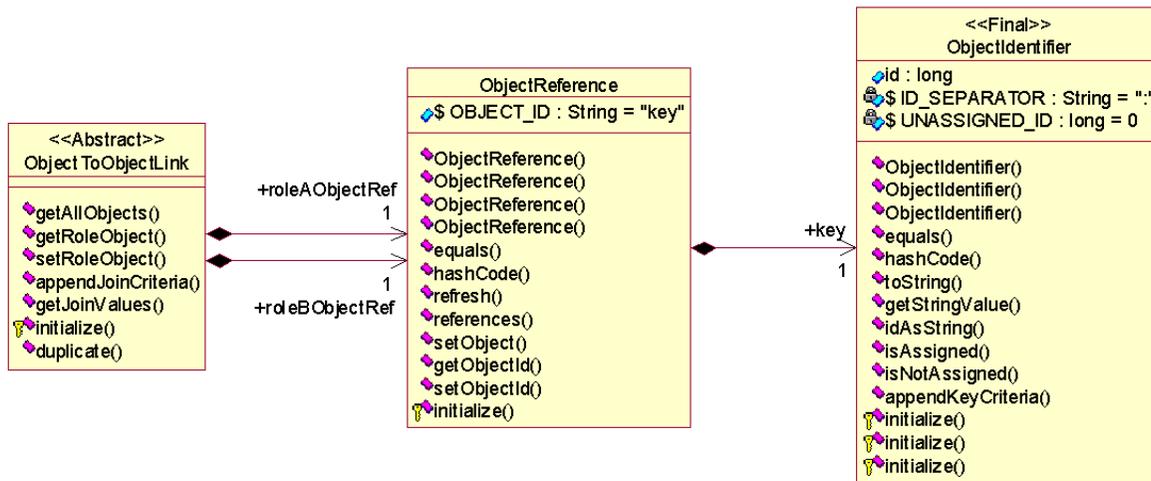
Represents the base class for all Windchill business information classes. Item and ObjectToObjectLink are subclasses of WObject.

Item

Represents a discrete business item.

ObjectToObjectLink

Represents a concrete binary association between two Persistable objects; that is, you can define a link between two items, between an item and a link, and between two links. Each link has a roleA side and a roleB side therefore, if you have a link, you can use it to navigate to all other objects associated with it. The ObjectToObjectLink class can be extended and therefore can have additional attributes and methods. As shown in the following figure, the ObjectToObjectLink aggregates ObjectReference for both role A and B. The ObjectReference in turn aggregates the primary key ObjectIdentifier as an overridden key to reference its object for both roles. The ObjectIdentifier extends QueryKey and adds the id as an additional attribute.



Object to Object Link

WTObj contains a few general-purpose methods that are inherited by every business object and provide a low level of functionality. For example, the `checkAttributes` method is called when the object is saved to perform elementary validity checking. If you have not supplied information for required attributes, an exception is thrown and the object is not made persistent.

4

The Command Layer

Topic	Page
Command Beans.....	4-3
Command Delegates.....	4-9
Attribute Handlers	4-16
Customizing Command Delegates	4-24

This chapter describes the classes and interfaces available in the following packages, and guidelines on how to develop commands:

```
com.ptc.core.command  
com.ptc.core.foundation  
com.ptc.core.query.command
```

Note: The above packages do not contain all of the commands available in the system. What they represent is a substantial core set of command beans, command delegates and attribute handlers.

The command layer provides a layer of abstraction and indirection. It encapsulates and hides from the consumer whether a command is executed locally, remotely or a combination of both. However, the remote executability of commands is only temporary based on a federated and type-based architecture, and thus clients should employ webjects or Info*Engine tasks for server-side commands to be executed in the server.

The command layer is designed such that it can be regarded as a Windchill design pattern that can be reused by many development teams. This Windchill command design pattern allows for commands and supporting classes to be easily reused to support a variety of consumers (e.g., JSP, Java applet/application, Webject), and to facilitate rapid development of new commands and execution logic.

The Windchill command design pattern supports the notion of a composite or macro command where it is a sequence of commands to be executed. The macro command can be executed, which then will execute each contained command in the given order from first to last. A macro command supports the ability for transactional blocks to be specified within the sequence of commands. This infers that not only would the whole sequence be able to be processed in a transaction, but also segments of commands within the sequence as well. Additionally, each command within the sequence can be adapted for instance to feed its output to the next command's input. The command adapters are only applicable within macro commands and can be extended to adapt a command's execution in other ways.

Commands in the command layer exchange type instances as input and output. They do not support heavyweight Windchill persistable types of objects. Windchill persistable objects are neither sent nor received by consumers when executing commands. Internally the commands' transform type instances as input to Windchill persistable objects. On output Windchill persistable objects are transformed back into type instances.

Commands support the ability to send intermittent and completion feedback to the client. As a part of this commands also support the logging of feedback as well as debug information to the server's file system.

Command Beans

Design Overview

The simplest way to begin an overview of the design is to describe an example. This does this by illustrating a “Create Part” scenario. At the start of this scenario the consumer makes a new `CreatePersistentEntityCommand` bean, sets various input fields, most notably the source `TypeInstance`, filter¹ and feedback specification, and executes the command.

By design since the `CreatePersistentEntityCommand` is a kind of command that is only executable in the server the corresponding command delegate is required to run in the method server’s JVM. Thus, a special type of command delegate that forwards to the server is made anew, the target delegate by name is set², and the forwarder is executed causing a remote method invocation to the method server passing itself. Once in the method server the forwarder makes a new command delegate given the name being the `CreatePersistentEntityCommandDelegate` and executes it passing along the command bean.

The `CreatePersistentEntityCommandDelegate` performs any pre-processing logic that is required. It then performs the execution logic which translates the given source `TypeInstance` into a `Persistable`, stores the `Persistable` using the persistence layer, and translates the returned `Persistable` from the persistence layer back into the result `TypeInstance`. Now the `CreatePersistentEntityCommandDelegate` performs any post-processing logic that is required³.

Once the `CreatePersistentEntityCommandDelegate` has finished with the post-processing it returns as a result a `TypeInstance` to the consumer. The consumer at this point is then free to do whatever with the result.

-
1. The filter that is an `AttributeContainerSpec` contains in part an array of `AttributeTypeIdentifiers` indicating what attributes to return, and two boolean flags indicating whether or not to include constraints and descriptors in the result.
 2. The name of the target delegate is used instead of the target delegate itself to ensure that no unnecessary class loading occurs within the consumer’s JVM, assuming it is a client. Also, what is not shown in is the lookup mechanism that enables the target delegate’s name to be found. This mechanism is the `TypeBasedWTServiceProvider`.
 3. This algorithm of pre-processing, execution, post-processing is an enforced logic pattern that all command delegates will do. The pre-and-post-processing segments represent macro-customization points where any specialized logic can be plugged in.

With the simple “Create Part” scenario described command bean design details can now be discussed. The Create Part Sequence Diagram shows the key part of the overall command bean classification hierarchy.

Within this classification hierarchy there are two main types of command beans, local and server commands. Local commands are executable in any JVM being in either the client or server. These types of commands are useful when only local work needs to be accomplished with a uniform and consistent set of APIs. Server commands are only executed within the server’s JVM. They currently are remotely executable within a client’s JVM. Again the remote executability of commands is only temporary based on a federated and type-based architecture, and thus clients should employ webjects or Info*Engine tasks for server-side commands to be executed in the server.

The two main types of server commands are entity and repository commands. Entity commands provide operations against business objects (i.e., a business object is notionally an entity) like CRUD, check out, check in, revise, etc. Repository commands provide operations against the container of persistent business objects like performing a variety of queries on a database. A repository represents anything that acts as a container of business objects. This container could, instead of a typical database, alternatively be a file system or some other mechanism that can keep business objects.

External Interface

The Command interface provides the main abstraction of a command bean. It specifies the following key APIs that all local and server types of commands must implement:

1. locale field: the locale to be used should any locale specific strings need to be created. Accessors are generated for this private field.
2. execute() method: carries out the execution of the command. This method employs the CommandDelegateFactory to get the appropriate CommandDelegate, and then executes it passing the instance of the Command. If for any reason there was erroneous, unrecoverable behavior then a type of CommandException is thrown.

The LocalCommand interface partitions the command bean classification hierarchy for local types of commands. The abstract class of the LocalCommand implements the execute() method to enforce that the command is executed within the present JVM. The LocalCommand does not specify any more APIs.

The ServerCommand interface partitions the command bean classification hierarchy for server types of commands. The abstract class of the ServerCommand implements the execute() method to enforce that the command is executed within only the server's JVM. The ServerCommand specifies the following key APIs that all server types of commands must implement:

1. filter field: the specification of what attributes, constraints and descriptors are to be returned in the command's result TypeInstance. Accessors are generated for this private field.
2. getResultList() method: derives a TypeInstance list [an array] from the underlying command bean's result type. This method is required to be overridden by subclasses that specify a particular kind of result within a command bean sub-classification where the result is different than what is currently supported. For example, entity commands are required to return a single TypeInstance, but in the case of the CheckoutFromVaultCommand it returns both the working and reserved entities. Thus, the getResultList() method is overridden to return an array of both of those business objects.

The AbstractServerTransaction class represents the notion of a special type of server only command being a transaction. A transaction is not a normal command in the sense that it is not executable. Instead its sole purpose for existing is to enable the marking of transaction blocks within a sequence of commands in a MacroServerCommand. Any consumer of this type of command that attempts to execute it will get a CommandException that wraps an UnsupportedOperationException.

The BeginServerTransaction class marks the beginning of where a transaction block is to start in the sequence of commands within a MacroServerCommand. This beginning of a transaction block can be placed anywhere within the sequence provided that an EndServerTransaction follows it.

The EndServerTransaction class marks the end of where a transaction block is to end and thus the transaction committed at that time in the sequence of commands within a MacroServerCommand. This end of a transaction block can be placed anywhere within the sequence provided that a BeginServerTransaction precedes it. And there must be at least one command within the transaction block.

The MacroServerCommand provides an implementation of a Composite design pattern, and an aspect of the Command design pattern where there's the notion of a "macro command" that contains a sequence of commands, all of which are executable in order. It is a type of Command itself and aggregates a list of commands that can be constructed by the consumer. This list is the sequence of commands. Transaction blocks can be included into the sequence of commands by means of the BeginServerTransaction and EndServerTransaction classes. Instances of them can be inserted into the list of commands to form transaction blocks. These special transaction types of commands then act as tags that mark the beginning and ending of a transaction block.

The MacroServerTransaction is a type of MacroServerCommand that for convenience provides a transaction block around the whole sequence of commands.

Entity Commands

The AbstractEntityCommand class provides an abstract representation of a command that can act upon an entity in the system. An entity is a business object that is either transient or persistent. This abstract class specializes the notion of a general command for an entity with the following key APIs.

1. source field: the input TypeInstance to be acted upon. Accessors are generated for this private field.
2. result field: the output TypeInstance to be returned. The filter specifies what attributes, constraints and descriptors to include in the TypeInstance for return. Accessors are generated for this private field.

The AbstractPersistentEntityCommand class provides an abstract representation of a command that can act upon a persistent entity in the system. This abstract class specializes the notion of a general entity command by partitioning the classification hierarchy such that below it is only commands designed to be used for persistent entities (e.g., CRUD).

The AbstractIterationEntityCommand class further partitions the classification hierarchy to define a set of commands that act upon persistent, iterated types of entities in the system. This is where the notion of being able to do version control and work-in-progress operations becomes realized. The AbstractIterationEntityCommand specifies the following key APIs:

- targetId field: the iteration or version identifier that is to be the next identifier of the iterated/versioned entity upon completion of the command's execution. For example, the CheckinToVaultCommand can have this field specified to be a different iteration identifier when the check in has completed where say the checked out iteration identifier is 3, the checked in iteration

identifier by default would be 4, but could be set to that or a number greater than 4. The same applies for a version identifier with applicable commands. Accessors are generated for this private field.

The `AbstractVaultCommand` class defines the notion of work-in-progress for the operations check out, check in and undo check out on persistent, iterated types of entities. This abstract class specializes the `AbstractIterationEntityCommand` with the following key APIs:

- `comment` field: the note that is specified by a user when performing either a check out and/or check in. This note is not applicable for an undo check out. Accessors are generated for this private field.

Repository Commands

The `AbstractRepositoryCommand` class provides an abstract representation of a command that can act upon a repository in the system. A repository is the notion of a container of objects that can be a database, flat file, file system, etc. This abstract class specializes the notion of a general command for a repository. From this abstract class it's expected that all repositories can be acted upon in both common and specialized commands. For instance, a query could be thought of as a general inquiry into a repository to "find" objects. The objects themselves could live in a database or file system. It would be up to the specific command delegate to realize the implementation. Additionally, as part of doing a general inquiry there could exist the notion of server-side paging where only a single page, size specified by the user, is returned to the client. Then the client could iterate through pages as applicable, and ultimately close the paging session when done. This abstract class provides the following key APIs:

- `repositorySelector` field: the repository to use to perform a repository command against. Accessors are generated for this private field.
- `resultContainer` field: the container of type instances that are a result of a command being executed against a repository. Accessors are generated for this private field.
- `resultSession` field: the paging session, if one has been established. A paging session represents the concept of iterating over a collection of items in a repository. A paging session allows for fetching results from the `resultSession` over multiple command executions. Accessors are generated for this private field.

The `AbstractIterationHistoryCommand` class specializes the `AbstractRepositoryCommand` to enable definition of specific commands that collect historical iterated/versioned information, like an iteration or version history. This abstract class provides the following key APIs:

- `source` field: the `TypeInstance` representing the point at which to start at in the iterated/versioned history. Accessors are generated for this private field.
- The `AbstractIterationHistoryDeletionCommand` class specializes the `AbstractIterationHistoryCommand` to define commands that perform

iteration/version history deletions (e.g., rollup or rollback of iterations). This abstract class provides the following key APIs.

- target field: the TypeInstance representing the point at which to end at in the iterated/versioned history. Accessors are generated for this private field.

Modeling Guidelines

PTC development practices use Rational Rose to model and develop command beans. Although this is the practice at PTC it is not required.

All types of command beans are modeled in Common subsystems such that their deployment is accessible to both Client and Server subsystems. This is true for server commands since they are currently remotely executable. PTC developed command bean classes not have any compile or run-time dependencies on Persistable classes.

Command Infrastructure Specification – Common Subsystem shows The Command Infrastructure Specification - Common subsystem where many of the command beans exist shown in the Command Bean Overview Class Diagram. Other command beans more specific to Windchill services and enterprise objects exist in similarly placed Common subsystems within the Windchill Foundation Common and Enterprise Common subsystems.

Naming Guidelines

All command beans should be named in a manner to explicitly depict the operation and on what they are designed for. That is to say the naming formula verb + noun + “Command” should be applied. For example, if CRUD commands didn’t already exist and were to be developed, they would apply only to persistent entities and would be named as follows:

- CreatePersistentEntityCommand
- RetrievePersistentEntityCommand
- UpdatePersistentEntityCommand
- DeletePersistentEntityCommand

And this is exactly how these pre-existing CRUD commands are named. Declaring command bean names in this manner makes it possible for consumers to readily know what the command does and on what. Since there is no other API on command beans that clarify their meaning, the name is required to do so.

Serialization Guidelines

All server types of command beans are modeled and generated as Serializable types of classes. This is true since server commands are remotely executable. Once server commands are no longer remotely executable they would no longer be required to be Serializable. But since they are currently remotely executable

they must be streamed from the client to the server and returned back to the client via Java RMI.

Implementation Guidelines

All local types of commands should extend from the `AbstractLocalCommand` class. All server types of commands should extend from the `AbstractServerCommand` class. This will enable inheritance of common functionality that will ensure correct behavior. Implementing `LocalCommand` or `ServerCommand` without extending the respective abstract class is not recommended and is not part of PTC development practices.

Javadoc Guidelines

Command beans represent the set of APIs that are to be depended on and used by consumers. What this means is that the command beans as a whole are the formal, type-aware APIs of the system. Because of this all of the public and protected command beans' APIs should be clearly and thoroughly documented.

Command Delegates

Design Overview

Command delegates are the realization or implementation of command beans. Command beans are the formal API and command delegates are the implementation details. Command delegates are a customization point where one-to-many command delegates can implement [most likely] one [and not very likely many] command bean(s).

For example, the `CreatePersistentEntityCommand` bean has an OOTB default command delegate being the `CreatePersistentEntityCommandDelegate`. In most cases this command delegate will function as required. However, in the case of some change objects they are required to be created in the repository related to another applicable change object. Thus, based on the `CreatePersistentEntityCommand` bean and type of change object a unique command delegate can be looked up and executed as the correct implementation.

Command Delegate Overview Class Diagram shows the key part of the overall command delegate classification hierarchy.

External Interface

The CommandDelegate interface provides the main abstraction of a command delegate. All delegates that realize commands as their implementation will either directly or indirectly assert that they implement this interface. The CommandDelegate interface mandates that the following three methods be implemented and executed in that order:

- preprocess()
- execute()
- postprocess()

First, the preprocess() method enables preconditions and validation to be performed before the actual command's execution. This method is enforced to be invoked via the execute() method's implementation. Second, the execute() method is the implementation establisher that enforces pre and post execution processing based it being implemented to do so. Third, the postprocess() method enables postconditions and postprocessing to be performed after the actual command's execution. This method is enforced to be invoked via the execute() method's implementation.

The AbstractCommandDelegate class provides the initial implementation for all command delegates. It sets forth strict adherence to the command execution algorithm for first preprocessing, second execution and third postprocessing as shown in the following code snippets.

```
public final Command execute( Command cmd )
    throws CommandException {
    cmd = preprocess( cmd );
    cmd = doExecution( cmd ); // Template method.
    cmd = postprocess( cmd );
    return cmd;
}

public final Command preprocess( Command cmd )
    throws CommandException {
    initialize( cmd );
    cmd = doPreprocessing( cmd ); // Template method.
    return cmd;
}

public final Command postprocess( Command cmd )
    throws CommandException {
    cmd = doPostprocessing( cmd ); // Template method.
    finalize( cmd );
    return cmd;
}
```

It achieves this by making final the preprocess(), execute() and postprocess() methods that invoke the doPreprocessing(), doExecution() and doPostprocessing() template methods. The doExecution() template method is the execution implementation of the command. The other two template methods are customization points for special pre-or-post-processing needs.

Note: Note: the initialize() and finalize() methods are overridable. But caution must be used when doing so. Since these methods implement detailed infrastructural logic for all commands it's not recommended to override them.

The CommandDelegateForwarder is the mechanism that enables a command to be executed on the server from the client. Based on what type of CommandDelegate is returned from the CommandDelegateFactory, if AbstractLocalCommandDelegate is *not* a superclass of the found delegate, then it is wrapped with this class to enforce server-side execution. Thus, this class acts as a forwarder that wraps the target CommandDelegate.

The AbstractLocalCommandDelegate class is subclassed from only for command delegates that can be employed in either a client or server local JVM. This class should not be extended from command delegates that should only be executed on the server.

The AbstractRemoteCommandDelegate class is subclassed from only command delegates that can be employed in the server JVM. This class should not be extended by command delegates that can be executed on either the client or server. This abstract class provides some useful inheritable methods as follows:

- populatePersistable() method: given a TypeInstance, this method refreshes the corresponding Persistable, compares update counts to make sure the TypeInstance is not stale, then translates the TypeInstance onto this retrieved Persistable in order to have a fully populated Persistable which contains any changes made in the TypeInstance. This is meant as a convenience method. In the cases where these three steps cannot be done consecutively, see ServerCommandDelegateUtility for the refresh(...), checkUpdateCount(...) and translate(...) methods which achieve the same results.
- populateTypeInstance() method: given a Persistable, TypeInstance and filter, this method returns a TypeInstance representing the Persistable populated governed by the filter. This is achieved via surface attribute translation, and potentially calculation of server attributes and a query of all other non-surface attributes.
- populateMissingContent() method: given a TypeInstance and filter, this method will populate any specified attributes that don't already exist in the TypeInstance by performing a query for those missing attributes and merge them into the given TypeInstance.
- populateOtherContent() method: given a command and a TypeInstance, this method will
 - Populate server calculated attributes in the TypeInstance.

- Populate constraints in the TypeInstance.
- Populate descriptors in the TypeInstance.

This method is used during the finalization of postprocessing.

- populateServerFunction() method: given a command, this method will expand the filter with server functions. This is used during the initialization of preprocessing.
- dispatchEvent() method: dispatches the given event to be processed by Windchill services that are listeners to that event.
- dispatchVetoableEvent() method: dispatches the given event that can be vetoed via an exception to be processed by Windchill services that are listeners to that event.
- getService() method: finds the cached Windchill service from the given class instance of the service interface. For example, passing PersistenceManager.class will return the active persistence manager's singleton in the server.

The MacroCommandDelegate class is a unique command delegate entirely devoted to processing a MacroServerCommand or MacroServerTransaction. This delegate is not used by any other command.

Event Processing

As documented in the external interface of the AbstractRemoteCommandDelegate class, it provides APIs to allow the dispatching of any event defined by all Windchill services. Since this is only possible for a subclass of the AbstractRemoteCommandDelegate, then these events will only be dispatched within the server JVM.

For example, in the following code snippet the RetrievePersistentEntityCommandDelegate dispatches either the PREPARE_FOR_MODIFICATION or PREPARE_FOR_VIEW PersistenceManager event depending on what the next operation is in the filter.

```

OperationIdentifier opid = retrieveCmd.getFilter().getNextOperation();
if (opid instanceof UpdateOperationIdentifier && !(opid instanceof CreateOper-
ationIdentifier)) {
    dispatchVetoableEvent(
        PersistenceManager.class,
        new PersistenceManagerEvent(
            (PersistenceManager)getService( PersistenceManager.class ),
            PersistenceManagerEvent.PREPARE_FOR_MODIFICATION, pers ));
}
else if (opid instanceof DisplayOperationIdentifier || opid instanceof SearchOper-
ationIdentifier) {
    dispatchVetoableEvent(
        PersistenceManager.class,

```

```

new PersistenceManagerEvent(
    (PersistenceManager)getService( PersistenceManager.class ),
    PersistenceManagerEvent.PREPARE_FOR_VIEW, pers ));
}

```

Modeling Guidelines

PTC development practices do not strictly establish whether or not command delegates should be modeled using Rational Rose. Some command delegates have been modeled while others have not.

Local types of command delegates are developed in Common subsystems such that their deployment is accessible to both Client and Server subsystems. Server types of command delegates are developed in Server subsystems to limit deployment and accessibility for server-only dependencies.

The Command Infrastructure Implementation – Common and Server Subsystems shows The Command Infrastructure Implementation - Common and Server subsystems where many of the command delegates exist shown in The Command Delegate Overview Class Diagram. Other command delegates more specific to Windchill services and enterprise objects exist in similarly placed Common subsystems within the Foundation Common and Enterprise Common subsystems.

Naming Guidelines

Command delegate naming conventions follow command bean naming conventions with an added “Delegate” at the end of the class name. In most cases where there’s a one-to-one mapping between bean and delegate the command bean’s name + “Delegate” should be used to name the command delegate. However, in cases of one-to-many, many-to-one or many-to-many command bean to command delegate mappings it will be necessary to be either more or less specific in the command delegate’s name. The following 2 examples illustrate the many-to-one and one-to-many mappings:

1. For all query command beans there exists only one mapped command delegate and thus should be name more general. It is named QueryCommandDelegate where the following command beans are mapped to it:
 - AttributeContainerQueryCommand
 - BasicQueryCommand
 - FindPersistentEntityCommand
2. For the CreatePersistentEntityCommand it has a general purpose CreatePersistentEntityCommandDelegate mapped to it. But also there exists the following more specialized delegates for change objects:
 - CreateWTAnalysisActivityCommandDelegate
 - CreateWTChangeActivity2CommandDelegate

- CreateWTChangeInvestigationCommandDelegate
- CreateWTChangeIssueCommandDelegate
- CreateWTChangeOrder2CommandDelegate
- CreateWTChangeProposalCommandDelegate
- CreateWTChangeRequest2CommandDelegate

Serialization Guidelines

No command delegate should ever be Serializable. They are the implementation details of commands where the command beans are sent across-the-wire, not the command delegates.

Implementation Guidelines

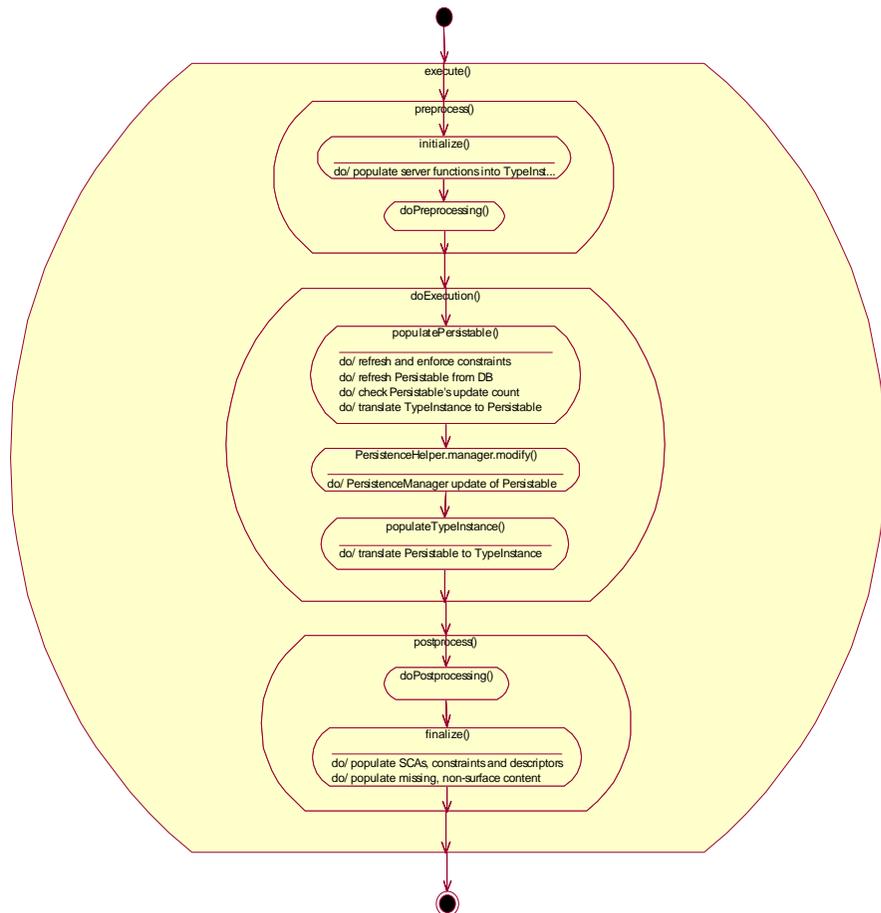
In general, a command delegate's implementation has the 3 main steps within the doExecution() method, assuming that no customization has been done for pre-or-post-processing:

1. Given the input TypeInstance populate a new Persistable. This by-in-large is an act of in-memory translations.
2. Perform the server-side API invocation(s) against the Persistable via Windchill services.
3. Populate the output TypeInstance with the acted upon Persistable. This has 3 minor steps where there's an in-memory translation, derivation of server calculated attributes and a query of all non-surface, missing attributes.

Additionally, the command delegate should be implemented where appropriate levels of debugging and feedback messages are issued.

This type of implementation is known as a "wrapper" command delegate around an existing Windchill service's API matching the command. This isn't always the case. Command delegates can also execute other commands as well. This is the case of the MacroCommandDelegate where it iterates through the sequence of commands and executes each one in turn.

The UpdatePersistentEntityCommandDelegate Activity Diagram shows an activity diagram for the UpdatePersistentEntityCommandDelegate. As can be seen within the doExecution() activity the basic 3 step pattern was applied. The only caveat is that constraints are refreshed and enforced as well. This pattern or slight variation of it has been applied to all "wrapper" command delegates.



UpdatePersistentEntityCommandDelegate Activity Diagram

The following textual description summarizes the doExecution() activity of the Activity diagram.

1. Refresh the constraints that apply to the operation being performed, then enforce those constraints against the type instance. The constraints are those that apply for the command (as denoted by an OperationIdentifier) currently being executed. This represents in-bound processing by Constraint Factories.
2. Refresh the Persistable from the database based on the TypeInstance's TypeInstanceIdentifier and ensure the input data is not based upon a stale object. In other words, check the update count and additionally ensure that an iterated object didn't iterate.

3. Translate all changes from the input `TypeInstance` to the `Persistable`. Only those attributes from the `TypeInstance` that are in a `NEW`, `CHANGE` or `DELETED` state are translated to the `Persistable`.
4. Use the `PersistenceManager` to perform an update of the `Persistable` to the database. For non `CRUD` operations, this is where other legacy APIs would be executed to either update or retrieve `Persistable` objects.
5. Using the filter supplied with the command, prepare the result `TypeInstance`. This processing may consist of only an in-memory translation from `Persistable` to `TypeInstance`. It may also add any server calculated attributes to the `TypeInstance` by deriving them via server functions, based on the filter. It can as well involve an additional data base query to retrieve attributes that are missing, again based on the filter, from the `TypeInstance` that were not either translated or calculated.

But command delegates can be much more involved with their processing logic. An example of this would be where a “deferred save” of an entire product structure’s changes from BOM markup was needing to be done within a transaction. Thus, the product structure represented as a single `TypeInstance` could be sent to the server and processed by a special command delegate that takes apart the `TypeInstance` and builds a corresponding product structure with all of the changes. Then within a single transaction part-by-part, link-by-link would be updated in the repository.

Attribute Handlers

Design Overview

Attribute Handlers are mechanisms for translating one of more attributes from a `Persistable` to `TypeInstance`, or vice versa. They are translating in memory objects only. Database hits should be avoided if at all possible. The order in which attributes will be handled is not guaranteed or deterministic. Therefore, if attributes require a certain order to their handling, a single Attribute Handler should be written that handles both of them, at the same time, whenever any of the related attributes are looked up. This handler then handles all of them immediately, but in the proper order, then marks all of the involved attributes as handled, so that they will not be handled multiple times.

Attribute Handlers generally do not validate arguments to make sure that business rules are being followed. Generally, values passed to the `get(...)` and `set(...)` methods are trusted, and if this results in an exception in a service, or a `ClassCastException`, etc, then these exceptions will be passed through.

See `StandardAttributeHandler` for more information on when a specific `AttributeHandler` must be written, and when the `StandardAttributeHandler` will suffice.

External Interface

The `AbstractAttributeHandler` class is the main abstraction to define attribute handlers. It provides some convenience methods for subclasses to reuse, but most importantly it defines the following key APIs to be implemented by subclasses:

- `get()` method: gets an attribute or attributes and their values from a `Persistable`, and puts them in a `TypeInstance`. The source `Persistable` and target `TypeInstance` are available via the translator command argument.
- `set()` method: sets an attribute or attributes on a `Persistable`, with the value or values from a `TypeInstance`. The source `TypeInstance` and target `Persistable` are available via the translator command argument.

The `StandardAttributeHandler` class is the standard implementation of `AbstractAttributeHandler`. If no specific attribute handler is to be used for a particular attribute, the `StandardAttributeHandler` will attempt to read or write that attribute. In the case of failure, an exception thrown will be wrapped in a `CommandException`.

Attribute handlers are called as part of a translation of a `Persistable` object to or from a `TypeInstance`. For each attribute encountered in the translation, the attribute name is looked up and compared against the entries in attribute handler properties files. If an entry is found, the handler listed in the entry will be invoked. Otherwise, the `StandardAttributeHandler` will be invoked. This means that unless the `StandardAttributeHandler` is sufficient to handle an attribute, the attribute must have its own attribute handler.

The `StandardAttributeHandler` will handle attributes as described in the following cases unless otherwise noted:

1. The attribute is a surface attribute of the `Persistable` being translated.
2. Attributes on cookies, or on an object which itself is a surface attribute are handled.
3. The attribute is a `NonpersistedAttribute`, `NonpersistedAssociation`, `InstanceBasedAttribute`, `InstanceBasedAssociation`, or `ModeledAttribute` but not `ModeledAssociation`.
4. `NonPersistentAttributes`, `NonPersistentAssociations`, `InstanceBasedAttributes`, and `InstanceBasedAssociations` are handled outside the attribute handlers, and the `StandardAttributeHandler` properly ignores them.
5. `ModeledAssociations` are not handled by the `StandardAttributeHandler`.
6. The attribute's type is compatible with the primitive types of content stored in `TypeInstances`, or is an `EnumeratedType`.
7. `TypeInstances` only store certain "primitive" types of content, such as `String`, `TypeInstanceIdentifier`, `Ratio`, `Boolean`, `Timestamp`, etc.

8. Attributes of these types generally must have attribute handlers that convert the values to and from one of these "primitive" forms for storage in the `TypeInstance`.
9. Attributes whose values are of type `EnumeratedType` or a subclass of `EnumeratedType` are handled by the `StandardAttributeHandler`, so long as they meet the other criteria above.

The `AttributeStates` class wraps two `Vectors` used to store `AttributeTypeIdentifiers` in order to manage which have been handled and which have not. Several methods are provided to access the state of attributes, and thus it is important not to access them directly, but to instead use the provided methods as follows:

1. `findUnhandled()` method: given an `AttributeTypeIdentifier`, this method searches through the unhandled attributes for an `AttributeTypeIdentifier` whose external form matches the external form of the supplied `AttributeTypeIdentifier`. If a match is found, the `AttributeTypeIdentifier` found with the unhandled attributes is returned, otherwise null is returned.
2. `findHandled()` method: given an `AttributeTypeIdentifier`, this method searches through the handled attributes for an `AttributeTypeIdentifier` whose external form matches the external form of the supplied `AttributeTypeIdentifier`. If a match is found, the `AttributeTypeIdentifier` found with the handled attributes is returned, otherwise null is returned.
3. `makeHandled()` method: given an `AttributeTypeIdentifier`, this method searches through the handled attributes for that exact `AttributeTypeIdentifier` (same actual object). If a match is found, the `AttributeTypeIdentifier` is moved from the unhandled to the handled attributes, and true is returned, otherwise false is returned.

Modeling Guidelines

PTC development practices do not strictly establish whether or not attribute handlers should be modeled using Rational Rose. Only the `StandardAttributeHandler` has been modeled while all others have not.

Since attribute handlers are purely implementation and no direct invocation is made to them they should be packaged ...server.impl packages.

Naming Guidelines

Attribute handlers should be named to show the attribute being handler + "AttributeHandler." Some examples of existing attribute handlers follow:

- `ContentItemRoleAttributeHandler`
- `DataFormatReferenceAttributeHandler`
- `ContentItemCreatedByAttributeHandler`
- `PersistInfoCreateStampAttributeHandler`

- PersistInfoModifyStampAttributeHandler
- CabinetReferenceAttributeHandler
- FolderReferenceAttributeHandler
- ProjectReferenceAttributeHandler
- IterationInfoCreatorAttributeHandler
- MasterReferenceAttributeHandler
- CheckoutInfoStateAttributeHandler

Serialization Guidelines

No attribute handler should ever be Serializable. They are the implementation details of in-memory translations initiated by command delegates.

Implementation Guidelines

Attribute handlers can be implemented to handle one to many related attributes. In the case of an attribute handler only handling one attribute the CheckoutInfoStateAttributeHandler is shown to illustrate how it can be implemented in the following code snippet:

```
public void get(
    AttributeTypeIdentifier attr_type_id,
    AbstractTranslatorCommand transl_cmd, AttributeStates
    attr_states )
    throws CommandException {
    if (attr_states.findHandled(attr_type_id) != null)
        return;
    Workable work = (Workable) transl_cmd.getPersistable();
    CheckoutInfo ci = work.getCheckoutInfo();
    WorkInProgressState state = null;
    if (ci != null)
        state = ci.getState();
    String value = null;
    if (state != null)
        value = state.toString();
    putAttributeContent( attr_type_id, transl_cmd, attr_states,
    value );
    attr_states.makeHandled( attr_type_id );
}

public void set(
    AttributeTypeIdentifier attr_type_id,
    AbstractTranslatorCommand transl_cmd, AttributeStates
    attr_states )
    throws CommandException {
    if (attr_states.findHandled(attr_type_id) != null)
        return;
    Workable work = (Workable) transl_cmd.getPersistable();
    String value = (String) getAttributeContent( attr_type_id,
    transl_cmd );
    CheckoutInfo ci = work.getCheckoutInfo();
```

```

        WorkInProgressState state =
WorkInProgressState.toWorkInProgressState( value );
        try {
            if (ci == null) {
                ci = CheckoutInfo.newCheckoutInfo();
                if (DEBUG) getDebugLog(transl_cmd).report("Setting
checkoutInfo = "+ci);
                work.setCheckoutInfo( ci );
            }
            ci.setState( state );
        }
        catch (Exception e) {
            throw new CommandException( e );
        }
        attr_states.makeHandled( attr_type_id );
    }
}

```

For attribute handlers that are required to deal with multiple, related attributes at the same time the VersionInfoAttributeHandler is shown to illustrate how it can be implemented in the following code snippet:

```

public void get(
    AttributeTypeIdentifier attr_type_id,
    AbstractTranslatorCommand transl_cmd, AttributeStates
attr_states )
    throws CommandException {
    AttributeTypeIdentifier versionIdATI =
        getAttributeTypeIdentifier(
"versionInfo.identifier.versionId", transl_cmd );
    AttributeTypeIdentifier versionLevelATI =
        getAttributeTypeIdentifier(
"versionInfo.identifier.versionLevel", transl_cmd );
    AttributeTypeIdentifier versionQualIdATI =
        getAttributeTypeIdentifier(
"versionInfo.lineage.series.value", transl_cmd );
    AttributeTypeIdentifier versionQualLevelATI =
        getAttributeTypeIdentifier(
"versionInfo.lineage.series.level", transl_cmd );
    AttributeTypeIdentifier[] ATIs = {
        versionIdATI, versionLevelATI, versionQualIdATI,
versionQualLevelATI };
    if (areHandled(ATIs, attr_states))
        return;
    Versioned version = (Versioned) transl_cmd.getPersistable();
    try {
        VersionInfo vInfo = version.getVersionInfo();
        VersionIdentifier vIdentifier = null;
        QualifiedIdentifier qIdentifier = null;
        MultilevelSeries mSeries = null;
        HarvardSeries hSeries = null;
        Integer vLevel = null;
        Integer qLevel = null;
        String versionId = null;
        Long versionLevel = null;
        String qualifiedId = null;
        Long qualifiedLevel = null;
        if (vInfo != null) {

```

```

        vIdentifier = vInfo.getIdentifier();
        if (vIdentifier != null) {
            mSeries = vIdentifier.getSeries();
            if (mSeries != null) {
                versionId = mSeries.getValue();
                vLevel = mSeries.getLevel();
                if (vLevel != null) {
                    versionLevel = new Long( vLevel.longValue() );
                }
            }
        }
        qIdentifier = vInfo.getLineage();
        if (qIdentifier != null) {
            hSeries = qIdentifier.getSeries();
            if (hSeries != null) {
                qualifiedId = hSeries.getValue();
                qLevel = hSeries.getLevel();
                if (qLevel != null) {
                    qualifiedLevel = new Long( qLevel.longValue() );
                }
            }
        }
    }
    putAttributeContent( versionIdATI, transl_cmd, attr_states,
versionId );
    putAttributeContent( versionLevelATI, transl_cmd,
attr_states, versionLevel );
    putAttributeContent( versionQualIdATI, transl_cmd,
attr_states, qualifiedId );
    putAttributeContent( versionQualLevelATI, transl_cmd,
attr_states, qualifiedLevel );
}
catch (VersionControlException vce) {
    throw new CommandException( vce );
}
attr_states.makeHandled( ATIs );
}

public void set(
    AttributeTypeIdentifier attr_type_id,
    AbstractTranslatorCommand transl_cmd, AttributeStates
attr_states )
    throws CommandException {
    AttributeTypeIdentifier versionIdATI =
        getAttributeTypeIdentifier(
"versionInfo.identifier.versionId", transl_cmd );
    AttributeTypeIdentifier versionLevelATI =
        getAttributeTypeIdentifier(
"versionInfo.identifier.versionLevel", transl_cmd );
    AttributeTypeIdentifier versionQualIdATI =
        getAttributeTypeIdentifier(
"versionInfo.lineage.series.value", transl_cmd );
    AttributeTypeIdentifier versionQualLevelATI =
        getAttributeTypeIdentifier(
"versionInfo.lineage.series.level", transl_cmd );
    AttributeTypeIdentifier[] ATIs = {
        versionIdATI, versionLevelATI, versionQualIdATI,

```

```

versionQualLevelATI };
    if (areHandled(ATIs, attr_states))
        return;
    Versioned version = (Versioned) transl_cmd.getPersistable();
    String versionId = (String) getAttributeContent( versionIdATI,
transl_cmd );
    Long versionLevel = (Long) getAttributeContent( versionLevelATI,
transl_cmd );
    try {
        MultilevelSeries series =
MultilevelSeries.newMultilevelSeries(
            VersionIdentifier.class.getName(), versionId, new Integer(
versionLevel.intValue() ));
        if (version.getVersionInfo() == null) {
            version.setVersionInfo( VersionInfo.newVersionInfo() );
        }
        VersionControlHelper.setVersionIdentifier( version,
VersionIdentifier.newVersionIdentifier( series ));
    }
    catch (Exception e) {
        throw new CommandException( e );
    }
    attr_states.makeHandled( ATIs );
}

```

For attribute handlers that deal with some form of object reference and server calculated attributes the ViewReferenceAttributeHandler is shown to illustrate how it can be implemented in the following code snippet:

```

public void get(
    AttributeTypeIdentifier attr_type_id,
    AbstractTranslatorCommand transl_cmd, AttributeStates
attr_states )
    throws CommandException {
    AttributeTypeIdentifier viewATI =
getAttributeTypeIdentifier("view.id", transl_cmd);
    if (attr_states.findHandled(viewATI) != null)
        return;
    ViewManageable vm = (ViewManageable)
transl_cmd.getPersistable();
    ViewReference viewRef = vm.getView();
    TypeInstanceIdentifier viewTii = null;
    if (viewRef != null)
        viewTii = TypeIdentifierUtility.getTypeInstanceIdentifier(
viewRef );
    putAttributeContent( viewATI, transl_cmd, attr_states, viewTii
);
    attr_states.makeHandled( viewATI );
}

public void set(
    AttributeTypeIdentifier attr_type_id,
    AbstractTranslatorCommand transl_cmd, AttributeStates
attr_states )
    throws CommandException {
    AttributeTypeIdentifier viewATI =
getAttributeTypeIdentifier("view.id", transl_cmd);

```

```

        AttributeTypeIdentifier viewNameATI =
getAttributeTypeIdentifier("view", transl_cmd);
        AttributeTypeIdentifier[] ATIs = { viewATI, viewNameATI };
        if (areHandled(ATIs, attr_states))
            return;
        ViewManageable vm = (ViewManageable)
transl_cmd.getPersistable();
        TypeInstanceIdentifier viewTii =
            (TypeInstanceIdentifier) getAttributeContent( viewATI,
transl_cmd );
        String viewName =
            (String) getAttributeContent( viewNameATI, transl_cmd );
        try {
            if (viewTii != null) {
                ViewReference viewRef = ViewReference.newViewReference(
(ObjectIdentifier)TypeIdentifierUtility.getObjectReference(
viewTii ).getKey() );
                vm.setView( viewRef );
            }
            else if (viewName != null ) {
                ViewReference viewRef = ViewReference.newViewReference(
(ObjectIdentifier)TypeIdentifierUtility.getObjectReference(
ServerCommandUtility.getViewTypeInstanceIdentifier(viewName)).getKey() );
                vm.setView( viewRef );
            }
            else {
                vm.setView( null );
            }
        }
        catch (Exception e) {
            throw new CommandException( e );
        }
        attr_states.makeHandled( ATIs );
    }

```

Customizing Command Delegates

Command delegates are the realization or implementation of command beans. Because of this the command delegate is a customization point where there exists a mapping from one-to-many command bean(s) to one-to-many command delegate(s). This mapping is achieved with, but not limited to, the property files shown below:

- `com.ptc.core.command.server.delegate.ServerCommandDelegate.properties`
- `com.ptc.core.foundation.FoundationServerCommandDelegate.properties`
- `com.ptc.core.foundation.windchill.enterprise.EnterpriseServerCommandDelegate.properties`

The most common case is a one-to-one mapping from command bean to command delegate. But there do exist some one-to-many and many-to-one mappings as well. These types of mappings are accomplished by specifying the selector, requestor, priority triad selector in the property entry. The selector is typically the fully qualified command bean's class name. The requestor is typically either "null" or the external form of the type of object being represented by a `TypeInstance` (e.g., `WCTYPE|wt.part.WTPart`). The "null" represents any type of object.

The selector and requestor are derived during run-time by the methods `getSelectorTypename()` and `getRequestor()`, respectively, as implemented or inherited in the command bean being executed. The default implementation of `getSelectorTypename()` returns this command bean's fully qualified class name. The default implementation of `getRequestor()` returns null. To customize these default implementations would require either a change to an existing command bean's implementation, or a new command bean that overrode the implementation of these methods.

The example below shows a one-to-one mapping of the `RetrievePersistentEntityCommand` that can be found in the `ServerCommandDelegate.properties` file where the selector, requestor, priority triad are underlined:

- `wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.RetrievePersistentEntityCommand/null/0=com.ptc.core.command.server.delegate.entity.RetrievePersistentEntityCommandDelegate/singleton`

The examples below show a one-to-many mapping of the `CreatePersistentEntityCommand` to many command delegates that can be found in the `ServerCommandDelegate.properties` and `EnterpriseServerCommandDelegate.properties` files:

- `wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/null/0=com.ptc.core.command.server.delegate.entity.CreatePersistentEntityCommandDelegate/singleton`

- wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/WCTYPE|
wt.change2.WTAnalysisActivity/0=com.ptc.windchill.enterprise.change2.server.CreateWTAnalysisActivityCommandDelegate/singleton
- wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/WCTYPE|
wt.change2.WTChangeActivity2/0=com.ptc.windchill.enterprise.change2.server.CreateWTChangeActivity2CommandDelegate/singleton
- wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/WCTYPE|
wt.change2.WTChangeInvestigation/0=com.ptc.windchill.enterprise.change2.server.CreateWTChangeInvestigationCommandDelegate/singleton
- wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/WCTYPE|
wt.change2.WTChangeIssue/0=com.ptc.windchill.enterprise.change2.server.CreateWTChangeIssueCommandDelegate/singleton
- wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/WCTYPE|
wt.change2.WTChangeOrder2/0=com.ptc.windchill.enterprise.change2.server.CreateWTChangeOrder2CommandDelegate/singleton
- wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/WCTYPE|
wt.change2.WTChangeProposal/0=com.ptc.windchill.enterprise.change2.server.CreateWTChangeProposalCommandDelegate/singleton
- wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/com.ptc.core.command.common.bean.entity.CreatePersistentEntityCommand/WCTYPE|
wt.change2.WTChangeRequest2/0=com.ptc.windchill.enterprise.change2.server.CreateWTChangeRequest2CommandDelegate/singleton

The example below shows a many-to-one mapping of the different query commands that are a type of `AbstractQueryCommand` to the one `QueryCommandDelegate` that can be found in the `ServerCommandDelegate.properties` file:

- `wt.services/svc/default/com.ptc.core.command.common.CommandDelegate/DEFAULT/com.ptc.core.query.command.common.AbstractQueryCommand/0=com.ptc.core.query.command.server.QueryCommandDelegate/singleton`

The interesting aspect of this example are that the selector is nothing specific, in particular not a command bean's fully qualified class name, which is what the requestor is. The reason for this is the algorithm for lookup is a hierarchical class-based search on the requestor. The selector is merely text. Thus, to map one command delegate to many command beans the lookup must find a common ancestor for each command bean that is to be mapped to the command delegate.

Further command delegate customization can be accomplished by overriding the command delegate's inherited implementation of the template methods `doPreprocessing()` and `doPostprocessing()`. Overriding `doPreprocessing()` allows the command delegate to pre-process the command before it's actually executed. Overriding `doPostprocessing()` allows the command delegate to post-process the command after it has finished being executed. See also *Chapter 4: The Command Layer* in the *Windchill Application Developer's Guide* for more information on pre-and-post-processing.

An example below illustrates how the `MacroCommandDelegate` overrides the pre-processing of a command in order to validate the sequence of commands, and then parse the segments tagged by `BeginServerTransaction` and `EndServerTransaction` each into a transaction block. If the transactional tagging is not well formed then an appropriate exception is thrown.

```
protected Command doPreprocessing( Command cmd )
    throws CommandException {
    CommandDelegateUtility.validateCommand(cmd,
MacroServerCommand.class);
    super.doPreprocessing( cmd );
    int macroCmdIndex = 0;
    MacroServerCommand macroCmd = (MacroServerCommand)cmd;
    TransactionalBlock trxBlock = new TransactionalBlock();
    setTransactionalBlockList( new Vector() );
    while (macroCmdIndex < macroCmd.getSequence().length) {
        macroCmdIndex = parseCommandSequence( macroCmd.getSequence(),
macroCmdIndex, trxBlock );
        getTransactionalBlockList().add( trxBlock );
    }
    return cmd;
}
private int parseCommandSequence( ServerCommand[] cmdList, int
cmdListIndex, TransactionalBlock trxBlock )
    throws CommandException {
    ServerCommand cmd = null;
    try {
        cmd = (ServerCommand)cmdList[cmdListIndex];
        cmdListIndex++;
    }
```

```

    }
    catch (ArrayIndexOutOfBoundsException ob) {
        throw new InvalidTransactionTaggingException( ob );
    }
    if (cmd instanceof BeginServerTransaction) {
        if (trxBlock.getCommandList() == null) {
            trxBlock.setGuaranteed( true );
            trxBlock.setCommandList( new Vector() );
        }
        cmdListIndex = parseCommandSequence( cmdList, cmdListIndex,
trxBlock );
    }
    else if (cmd instanceof EndServerTransaction) {
        if (trxBlock.getCommandList() == null) {
            throw new InvalidTransactionTaggingException();
        }
        else if (trxBlock.getCommandList() != null &&
trxBlock.getCommandList().size() == 0) {
            throw new EmptyTransactionTaggingException();
        }
        else {
            return cmdListIndex;
        }
    }
    else {
        if (trxBlock.getCommandList() == null) {
            trxBlock.setGuaranteed( false );
            trxBlock.setCommandList( new Vector() );
        }
        trxBlock.getCommandList().add( cmd );
        if (cmdListIndex < cmdList.length) {
            cmdListIndex = parseCommandSequence( cmdList, cmdListIndex,
trxBlock );
        }
    }
    return cmdListIndex;
}

```

Customizing Attribute Handlers

Attribute handlers are the mechanisms for translating one or more attributes from a Persistable to TypeInstance, or vice versa. They represent the micro customization point where command delegates are the macro customization point. The selector, requestor, priority selector triad is used in a similar manner as with the command delegates where mapping is achieved with, but not limited to, the property files shown below:

- com.ptc.core.foundation.FoundationAttributeHandler.properties
- com.ptc.windchill.enterprise.EnterpriseAttributeHandler.properties

The selector is derived from the attribute's AttributeTypeIdentifier where only the name of the attribute is used. The requestor is either "null" or the external form of the type of object being represented by a TypeInstance (e.g., WCTYPE|wt.part.WTPart).

Typically attribute handlers are only necessary when dealing with attributes that are a form of reference to an associated attribute on another object, or when there are nonexistent public accessors for the attribute. The `StandardAttributeHandler` should handle all other cases. However, if special set/get processing is required for an attribute an existing or new attribute handler could be implemented with this special logic to handle the attribute. See also *Chapter 4: The Command Layer* in the *Windchill Application Developer's Guide* for more information on and examples of attribute handlers.

The examples below show a one-to-one mapping of an attribute to its attribute handler that can be found in the `FoundationAttributeHandler.properties` file where the selector, requestor, priority triad are underlined:

- `wt.services/svc/default/com.ptc.core.command.server.delegate.io.AbstractAttributeHandler/ownership.owner/null/0=com.ptc.core.foundation.ownership.server.impl.OwnershipOwnerReferenceAttributeHandler/singleton`
- `wt.services/svc/default/com.ptc.core.command.server.delegate.io.AbstractAttributeHandler/format/WCTYPE
wt.content.FormatContentHolder/0=com.ptc.core.foundation.content.server.impl.DataFormatReferenceAttributeHandler/singleton`
- `wt.services/svc/default/com.ptc.core.command.server.delegate.io.AbstractAttributeHandler/format/WCTYPE
wt.content.ContentItem/0=com.ptc.core.foundation.content.server.impl.ContentItemFormatAttributeHandler/singleton`

The examples below show a many-to-one mapping from two attributes to the one `ViewReferenceAttributeHandler` that can be found in the `FoundationAttributeHandler.properties` file:

- `wt.services/svc/default/com.ptc.core.command.server.delegate.io.AbstractAttributeHandler/view/null/0=com.ptc.core.foundation.vc.views.server.impl.ViewReferenceAttributeHandler/singleton`
- `wt.services/svc/default/com.ptc.core.command.server.delegate.io.AbstractAttributeHandler/view.displayIdentifier/null/0=com.ptc.core.foundation.vc.views.server.impl.ViewReferenceAttributeHandler/singleton`

5

The Enterprise Layer

This chapter describes the classes and interfaces available in four packages:

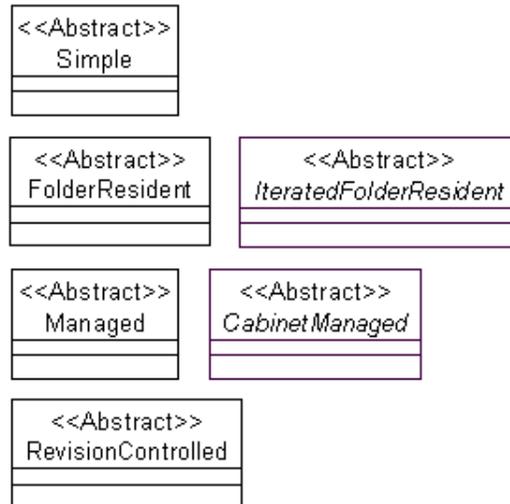
- wt.enterprise
- wt.doc
- wt.part
- wt.change2

The classes provided in these packages were designed and intended for you to extend as you customize Windchill for your own use.

Topic	Page
Enterprise Abstractions	5-2
Document Abstractions	5-10
Part Abstractions	5-13
Change Abstractions.....	5-20

Enterprise Abstractions

The wt.enterprise package provides the basic business objects used in the Windchill system. Most business classes you construct will be extended from the enterprise classes or their subclasses.



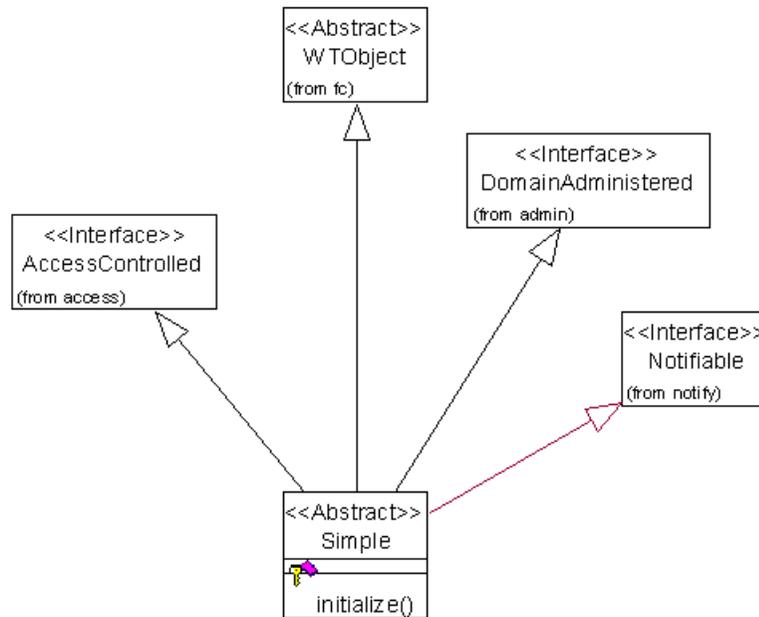
The enterprise Package

Business classes should be extended from one of the abstract classes included in the wt.enterprise package to take advantage of the capabilities and services they offer. This diagram shows convenience classes designed to consolidate a basic set of features that various business classes might require. Most of the business classes in your model are expected to extend one of these classes and simplify your implementations.

Simple Business Class

A simple business class is a first class information object that is subject to business rules, such as access control. These objects have no managed life cycle and are not organized into folders. Because these objects are not visible via

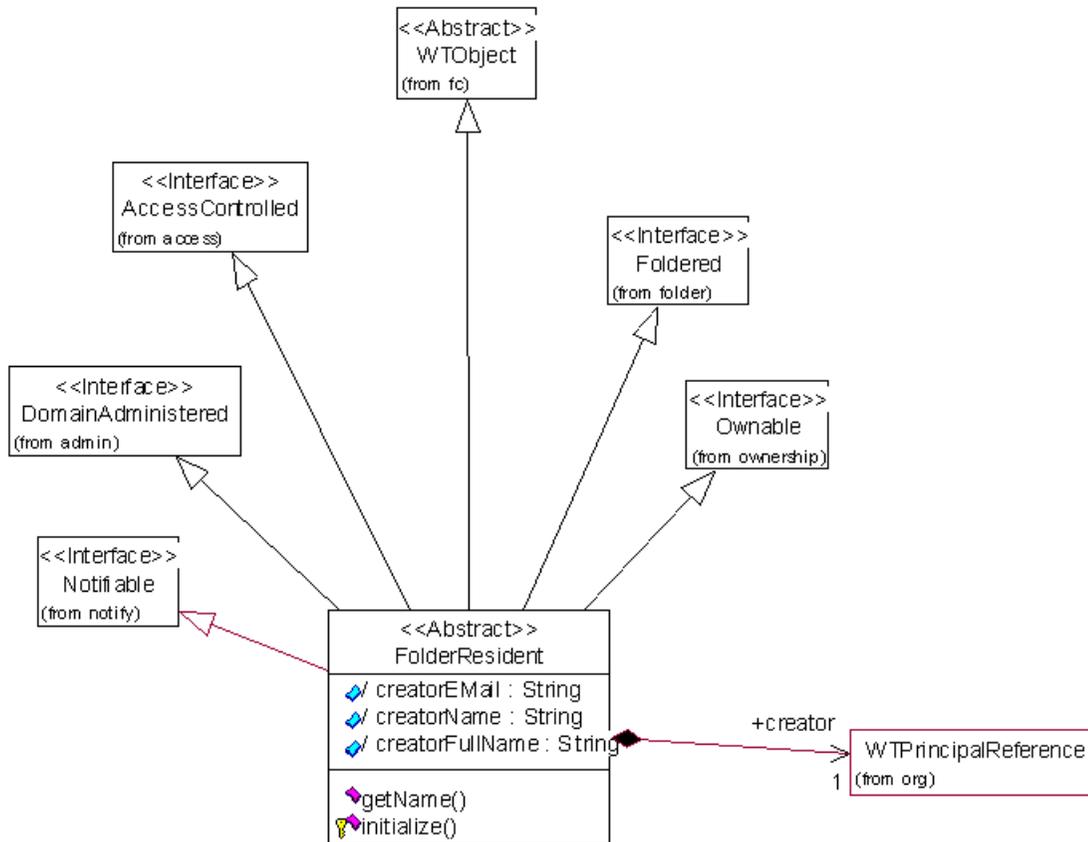
folders, they tend to be more administrative in nature (that is, created by administrators but referenced by end users).



Simple Business Class

Folder Resident Business Class

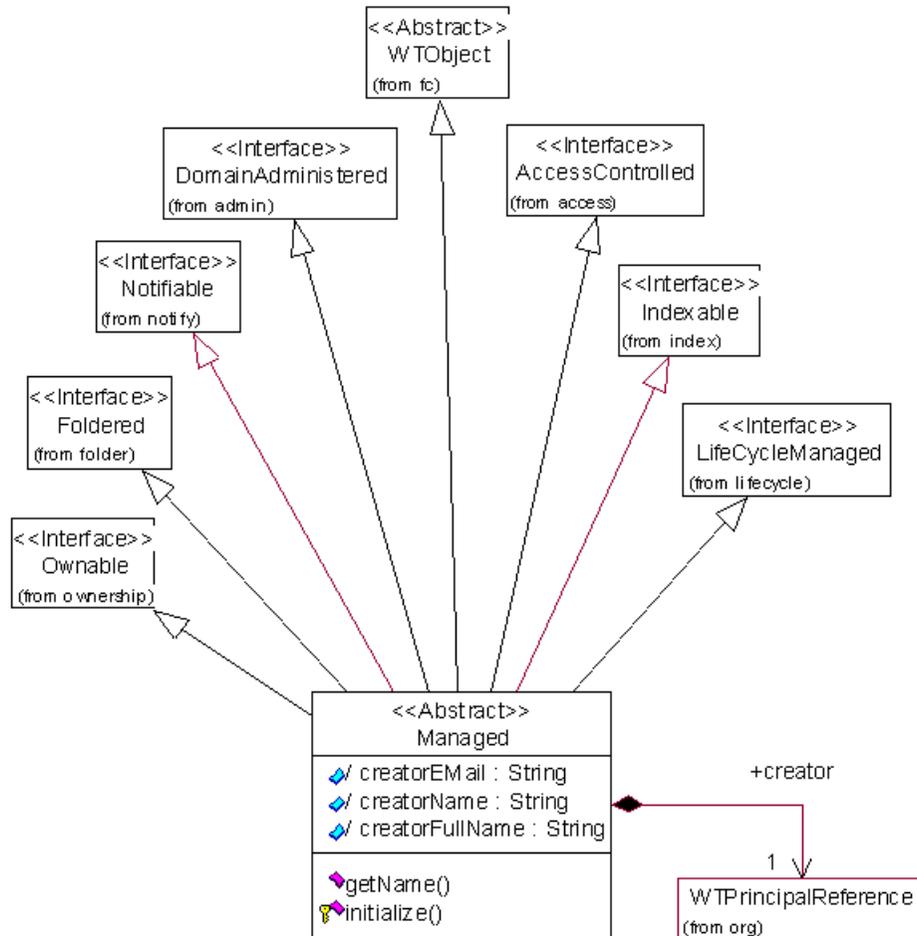
A folder resident business class is a business object that resides in a folder. Because these objects are accessible in folders, they are visible to users in the Windchill Explorer. These objects are subject to access control rules. FolderResident business objects are not subject to life cycle management. Therefore, they are more administrative in nature. All FolderResident objects automatically record the principal (user) who created them.



Folder Resident Business Class

Managed Business Class

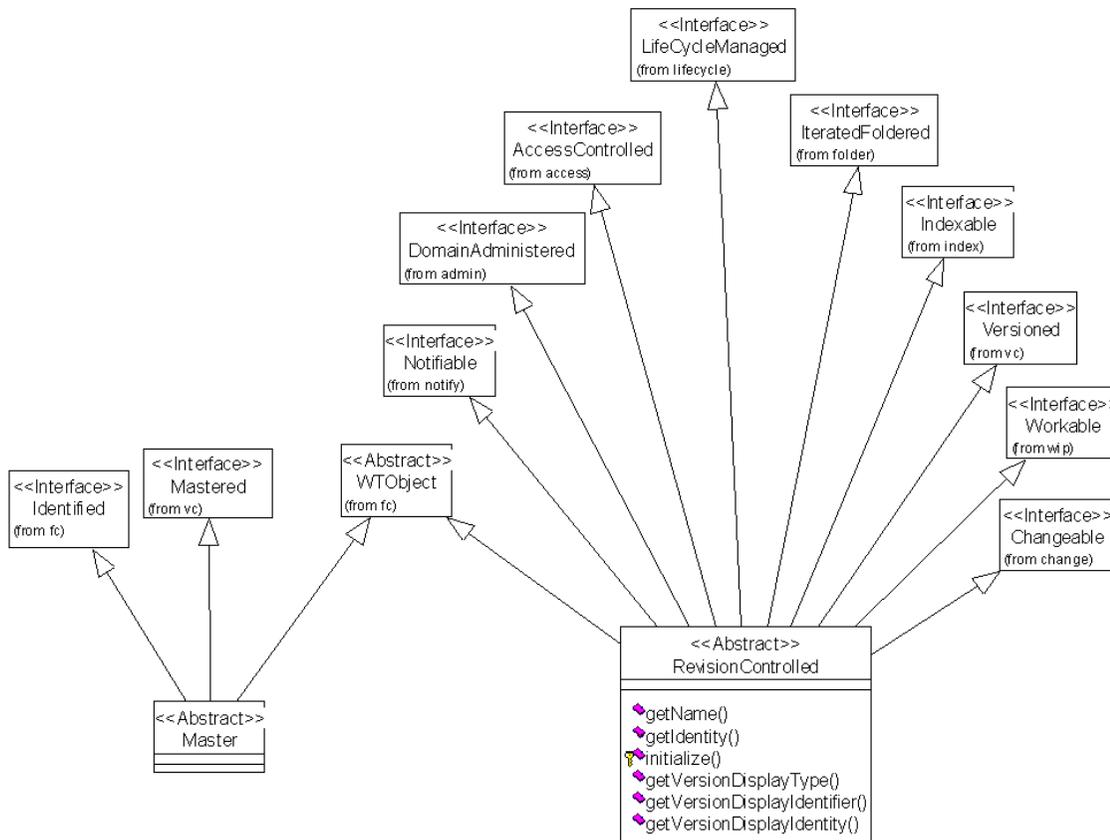
Managed business class objects are subject to a controlled life cycle. They reside in folders for organizational purposes. They are non-revisable objects that the system manages and controls via access control. These objects record the principal (user) who created them. The creator can be used as a role in certain processing for life cycle and workflow operations.



Managed Business Class

Revision Controlled Business Class

RevisionControlled business objects are identified by a revision identifier. They are managed and changed via a checkin/checkout mechanism. They are subject to life cycle management and other forms of management by the enterprise. They have a creator by virtue of being a Workable object (which is an Iterated object). Because the class is also Versioned, there can be multiple business versions of the Master object, such as revision A and revision B of a single document.



Revision Controlled Business Class

Revision controlled business objects are managed as two separate classes:

Master

Represents the version independent idea of a business concept. It contains the information that identifies the business objects. Typically this is information such as a name and number that remains constant for all versions (or, more accurately, is the same for all versions).

Revision controlled

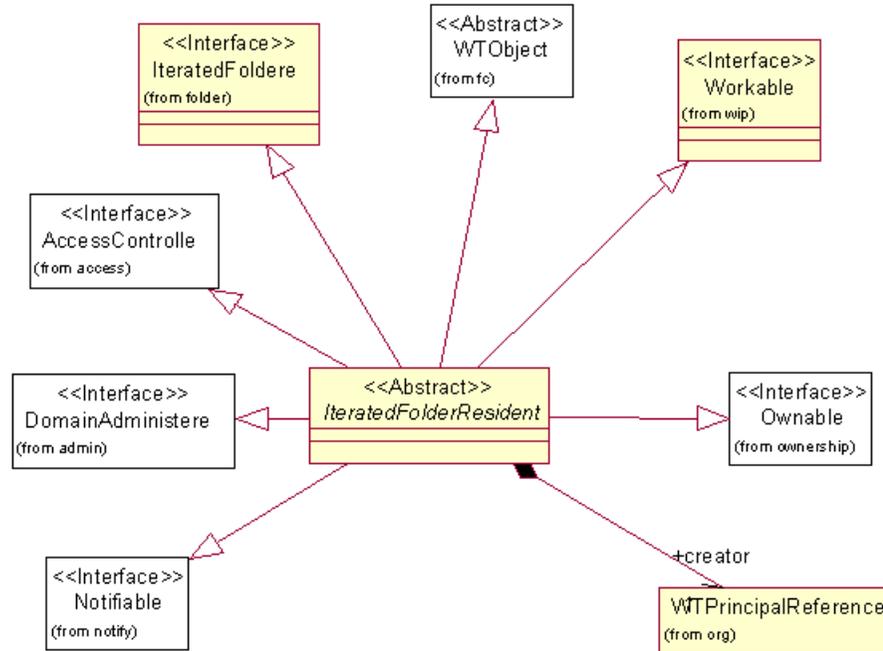
Represents the successive changes to the business objects as it is developed over time. A RevisionControlled object represents an official version (for example, revision A or B) but also provides access to previous iterations of

that version. The previous iterations are considered a history of work-in-progress activity (checkouts and checkins) for the object.

Iterated Folder Resident Business Class

IteratedFolderResident business class objects reside in folders where they are visible to users of the Windchill Explorer. Users create new iterations of these objects using checkout/checkin operations. They are also subject to access control and automatically record the principal (user) who created them.

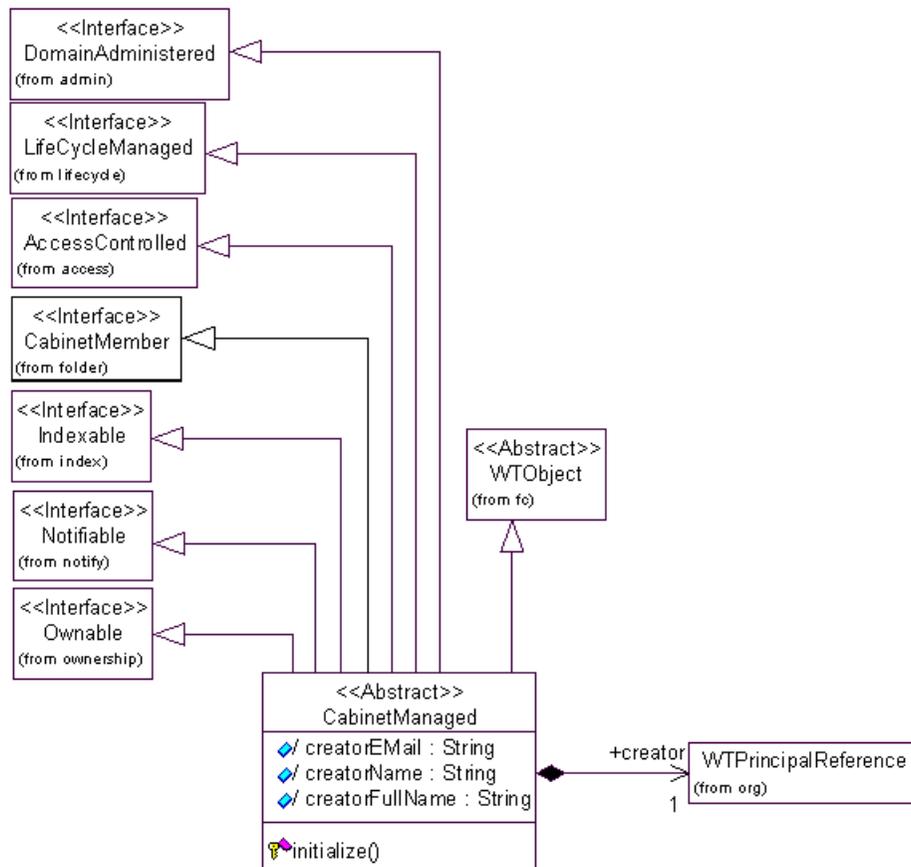
IteratedFolderResident objects are similar to RevisionControlled objects. However, they are lighter weight objects in that they are neither versioned nor life cycle-managed, as is the case with RevisionControlled objects.



Iterated Folder Resident Business Class

Cabinet Managed Business Class

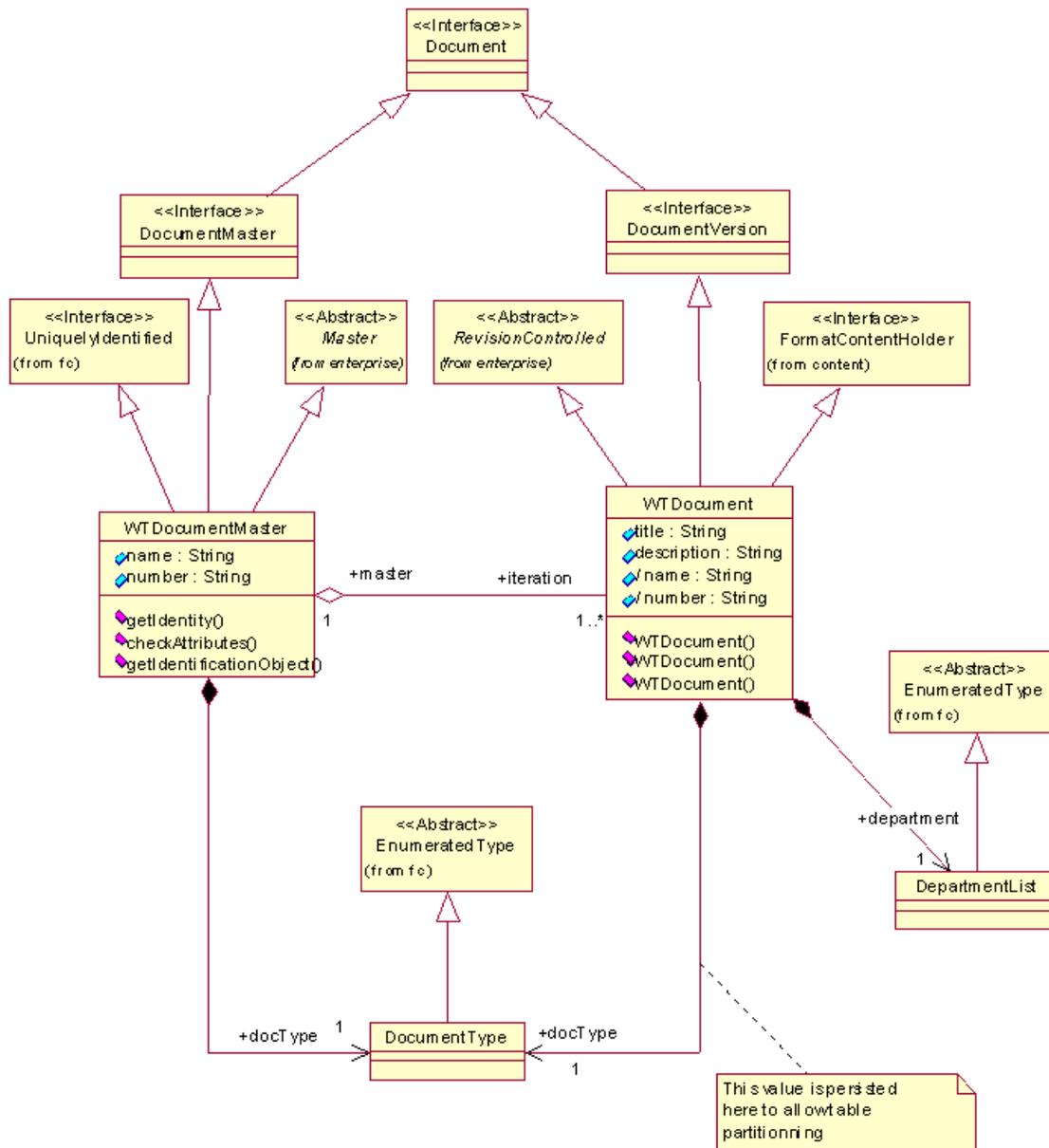
Cabinet managed business class objects are non-iterated, life cycle-managed objects. Because cabinet managed business objects are not Foldered, they do not reside in folders and are not visible to users of the Windchill Explorer. They are, however, associated with cabinets for reasons of access control and local search.



Cabinet managed business class

Document Abstractions

The wt.doc package provides a standard implementation of managed documents. The model for these documents is shown in the following figure:



Doc Package

The document classes are implemented based on the pattern established for revision controlled objects in the wt.enterprise package. These classes, `WTDocumentMaster` and `WTDocument`, provide concrete classes exhibiting the management characteristics established in wt.enterprise and add specifics for

documents. The properties of a document are specified on the `WTDocument` class. Then, for normalization purposes, the sum of the properties are stored on the `WTDocumentMaster`. More specifically, `WTDocument` implements `FormatContentHolder` to give it a primary content item and multiple secondary content items. `WTDocument` can create two types of relationships to other documents. The first, `WTDocumentUsageLink`, is similar to the `WTPartUsageLink` in that it also subclasses `IteratedUsageLink`. It does not have a quantity. `WTDocumentUsageLink` is used to create uses relationships between documents or document structure. Documents should use this if a document is made up of sub-documents and the sub-documents can be reused by other documents, or need to be controlled separately. Similar to the part implementation, the `WTDocumentService` has convenience methods for navigating this relationship and there is a `WTDocument ConfigSpec` to filter results. The second, `WTDocumentDependencyLink` is an iteration to iteration link between two documents. This relationship is shown in the client as references. A reference between two documents can be created to show a dependency on another document. A document may reference some information in another document, so during a create or update, a reference to that document is added. The references relationship has a comment attribute that can be used to explain why the reference exists or what the dependency is. `WTDocument Service` also has convenience methods for navigating the `WTDocumentDependencyLink`.

The doc package is an example of implementing a Revision Controlled Business subclass. The concrete document business classes inherit from the Revision Controlled Business model (Master and RevisionControlled) template in the enterprise model. Document inherits most of its functionality from the enterprise object `RevisionControlled`. `RevisionControlled` pulls together the following plug and play functionality: `Foldered`, `Indexable`, `Notifiable`, `DomainAdministered`, `AccessControlled`, `BusinessInformation`, `LifeCycleManaged`, `Version`, `Workable`, and `Changeable`. In addition, it includes interfaces from the content package. This means that a `WTDocument` is a content holder; that is, it can have files or URLs included in it.

Attributes are on either `WTDocumentMaster` or `WTDocument`. Attributes on `WTDocumentMaster` have the same value for all versions and iterations. If an attribute on the master changes after several versions and iterations have been created, the change is reflected in all the versions and iterations. Attributes on `WTDocument` can generally have different values for each iteration, so changes impact only one iteration. This is why content holder is implemented on `DocumentIteration`. It should be noted, however, that the `docType` attribute of a document is held in common for all iterations and versions. It is stored in the `WTDocument` merely to allow for database partitioning based on the document type attribute. Customers wishing to create new document types will add values to the `DocumentType` resource bundle.

The `DocumentType` resource bundle defines all the types of documents. When users construct documents, they may pick a document type from the enumerated list. Customers may add new document types to the list by putting additional values in the resource bundle. A "\$\$" prefix on a document type means it is a

Windchill-provided document type. The "\$\$" prefix should not be used for customer types.

Using the DocumentType resource bundle, it is possible to construct new types of documents that users can pick from. This has the following impacts from an administrative perspective:

- Administrative rules do not recognize the new document types. Therefore, all document types are the same from an administrative perspective; they receive the same access control and indexing rules.
- From a workflow point of view, the docType property can be used as an activity variable for workflow branching logic.

To add new document types that can have different administrative controls, the WTDocument class must be extended. Subclassing of WTDocument also is preferable if there are specific associations in which only some documents can participate. These kinds of rules are difficult to specify without subclassing WTDocument. Use the following rules when extending WTDocument:

- For every new child of WTDocument, you must make a corresponding entry in the DocumentType resource bundle. This ensures that the WTDocumentMaster object for each WTDocument child knows the type for its document version.
- When adding new classes of documents, it is not necessary to extend the WTDocumentMaster class, only the WTDocument class. All children classes of WTDocument can share the same WTDocumentMaster class.
- Follow the constructor pattern established in WTDocument. Override the appropriate initialize methods from WTDocument, invoking super.initialize() and then performing your class specific logic. Specifically, invoke the method initialize(number, name, type) where type is substituted for a value that has been added to DocumentTypeRB.java.

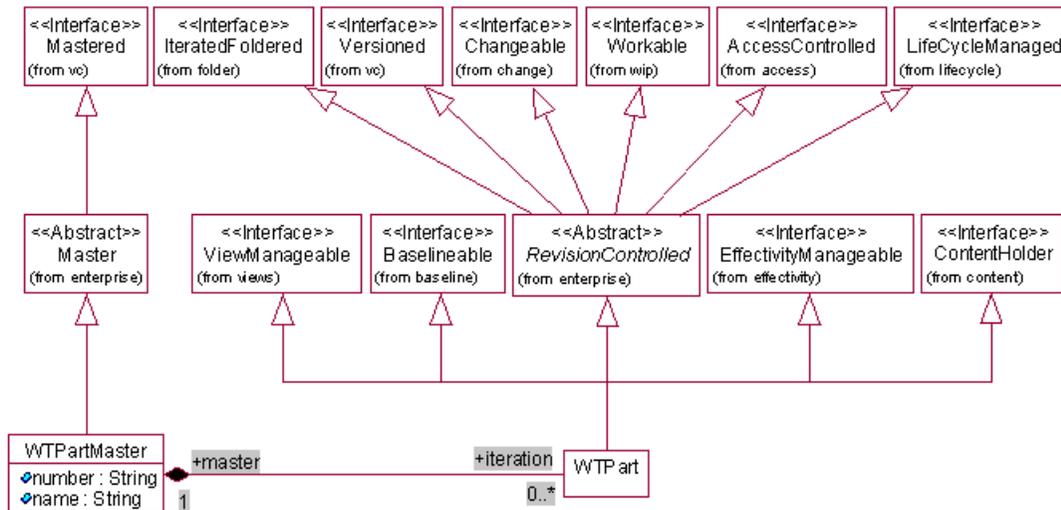
Department is implemented as an enumerated type attribute or a valid value list. The valid values are defined in the wt.doc.DepartmentListRB.java file. The values in DepartmentListRB.java can be changed, the file compiled, and replaced in the codebase. For further information, see the section on [Extending the EnumeratedType class](#) in chapter 9, [System Generation](#) and the *Windchill Customizer's Guide* appendix, Enumerated Types.

Part Abstractions

The wt.part package provides a standard implementation of parts. A part is an item that can be produced or consumed, such as, an engine, a bolt, or paint. Parts can be assembled to produce other parts; for example, the drive train of an automobile can be thought of as a part composed of an engine, transmission, shaft, differential, and so on.

Design Overview

The following figure illustrates the basic concepts encapsulated by the Windchill part reference implementation.



Part Reference Implementation

The part classes are implemented based on the pattern established for revision controlled objects in the wt.enterprise package. These classes, WTPartMaster and WTPart, provide concrete classes exhibiting the management characteristics established in wt.enterprise and add specifics for parts. The properties of a part are specified on the WTPart class. Then, for normalization purposes, the sum of the properties are stored on the WTPartMaster.

The part package is an example of implementing a Revision Controlled Business subclass. The concrete part business classes inherit from the Revision Controlled Business model (Master and RevisionControlled) template in the enterprise model. Part inherits most of its functionality from the enterprise object RevisionControlled. RevisionControlled pulls together the following plug and play functionality: Foldered, Indexable, Notifiable, DomainAdministered, AccessControlled, BusinessInformation, LifecycleManaged, Version, Workable, and Changeable.

Attributes are on either WTPartMaster or WTPart. The WTPartMaster, as a Mastered object, represents the part's identity. As such, "number" and "name" have been placed on it. The part's number is the stamp the enterprise recognizes and uses for tracking purposes. The name is the human-readable component. These properties of the part are assigned carefully and rarely changed.

Attributes on WTPartMaster have the same value for all versions and iterations. If an attribute on the master changes after several versions and iterations have been created, the change is reflected in all the versions and iterations.

The WTPart, as a Versioned and Workable object, undergoes change that is recorded in its versions and iterations as a result of a check-out and check-in process. Attributes on WTPart can generally have different values for each iteration, so changes impact only one iteration.

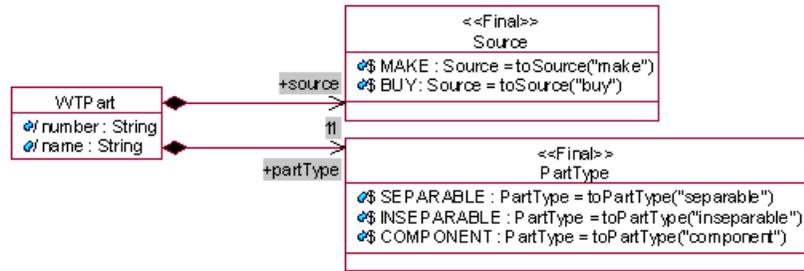
An iteration occurs every time a part is checked out and in. It can be viewed as a working copy of the part. Iterations are assumed to happen many times between versions. Versions, however, represent a business increment; that is, an approved, major change to a part. A typical scenario is that version A of a part is approved and put into production. Then a change is determined to be necessary. The part goes through many iterations while the change is being investigated and tested. Finally, version B is approved.

Also, being ViewManageable, WTPart can be assigned to views, allowing it to progress through stages of development, such as engineering and manufacturing stages. It resides in folders, is subject to access control, progresses through life cycles, and is part of the change process as a consequence of being RevisionControlled. It can also be assigned to baselines to preserve a specific implementation and its versions can be made effective to indicate to a manufacturing process what to build.

Although shown in the preceding figure, WTPart is no longer a ContentHolder by default. The capability to hold files and URLs still exists, but it is no longer exposed to the user. You can change this default functionality, however, as described in the chapter on Customizing GUIs in the *Windchill Customizer's Guide*.

Although not shown in the preceding figure, WTPart is also an IBAHolder. When users create or update a part, they can add pre-defined instance-based attributes (IBAs) to the part and set values for them. WTPartMaster is also an IBA holder but setting values in a WTPart does not affect it. Selecting the Publish option, available from the Product Information Explorer's Part menu, causes IBAs to be copied from the WTPart to the WTPartMaster.

The WTPart also contains as aggregated properties a source and a type (as shown in the following figure).



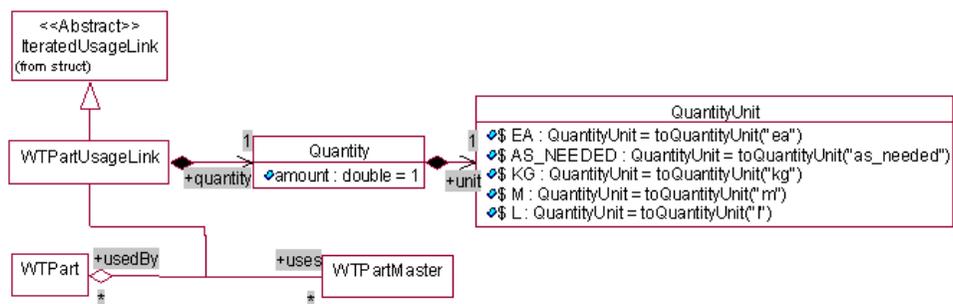
WTPart Properties

The source can be used to indicate how the part is procured, for example by being made or bought. The type specifies how it can be decomposed, for example by being separable (is assembled from components that can be taken apart to be serviced), inseparable (is assembled, but can not be disassembled), or component (is not assembled). The values of these properties can be altered by editing their resource bundles.

Also, note that number and name are modeled as derived and are implemented to set and get the real values from its WTPartMaster. The DerivedFrom property in the Windchill tab of the attribute specification has been used to indicate that it has been derived from the master’s attributes by specifying the database derivation; also, the getters and setters have been overridden in a manner similar to the following:

```
((WTPartMaster) getMaster()).get/set...(...)
```

WTParts can use other parts to build assemblies using the WTPartUsageLink as shown in the following figure.

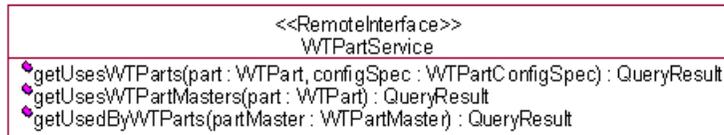


Building Assemblies with the WTPartUsageLink

The WTPartUsageLink is a type of IteratedUsageLink, an association defined to be used to build structures. The WTPartUsageLink’s aggregated Quantity can be

used to indicate the amount of the component that is being consumed. The QuantityUnit's values can be altered by editing its resource bundle.

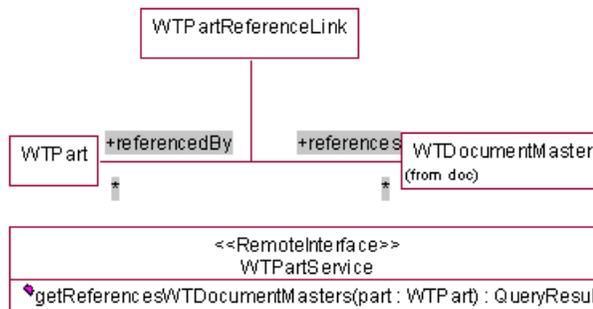
The WTPartUsageLink can be navigated using the PersistenceManager's navigate APIs, or even the StructService's navigateUses and navigateUsedBy APIs. Be aware that navigating the usedBy role results in the returning of all part iterations; StructService's navigateUsedBy API returns only versions. However, the StructService's APIs navigate using the IteratedUsageLink as its target; the WTPartUsageLink might not be the only IteratedUsageLink in a customization. We recommend using the APIs in the following figure.



Navigating the WTPartUsageLink

getUsesWTParts navigates to the WTPartMaster and resolves WTParts from the masters using a WTPartConfigSpec, returning a QueryResult of Persistable[]'s in which the WTPartUsageLink is in the 0th position and the WTPart/WTPartMaster in the 1st. getUsesWTPartMasters simply navigates the WTPartUsageLink and returns a QueryResult of WTPartUsageLinks. Finally, getUsedByWTParts returns a QueryResult of WTParts (the versions, not simply all iterations) representing the implementations that call out the part.

WTParts can also reference documents (see the following figure).



WTPartReferenceLink and Navigate API

Parts generally reference documents for one of two reasons:

- The part is not the logical owner of a document. An example of such a document is a standards document. A standards document is independent of a part, but may be used to verify conformance to the document.
- A document (file) is conceptually owned by the part, but must be separately life cycle managed, checked in and out independently of the file, and so on.

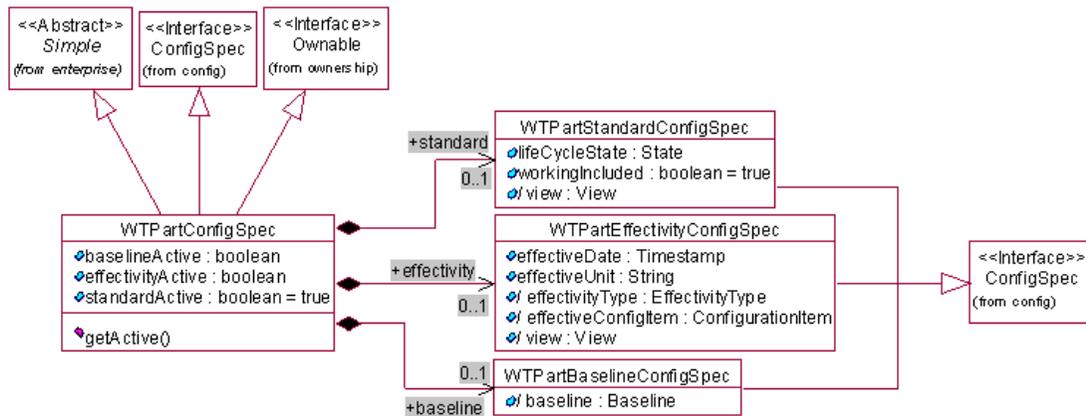
Note that the WTPartReferenceLink may not be appropriate if the document's versions are not necessarily interchangeable from the perspective of the WTPart. If a specific version of a document should be linked to a specific version of a part, use the DescribedBy link (as described later in this section) instead.

The WTPartReferenceLink can be navigated using the WTPartService's getReferencesWTDocumentMasters API.

A WTPart can also be linked to a document that describes it on a version-specific level using WTPartDescribedByLink. An example of such a document is a CAD drawing that shows exactly how a specific version of a part is designed and should be built. If a change is made to the part and a new version created, the revised version of the CAD drawing, that reflects that change, should be linked to the new part using the DescribedBy functionality.

To summarize, a reference should be considered supplemental information that is useful but not required. It is likely to have its own life cycle and change independently of the part referencing it. A document linked to a part by a DescribedBy link contains information you may need specifically for that version of the part. A specific version of the document is linked to a specific version of the part.

The WTPartConfigSpec was alluded to by the getUsesWTParts API. It is used by the Product Information Explorer during its navigations. It consists of three ConfigSpecs: the WTPartStandardConfigSpec, the WTPartEffectivityConfigSpec, and the WTPartBaselineConfigSpec (as shown in the following figure).



WTPartConfigSpec

A concept of zones has been added to the `WTPartConfigSpec` to determine which `ConfigSpec` is active at any given time. The `WTPartConfigSpec` is stored, one per principal, using the `WTPartService`'s APIs listed in the following figure.

<code><<RemoteInterface>></code> <code>WTPartService</code>
<code>findWTPartConfigSpec() : WTPartConfigSpec</code> <code>saveWTPartConfigSpec(a_PartConfigSpec : WTPartConfigSpec) : WTPartConfigSpec</code>

Finding and Saving the `WTPartConfigSpec`

The `ConfigSpecs` aggregated by the `WTPartConfigSpec` have the following behavior:

WTPartStandardConfigSpec

When active, `WTParts` are filtered based on their state and their view membership. `workingIncluded` can be used to allow users to toggle between their working copies and their checked-out versions.

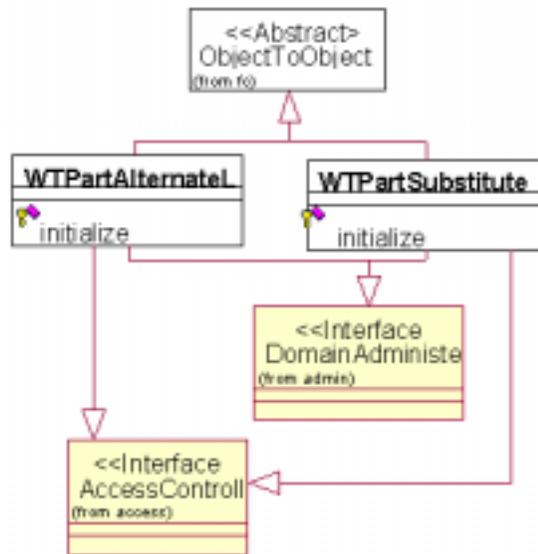
WTPartEffectivityConfigSpec

When active, allows the user to see structures based on effectivity and view. Only `WTParts` designated as effective are shown (see the `wt.effectivity` package for additional information).

WTPartBaselineConfigSpec

When active, displays only those `WTParts` assigned to the specified baseline (see the `wt.vc.baseline` package for additional information).

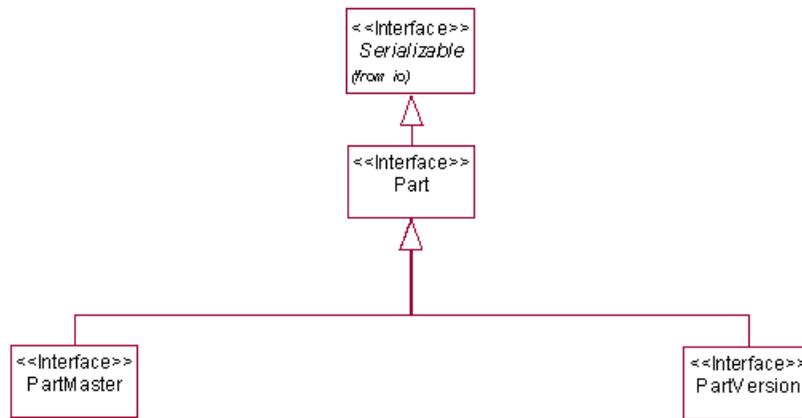
Parts can often be replaced by other parts, either globally or in the context of an assembly. This interchangeability is used to indicate that one part is equivalent to another in a given situation. The WPartAlternateLink (shown in the following figure) is used to indicate global interchangeability, while the WPartSubstituteLink indicates interchangeability within the context of an assembly. Note that the WPartSubstituteLink is copied whenever the WPartUsageLink is copied.



Alternate and Substitute Links

Both of these links can be navigated using the Persistence Manager's navigate APIs. In addition, the WPartService offers `getAlternatesWPartMasters` and `getAlternateForWPartMasters` methods for navigation of WPartAlternateLinks and `getSubstitutesWPartMasters` and `getSubstituteForWPartUsageLinks` methods for navigation of WPartSubstituteLinks. Both WPartAlternateLinks and WPartSubstituteLinks are access controlled, so permission to perform operations such as creation and deletion of links is defined using the access control service.

The Part, PartMaster, and PartIteration classes modeled in the wt.part package (see the following figure) are placeholders for future functionality.



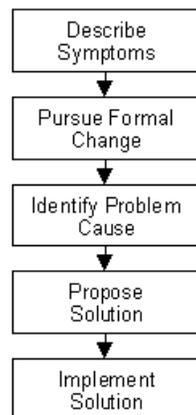
Placeholders

Change Abstractions

The change2 package includes the basic service methods and change item classes necessary to support change management. The change management module provides the means by which users can identify, plan, and track changes to the product information managed by the Windchill product data management system.

Note: The change2 package replaces the change package available in releases prior to Release 4.0.

The following figure shows the five conceptual steps in the change management process.



Change management process

To understand the Windchill change management object model, it is important to understand these conceptual steps, as described below. Although the order of these steps is not fixed, they are presented here in a logical sequence.

Describe symptoms

The symptoms of a perceived problem are recorded. The person experiencing the symptoms could be an internal employee, a customer, or any other end user or person. This person records the symptoms.

Pursue formal change

At some point, the group of symptoms is evaluated. A formal decision is made to investigate the symptoms.

Identify problem cause

By investigating the symptoms and performing analysis, the root cause of the problem is determined. As part of this work, the person investigating the problem may identify relevant parts or documents.

Propose solution

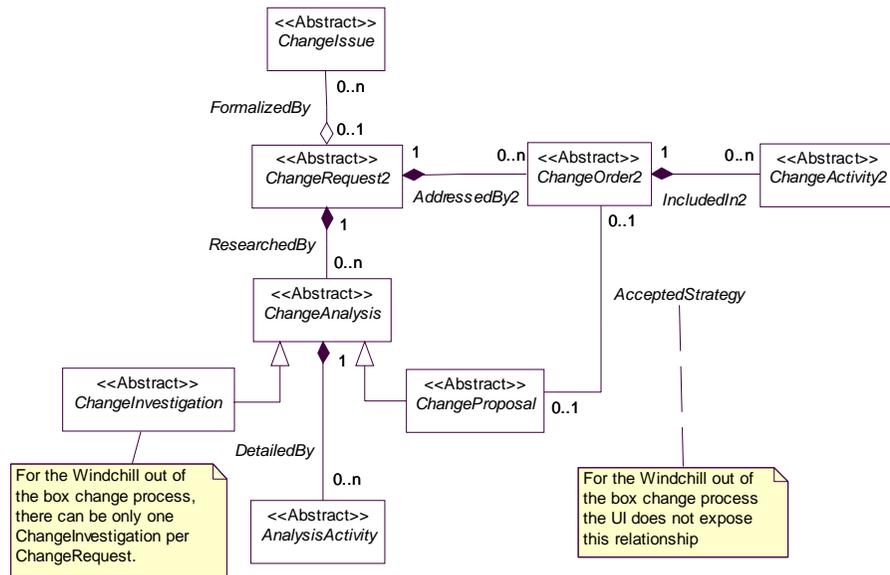
A course of action to fix the problem is proposed. As part of this work, the person preparing a solution may identify relevant parts or documents.

Implement solution

A solution is chosen and implemented. As part of the implementation work, the users of the change process identify part or document revisions, both old revisions (that is, those that require a change) and new revisions (that is, those that have been changed). This step includes the incorporation of the solution into production, if appropriate.

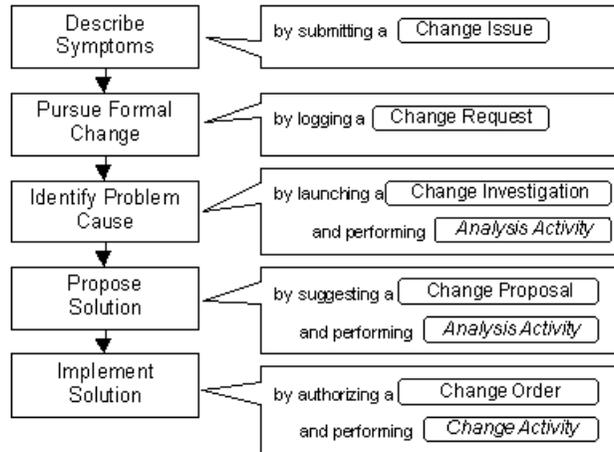
Change Item Classes

The following figure shows the model for the change classes provided by Windchill's change management module.



Change Management Class Model

The following figure shows the relationship between the change item classes and the change management process shown earlier.



Change Management Process and Related Classes

Following are descriptions of the available change objects:

Change issue

A change issue holds information about the problem’s symptoms. A change issue can be thought of as a suggestion box.

Change request

A change request is the object that organizes the other change objects. It represents a formal, traceable change. This object can be associated with product data versions (for example: parts, products, product instances, documents, or CAD documents).

Change investigation

A change investigation organizes the information pertaining to the root cause of the problem. It is used when the root cause is not trivial or obvious. If the research to determine the root cause is very detailed or complicated, analysis activities are used to organize the effort into logical work breakdowns.

Change proposal

A change proposal organizes the information pertaining to a solution for the problem. It is used when the problem solution is not trivial or obvious. If the research to determine the solution is very detailed or complicated, analysis activities are used to organize the effort into logical work breakdowns.

Analysis activity

An analysis activity is used in conjunction with either a change investigation or a change proposal. When the cause of the problem is complex, an analysis activity helps to further organize a change investigation. When the solution to a problem is complex, an analysis activity helps to further organize a change proposal. This object can be associated with product data versions that are relevant to the analysis work.

Change order

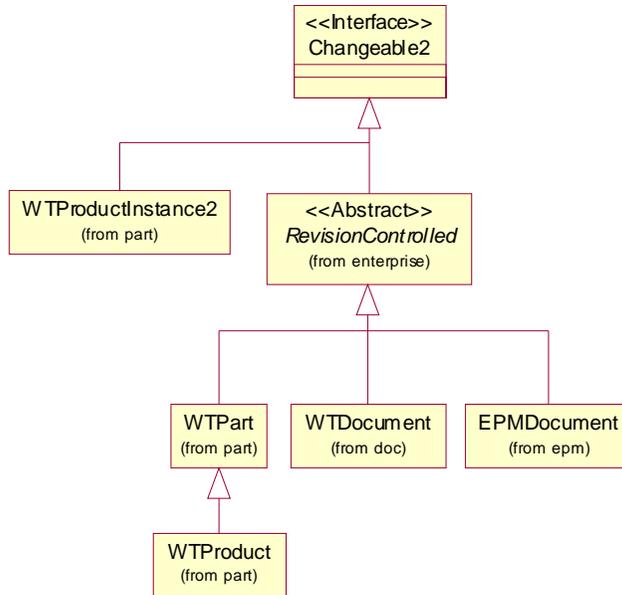
A change order is created if a change proposal is chosen for implementation.

Change activity

A change activity serves as the work breakdown for a change order. This object can be associated with product data versions for two distinct reasons: the product data is defective or otherwise must be changed, or the product data version is a result of a change. This allows users of the system to track the reason for change to product data.

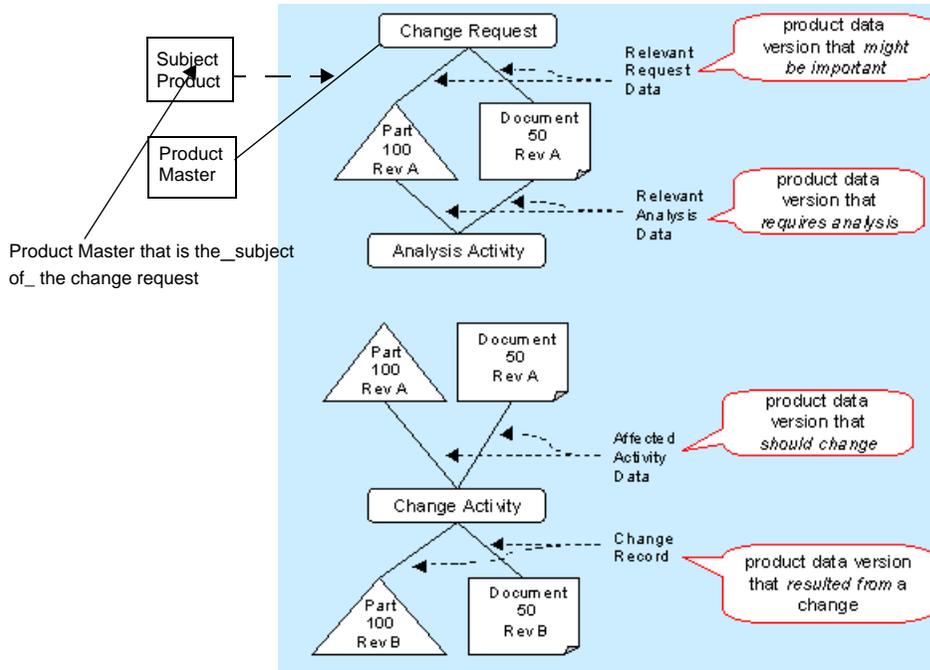
Associations with Product Information

A changeable is a PDM object whose changes can be tracked by the change management system. These are referred to as product data versions to the end user. The term product data version appears on the Change Management user interface rather than the term "changeable." At this release, Windchill parts, products, product instances, documents, and EPM documents are the only changeables, as shown in the following figure.



Changeable Objects

The change management model defines four distinct relationships between change objects and changeables. Any number of changeables can be associated with each of the relationships. In the following figure, one part and one document is associated to the change object in each case. Also shown is a relationship between a change request and a product master, which is not changeable.



Relationships to Product Information

The following are the four relationships between change objects and changeables and the relationship between a change request and a product master.

Relevant request data

The relevant request data association identifies changeables that are relevant to the change request for any reason.

Relevant analysis data

The relevant analysis data association identifies changeables that are relevant to the analysis for a change investigation or change proposal.

Affected activity data

The affected activity data association identifies changeables that must be changed to satisfy a change activity.

Change record

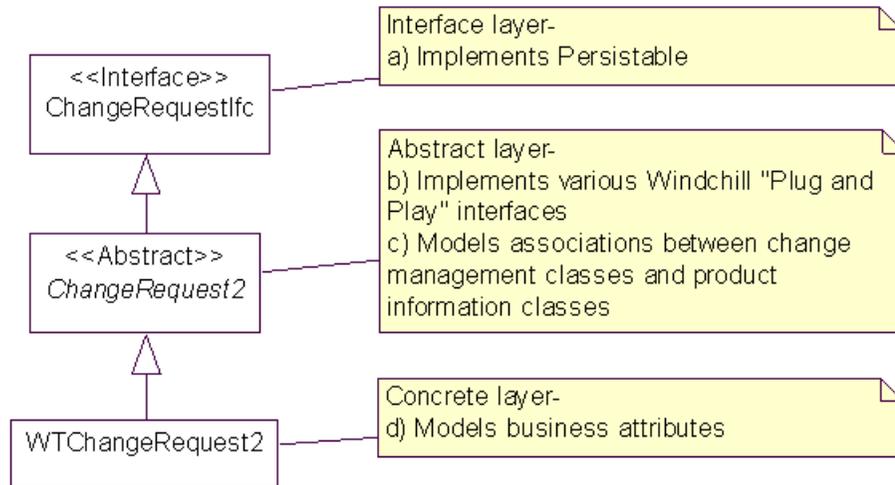
The change record association identifies changeables that have been changed as a result of a change activity.

Subject product

The subject product association identifies product masters that are the subject of the change request. Product masters are not changeables and may not be directly affected by the change request.

Change Item Modeling Approach

Each change management object is modeled using a three-tiered approach: interface, abstract class, and concrete class, as shown in the following figure:



Change Item Modeling Approach

Interface layer

The interface classes simply implement the Persistable interface to indicate that the change management classes are objects stored by Windchill.

Abstract classes

The abstract classes implement various Windchill plug and play interfaces and enterprise package classes to obtain standard Windchill functionality. Each abstract class implements ContentHolder (see the content package in chapter 5, Windchill Services), which provides the ability to attach files. In addition, each abstract class extends either Managed or CabinetManaged (see the enterprise package, earlier in this chapter for an explanation of these classes). The abstract classes also contain modeled associations among change objects, and between change objects and product information objects.

Concrete classes

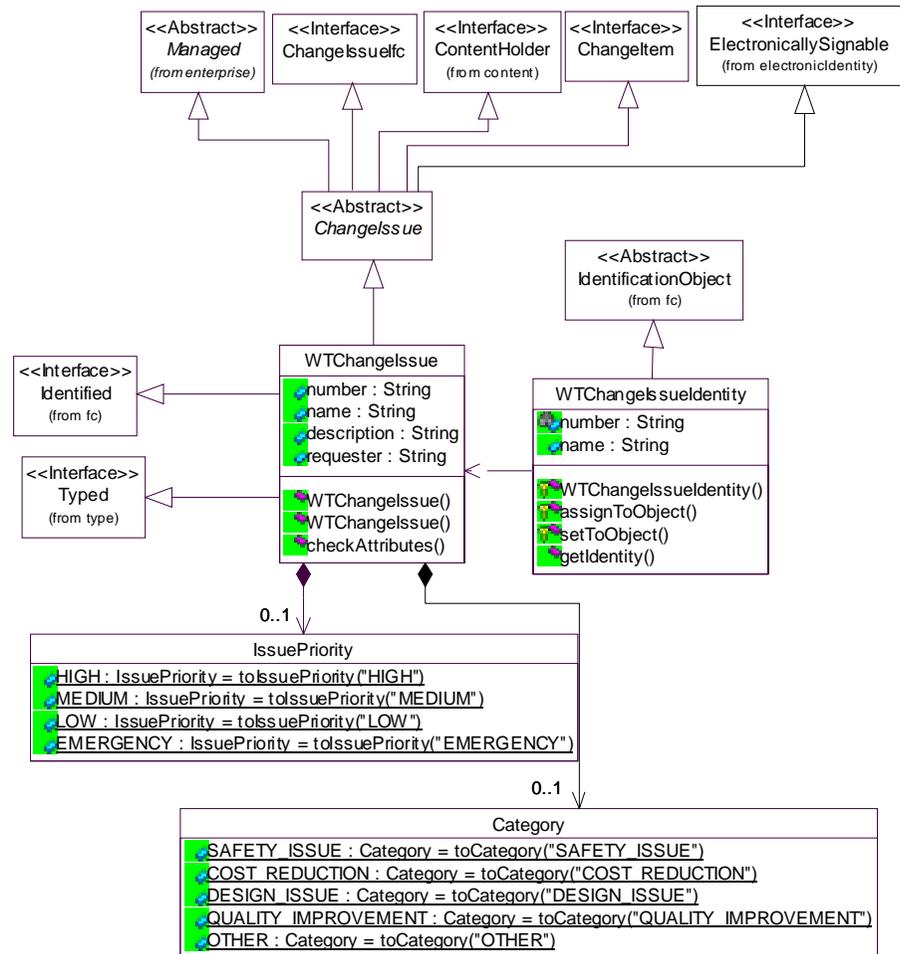
The concrete classes contain modeled business attributes.

Change Items Classes

The following sections show the models for the change item classes. Because many attributes are common, descriptions of all the attributes defined on these items are shown at the end of this section under [Change Item Attribute Descriptions](#).

Change issue

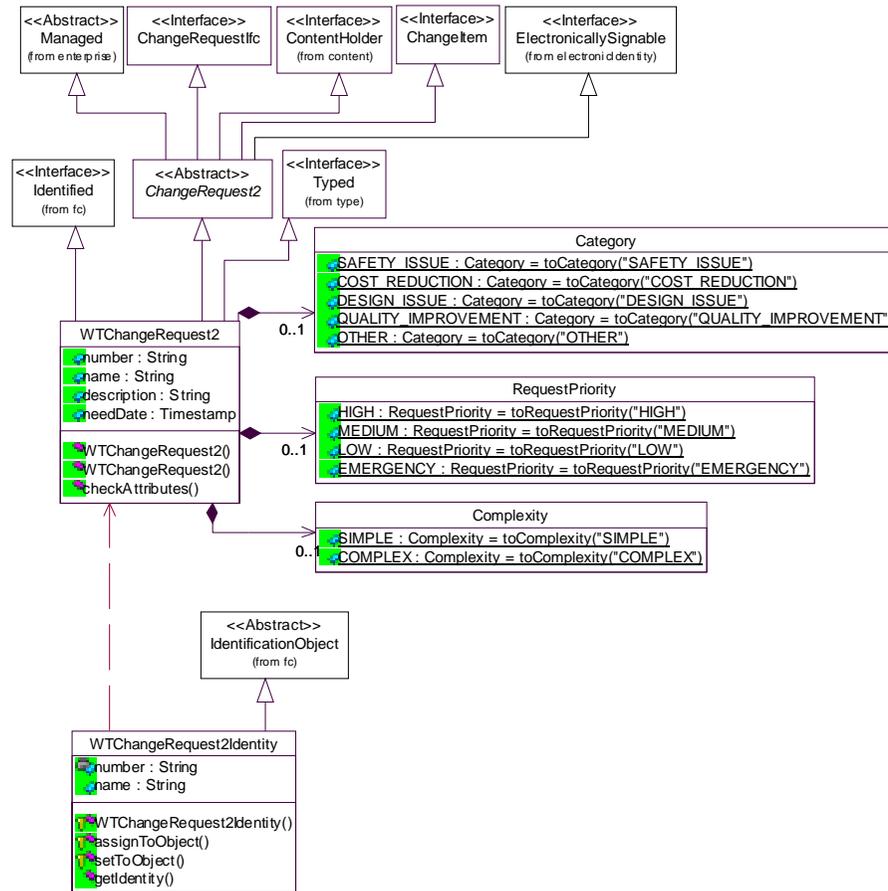
The following figure shows the model for change issues:



Change Issue Model

Change Request

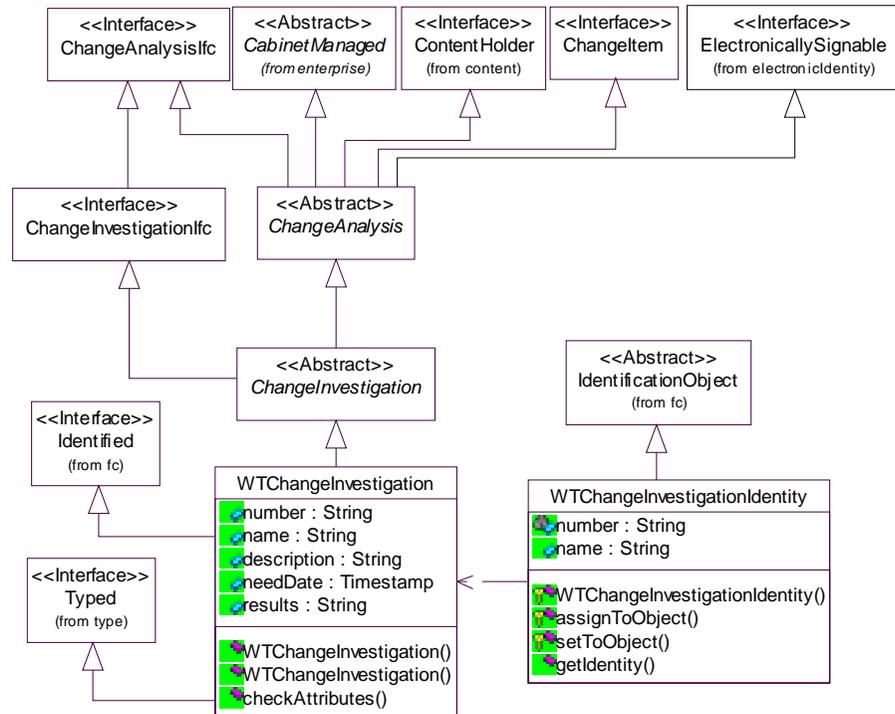
The following figure shows the model for change requests:



Change Request Model

Change Investigation

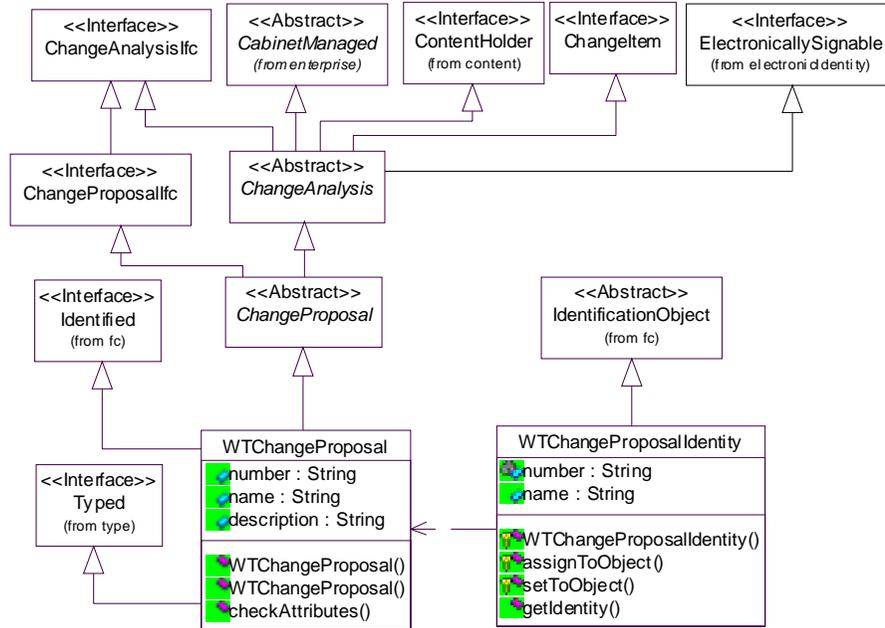
The following figure shows the model for change investigations:



Change Investigation Model

Change Proposal

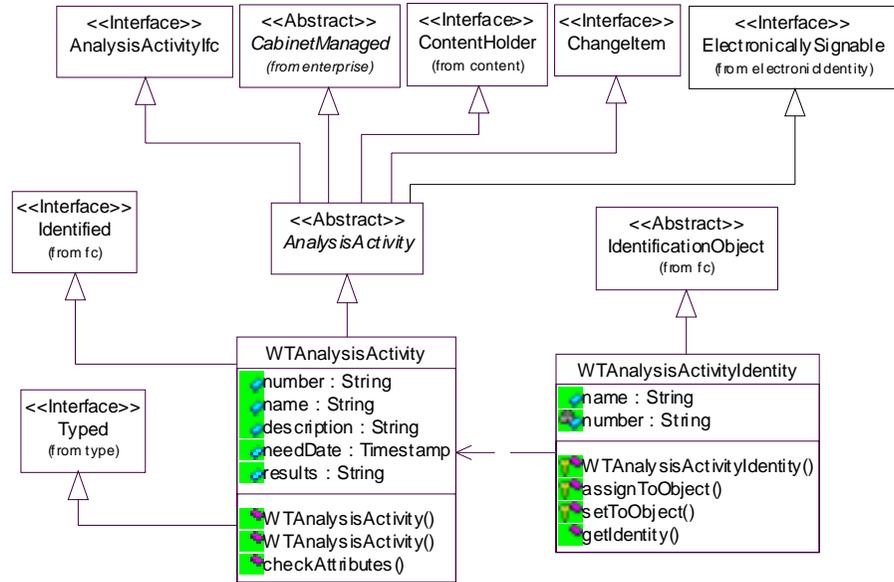
The following figure shows the model for change proposals:



Change Proposal Model

Analysis Activity

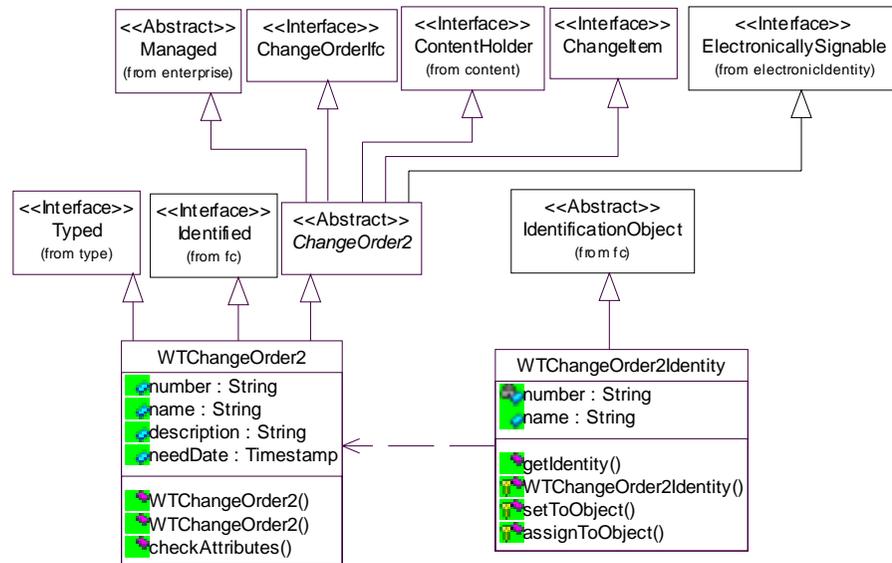
The following figure shows the model for analysis activities:



Analysis Activity Model

Change Order

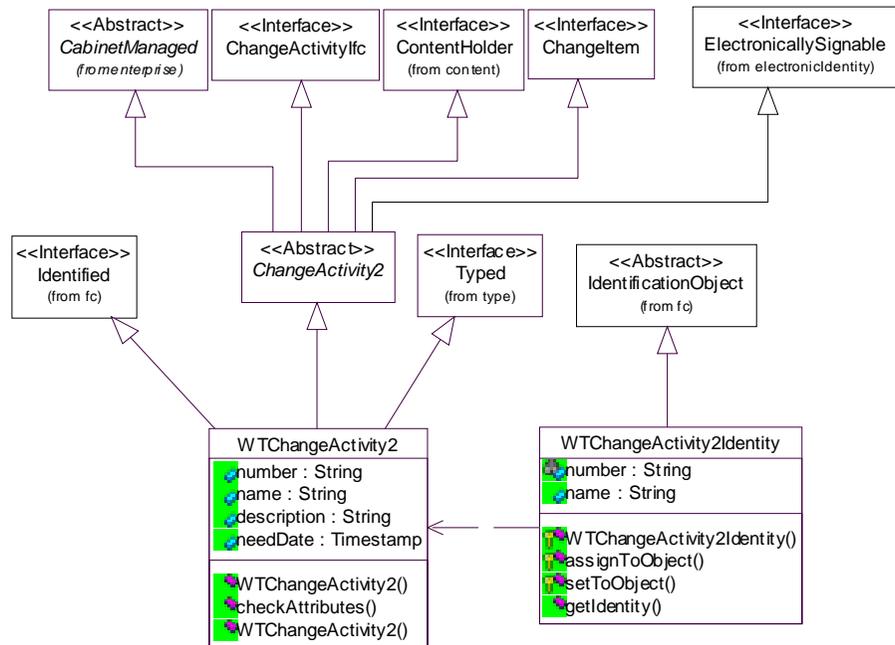
The following figure shows the model for change orders:



Change Order Model

Change Activity

The following figure shows the model for change activities:



Change Activity Model

Change Item Attribute Descriptions

The following is a list of the attributes on change manager items:

category

The category to which the change object belongs. The category identifies the general reason for the suggested change (for example, a cost reduction, quality improvement, or safety issue).

complexity

The complexity of the change object.

description

A description of the change object.

issuePriority or requestPriority

The priority of the change object.

name

The summary of the change object. This attribute is displayed on the user interface as "summary."

needDate

The target date by which the change object should be resolved and closed.

number

The automatically generated number of the change object.

requester

The user who created the change object.

results

The results of the change investigation or change proposal.

6

Windchill Services

Topic	Page
Windchill Packages	6-3
access package — Access Control Service	6-5
admin package — Domain Administration Service	6-9
content package — Content Handling Service.....	6-11
content replication — Content Replication Service	6-17
effectivity package — Effectivity Service	6-19
federation package — Federation Service.....	6-24
folder package — Foldering Service.....	6-27
fv package — File Vault Service	6-30
identity package — Identity Service	6-32
Import and Export Package	6-36
index package — Indexing Service.....	6-37
lifecycle package — Life Cycle Management Service	6-40
locks package — Locking Service	6-45
notify package — Notification Service	6-48
org package — Organization Service.....	6-51
ownership package — Ownership service	6-52
project package — Project Management Service.....	6-53
queue package — Background Queuing Service	6-55
router package — Routing Service	6-57
scheduler package — Scheduling Service.....	6-61
session package — Session Management Service	6-62
Standard Federation Service doAction() and sendFeedback().....	6-63
vc package — Version Control Service	6-69
vc package — Baseline Service	6-75
vc package — Configuration Specification Service.....	6-78
vc package — Version Structuring Service	6-85
vc package — Version Viewing Service.....	6-87
vc package — Work in Progress Service	6-89

workflow package — Workflow Service	6-92
Engineering Factor Services	6-98
EPMMemberLink	6-105

Windchill Packages

Windchill's functionality — that is, its services and utilities — is generally separated into packages. These packages are available in the wt directory within the Windchill source directory, src. You can use these packages as you build new applications and customize existing applications. Model files are provided so you can modify the models and then generate code from them.

This section lists a subset of the packages available and gives an overview of their functionality. The remainder of this chapter then describes the services provided by these packages in more detail. If not described in this chapter, the following list refers to the chapters where they are described.

access

Functionality for access control; used to define access policies (that is, define rules for what principals have access to what information).

admin

Functionality to create administrative domains and policies.

content

Functionality for handling content data (attaching files and URLs to content holders, such as documents and change objects) and associating business information metadata (such as the author) with content.

content replication

Functionality for increasing the speed at which users can access data. Data is stored on more rapidly accessible external vaults known as replica vaults. Base packages cannot be extended or modified in any way. Additional information about content replication is discussed in the Windchill Administrator's Guide.

effectivity

Functionality to assert that a PDM object is effective under certain conditions.

federation

The federation service (wt.federation package) provides functionality to create and manage proxy objects of remote systems and perform utility functions supporting the federation system.

folder

Functionality to put information into folders and cabinets for navigational purposes.

fv

Functionality to define and execute the vaulting algorithm for content items.

identity

Functionality to display the identity of business objects; that is, their type and identifier (for example, a type of part and an identifier of part number).

Import and Export

Windchill Import and Export can assist you in moving complete Windchill content and metadata to and from Windchill sites and Windchill ProjectLink portals by placing the data in Jar files.

index

Functionality to index metadata and content data, controlling how information is put into search indexes.

lifecycle

Functionality to define and use life cycles.

locks

Functionality to lock and unlock objects.

notify

Functionality to define rules and create subscriptions such that when certain events occur to certain objects, E-mail notification is sent.

org

Organization services; functionality to create users and groups (generically called principals).

ownership

Functionality to define ownership of an object.

project

Functionality to create projects, associate projects to business objects, and resolve roles to principals.

queue

Functionality to define and manage queues. Queues are used to persistently record deferred processes. Because queued processes are persistently stored, they are guaranteed to execute at a later time.

router

Functionality to distribute execution of tasks to multiple Windchill method servers.

scheduler

Functionality to schedule execution of resource intensive method invocations and keep a history of their outcomes.

session

Functionality to define and manage user sessions.

Standard Federation Service doAction() and sendFeedback()

Executes one or more Info*Engine tasks selected by a specific logical action name, and the types and physical locations of a specific set of objects passed as parameters. sendFeedback() sends feedback objects to the client

vc

Version control; functionality to handle versions of objects. Version control services described in this chapter include baselines, configuration specifications, version structuring, version viewing, and work in progress.

workflow

Functionality to create and manage workflow definitions, initiate and manage process instances, and distribute work items to users and groups.

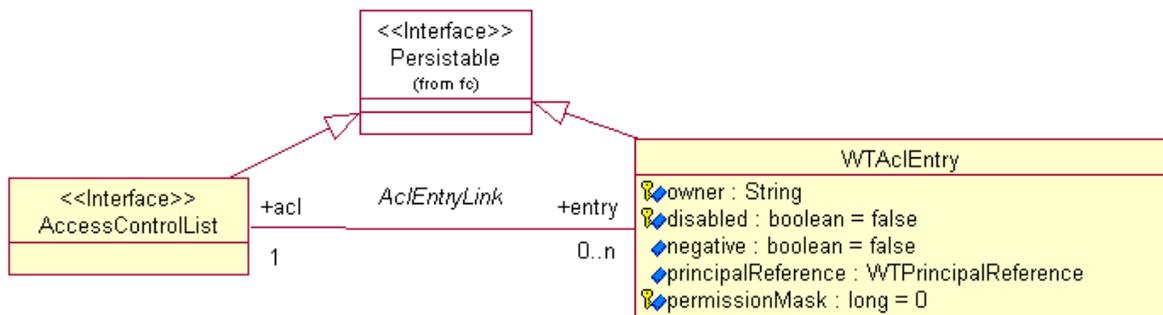
access package — Access Control Service

The access control manager is responsible for defining and enforcing the access to business and system objects. The access control manager resides in the wt.access package.

Design Overview

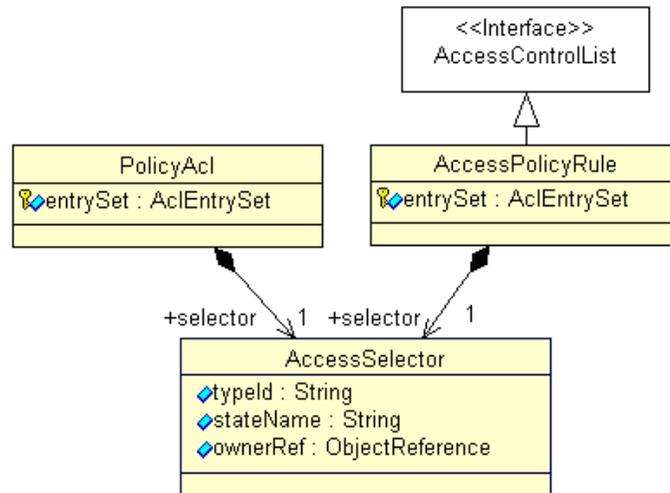
An object is subject to access control if its type implements the AccessControlled interface and implements either the DomainAdministered or the AdHocControlled interface. The AccessControlled interface is simply a marker interface, without any methods. Access control is enforced by access control lists (ACLs), which associate principals to positive or negative sets of access permissions. Permissions are defined by the AccessPermission class. When a user attempts to perform an operation on an object, the ACLs are evaluated according to the model set forth by the java.security.acl.Acl interface, and the access is granted or denied.

Two types of ACLs can be associated to an object: a policy ACL, generated from access control policy rules, and an ad hoc ACL, generated from ad hoc access control rules. A single policy ACL applies to all objects of a specific type while the ad hoc ACL is specific to a single object. The AccessPolicyRule class defines policy rules and the AdHocControlled interface defines ad hoc rules. Both the AccessPolicyRule class and the AdHocControlled interface implement the AccessControlList interface, which associates a set of entries to a rule, and both contain an entry set attribute (defined by the AclEntrySet class), which is a cache of the entries. An access control rule entry is defined by the WTAclEntry class.



Access Control Entries

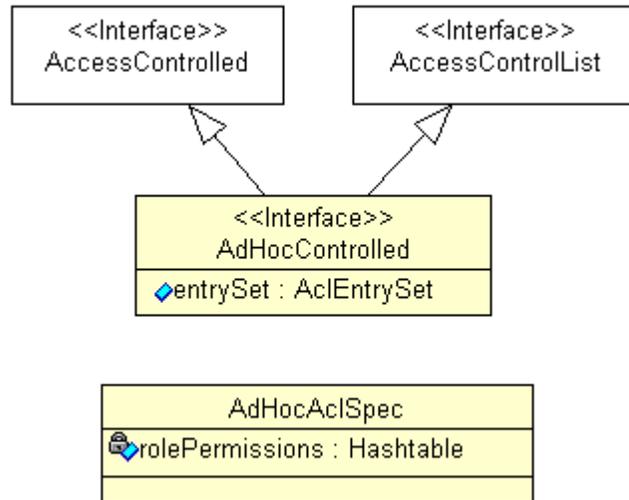
To be associated to a policy ACL, the object must belong to a domain; that is, its type must implement the DomainAdministered interface (see the Domain administration section later in this chapter). Each domain has its own access control policy that may contain many policy ACLs and rules.



Access Control Policy

Both the policy ACL and the rule contain a selector object that contains information about which domain, type and state they are associated with. They also contain a cached set of entries, each entry representing a specific mapping between a principal (that is, a user, a group, or an organization; see the Organization service section later in this chapter) and a set of permissions. The difference between the policy ACL and the rule is that an ACL is derived from all of the rules that apply to a domain, type and state. Applicable rules include those inherited from ancestor domains and types. See the Windchill Business Administrator's Guide for additional information. Policy ACLs are created on demand and, for performance reasons, stored persistently in the database and in a server cache.

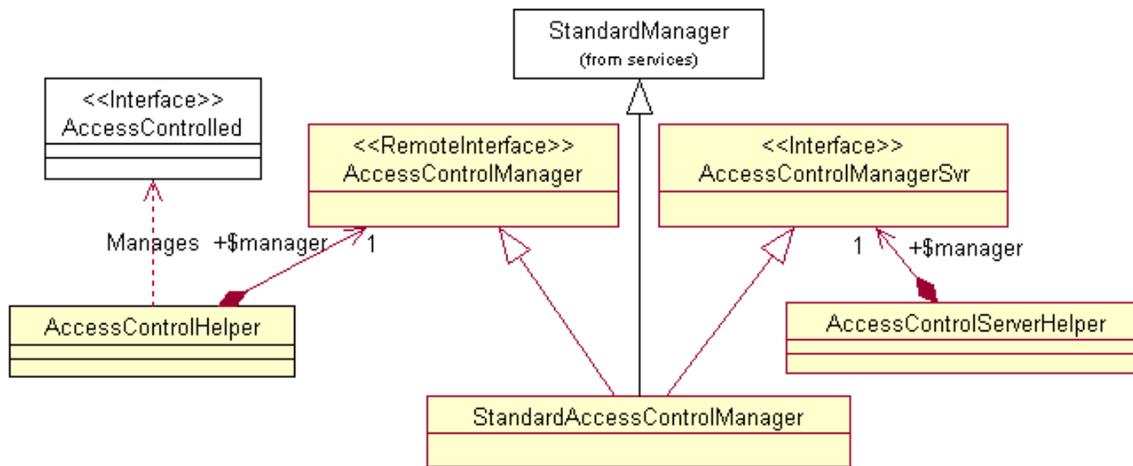
Every object that implements AdHocControlled holds an ACL that is specific to the object, as shown in the figure below.



Ad Hoc ACLs

If an object is both DomainAdministered and AdHocControlled, then both the policy and the ad hoc ACLs determine its access, in an additive manner. In the same way as the policy ACL, the ad hoc ACL also contains a cached set of entries.

Although ACLs can map only principals to permissions, an ad hoc ACL specification (represented by the AdHocAcISpec class) may also map roles to permissions. Ad hoc access control rules can be generated from the ad hoc specification by resolving the roles into principals. Depending on the role to principal mapping used to resolve the roles, the same specification may generate many different ad hoc ACLs.



Access Control Manager

External Interface

The functionality of this service is represented by the `AccessControlManager` (accessible through the `AccessControlHelper`), and the `AccessControlManagerSvr` (accessible through the `AccessControlServerHelper`).

`StandardAccessControlManager` defines the access control enforcement, access control policy, and ad hoc access functionality. For enforcement, the most important method is `checkAccess`, which takes a permission and either an object or a domain and an object type as parameters and throws a `NotAuthorizedException` if access is not granted to the current principal.

Business Rules

The rule for access control enforcement returns true (permission granted) if the object is not access controlled, or if it is not ad hoc controlled and doesn't belong to a domain. If the object belongs to a domain, the policy ACL is generated (or retrieved if it already exists) and the ACL is evaluated. In case the object does not belong to a domain or access is not granted by the ACL, the object is checked to see if it is ad hoc controlled and has an ad hoc ACL. If it does, the ACL is evaluated and access is granted accordingly. Otherwise, access is denied.

When applying access control enforcement to a `DomainAdministered` object, the policy ACLs used are those associated with the domain to which the object belongs and those associated with all ancestor domains. This allows objects to have very different access control behavior even if they are of the same type.

Event Processing

No event is generated. The access control manager listens to domain change events and reevaluates policy ACLs affected by any changes to the domain hierarchy. Control is passed to the manager through explicit method calls.

The access control service now listens for the following events:

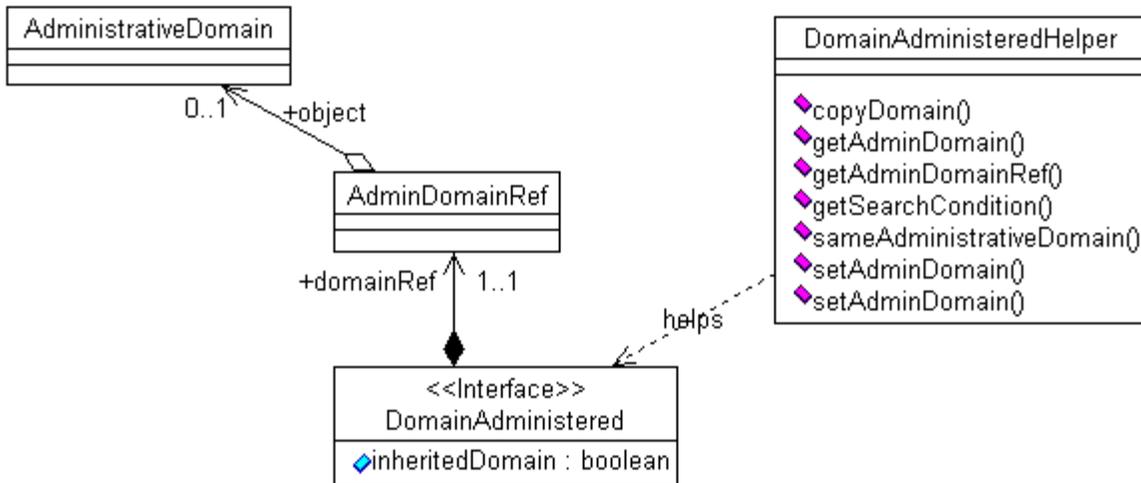
- `AdministrativeDomainManagerEvent.POST_CHANGE_DOMAIN`
- `OrganizationServicesEvent.POST_DISABLE`
- `PersistenceManagerEvent.CLEANUP_LINK`
- `PersistenceManagerEvent.INSERT`
- `PersistenceManagerEvent.REMOVE`

admin package — Domain Administration Service

The administrative domain manager (wt.admin package) is responsible for defining domains and assigning domains to business and system objects. Policies supported include access control (see Access control earlier in this chapter), file vaulting (see File vault service later in this chapter), indexing (see Indexing service later in this chapter), and notification (see Notification service later in this chapter).

Design Overview

An administrative domain can be regarded as the set of all objects that have the same administrative behavior. Administrative policies are defined for objects that belong to a domain. For an object to belong to a domain, its class must implement the DomainAdministered interface. A DomainAdministered object contains a reference to the domain to which it belongs.



Administrative Domain Model

The **AdministrativeDomain** class extends **Item** (from the `wt.fc` package) that is also **DomainAdministered**. That is, domains are themselves subject to administrative rules and belong to another domain. All domains are descendants of the root domain. The **AdministrativeDomain** class implements the **AdHocControlled** interface (from the `wt.access` package) and the **WTCContained** interface (from the `wt.inf.container` package).

External Interface

Access and manipulation of the domain of an object is accomplished by the **DomainAdministeredHelper**, which both defines and implements the API. The methods in this class are static. The methods for setting domains can be used only with objects that have not yet been stored in the database (persisted). The functionality defined in the **AdministrativeDomainManager** and **AdministrativeDomainManagerSvr** can be accessed through the helper classes **AdministrativeDomainHelper** and **AdministrativeDomainServerHelper**, respectively. Changing the domain of a persistent object causes the posting of PRE and POST events (see Event processing later in this section).

Business Rules

Four special domains are defined in the Site container during the installation process: the Root, the System, the Default and the User domains. These

predefined domains cannot be deleted or modified (names however can be set in the wt.properties file). For more information about these domains and about administering domains, refer to the Windchill Business Administrator's Guide.

Changing the domain of an object is a valid operation and can be used to alter the administrative behavior of the object (for example, moving an object to the User domain gives the owner of the object unrestricted rights over the object).

All domain administered objects must belong to a domain. The domain should be assigned during object initialization (by the initialize method). Objects that extend the Item class are placed in the Default domain in the Item's initialization. This can be overridden (or not) by the extending class. Failure to assign a domain to a domain administered object may cause runtime errors.

Event Processing

This service posts two events in case a change of domain is performed on a domain administered object that is persistent. These events are: PRE_CHANGE_DOMAIN and POST_CHANGE_DOMAIN. The administration service listens for the following events:

PersistenceManagerEvent.PRE_DELETE - Listen to attempts to delete AdministrativeDomain objects, to veto deleting the pre-defined and referenced ones.

PersistenceManagerEvent.PRE_MODIFY - Listen to attempts to modify AdministrativeDomain objects, to veto renaming the pre-defined ones.

PersistenceManagerEvent.INSERT - Listen to attempts to create AdministrativeDomain objects, to veto attempts to create domains in the Windchill PDM container with the same name and parent as one of the four pre-defined domains in the Site container.

content package — Content Handling Service

The content package allows content — files, URL links, and aggregates (multiple pieces of content that behave as a single file — to be associated with business classes. These business classes are referred to as ContentHolders. It appears to users that content is contained in the business classes. The content itself is treated like any other attribute of a class and requires read and/or write access on the ContentHolder.

The `FormatContentHolder` interface is used for classes like `document` (see `wt.doc.WTDocument`), where you want to associate a primary format with the document. This format is set automatically when using the `ContentService`. For example, a document containing one Microsoft Word file will get a format of Microsoft Word.

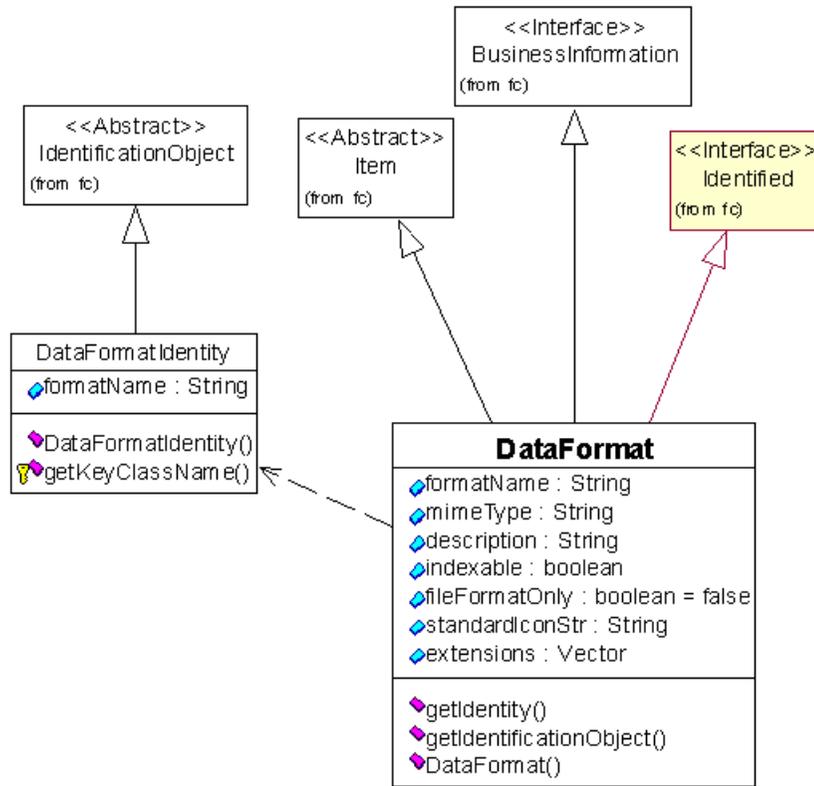
The `ContentRoleType` is an enumerated type that exists primarily for customization. It is not used by the `ContentService` to perform any particular action or behavior except in the case of `FormatContentHolders`. It is used when one piece of content is to be stored as the main piece of content for the document, as opposed to a related attachment. Primary content is, for example, downloaded on a checkout of a document. Its role should be set as `PRIMARY` before it is stored. The code to set the primary content is as follows:

```
ContentItem item = . . .  
item.setRole( ContentRoleType.PRIMARY );
```

The `ContentRoleType` can also be used to specify specific `ContentItems` as renditions, an attached drawing, and so forth. The default enumeration can be updated in `wt/content/ContentRoleTypeRB.java`. However, for proper execution of the system, do not remove the existing values from this resource bundle.

Note that the aggregate pieces of content are not currently supported for customization.

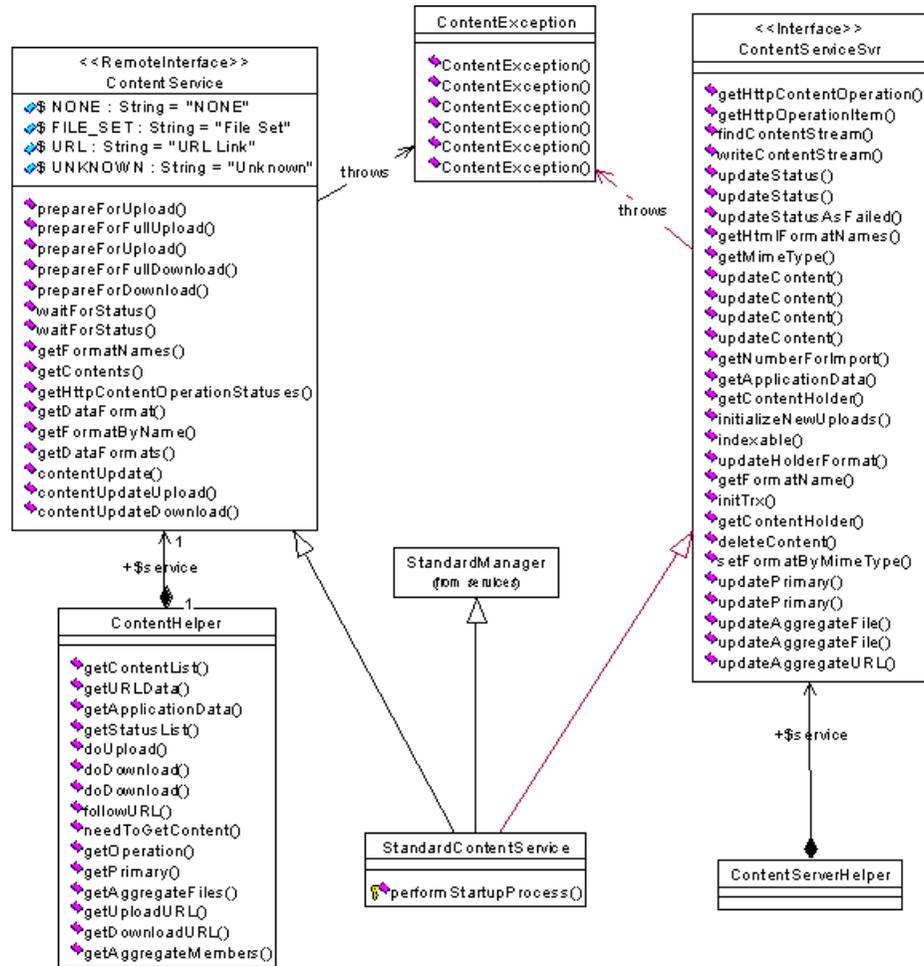
The following figure is a conceptual representation of the DataFormat class.



DataFormat Class

The DataFormat class is an Administrative class that is used to associate a primary format with FormatContentHolders and ContentItems. These should be unique on

formatName, can be created only by an administrator, and can never be deleted. Note that formats on files are set based on file extension and/or mime type.



Content Services

The ContentService is available on the client and server side. The ContentServiceSvr is available only on the server side. The service methods are to be referenced through the corresponding helper class. For further information, see the javadoc for ContentService.

Working with ContentItems

Querying

The method call to get the secondary content associated with a ContentHolder is as follows:

```

ContentHolder holder;
holder = ContentHelper.service.getContents( holder );
Vector contents = ContentHelper.getContentList( holder );

```

This vector contains all the secondary ContentItems associated with the passed ContentHolder.

The method call to get the primary content associated with a FormatContentHolder is as follows:

```

ContentHolder holder;
holder = ContentHelper.service.getContents( holder );
ContentItem item = ContentHelper.getPrimary( holder );

```

Creating, Updating, and Removing

PTC recommends that you set up a signed applet and post to the content service via HTTP. The method call to set up the URL to post to the content service is as follows:

```

ContentHolder holder = . . .
URL postURL = ContentHelper.getUploadURL( holder );

```

To receive a stream from the ContentService, use the following:

```

ContentHolder holder = . . .
URL postURL = ContentHelper.getDownloadURL( holder );

```

When posting to the ContentService, the data must be posted in a particular format. An outline of the format as is follows:

```

attribute name 1 attribute value 1
attribute name 2 attribute value 2
...
attribute name x attribute value x
content_description value <file path here>
attribute name 1 attribute value 1
attribute name 2 attribute value 2
...
attribute name x attribute value x
content_description value <file path here>
... <as many ContentItems as desired>
EndContent null <You must mark the end with this>

```

The content_description line should appear as follows for a new file:

```

newFile c:\path\uploaded_file.xxx file stream . . .

```

The content_description line should appear as follows for an ApplicationData that already exists in the ContentHolder (that is, replace):

```

fileoid c:\path\uploaded_file.xxx file stream . . .

```

The value for fileoid can be generated using the following method call:

```
ApplicationData applicationDataObj = . . .
String oidstr = PersistenceHelper.getObjectIdentifier(
    applicationDataObj ).getStringValue( );
```

The content_description line should appear as follows for a new URL:

```
newURLLink http://xxx.yyy.zzz
```

The content_description line should appear as follows for an existing URL, generating the urloid, like the one for the preceding files:

```
urloid http://xxx.yyy.zzz
```

The ContentService currently handles the following attributes:

```
roleName (see wt.content.ContentRoleType)
```

```
remove
```

```
description
```

The attributes can be passed in any order, as long as the preceding content_description line is passed last for each object being worked on.

Business Rules

The read and modification of content is based on the access rules of the ContentHolder.

Event Processing

At this time, the ContentService does not broadcast any events.

content replication — Content Replication Service

There are six content replication packages. Do not attempt to extend or modify these base packages. Windchill content replication increases the productivity of users by reducing the time it takes to access data.

fv.master — Master Service

The Master service is responsible for replicating to replica sites.

External Interface

The Master service has no API for developers to use.

Event processing

At this time, the Master service does not emit any events for developers to use.

fv.replica — Replica Service

The Replica service is responsible for storing replicas.

External Interface

The Replica service has no API for developers to use.

Event processing

At this time, the Replica service does not emit any events for developers to use.

intersvrcom — InterSvrCom Service

The InterSvrCom service manages the replica sites known to a particular master site. security. It generates and remembers public and private keys. It provides verification services on the replica site.

External Interface

The InterSvrCom service has no API for developers to use.

Event processing

The InterSvrCom service does not emit any events for developers to use.

wrmf.delivery — Shipping Service

The Shipping service is one of the three services in the delivery and transport system. It is responsible for packaging and sending messages and the content files to the replica sites. It can operate without access to an Oracle Instance.

External Interface

The Shipping service has no API for developers to use.

Event processing

The Shipping service does not emit any events for developers to use.

wrmf.delivery — Receiver Service

The Receiver service is like a mailbox that the replica site accesses to find requests to process. It is one of the three services in the delivery and transport system. It can operate without access to an Oracle Instance.

External Interface

The Receiver service has no API for developers to use.

Event processing

The Receiver service does not emit any events for developers to use.

wrmf.transport — Generic Transport Service

The Generic Transport service transports content to replica sites. It is one of the three services in the delivery and transport system. It can operate without access to an Oracle Instance.

External Interface

The Generic Transport service has no API for developers to use.

Event processing

The Generic Transport service does not emit any events for developers to use

Business Rules — content replication

Leveraging Content Replication in Java Applications and Applets — The business reason for taking advantage of Content Replication is that it can dramatically improve download times in a collaborative product commerce scenario. If you are developing in an applet, use the standard Windchill download bean:

`wt.clients.util.http.Download`. The recommended paths for extending the HTML Client are the following:

- If the page is to solely present information, use the `TemplateProcessor` subclass `wt.templateutil.processor.DefaultTemplateProcessor`.
- If the page requires user input, for example, it will have an HTML form in it, use the `TemplateProcessor` subclass `wt.templateutil.processor.GenerateFormProcessor`.

For detailed information about developing HTML clients, see the chapter on customizing the HTML client in the *Windchill Customizer's Guide*.

effectivity package — Effectivity Service

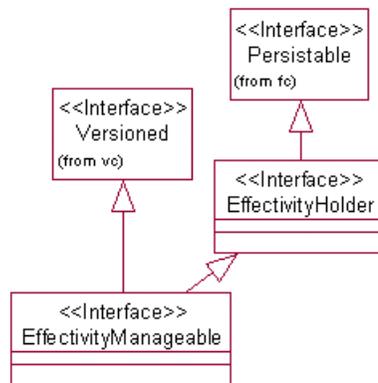
The effectivity package provides a mechanism to assert that a PDM object is effective under certain conditions. This package is based in part on the `PdmEffectivity` section (2.8) of the PDM Enablers specification.

Design Overview

This section describes the effectivity interface, effectivity types, and configuration item.

Interface

The following figure illustrates the effectivity interface.



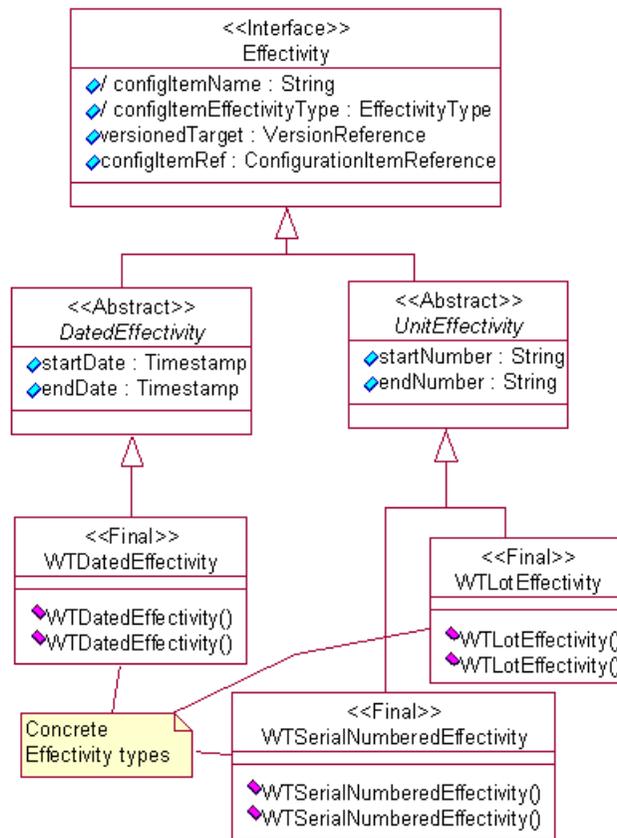
Effectivity Interface

EffectivityHolder serves as a plug-and-play interface to store effectivity information for non-versioned objects. A class that implements EffectivityHolder can reference one or more Effectivity objects. The purpose of this class is to support the storage of effectivities for non-versioned objects, although this is used in the current Windchill release. Note that this release does not include any direct use of EffectivityHolder.

The EffectivityManagable interface provides a plug-and-play interface to store effectivity information for versioned objects. An EffectivityManagable object manages any number of effectivities, although the current Windchill user interface supports only a single effectivity per EffectivityManagable version. All EffectivityManagable objects are Versioned; however, the converse is not necessarily true. If an EffectivityManagable object is deleted, all Effectivity objects that reference the EffectivityManagable will be deleted automatically. wt.part.WTPart will be the only class that asserts EffectivityManagable in this release. That is, only parts will be subject to effectivity functionality.

Effectivity Types

The following figure illustrates effectivity types.



Effectivity Types

Effectivity is an abstract class that supports the concept that a PDM object is effective under certain conditions. Abstract subclasses of Effectivity include DatedEffectivity and UnitEffectivity. Effectivity contains a reference to a ConfigurationItem (configItemRef). This ConfigurationItem reference is necessary for UnitEffectivity subclasses, and is optional for DatedEffectivity subclasses.

Effectivity has two mechanisms for referencing its associated EffectivityManageable object.

- versionedTarget is a reference to a versioned object and is used for classes that implement EffectivityManageable.
- standardTargetReference is a reference to a non-versioned object and is used for classes that implement EffectivityHolder.

The implementer simply calls EffectivityHelper.setEffectivityTarget, which automatically initializes the proper reference.

WTDatedEffectivity, WTLotEffectivity, and WTSerialNumberedEffectivity are concrete classes for storing date, lot number, and serial number effectivity. Each are represented by a separate database table.

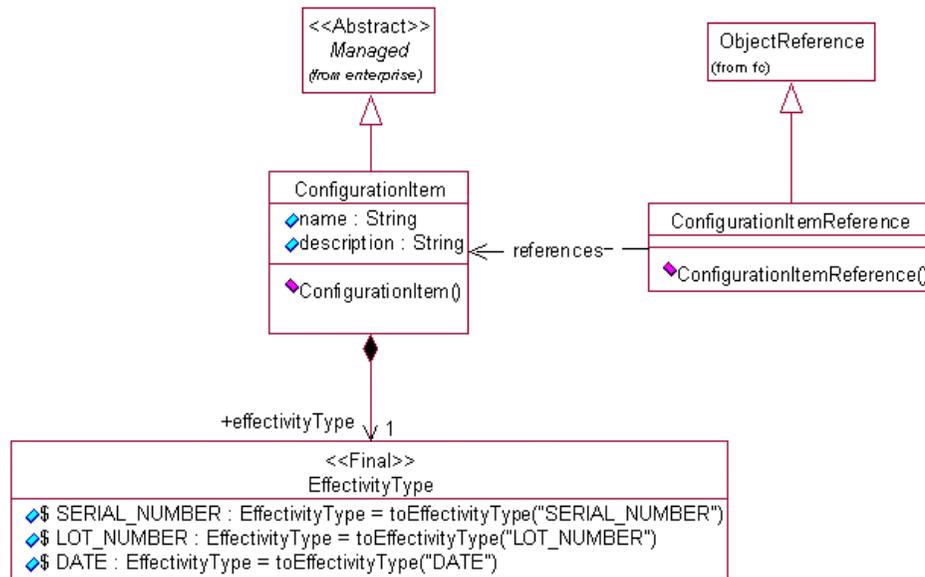
No persistent link classes are used to associate the concrete effectivity types to EffectivityManageable. This association is achieved through the versionedTarget, and standardTargetReference attributes, described above. In this way, an EffectivityManageable object is associated to its concrete effectivity object through a foreign key reference.

A DatedEffectivity is used to indicate that an EffectivityManageable object is effective while a configuration item is being produced during a date range. WTDatedEffectivity is a concrete subclass of DatedEffectivity.

A UnitEffectivity is used to indicate that an EffectivityManageable item is effective as a configuration item is being produced in a range of numbered units. WTSerialNumberedEffectivity and WTLotEffectivity are concrete subclasses of UnitEffectivity, for serial number and lot number effectivity, respectively.

ConfigurationItem

The following figure illustrates the ConfigurationItem.



ConfigurationItem

PDM Enablers defines ConfigurationItem as follows: "A ConfigurationItem is referenced by Effectivity objects ... to specify the product for which the effectivity is being expressed." ConfigurationItem is used to provide a context for unit effectivity (serial number or lot number). The Effectivity class (described above) stores a reference to a single ConfigurationItem. ConfigurationItem is a Managed object. Each ConfigurationItem object aggregates a single

EffectivityType object. The deletion of ConfigurationItem is allowed only if the ConfigurationItem is not being referenced by any Effectivity object.

EffectivityType is a subclass of EnumeratedType, which provides a mechanism for using a Java resource bundle to enumerate the valid Effectivity types.

External Interface

The EffectivityHelper class provides methods for setting the configuration item and target Effectivity Manageable object (WTPart) for an Effectivity object. When creating WTLotEffectivity and WTSerialNumbered objects, both of these methods should be called. When creating WTDatedEffectivity objects, setting a Configuration Item is optional.

The EffectivityManager provides methods for retrieving Effectivity objects from the database, for retrieving configuration item objects from the database, and for storing, modifying, or deleting Effectivity objects. These methods are accessed through EffectivityHelper.service.

Business Rules

The following business rules apply to effectivity:

- Any user who has access to modify a change order or change activity also has permission to change the effectivity for all parts that are referenced by the change order or change activity.
- The data model supports multiple effectivities per part version.
- Effectivity objects are never shared. Each part has unique effectivity information.
- The user may not delete a configuration item if it is referenced by any effectivity object.
- If the user deletes a part with effectivity, its effectivity information will be deleted automatically from the database.

Event Processing

No events are posted by this service. This service listens for and acts upon the following two standard Windchill events:

- When a POST_DELETE event for an EffectivityManageable object is emitted, the EffectivityService will delete all of the Effectivity objects that reference the deleted EffectivityManageable.
- When a PRE_DELETE event for a ConfigurationItem object is emitted, the EffectivityService checks to make sure that no Effectivity objects reference the ConfigurationItem. If any do, the EffectivityService will veto the delete, and the ConfigurationItem may not be deleted.

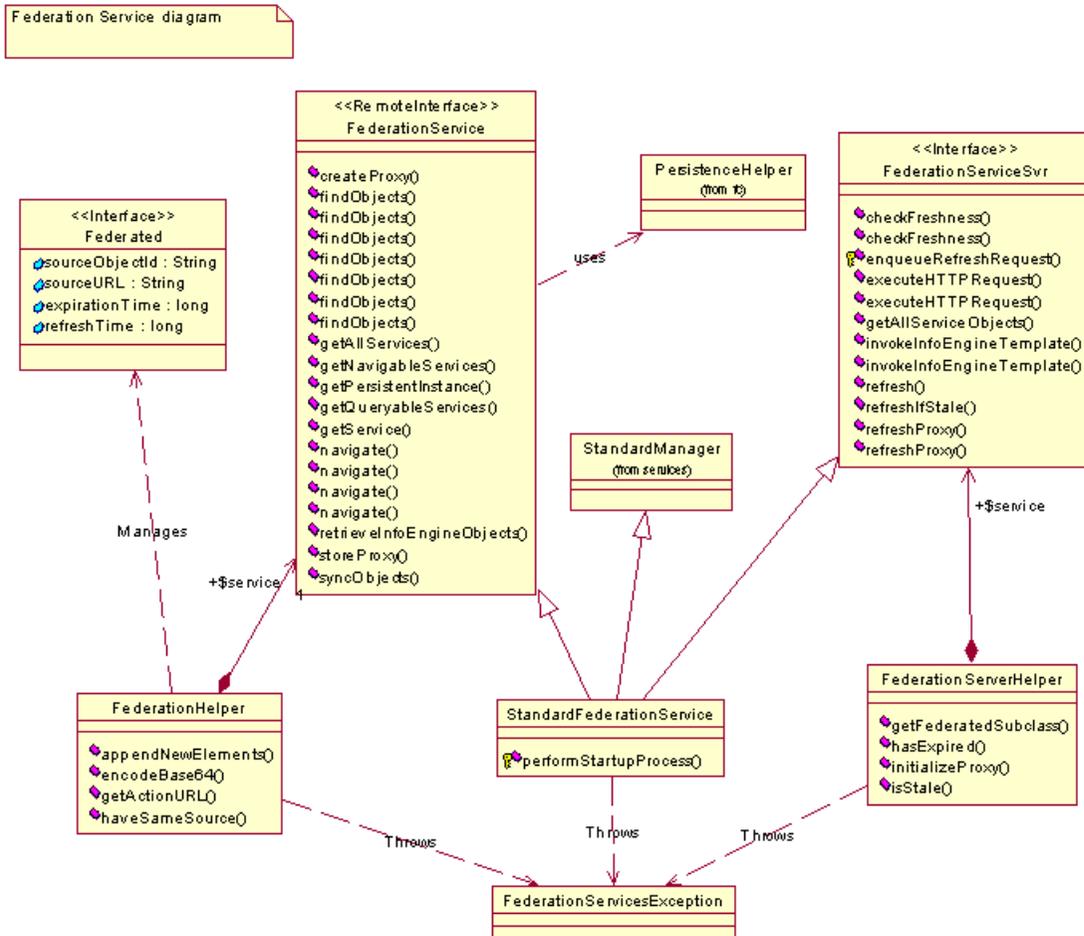
federation package — Federation Service

The federation service (wt.federation package) provides functionality to create and manage proxy objects of remote systems and perform utility functions supporting the federation system.

Design Overview

The federation service is designed to be a plug and play component in the Windchill system. The federation service is intended to be used for both client and server development. Business objects, asserted as being federated in the object model, are assigned a remote system information (serviceID) at creation and can be promoted throughout the defined phases of an associated federation. The proxy source information is held in the serviceID cookie, but instead operate on it through the federation service's external interface.

The following figure contains a representation of the object model for federation service.



Federation Service Model

External Interface

The Federated interface provides an abstraction of a plug-and-play component. The intent is that, in an object model, a business object would assert that it is Federated by implementing the Federated interface. With this assertion, the business object can then be created as a proxy of remote object.

The FederationHelper provides an abstraction as the API to the FederationService. The API's method can be categorized as either local or remote invocations. The local methods are getters of information, typically from cookies that are held in the business object. The remote methods serve as wrappers to a service that promotes server-side functionality.

The FederationService provides an abstraction that specifies and promotes server-side functionality as a service that is remotely available for use by a client. The intent is that this interface defines all the necessary server-side functionality for federation.

The FederationServerHelper provides an abstraction of server-side API to the federation services. These methods can be invoked only from server-side processing.

The FederationServicesException provides an abstraction of an abnormal occurrence or error in usage or processing of the federation service. This exception can be localized through a given resource bundle, and other exception can be nested with in it.

Business Rules

As defined by the standard federation service access control rules, no constraints are placed on the access of proxy objects.

Event Processing

The federation service is an event listener. The service listener listens for and acts upon the following standard Windchill events.

When a PRE_STORE or a PRE_MODIFY event is emitted for a proxy object, the federation service provides the opportunity to store or update associated objects prior to storing or updating the proxy object itself by doing:

```
if ( obj instanceof Federated )
    ((Federated)obj).prepareForStore ();
```

Revising a proxy object is not allowed. Proxy objects are created against source objects from remote system. The revision is prevented to ensure that the proxy objects are consistent with the source objects.

When a POST_STORE or a POST_MODIFY event is emitted for a proxy object, the federation service provides the opportunity to store or update associated objects after storing or updating the proxy object itself by doing:

```
if ( obj instanceof Federated )
    ((Federated)obj).postStore ();
```

When a PREPARE_FOR_MODIFICATION is emitted for a proxy object, the federation service will throw an exception with a message indication that updating federated objects is not allowed.

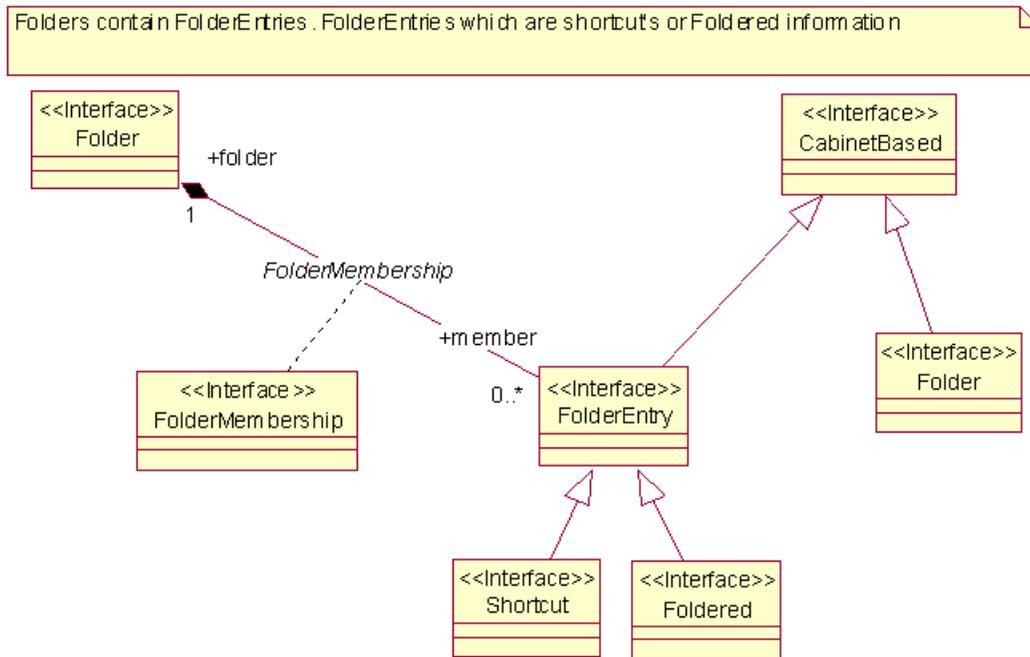
When a PRE_CHECKOUT event is emitted for a proxy object, The checkout of a proxy object is not allowed for R5.0.

folder package — Foldering Service

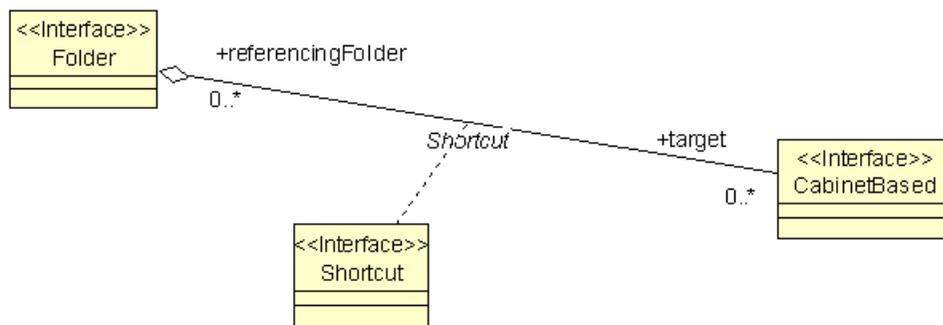
The folder service resides in the package `wt.folder`. It provides the means to organize information into a cabinet and folder hierarchy, much the same way that this is done in operating systems available today. (Note that these are not operating system folders, merely objects that act much like operating system folders.) The cabinets provide not only a means to group information, but also a means to associate information with administrative policies for access control, indexing, and notifications. Folders provide a further breakdown of the information in a cabinet. Because it is common to view information with more than one organization, it is possible to construct alternative folder organizing structures that contain shortcuts (like operating system shortcuts, symbolic links, and aliases) to information residing in other folders.

Design Overview

The following figure shows the essential associations and interfaces involved with managing folder organized information.

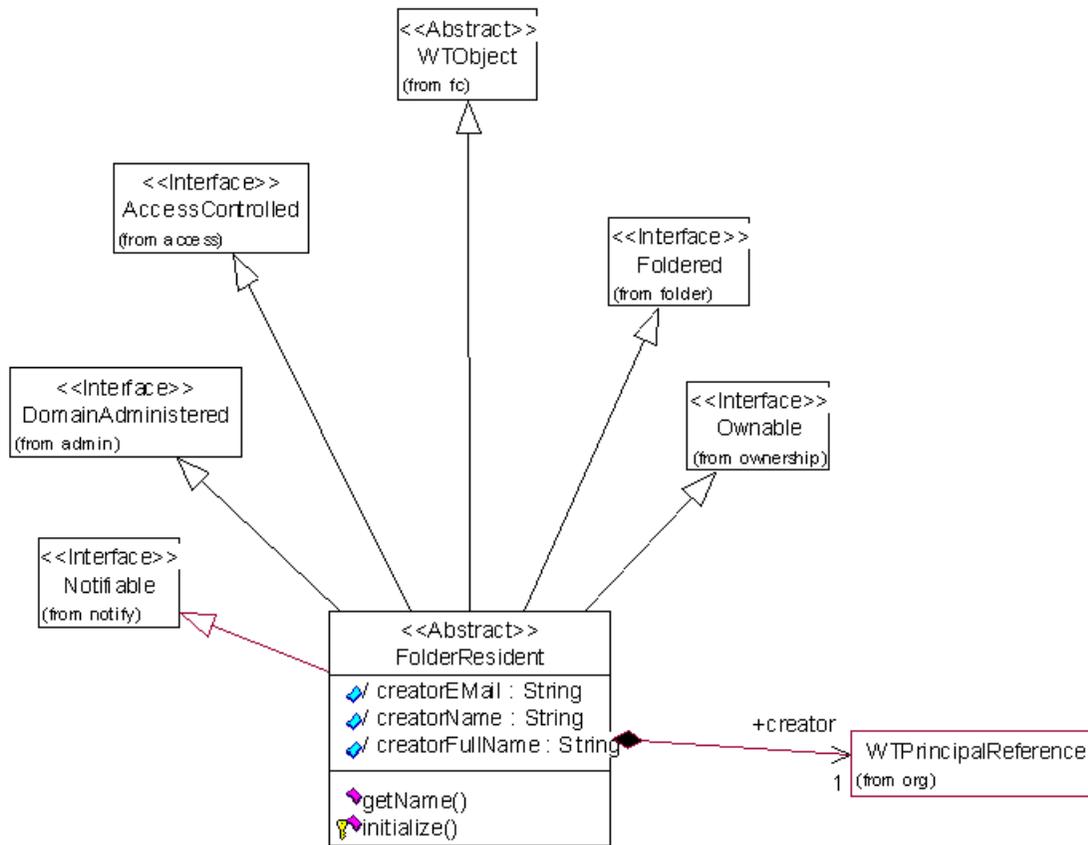


A Shortcut is an interface that defines an association between a Folder and some CabinetBased information.



Key Concepts of Information Foldersing

The following figure contains a representation of the object model for foldered information.



Folder Model

Typically, new objects that are to be foldered should extend the interfaces provided in the wt.enterprise package. For simple foldered objects, use the FolderResident class shown above. If additional characteristics (such as life cycle management or versioning) are required, use the Managed or RevisionControlled classes.

External Interface

The CabinetBased interface defines characteristics of information organized by the foldering service. It has a name and a location that indicates the path to the object in the cabinet and folder hierarchy.

The Foldered interface is asserted by objects that are to be organized into a folder. An object can reside in (that is, be a member of) one and only one Folder.

The Folder interface is asserted for an object that organizes information. Both a Cabinet and a SubFolder perform this organizing mechanism. There are two types

of Cabinets: personal and shared. A personal cabinet is intended to hold information belonging to (owned by) a specific user. A shared cabinet contains information that is intended for general access by other users/groups in the system (depending on administrative policies).

The FolderHelper is a class that provides access to the functions of the foldering package. Using these methods, an object is associated with a Folder for its initial creation, and can determine its full folder path, its cabinet, and so on. The FolderHelper also provides a member service variable which gives access to the server functionality of the FolderService.

The FolderService interface defines the client-server API of the folder service. These operations are used to determine the contents of folders and move an object from one folder to another, as well as perform operations to assist in programmatically constructing a cabinet and folder hierarchy.

The FolderServiceEvent is an event emitted when an object moves from one folder to another. The intent of the event is to allow other services and managers the opportunity to validate, veto, or perform additional work when an object is moved from one folder to another.

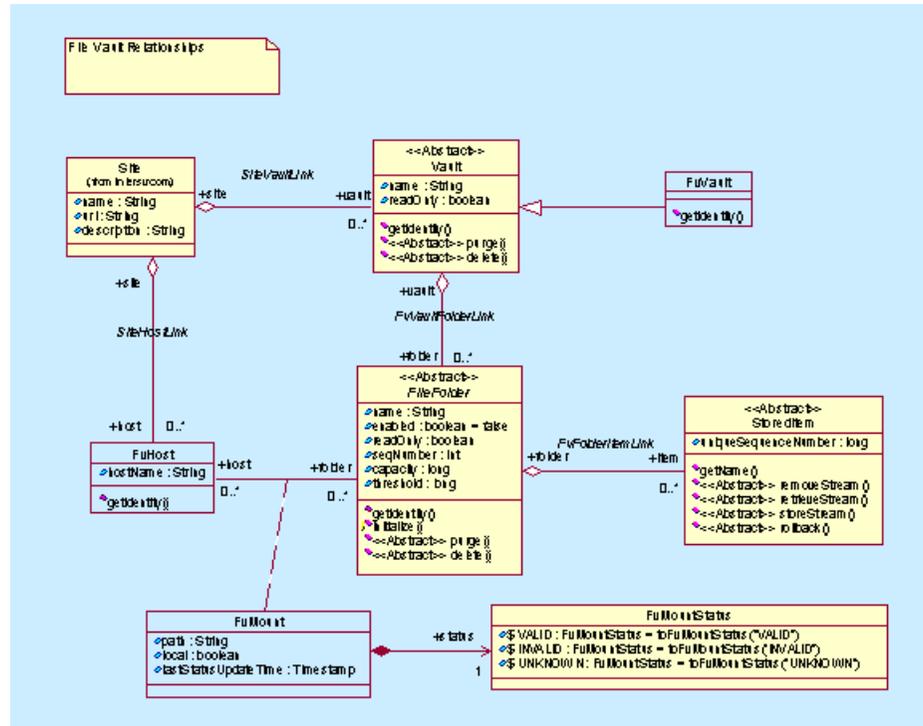
Business Rules

The business rules for foldering are based on the access control that it enforces. The most basic rule is that to put an object into a folder, or remove it from the folder, you must have modify permission on that folder. This is in addition to whatever permission you must directly have to create or delete the object. To move an object from one folder to another, you must have modify permission on the source and destination folders, as well as permission to delete the object from its source folder and to create it in its destination folder. An additional rule is that you can not move an object from a shared cabinet to a personal cabinet.

fv package — File Vault Service

The file vault service (wt.fv package) is responsible for the definition and execution of rules that define the vaulting algorithm for content items. This allows the content service to store content items in external files as opposed to ORACLE BLOBs to speed up the upload and download process.

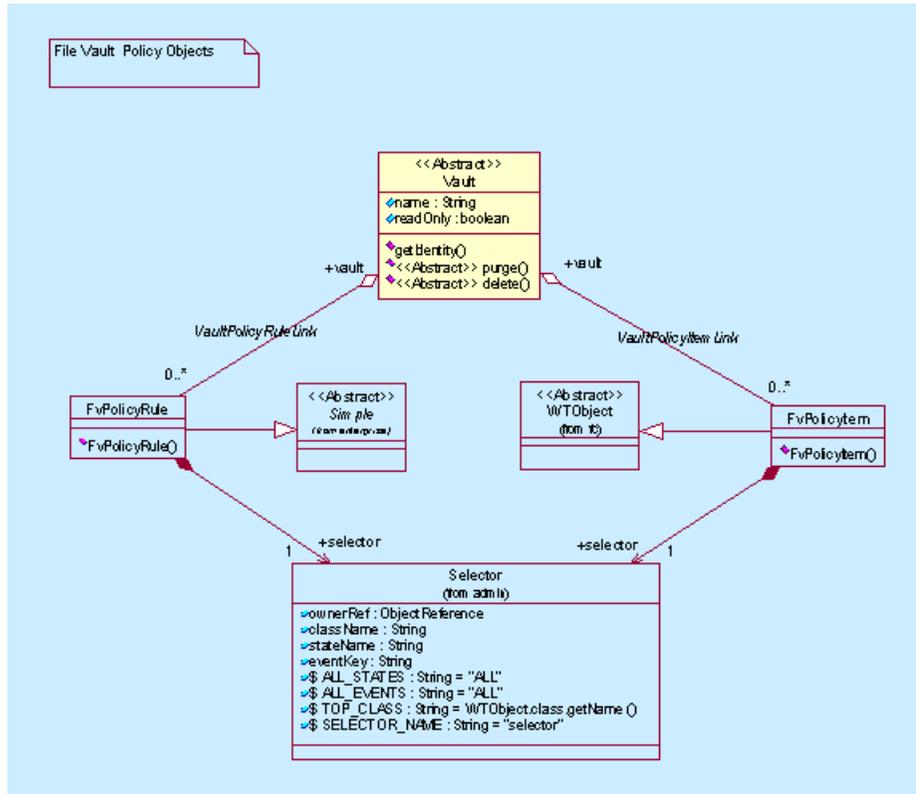
Design Overview



File Vault Model

The file folder is the equivalent of a directory on the disk drive. Folders are logically grouped to form vaults. Each folder has a sequence number that defines the order in which the folders within the vault will be used for storing content items. When a folder overflows, the vault manager makes it read-only and begins using the next available folder in the sequence. All the folders must be mounted to all the method server hosts in order to provide access to content items stored in them. Depending on the domain a content holder belongs to, a class and life cycle state content item is stored in one of the vaults or in the ORACLE BLOB.

The following is the class diagram for vaulting rules:



External Interface

There is no external API for customer use.

Event Processing

The service emits OVERFLOW events with the FvFolder object as a target on folder overflow and the FvVault object as a target on vault overflow. An event is emitted from within a transaction. Vetoing an event or throwing an exception has no effect on the operation that caused the event to be emitted.

identity package — Identity Service

The identity service (wt.identity) provides the classes and interfaces to manage the display identification of business objects. Display identification is the ability to get locale-sensitive, displayable information about an object's identity. An object's identity consists of its type (such as a part, document, or folder) and an identifier (such as a part number, document name, or folder name).

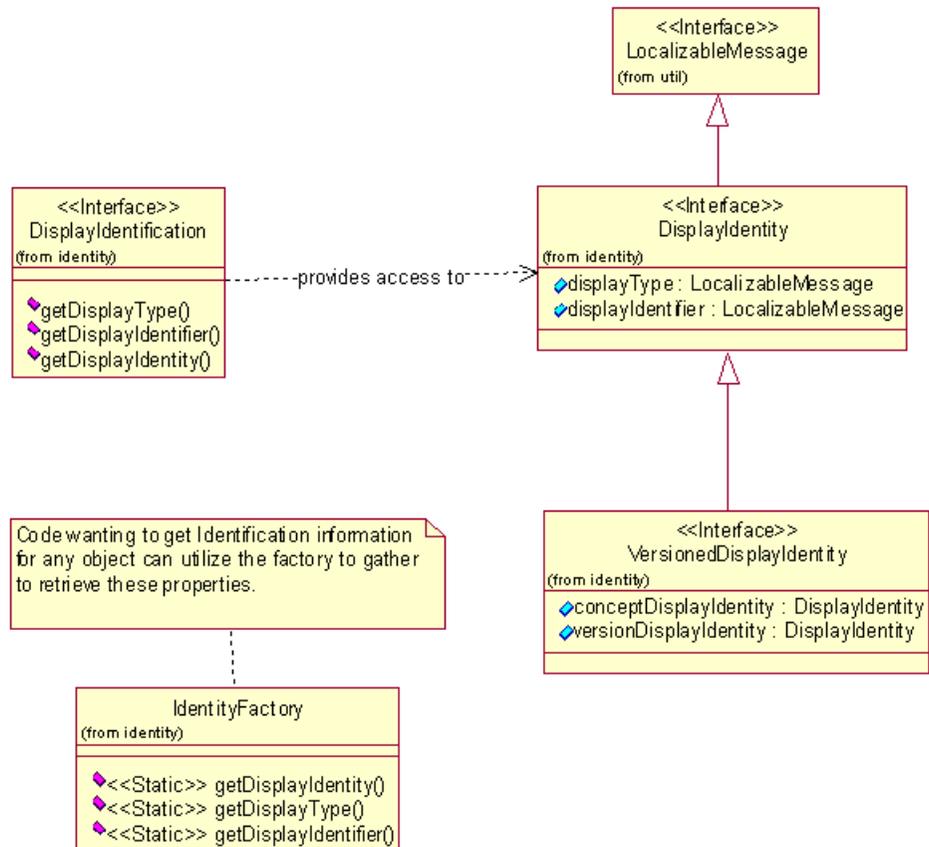
This information must be shown in various clients, both java applets and HTML pages, in a locale-sensitive manner. This package implements some standard

interface specifications for the characteristics of object identification for display purposes, as well as a mechanism to construct new display identification objects for new types of business objects as needed.

The Windchill implementations use resource bundles to manufacture the correct LocalizableMessage for any DisplayIdentity object.

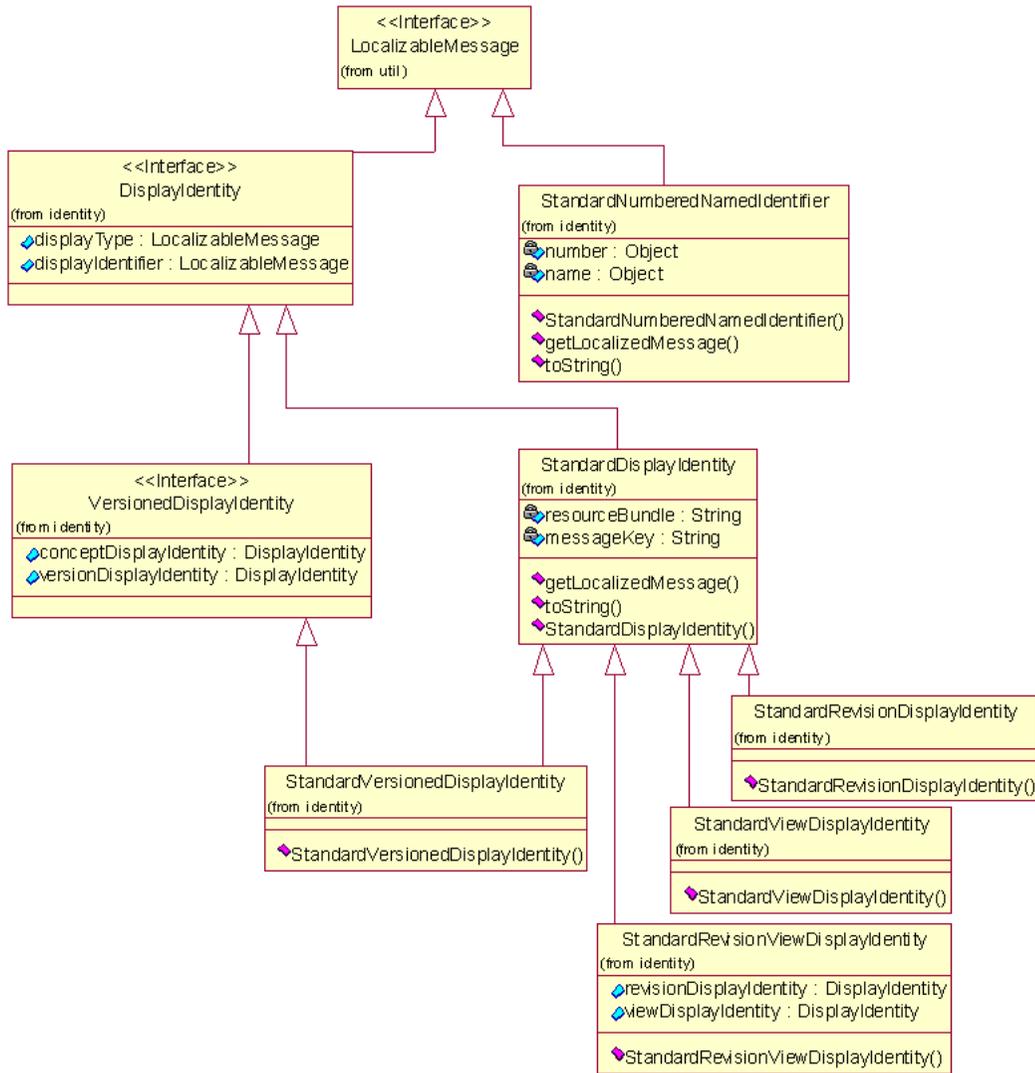
Object Model

The following figure shows the basic model to provide display identification information about business objects. The following figure illustrates the basic specification of identifiability and the identity factory mechanism.



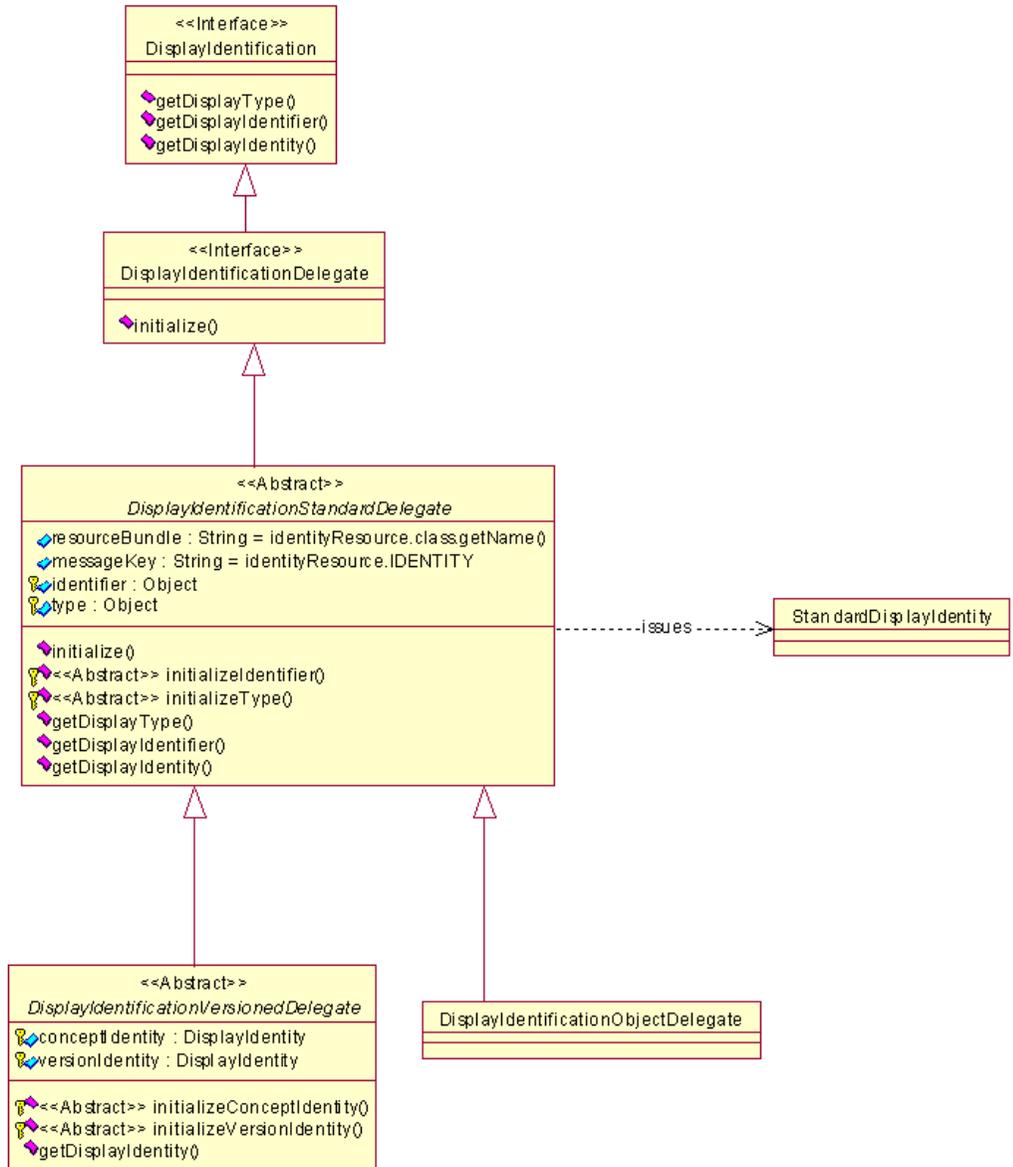
Display Identification for Objects

The following figure illustrates the standard identity type objects.



Standard Identity Type Objects

The following figure illustrates delegate classes that provide identification information objects.



Delegate Classes for Display Identification

These delegates are generally not aware of specific Windchill plug-and-play component characteristics, allowing them to be used with a large variety of object types.

Implementing New Display Identification Delegates

The display identification mechanism is based on delegates that are constructed based on information in the service.properties file. The delegates are constructed based on the model specified above, implementing the abstract methods specified.

The service.properties file

The following entry from the service.properties file specifies that the WTDocument class should use the delegate DisplayIdentificationWTDocumentDelegate:

```
wt.services/svc/default/wt.identity.DisplayIdentification/null/  
wt.doc.WTDocument/2=wt.identity.DisplayIdentificationWTDocumentDelegate/  
duplicate
```

Implementation

When implementing a delegate, the delegate must be a subclass of either the DisplayIdentificationStandardDelegate or the DisplayIdentificationVersionDelegate. The implementer must implement the appropriate initialize methods as determined by the parent class. Typically, the initialize() method itself is not overridden, but rather the initializeXXXX() methods for the specific attributes of the identification object. Note that the delegate objects should be written to be almost foolproof; that is, so that they can always produce a DisplayIdentification object without throwing an exception.

From release 6.0, another option for subclassing delegates is the DisplayIdentificationPersistableDelegate. The initializeType() method in the existing DisplayIdentificationPersistableDelegate has changed from initializing the object using the introspection information, to dispatching the initialization functionality to either "NonTypedDelegate" or "TypedDelegate" based on if the object is "Typed" and let the proper delegate calculate the proper "LocalizableMessage" for the object and set the object's type to its correspondent "LocalizableMessage". The subclass delegate should then implement the initializeType() method by just calling its super's initializeType() method to let the DisplayIdentificationPersistableDelegate handle the work.

Import and Export Package

Windchill Import and Export can assist you in moving complete Windchill content and metadata to and from Windchill sites and Windchill ProjectLink portals by placing the data in Jar files.

Windchill Export places in Jar files on your file system all the data held in high-level Windchill objects in the local Windchill database. Windchill Import extracts such Jar files to the local Windchill database.

A DTD for core Windchill objects could be of interest if a problem arises in importing an object and troubleshooting is needed. The DTD could be of interest

to people who wish to publish from Windchill into a different XML format. Modifying the DTD could allow that operation.

A commented DTD for core Windchill objects is in the following location:

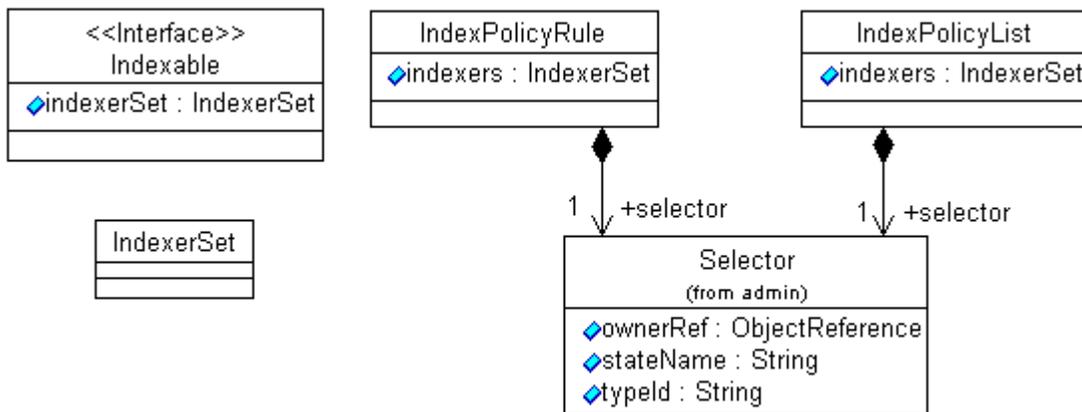
```
windchill\codebase\registry\ixb\dtds\standard\coreobjects.dtd
```

index package — Indexing Service

The index policy manager (wt.index package) is responsible for the definition and execution of rules that automate index maintenance. To allow the global search of business objects, those objects are indexed in appropriate external search engine indices. A collection or library is a gathering of information that is indexed by a search engine and available for searching. At any time, a business object can belong to one or more collections. Depending on the occurrence of events, such as whether the object has been released, the system may have to add or remove objects from the collections. Even if the object stays with the collection, it may have to be re-indexed to ensure that the most up to date information describing the object is indexed.

Design Overview

The design of the index policy manager follows the same pattern as the other administrative policies (see Access control and Notification service, also in this chapter). The Indexable interface marks those classes whose objects may be indexed in external collections. Every Indexable object holds a list of the collections (IndexerSet) in which it is currently indexed. Access to this list is possible through the static methods of the IndexerHelper class.



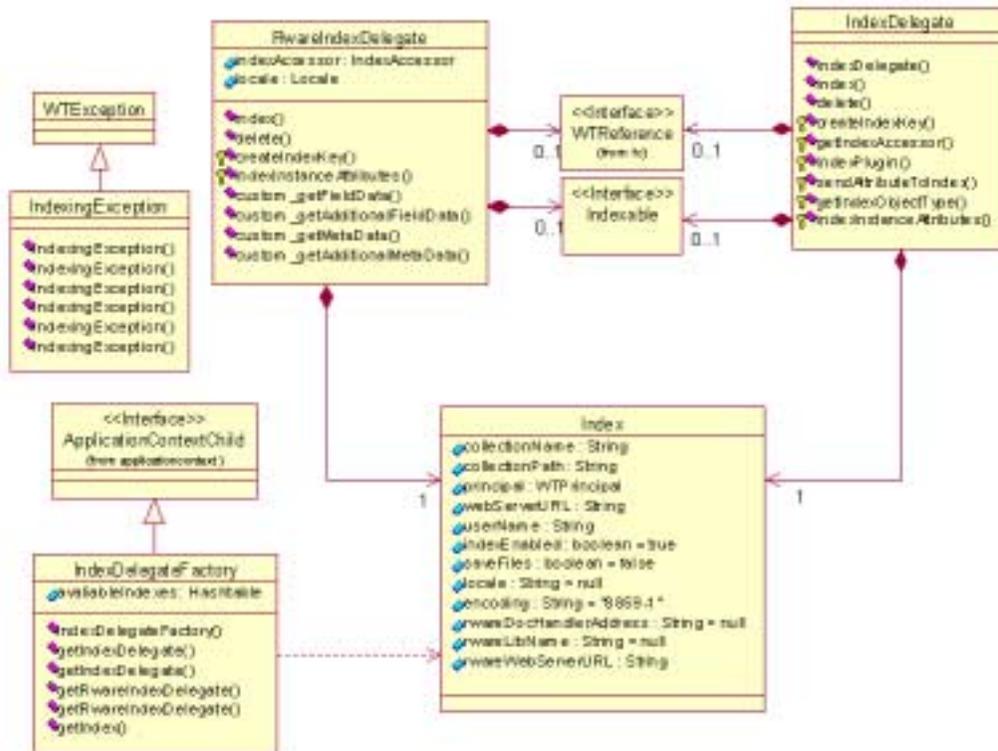
Indexing Model

For each type and state, it is possible to define the collections the object should be in. This definition is called an index policy rule and is represented by the IndexPolicyRule class. Indexing policy rules are enforced by indexing policy lists (IndexPolicyLists class). The lists are derived from all of the rules that apply to a

domain, type and state. The rules and lists contain a Selector object (domain/type/state) and a set of indexers. When an event occurs to an Indexable object, the indexer set in the policy list is the union of the collections for all the applicable rules. Index policy lists are created on demand and for performance reasons, stored persistently in the database and in a server cache. The set of indexers in the policy list is used to update the object's indexer set.

Besides providing methods for managing policies, the IndexPolicyManager also listens to events and triggers collection updates. These are not performed immediately but queued for asynchronous execution.

The following figure shows the classes that map the attributes of Windchill business objects to the Verity collection.



Collection Attributes

The default `RwareIndexDelegate` indexes every attribute, every instance-based attribute (IBA), the identity of every object reference connected with the business object (that is, the owner), the text of content files, and the links and descriptions of attached URLs. By extending `RwareIndexDelegate` and overriding the `index` method, information can be indexed in a way other than the default.

Note: Versions of Windchill prior to 5.0 used only the IndexDelegate and classes inheriting from IndexDelegate for indexing objects. At release 5.0 the search engine that is distributed with Windchill changed from Verity Information Server to Excalibur RetrievalWare and there was a need to introduce the RwareIndexDelegate class. This is so that objects can continue to be indexed to Verity using the IndexDelegate while RetrievalWare indices are being created using the RwareIndexDelegate class.

External Interface

The Index Policy Manager methods can be accessed through the IndexPolicyHelper class.

Business Rules

Although the indexing mechanism can use any event posted by any service, the implementation provided listens only to events that may cause the object's index entry to become out of sync with the object. This list of events ensures maximum efficiency and ensures that index entries always remain up to date.

There is no need to create a rule that removes the object from the collections in the event that the object is deleted. The Index Policy Manager creates an implicit rule that does that.

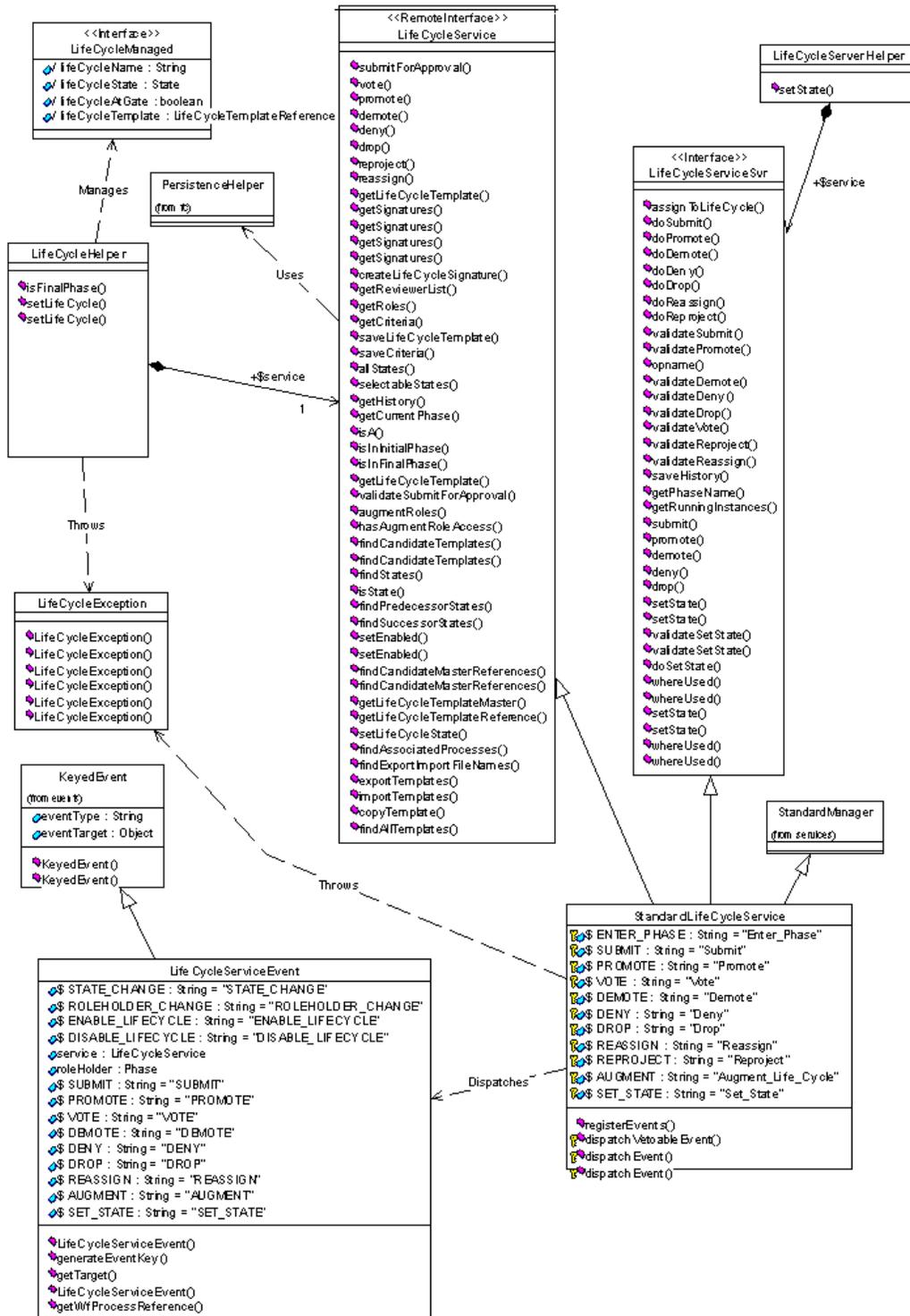
Event Processing

```
/wt.admin.AdministrativeDomainManagerEvent/POST_CHANGE_DOMAIN  
/wt.vc.wip.WorkInProgressServiceEvent/POST_CHECKIN  
/wt.fc.IdentityServiceEvent/POST_CHANGE_IDENTITY  
/wt.fc.PersistenceManagerEvent/POST_DELETE  
/wt.fc.PersistenceManagerEvent.POST_STORE  
/wt.fc.PersistenceManagerEvent/POST_MODIFY  
/wt.folder.FolderServiceEvent/POST_CHANGE_FOLDER  
/wt.content.ContentServiceEvent.POST_UPLOAD  
/wt.lifecycle.LifeCycleServiceEvent/STATE_CHANGE  
/wt.vc.sessioniteration  
.SessionIterationEvent/POST_COMMIT_SESSION_ITERATION  
/wt.vc.VersionControlServiceEvent/NEW_ITERATION  
/wt.vc.VersionControlServiceEvent/PRE_SUPERSEDE
```

lifecycle package — Life Cycle Management Service

The life cycle service resides in the package `wt.lifecycle`. It provides functionality to manage the states that information passes through, the transitions required to move from state to state, and the behavior associated with an object while it is in a specific information state.

Design Overview



Life Cycle Model

The life cycle service is designed to be a plug-and-play component in the Windchill system and is intended to be used for both client and server development. Business objects, asserted as being `LifeCycleManaged` in the object model, are assigned a state at creation and can be promoted through the defined phases of an associated life cycle. Life cycle state information is held in a state cookie. The business object should not interact directly with the state cookie, but instead operate on it through the life cycle service's external interface.

External Interface

The `LifeCycleManaged` interface provides an abstraction of a plug-and-play component. The intent is that, in an object model, a business object would assert that it is `LifeCycleManaged` by inheriting (that is, it implements) the `LifeCycleManaged` interface. With this assertion, the business object can be transitioned to successive states as defined by a life cycle definition.

The `LifeCycleHelper` provides an abstraction as the API to the life cycle service. The API's methods can be categorized as either local or remote invocations. The local methods are getters of information, typically from cookies that are held in the business object. The remote methods serve as wrappers to a service that promotes server-side functionality.

The `LifeCycleServerHelper` provides an abstraction of the server-side API to the life cycle services. These methods can be invoked only from server-side processing.

The `LifeCycleService` provides an abstraction that specifies and promotes server-side functionality as a service that is remotely available for use by a client. This interface is intended to define all the necessary server-side functionality for life cycle management.

The `LifeCycleServiceEvent` provides an abstraction of a specialized keyed event used by the life cycle service to signal other services that something has occurred. This gives other services in a plug-and-play architecture the opportunity to act accordingly upon these events. Validation, vetoing, and post-processing are typical reactions to events.

The life cycle service emits the following events:

STATE_CHANGE

Emitted when an object's state changes. This occurs at creation, promotion, or demotion of an object.

ROLEHOLDER_CHANGE

Emitted when the role participants of a role are modified. This occurs when the object is dropped or reassigned to a different life cycle.

ENABLE_LIFECYCLE

Emitted when a life cycle is enabled.

DISABLE_LIFECYCLE

Emitted when a life cycle is disabled.

SUBMIT

Emitted when a life cycle managed object is submitted to the gate.

PROMOTE

Emitted when a life cycle managed object is promoted to the next phase.

VOTE

Emitted when a reviewer or promoter votes.

DEMOTE

Emitted when the life cycle managed object is demoted to the previous phase.

DENY

Emitted when a life cycle managed object is denied (that is, it is moved from the gate back to the current phase).

DROP

Emitted when a life cycle managed object is dropped (that is, it is no longer associated with a life cycle).

REASSIGN

Emitted when a life cycle managed object is reassigned to a different life cycle.

AUGMENT

Emitted when a life cycle role participant list is updated.

SET_STATE

Emitted when the Set Life Cycle State action is performed.

The LifecycleException provides an abstraction of an abnormal occurrence or error in the usage or processing of the life cycle service. This exception can be localized through a given resource bundle, and other exceptions can be nested within it.

Business Rules

Life cycle actions on life cycle managed objects are authorized by role assignments. Roles are associated to principals in the definition of a life cycle phase and a project, and are specific to a life cycle phase. Resolvable roles and default assignments are identified in the definition of a life cycle phase. At runtime, these roles are resolved to principals from a project specification, if the corresponding roles are designated in the project.

There are four standard roles. The Submitter role players have authority to submit the object for review. The Reviewer role players have authority to enter comments and a signature designating approve or reject. The Promoter role players have authority to promote, demote, deny, or drop the object. The Observer role players have the right to view the object and view reviewer comments and signatures. Additional roles can be added by updating the role resource bundle.

Event Processing

The life cycle service is an event listener. The service listens for and acts upon the following standard Windchill events:

- When a `PRE_STORE` event is emitted for a life cycle managed object, the life cycle service initializes the state cookie by assigning an initial state to the object. The save of a life cycle is vetoed if the folder location is not a personal cabinet or the System cabinet.
- When a `POST_STORE` event is emitted for a life cycle managed object, the life cycle service associates phase information, such as role players and access rights, to the object.
- When a `CLEANUP_LINK` event is emitted for a life cycle managed object, the life cycle service completes its portion of the delete action by removing the life cycle associated data. When emitted for a life cycle, the life cycle service completes its portion of the delete action by removing the `PhaseLink`, `AdHocAcILink`, and `DefaultCriterion` links.
- When a `PRE_DELETE` event is emitted for a life cycle, and the template is in use by a life cycle managed object, the delete is prohibited. When the object being deleted is a `WfProcessTemplate` or a `WfProcessTemplateMaster`, and that `WfTemplate` is referenced by a phase or gate of the life cycle, the delete is prohibited.
- When a `PRE_CHECKIN`, `PRE_CHECKOUT`, or `PRE_MODIFY` event is emitted for a life cycle managed object, the life cycle service prevents checkin, checkout, or modification of the object, if that object is pending promotion (that is, the object is at the gate).

The checkin of a life cycle is not allowed when that life cycle is in use. Life cycle managed objects that are created against a working copy of the life cycle are assumed to be test objects. The checkin is prevented to ensure that the test objects are cleaned up.

When a life cycle is checked out, the objects and links associated with the life cycle are copied.

Once you create a life cycle managed object that references a working copy of a life cycle, changes to the working copy the life cycle are no longer allowed. You must delete the life cycle managed objects that reference the working copy of the life cycle before you can make additional changes to that working copy.

- Life cycle managed objects are project managed objects. When the `REPROJECT` event is emitted for a life cycle managed object (that is, the object is reassigned to a new project), the life cycle service updates the data associated with this change.
- When a `POST_ROLLBACK` event is emitted for any life cycle managed object (that is, an object is rolled back to a previous iteration), the life cycle

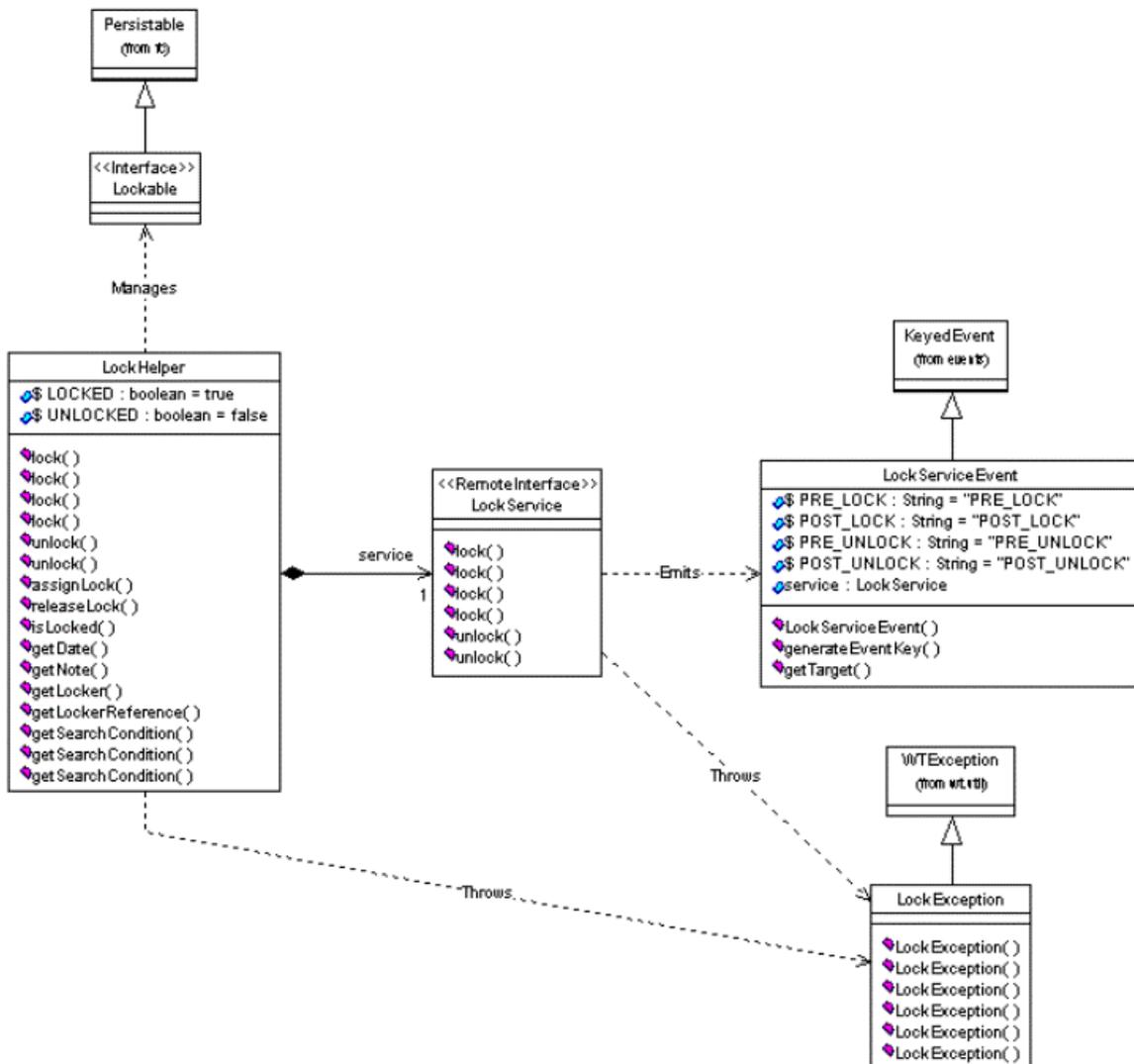
service backs up the state of the object to match this old iteration. Also, the history associated with these now obsolete iterations is also removed.

- When a POST_CHANGE_DOMAIN event is emitted, any attempt to move the life cycle to a location other than the personal cabinet or the System cabinet is vetoed.

locks package — Locking Service

The locking service (wt.locks package) provides functionality to place and release in-memory and/or persistent locks on lockable objects. The persistent locks are not to be confused with database locks; they are two separate mechanisms.

Design Overview



Locking Service Model

The locking service is designed to be a plug and play component in the Windchill system, which is by default enabled to execute.

The locking service is intended to be used for both client and server development. Business objects, asserted as being lockable in the object model, can be locked and unlocked through the locking service's external interface. Once a business object is lockable, its lock can be seized to prevent concurrent access. Once the lock has been released, the business object is available to be locked again. A lockable object is not required to be locked prior to modification but, if it is locked, the lock is honored. The lock itself is a cookie that a lockable business object aggregates. The business object should not interact directly with the lock cookie, but instead operate on it through the locking service's external interface.

External Interface

The Lockable interface provides an abstraction of a plug and play component. The intent is that, in an object model, a business object would assert that it is Lockable by inheriting (that is, it implements) the Lockable interface. With this assertion, the business object can then be locked and unlocked.

The LockHelper provides an abstraction as the API to the locking service. The API's methods can be categorized as either local or remote invocations. The local methods are getters of information, typically from cookies that are held in the business object. The remote methods serve as wrappers to a service that promotes server-side functionality.

The LockService provides an abstraction that specifies and promotes server-side functionality as a service that is available remotely for use by a client. The intent is that this interface defines all the necessary server-side functionality for locking.

The LockServiceEvent provides an abstraction of a specialized keyed event used by the locking service to signal other services that a locking activity is about to begin or has occurred. This gives other services the opportunity in a plug and play architecture to act accordingly on these events. Validation, vetoing, and post-processing are typical reactions to events.

The LockException provides an abstraction of an abnormal occurrence or error in the usage or processing of the locking service. This exception can be localized through a given resource bundle, and other exceptions can be nested within it. The most common occurrence of this exception is an attempt to lock or unlock a business object that is already locked or unlocked.

Business Rules

As specified by the locking service's standard access control rules, when an attempt is made to lock an object, if it is not already locked and the given principal has modify access to the object, then it is locked. If an attempt is made to unlock an object that is already locked and the given principal has administrative access, or the given principal is the one who originally placed the lock and has modify access to the object, then it is unlocked. Otherwise, an exception is thrown

indicating the failure. Additionally, when a lock is placed on an object by a principal, that user or group is the only principal able to then modify the object while it is in a locked state.

Event Processing

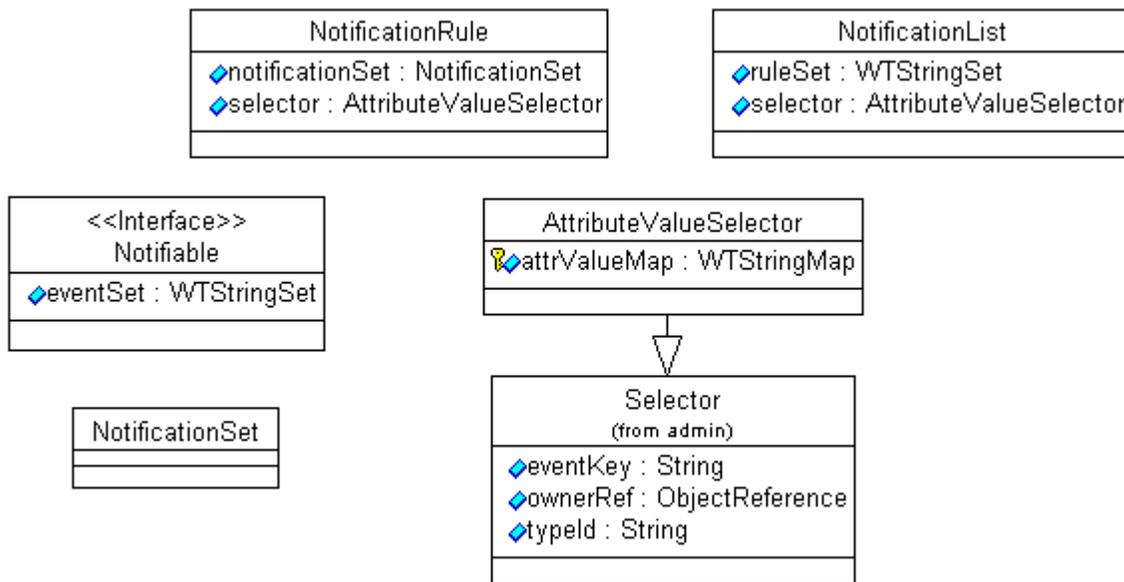
Event-based processing is performed on business objects asserted as being lockable during [preparation for] database modifications and full restorations. When a business object is [prepared for] being modified in the database, the locking service listens to a dispatched event indicating that the modify is about to begin and vetoes it if the business object is locked and the current session's principal is not the one who originally placed the lock. Otherwise, the modification is allowed to take place. Therefore, it is valid to modify a business object that is lockable if it is not locked. When a business object is being fully restored from the database, the locking service listens to a dispatched event indicating that the full restoration is beginning and restores the principal who holds the lock in the business object's lock cookie.

notify package — Notification Service

The Notification manager (wt.notify package) performs the definition and execution of notification. The notification can be defined through a notification policy or an object subscription and triggered by an event.

Design Overview

The Notification policy design follows the pattern for the administrative policy. The Notifiable interface is created so that it can be implemented by the classes for which notification should be sent. This interface contains no method. A notifiable object may hold an event set that is specific to the object.

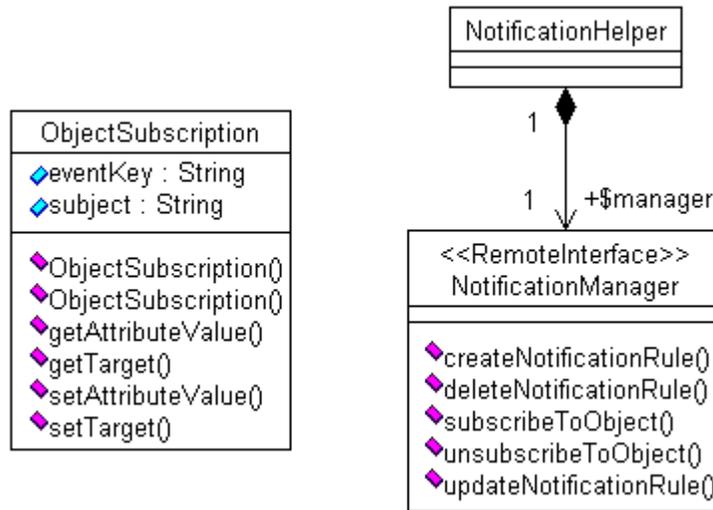


NotificationPolicy

A Notification Policy is composed of many rules: NotificationRule objects. Each rule contains an antecedent: a Selector object composed of a domain, type, and event, and a consequent: a NotificationSet object (identifying the principals to receive the notification). Similarly, each notification list contains a selector (that works as a locator for the list) and a rule set. The difference between lists and rules is that lists are entirely derived from the rules by evaluating all rules that may possibly apply to a selector (for example, a rule that refers to a more general type also applies to derived type). Notification lists, once constructed, are stored persistently and cached for performance reasons. The message sent as a result of a notification policy is very simple. It basically informs the recipient that the event that triggered the notification occurred to the object in question.

The NotificationManager interface supports the notification functionality with methods for managing rules as well as methods for event subscription. Besides providing these methods for storing and retrieving lists, the StandardNotificationManager also listens to events and triggers notifications based on the notification policy lists. These are not performed immediately but rather queued for asynchronous execution.

Notification can be sent to users either as a result of a notification policy or an object subscription created by a service or application. For example, the properties page for some objects contains a subscription menu item that allows users to subscribe to events on the object. The following figure shows the main classes involved in ad hoc notifications.



Ad Hoc Notification

The object subscription class supports ad hoc notification.

External Interface

The Notification Manager methods can be accessed through the NotificationHelper class.

Business Rules

The users that are the final recipients of the messages must have an e-mail attribute defined. Additionally, for messages generated by notification policies, the user must have read access over the object to which the event occurred.

Although the notification policy mechanism can use any event posted by any service, in practice the notification is limited to the events listed in the wt.admin.adminEventResource class because these are used by the administrator

client to construct rules. The events must also be listed in the notify.properties file.

Event Processing

No event is generated. However, this service listens to the events specified in the notification policy rules.

The list of events for the notification service includes the following in addition to events defined in notify.properties:

/wt.admin.AdministrativeDomainManagerEvent/POST_CHANGE_DOMAIN

/wt.fc.PersistenceManagerEvent/POST_DELETE

/wt.fc.PersistenceManagerEvent/PRE_DELETE

/wt.vc.VersionControlServiceEvent/NEW_ITERATION

/wt.vc.sessioniteration.SessionIterationEvent/POST_COMMIT_SESSION_ITERATION

/wt.vc.wip.WorkInProgressServiceEvent/POST_CHECKIN

External Interface

The Ownable interface provides an abstraction of the ownership capability. The intent is that in an object model, an object will assert that it implements the Ownable interface, therefore allowing it to be owned.

The OwnershipHelper provides access to the API for operation on Ownable objects. The Ownable interface can be extended to create your own business object classes, but the Ownership package does not provide any supported API's to directly manage those objects.

The OwnershipService defines the server-side functionality available to Ownable objects. These operations offer complete transactions that implement the business rules for Ownable objects.

The OwnershipServiceEvent provides the mechanism to inform other plug and play components of the system of operations that impact Ownable objects. This allows other services or managers to perform validation, vetoing, or other pre/post processing related to ownership activities.

Business Rules

Ownership is typically asserted on a business object prior to being stored in the database. For example, in the Windchill implementation, this is done for foldered objects as part of putting the objects into folders; the owner for the folder becomes the owner for the object. Ownership may change as an object moves from one folder to another, as an object moves from one folder to another.

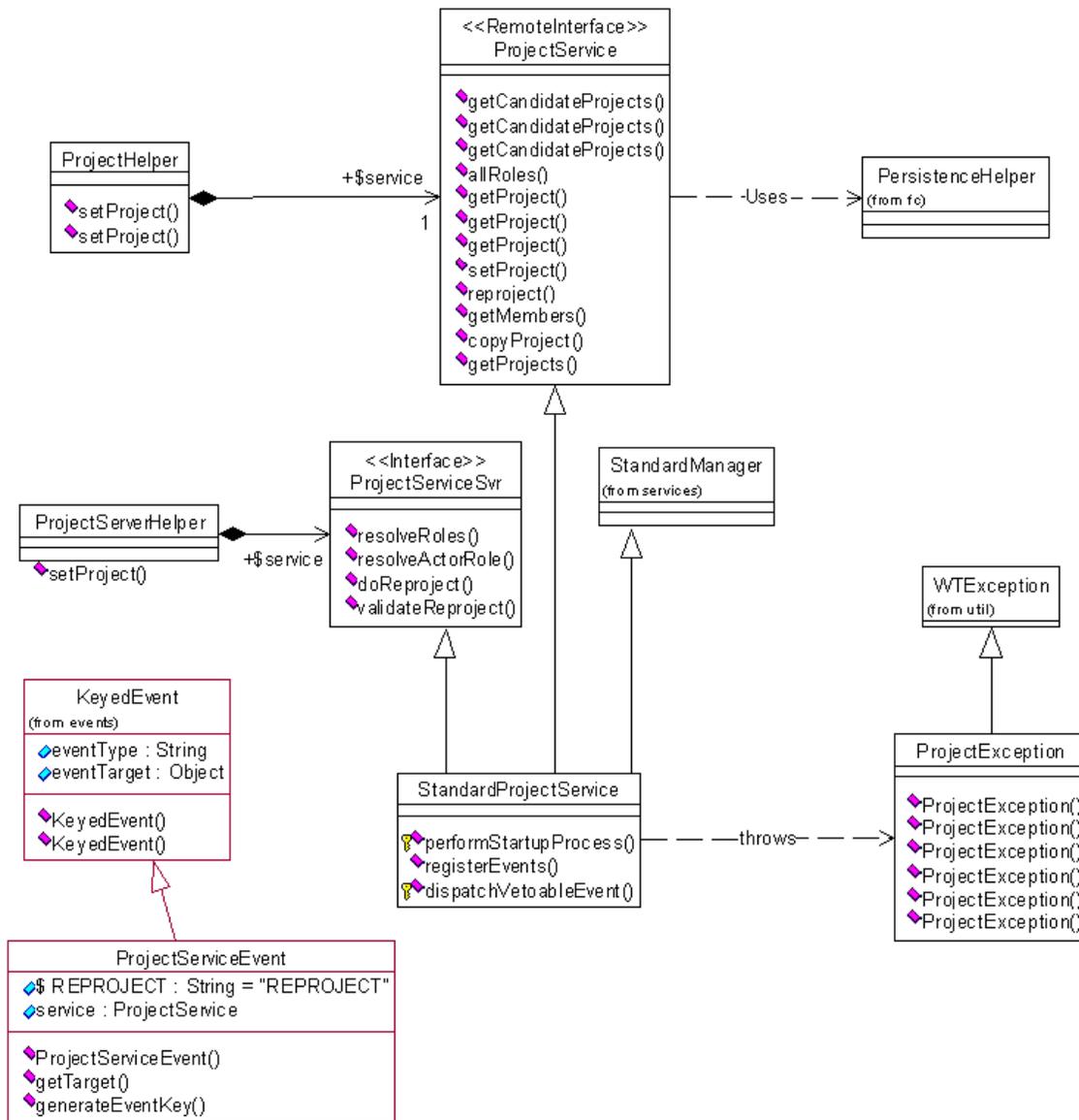
Event Processing

Event-based processing is performed on Ownable objects as ownership changes take place. An event is emitted just before the ownership change takes place, inside the transaction that makes the change, and another event is emitted just after the change has been made to the database. These events are PRE_CHANGEOWNER and POST_CHANGEOWNER.

project package — Project Management Service

The project service resides in the package wt.project. It provides functionality to create projects, associate projects to business objects and resolve roles to principals.

Design Overview



Project Model

The project service is designed to be a plug-and-play component in the Windchill system and is intended to be used for both client and server development. Business objects, asserted as being `ProjectManaged` in the object model, can be assigned a project at creation. Project information is held in a `ProjectId` cookie. The business object should not interact directly with this cookie, but instead operate on it through the project service's external interface.

External Interface

The ProjectManaged interface provides an abstraction of a plug-and-play component. The intent is that, in an object model, a business object would assert that it is ProjectManaged by inheriting (that is, it implements) the ProjectManaged interface.

The ProjectHelper provides an abstraction as the API to the project service. The API's methods can be categorized as either local or remote invocations. The local methods are getters of information, typically from cookies that are held in the business object. The remote methods serve as wrappers to a service that promotes server-side functionality.

The ProjectServerHelper provides an abstraction of the server-side API to the project services. These methods can be invoked only from server-side processing.

The ProjectService provides an abstraction that specifies and promotes server-side functionality as a service that is remotely available for use by a client. The intent is that this interface defines all the necessary server-side functionality for project management.

The ProjectException provides an abstraction of an abnormal occurrence or error in the usage or processing of the life cycle service. This exception can be localized through a given resource bundle, and other exceptions can be nested within it.

Event Processing

The LifeCycleService is an event listener. The service listens for and acts on four standard Windchill events.

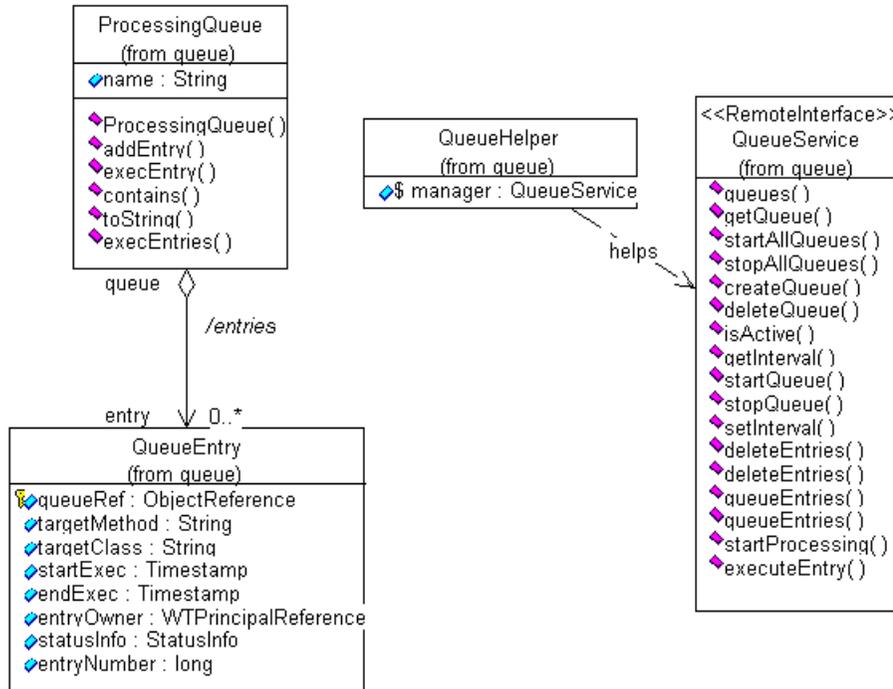
- When a PRE_STORE event for a ProjectManaged object is emitted, the LifeCycleService will initialize the state cookie by assigning an initial state to the object.
- When a PRE_DELETE or PRE_CHANGE_IDENTITY event for a ProjectManaged object is emitted, the service prevents the deletion or name change.
- When a PRE_SUPERCEDE event for a ProjectManaged object is emitted, the service manages the cookie information. The cookie information of the superseded iteration is copied to the superseding object.

The project service emits a REPROJECT event when the project specified in a ProjectManaged object is changed.

queue package — Background Queuing Service

The Queue service (package wt.queue) is responsible for creating, executing, monitoring, and managing queues for processing requests.

The queues created in this service store processing requests in the form of Java method calls. The requests are stored as queue entries and are executed following a first-in first-out (FIFO) schedule.



Queue Service Classes

Queues are named and may contain many entries. Each entry may be in one of the following states:

- READY**
Not yet executed.
- EXECUTING**
Executing.
- COMPLETED**
Executed successfully.
- FAILED**
Exception occurred during execution.
- SUSPENDED**
Operator suspended.

The main classes in this service and their relationships are shown in the figure above. The ProcessingQueue class represents named queues created by the applications or other services. Processing queues can be in one of two states: active, in which requests are executed, and inactive, in which requests aren't executed (but new entries can be added). It supports methods to add and execute

entries. The QueueEntry class holds specific information about the processing request and its execution.

External Interface

The external interface of the QueueService can be accessed through the QueueHelper.manager object. It provides methods to retrieve queues and associated information, set polling intervals, change queues from active to inactive (start/stop), create and delete queues and entries, and even execute specific queue entries.

Business Rules

The only two requirements for an arbitrary Java method to be able to be queued are that it be public and static. Additionally, the programmer must provide a principal (user or group) on behalf of whom the request will be executed. If the method returns a value, the value is lost. Finally, the methods should preferably not require operator intervention and certainly should not depend on end user interaction.

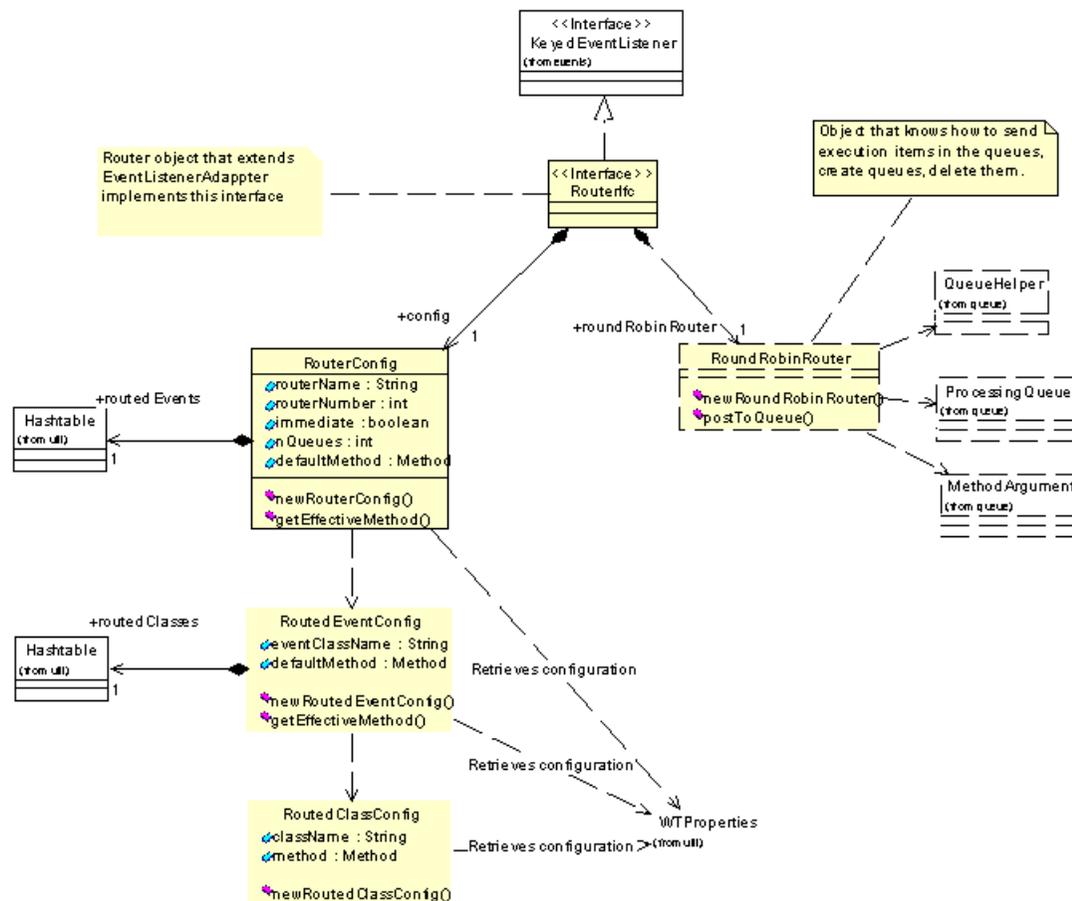
Event Processing

No events are posted by this service.

router package — Routing Service

The routing service is designed to aid in scalability for different agent services by providing a convenient way to distribute execution of tasks to multiple Windchill method servers. The persistent routing service uses standard Windchill queue service and uses the scheduler service on demand.

Design Overview



Router Model

The routing service is a keyed event listener which listens for `RoutingEvent` objects. The association between a `routerEvent` and the corresponding router object determines how the event is handled. These elements are registered in the `wt.properties` file.

A router object can operate in two modes: immediate and scheduled. Immediate mode router objects, upon catching an event they are registered to handle, immediately create and send an execution item to the queue. Scheduled mode router objects, upon catching an event they are registered to handle, store the data in a to-do list. The `StandardSchedulingService` invokes the static method `processToDoList` defined in the scheduled mode router object. This method scans the to-do list, creates execution items, and sends them to queues.

Resource intensive operations should use scheduled mode; non-resource intensive applications should use immediate mode. The ultimate choice depends on the design intent of your system as a whole.

A RoutedEvent object is a subclass of KeyedEvent. Extend this class to classify the events and to define router actions for each of them. The wt.properties file entries specify the method name but not its signature. All invoked methods must be of the following form:

```
static void a_method( Serializable targObject, Serializable[ ] args);
```

The following table describes various property names.

Property Name	Description
Wt.router.X, where X = 1,2...*	Router name. The router names must be sequential and in ascending order.
ROUTER_NAME.immediate = false	Mode of the specified router (where ROUTER_NAME conforms to the router name syntax). Default is false; that is, scheduled mode.
ROUTER_NAME.NumOfQueues = 1	Number of queues used by the specified router. Default is 1.
ROUTER_NAME.method	Default fully-specified method name for all the events caught by the specified router (for example, wt.router.1.myroot.mypack.myClass.myMethod).
ROUTER_NAME.event.Y, where Y = 1,***	Fully-specified event class caught by the specified router.
ROUTER_NAME.event.Y.method, where Y = 1.*	Default fully-specified method name for event Y caught by the specified router.
ROUTER_NAME>event.Y.class.Z, where Y and Z are 1...*	Fully-specified target-object class caught with event Y by the specified router.
ROUTER_NAME.event.Y.class.Z.method	Fully-specified method name for event Y and target-object class Z caught by the specified router.

The names of all the queues started by the persistent router service are defined as follows:

wt.router.X.Y:

wt.router.X

The name of the router.

Y

A sequence number that ranges from 1 to the value of the NumOfQueues property associated with the router name.

Example

This example shows how you can use this service by following these four steps:

1. Define an event:

```
Class SampleEvent extends wt.router.RoutingEvent
{
    public SampleEvent(String evtType, Serializable eventTarget)
    {
        super(evtType, eventTarget);
    }
}
```

2. Define a method to invoke:

```
Class SampleClass
{
    . . .
    static void aSampleMethod(Serializable targetObject){
        //some code to save the world}
}
```

3. Emit the event:

```
. . .
SampleEvent se = new SampleEvent("MY_SAMPLE_EVENT",
    a_serializableObject);
getManagerService().dispatchVetoableEvent(se,se.getEventKey());
```

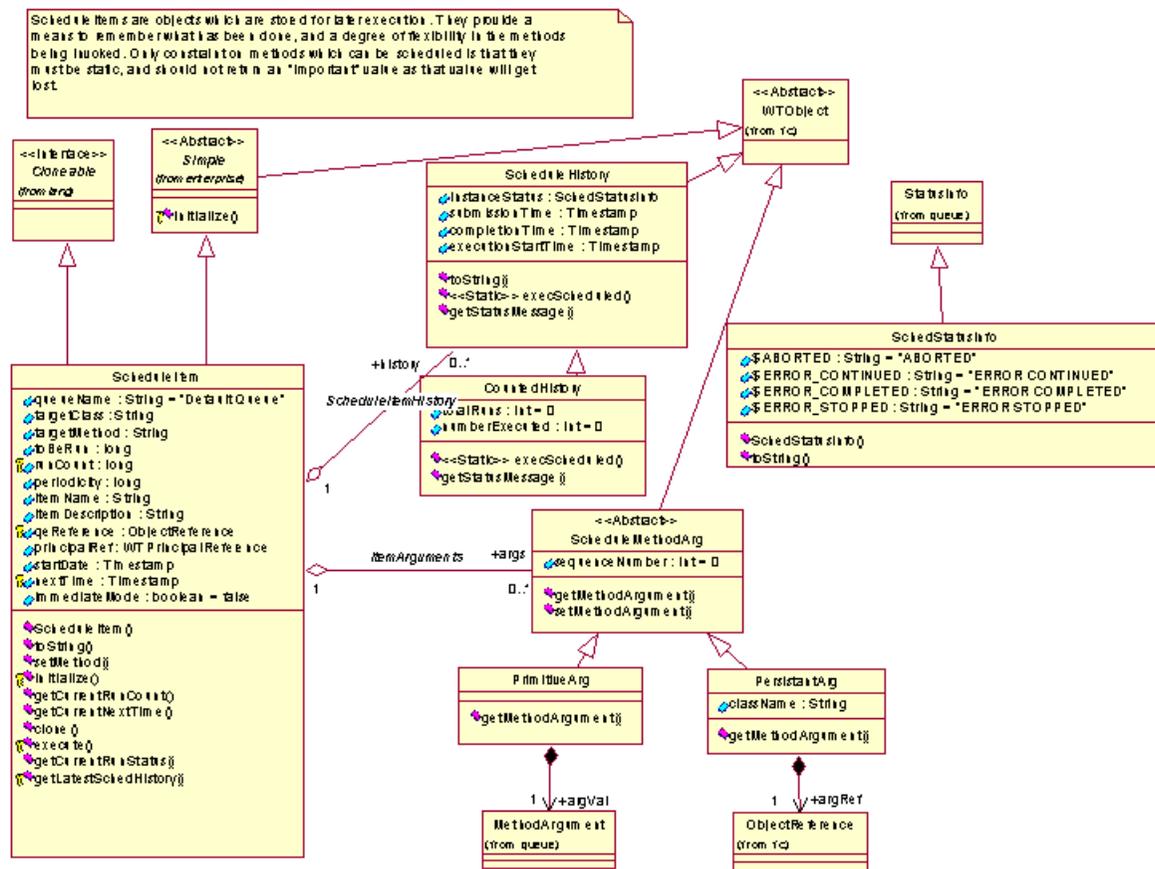
4. Configure the router by placing the following five lines in the wt.properties file:

```
Wt.router.1 = myRouter
myrouter.immediate=false

myRouter.NumberOfQueues=1
myRouter.method=SampleClass.aSampleMethod
myRouter.event.1=SampleEvent
```

scheduler package — Scheduling Service

The scheduling service, wt.scheduler, is responsible for scheduling the execution of resource-intensive method invocations, and keeping a history of their outcomes. It can be used to fire off agent processes, in addition to scheduling a method invocation.



Scheduler Model

The scheduling service manages the schedule items in the system. Schedule items can be scheduled to run once or periodically, and can start immediately or be deferred to start at a later time. Schedule items maintain a collection of their histories and their method arguments. Method arguments can be any object. The scheduler provides a higher-level interface to the wt.queue package. Schedule items are executed with the same privileges as the user who created them.

The ScheduleHistory class provides the history of the execution instances. A default CountedHistory object is supplied for use by agents that perform a number of internal operations. The execScheduled method is provided so that when the

execution fires off an autonomous agent, the agent can record its results to the schedule history.

A running instance can be in the following states:

READY

The item is scheduled and awaiting execution.

EXECUTING

The schedule item is executing its method.

SUSPENDED

The operator has suspended the method.

COMPLETED

The operation has run successfully.

FAILED

The operation has failed.

The statuses that pertain to agent processes are as follows:

ABORTED

The operation has been aborted.

ERROR_CONTUNUED

An error has occurred in the agent session, but the agent has decided to continue.

ERROR_COMPLETED

An error has occurred in the agent session, but the agent process continued to completion.

ERROR_STOPPED

An error has occurred in the agent session and the agent has stopped processing.

External Interface

There is no external API for customer use.

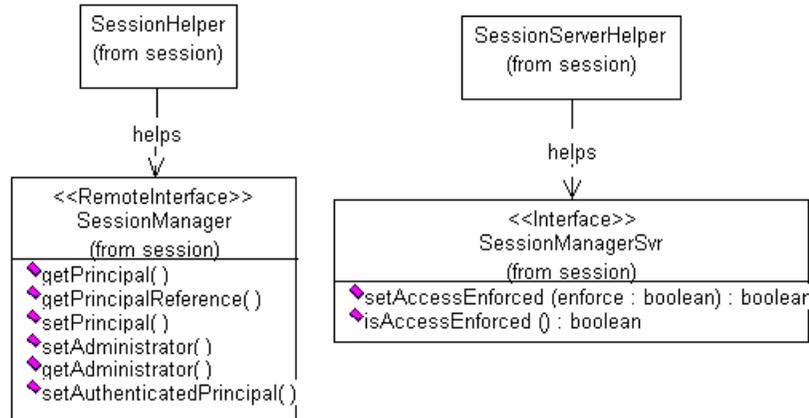
Event Processing

This service does not emit any events.

session package — Session Management Service

The Session manager (wt.session package) maintains information specific to the user session. This includes information about the session user.

The classes shown below are responsible for managing the session user (SessionManager) and setting access control ON or OFF (SessionServerHelper). The first is a remote interface and therefore accessible from the client, while the second is a simple interface and can only be accessed in the server.



Session Queue Service Classes

External Interface

The SessionManager contains methods for setting and getting the current principal. The SessionManagerSvr contains methods for setting/resetting.

Business Rules

Although it is possible to retrieve the current principal from the client, the same cannot be said about setting the current principal. This is only possible if the current principal is itself a member of the Administrators group. Another way is to set the wt.clientAuthenticatedLogin property (in the wt.properties file) to false. This, however, should be done only for development, as it can be a serious security breach.

Event Processing

No events are posted or listened to by this service.

Standard Federation Service doAction() and sendFeedback()

The Standard Federation Service-doAction() Executes one or more Info*Engine tasks selected by a specific logical action name, and the types and physical locations of a specific set of objects passed as parameters. sendFeedback() sends feedback objects to the client.

Design Overview

The Federation Send Feedback mechanism uses the Info*Engine Web Event Service (WES) to enable the sending of Feedback objects to clients. The Info*Engine WES requires a Message Oriented Middleware (MOM) product to be installed, configured and started. More information about MOM can be found in the *Info*Engine Installation and Configuration Guide*.

The MOM has to be configured to recognize the "CommandDelegateFeedback" Event. Actually the Event name only has to be the same as the EVENT parameter value for the "Subscribe-Event" webject in the wt/federation/StandardFederationServicesStartup Info*Engine task and the "Emit-Event" webject in the wt/federation/CmdDelegateFeedbackEmit Info*Engine task. These tasks are shipped with EVENT set to "CommandDelegateFeedback". It's possible to change the value in these tasks to something other than "CommandDelegateFeedback" although I don't know that it is necessary to advertise this.

For sendFeedback() to work, Federation has to subscribe to receive Info*Engine WES Events, specifically the "CommandDelegateFeedback" Event. This is done via the wt/federation/StandardFederationServicesStartup task, which can be executed at Federation Service startup time. For this task to execute during Federation Service startup, the following line must be added to wt.properties:

```
wt.federation.task.startup=wt/federation/StandardFederationServicesStartup.xml
```

If the StandardFederationServicesStartup task is executed during Federation Service startup and no MOM is installed/started, Federation Service startup (and Method Service startup) continue it just means that sendFeedback() won't work if it is called. Also, if the Federation Verbose flag is true, the following message is logged to the Method Server log file when the Federation Service is started:

```
Wed 6/13/01 20:00:53: main:  
wt.federation.StandardFederationService.performStartupProcess() -  
caught IEEException. The nested exception is:  
javax.jms.JMSEException: java.net.ConnectException: Connection  
refused: no further information: tcp://<server>:<port>.
```

The most likely cause is no Messaging Service is started. This is a non-fatal condition. The only consequence is that Command Delegate Feedback is disabled. The stack trace follows.

If the StandardFederationServicesStartup task is executed and the MOM isn't configured with the proper Event (i.e., the CommandDelegateFeedback Event), the following message is logged and the Method Server terminates.

```
Tue 6/19/01 20:28:22: main: *ERROR*:  
(com.infoengine.util.IEResource/115)  
com.infoengine.exception.fatal.IEFatalServiceException: Lookup of  
Administered object with uri "CommandDelegateFeedback" returned  
null. Similar results will occur (i.e., something will be logged  
for the error and the Method Server terminates) if the  
StandardFederationServicesStartup task is executed and there are
```

other MOM configuration errors.

If `sendFeedback()` is called and no MOM is installed/started (regardless of whether or not the `StandardFederationServicesStartup` task was executed), `sendFeedback()` throws a `FederationServicesException` exception and, if the Federation Verbose flag is true, the following is logged to the Method Server log file:

```
Tue 6/19/01 18:57:30: SocketThread0:
wt.federation.StandardFederationService.sendFeedback() caught
exception.
The stack trace follows.
```

If `sendFeedback()` is called without executing the `StandardFederationServicesStartup` task at Federation Service startup time (and a MOM is installed/configured/started), no Feedback message is sent to the client. In this scenario no errors occur on the Server so there are no exceptions from `sendFeedback()` and no log messages in the MethodServer log file. The only indication that something is wrong is that the client isn't receiving Feedback messages.

If `sendFeedback()` is called without executing the `StandardFederationServicesStartup` task at Federation Service startup time (and a MOM is installed/started but not configured with the `CommandDelegateFeedback` Event), `sendFeedback()` throws a `FederationServicesException` exception and, if the Federation Verbose flag is true, the following is logged to the Method Server log file:

```
Wed 6/20/01 14:15:36: SocketThread1:
wt.federation.StandardFederationService.sendFeedback() caught
exception.
The stack trace follows.
```

External Interface

This is a supported API. The corresponding UI in Windchill is the Delegate Administrator.

Event Processing

`doAction` executes one or more Info*Engine tasks selected by a specific logical action name, and the types and physical locations of a specific set of objects passed as parameters. `sendFeedback` sends feedback objects to clients.

doAction

```
public com.infoengine.object.factory.Group
doAction(String action,
Object[][] argv)
throws WTEException
```

Executes one or more Info*Engine tasks selected by a specific logical action name, and the types and physical locations of a specific set of objects passed as parameters.

Example

```
import wt.federation.FederationHelper;
import wt.util.WTException;
import com.infoengine.object.factory.Att;
import com.infoengine.object.factory.Element;
import com.infoengine.object.factory.Group;
import com.ptc.core.util.feedback.common.FeedbackSpec;

import java.util.Enumeration;

public class DoActionExample {

    public static void main (String[] args) {

        Element testElement = new Element();
        testElement.addAtt( new Att("obid",
"VR:wt.part.WTPart:73694:551318183-990560613281-2091086-126-8-253-
132@Windchill.mn.ptc.com") );
        testElement.addAtt( new Att("name", "ENGINE") );
        testElement.addAtt( new Att("number", "2623844395") );
        testElement.addAtt( new Att("CLASS", "wt.part.WTPart") );
        Element[] elementArray = { new Element("arrayElement1"),
                                   new Element("arrayElement2") };

        Group group1 = new Group( "group1" );

        group1.addElement( group1Elem1 );
        group1.addElement( group1Elem2 );

        ...

        try {
            Object[][] argv = { {"testElement", testElement },
                                {"group1", group1},
                                {"elementArray", elementArray},
                                {"bool1", new Boolean(true)},
                                {"char1", new Character('a')},
                                {"int1", new Integer(100)},
                                {"byte1", new Byte((byte) 9)},
                                {"double1", new Double(10000)},
                                {"short1", new Short((short) 99)},
                                {"float1", new Float(100.25)},
                                {"long1", new Long(4000)},
                                {"string1", "argv Test String" } };

            // Federation Service doAction() is accessed through
            // FederationHelper
            Group resultGroup =
                FederationHelper.doAction( "testAction", argv );

            //Example calling doAction() with a FeedbackSpec object.
            Object key = TdTest.class.getName();
            FeedbackSpec feedbackSpec = new DefaultFeedbackSpec(key,
true);
```

```

        Group resultGroup =
            FederationHelper.doAction( "testAction", argv,
feedbackSpec );

        if ( resultGroup != null ) {
            System.out.println( "resultGroup:" );
            showIeGroup( resultGroup );
        }
        else {
            System.out.println( "resultGroup is null." );
        }
    }

    catch ( WTEException exc ) {
        ...
    }
    ...
}

private static void showIeElement(Element inElement) {

    Enumeration attrs = null;
    Att attr = null;

    if ( inElement != null ) {
        System.out.println( "Element:" );
        System.out.println( " Element Name: " +
inElement.getName() );

        System.out.println( " Element Attributes:" );
        attrs = inElement.getAtts();
        while ( attrs.hasMoreElements() ) {
            attr = (com.infoengine.object.factory.Att)
attrs.nextElement();
            System.out.println( " " + attr.getName() + ": " +
attr.getValue() );
        }
    }
}

private static void showIeGroup(Group inGroup) {

    if ( inGroup != null ) {
        Enumeration elements = null;

        System.out.println( "Group:" );

        System.out.println( "Group Name: " + inGroup.getName() );
        System.out.println( "Number of Elements: " +
inGroup.getElementCount() );

        System.out.println( "\nGroup Elements:" );
        elements = inGroup.getElements();
        while ( elements.hasMoreElements() ) {
            showIeElement( (Element) elements.nextElement() );
        }
    }
}
}

```

```
}
```

sendFeedback

```
public void  
    sendFeedback(MethodFeedback feedbackObject)  
        throws FederationServicesException
```

Sends Feedback objects to the client. Supported API: true. Parameters:
feedbackObject - Required. The feedback object to be sent to the client. Throws:
wt.federation.FederationServicesException - Supported API: true. Parameters:
action - Required. The name of the action to perform on the input object(s).

argv - Required. Variable argument list. The format is an array of arrays, each containing a Name, Value pair. The Name is the argument name and must be a String type. The Value is the argument value. Values can be of type:

- § com.ptc.core.meta.type.common.TypeInstance or TypeInstance[],
- § com.infoengine.object.factory.Element or Element[],
- § com.infoengine.object.factory.Group,
- § java.lang.Boolean, Byte, Character, Double, Float, Integer, Long, Short, or String.

At least one argument name, value pair must be specified and at least one argument value must be of type TypeInstance (or TypeInstance[]), Element (or Element[]), or Group.

An example of three input values would look like:

```
Object[][] argv = { {"arg1Name", arg1Value},  
                    {"arg2Name", arg2Value},  
                    {"arg3Name", arg3Value} };
```

feedbackSpec - The feedback specification object. If there is no feedback specification, specify null. Returns: com.infoengine.object.factory.Group Info*Engine Group object containing one or more Elements or Type Instances. Throws: wt.util.WTException

Example

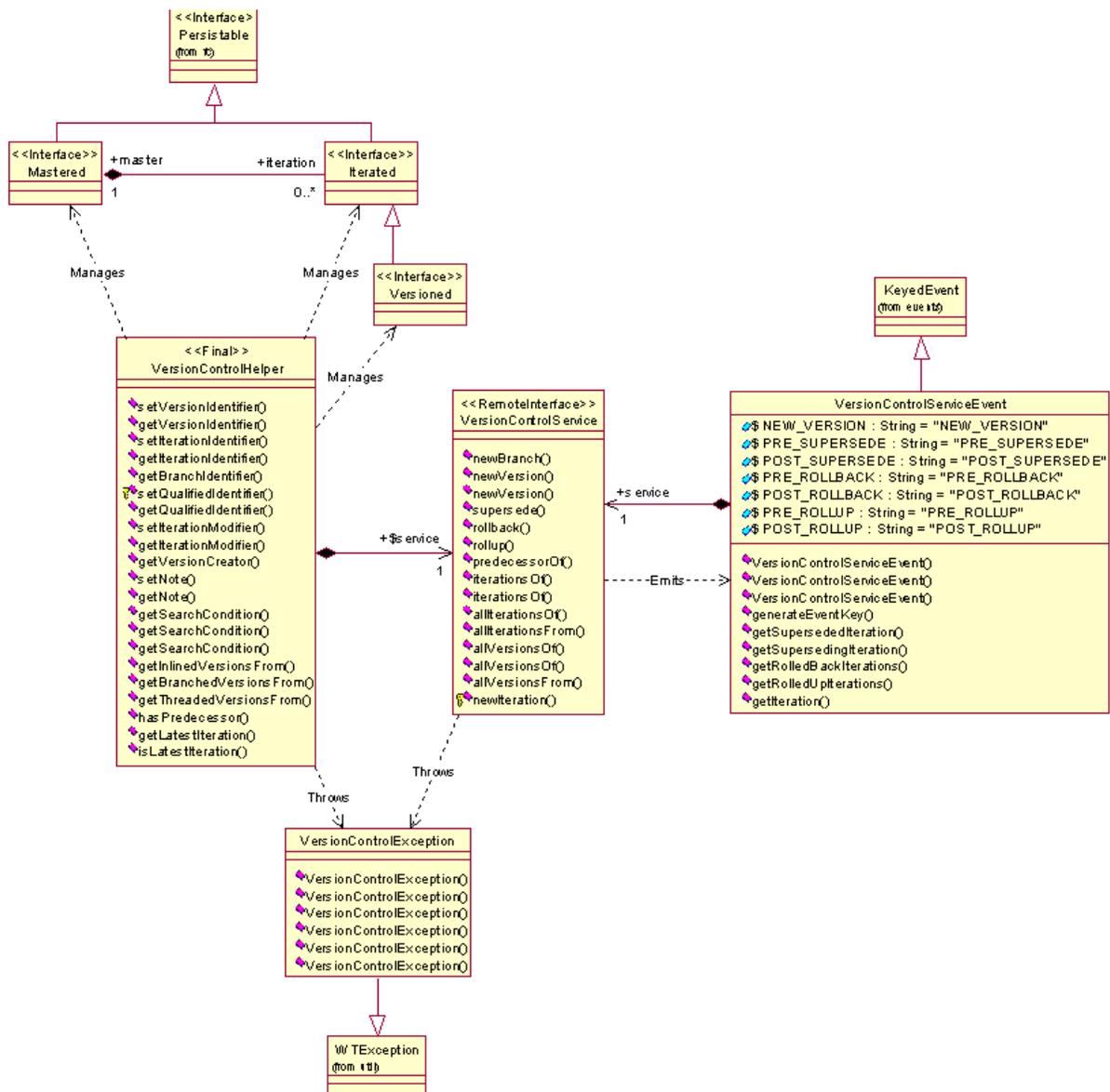
```
import wt.federation.FederationServerHelper;  
import wt.federation.FederationServicesException;  
import wt.feedback.StatusFeedback;  
  
public class SendFeedbackExample {  
  
    public static void main (String[] args) {  
        ...  
  
        try {  
            // Send the client a StatusFeedback type feedback object.  
            // Federation Service sendFeedback() is accessed through  
            // FederationServerHelper.service
```

```
        FederationServerHelper.service.sendFeedback(  
            new StatusFeedback( "Test Feedback Message" );  
        }  
    catch ( FederationServicesException exc ) {  
        ...  
    }  
    ...  
}  
}
```

vc package — Version Control Service

The version control service (wt.vc package) provides functionality to place versioned objects under iterative change control and iterated objects under incremental change control. Version control (VC) is typically regarded as a major and minor numbering scheme along with a change control mechanism for configuration management. The major number represents the iterative enterprise-level significance of a particular version or revision, and the minor number represents a less significant incremental work in progress change.

Design Overview



Version Control Model

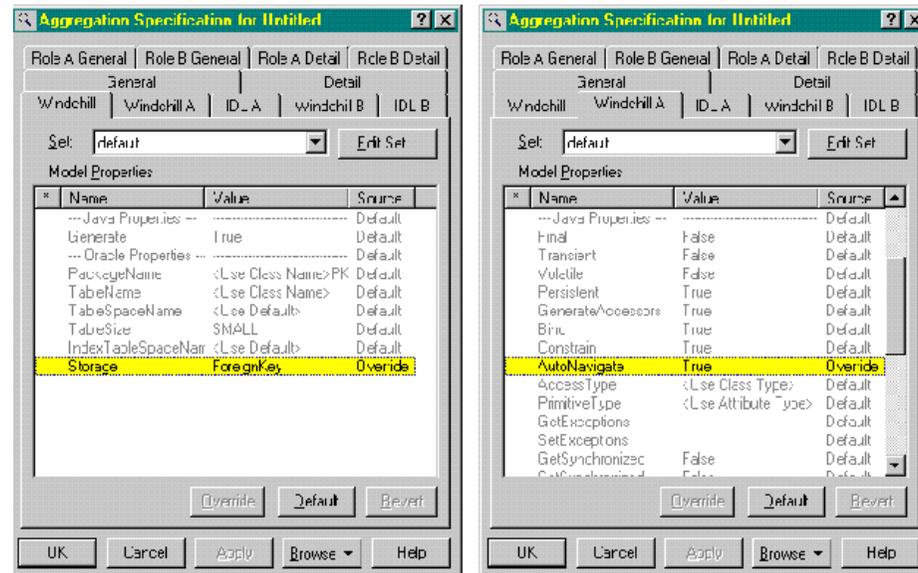
The VC service is designed to be a plug-and-play component in the Windchill system, which is by default enabled to execute.

The VC service is intended to be used for both client and server development. Business objects asserted as being Versioned in the object model can be revised, branched, and copied to new versions through the VC service's external interface. Business objects asserted as being Iterated in the object model can be incrementally changed by superseding the latest iteration, rolled back from a

superseding iteration to a superseded one, and rolled up from a superseded iteration to a superseding one. Objects asserted as Iterated only can be neither revised, branched, nor copied to new versions. With this design, version control allows for business objects to be iterated only or versioned, which is iterated.

The generalized association between Mastered and Iterated objects stipulates that if there is a branch, a master must exist but zero to many iterations of that master can exist. For iterations only, one and only one branch of development exists for an iteration. For Versioned objects, one to many branches of development can exist. For example, revisions and manufacturing views are separate branches of development. The branch itself is not a mechanism but an implementation detail of version control.

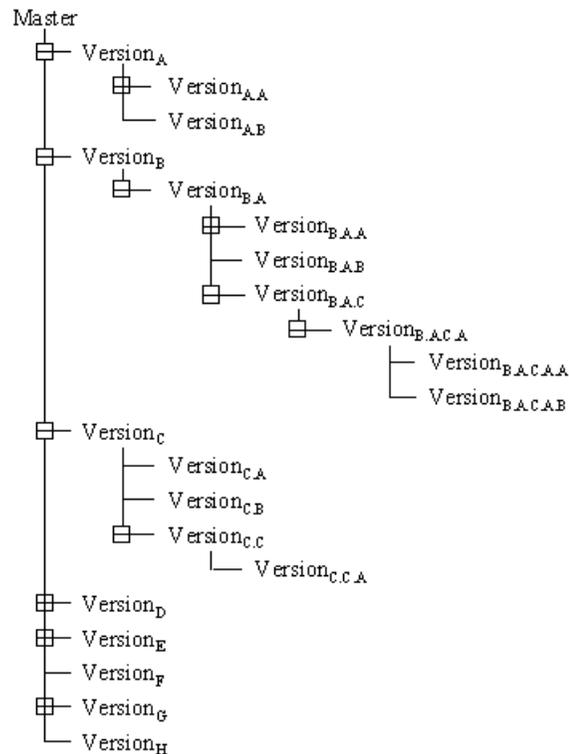
Additionally, the association between the Mastered and Iterated interfaces is significant and must be overridden between a concrete master subclass and either a concrete or abstract iteration subclass. This association specifies that a master contains its iterations but each iteration has a foreign key to its master by means of an object reference, not a link. Also, the master should be auto navigated when the iteration is retrieved from the database. This ensures that one SQL statement fetches both the iteration and its master via a database view for each concrete iteration subclass. See the figure below for the Windchill properties on how to specify a foreign key, auto navigated association.



Foreign Key and Auto-Navigate Properties

A business object's master represents a version independent concept of some piece of information. Typically, a master is the normalized identity common to all its versions, which through time is not subject to change. Additionally, it is often said the master represents the interface (for example, the form, fit, and function) for the business object. A business object's version represents a form or variant of the original business object.

A version is a notational enterprise-level significant object where it is the focal point for usage, and physically is the latest iteration in a branch of development. Versions are identified by unique values within a series. A version can be constructed as an in-lined or branched version from another version. The in-lined versions can be thought of as being revised versions, whereas the branched versions allow for parallel work against an object by different organizations or people at the same time. In the reference implementation of Windchill, if a version was identified by the letter A, then a revised version of it would be identified as B and a branched version from it would be identified as A.A. The revising and branching of versions has the effect of making a version tree where the revisions add breadth and the branches add depth as shown in

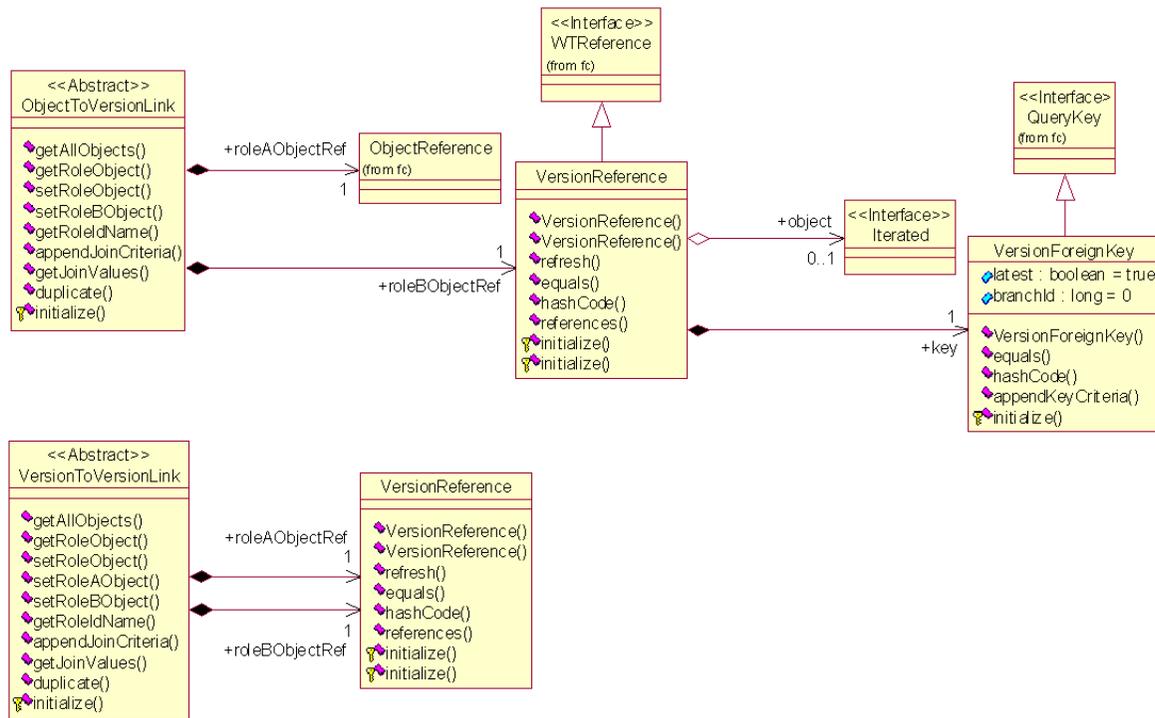


Versions

A business object's iteration holds the object's state and business qualifications. These qualifications are used to form configuration specifications but in general could be used as search criteria for ad hoc queries. Each iteration represents the working history of a version. Incremental changes in the object, represented by successive iterations, occur as the information is developed and changed. The latest iteration (that is, the version) represents a development branch's current implementation; the previous iterations are all history. Iterations are identified by unique values within a series. Additionally, unique fully-qualified identifiers can also identify iterations. These fully-qualified identifiers show the history of a particular iteration as it has been changed over time. In the reference

implementation of Windchill, if an iteration of version B was identified by the number 1, then a new iteration would be identified as 2. Its fully-qualified identifier, assuming that version B was constructed from iteration 9 of version A, would be A.9.B.2. This identifier states that the latest iteration is at 2 and iteration 1 of version B originated from iteration 9 of version A.

As shown in the figure below, version control also provides specialized abstractions to support the handling of the notional versions.



Version Support Mechanisms

The **ObjectToVersionLink** provides a binary type of link between a persistable object (role A) and a version (role B). When used in navigating from the persistable object to its associated version, only the latest iteration is found. Navigating from the version to its associated persistable object acts a typical navigation, but is applicable only from the latest iteration.

The **VersionToVersionLink** provides a binary type of link between two versions. Navigating between them is applicable only from the latest iteration on either side, and results in finding the other side's latest iteration.

The underlying mechanisms used by these two links are the **VersionReference** and **VersionForeignKey**. The **VersionReference** is a kind of reference to **Iterated** objects by means of the overridden key. This key is a foreign key to the latest iteration in a branch. When the reference's object is retrieved, but is not in memory, a query is performed to find the version and it is subsequently returned.

External Interface

The Mastered interface provides an abstraction of a plug-and-play component in conjunction with the Iterated interface. The intent is that, in an object model, a business object would assert that it is a master by inheriting (that is, it implements) the Mastered interface. With this assertion, iterations can then be identified by attributes in concrete masters (for example, name and number).

Note: Version control does not require Mastered objects to have any iterations, but allows for many.

The Iterated interface provides an abstraction of a plug-and-play component in conjunction with the Mastered interface. The intent is that, in an object model, a business object would assert that it is an iteration by inheriting (that is, it implements) the Iterated interface. With this assertion, the iterations can then be incrementally superseded, rolled back, and rolled up.

Note: Version control requires Iterated objects to have one and only one master. This is a design constraint in that foreign key associations that can be auto navigated (that is, the association between a master and its iterations) are required to have one side being of cardinality equal to one (1).

The Versioned interface provides an abstraction of a plug-and-play component that is a kind of Iterated object. The intent is that, in an object model, a business object would assert that it is a version by inheriting (that is, it implements) the Versioned interface. With this assertion, the business object can then be revised, branched, and copied to new versions.

The VersionHelper provides an abstraction as the API to the VC service. The API's methods can be categorized as either local or remote invocations. The local methods are getters of information, typically from cookies that are held in the business object. The remote methods serve as wrappers to a service that promotes server-side functionality.

The VersionService provides an abstraction that specifies and promotes server-side functionality as a service that is available remotely for use by a client. The intent is that this interface defines all the necessary server-side functionality for VC.

The VersionServiceEvent provides an abstraction of a specialized keyed event used by the VC service to signal other services that a VC activity is about to begin or has occurred. This gives other services the opportunity in a plug-and-play architecture to act accordingly on these events. Validation, vetoing, and post processing are typical reactions to events.

The VersionException provides an abstraction of an abnormal occurrence or error in the usage or processing of the VC service. This exception can be localized through a given resource bundle and other exceptions can be nested within it. The most common occurrences of this exception is when an attempt is made to use a

business object incorrectly during a database insertion, modification, deletion, or navigation.

Business Rules

As defined by the standard VC service's access control rules, no constraints are placed on the access of versioned objects. Furthermore, no constraints are placed on the access of either mastered or iterated objects.

Note: Direct manipulation or deletion of both masters and superseded iterations is currently unsupported. All functions should be carried out on versions.

For additional information, see the section on updating a master through an iteration in chapter [8](#), Developing Server Logic.

Event Processing

Event-based processing is performed on business objects asserted as being Iterated during database storing, deletions, and full restorations.

When a business object is being stored in the database, the VC service listens to a dispatched event indicating that the store is about to begin and stores the version's master, if needed.

After a successful store, the VC service listens to a dispatched event indicating that the store has completed and signals all other interested services that a new version exists if the stored iteration is either the first one created or is the first in a new branch.

When a business object is being deleted from the database, the VC service listens to a dispatched event indicating that the deletion is about to begin and vetoes the deletion if the version is not the latest one in that branch of development. Otherwise, it passes on vetoing the deletion.

After a successful deletion, the VC service listens to a dispatched event indicating that the delete has completed and deletes all of the version's iterations. If the deleted version is the only one remaining related to a master, then it is deleted as well.

When a business object is being fully restored from the database, the VC service listens to a dispatched event indicating that the full restoration is beginning and restores the iteration cookie's creator/modifier, predecessor and master references.

vc package — Baseline Service

Design Overview

The baseline is a fundamental concept of configuration management. Because PDM business objects are versionable and subject to change, numerous

configurations of a product structure will be created over time. Baselines identify and establish the product structure configurations that are of significant interest to the enterprise. As such, baselines represent product structure snapshots that are created by the various organizations of an enterprise for formal as well as informal reasons. Note that Baseline does not provide a general-purpose archiving utility that can reconstruct the state of the system as it existed at a point in time. There are many properties of the system that will not be recorded as part of Baseline, such as the foldering information, life cycle state, and so on.

External Interface

The Baselineable interface provides an abstraction of a plug-and-play component. The intent is that, in an object model, a business object would assert that it is Baselineable by inheriting (that is, it implements) the Baselineable interface. With this assertion, the business object can then be part of a Baseline. The Baselineable interface extends the Iterated interface and, therefore, must play the Iteration role in the Master-Iteration (MI) pattern.

The Baseline interface is asserted for an object that can be associated with Baselineable items. An out-of-the-box ManagedBaseline object implements the Baseline interface and has name, number, and description attributes. It is also Foldered and Life Cycle managed.

The Baseline service provides an abstraction that specifies and promotes server-side functionality as a service that is available remotely for use by a client. The intent is that this interface defines all the necessary server-side functionality for baseline. The BaselineHelper provides an abstraction as the API to the Baseline service.

The Baseline service provides basic APIs for adding and removing Baselineable items. There are also operations for retrieving baseline information. A convenience operation is also provided to automatically populate a Baseline by navigating structured data. As the structure is navigated, each Baselineable item is added to the Baseline. This navigation is customizable and an out-of-the-box implementation is provided for navigating a part structure via the uses relationship.

The BaselineServiceEvent provides an abstraction of a specialized keyed event used by the Baseline service to signal other services that a baseline activity is about to begin or has occurred. This gives other services the opportunity in a plug-and-play architecture to act accordingly on these events. Validation, vetoing, and post-processing are typical reactions to events.

Business Rules

The following business rules apply to baseline:

- Adding or removing items from a baseline is equivalent to modifying the Baseline object and is subject to the access control properties of the baseline. For example, if ManagedBaseline object is used, the life cycle state InWork

could have access rights set so that any user can modify the object and be able to add or remove items. The Released state could be set so that a normal user does not have modify access rights. This would allow for the concept of a frozen baseline. The baseline would be available for adding and removing items until it was frozen by changing its state to Released.

- A baseline can have zero or more Baselineable objects associated with it, and a Baselineable object can be part of zero or more baselines. However, for a given baseline, only a single iteration of a given master can be part of that baseline.
- Once an iteration has been added to a baseline, that iteration cannot be modified or deleted. Note that other attributes of a Baselineable object that are not specifically related to an iteration can still be modified (for example, life cycle state, folder, master attributes, and so on). This is the normal case for an Iterated object that is RevisionControlled. An object can be modified only if it is checked out.
- For operations that add multiple Baselineable objects to a baseline as a single action (for example, automatic population), the operation is performed as a single transaction.
- A baseline can be used in a ConfigSpec that essentially selects a single iteration of a master item, depending on which iteration is part of the specified baseline. For example, when navigating a product structure (such as, using the Product Information Explorer, generating a BOM, and so on), a Baseline ConfigSpec can be used. During search or navigation, if an item iteration is found by the Baseline ConfigSpec, then it will be returned. Otherwise, it will be considered baseline independent and only the master for that item will be returned. Because it is a master, navigation will not proceed below that object. (See the Configuration Specification section for more information.)

A baseline business object will be locked when it is edited to support concurrent usage. This is necessary because the Baseline service is responsible for ensuring that only a single iteration for each master is in the baseline. Without locking, if separate clients added different iterations for the same master, it would be possible for both iterations to be added. The Persistence Manager row level locking API is used. The baseline is locked within a transaction to prevent any other operations on that baseline. If some error occurs, then all changes are rolled back and the baseline is automatically unlocked.

- Whenever Baselineable items are added to or removed from a baseline, the modification timestamp is updated on the Baseline object. The contents of a baseline are considered business attributes of the Baseline object. Another desired side effect of the baseline update is that access permissions are checked on the Baseline object. If the user does not have authority to modify the Baseline object, then the entire baseline operation fails.
- There are no restrictions on adding Baselineable items to a baseline. For example, an item in a checked out or personal folder can be added to a

baseline. However, this can lead to situations that may be confusing to a user. When viewing a baseline, only those items that the user has access to (that is, in shared folders) are displayed. Another example is if a checked out item is placed in a baseline and then the undo-checkout operation is performed. Undo-checkout deletes the working copy, which will cause the Baseline service to throw an exception (items in a baseline cannot be modified or deleted).

Event Processing

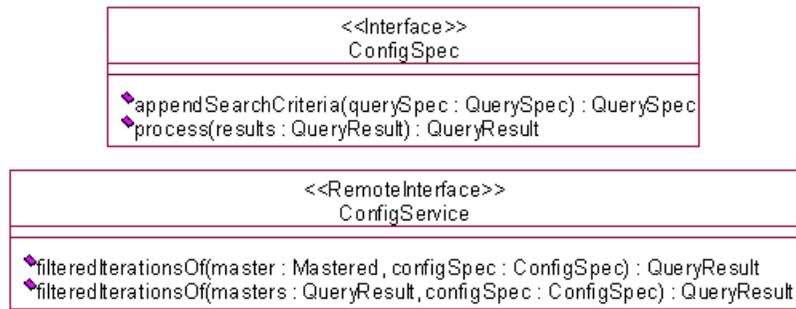
The Baseline service is both a producer and consumer of events. Whenever an item is added, removed, or replaced in a baseline, events are generated prior to and after the operation. For example, adding an item generates `PRE_ADD_BASELINE` and `POST_ADD_BASELINE` events. Developers can use the basic event listening mechanism to perform customized actions during baseline operations. The Baseline service listens for the pre-modify, pre-delete, and prepare for modify events to enforce business rules for items that are part of a baseline.

vc package — Configuration Specification Service

Engineers working with iterated and versioned data do so within a context, the same way software engineers using source control tools work in a branch. The engineer specifies the context to the system and the system uses it to calculate the appropriate representation. For example, a software engineer in the "Release 2" branch would work against a different version of the file `ConfigSpec.java` than would the software engineer maintaining "Release 1.0".

The `wt.vc.config` package provides the functionality needed to define a context and calculate appropriate versions based on the context; however, the versioning model must be understood before the config package can be understood. The Windchill version control model creates versions when masters are created or when existing versions are revised, branched, or checked out (check-out creates a temporary working version). The checkout/checkin process causes existing iterations to be superseded. While a version is conceptually the latest iteration, superseded iterations are also stored as versions in the implementation model because the version/iteration concepts have been joined at the datastore into a single table. VC's `iterationsOf` and `versionsOf` serve to highlight the distinction; `versionsOf` returns only latest iterations. However, the conceptual version is not always a latest iteration. A user specifying a baseline context, for example, wants the version in the baseline, which need not equate to a latest iteration. The config package calculates appropriate versions, without assuming that only latest iterations are versions.

The figure below illustrates the fundamental wt.vc.config classes and methods.



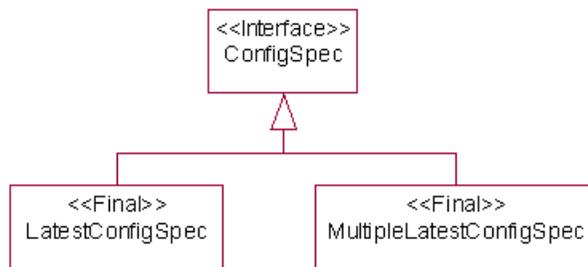
Fundamental wt.vc.config Classes and Methods

The config package encapsulates the concept of a context as the ConfigSpec interface. The concept of calculating the appropriate representation (versions) is a process by which a master or set of masters is converted into versions. This process is encapsulated in the filteredIterationsOf APIs defined in the ConfigService. The filteredIterationsOf API is composed of three phases.

The first establishes a QuerySpec based on the masters (passed as an argument) and a ConfigSpec; the QuerySpec's target class is Iterated or a subclass of it. The ConfigSpec's appendSearchCriteria is called during this phase, allowing the ConfigSpec to add any search criteria it needs to prefilter as a part of the query. Note that, because config does not assume that a version has to be a latest iteration, a query that does not include a "latest = true" condition will result in every iteration for each master to be returned.

The second phase is the query phase. The QuerySpec that was built is passed to the PersistenceHelper's find API (the resulting versions are, therefore, subject to access control).

The third and final phase allows the ConfigSpec to further filter the results of the query using an algorithm it defines in its process API if the query criteria it appended was not sufficient to guarantee the appropriate results. The following out-of-the-box ConfigSpecs illustrate the use of appendSearchCondition and process.



Basic ConfigSpecs

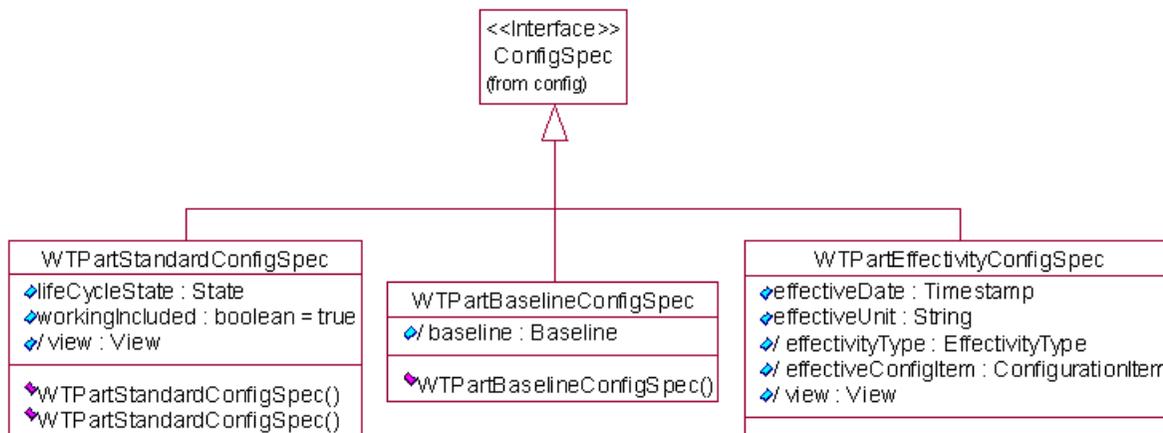
wt.vc.config.MultipleLatestConfigSpec

This ConfigSpec is very similar to VC's allVersionsOf, except that versions not owned by the current principal are not returned. It is useful when a user wants to see every version of a master to obtain a global sense of the evolution of the object, or simply wants to find these versions to do comparisons. The appendSearchCriteria API is implemented to append a "latest = true" condition in the same manner as versions are defined by VC. The process API removes those versions that are not owned by the current principal (that is, checked out to another user). Note that the appendSearchCriteria API could append conditions for ownership that would allow it to determine the valid versions without processing, but this can be done only when the target class itself is Ownable. Even so, a process is necessary because the target class may not be Ownable and have children that are.

wt.vc.config.LatestConfigSpec

Similar to the MultipleLatestConfigSpec, this ConfigSpec takes all those versions that would be valid in the MultipleLatestConfigSpec and processes them down to a single version per master. Here, the bulk of the work is done in the process, where the latest version is determined by comparing versions (for example, A.A is later than A and A.2 is later than A.1). Since most contexts are intended to find one version, this ConfigSpec is often incorporated into other ConfigSpecs.

Clients working with versioned data may use the config's functionality to return all versions (via the MultipleLatestConfigSpec) or to return the current principal's working copy, if one exists. The wt.part package makes heavy use of the ConfigSpec in the Product Information Explorer to present a coherent picture of parts and their structures. The following ConfigSpecs have been defined for use by the part package.



wt.part PackageConfigSpecs

wt.part.WTPartStandardConfigSpec

This ConfigSpec is specifically written against parts (although it would work for any ViewManageable, Workable, and LifeCycleManaged Iterated class).

It combines the state a part is in and the view it may be assigned to in order to produce a context that is based on state and view for latest iterations. If the lifeCycleState value is assigned, the version must be in that particular state (multiple states are not supported). If the view value is assigned, the version must be assigned to that view (or one of its parents if no version has been assigned to the view) or view independent. There is also a workingIncluded that, if false, removes anything that is owned, even if it is owned by the current principal. This ConfigSpec allows a user to state, for example, that he works in the Engineering view and would like to see the Released product. Note that the LatestConfigSpec is used to both append to the QuerySpec and process to a single version per master.

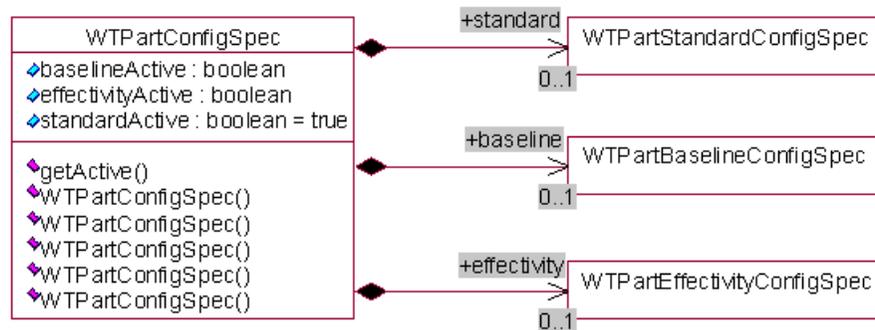
wt.part.WTPartBaselineConfigSpec

This ConfigSpec (which could be used with any Baselineable Iterated class) is the only out-of-the-box ConfigSpec to not follow VC's definition of a version as the latest iteration. It returns only those iterations that are assigned to the specified baseline.

wt.part.WTPartEffectivityConfigSpec

This ConfigSpec works with the effectivity package and ConfigurationItems to return only those versions that are effective given an effective date/unit and possibly a ConfigurationItem and View. This ConfigSpec handles the view setting in the same manner as WTPartStandardConfigSpec.

The following figure illustrates WTPartConfigSpec.

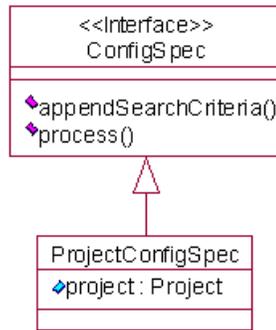


WTPartConfigSpec

The WTPartStandardConfigSpec, WTPartBaselineConfigSpec, and WTPartEffectivityConfigSpec are aggregated into the WTPartConfigSpec. This ConfigSpec establishes an active zone and simply forwards the appendSearchCondition and process commands to the active ConfigSpec.

The preceding examples are not the only ways contexts could be established. Other potential ways to establish a context are by project or cabinet/folder membership. These can all be created via simple customizations by implementing ConfigSpec and its APIs. The following illustrates the possible implementation of a ProjectConfigSpec with a project attribute of type Project. (Note that this should

serve as an example only; it has not been tested and is not supported by Windchill as code.)



ProjectConfigSpec Example

After modeling the ProjectConfigSpec and generating its code, the appendSearchCriteria will be implemented to add SearchConditions for the value of the project attribute and for ownership if the target class is Ownable. Note the use of LatestConfigSpec.

```
public QuerySpec appendSearchCriteria( QuerySpec querySpec )
    throws WTEException, QueryException {
    //##begin appendSearchCriteria% [ ]36484C990290.body preserve=yes

    //We want only iterations, use LatestConfigSpec's
    //appendSearchCondition
    //to append a "latest = true" search condition.
    QuerySpec qs = (new LatestConfigSpec()).appendSearchCriteria(
        querySpec);

    //Add the condition for project.
    qs.appendAnd();
    qs.appendSearchCondition(new SearchCondition(qs.getTargetClass(),
        ProjectManaged.PROJECT_ID + "." + ObjectReference.KEY,
        SearchCondition.EQUAL,
        PersistenceHelper.getObjectIdentifier(project)));

    //If the target class is Ownable, make sure it's either owned by
    // me or not owned at all (in a vault).
    if (qs.getTargetClass() instanceof Ownable) {
        qs.appendAnd();
        qs.appendOpenParen();

        qs.appendSearchCondition(OwnershipHelper.getSearchCondition(
            qs.getTargetClass(), SessionHelper.manager.getPrincipal(),
            true));

        qs.appendOr();

        qs.appendSearchCondition(OwnershipHelper.getSearchCondition(
            qs.getTargetClass(), false));

        qs.appendCloseParen();
    }
}
```

```

return qs;
}
}

```

The process API is simple, although it does require some implementation. The target class might not have been Ownable and yet may have had Ownable children, so ownership needs to be processed. Also, this ConfigSpec should return a single version per master, so multiple versions will need to be processed down to one. Fortunately, the LatestConfigSpec does all of these things, so this implementation will simply return the result of executing its process API. The code would look similar to the following:

```

public QueryResult process( QueryResult results )
    throws WTEException {
    ///##begin process% [ ]353B588501C5.body preserve=yes

    //We can not simply return the results, children of the target
    //class may have been Ownable, even though the target
    //class was not.
    //Let
    //LatestConfigSpec handle this case because its process API
    //accounts
    //for that.
    return (new LatestConfigSpec()).process(results);;
    ///##end process% [ ]353B588501C5.body
}

```

The config package, via a ConfigSpec and the ConfigService's filteredIterationsOf APIs, is fundamentally capable of converting masters to iterations. The ConfigHelper has APIs to extend its usefulness beyond simply taking masters and returning iterations.

<<Final>> ConfigHelper
<ul style="list-style-type: none"> ◆ mastersOf(links : QueryResult, role : String) : QueryResult ◆ mastersOf(arrayedElements : QueryResult, index : int) : QueryResult ◆ iterationsOf(links : QueryResult, role : String) : QueryResult ◆ iterationsOf(arrayedElements : QueryResult, index : int) : QueryResult ◆ recoverMissingMasters(masters : QueryResult, iterations : QueryResult) : QueryResult ◆ buildConfigResultFromLinks(linksToMasters : QueryResult, role : String, configResults : QueryResult) : QueryResult ◆ removeExtraLinks(linksToIterations : QueryResult, role : String, configResults : QueryResult) : QueryResult

ConfigHelper

The mastersOf APIs allow the conversion of a QueryResult of links to masters or Persistable[]'s in which masters can be found at a certain position. The conversion returns a QueryResult of masters that can then be applied to filteredIterationsOf. Similarly, iterationsOf converts to QueryResults of iterations that can then be processed by a ConfigSpec (useful for links to iterations). If no versions for a particular master match once it has been processed, it is possible to recover the

master so it can be displayed in place of the version or versions (to indicate that no version matched, but there is data). This can be done via the `recoverMissingMasters` API. If iterations drop out as a result of being processed, links to those iterations can be removed by calling the `removeExtraLinks` API.

The Product Information Explorer navigates `WTPartUsageLinks` from part versions to part versions. The `WTPartUsageLink` (a type of `IteratedUsageLink` from `wt.vc.struct`) is defined between a `WTPart` and a `WTPartMaster`. No link is defined as being between `WTPart` and `WTPart`; however, the requirement for such a link comes primarily (as is the case for the Product Information Explorer) from display. The `buildConfigResultsFromLinks` API allows iteration-to-master associations to be represented as iteration-to-iteration associations by a client. It does this by pairing the master (on the master role) with its corresponding version and returning a `QueryResult` a `Persistable`'s. Each element, a `Persistable`'s, is of length 2 and contains the link in the 0th position and the version in the 1st position. The client then takes the real link's information (such as `Quantity`) and simply displays the version instead of the master. The following code illustrates the use of the `ConfigSpec`, `ConfigService`, and `ConfigHelper` to navigate an `IteratedUsageLink` to return either the other side (versions) or the links (and versions):

```
public QueryResult navigateSomeIteratedUsageLinkAlongTheUsesRole(
    Iterated someIteration, boolean onlyOtherSide,
    ConfigSpec someConfigSpec )
    throws WTEException {

    //Navigate the link to its masters.
    QueryResult uses = PersistenceHelper.manager.navigate(someIteration,
        SomeIteratedUsageLink.USES_ROLE, SomeIteratedUsageLink.class,
        onlyOtherSide);

    //If the navigation returns links, convert those links to masters.
    QueryResult usesMasters = (onlyOtherSide) - uses :
        ConfigHelper.mastersOf(uses, IteratedUsageLink.USES_ROLE);

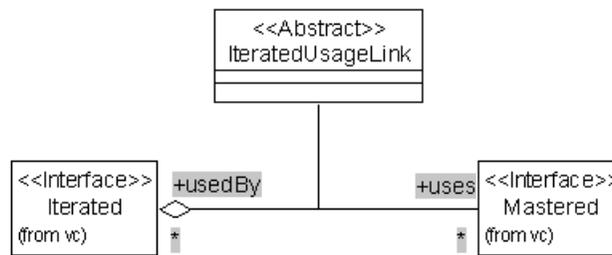
    //Get the iterations for these masters. If masters are lost because
    //no matching iterations were found, recover those masters for
    //display.
    QueryResult iterations =
        ConfigHelper.recoverMissingMasters(usesMasters,
            ConfigHelper.service.filteredIterationsOf(usesMasters,
                someConfigSpec));

    //If the user wants only the other side objects, return the
    //iterations. Otherwise, pair up the links with the versions
    //for those links.
    if (onlyOtherSide)
        return iterations;
    else
        return ConfigHelper.buildConfigResultFromLinks(uses,
            IteratedUsageLink.USES_ROLE, iterations);
}
```

vc package — Version Structuring Service

The wt.vc.struct package introduces an associative model for relating versions. Versioned data forms links to other versioned data to further describe that data - parts associate to other parts to form structures, indicating that a part is composed of or built from other parts. Similarly, it is common for a document to refer to other documentation. The struct package defines a "uses" and a "references" concept to reflect these kinds of associations.

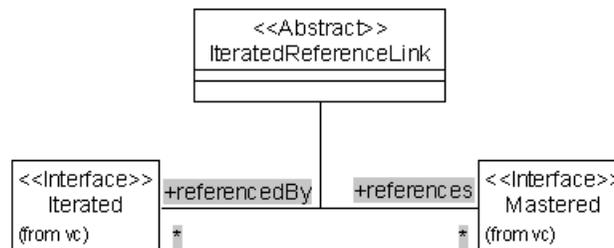
The following figure illustrates the uses concept.



Link Representing the Uses Concept

The uses concept is a structuring concept indicating a tight coupling. An object that uses another object identifies that object as essential to its implementation. For example, a part built from other parts is not complete without these part; they are required components. The struct package encapsulates the uses concept for use by versioned data in the IteratedUsageLink.

The following figure illustrates the references concept.

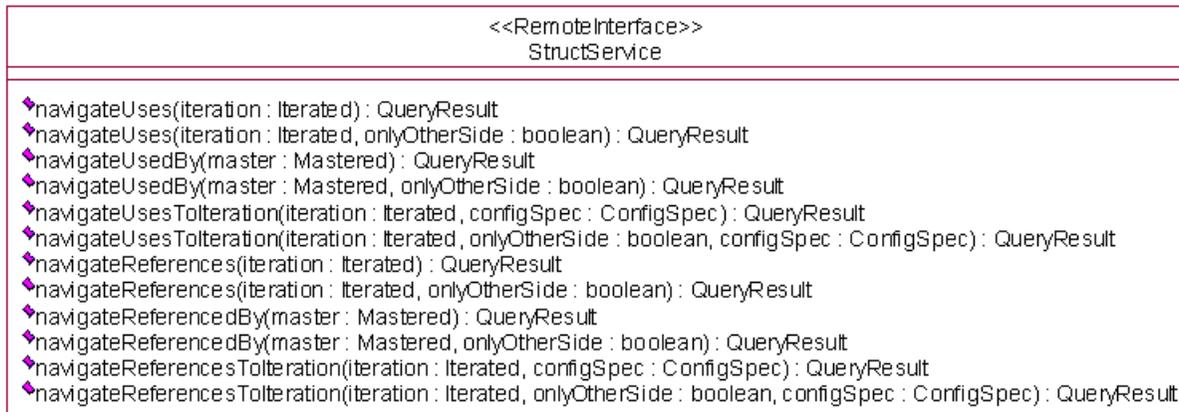


Link Representing the References Concept

The references association couples data more loosely than the uses association. For example, the referenced data may be important to a document, but the document can be disseminated without it. For example, a design specification that references the requirements specification it was written for is still complete on its own. The struct package's IteratedReferenceLink allows versioned data to participate in references associations.

The wt.vc.struct package's IteratedUsageLink and IteratedReferenceLink both link Iterated data (versions) to Mastered data (masters). The fact that a master, not an iteration (or version), is used or referenced by an implementation is an important aspect of the links definition. They assume the versions for the master are interchangeable and that changes that are not interchangeable result in a new master (not a new, incompatible version). Note that the shared aggregation modeled on the IteratedUsageLink shows its tighter coupling.

The following figure illustrates the struct package APIs.



structPackage APIs

The struct package provides navigate APIs for both the IteratedUsageLink and IteratedReferenceLink for convenience and because of minor behavioral nuances. The navigateUses and navigateReferences APIs behave exactly as would PersistenceManager's navigate, however navigateUsedBy and navigateReferencedBy return only versions; they filter out any iteration that is not the latest iteration. The navigateUsesToIteration and navigateReferencesToIteration APIs navigate the links to the masters, then resolve the masters to their appropriate versions using the passed ConfigSpec. The masters for which there are no appropriate versions are returned to indicate that the association exists but has no qualifying representation given the ConfigSpec. The return value for the navigate to iterations API, a QueryResult, consists of those versions (and possibly masters) if onlyOtherSide is true. If it is false (that is, the links are desired as well) a QueryResult containing Persistable's as elements is returned. Each element is an array of length 2 with the link in the 0th position and the version (or master) in the 1st position.

The IteratedUsageLink and IteratedReferenceLink, while conceptually different, have the same internal behavior. The following characteristics are shared by both associations:

- When an Iterated object is created from an existing one (as a consequence of a check-out, new version, or new branch), all IteratedUsageLinks and IteratedReferenceLinks to that iteration are copied. This copy uses PersistenceManagerSvr's copyLink API, which allows listeners of the COPY_LINK event to copy any links to the IteratedUsageLink/IteratedReferenceLink that should be copied.
- To create, delete, or update IteratedUsageLinks or IteratedReferenceLinks, the user must have permission to modify the Iterated object.
- Creating these links between an iteration and its own master is not allowed.
- A master can not be deleted if it has IteratedUsageLinks or IteratedReferenceLinks to it.

The customization, via extension, of the IteratedUsageLink and the IteratedReferenceLink follows the same process as does any ObjectToObjectLink. Note that the links are protected by the iterations; if the iteration is frozen, so are these links. For this reason, it should not be necessary to build a lot of business knowledge into the links; the business knowledge should be put on the versions. Also note that struct's navigate APIs will navigate at the abstract IteratedUsageLink/IteratedReferenceLink level. If extended and it is desired to navigate only the new, concrete link, new APIs are required to guarantee that only the new link is navigated.

vc package — Version Viewing Service

The views package introduces the concept of ViewManageable versions. In many enterprises, the development of a product is such that a group may work on it to a certain point, hand it off to another group that makes value-added changes, and so on. While a downstream group is working against a handed-off version, the upstream group can continue to work on it.

In a simple example, the Engineering group releases revision A of a part to the Manufacturing group. At this point, the Manufacturing group restructures the part, making necessary changes for production purposes while the Engineering group begins work on revision B of the part.

The views package captures such a process. The Engineering and Manufacturing groups are represented by Views. These views can be associated in a manner that encapsulates the process. For example, the Manufacturing view becomes a child of the Engineering view. Engineers can then assign their ViewManageable data (parts in this example) to views, indicating that these objects are view-dependent and can be identified as Engineering or Manufacturing data. Most ViewManageable objects are assigned to the root view, because it is generally not the responsibility of downstream views to create new data as much as to modify

the data to fit requirements. However, phantom parts are parts that the manufacturing group might create when a group of parts are actually pre-built.

The Windchill model builds view management capabilities on top of the existing versioning capabilities. A ViewManageable version, when assigned to a view, is qualified by the view. When a downstream group is ready to accept and add value to the version, it branches its own version (qualified by its view). The following example illustrates this concept.

- The Engineering group creates part 100, assigning it to the Engineering view. The version of this part is A.
- After several iterations, part 100 is released.
- The Manufacturing group branches part 100, creating version A.A of part 100, which is assigned to the Manufacturing view.
- Meanwhile, the Engineering group revises part 100, creating version B (which is also assigned to the Engineering view).
- The Manufacturing group releases version A.A and puts it into production.

ViewManageable versions have the following characteristics:

- The view a ViewManageable version is assigned to can not be changed once persisted. In the preceding example, version A of part 100 can not be re-assigned to the Manufacturing view.
- View-dependent versions can not become view-independent and view-independent versions can not become view-dependent. The first version of a master determines that all versions of that master are view-dependent or view-independent.
- View-dependent versions can be branched only into downstream views. When creating a branch for a particular view from a view-dependent version, only those views that are downstream from the current view are valid for the new version. The view to which the initial version was assigned is considered the principal owner of the master; downstream views provide value-added changes.
- Branching is used as the underlying mechanism, allowing downstream views to make value-added alterations. Because this mechanism creates new versions, changes made are isolated to the branched versions, allowing concurrent development. Note that as separate versions, the view-dependent versions have independent life cycle states and can be independently life-cycle managed.
- The ViewQualification provides a qualifier for ViewManageable data. When applied, it filters out all view-dependent versions not assigned to the view specified. By default, if no view-dependent versions have been branched into the view specified, the parent views are checked until either no views remain or at least one version is assigned to a parent view. Setting parentSearched to false disables the parent-searching behavior.

Views and their associations can be created using the LoadViews application. See the Administrator's Guide for details. Views and view associations exhibit the following behavior:

- Views, once created, can not be deleted, but can be renamed.
- Views can be inserted between a parent and its children at any time. However, this is the only way to alter the structure after it has been established.

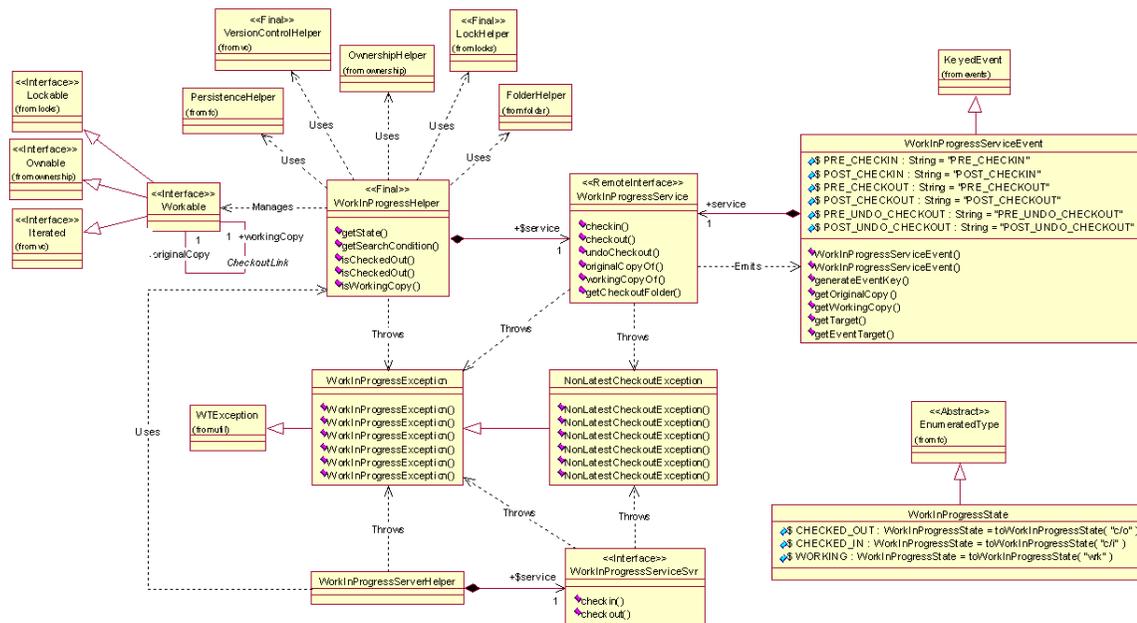
The ViewHelper and ViewService provide the functionality associated with view management. The helper provides the client-side APIs to assign the version to a view and get the view to which a version is assigned. The service provides APIs to branch view-dependent versions into downstream views, and to obtain information about the views and their associativity. When developing against this package the following should be noted:

- The service employs caching, which the ViewReference uses. It is more efficient to call the service APIs than to go to the database for view/view associativity information.
- The ViewService does not emit an event when a new version is branched in a downstream view. VersionServiceEvent's NEW_VERSION event should be subscribed to if you are interested in the creation of a new view-dependent version or branch.

vc package — Work in Progress Service

The work in progress service (wt.vc.wip package) provides functionality to perform Work In Progress (WIP) on workable objects, generally known as checkout and checkin, or undoing of a checkout.

Design Overview



Work in Progress Model

The WIP service is designed to be a plug-and-play component in the Windchill system, which is by default enabled to execute.

The WIP service is intended to be used for both client and server development. Business objects asserted as being Workable (Iterated, Lockable, and Ownable) in the object model can be checked out and then either checked in or undone from being checked out. When an object is checked in, it exists in a vault. When it is checked out, it is outside the vault. The protection from modification that is offered information is somewhat different when it is inside the vault versus when it is outside the vault.

When a checkout occurs, the original object is locked, and a working copy is made from the original and linked to it via the CheckoutLink. If the Workable is instance of Foldered then the working copy is placed in the user's Checked Out folder (in the default implementation). By being placed in the user's Checked Out folder, the user becomes the owner of the working copy with the access rights that have been established for it. If the Workable is not foldered, the working copy is moved to the USER domain and the user owns it. The working copy is considered an exact duplicate of the original (that is, it has the same attributes and associations). For example, if iteration 5 of version B is checked out, the working copy is iteration 5 of version B.

Upon a checkin, the original object's latest iteration is superseded with the working copy and its identifier is incremented to the next adjacent value in the

series. Given the checkout example above, the checked in object would now be at iteration 6 of version B. Mechanically, a checkin can be done from either the original object or its working copy. The system will understand what is meant and perform the checkin. When the checkin completes, the lock on the original object is released and the working copy is removed from the checked out folder.

Upon an undo of a checkout, the working copy is deleted along with the link to the original object. Additionally, whenever the checked-out original object or working copy is deleted, the checkout is undone automatically.

External Interface

The `Workable` interface provides an abstraction of a plug and play component. The intent is that, in an object model, a business object would assert that it is `Workable` by inheriting (that is, to implement) the `Workable` interface. With this assertion, the business object can then be checked out and checked in.

The `WorkInProgressHelper` provides an abstraction as the API to the WIP service. The API's methods can be categorized as either local or remote invocations. The local methods are getters of information, typically from cookies that are held in the business object. The remote methods serve as wrappers to a service that promotes server-side functionality.

The `WorkInProgressService` provides an abstraction that specifies and promotes server-side functionality as a service that is remotely available for use by a client. The intent is that this interface defines all the necessary server-side functionality for WIP.

The `WorkInProgressServiceEvent` provides an abstraction of a specialized keyed event used by the WIP service to signal other services that a WIP activity is about to begin or has occurred. This gives other services the opportunity in a plug-and-play architecture to act accordingly on these events. Validation, vetoing, and post-processing are typical reactions to events.

The `WorkInProgressException` provides an abstraction of an abnormal occurrence or error in the usage or processing of the WIP service. This exception can be localized through a given resource bundle and other exceptions can be nested within it. The most common occurrences of this exception is when an attempt is made to check in/out a business object but it already is checked in/out, and when used incorrectly during a database modification, deletion, and navigation.

The `WorkInProgressState` provides an abstraction of the valid set of work-in-progress states that a `Workable` object can exist within. The three supported states are checked out, checked in, and working.

Business Rules

As defined by the standard WIP service's access control rules, since a `Workable` object is asserted as being `Lockable`, the service relies on the locking service for applicable access control. Additionally, when an object is checked out, neither the original checked out nor working copies can be checked out again.

Event Processing

Event-based processing is performed on business objects asserted as being workable during database storing, [preparation for] modifications, deletions, and folder changes.

When a business object is being stored in the database, the WIP service listens to a dispatched event indicating that the store is about to begin and initializes the state of the object to being checked in if and only if its checkout info cookie is null. Since a workable may also be a Foldered, the WIP service listens to a dispatched event indicating that a store on a cabinet has successfully completed and checks if the cabinet exists in the user domain (that is, a personal cabinet) and, if so, stores a checkout folder in that cabinet.

When a business object is [prepared for] being modified in the database, the WIP service listens to a dispatched event indicating that the modify is about to begin and vetoes it if either of the following conditions are true:

- The business object is checked in and not owned by the current session's principal.
- The business object is checked out to enforce a strong vaulting notion.

Otherwise, the WIP service allows the modification to take place.

When a business object is being deleted in the database, the WIP service listens to a dispatched event indicating that a deletion is about to begin and undoes a checkout if the object is checked out or a working copy of a checked out object.

workflow package — Workflow Service

The workflow service resides in the package `wt.workflow`. It provides functionality to create and manage workflow definitions, initiate and manage process instances, and distribute work items to users and groups.

For additional information about the creation of workflow process definitions, initiation of process instances, and participation in workflow processes, see the *Windchill Workflow Tutorial*.

Design Overview

The workflow package contains the following five subpackages:

engine

Directly responsible for the flow of control and data from an execution point of view.

definer

Definition of workflow objects, that is, processes, activities, and the other objects that compose the workflow network.

work

Responsible for the delivery and execution of assigned activities and work list.

robots

Responsible for the execution of robot activities, that is, activities that are executed by the system rather than assigned to a participant.

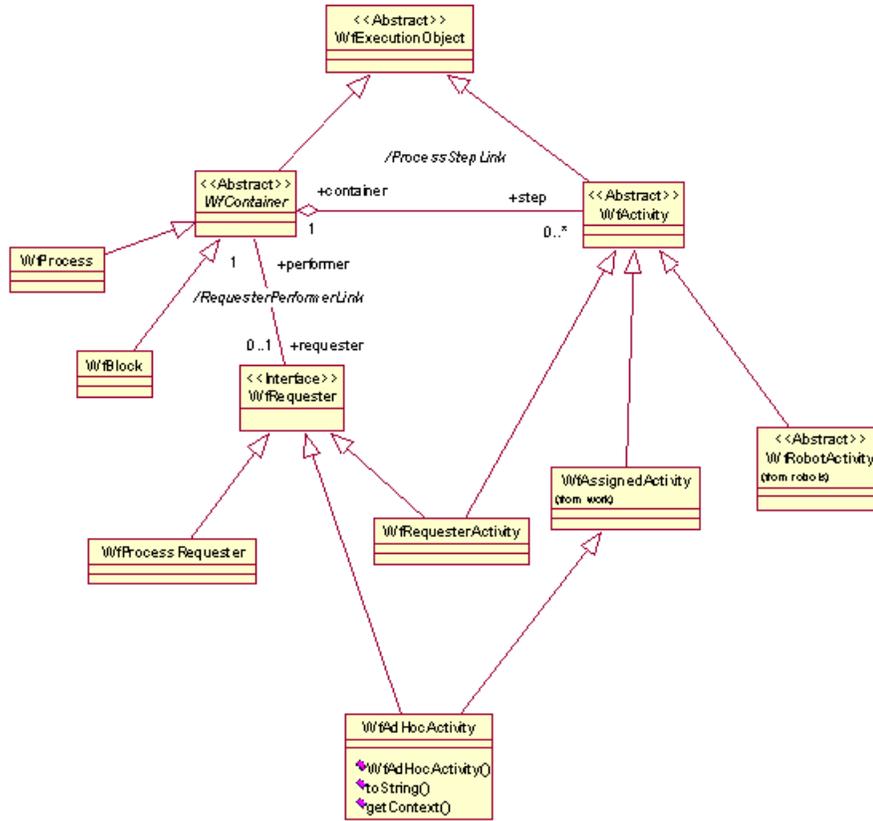
worklist

Responsible for server-side support of work list construction.

Workflow Engine Package

This package contains the classes responsible for the workflow enactment. Workflows are represented by processes (WfProcess objects) or blocks (WfBlock objects) that are composed of activities (WfActivity objects or WfAdhocActivity objects). Each activity may be performed by humans (WfAssignedActivity and WfAdhocActivity), or by applications (represented by a WfRobotActivity object). Finally, an activity may also be a reference (represented by a WfRequesterActivity object).

Both WfProcess and WfActivity classes extend the WfExecutionObject. WfExecutionObject is responsible mainly for object identification, state maintenance (allowing only valid transitions), and information holding.



WorkflowEngine Package

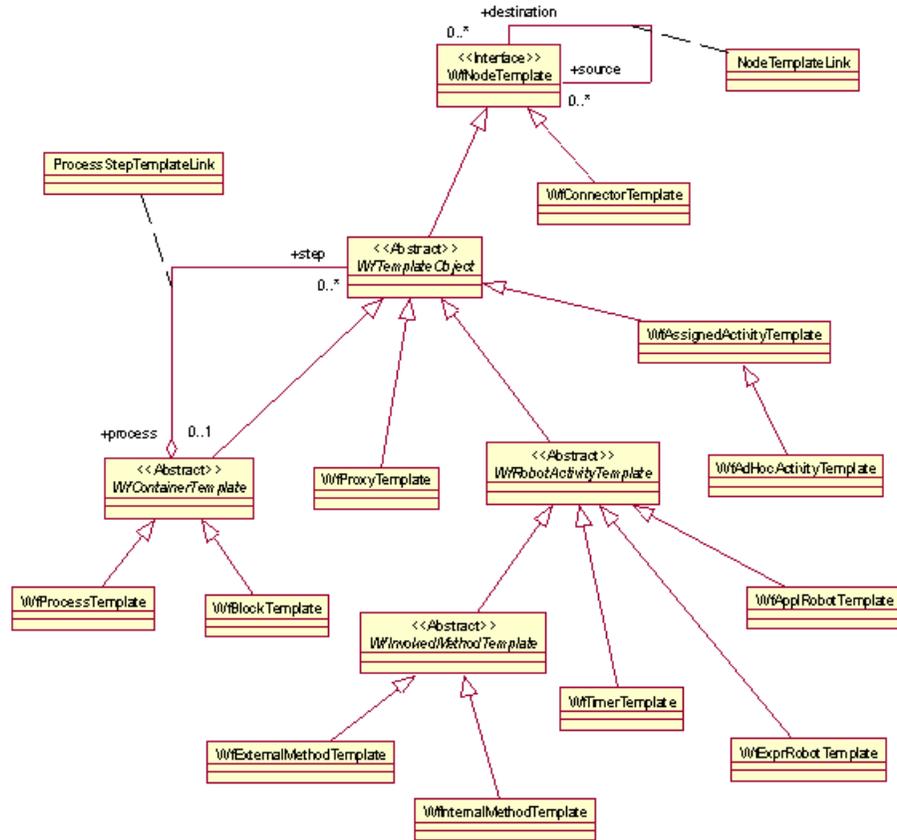
Workflow Definer Package

The base class for the workflow template objects is the WfTemplateObject class. It contains identification attributes and meta information about the data manipulated by the execution object. Its subclasses — WfProcessTemplate, WfRobotActivityTemplate, WfAssignedActivityTemplate, WfAdhocActivityTemplate, and WfProxyTemplate — are templates (factories) used to generate WfProcess, WfRobotActivity, WfAssignedActivity, WfAdhocActivity, and WfProcessReference objects, respectively.

Also shown in the following figure is the relationship of WfProcessTemplate and other template objects. A process template contains zero or more templates (activity or process templates).

WfRequester objects have no template. They are generated dynamically when the WfProcess is created, based on information contained in the processes involved.

The mapping between local and global variables does not belong to the activity definition. It is part of the relationship between a process and a step. They become part of the activity execution object when it is instantiated. In this way, an activity template can be reused independently of the process in which it will be used.



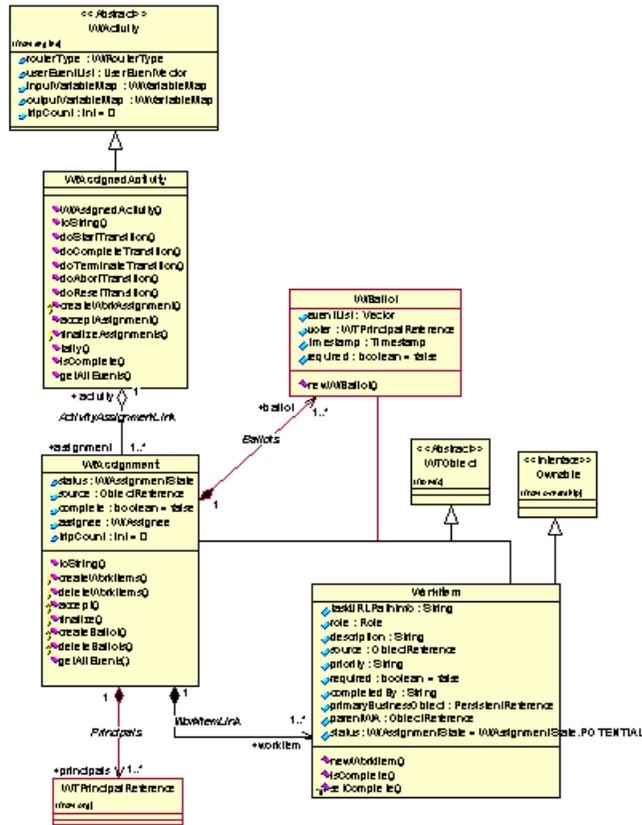
Workflow Definer package

Workflow Work Package

The workflow work package contains classes that are responsible for managing the work assigned to individual users. The activity's work may be assigned to one or more resources. These assignees can be declared to be required or optional. This relationship is modeled by the association between the WfAssignedActivity and the WfAssignment class. These two classes, along with the WorkItem and WfBallot classes, provide for parallel work item distribution and completion by the assignees, and for decision-based event propagation based on the tallied results of the completed work.

WfActivity inherits many attributes and behaviors from the WfExecutionObject, including name, description, and input/output data ('context' attribute). A WfAssignment has a status (which is not a WfState) and is not directly associated with a context. Instead, it inherits the context from the WfActivity to which it belongs. A WfAssignment maintains a set of ballots (Wfbalot) that captures

dispositions made by assignees. These ballots can then be used to tally the votes based on a voting policy to determine what routing events should fire when the activity is completed. Both WfAssignedActivity and WfAssignment have a trip counter attribute that is used to manage information generated as a result of loop iterations.



Workflow Work package

Workflow Robots Package

The workflow robots package is responsible for the execution of robot activities, that is, activities that are executed without human intervention. The WfInternalMethod class encapsulates a Windchill API and causes it to be invoked when the activity start event is fired.

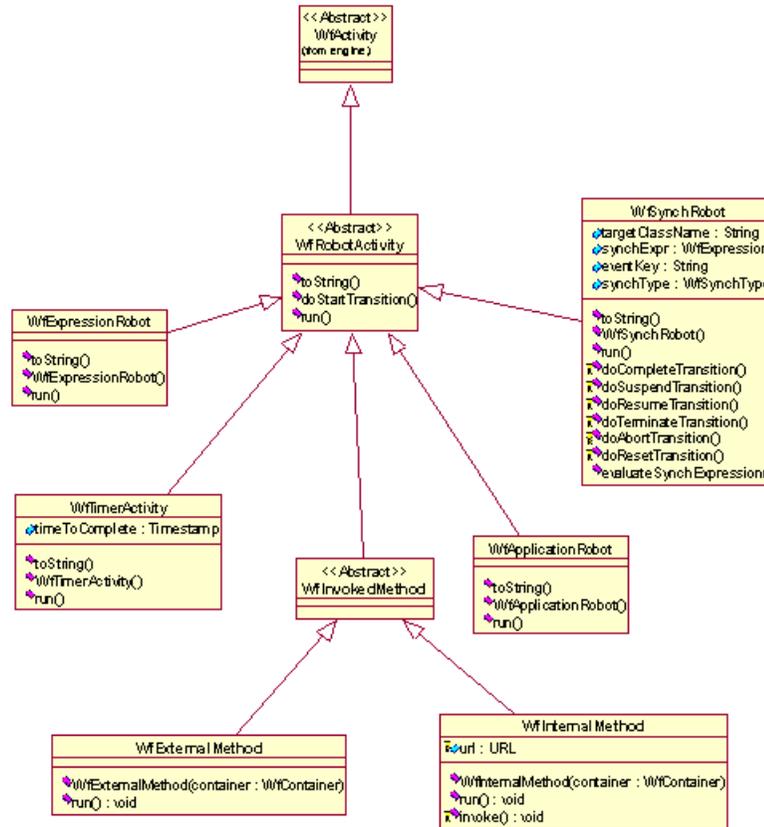
WfSynchRobot allows a workflow to be synchronized based on events that are emitted by Windchill services or an arbitrary Java expression.

WfTimerActivity allows a workflow to be synchronized on time.

WfExpressionRobot allows an arbitrary Java expression to be executed by the workflow engine.

WfApplicationRobot allows a parameterized command line to be executed by the workflow engine.

Note: WfExternalMethod is not currently used.



Workflow Robots package

Workflow Worklist Package

The workflow worklist package contains classes that construct the interface for users to view and act on their work items. The WfWorkListProcessor extends wt.templateutil.processor.GenerateFormProcessor to create an HTML work list page. WfWorkListProcessor looks at the wt.workflow.worklist.column.<#> entries in the wt.properties file for the definition of work list columns. WorkListField is an abstract class that can be extended to create new work list fields.

WfTaskProcessor also extends wt.templateutil.processor.GenerateFormProcessor to construct workflow task HTML pages. The user input to these pages is acted on by TaskCompleteDelegate, an extension of wt.templateutil.processor.FormTaskDelegate. After the delegate has processed the data, it returns control to the calling processor, which generates a response page based on the outcome of the action. Information is shared via the HTTPState object.

Similarly, the `AdHocActivitiesTaskDelegate` constructs subtemplates for defining ad hoc workflow activities. This data is processed by the `AdHocActivitiesTaskDelegate`.

External Interface

With the exception of `wt.workflow.robots` and `wt.workflow.worklist`, each workflow subpackage has its own independent service.

The `wt.workflow.engine.WfEngineHelper` provides an abstraction of the API required to instantiate and initiate workflow process instances. This service also supports the invocation of robot activities through a specification of `WfActivity`.

The `wt.workflow.definer.WfDefinerHelper` provides an abstraction of the API required to create and edit workflow definitions.

The `wt.workflow.work.WorkflowHelper` provides an abstraction of the API to the workflow service to handle work item distribution, voting, and completion.

Business Rules

Work items are created by the workflow engine and are delivered to the user through the work list interface and optionally via e-mail. An assigned activity may have many assignees. Each assignee receives a work item in their work list. The completion of an activity is based on the completion policy specified at define time. For example, an assignment to a group may declare the policy as follows: all members must act, a single member may act on behalf of the group, or a specific number of group members must act. Depending on the completion policy, when an assignee "accepts" or "completes" a work item, corresponding work items on the work list of others are removed.

`LifeCycle` definition allows the integration of workflows into the specification for each phase and gate. The life cycle service will transparently launch these workflows when the life cycle managed object enters the phase or gate.

Engineering Factor Services

The EPM classes implement a programming model designed to perform the following activities:

- Allow the loading of CAD models (including information about family tables) into a Windchill database
- Relate the CAD models to parts in the enterprise
- Support the propagation of changes made to the models into the enterprise parts

EPM provides Windchill `knower` and `doer` classes that allow the engineer to express the structure of a CAD model and to load the CAD files into the Windchill

database. EPM also provides background services that ensure data consistency in the database.

Windchill supports the concept of a *build* operation, which allows a graph of dependencies in one set of objects (called target objects) to be maintained automatically as a by-product of changes to another set of objects (called source objects). EPM implements a build operation specific to creating part objects from describing CAD documents.

For more information, see the appropriate Javadoc.

Handling CAD Models

CAD models are represented in EPM by EPMDocument objects. If the CAD model has an associated family table, each line in the family table is represented by EPMDocument. This section describes EPMDocuments and its associated links in more detail. EPMDocument

EPMDocument Model

An EPMDocument can represent a component or an assembly, using other EPMDocuments as components. (This concept is described further in the section on Handling Model Structure later in this section.)

As shown in the following model, EPMDocument implements a number of Windchill interfaces that allow it to be used in various Windchill services:

RevisionControlled

Allows the EPMDocument to be checked in and out

Baselineable

Allows the EPMDocument to be stored in a baseline

FormatContentHolder

Allows the EPMDocument to have content

DocumentVersion

Allows the EPMDocument to be treated as a document by the Windchill Explorer

BuildSource

Allows the EPMDocument to be used by the Windchill build mechanism to maintain a part structure

SupportingDataHolder

Allows the EPMDocument to have EPMSupportingData. An EPMSupportingDataHolder can hold arbitrary Java objects, each tagged with a name and an owner application name. This allows an application to store application-specific data with the EPMDocument without having to extend the Windchill schema.

EffectivityManageable

Allows the EPMDocument to be assigned an effectivity.

Storing content on EPMDocument

EPMDocument is a Windchill ContentHolder (as shown in the figure above) and, thus can contain data files, URLs, or both. This allows the engineer to upload the actual Pro/ENGINEER model file (for example, the .ASM file) and maintain it in the Windchill database.

An EPMDocument can contain a number of ContentItems. This allows EPM to store alternate representations of the CAD model (produced by the topology bus, for example) in the same document. Content is maintained using the normal Windchill ContentService.

Creating an EPMDocument

An EPMDocument must be created via the factory method newEPMDocument:

```
EPMDocument doc = EPMDocument.newEPMDocument (  
    <number >, <name >, <EPMAuthoringAppType authoring App >, <EPMDocumentType docType>, <CADName>);
```

The number of the document must be unique within the Windchill database. The ownerApplication is an EnumeratedType indicating the application that intends to manage changes to the document. The concept of owner application is no 2 set via a call to EPMContextHelper.setApplication (EPMAApplicationType appType), where EPMAApplicationType is an EnumeratedType. Set the "current application" for use by checking code. This value is cached on the client-side, but is also sent (via the EPMContextManager) to the server-side SessionContext.

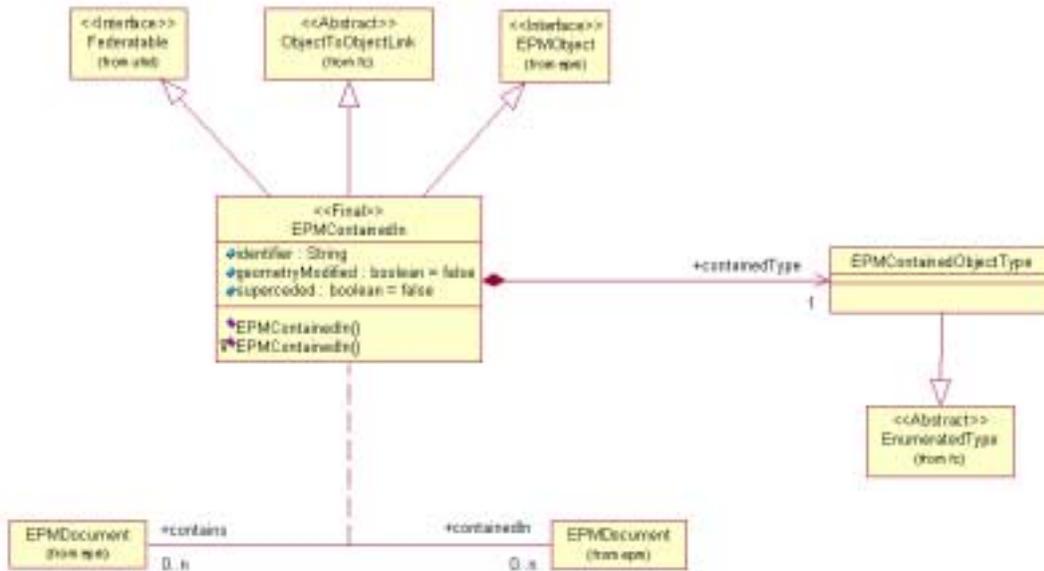
Once setApplication () has been called, all objects created will be tagged with the specified application. For Pro/E authored CADDocuments, CADName needs to be specified. EPM Services ensure that CADName is unique in a Windchill database.

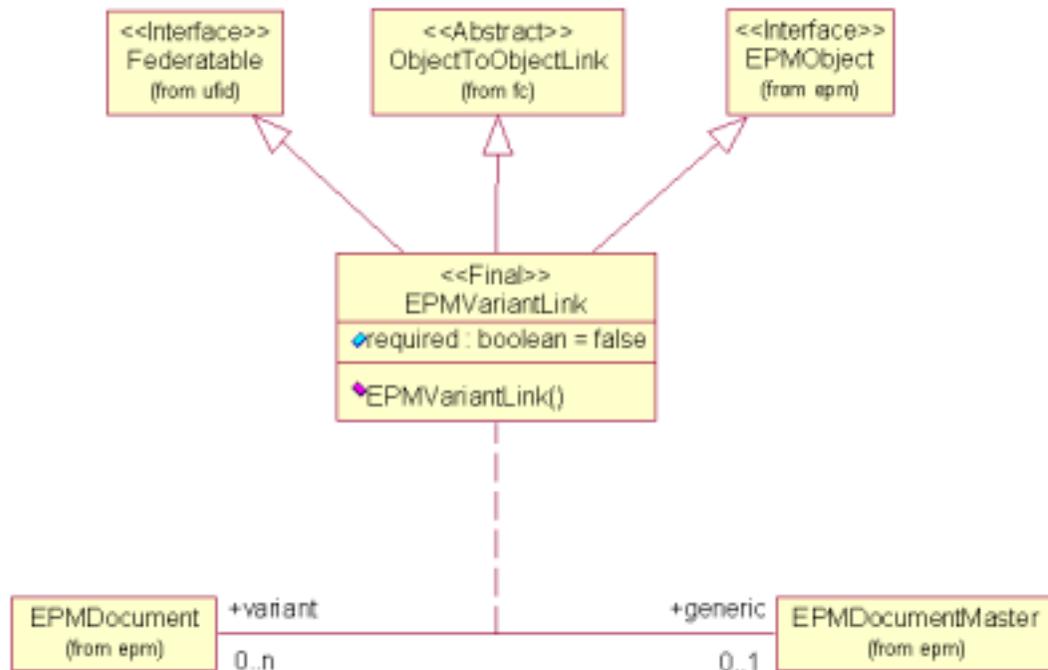
Handling Family Instances

If the CAD model has an associated family table, each line in the family table is represented by a separate EPMDocument. As shown below, relations between these EPMDocuments are handled by two types of links viz. EPMContainedIn and EPMVariantLink.

The EPM ContainedIn relation is an Object to Object link that associates two EPM Documents where one holds an object that is physically contained in the other, such as a family instance and the Pro/Engineer model that holds the family table.

EPM Variant Link is an Iteration to Master link that denotes that an EPM Document is a variant of another EPM Document.



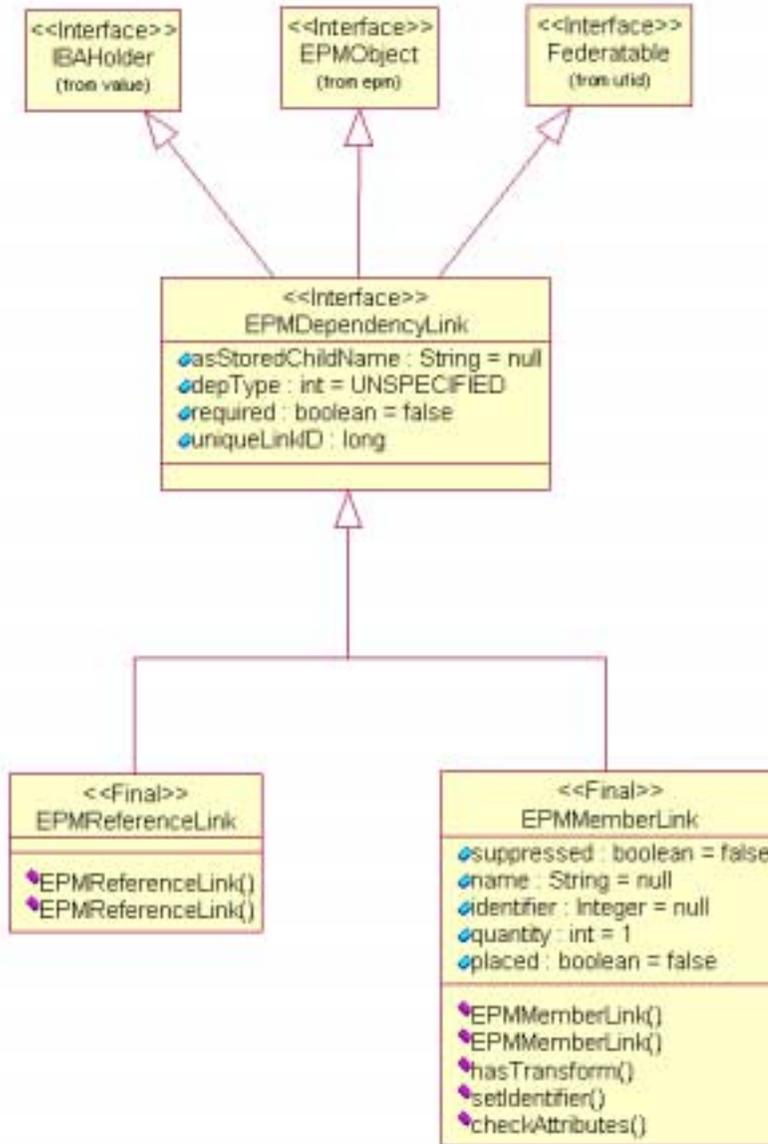


Handling Model Structure

EPMDependencyLink

When an EPMDocument represents an assembly, it *uses* other EPMDocuments via EPMMemberLinks. It can also *reference* other EPMDocuments via EPMReferenceLinks; reference links are non-structural, but are used to include objects that provide scenery, or constraints.

Both EPMMemberLinks and EPMReferenceLinks are types of EPMDependencyLink (shown in the following figure), which is used to represent the concept of one EPMDocument depending on another.



The EPMDependencyLink represents a relation between two EPMDocuments.

EPMStructureService can be used to read the EPMDependencyLinks between EPMDocuments. A number of interfaces similar to the following are available:

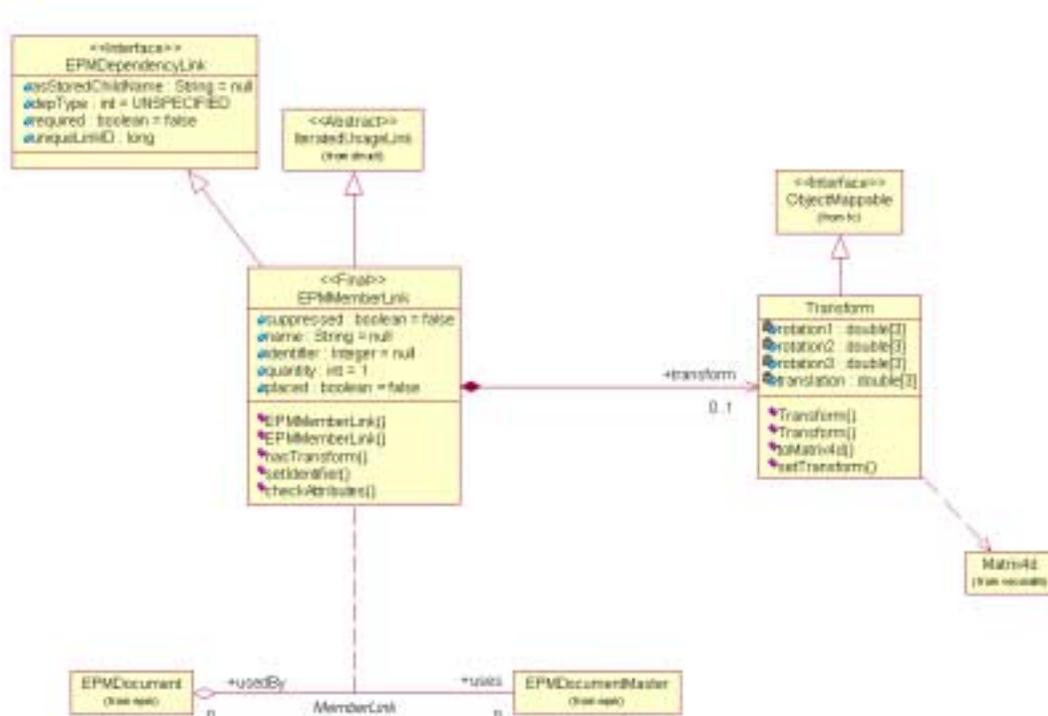
```

public QueryResult navigateUses( EPMDocument document,
    QuerySpec querySpec )
    throws WTEException;

```

EPMMemberLink

The EPMMemberLink represents the use of one model by another. It includes data such as quantity and positioning, as shown in the following figure.



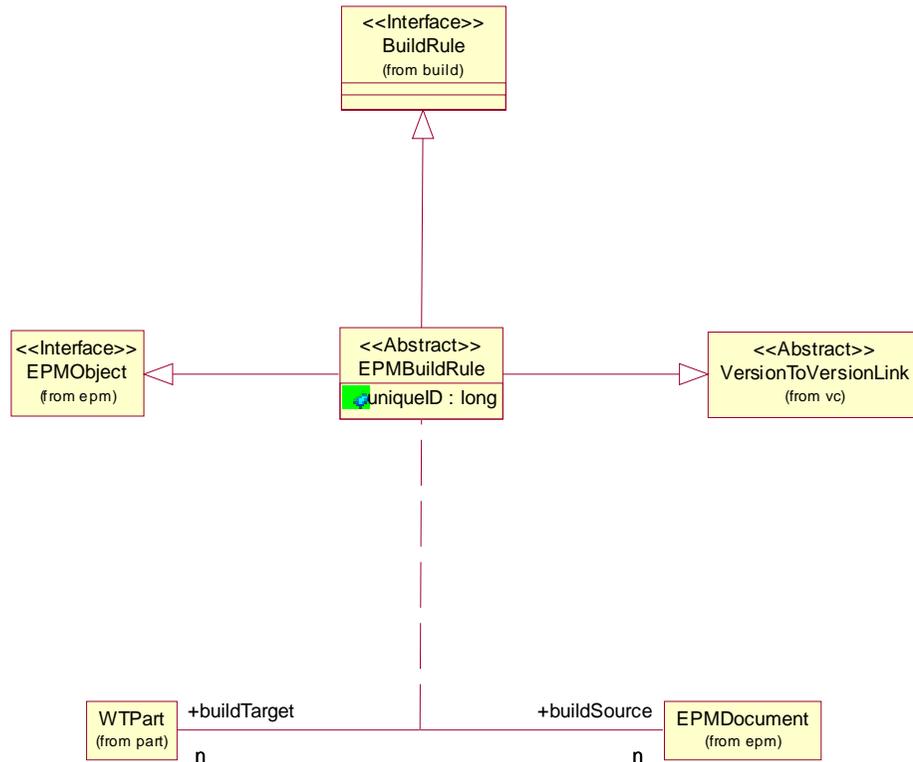
EPMReferenceLink

The EPMReferenceLink represents reference to an object by a model. While an EPMMemberLink can use only another EPMDocument, any Iterated object can be referred to (for example, a specification in Microsoft Word). The relationship between an EPMDocument containing a drawing and the EPMDocument containing the model is represented using an EPMReferenceLink. The direction of this link should be such that the drawing describes the model. The model never describes the drawing.

The Relationship Between CAD Models and WTParts

Please note that the Build Model will change in future releases. To prepare for this change, many of the classes and methods from the build package have been

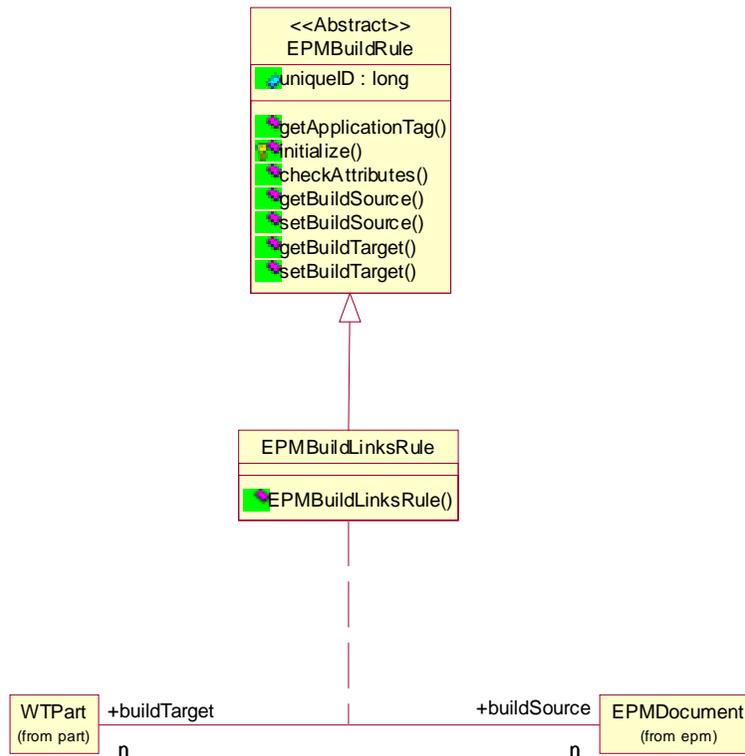
deprecated. These classes and methods will be replaced. Please refer to the javadoc to see the deprecated classes and methods. Windchill defines a build process, which is used to publish usage links and attributes (see the following figure).



Build Model

You cannot associate an EPMBuildRule to an EPMDocument that has an effectivity assigned to it. To create an EPMBuildRule, you must first have created the WTPart and EPMDocument that you intend to relate. In addition, the WTPart, but not the EPMDocument, must be either checked out or in your personal folder.

Currently, only one subtype of EPMBuildRule is implemented — EPMBuildLinksRule. This rule allows for the publication of usage links to the part structure (see the following figure).



Build Rule

To create the rule, you must use the factory method defined on the class:

```
EPMBuildLinksRule rule = EPMBuildLinksRule.newEPMBuildLinksRule (  
<document >, <part>);
```

Once this has been done, you can use the Windchill build process to publish the document structure to the part structure. This can be done in either of two ways:

- By building the EPMDocument:

```
BuildHelper.service.buildTarget (<document >,<config spec >);
```

- By building the WTPart:

```
BuildHelper.service.buildTargetsForSource (<part >,<config spec >);
```

If you are building a vaulted WTPart, the build process creates a new iteration of the WTPart. If the WTPart is either checked out or in your personal folder, no new iteration is created; instead, the usage links on the latest iteration are changed to reflect the results of the build.

7

Persistence Management

Topic	Page
Persistence Manager.....	7-2
Query	7-12
Transaction	7-16
Paging.....	7-17

The PersistenceManager is an interface for database access. To use the PersistenceManager, a class must implement the wt.fc.Persistable interface. By implementing Persistable, the classes have the necessary code generated to read and write data to and from the database and display its identity. Applications should access the PersistenceManager through a convenience class named PersistenceHelper.

Operations that fetch objects from the database perform access control checks on each object returned in the result set. These operations post POST_FIND or POST_NAVIGATE events upon successful completion. These query operations use wt.query.QuerySpec to define the classes of objects to select from the database. QuerySpec uses the wt.query.SearchCondition class to define search criteria. That is, in the SELECT statement that is generated, the QuerySpec defines which columns to select and which tables to select from, and SearchCondition defines the WHERE clause. The fetch operations return wt.fc.QueryResult.

The operations that delete, modify, and store a Persistable object all perform access control checks before continuing with the operation. These operations return the Persistable object with updated persistence information (see wt.fc.PersistInfo). These operations also post a PRE_DELETE, PRE_MODIFY, or PRE_STORE event before executing the database operation, and a POST_DELETE, POST_MODIFY, or POST_STORE event after the operation has completed successfully. Applications can listen for these events and perform some related action in the same transaction.

wt.pom.Transaction controls transaction blocks. Transaction blocks are used to group persistence operations so changes are committed only if all database operations in the block were successful. Transaction.rollback() returns the data in the database to the original state, but it is the application's responsibility to refresh objects in memory. This class should be used only by methods executing on the server, not the client.

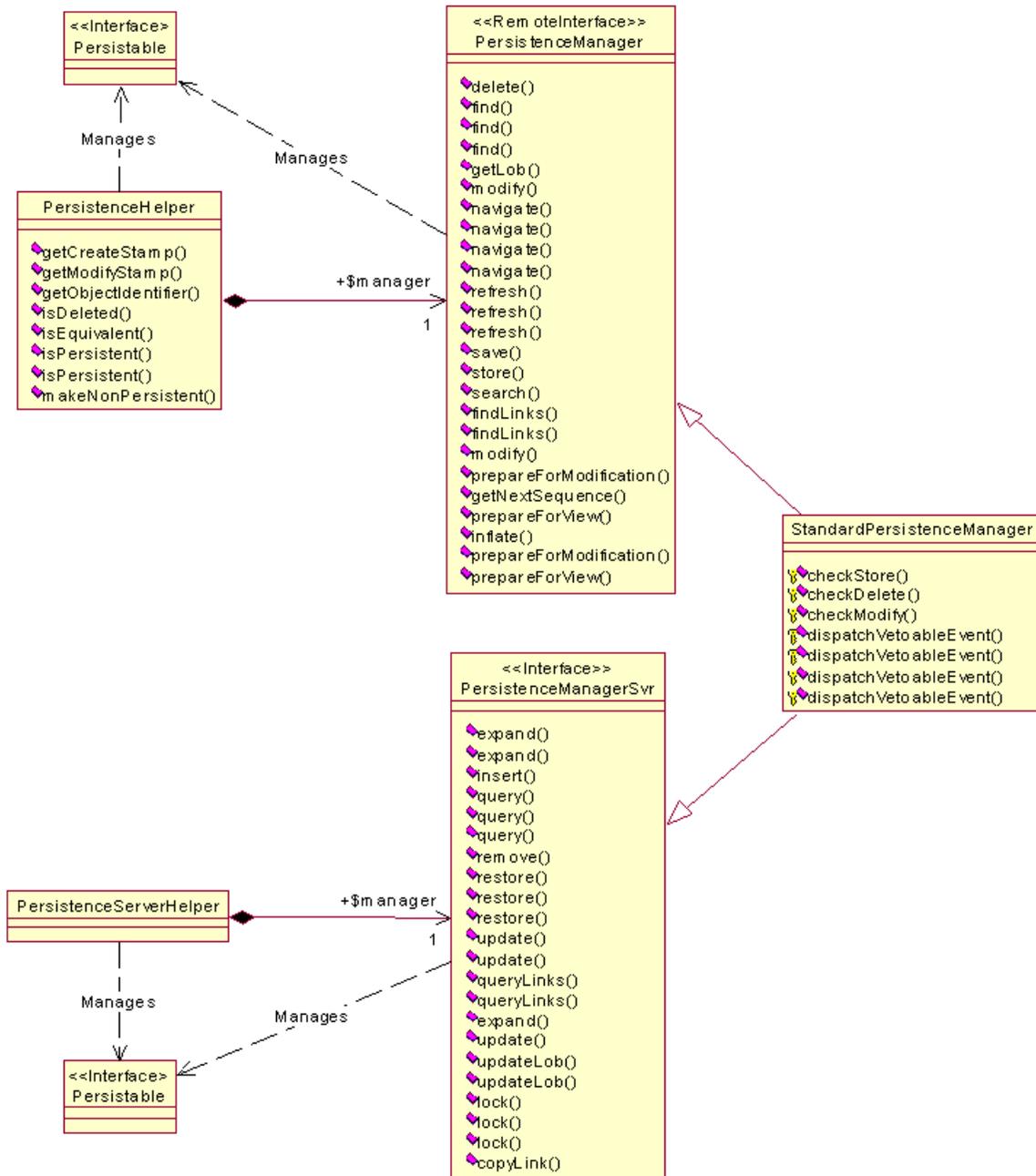
Details on the individual public operations are given in the following sections.

Support for large result sets and long running queries is provided via the paging mechanism. Much like a web search, paging allows the user to execute a query and view results in pages via fetch requests. Result pages are defined in terms of an offset and range. A paging session corresponding to a specify query is available for paging fetch requests for a limited amount of time. This is due to the limited system resources that must be used to implement the paging session. Therefore, the user must be prepared to handle exceptions caused by a paging session timeout.

Persistence Manager

The PersistenceManager is the base business manager that other managers use to handle persistence. For programming convenience, a helper class has been implemented to facilitate client (and server) usage of the PersistenceManager methods. The name of the class is PersistenceHelper. The sections that follow

show the method declarations of the methods in the PersistenceManager class and briefly describe what they do.



Persistence Manager

Store

A factory method is responsible for constructing an initialized business object in memory that the client can manipulate. The store method ensures that the state of the business object is valid and makes it persistent in a database so it can be used by the rest of the enterprise. The store method has the following declaration:

```
// Store the given object in the database
public Persistable store(Persistable wto)
    throws WTEException;
```

The store method performs the following operations:

- Dispatches a PersistenceManagerEvent.PRE_STORE event.
- Ensures that this Persistable object is not yet persistent.
- Ensures that attributes are valid (by invoking checkAttribute).
- Uses Access Control to determine if the user is allowed to create this business object.
- Inserts the Persistable object into the database and, in so doing, assigns an ObjectIdentifier and sets the createTimeStamp and modifyTimeStamp.
- Dispatches a PersistenceManagerEvent.POST_STORE
- Returns the Persistable object to the caller.

The general pattern for constructing a persistent business object is as follows:

1. Invoke the factory method to get a new business object.
2. Set the attribute values of the new object.
3. Invoke the store method to make the object persistent.

Example:

```
Customer c = Customer.newCustomer("Acme");
c.setPhone("555-1234");
c.setAddressUS(anAddress);
c = (Customer)PersistenceHelper.manager.store(c);
```

Note: Customer.newCustomer("Acme") calls initialize("Acme"), whereas Customer c = new Customer() does not. Always construct business objects with the factory method created by system generation.

Also, note that setting the local variable to the return value is important in programming the client because the underlying RMI makes copies of the objects referenced by the arguments of the method. This means that objects passed as arguments by the client in the create method are not changed by the operations that occur on the method server.

Modify

The modify method has the following declaration:

```
// Modify the given object in the database
public Persistable modify(Persistable wto)
    throws WTEException;

// Modify only the specified structured attribute in the given
// object in the database
public Persistable modify( Persistable obj, String attrName,
    ObjectMappable objAttr )
    throws WTEException;
```

The modify method performs the following operations:

- Dispatches a PersistenceManagerEvent.PRE_MODIFY.
- Ensures that attributes are valid (by invoking checkAttribute).
- Ensures that this Persistable object is already persistent.
- Uses Access Control to determine if the user is allowed to modify this business object.
- Updates this Persistable object in the database, ensuring that the object has not gone stale (that is, someone else has already modified this object in the database). Note that any attributes derived from values that exist in other tables are not updated in the database.
- Sets new values in the persistent information attributes, such as update count and modify timestamp.
- Dispatches a PersistenceManagerEvent.POST_MODIFY.
- Returns the Persistable object to the caller.

Example:

```
aCustomer.setPhone("555-0000");
aCustomer = (Customer)PersistenceHelper.manager.modify(aCustomer);
```

Note: Setting the local variable to the return value is important in programming the client because the underlying RMI makes copies of the objects referenced by the arguments of the method. This means that objects passed as arguments by the client in the modify method are not changed by the operations that occur on the method server.

Save

The save method has the following declaration:

```
// Save the given object in the database
public Persistable save(Persistable wto)
    throws WTEException;
```

The save method performs the following operations:

- Invokes the store method if the object is not yet persistent.
- Invokes the modify method if the object is already persistent.

Example:

```
Customer c = newCustomer("Acme");
c.setPhone("555-1234");
c.setAddressUS(anAddress);
// save = store
c = (Customer)PersistenceHelper.manager.save(c);

c.setPhone("555-000");
c = (Customer)PersistenceHelper.manager.save(c);
//save = modify
```

Note: Setting the local variable to the return value is important in programming the client because the underlying RMI makes copies of the objects referenced by the arguments of the method. This means that objects passed as arguments by the client in the save method are not changed by the operations that occur on the method server.

Delete

The delete method has the following declaration:

```
// Delete the given object from the database
public Persistable delete(Persistable wto)
    throws WTEException;
```

The delete method performs the following operations:

- Dispatches a PersistenceManagerEvent.PRE_DELETE event.
- Ensures the object is already persistent.
- Ensures the user is allowed to delete the object.
- Deletes the object from the database, ensuring the object has not gone stale (that is, someone else has already modified this object in the database) and flagging the PersistInfo as deleted.
- Deletes all the links in which the object is a participant.
- Dispatches a PersistenceManagerEvent.POST_DELETE event.
- Returns the non-persistent object to the caller.

Example:

```
aCustomer =
(Customer)PersistenceHelper.manager.delete(aCustomer);
```

Note: Setting the local variable to the return value is important in programming the client because the underlying RMI makes copies of the objects referenced by the arguments of the method. This means that objects passed as arguments by the client in the delete method are not changed by the operations that occur on the method server.

Refresh

The refresh method ensures that neither the client nor the server has a stale business object. The refresh method has the following declaration:

```
// Refreshes the given object
public Persistable refresh(Persistable wto)
    throws WTEException;

// Retrieves a Persistable object from the database given its object
// identifier. Object references of the target object are not refreshed.

    public Persistable refresh( ObjectIdentifier objId )
        throws WTEException, ObjectNoLongerExistsException;

// Retrieves the given Persistable object from the database to restore
// its state.
// fullRefresh should be set to true if the object references for the
// target object should be refreshed as well.
    public Persistable refresh( Persistable obj, boolean fullRefresh )
        throws WTEException, ObjectNoLongerExistsException;
```

The refresh method performs the following operations:

- Dispatches a PersistenceManagerEvent.PRE_REFRESH event.
- Ensures the object is already persistent.
- Retrieves the state for the object from the database. All persistent attributes are retrieved, including values from autonavigated relationships.
- Ensures the user is allowed to see the object, given its state.
- Dispatches a PersistenceManagerEvent.POST_REFRESH event.
- Returns the object to the caller.

```
aCustomer =
    (Customer)PersistenceHelper.manager.refresh(aCustomer);
```

Note: Setting the local variable to the return value is important in programming the client because the underlying RMI makes copies of the objects referenced by the arguments of the method. This means that objects passed as arguments by the client in the refresh method are not changed by the operations that occur on the method server.

Client code typically provides mechanisms that let you decide when business information should be refreshed. Server code should refresh the business object before performing any significant operation on that object. There is, however, one

important exception to this rule: do not refresh the target object of an event in a service event listener and then subsequently update or modify the object. Because the refresh method actually returns a new object, the update count of the event's target object will not be incremented when the update operation is called. Then, when any subsequent listeners are notified of the event and try to update it, they will get an `ObjectIsStaleException` because the update count of the in-memory object differs from that in the database.

The following example illustrates this point:

```
getManagerService().addEventListener(  
    new ServiceEventListenerAdapter(this.getConceptualClassname()) {  
        public void notifyVetoableEvent(Object event)  
            throws WTEException {  
            PersistenceManagerEvent pmEvent =  
                (PersistenceManagerEvent)event;  
            Persistable target = pmEvent.getTarget();  
  
            // target now points to a new object, not the original one  
            // obtained from the event  
            target = PersistenceHelper.manager.refresh(target)  
  
            // modify target in some way...  
  
            // the update count of target is incremented when modify is  
            // called, causing the target object of the event to look  
            // stale if another  
            // listener were to attempt a modify.  
            PersistenceHelper.manager.modify(target)  
        }  
    },  
    PersistenceManagerEvent.generateEventKey(  
        PersistenceManagerEvent.POST_MODIFY));  
}
```

Find

The find method has the following declaration:

```
// Queries for Persistable objects given a search criteria  
public static QueryResult find(StatementSpec ss)  
    throws WTEException;  
  
// Retrieves any and all link objects that exist between two Persistable  
// objects given their object identifiers. The method validates the  
// resulting set of link objects retrieved from the database before  
// returning them to the invoking method.  
  
public QueryResult find( Class targetLinkClass, ObjectIdentifier  
    obj1Oid, String obj1Role, ObjectIdentifier obj2Oid )  
    throws WTEException, InvalidRoleException;  
  
// Retrieves any and all link objects that exist between two given  
// Persistable objects. The method validates the resulting set of  
// link objects retrieved from the database before returning them  
// to the invoking method.
```

```
public QueryResult find( Class targetLinkClass, Persistable obj1,
    String obj1Role, Persistable obj2 )
    throws WTEException;
```

The find method performs the following operations:

- Queries the database for objects given the search criteria stored in the query specification (described later in this chapter in the section on utility classes). The target classes of the query specification and all of the associated subclasses are searched.
- Removes from the result set any objects the user is not allowed to view.
- Dispatches a PersistenceManagerEvent.POST_FIND event.
- Returns the filtered result set to the caller.

Example:

```
QuerySpec criteria = new QuerySpec(helpdesk.Product.class);
QueryResult results = PersistenceHelper.manager.find(criteria);
Product p;
while (results.hasMoreElements()) {
    p = (Product) (results.nextElement());
    // do something with Product p
}
```

In this example, the find method used an empty query specification, QuerySpec, to retrieve all products. The target of the QuerySpec can be an abstract class, allowing the query to span multiple subclasses. The find method returns its results in a QueryResult object that acts like an enumeration.

Navigate

There are four navigate methods:

```
public QueryResult navigate(Persistable obj, String
    role, Class linkClass) throws WTEException

public QueryResult navigate(Persistable obj, String
    role, QuerySpec criteria) throws WTEException

public QueryResult navigate(Persistable obj, String
    role, Class linkClass, boolean onlyOtherSide) throws WTEException

public QueryResult navigate(Persistable obj, String
    role, QuerySpec criteria, boolean onlyOtherSide) throws WTEException
```

The first two methods have the same result as the last two with onlyOtherSide set to True. When onlyOtherSide is true, the QueryResult contains the other side objects rather than the links. When onlyOtherSide is false, the fully populated links are returned.

The navigate method performs the following operations:

- Searches for objects of which the target object is a member, given the specified role and QuerySpec.

- Retrieves the related objects given the resulting set.
- Removes from the result set any objects the user is not allowed to view.
- Dispatches a PersistenceManagerEvent.POST_NAVIGATE event.
- Returns the filtered result set to the caller.

Example:

```
c = cust;

/* This QuerySpec constructor allows for a target and a link class. */
QuerySpec lqs = new QuerySpec(helpdesk.Product.class,
                             helpdesk.ProductRegistration.class);

lqs.appendWhere(new SearchCondition(
    helpdesk.Product.class,
    "name",
    SearchCondition.EQUAL,
    "AmazingApplets"), new int[] { 0 });

lqs.appendAnd();

lqs.appendWhere(new SearchCondition(
    helpdesk.ProductRegistration.class,
    "platform",
    SearchCondition.EQUAL,
    "NT"));

/* This will return ProductRegistrations. */
results =
PersistenceHelper.manager.navigate(c,ProductRegistration.PRODUCT_ROLE,
    lqs, false);

/* This will return Products. */
results =
PersistenceHelper.manager.navigate(c,ProductRegistration.PRODUCT_ROLE, lqs);

/* this will return all Products registered for this customer. */
results =
PersistenceHelper.manager.navigate(c,ProductRegistration.PRODUCT_ROLE,
    ProductRegistration.class,true);
/* this will return all objects where the role is product for this customer. */
results =
PersistenceHelper.manager.navigate(c,ProductRegistration.PRODUCT_ROLE,
    Link.class);
```

Get LOB

The getLob method has the following declaration:

```
public InputStream getLob( LobLocator lob )
    throws WTEException;
```

Given the lob locator, the getLob method returns the lob as an InputStream.

Get Next Sequence

The getNextSequence method has the following declaration:

```
public String getNextSequence( String sequenceName )
    throws WTEException;
```

Given the name of the sequence as input, the getNextSequence method returns the next value.

Example:

```
String sequenceNum =
    PersistenceHelper.manager.getNextSequence( "SOMENAME_seq" );
```

Note: You must create the sequence before using this. For example:

```
wtpk.createSequence( 'SOMENAME_seq', 1, 1)
```

Prepare for Modification

The prepareForModification method refreshes the given object and then checks whether the user has access control rights to modify it.

The prepareForModification method has the following declaration:

```
public Persistable prepareForModification( Persistable obj )
    throws ModificationNotAllowedException,
    ObjectNoLongerExistsException,
    NotAuthorizedException, WTEException;
```

The method throws the ModificationNotAllowedException if the user is not allowed to modify the business object but is allowed to view it. Invoke the getNonModifiableObject method on the ModificationNotAllowedException to get a refreshed copy of the object if viewing is desired.

The method throws the NotAuthorizedException if the user is allowed to neither modify nor view the given object.

The prepareForModification method performs the following operations:

- Retrieves a fresh copy of the object from the database.
- Checks the object for MODIFY permission.
- Posts a PersistenceManagerEvent.PREPARE_FOR_MODIFICATION event.
- Returns the object.
- If MODIFY permission was not allowed, READ permission is checked and a ModificationNotAllowedException is thrown with the readable object.

Prepare for View

The `prepareForView` method refreshes the given object and then checks whether the user has access control rights to view it.

The `prepareForView` method has the following declaration:

```
public Persistable prepareForView( Persistable obj )
    throws ObjectNoLongerExistsException,
    NotAuthorizedException, WTEException;
```

The method throws the `NotAuthorizedException` if the user is not allowed to view the given object.

The `prepareForView` method performs the following operations:

- Retrieves a fresh copy of the object from the database.
- Checks the object for READ permission.
- Posts a `PersistenceManagerEvent.PREPARE_FOR_VIEW` event.
- Returns the object.

Note: The `prepareForView()` method should be called only on objects that actually exist in persistent storage. Otherwise, when retrieving a fresh copy of the object, an error may occur or an invalid object may be returned.

Search

The search method queries for a persistent object using the vector of `AttributeSearchSpecifications` for the criterion. If any pre-query processing of the data is required, it performs that processing.

The search method has the following declaration:

```
public QueryResult search( Class classname, Vector searchSpecVector )
    throws WTEException;
```

The search method performs the following operations:

- Creates a `QuerySpec` for the given class.
- Creates a `SearchCondition` from each `AttributeSearchSpecification` in the `searchSpecVector` and appends each to the `QuerySpec`.
- Executes a find using this `QuerySpec` and returns the result.

Query

The `wt.query` package contains the following classes for creating queries which retrieve objects from the database.

When used with the find operation, `QuerySpec` can perform both single and multiple class queries. The following sections discuss how to perform single class

queries using QuerySpec, SearchCondition, and QueryResult. The last section describes how all three are used in multiple class queries.

The information in this section is sufficient for most database queries. If you require more advanced SQL queries, see appendix [C, Advanced Query Capabilities](#).

QuerySpec

QuerySpec is used as an argument on the find and navigate operations to define the classes of objects to retrieve. The SearchCondition object (described later in this section) defines the criteria on those classes.

QuerySpec in a Find Operation

You can specify a single class on the constructor for the find operation as follows:

```
QuerySpec(Class targetClass);
```

The following example retrieves all Customers from the database.

```
QuerySpec qs = new
QuerySpec(wt.part.WTPart.class);
QueryResult qr = PersistenceHelper.manager.find(qs);
```

Since this is usually not the desired result, search conditions are used to limit the number of objects returned. For example, to add a search condition to the QuerySpec, use the following form:

```
QuerySpec.appendWhere(WhereExpression where, int[] classIndices)
```

The classIndices parameter is used to apply the search condition to the appropriate classes in the QuerySpec. The classes referenced from the SearchCondition must match the QuerySpec classes at the specified index.

QuerySpec in a Navigate Operation

For the navigate operation, you must specify both the targetClass and linkClass on the constructor as follows:

```
QuerySpec(Class targetClass, Class linkClass);
```

The targetClass is the class of the "other side role" in the navigate. SearchConditions can be appended based on either the target or the link.

Example:

```
QuerySpec qs = new QuerySpec( wt.part.WTPartMaster.class,
                             wt.part.WTPartUsageLink.class);

CompositeWhereExpression where =
    new CompositeWhereExpression(LogicalOperator.AND);
where.append(new
SearchCondition(wt.part.WTPartMaster.class,
wt.part.WTPartMaster.NAME, SearchCondition.EQUAL, "XYZ" ));
```

```

        where.append(new
SearchCondition(wt.part.WTPartUsageLink.class,
                WTAttributeNameIfc.CREATE_STAMP_NAME,true,
                new AttributeRange(beginTime,endTime)));
qs.appendWhere(where, new int[] {0, 1});

```

SearchCondition

The number of objects retrieved by a QuerySpec is limited by criteria defined with SearchCondition. The most common format of a SearchCondition constructor is as follows:

```

SearchCondition(Class targetClass, String attributeName, String operation,
Object value)

```

The targetClass can be a concrete class, abstract class, or interface. When appended to a QuerySpec, the SearchCondition is responsible for creating a WHERE clause on the query.

The attributeName can be any attribute of the target class that maps to a column in the table being queried. In the case of a target class that has AutoNavigate associations to other classes, any attributes that map to columns in the base class or associated class can be used. Not all attribute types can be used in a search condition, for example, attributes stored in BLOB columns. To verify which attributes can be used, inspect the InfoReport for the target class, looking at the attribute's PropertyDescriptor to ensure that its QUERY_NAME property is not null.

SearchCondition has constructors for each of the java primitives and their corresponding classes, plus Enumerated and AttributeRange. AttributeRange is used as an argument to a SearchCondition constructor to create range conditions in the WHERE clause (for example, myInt BETWEEN 1 AND 50). SearchCondition also defines constants to use for the operation argument on the constructors, such as EQUAL and GREATER_THAN.

QueryResult

The QueryResult object is the standard container returned from all Persistence queries and navigations. QueryResult implements the standard java.util.Enumeration plus the added abilities to determine the number of objects in the QueryResult and reset the QueryResult to process it again.

Example: (using the QuerySpec created in the previous section):

```

QueryResult qr =
PersistenceHelper.manager.navigate(thePart,
                wt.part.WTPartUsageLink.USES_ROLE,qs,false);

```

The QueryResult in this example will contain WTPartUsageLinks with both the WTPartMaster and WTPart roles populated because onlyOtherSide is false. If onlyOtherSide had been true, QueryResult would contain WTPartMaster objects.

The SearchConditions narrow the search to only those WTPartMasters with the name of "XYZ" who had usage links created between a specified beginTime and endTime.

Multiple Class Queries

QuerySpec also supports multiple class queries when used with the find operation. Any number of classes can be added to the QuerySpec and objects of those classes will be returned from the database. The following two methods can be used for adding classes:

```
addClassList(Class targetClass, boolean isSelectable)
```

```
appendClassList(Class targetClass, boolean isSelectable)
```

The difference between the two is that appendClassList() always adds a new class to the QuerySpec. If the method addClassList() is used and the specified class is already in the QuerySpec, the index of the existing class is returned. The index returned when adding (or appending) a class is used when appending a search condition for that class.

When using multiple class queries, all of the classes usually should be joined to avoid redundant results. Use the following constructor for SearchCondition to join two classes:

```
SearchCondition(Class targetClass, String targetAttribute, Class linkClass,  
String linkAttribute)
```

When appending this type of SearchCondition to a QuerySpec, both class indexes must specify classes in the QuerySpec.

When multiple classes are in the QuerySpec, the elements in the QueryResult will no longer be Persistable objects. If the full object is selected (via the isSelectable argument, then the QueryResult elements will be an array of wt.fc.Persistable objects (that is, Persistable[]). The array is needed because more than one class can be returned. The exact type of the element will be Persistable[]. If any attributes of the class are selected, then the QueryResult elements will be an array of java.lang.Object objects (that is Object[]).

When adding (or appending) a class, the boolean parameter isSelectable specifies whether objects of that class will be returned in the QueryResult. The QueryResult Persistable array will have the same order as the classes in the QuerySpec. However, classes that are not Selectable will not appear in the QueryResult. For example, if the QuerySpec contains the classes W, X, Y, and Z with X and Z selectable, the QueryResult Persistable array will contain X and Z. All of the classes in a multiple class query are subject to Access Control regardless of whether objects of that class are returned in the QueryResult.

Example:

```
QuerySpec qs = new QuerySpec();

    // Append the classes
    int partIndex = qs.appendClassList(wt.part.WTPart.class,
true);
    int viewIndex = qs.appendClassList(wt.vc.views.View.class,
true);

    // Join the WTPart class to the View class via the View
object ID and
    // the WTPart Foreign Key
    SearchCondition sc = new SearchCondition(
        wt.part.WTPart.class, wt.part.WTPart.VIEW + "." +
        wt.vc.views.ViewReference.KEY + "." +
        wt.fc.ObjectIdentifier.ID,
        wt.vc.views.View.class, WTAttributeNameIfc.ID_NAME);

    qs.appendWhere(sc, new int[] { partIndex, viewIndex });
    QueryResult result = PersistenceHelper.manager.find(qs);
    while(result.hasMoreElements())
    {
        Persistable[] persistableArray = (Persistable[])
            result.nextElement();
        wt.part.WTPart part = (wt.part.WTPart)
            persistableArray[partIndex];
        wt.vc.views.View view = (wt.vc.views.View)
            persistableArray[viewIndex];
    }
}
```

Transaction

Transaction objects provide a mechanism that supports the standard concept of a database transaction. It has the following methods:

start

After a transaction is started, all subsequent database inserts, deletes, and updates are part of that transaction.

commit

Commits the pending database operations to the database.

rollback

Discards the pending database operations.

The pattern for a method that can throw `WTEException` is as follows:

```
Transaction trx=new Transaction();
try {
    trx.start();
    <your code here>
    trx.commit();
    trx=null;
}
finally {
    if (trx!=null)
        trx.rollback();
}
```

If you create a transaction in which to perform some activities but never reach your commit statement, be sure you mark the current transaction for rollback. Someone else may accidentally ground out an exception and later try to commit your partially completed work. The rollback call in a finally block, marks any enclosing transaction so partial results cannot be accidentally committed later. If code following notices a problem and calls rollback, the database is safe but, if your code is the deepest transaction, it is your responsibility to call rollback if you do not get to your commit call. Because you may not know for certain if your code is the deepest transaction at the time an exception is thrown, you should always do it.

Paging

The basic idea behind paging is the concept of a "snapshot" query. When a paging session is opened, a "snapshot" is taken of the results. These results can then be fetched in pages over multiple calls. When all desired fetches are complete, the paging session should be closed. This cleans up any data associated with the paging session to free system resources. These system resources are significant so a timeout property exists. If the timeout limit is reached, then the paging session is automatically closed. Any further fetch requests would result in an exception. Another configurable property is the paging limit (there is a system wide value and this value can also be overridden when opening a paging session). Because of the system resource required, a limit can be set such that if the result set size of the "snapshot" query is less than this value, then all of the results are returned immediately and no paging session is established. Note also that the results of the initial "snapshot" query are access controlled. Only data that the user can access (i.e. data that the user has read permission for) will be stored for subsequent page

requests. In addition, the total count of the size of the result set will also be returned when the paging session is opened.

This "snapshot" behavior is important to understand in terms of how the underlying query results can change. Consider a paging session that is established and a total paging size of 50 is available for fetch requests. The first 25 objects are returned and displayed to the user. The set of data can be modified by other user operations. For example, another user could delete an object in the second set of 25 objects. Now when a fetch request is made for objects 25 through 50, the object that was deleted will not be available. The paging results will still contain 25 elements. However, for the object that was deleted, a null value would be returned in the paging results. Another situation can occur for updates. Consider a paging session that is established for a query that returns data where a numeric attribute of an object is less than some value. Between the time that the paging session was opened and a subsequent fetch request, an object from the results could have been modified and the numeric attribute changed such that it no longer meets the original query's criteria. Yet, it would still be part of the paging session results because it did meet the criteria at the time that the paging session was established. This is another reason for the timeout limit on paging sessions.

The definition of the "snapshot" query uses the same query constructs as normal queries. Both QuerySpec and CompoundQuerySpec objects can be specified. The classes or column expressions that are selected will be returned in the fetch requests. Criteria and sorting will be applied when executing the "snapshot" query. The sorting (if any is specified) will be maintained on fetch requests. When specifying a sort on a QuerySpec that will be paged, each ColumnExpression must have a unique column alias. This should be specified using the ColumnExpression.setColumnAlias() method. The actual fetch requests return a PagingQueryResult, which is a sub-class of QueryResult. The PagingQueryResult has additional paging attributes such as the paging session ID (used for subsequent fetch requests) and the total paging size.

The Paging APIs are specified as static methods in the wt.fc.PagingSessionHelper class (see Javadocs for full details). There are several types of APIs that are available for opening a paging session, fetching from a paging session, and closing a paging session. The following example shows how these methods can be used to perform an interactive paging session for an arbitrary query passed to the method as an argument. The method consists of a while loop that prompts the user for an offset and range for a fetch request (or to end the paging session). The first time through the loop the paging results are null so the paging session is opened. The paging session ID and total paging count values are then stored in local variables. The paging session ID will be used to execute fetch requests and eventually close the paging session. If the paging session has already been opened, then a fetch is made using the offset and range. The paging results are then displayed along with the offset, range, and total count. The last piece of code in the loop checks the paging session ID to ensure that a paging session has been established. If the query returned no results or the paging limit was not reached, then no paging session would exist. This entire loop is enclosed in a try/finally

block to ensure that the paging session is always closed (if one has been established).

```
public void executePaging(QuerySpec a_querySpec)
    throws WTEException, WTPROPERTYVETOException
{
    long pagingSessionId = 0;

    try
    {
        PagingQueryResult pagingResults = null;
        int offset = 0;
        int range = 0;
        int totalCount = 0;
        boolean done = false;

while(true)
    {
        // Prompt user for offset and range
        offset = ..;
        range = ..;
        done = ..;
        if(done)
        {
            break;
        }

        if(pagingResults == null)
        {
            // Open Paging Session
            pagingResults = PagingSessionHelper.openPagingSession(
                offset, range, a_querySpec);
            pagingSessionId = pagingResults.getSessionId();
            totalCount = pagingResults.getTotalSize();

        else
        {
            pagingResults = PagingSessionHelper.fetchPagingSession(
                offset, range, pagingSessionId);
        }

        // Display QueryResult items
        System.out.println("Displaying " + offset + " to " +
            (offset + range) + " of " + totalCount);

        if(pagingSessionId <= 0)
        {
            // No paging session was established
            break;
        }
    }
}
finally
{
    if(pagingSessionId > 0)
    {
        PagingSessionHelper.closePagingSession(pagingSessionId);
    }
}
```

```
}  
}
```

8

Developing Server Logic

Topic	Page
Service Management	8-2
Service Event Management.....	8-3
Implementing Business Data Types	8-7
Lightweight Services	8-17
Updating a Master Through an Iteration	8-22

Developing server logic is primarily a task of designing and implementing business data types and services. To develop these kinds of abstractions, you must understand applicable Windchill design patterns, service and event management in general, and guidelines used in detailed design and implementation.

A business data type can be characterized as an important piece of information known in the business domain. This information consists of data and/or state once it exists and is recognized in the system. A business service can be characterized as a set of one or more functions that carry out business-specific processing. The purpose of these kinds of abstractions is to separate the entity objects from control objects. Once they are separated, developmental impacts and risks, typically caused by evolving requirements and behavior, are reduced. That is, data and behavior changes are less likely to directly affect each other given sound abstraction and encapsulation.

In addition to the information presented here, see also Appendix B, Windchill Design Patterns. This Appendix describes design patterns that represent Windchill's current best practices regarding development of server logic.

The *Windchill Customizer's Guide* also includes descriptions of how to use and customize Windchill services used in server development.

Service Management

The development of the standard, reusable Windchill services caused the need for a general mechanism to manage the behavior of and interaction between these services. This service management mechanism specifies a protocol for startup, shutdown, and communication between Windchill services.

Automatic Service Startup

The `wt.properties` configuration file includes entries that specify which services should be started automatically when the server is launched. In addition, each entry specifies a pairing between the service interface and which implementation of the service should be constructed. That is, if other implementations are supplied in addition to the standard implementation of the service, one of the other implementations can be declared to execute in these entries. On the other hand, multiple service interfaces can also be implemented by a single class. For example, the `PersistenceManager` and `PersistenceManagerSvr` are both implemented by `StandardPersistenceManager`. The following is an example of several entries for the standard Windchill services:

```
# Application Services
wt.services.service.1 = wt.session.SessionManager/
    wt.session.StandardSessionManager
wt.services.service.2 = wt.queue.QueueService/
    wt.queue.StandardQueueService
wt.services.service.3 = wt.fc.PersistenceManager/
    wt.fc.StandardPersistenceManager
wt.services.service.4 = wt.fc.PersistenceManagerSvr/
    wt.fc.StandardPersistenceManager
```

The number indicates the startup order of the service. If a service depends on other services, those services must be started first.

There are entries that are termed either a service or a manager. During the initial stages of Windchill development, the use of the terms manager versus service was unclear. It has been agreed that service represents the more general concept and manager represents those services that are characterized more as managing groups of objects. Currently the terms manager and service appear interchangeably.

Service Startup and Shutdown

In order for services to be managed, they must implement the `wt.services.Manager` interface. This interface specifies methods for starting and stopping services.

A reference implementation of `wt.services.Manager`, named `wt.services.StandardManager`, provides most of the base functionality required for service management. If a new type of service does not require any special startup or shutdown processing, it can extend the base class `wt.services.StandardManager` without overriding any of its methods.

Two methods in class `wt.services.StandardManager` are intended to be overridden to specialize startup and shutdown processing. These methods are `performStartupProcess` and `performShutdownProcess`. Examples of startup processing include subscribing to service events and establishing queues for use in background processing. For further information, see [Service Event Subscription](#) later in this chapter.

Service Management

`ManagerService` is a manager which is used to startup and provide access to a pre-defined list of managers. This list includes different managers for services mentioned in [Chapter 6, Windchill Services](#).

In addition to managing managers, the `ManagerService` provides a synchronous event dispatch service. It could dispatch a vetoable or non-vetoable event to all listeners for the event key. The listener may or may not object to the event. It performs a synchronous "in thread/transaction" notification of each event listener for the event branch identified by the event key. It calls the `notifyEvent` operation on each subscriber

Service Event Management

One means of interservice communication is by direct peer-to-peer cooperation and collaboration. No special mechanisms are required to support this kind of communication because it is already defined as explicit invocations.

However, with a plug-and-play architecture, services become more independent and unaware of other services. To facilitate interservice communication between

these autonomous services, general mechanisms are needed to convey information between these services and to manage unordered synchronous communications.

Each service is responsible for specifying service events that can be emitted when significant events that may be of interest to other services occur within the service. Also, each service must establish listeners to other service events in order to be notified of and react to other significant events occurring within the system.'

Service Event Registration

Services can register their events with the `wt.services.StandardManager` when they are started. Registering service events at startup time makes the events known to processes, such as access control. The following example illustrates how events are registered in an overridden `wt.services.StandardManager`'s `registerEvents` method for the version control service:

```
public void registerEvents( ManagerService manager ) {
    manager.addEventBranch(
        VersionServiceEvent.generateEventKey(
            VersionServiceEvent.NEW_VERSION ),
        VersionServiceEvent.class.getName(),
        VersionServiceEvent.NEW_VERSION );
    manager.addEventBranch(
        VersionServiceEvent.generateEventKey(
            VersionServiceEvent.PRE_SUPERSEDED ),
        VersionServiceEvent.class.getName(),
        VersionServiceEvent.PRE_SUPERSEDED );
    manager.addEventBranch(
        VersionServiceEvent.generateEventKey(
            VersionServiceEvent.POST_SUPERSEDED ),
        VersionServiceEvent.class.getName(),
        VersionServiceEvent.POST_SUPERSEDED );
    manager.addEventBranch(
        VersionServiceEvent.generateEventKey(
            VersionServiceEvent.PRE_ROLLBACK ),
        VersionServiceEvent.class.getName(),
        VersionServiceEvent.PRE_ROLLBACK );
    manager.addEventBranch(
        VersionServiceEvent.generateEventKey(
            VersionServiceEvent.POST_ROLLBACK),
        VersionServiceEvent.class.getName(),
        VersionServiceEvent.POST_ROLLBACK);
    manager.addEventBranch(
        VersionServiceEvent.generateEventKey(
            VersionServiceEvent.PRE_ROLLUP),
        VersionServiceEvent.class.getName(),
        VersionServiceEvent.PRE_ROLLUP);
    manager.addEventBranch(
        VersionServiceEvent.generateEventKey(
            VersionServiceEvent.POST_ROLLUP),
        VersionServiceEvent.class.getName(),
        VersionServiceEvent.POST_ROLLUP);
}
```

Note: A more implicit means of registering events is by not doing so in the `registerEvents` method, but by allowing the event to be registered when it is first emitted. Once this occurs, all listeners subscribing to this event will be notified.

Service Event Subscription

In order for services to be notified of events occurring within the system, they must subscribe to each particular service event of interest. To do this, you must invoke the `wt.services.ManagerService.addEventListener` method with a listener and key identifying the event of interest.

The listener specified during subscription must implement the `wt.events.KeyedEventListener` interface. It defines the `notifyEvent` and `notifyVetoableEvent` methods. The `wt.services.ServiceEventListenerAdapter` class is provided as a utility whereby a listener can extend it and override only the methods that are required for notification. The `notifyEvent` is a general method that can be used in all subscription cases. The `notifyVetoableEvent` method is more specialized; its intended use is to provide a means of vetoing an event via an exception.

Typically event listeners are implemented using either anonymous or named instances of inner classes. This allows the outer class to be designed without implementing the `wt.events.KeyedEventListener` interface. This keeps the outer class pure in that its type directly reflects its purpose without having to implement a listener interface. The following is an example of how to override the `performStartupProcess` method and define and add a listener for a specific event:

```
protected void performStartupProcess()
    throws ManagerException {

    // At a request prior to a modification, if the target is a
    // lockable object then
    // validate if a modify on the object would be accepted. If
    // not then veto it.

    getManagerService().addEventListener(
        new ServiceEventListenerAdapter(
            this.getConceptualClassname() ) {
                public void notifyVetoableEvent( Object event )
                    throws WTEException

                    PersistenceManagerEvent pmEvent =
                        (PersistenceManagerEvent)event;
                    Persistable target = pmEvent.getTarget();

                    if (target instanceof Lockable)
                        validateLock( (Lockable)target);
                }
            },
        PersistenceManagerEvent.generateEventKey(
            PersistenceManagerEvent.PREPARE_FOR_MODIFICATION ) );
}

protected void validateLock( Lockable object )
    throws WTEException, LockException {
```

```

if (object.getLock() != null) {
    if (object.getLock().isSeized()) {
        if (!object.getLock().getLocker().getObjectId().equals(
            (Object) PersistenceHelper.getObjectIdentifier(
                SessionHelper.manager.getPrincipal() )))
            throw new LockException( RESOURCE, "5", null );
        }
    }
}

```

Service Event Notification

When a service's listeners have been subscribed, a list of them and others for a particular event is maintained by the `ManagerService`. This list is traversed in an unordered fashion and each listener is executed one after the other using synchronous Java method invocations. Listener notification methods execute in the same thread and database transaction as the event emitter. This means that database operations performed by a listener in response to a notification call are included in the transaction of the event emitter. The following is an example that shows the dispatching of an event like

`PersistenceManagerEvent.PREPARE_FOR_MODIFICATION`:

```

protected void dispatchVetoableEvent( String eventType,
    Persistable obj )
    throws WTEException {
    PersistenceManagerEvent event =
        new PersistenceManagerEvent( this, eventType, obj );
    getManagerService().dispatchVetoableEvent( event,
        event.getEventKey() );
}

```

Service Event Exception Handling

When a listener vetoes an event, it does so by throwing an exception. When an exception is thrown, each listener that has already been notified of this event can handle this exception by either passing the exception through, catching and re-throwing it to clean up, or by implementing a "finally" clause to clean up.

As discussed earlier, event notification is dispatched within the same thread and transaction frame as the event emitter. This means that an event emitter must roll back any database updates that it has made if there has been a veto by another listener. The following example illustrates this guideline. If the `POST_STORE` event is vetoed, control is transferred to the "finally" clause, where the transaction will be rolled back because the line that set the transaction to null was never reached.

```

public Persistable store(Persistable obj) throws WTEException {
    checkStore(obj);
    obj.checkAttributes();
    dispatchVetoableEvent( PersistenceManagerEvent.PRE_STORE,

```

```

        obj );
    Transaction trx = new Transaction();

    try {
        trx.start();
        insert(obj);
        dispatchVetoableEvent( PersistenceManagerEvent.POST_STORE,
obj );
        trx.commit();
        trx = null;
    }

    finally {
        if ( trx != null )
            trx.rollback();
    }

    return obj;
}

```

Service Event Conventions

The meaning of any particular service event is up to the designer of the event. Events should include all the information required by event listeners for follow-on processing. The `wt.events` package includes a base event class, `wt.events.KeyedEvent`, which can be extended to create new event types.

A common event design pattern includes pre- and post-events. Pre-events are used to indicate to listeners that an event is about to begin. They are typically intended to give listeners the opportunity to validate and possibly veto the event. Post-events are used to notify listeners that an event has finished. They are most useful when a service needs to perform some kind of post-processing as a result of the event. Not all events have to be either pre- or post-events. A singular definition of an event can be used to indicate some special occurrence, which may or may not have happened.

Implementing Business Data Types

A business data type can be characterized as an entity object (that is, a knower) that deals basically with abstracting some crisp and clearly definable piece of information in the problem domain. Therefore, business data types are primarily defined by data, not behavior, as in control objects (that is, doers). Because most of the specification of a business data type is by its attributes, implementation of this business information typically focuses on the classes' attributes.

In addition, because business data types are entity objects and are typically lightweight compared to control objects, they efficiently transport information between an application and the server in the three-tier (client, server, and database) architecture.

Initializing Business Attributes

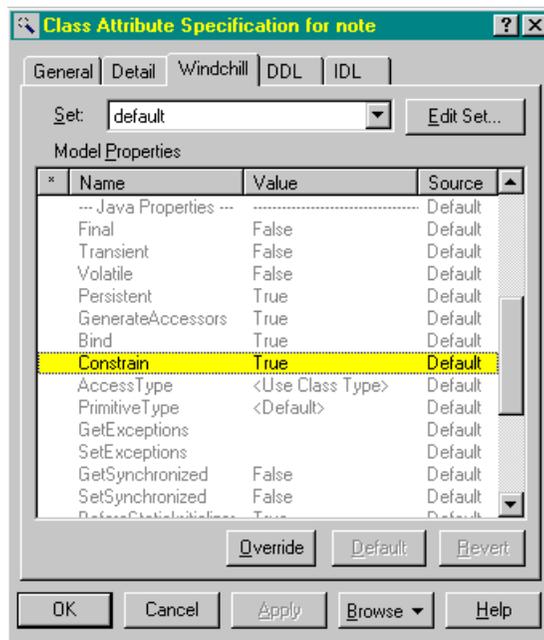
Initialization of business data type attributes is done by implicitly code generated "initialize" methods. No constructors are generated for modeled business data types. Instead, factories are created for each modeled constructor and, within the factory, an instance is made. Then the initialize method with a signature that matches the factory is invoked.

These initialize methods are empty at the first code generation and must be hand-implemented because the code generator currently cannot make any assumptions on what attributes should or should not be initialized. By default, once these methods are implemented, their code is preserved for subsequent use. To mimic Java constructor chaining, each initialize method should always invoke its parent initialize method before doing anything else.

Business Attribute Accessors

Attributes are always generated as private fields with two accessors: a getter and setter method, like JavaBean properties. The getter is generated as a no-arg method that returns the value of the attribute.

The setter is generated as a single-arg method which sets the attribute to the value of the arg. Depending on whether the attribute is constrained, the setter is generated with or without a `wt.util.WTPropertyVetoException` throws clause. This exception extends `java.beans.PropertyVetoException` and allows for a customized message. The following example shows the property for constraining an attribute, and a getter and setter for an attribute named "note".



Constraining an Attribute

```

public String getNote() {
    ///##begin getNote% [ ]348C64E401C5g.body preserve=no
    return note;
    ///##end getNote% [ ]348C64E401C5g.body
}

protected void setNote( String a_Note )
    throws WTPPropertyVetoException {
    ///##begin setNote% [ ]348C64E401C5s.body preserve=no

    noteValidate( a_Note ); // throws exception if not valid
    note = a_Note;
    ///##end setNote% [ ]348C64E401C5s.body
}

```

Note that the body of the methods are flagged as "preserve=no." This instructs the code generator to overwrite the code within a method. Getters and setters can be preserved by setting this flag to "yes", but in general this is not recommended. On the other hand, the code generator can be instructed to not generate a getter and setter for an attribute with the "GenerateAccessors" property on the Windchill tab set to "False."

Overriding Accessor Methods

Typically, overridden accessors do the following:

- Attribute validation (as shown in the example)
- Lazy attribute initialization
- Computed attribute access

The following is an example of overriding accessor methods.

```

public void setStatus(int status)
    throws PropertyVetoException {
    // Set the status to the closed-code only if
    // closure comments were provided.
    if (status == CLOSE_CODE) {
        if ((getClosureComments() == null) ||
            (getClosureComments().equals("")) {
            throw new PropertyVetoException();
        }
    }
    super.setStatus(status);
}

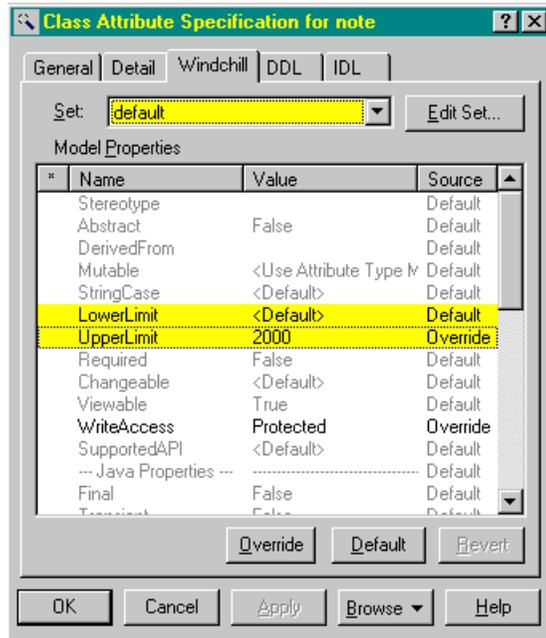
```

Validating Business Attributes

Validating business data type attributes is handled at two distinct levels. The simplest and completely code-generated level is implemented as a validation method for a constrained attribute that is invoked by its setter. The mechanism used to veto the setting of an attribute is a `wt.util.WTPPropertyVetoException`

capable of being thrown from the setter. If an attribute is not constrained, the setter is generated without the capability of an exception being thrown.

The properties for specifying limits on a String or numeric attribute are LowerLimit and UpperLimit. In both cases, the value supplied is treated as a literal or constant. This means that the validation method is code generated with "if" statements that use Java less than or greater than operators to test against the specified value in the LowerLimit or UpperLimit property, respectively. The following examples illustrate the use of these two properties and the validation code that is generated automatically:



Validation Example

```
private void noteValidate( String a_Note )
    throws WTPPropertyVetoException {

    if ( a_Note != null &&& a_Note.length() > MAX ) { // upper limit check
        Object[] args = { "note", "MAX_LENGTH" };
        throw new WTPPropertyVetoException( "wt.fc.fcResource",
            wt.fc.fcResource.UPPER_LIMIT, args,
            new java.beans.PropertyChangeEvent( this, "note", note, a_Note ) );
    }

    if ( a_Note != null &&& a_Note.length() < MIN ) { // lower limit check
        Object[] args = { "note", "MIN_LENGTH" };
        throw new WTPPropertyVetoException( "wt.fc.fcResource",
            wt.fc.fcResource.LOWER_LIMIT, args,
            new java.beans.PropertyChangeEvent( this, "note", note, a_Note ) );
    }
}
```

The other more general level of validating one or more attributes is to override and implement the "checkAttributes" method inherited from wt.fc.WTObject. This method is invoked before the object is stored in the database initially and every time it is modified. In this case the exception thrown is wt.fc.InvalidAttributeException, not wt.util.WTPropertyVetoException.

Implementing the checkAttribute Method

The checkAttributes method shown in the following example is an alternative to the setStatus method shown in the example in Overriding Accessor Methods earlier in this section. It ensures that closure comments are provided if the status is to be set to 2.

The checkAttributes method is called by the PersistenceManager to ensure the state of the object is correct before storing it in the database.

The following is an example of implementing the checkAttribute method.

```
// Example checkAttributes method
public void checkAttributes() throws
    InvalidAttributeException {

    // Ensure that closure comments are provided if the
    // status is set to the closed-code.
    if (getStatus() == CLOSE_CODE) {
        if ((getClosureComments() == null) ||
            (getClosureComments().equals(""))) {
            throw new InvalidAttributeException ();
        }
    }
}
```

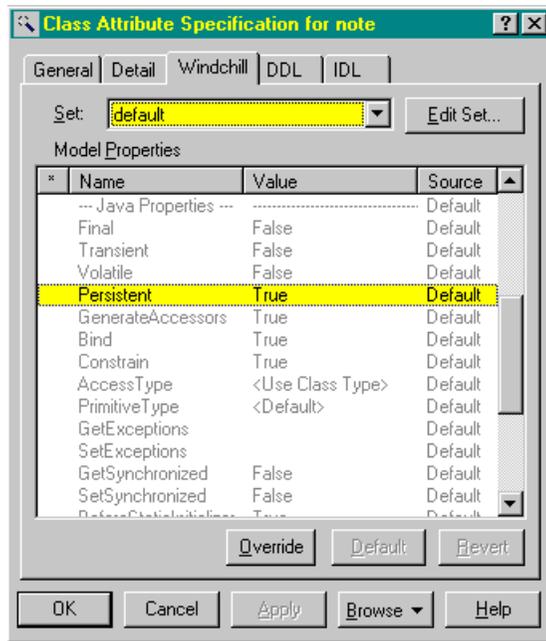
Business Attribute Aggregation

Structured attributes (that is, non-first-class) at all levels of associations are placed in the same database table as the first-class object that stores the structure attributes. The aggregated attributes must be object mappable and, regardless of cardinality, will always exist in the database table as a flat structure. The flatness of the tables is due to the fact that the database tier in the three-tier architecture is assumed to be a relational database, not an object-oriented database.

However, if the top-level structured attribute's cardinality is 0..1 or 0..* and it is not flagged as being required, it can be nullified in the database. Care must be taken if this top-level structured attribute aggregates any other structured attribute because the nested structured attributes will not be nullified.

Business Attribute Persistence

A business attribute is persisted as a column in a first-class object's database table if it is flagged as being "Persistent" as shown in



Persistent Attribute

Otherwise, it is treated as an in-memory attribute only and is not mapped to a column in the database table.

Business Attribute Derivation

Derived attributes are strictly generated as getters and setters without any such named field for the modeled derived attribute. Typically, the purpose of modeling a derived attribute is to make it act as a computed value when accessed. One side effect of derived attributes is they are neither serializable nor externalizable. They could be an implementation detail in the body of a class, but Windchill's generated externalization methods will not recognize the attribute as existing. That is, the externalization methods to read and write an object's data are generated based on what is known in the object model, not the implementation.

If an object with a derived attribute is either being transmitted across-the-wire, or is being stored/retrieved externally, the derived attribute within it is non-existent. Any assumption made in implementation where a derived attribute is required for processing after the object has been serialized or externalized may produce erroneous results unless the accessor is called to re-derive the attribute.

Implementing Business Services

A business service can be characterized as a set of abstractions — based on the business service design pattern — that collectively provide essential behavior and processing for a problem domain. For further information on the business service design pattern, see Appendix B, Windchill Design Patterns. The main kinds of classes in a business service act as control objects (that is, doers) that carry out processing on business objects. Additionally, cookies are maintained by the business service that hold state and key information on a per object basis.

Cookie classes are aggregated to interfaces that are managed by business services. For example, `PersistInfo` is a cookie class that is aggregated to the `Persistable` interface. Any class that implements `Persistable` will also aggregate `PersistInfo`, and the `PersistenceManager` business service will manage the attributes of `PersistInfo` such as `createStamp` and `modifyStamp`.

Initializing Business Services

Business services are designed to execute as singletons in the server only. When the server is launched, the `wt.properties` file is read to determine what services are to be started automatically. These specified services are constructed and statically initialized, service events are registered, and the services are then started. The ordering specified by `wt.services.service` entries in the `wt.properties` file controls the order in which services are started.

If administrative access is required on service startup, it is necessary to create a new `SessionContext` and set the current user to the Administrator. It is equally important to reset the session context at the end of a service's startup processing. To guarantee that a session context is always reset, it should be done in a "finally" clause.

```
protected void performStartupProcess()
    throws ManagerException {

    SessionContext previous = SessionContext.newContext();
    // create new SessionContext
    try {
        // do startup processing such as registering
        // event listeners

        try {
            SessionHelper.manager.setAdministrator(); //
        }
        catch (UserNotFoundException wex) {
            System.err.println ("Administrator user
                doesn't exist (OK if installation)");
            return;
        }

        // do startup processing that requires
        // administrative permissions such as
        // creating a queue
    }
    catch (WTException wex) {
```

```

        throw new ManagerException (this,
            "Failed to initialize service.");
    }

    finally {
        SessionContext.setContext(previous);
        // restore initial SessionContext
    }
}

```

Business Service Operations

Business service operations can be invoked either locally or remotely. Only remote invocations are made by a client. Local invocations can be either from a client or server.

The business service's Helper abstraction contains only methods that can be invoked locally. These methods are generally used for the accessing of the service's cookie information. Other types of methods that aid in processing may also be implemented in a Helper. The Service abstraction contains methods that can be invoked locally on the server and invoked remotely from the client if the Service is stereotyped as being a "RemoteInterface." With this stereotyping and aggregation of the "service" in the helper, all public methods are available for use. The business service should ensure that all externally available operations are exposed either through the Helper or Service class. All other operations should only be available internally with the possible exception of methods on utility classes that are used by other services.

Often business service operations executing on the server must perform database actions. To maintain database integrity, service independence, and loose coupling between other services, database changes should be protected by the use of transaction blocks. The following section of sample code serves as a guideline on how to implement transaction blocks:

```

public Persistable store( Persistable obj )
    throws WTEException {
    //##begin store% [ ]3458AD98008C.body preserve=yes

    Transaction trx = new Transaction();

    try {
        trx.start();
        dispatchVetoableEvent( PersistenceManagerEvent.PRE_STORE,
            obj );
        checkStore(obj);
        obj.checkAttributes();
        insert(obj);
        dispatchVetoableEvent( PersistenceManagerEvent.POST_STORE,
            obj );
        trx.commit();
        trx = null;
    }
    finally {

```


Lightweight Services

Lightweight services reside in the application layer between the client and the business service layer. Lightweight services are *light* because they do not start automatically and dispatch events. Consequently, lightweight services are not specified in the `wt.properties` file with `wt.services` entries.

Lightweight service methods should be designed to do the following:

- Reduce the number of round trips between the client and the server.
- Provide task-specific and higher-level functionality than business service methods.
- Ensure client transactional integrity.

Lightweight services can dispatch events but should not listen for them. If a service is not started automatically, it will not be able to hear events that it is supposed to listen for until it is started.

Lightweight services are an effective means to ensure client transactional integrity. Several client-server operations can be grouped into a single, lightweight service method call that will carry out these operations on the server in a single transaction.

Lightweight services can be implemented in the following two ways:

- Through a modeled class that extends `wt.services.StandardManager`
- Through a non-modeled, inner class that implements `wt.method.RemoteAccess`

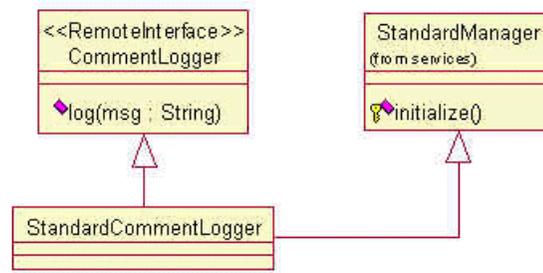
The following subsections describe both implementations.

The Modeling Implementation

Lightweight services are modeled like business services in that they are defined by a remote interface and have an implementation class that extends `StandardManager`. The remote interface enables the service to run in the method server and be invoked by remote clients (for example, applets). The mapping between the service interface and implementation class is accomplished with a naming convention. The implementation class for a lightweight service must have the same name as the interface with "Standard" at the beginning of the name.

To use a lightweight service, create an instance of the generated Forwarder class and call the service's methods through the Forwarder. This is similar to a business service where a static attribute on the Helper class is initialized with an instance of the Forwarder class. In both cases, the mechanism for invoking service operations is identical.

The following is an example of a lightweight service named CommentLogger. This model can be found in WTDesigner.mdl in the wt.services.test package.



CommentLogger Service Example

In this example, the CommentLogger interface defines a lightweight service that will log a message to the server log file. This service could be useful for applet clients that want to include a message in the server log. Without such a service, all System.out.println calls will send their output to the browser’s Java console window instead of the server log file. StandardCommentLogger implements this service by extending wt.services.StandardManager and the CommentLogger interface. Note that the name must be StandardCommentLogger in order for the ManagerService to find the implementation class for the CommentLogger service. Since CommentLogger has a stereotype of RemoteInterface, a Forwarder class named CommentLoggerFwd is automatically generated. It is this class that is used to invoke the CommentLogger service methods:

```

CommentLoggerFwd logger = new CommentLoggerFwd();

logger.log("a message");
  
```

The Inner Class Implementation

The second way of implementing a lightweight service is through a non-modeled, inner class that implements wt.method.RemoteAccess. An example of an inner class implementation follows. The objectives are to invoke a complex server-side activity, collecting information of interest to the client, while minimizing the number of classes loaded by the client. These techniques improve performance for the client because loading classes can be slow and expensive.

In this example, note the type of the inner class. To avoid IllegalAccess violations, the forwarder must be declared as a "public static class". Using this declaration, the MethodServer is able to instantiate and invoke methods on the inner class.

Note also the technique for invocation of the desired server method. To avoid loading classes specified as part of the action to be carried out in the MethodServer, we specify the target inner class and method of interest as Strings. Any reference to the class itself will cause it and all the classes it needs to be loaded into the client VM, which is not desirable.

```

package wt.appfwd;
  
```

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.InvocationTargetException;
import java.rmi.RemoteException;
import java.io.Serializable;
import java.util.Vector;

import wt.util.WTContext;
import wt.method.RemoteMethodServer;
import wt.method.RemoteAccess;

import wt.fc.QueryResult;
import wt.fc.PersistenceHelper;
import wt.fc.PersistenceManager;
import wt.part.WTPart;
import wt.part.WTPartMaster;
import wt.query.QuerySpec;
import wt.query.SearchCondition;

public class AppLightFwd extends Applet {

    // Class name of our inner class that runs in the server
    private static final String SERVER_CLASS =
        AppLightFwd.class.getName() + "$Server";

    private Button action;
    private RunnerEventListener rel;
    private Label partNames;
    private TextField text;
    private Label queryCount;
    private TextField countVal;
    private TextArea feedback;

    public void init() {

        WTContext.init(this);
        action = new Button ("Get Parts");
        this.add(action);
        rel = new RunnerEventListener();
        action.addActionListener(rel);
        partNames = new Label( " with names like ... " );
        this.add(partNames);
        text = new TextField("", 25);
        this.add(text);
        queryCount = new Label( "Number of parts to return" );
        this.add(queryCount);
        countVal = new TextField( "5", 4);
        this.add(countVal);
        feedback = new TextArea("",10, 40);
        this.add(feedback);

    }

    public void start() {
        WTContext.getContext(this).start(this);
        super.start();
    }
}

```

```

public void stop() {
    WtContext.getContext(this).stop(this);
    super.stop();
}

public void destroy() {
    WtContext.getContext(this).destroy(this);
    super.destroy();
}

// Applet event listener
class RunnerEventListener implements ActionListener {

    public void actionPerformed (ActionEvent event) {
        Object o = event.getSource();
        if (o == action) {
            String name = text.getText();
            String count = countVal.getText();
            if (!name.equals(""))
                doSomeWork(name, count);
            else
                text.setText("must enter a part name search key" );
        }
    }
}

// Food for thought:
// here we should disable appropriate GUI components and
// spin off a separate thread to do the actual work so
// we don't hang the AWT-Thread ( the GUI thread )

public void doSomeWork(String name, String count) {
    Vector results = null;
    String like = "% [" +name.toUpperCase()+"% [" ];

    feedback.setText("");
    try
    {
        Integer cnt = null;
        try {
            cnt = new Integer(count);
        }
        catch (Exception e) {
            // some parse exception, just get default count
            try {
                cnt = new Integer(5); // this will work
            } catch (Exception e2){}
        }
    }

    // construct arguments for call
    Class [] argTypes = { String.class, Integer.TYPE };
    Object [] args = { like, cnt };

    // Run to server and do some work there.
    // Build Server inner class name as a string so we don't
    // load the class here.
    results = (Vector) RemoteMethodServer.getDefault().invoke(
        "doSomeWork", SERVER_CLASS, null, argTypes, args);
}

```

```

// display results in text area
for (int i=0;i results.size(); i++) {
    PartMasterInfo pmi
        = (PartMasterInfo)results.elementAt(i);
    feedback.append("> "+pmi.getName()+"
        # "+pmi.getNumber()+"\n");
    }
}
catch (RemoteException e)
{
    //
    // Put localized Exceptions
    // into a Dialog popup
    //
    feedback.append(e.toString());
}
catch (InvocationTargetException e)
{
    // Localize in a dialog
    feedback.append(e.toString());
}
}

// "Public" static inner class.
// Yes 2 public classes in the same file, this is the
// only exception

public static class Server implements RemoteAccess {

    public static Vector doSomeWork (String name, int count)
    {
        int i=0;
        Vector parts = new Vector(count);
        WTPartMaster wtpm;

        try {
            //
            // Use feedback mechanism to send progress updates
            // to the user
            // and of course be sure to Localize it
            //
            QuerySpec queryspec = new QuerySpec(WTPartMaster.class);
            queryspec.appendSearchCondition(
                new SearchCondition(WTPartMaster.class,
WTPartMaster.NAME,
SearchCondition.LIKE,
name) );

            QueryResult queryresult =
                PersistenceHelper.manager.find(queryspec);
            // create a vector of PartMasterInfo object to return
            // to the client
            while (queryresult.hasMoreElements()) {
                wtpm = (WTPartMaster)queryresult.nextElement();
                parts.addElement(new PartMasterInfo(wtpm.getName(),
                    wtpm.getNumber()));
                if (++i >= count)
                    break;
            }
        }
    }
}

```

```

    }
    catch (Exception e) {
        // Localize
        parts.addElement(new PartMasterInfo(e.toString(), "-1"));
        e.printStackTrace();
    }
    return parts;
}

// simple support (inner) class which contains
// the information we are interested in returning to the
// client for display purposes

public static class PartMasterInfo implements Serializable {
    String name;
    String partNumber;

    public PartMasterInfo( String name, String number ) {
        this.name = name;
        partNumber = number;
    }

    public String getName() { return name; }
    public String getNumber() { return partNumber; }
}
}

```

Updating a Master Through an Iteration

Introduction

The master-iteration design pattern (described in Appendix B, Windchill Design Patterns) specifies that the Mastered and Iterated classes work together closely. A class that implements Mastered (henceforth called the *master*) contains all the version-independent information, whereas the Iterated class (henceforth called the *iteration*) contains the incremental change that an object undergoes. When applying the master-iteration pattern, it is important to consider whether the master and iteration will be treated as a single logical object or as two distinct objects. Specifically, should a client need to know that a document is actually composed of an iteration and a master-

The Windchill reference implementations WTPart and WTDocument define the master attributes on the iteration as derived attributes, thus hiding the existence of the master from clients. For both WTPart and WTDocument, the only attributes on the master are number and name. In addition to being master attributes, they are also identifying attributes. This means that they can be modified only by using the IdentityService after the object has been persisted.

The Windchill reference implementation does not have an example of a master class with non-identifying attributes. Currently, there is no support for automatically persisting a master if its non-identifying attributes are updated using derived attributes on the iteration. That is, the attribute could be modified by

calling its setter on the iteration, but the client would have to persist the master to save the changes.

The following is a discussion of different ways master attributes can be exposed on the iteration and how changes to master attributes through the iteration can be persisted.

Read-only and Queryable Master Attributes on the Iteration

It is quite easy and often desirable to view and/or query attributes from the iteration and master together. This capability is supported at the database level by an auto-navigate feature. That is, when an iteration is retrieved from the database, the association between Iterated and Mastered is auto-navigated to retrieve the corresponding master in a single query.

To take advantage of auto-navigate, perform the following actions:

1. Add derived attributes on the iteration for each master attribute that you want to be viewed and/or queried along with the iteration. The "DerivedFrom" model property must be as follows:

master> *attribute_name*

For example, part number is modeled on WTPart with the following:

```
DerivedFrom = master>number
```

2. To make derived attributes read-only, set their WriteAccess to something other than Public.

Updating Master Attributes via the Iteration

The goal of this action is to be able to automatically update the master if any of its attributes have been modified via derived attributes on the iteration. The advantage is that clients have to deal directly with only one object, the iteration.

The recommended approach at a high level is to add a boolean "dirty" flag to the iteration class which is set to true if any of the setter methods that update master attributes are invoked. Next, implement a special save method that clients will use to save the iteration instead of using PersistenceHelper.manager.save(Persistable obj). This method is implemented in a lightweight service class so that it runs on the server in the context of a transaction. It is implemented to check if the dirty flag is true; if so, it will save the master as well as the iteration.

Updating the Master and Iteration Separately

Updating the master and iteration separately may be appropriate depending on the business rules. For example, for parts that go through a design review life cycle, modifying a master attribute would affect all versions, some of which are in a "released" state that implies they should not be modified. Changing a master attribute is significant because it violates the basic business rule that released parts should not change. For this reason, the master class could have a more restrictive

set of access rules, and any changes to it would be against the master object directly, not derived attributes on the iteration.

If this approach is taken, the client must update and persist masters and iterations separately. There may still be derived master attributes on the iteration to facilitate querying and/or read-only access of master attributes via the iteration. However, these attributes would not be set via the iteration.

The Identifying Attributes

The identifying attributes that are part of the master are not updated by calling setter methods, but rather by using `IdentityService.changeIdentity`. The client must know about the master in order to pass the master to the `IdentityService`. That is, the iteration can not be passed to the `IdentityService`. When the `IdentityService` changes the identity, it re-persists the modified master; thus, the client does not have to directly use the `PersistenceManager` to modify the master.

Master identifying attributes can only be set via the iteration before the object is persisted. When the first iteration is saved, the master is automatically created and saved as well. If you want to expose identifying attributes on the iteration, perform the following steps:

1. Add a derived attribute on the iteration for the master identifying attribute by setting the model property "Derived" to the following:

```
master> attribute_name
```

2. Set the attribute model property "Changeable" to "ViaOtherMeans" on the master. This allows the attribute to be set only until the object is persisted. After it is persisted, the `IdentityService` is the only way to change the value.
3. Add a protected `setIdentificationObject` method to the master which directly sets the identifying attributes on the master from the `IdentificationObject` values.

After the master is persisted, the identifying attributes can be changed only by the `IdentityService`, which invokes the `setIdentificationObject` method on the master via the `IdentificationObject.setToObject` method.

For additional information, see the section on Implementing Identified Objects in the *Windchill Customizer's Guide*.

Overriding Accessors for Master Attributes Derived on the Iteration

Code generation is not yet capable of determining which subtype of `Mastered`, such as `WTPartMaster`, is applicable in order to invoke the accessor methods specific to a concrete `Mastered` class. This means that the accessors must be implemented by hand. Following is an example for `WTPart`:

```
public String getName() {  
  /##begin getName% [ ]364F678B0148g.body preserve=yes  
  try { return ((WTPartMaster) getMaster()).getName(); }  
  catch (NullPointerException npe) { return null; }  
}
```

```
    /##end getName% [ ]364F678B0148g.body
  }

  public void setName( String a_Name )
  throws WTPPropertyVetoException {
  /##begin setName% [ ]364F678B0148s.body preserve=yes
    ((WTPartMaster) getMaster()).setName( a_Name );
  /##end setName% [ ]364F678B0148s.body
  }
```


9

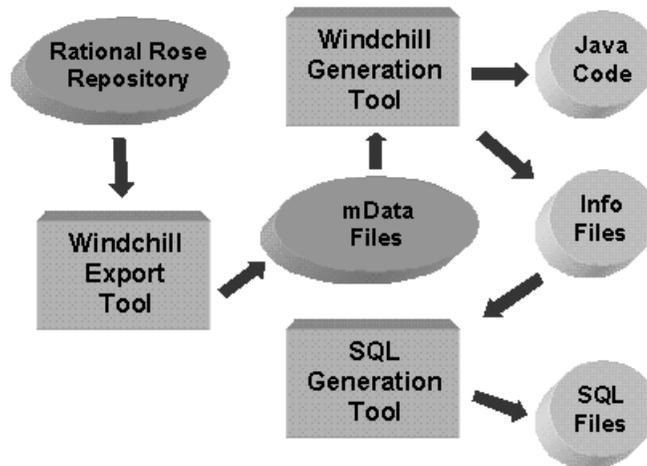
System Generation

When you have finished modeling, the next step is system generation. Using Windchill extensions to Rose's original functionality, Windchill generation tools generate Java code, Info files containing class metadata used by the runtime environment, and database DDL from the models you create. This chapter describes how to use the system generation tool and how the classes you model in UML (in Rose) correspond to the code that is generated.

Topic	Page
Overview of System Generation	9-2
How Rose UML Maps to Java Classes	9-2
Implicit Persistable Associations Stored with Foreign ID References.....	9-19
Extending the EnumeratedType class	9-33
How Rose UML Maps to Info Objects	9-35
How Rose UML Maps to Database Schema	9-37
How Rose UML Maps to Localizable Resource Info Files	9-40
Header	9-41
Resource Entry Format.....	9-41
Using the Windchill System Generation Tool	9-43
Using Windchill System Generation in a Build Environment	9-47

Overview of System Generation

The figure below shows the system generation process.



System Generation

Using the models in the Rational Rose repository, the Windchill export tool produces mData files. The system generation tools then use the mData files to produce Java code, Info files (metadata used by the runtime environment), and database schema. mData files have a non-proprietary file format so that, in the future, different modeling tools can be used without rewriting portions of the system generation tool.

The following sections describe how the Rose UML is mapped to each of the types of output (Java code, Info files, and database schema). The final section of this chapter describes how to run the system generation tool.

How Rose UML Maps to Java Classes

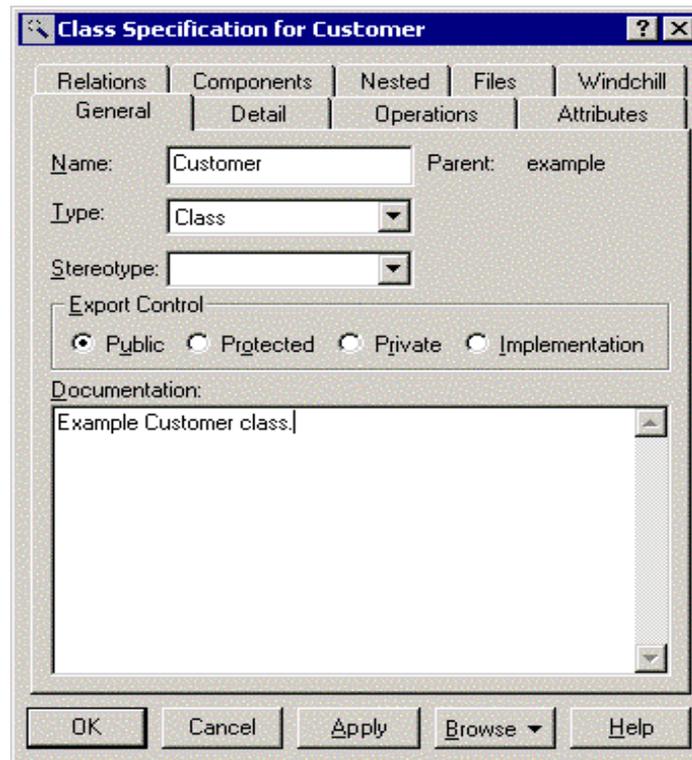
You control aspects of system generation by specifying values within Rose for the following model elements:

- Classes
- Operations
- Attributes
- Associations and their roles
- Generalize relationships
- Dependency relationships

The following sections, which describe how each of these elements are mapped to Java, also include the specifications you must set for each kind of element to

ensure the correct code is generated. Within the Rose dialogs where you set these values, there are other tabs and fields. Any tabs or fields not described in this manual are either ignored by the system generation tools or have preferred default values.

The figure below shows a sample specification dialog in Rose. To access this dialog, double click on an item in a diagram; or select a diagram item, then select Open Specification from the right click pop-up menu. Rose displays a specification that corresponds to the item you selected. The documentation for code generated items is placed in the generated source code in the format of Javadoc style comments.

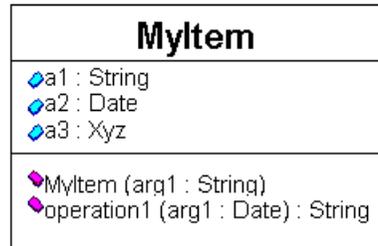


Rose Specification Dialog

Mapping Modeled Classes to Java

Classes modeled in Rose will be generated as Java classes or interfaces. An interface will be generated when the class is modeled with the Interface stereotype.

The figure below shows a class modeled in Rose that is mapped to Java.



Model of a Class

Copyright Statement

Rose provides for a default copyright, and also for a copyright to be specified for a particular package. This copyright statement is generated into the Java source code for each class.

Package Statement

The package statement corresponds to the Rose UML package that owns the class. For example,

```
// Example of a generated package statement
package example;
```

Import Statements

Java import statements are generated based on references made to other classes through the following model elements:

- Class and Interfaces *inherited* by the class.
- *Dependencies* modeled for the class.
- Non-Persistable *Association types*.
- *Attribute* types declared by the class.
- *Argument type* s specified by the methods of the class.
- *Return value types* specified by the methods of the class.
- *Exceptions* thrown by the methods of the class.

For example,

```
// Examples of a generated import statements
import example.MyAddress;
import example.MySize;
import java.lang.String;
import java.sql.Date;
import java.util.Vector;
import wt.fc.Item;
```

```

import wt.pds.PersistentRetrieveIfc;
import wt.pds.PersistentStoreIfc;
import wt.pom.DatastoreException;
import wt.util.WTException;
import wt.util.WTPropertyVetoException;

///  
begin user.imports preserve=yes
///  
end user.imports

```

The generated import statements depend on input from the classRegistry.properties file (discussed later in this chapter) that contains a list of classes and their associated packages.

The user.imports section is provided as a safe place for the developer to enter additional imports to support the implementation of the class. This section is preserved during subsequent generations of the class.

Class Documentation

Any documentation that is entered in Rose will be generated into the class in the form of Javadoc style comments. For example,

```

///  
begin MyItem% [ ]34F19D1A00B3.doc preserve=no
/**
 * An example class to demonstrate system generation
 *
 * @version 1.0
 **/
///  
end MyItem% [ ]34F19D1A00B3.doc

```

Class Declaration

A class is declared as modeled and will extend and implement all of the classes that were modeled as having a generalization relationship. If a class is modeled to extend a class that was not modeled as Extendable, via the Windchill SupportedAPI property, the generator will not allow it to be generated until that modeled generalization is removed. The class can be modeled as concrete, abstract, or as an interface. For example,

```

// Example of a class declaration
public class MyItem extends Item implements Externalizable{

// Example of an abstract class declaration
public abstract class Item extends WObject {

// Example of an interface declaration
public interface Persistable extends ObjectMappable {

```

Class Body

Some constants are generated into the body of each class. The following are examples of generated class constants:

```

// Constants used by the class
private static final String RESOURCE =
    "example.exampleResource";

```

```
private static final String CLASSNAME =  
    MyItem.class.getName();
```

The **RESOURCE** constant identifies the resource bundle the class is to use for localizable messages.

The **CLASSNAME** constant provides an easily accessible, programmatic reference to the name of the class.

The rest of the class body contains the generated results of elements modeled as features of the class and as relationships between classes. These include operations, attributes, associations, and generalizations. The details of these will be covered in subsequent sections.

Rose Class Specification

On the **General** tab, set the following values:

- **Name** specifies the name of the corresponding Java class.
- **Stereotype**:
 - For a class, leave blank.
 - For a final class, select **Final**.
 - For an abstract class, select **Abstract**.
 - For an interface, select **Interface**.
 - For a remote interface, select **RemoteInterface** (see explanation which follows).
- **Export Control** should be set to **Public** or **Implementation**. Implementation classes are visible only to other classes within the same package.
- **Documentation** specifies a description for the element. The documentation will be generated into the Java code, as Javadoc style comments.
- **Type** is ignored by the code generator.

On the **Windchill** tab, set the following values:

Note: Some specification dialogs provide multiple property sets. Every specification provides a "*default*" set. The class specification also provides "*EnumeratedType*" and "*System*" sets. The *EnumeratedType* set contains the properties that apply to an *EnumeratedType* class. The *System* set contains the properties that apply to non business domain class. Property sets may provide different default values for the same property.

- **SupportedAPI** should be set to communicate to users the degree to which the class will be supported.
 - **<Default>** to ignore the Supported API concept.

- **Private** if the class will not be supported at all.
- **Public** if use of the class is supported.
- **Extendable** if extension of the class is supported.
- **Deprecated** if the class is obsolete and support is being phased out. (This setting is superceded by the Deprecated property.)
- **Deprecated** should be set when the element is obsolete.
 - **<Default>** indicates the class is not deprecated, unless, for backward compatibility, SupportedAPI is set to Deprecated.
 - **deprecated** if the class is obsolete and support is being phased out.
- **Java Properties**
 - **Generate** should be set to **False** if the system generation tools should ignore this modeled class.
 - **CodeGenerationName** specifies the name of the class that will be generated. Leave it blank to have the generated name be the same as the name of the modeled class.
 - **Serializable** indicates if the generated class will implement the Serializable or Externalizable interface. **Default** evaluates to **Externalizable**, if possible; otherwise, **Serializable**. **Externalizable (basic)**, can be used to generate simple Externalization support that does not include support for reading old versions. For a class that will have instances serialized into BLOB columns in the database, set the property to **Evolvable**. (For further information, see appendix D, Evolvable Classes.) To have the class implement neither interface, set the property to **None**.
 - **PrimitiveType** indicates which primitive value-type a class can be decomposed into.
 - **StaticInitBeforeFields** indicates whether the static initializer will be placed before the field declarations.
 - **GenAttributeLabels** indicates whether label constants will be generated for the modeled attributes and roles of the class. If **True**, they will be generated. If **Default**, they will be generated for classes that implement ObjectMappable and for interfaces.
 - **ExplicitOrder** (EnumeratedType set only) indicates whether the modeled EnumeratedType options will be explicitly order in the resource info (rbInfo) file. By default, a locale specific, alphabetical order will be determined at run-time.
- **Oracle Properties**

- **TableName** is the database table name to use for the class.
- **TableSpaceName** is the tablespace to use for storage option of the table.
- **TableSize** indicates if the relative size required for the objects that will be stored (the actual storage values are mapped from the size via property file settings).
- **IndexTableSpaceName** is the tablespace to use for the storage option of the indices of the table.
- **CompositeIndexN** indicates the columns for composite index on the table.
- **CompositeUniqueN** indicates the columns for composite unique index on the table.
- **UI Properties**
 - **StandardIcon** is the file name of the standard icon for the class.
 - **OpenIcon** is the file name of the open icon for the class.
 - **Localizable** indicates if the class will have a localizable display name stored in the resource bundle for the package.
- The rest of the dialog is ignored by the code generator.

Mapping Operations to Java

The figure below shows an operation modeled in Rose that is mapped to a method in Java.



Model of Operations

The following code is generated for this example.

```

public class MyItem extends Item implements Serializable {
    /**
     * The example operation.
     *
     * @param arg1 The Date argument
     * @return StringThe String return value
     */
    /**end operation1% [ ]34F19DCB01D0.doc

    public String operation1( Date arg1 ) {
        /**begin operation1% [ ]34F19DCB01D0.body preserve=yes
  
```

```

        return null;
        //##end operation1% [ ]34F19DCB01D0.body
    }
}

```

Operations modeled in Rose are mapped as Java methods of the generated class. For operations that are not modeled as abstract, stubs are created in the generated class where you add your own code. All documentation, keywords, parameters, and return types that are modeled in Rose are generated into Java classes. Information about operations that you can specify in Rose is detailed below.

The begin and end markers that are generated into editable files denote the sections that you can edit. These sections are preserved as is during subsequent generations of the files.

Some sections are marked with `preserve=no`. This is true for all Javadoc comment sections and some code sections, where some implementation is generated into an editable file. The "no" value for `preserve` indicates that the section is a generated default, which you may choose to edit. If you do edit any of these sections, you must change the `preserve` value from "no" to "yes"; otherwise, it will not be preserved.

If `MyItem` were an interface, only the operation declaration would be generated into `MyItem`, because a Java interface can contain no implementation. To the degree possible, the implementation aspects of interface operations will be generated into concrete subclasses. See the [Implementing Interfaces](#) section for details.

Rose Operation Specification

On the **General** tab, set the following values:

- **Name** specifies the name of the resulting Java method.
- **Return class** specifies the type of the method's return value.
- Export control should be set to **Public**, **Protected**, or **Implementation**. Public methods are external API to the class. Protected methods are customization points that extending classes can choose to override. Private methods need not be part of the business model. Developers can implement them in their classes as needed.
- **Documentation** specifies a description for the element. The documentation will be generated into the Java code, as Javadoc style comments.

On the **Detail** tab, set the following values:

- **Argument Name** specifies the name of an argument in the method.
- **Argument Type** specifies the argument type.
- **Exceptions** specifies the exceptions thrown by the method.
- The rest of the dialog is ignored by the code generator.

On the **Windchill** tab, set the following values:

- **SupportedAPI** should be set to communicate to users the degree to which the operation will be supported.
 - **Private** if the operation will not be supported at all.
 - **Public** if use of the operation is supported.
 - **Deprecated** if the operation is obsolete and support is being phased out. (This setting is superseded by the **Deprecated** property.)
- **Deprecated** should be set when the element is obsolete.
 - **<Default>** indicates the operation is not deprecated, unless, for backward compatibility, **SupportedAPI** is set to **Deprecated**.
 - **deprecated** if the operation is obsolete and support is being phased out.
- **Java Properties**
 - **Abstract** should be set to **True** to make the resulting method abstract.
 - **Static** should be set to **True** to make the resulting method static.
 - **Final** should be set to **True** to make the resulting method final.
 - **Native** should be set to **True** to make the resulting method a native method.
 - **Synchronized** should be set to **True** to make the resulting method a synchronized method.
 - **RemoteInvocation** should be set to **True** to make the resulting method executable only at the server. This setting is not needed for classes that implement a **RemoteInterface**.

Mapping Attributes to Java

Attribute Implementation

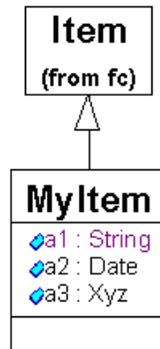
The external Java representation of a class attribute modeled in Rose consists of a *getter* operation, a *setter* operation, and an identifying label. The export visibility of the getter and setter operations is based on the export visibility of the modeled attribute in Rose.

The internal Java implementation of the class attribute is always a private field. This field is accessed strictly through getter and setter operations. Encapsulating the attribute as a private field provides several benefits:

- The class controls the integrity of the attribute by monitoring changes to the attribute.

- The class can support subscribe/notify functionality that allows listeners to monitor the state of the attribute.
- Access through getter and setter operations provides the opportunity to synchronize access to the attribute.

The following model shows the class MyItem, with attributes a1, a2, and a3, which extends the class Item.



Model of Attributes

Attribute Label Constants

String field constants are generated as attribute labels so you can use them in APIs that require an attribute name, such as queries. Thus, the Java compiler can detect typographical errors.

```
// Examples of attribute name constants
public static final String A1 = "a1";
public static final String A2 = "a2";
public static final String A3 = "a3";
```

Field Declarations

All attributes modeled in Rose are implemented as private fields of the Java class. For example,

```
// Examples of attribute declarations
private String a1;
private Date a2;
private Xyz a3;
```

The accessor methods to these attributes are public or protected, depending on the export control as defined in the model. Examples of accessor methods follow.

Accessor Methods

Public and protected accessor methods are generated in the Java class. For example,

```

// Example of a "getter" method
public String getA1() {
    return a1;
}
// Example of a "setter" method
public void setA1( String theA1 )
    throws WTPROPERTYVetoException {
    a1 = theA1;
}

```

Accessor methods are code-generated from the attributes modeled on the business class in the Rose model. They need not be modeled on classes in the UML model. These accessor methods follow the Java beans naming convention. Attributes that were modeled as public get public accessor methods. Attributes that were modeled as protected get protected accessor methods.

The system generation tool generates accessors that enforce attribute validation based on attribute properties specified in the model.

If the constrain property (a Rose specification) is set to true for an attribute modeled in Rose, the setter method is declared to throw a `wt.util.WTPROPERTYVetoException`, which is derived from `java.beans.PropertyVetoException`. This exception is thrown if the setter method has determined that a value being set in the attribute is invalid. Developers can change the default accessor methods by writing code in the preserve region.

Validation code will be generated if the attribute is modeled with a lower or upper bound, as unchangeable, or as a required attribute. Each of these properties appear, in Rose, on the Windchill tab for the attribute. Validation code will also be generated if the attribute is modeled with a constrained type. That is, the attribute is redefining an attribute, in the hierarchy, to be of a sub-type of the original definition. If a validation method is generated, the setter code will invoke it. If validation were generated for the "a1" attribute, the method would be "validateA1".

If `MyItem` were an interface, only the accessor declarations and the label constant would be generated into `MyItem`, because a Java interface can contain no implementation. To the degree possible, the implementation aspects of interface attributes will be generated into concrete subclasses. See the Implementing Interfaces section for details.

Rose Attribute Specification

On the **General** tab, set the following values:

- **Name** specifies the names of the generated attribute.
- **Type** specifies the attribute's class.
- **Initial value** specifies the initial value for the attribute.
- **Export control** should be set to **Public**, **Protected**, or **Implementation**. If you want private methods, declare them in your source code.

- **Documentation** specifies a description for the element. The documentation will be generated into the Java code, as Javadoc style comments.

On the **Detail** tab, set the following values:

- **Static** should be selected if the attribute is to be generated as a static field.
- **Derived** should be selected if the generated private field is not desired. Also see the `DerivedFrom` property below.
- The rest of the dialog is ignored by the code generator.

On the **Windchill** tab, set the following values:

In addition to the default property set, attributes have two predefined sets. First, named "constant", which can be used for constant attributes. Second, named "constantEnumeratedType", which can be used for constant attributes on EnumeratedTypes. While these property sets can be used to quickly set the Windchill properties for these constant attributes, the `Static` property still needs to be selected on the detail tab.

- **Abstract** should be set to **True** if the implementation of a field for the attribute will be deferred to a subclass. Accessor methods generated for this attribute will be abstract.
- **DerivedFrom** should be specified if this attribute is derived from another modeled element. The value will specify a traversal path to the source. For example, `homeAddress.city` indicates that this class has a non-first-class aggregation named `homeAddress` which contains an attribute named `city`. To specify that an attribute's derivation traversal goes through an `ObjectReference`, a different delimiter is used. For example, `homeAddress>city` indicates that this class has an opposite-side role on a first-class association named `homeAddress` which contains an attribute named `city`.
- **StringCase** should be set to **LowerCase** to force the value to lowercase. It should be set to **UpperCase** to force the value to uppercase. The enforcement of this constraint will be generated into the setter method for the attribute.
- **LowerLimit** constrains the valid values for the type. For String types, it specifies the minimum length of the String. For numeric types, it specifies the minimum value of the attribute. Date and Time types are not currently supported by this property. The constraint is enforced in the validation that is generated for the setter.
- **UpperLimit** constrains the valid values for the type. For String types, it specifies the maximum length of the String. For numeric types, it specifies the maximum value of the attribute. Date and Time types are not currently supported by this property. The constraint is enforced in the validation that is generated for the setter.

- **Required** should be set to **True** if the database will not allow a null value in the column that persists the attribute's value, and the setter validation will not allow the attribute's value to be set to null.
- **Changeable** should be set to **Frozen** if the value cannot be changed once it has been persisted. It should be set to **ViaOtherMeans** if the normal setter method is not allowed to change the value once it has been persisted. (For example, **ViaOtherMeans** is used for those attributes that are part of the identity of the class which must be changed through the identity service.)
- **WriteAccess** should be set if the access of the setter should be different than that of the getter that will be generated.
- **SupportedAPI** should be set to communicate to users the degree to which the attribute will be supported.
 - **<Default>** to ignore the Supported API concept.
 - **Private** if the attribute will not be supported at all.
 - **Public** if use of the attribute is supported.
 - **Deprecated** if the attribute is obsolete and support is being phased out. (This setting is superseded by the **Deprecated** property.)
 - **Deprecated** should be set when the element is obsolete.
 - **<Default>** indicates the attribute is not deprecated, unless, for backward compatibility, **SupportedAPI** is set to **Deprecated**.
 - **deprecated** if the attribute is obsolete and support is being phased out.
- **Java Properties**
 - **Final** should be set to **True** if the resulting field should be final.
 - **Transient** should be set to **True** if the resulting field should be transient.
 - **Volatile** should be set to **True** if the resulting field should be volatile.
 - **Persistent** should be set to **True** if the field that holds the value will be persisted to the database.
 - **GenerateAccessors** should be set to **False** if no accessors are to be generated.
 - **Constrain** should be set to **True** if the resulting setter method should declare that it throws a **WTPPropertyVetoException**.
 - **GetExceptions** specifies exceptions that will be declared as thrown by the generated getter.
 - **SetExceptions** specifies exceptions that will be declared as thrown by the generated setter.

- **BeforeStaticInitializer** should be set to **True** if field declaration should be placed prior to the static initializer.
- **Oracle Properties**
 - **ColumnType** indicates the database column mapping.
 - **Index** indicates if an index should be created for this attribute.
 - **Unique** indicates if only unique values are allowed for this attribute (enforced at the database level).
 - **Updateable** indicates if the attributes column will be updated after the initial insert.
 - **ColumnName** specifies the table column name for this attribute.

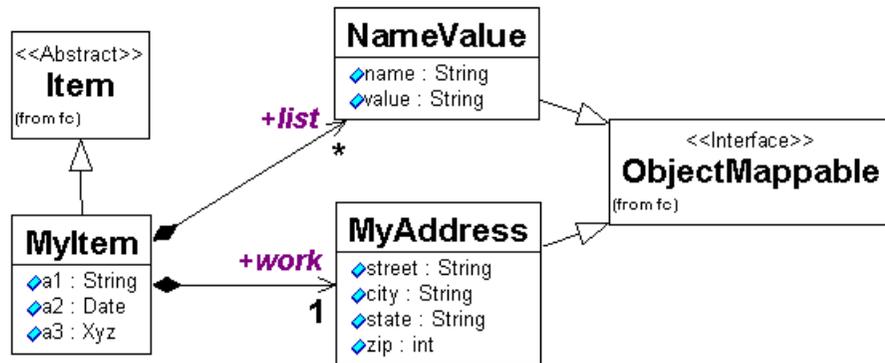
- **UI Properties**
 - **Localizable** indicates if the attribute will have a localizable display name stored in the resource bundle for the package.
 - **Display** (constantEnumeratedType set only) specifies the localizable display name for the attribute, if the attribute is a constant for an EnumeratedType.
 - **DefaultValue** (constantEnumeratedType set only) should be set to True if the attribute is the default option for the EnumeratedType. Only one attribute can be set to True.
 - The rest of the dialog is ignored by the code generator.

A constant attribute is one that is specified as static and final.

Mapping Associations to Java

Non-Persistable Associations

The following model shows associations with non-persistable objects (MyAddress and NameValue).



Associations between Non-Persistable Objects

The following code is generated from this example:

```

public class MyItem extends Item {
    private String a1;
    private Date a2;
    private Xyz a3;
    private MyAddress work;
    private Vector list; // contains a set of NameValue objects
    // This class also contains the constants and the accessor
    // methods for a1, a2, a3, work and list.
}
  
```

The implementation of a non-persistable association is the same as the implementation of a modeled attribute. That is, it will be implemented with a private field, a constant label, and accessor methods.

If classes modeled in Rose extend the class `Item`, these classes are automatically both `ObjectMappable` and `Persistable`. But in this example, an object called `MyAddress` is modeled that is a composite aggregation. This means that the instance `MyAddress` never exists on its own. It exists only as a part of another instance. In this case, the code generator creates a field in `MyItem` using the name of the composite aggregation, that is, the name of the navigable role on the association (in this example, `work`). Note that although modeling it as a composite aggregation communicates the ownership, the role must be navigable to accomplish the generated implementation.

If the association's cardinality is greater than one, the attribute is represented as a Java `Vector`. For example, the class `NameValue` is also a composite aggregation and it contains multiple name and value strings. Because the total number that could occur is not known during system generation, the code generator creates a `Vector` field with the name of the composite aggregation (in this example, `list`). In this example, the field is a list of names and values.

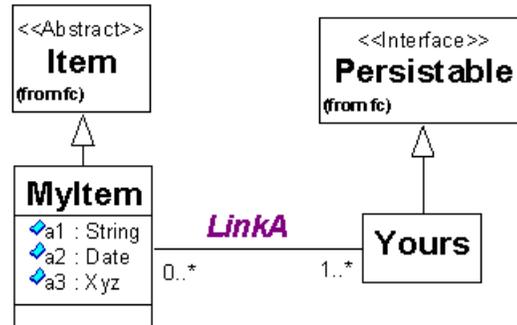
The generated class also contains public accessor methods for all the attributes.

Note also that if the role's cardinality is optional, but the class being aggregated contains required elements, the aggregated class will be treated as required. The system generator must enforce this coordination of required attributes because of the way the PDS (Persistent Data Service) enforces required attributes. The PDS accomplishes the enforcement through the `NOT NULL` column feature of SQL databases. Because a column within the structure will be generated as `NOT NULL`, the structure as a whole cannot be nullable.

If `MyItem` were an interface, only the accessor declarations, and the label constant would be generated into `MyItem`, because a Java interface can contain no implementation. To the degree possible, the implementation aspects of interface associations will be generated into concrete subclasses. See the `Implementing Interfaces` section for details.

Implicit Persistable Associations

The following model shows an association between persistable objects where the association has no attributes.



An Association with No Attributes between Persistable Objects

The following code is generated from this example:

```
public final class LinkA extends ObjectToObjectLink
    implements Externalizable {

    // role name constants
    public static final String MY_ITEM_ROLE = "theMyItem";
    public static final String YOURS_ROLE = "theYours";
    public MyItem getMyItem() { // getter

        return (MyItem)getRoleAObject();
    }

    public void setMyItem( MyItem theMyItem ) // setter
        throws WTPROPERTYVETOException {

        setRoleAObject( theMyItem );
    }
    // two-arg factory for link classes
    public static LinkA newLinkA( MyItem theMyItem,
        Yours theYours )
        throws WTEException {

        LinkA instance = new LinkA();
        instance.initialize( theMyItem, theYours );
        return instance;
    }
}
```

In this case, the code generator creates persistable Java classes that extend the ObjectToObjectLink class and are capable of maintaining a persistent association.

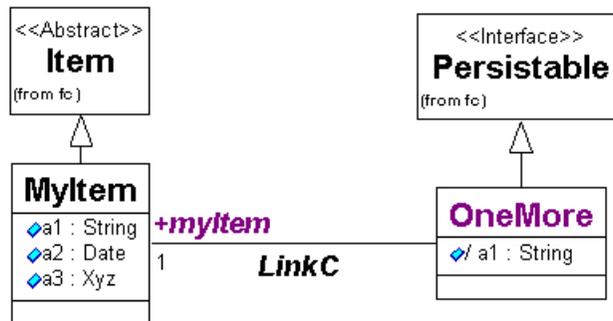
Code-generated accessor methods return the role A and role B objects of the link. This means that the developer need not be concerned with which object is the role A object and which is the role B object.

Code-generated factory methods for Link classes take at least two arguments: the role A object and the role B object. The factory methods are a result of the Link

interface extending the NetFactor interface. A full explanation of factory methods is included in the section on Implementing the NetFactor Interface.

Implicit Persistable Associations Stored with Foreign ID References

The following model shows an association between persistable objects where the association has no attributes.



A Persistable Association Stored with a Foreign ID Reference

The link class is generated for LinkC, just as it was for LinkA, in the preceding example. In addition, the following code is generated in the class that plays the role opposite the role that has a cardinality of one.

```

public class OneMore implements Persistable, Externalizable {

    public static final String A1 = "myItem>a1";
    public static final String MY_ITEM_REFERENCE =
        "myItemReference";

    private ObjectReference myItemReference;

    public String getA1() {
        try { return getMyItem().getA1(); }
        catch (NullPointerException npe) { return null; }
    }

    public void setA1( String a_A1 )
        throws WTPROPERTYVETOEXCEPTION {
        getMyItem().setA1( a_A1 );
    }

    public MyItem getMyItem() {
        if ( myItemReference == null )
            return null;
        return (MyItem)myItemReference.getObject();
    }

    public ObjectReference getMyItemReference() {
        return myItemReference;
    }
}
  
```

```

public void setMyItemReference( ObjectReference
    a_MyItemReference )
    throws WTPPropertyVetoException {
    myItemReferenceValidate( a_MyItemReference );
    // throws exception if not valid
    myItemReference = a_MyItemReference;
}
}

```

In this case, since the association is persisted via a foreign id in the table for OneMore rather than in a separate link table, the OneMore class will hold a reference to myItem and will get myItemReference accessors generated. In addition to the getter and setter that are generated for the reference, a convenience getter is generated for the myItem object.

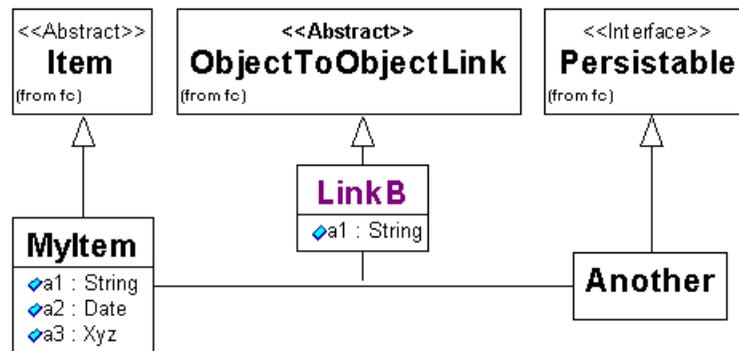
Although these additional accessors are created for developer convenience, the LinkC class that is generated for the association can be operated on in the same manner as a link class that is stored in a separate link table. This provides a common API for manipulating links, regardless of how the database storage is implemented.

The example also had a derived attribute modeled for the OneMore class. The DerivedFrom property for the attribute was defined as myItem>a1, which caused the A1 label constant and accessors to be generated for the a1derived attribute. (If this were a non-persistable association, the syntax for the derived attribute source would be myItem.a1.)

Care should be taken in using this feature with persistable associations since allowing the generation and use of a derived setter will cause a state change in a different persistable object (myItem) which may not be obvious to the developer who is using your class (OneMore). The generation of the setter can be turned off while retaining the generation of the getter by setting the attribute's **WriteAccess** property to **Private**.

Explicit Persistable Associations

The following model shows an association between persistable objects where the association has attributes.



An Association with Attributes between Persistable Objects

The following code is generated from this example:

```
public class LinkB extends ObjectToObjectLink implements
    Externalizable {
    . . .
    /* everything that the implicit link has, plus normal
       class generation for attributes, operations, etc. */
}
```

If an association has attributes associated with it, the code generator creates a Java class that has all of the attributes for the association and the accessor. The name of the generated Java class is the name of the attributing class.

The generated Java class extends the `ObjectToObjectLink` class if the attributing class does not extend another class in the model. If the attributing class extends another class in the model, the Java class extends that class. (The class being extended must be a subclass of `Link`.)

Rose Association Specification

For the tabs that specify role information, there is a **Role A** tab and a **Role B** tab. Both work essentially the same with the only differences being whether role A or role B is referred to.

On the **General** tab, set the following values:

- **Name** specifies the name association that is the name of the resulting `Link` class. If blank, it defaults to a role A/role B concatenation.
- **Role A** specifies the role A name. If blank, it defaults to the name of the role A class (in this case, the class name is prefixed with "the" to avoid a naming conflict). The popup menu for associations provides an alternative means for specifying role A.
- **Role B** specifies the role B name. If blank, it defaults to the name of the role B class (in this case, the class name is prefixed with "the" to avoid a naming conflict). The popup menu for associations provides an alternative means for specifying role B.
- The rest of the dialog is ignored by the code generator.

On the **Detail** tab, set the following value:

- **Derived** should be set to **True** to indicate that the implementation of the association will be derived, as determined by the developer.
- The rest of the dialog is ignored by the code generator.

On the **Windchill** tab, set the following values:

- **SupportedAPI** should be set to communicate to users the degree to which the association class will be supported.
 - **Private** if the association class will not be supported at all.

- **Public** if use of the class is supported.
- **Extendable** if extension of the class is supported.
- **Deprecated** if the class is obsolete and support is being phased out. (This setting is superceded by the Deprecated property.)
- **Deprecated** should be set when the element is obsolete.
 - **<Default>** indicates the class is not deprecated, unless, for backward compatibility, SupportedAPI is set to Deprecated.
 - **-deprecated** if the class is obsolete and support is being phased out.
- **Java Properties**
 - **Generate** should be set to **False** if the system generation tools should not generate a Java class for this modeled association.
- **Oracle Properties**
 - **Storage** indicates if the association will be stored in a link table or as a foreign id reference in the table for one of the role classes. **Default** evaluates to **ForeignKey** if there is a role with single cardinality and the class opposite that role resides in the same package as the association; otherwise, evaluates to **LinkTable**.
- The rest of the dialog is ignored by the code generator.

On the **Role <A/B> General** tab, set the following values:

- **Role** specifies the name of the role. (See **Role A** (or **B**) description for **General** tab above.)
- **Export control** should be set to **Public**, **Protected**, or **Implementation**. If you want private attributes, declare them in your source code.
- **Documentation** specifies a description for the element. The documentation will be generated into the Java code, as Javadoc style comments.

On the **Role <A/B> Detail** tab, set the following values:

- **Role** specifies the name of the role. (See **Role A** (or **B**) description for **General** tab above.)
- **Cardinality** specifies the cardinality of the role. The popup menu for associations provides an alternative means for specifying cardinality.
- **Navigable** should be selected if the role B (or role A) object should get a method for accessing the role A (or role B) object. The popup menu for associations provides an alternative means for specifying navigability. Navigable is ignored for Persistable associations.

- **Aggregate** should be selected if the association is an aggregate. The popup menu for associations provides an alternative means for specifying aggregation.
- **By value** should be selected if the aggregate is a composite aggregation. The popup menu for associations provides an alternative means for specifying composite aggregation.
- The rest of the dialog is ignored by the code generator.

On the **Windchill<A/B>** tab, set the following values (ignored for Persistable associations):

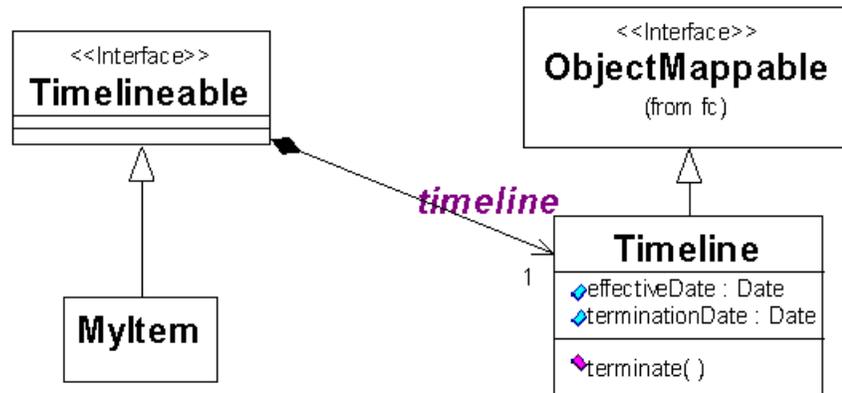
- **Abstract** should be set to **True** if the implementation of a field for the role will be deferred to a subclass. Access methods generated for this role will be abstract.
- **StringCase** should be set to **LowerCase** to force the value to lowercase. It should be set to **UpperCase** to force the value to uppercase. The enforcement of this constraint will be generated into the setter method for the attribute.
- **LowerLimit** constrains the valid values for the type. For String types, it specifies the minimum length of the String. For numeric types, it specifies the minimum value of the role. Date and Time types are not currently supported by this property. The constraint is enforced in the validation that is generated for the setter.
- **UpperLimit** constrains the valid values for the type. For String types, it specifies the maximum length of the String. For numeric types, it specifies the maximum value of the role. Date and Time types are not currently supported by this property. The constraint is enforced in the validation that is generated for the setter.
- **Changeable** should be set to **Frozen** if the value cannot be changed once it has been persisted. It should be set to **ViaOtherMeans** if the normal setter method is not allowed to change the value once it has been persisted.
- **WriteAccess** should be set if the access of the setter should be different than that of the getter that will be generated.
- **Dependency** should be set to **True** if the opposite-side role class is dependent on this role.
- **SupportedAPI** should be set to communicate to users the degree to which the role will be supported.
 - **Private** if the role will not be supported at all.
 - **Public** if use of the role is supported.
 - **Deprecated** if the role is obsolete and support is being phased out. (This setting is superceded by the **Deprecated** property.)
- **Deprecated** should be set when the element is obsolete.

- <**Default**> indicates the role is not deprecated, unless, for backward compatibility, SupportedAPI is set to Deprecated.
- **deprecated** if the role is obsolete and support is being phased out.
- **Java Properties**
 - **Final** should be set to **True** if the resulting field should be final.
 - **Transient** should be set to **True** if the resulting field should be transient.
 - **Volatile** should be set to **True** if the resulting field should be volatile.
 - **Constrain** should be set to **True** if the resulting setter method should declare that it throws a WTPPropertyVetoException.
 - **ReferenceType** specifies the class of ObjectReference that will be used for first-class associations that are stored as with a ForeignKey and thereby make use of a held ObjectReference. By default, the generic ObjectReference will be used.
 - **AutoNavigate** should be set to **True** if the object playing this role should be automatically retrieved (instantiated) from the database whenever the object on the other side is retrieved. Update operations are not impacted by this property. This feature is dependent on the association being implemented as a ForeignKeyLink.
 - **Persistent** should be set to **True** if the field that holds the value will be persisted to the database.
 - **GetExceptions** specifies exceptions that will be declared as thrown by the generated getter.
 - **SetExceptions** specifies exceptions that will be declared as thrown by the generated setter.
 - **ContainerClass** specifies that class of container to use for roles of unbounded multiplicity (cardinality). Default container is Vector.
 - **InitialValue** specifies the value to which the field will be initialized upon declaration.
 - **DelegatedInterface** specifies that the object playing the role is a delegate that supplies an implementation for the specified interface. All the methods necessary to fulfill (implement) the interface will be generated as methods that forward to the role object.
 - **BeforeStaticInitializer** should be set to **True** if field declaration should be placed prior to the static initializer.
- **UI Properties**
 - **Localizable** indicates if the role will have a localizable display name stored in the resource bundle for the package.

- The rest of the dialog is ignored by the code generator.

Implementing Interfaces

In Rose, an interface is modeled as a class that has the stereotype <<Interface>>.



Model of Implementing an Interface

In this example, MyItem, which is concrete, is said to implement the interface named Timelineable, which aggregates Timeline. The following code is generated for this example.

```
public class MyItem extends Item implements Timelineable,
    Externalizable {

    private Timeline timeline;

    public Timeline getTimeline() {
        return timeline;
    }

    public void setTimeline( Timeline a_Timeline )
        throws WTPROPERTYVETOException {
        timelineValidate( a_Timeline );
        // may throw exception if not valid
        timeline = a_Timeline;
    }

    private void timelineValidate( Timeline a_Timeline )
        throws WTPROPERTYVETOException {
        if ( a_Timeline == null ) {
            // required attribute check
            Object[] args = { new wt.introspection.PropertyDisplayName(
                CLASSNAME, "timeline" ) };
            throw new WTPROPERTYVETOException( "wt.fc.fcResource",
                wt.fc.fcResource.REQUIRED_ATTRIBUTE, args,
                new java.beans.PropertyChangeEvent( this, "timeline",
                    timeline, a_Timeline ) );
        }
    }
}
// The constant label is generated into the
// Timelineable interface
}
```

In general, system generation causes the following actions:

- Classes in the model that extend an interface cause an implements clause to be created on the class declaration of the generated class.
- Stubs for the operations of the interface are created in the generated class if the inheriting class is concrete. This is where you insert your implementation code. (This applies even for protected methods modeled on an interface, which will not appear on the generated interface since an interface can have only public features.)
- Non-constant attributes modeled on the interface cause private fields to be created on the generated classes that implement the interface, along with generated accessor methods.
- Associations with non-persistable classes are handled similar to modeled attributes, where the API is generated on the interface and the implementation is generated into the concrete subclasses.

Implementing the NetFactor Interface

The vast majority of business classes provided with Windchill applications and foundation components implement the NetFactor interface, either directly or indirectly. Implementing this interface, in fact, indicates that a class is a business class participating in the Windchill architecture.

Factory operations

To give flexibility to the system architecture, Windchill uses a factory design pattern for the construction of Java objects. A constructor signature is one where the operation name matches the name of the class and has no return type. Object constructors are modeled in Rose but are not generated directly in Java code. Instead of constructors, Windchill generates factory operations that are used to construct objects.

Using factory operations instead of constructors provides the opportunity to vary the class of the returned object. When constructors are used to create an object, the class of the returned object must match exactly the class requested. However, when factories are used, the class of the returned object will be polymorphically compatible with the requested class but need not match the requested class exactly. This allows the return of objects whose class may vary depending on the context.

When a constructor is specified in a Rose model, two operations are generated in the Java code: a public factory operation and a protected initialization operation. The factory operation is called directly by the application to create a Java instance. The factory method calls the initialize operation to put the instance into an initial state.

Note that for optimization reasons, an initialize method is generated only when one having the same signature is not provided by a superclass. You can manually supply an override initialize method if you wish.

If the modeled class (directly or indirectly) inherits from NetFactor, this is a cue to the code generator to generate factory and initialize methods in the Java class. The following modeled class has a constructor that takes an argument of a string.



Factory Operation for an Item

The following code is generated for this example (documentation omitted).

```
public class MyItem extends Item {
    public static MyItem newMyItem( String arg1 )
        throws WTEException {
        /##begin newMyItem% [ ]34F1E6BF02D9f.body preserve=no

        MyItem instance = new MyItem();
        instance.initialize( arg1 );
        return instance;
        /##end newMyItem% [ ]34F1E6BF02D9f.body
    }

    protected void initialize( String arg1 )
        throws WTEException {
        /##begin initialize% [ ]34F1E6BF02D9.body preserve=yes
        /##end initialize% [ ]34F1E6BF02D9.body
    }
}
```

A factory method instantiates and initializes business objects. It invokes the constructor of the implementation class and then invokes an initialize method with the same parameters as the factory method.

The code generator also generates the correct method stubs so you can implement the initialize method. This method is responsible for setting the initial state of a new object.

Factory methods have the form:

```
public static <class> new <class> (args)

throws WTEException
```

The factory method has the same name as the class in the Rose model plus the prefix "new" (for example, newMyItem).

If no constructor is modeled for a class, the generator assumes a default, no-arg constructor. For link classes, the default factory method includes a reference to the two objects being related by the link. For details on link class generation, see the Persistable Associations section.

Conceptual class name accessor

Every class that inherits from NetFactor implements a `getConceptualClassname` instance method:

```
public String getConceptualClassname() {  
    return CLASSNAME;  
}
```

This method returns the fully-qualified conceptual class name of the object. Because the instantiated object may really be an instance of some other implementation class, `getConceptualClassname` is a useful method to find the business class in an object.

You should not use `object.getClass().getName()` to determine the class name for software objects in the Windchill system because it may not return the name of the conceptual class.

Info object accessor

Every class that inherits from NetFactor has an info object generated for it. The code generator therefore ensures that each NetFactor instance supports the `getClassInfo` method. This method is generated into the top class in each hierarchy. For example:

```
public ClassInfo getClassInfo()  
    throws WTIntrospectionException {  
    return WTIntrospector.  
        getClassInfo( getConceptualClassname() );  
}
```

This method returns the `ClassInfo` instance that contains the metadata from the installed model.

Implementing the ObjectMappable interface

The database methods `readExternal` and `writeExternal` are code-generated for any class that was modeled to implement the `ObjectMappable` interface. The `readExternal` method reads the object's state from the database. The `writeExternal` method writes the object's state to the database.

The class defers to its superclasses to read (or write) the state for the persistent attributes that those classes declared. The class defers to its structured attribute classes, such as `Address`, to read (or write) the state for the persistent attributes that those classes declared with the `readObject` and `writeObject` methods.

Because the class defers the reading of some of the fields from the database to its superclass, the types of structured attributes are hard-coded into the super class.

The exception to this rule of hard-coded types is for ObjectReferences that are generated for roles of Persistable associations. The Persistent Data Service (PDS) does a runtime look-up in the introspection information to see if the ReferenceType for the role was redefined at a lower level in the class hierarchy. If so, the PDS uses that type. But even if the type is redefined, its persistent structure must be compatible with the type that was used by the super class, because the column definitions defined by a super class cannot be changed by a subclass.

The PersistentRetrieveIfc argument for readExternal contains the state for the object being read. The PersistentStoreIfc argument for writeExternal receives the state of the object being written.

Examples of the readExternal and writeExternal methods follow:

```
// Example of a database writeExternal method
public void writeExternal( PersistentStoreIfc output )
    throws SQLException, DatastoreException {
    super.writeExternal( output );
    output.setString( "a1", a1 );
    output.setDate( "a2", a2 );
    output.setObject( "a3", a3 );
    output.writeObject( "work", work,
        wt.tools.generation.example.MyAddress.class, true );
    output.setObject( "list", list );
    output.setString( "size", size == null - null :
        size.toString() );
    output.writeObject( "timeline", timeline,
        wt.tools.generation.example.Timeline.class, true );
}

// Example of a database readExternal method
public void readExternal( PersistentRetrieveIfc input )
    throws SQLException, DatastoreException {
    super.readExternal( input );

    a1 = input.getString( "a1" );
    a2 = input.getDate( "a2" );
    a3 = (Xyz)input.getObject( "a3" );
    work = (wt.tools.generation.example.MyAddress)input.readObject(
        "work", work,
        wt.tools.generation.example.MyAddress.class, true );
    list = (Vector)input.getObject( "list" );
    size = MySize.toMySize( input.getString( "size" ) );
    timeline = (wt.tools.generation.example.Timeline)
        input.readObject( "timeline", timeline,
            wt.tools.generation.example.Timeline.class, true );
}
```

Implementing the Externalizable interface

A core feature of Java is the capability of streaming objects. This feature is used for purposes such as saving objects into jar files or passing Java objects by value on RMI calls. There are two ways to support externalization: by using the tagging interface Serializable or by using the interface Externalizable. When possible, Windchill classes implement the more efficient Externalizable interface.

Sometimes it is not convenient to implement `Externalizable` and the less efficient `Serializable` is used instead.

A class is generated to implement `Externalizable`, by default, if it meets the criteria for Externalization. The criteria are as follows:

- Must have a public, no-argument constructor.
- Any parent class must be `Externalizable`.

A property in Rose allows you to override the default determination. This property, named *Serializable*, is in the Java Properties on the Windchill tab for the class. With this property, you can force the class to `Serializable`, `Externalizable`, or neither.

The `Evolvable` option has been added to this property to better manage which classes can be serialized into BLOB columns in the database. This option was added in conjunction with a change to the generated Externalization stream format. Because of this change to the stream format, much externalization code is generated to maintain backward compatibility. In a future release, only the `readExternal` and `writeExternal` methods will be generated for classes that implement `Externalizable` but not `Evolvable`. Until the backward compatibility is removed, new classes can be forced to generate simplified externalization code by selecting the "Externalizable (basic)" option. The evolvable aspects of externalization will not be shown below. For details on when to use `Evolvable` and how to manage `Evolvable` classes, see appendix D, `Evolvable Classes`.

The `serialVersionUID` constant

The `serialVersionUID` is used to determine compatibility of classes when reading them from an input stream. The mechanisms that perform the serialization and externalization compute a `serialVersionUID` value at runtime when a class has not explicitly defined the value to something other than zero.

When the code generator is generating externalization for a class, it sets the `serialVersionUID` to 1 since compatibility is managed in the externalization methods (as shown in the following section). This value must remain unchanged to enable the hooks for reading old versions to be used.

The following is an example of defining the value for the constant:

```
static final long serialVersionUID = 1;
```

Externalization methods

In the generated externalization methods, all non-transient, non-static fields that were modeled will be handled. The externalization is generated in a manner that provides a hook (`readOldVersion`) for reading in previous versions of the class, which have been externalized. Code generation detects when the externalizable signature of a class changes, and changes its internal version UID accordingly.

You have the ability to take control of the externalization code, but it is not highly recommended because it requires careful management of the externalizable signature of the class.

Examples of the externalization methods follow:

```
// Example of a writeExternal method
public void writeExternal( ObjectOutputStream output )
    throws IOException {
    /**begin writeExternal% [ ]writeExternal.body preserve=no

    output.writeLong( EXTERNALIZATION_VERSION_UID );

    super.writeExternal( output );

    output.writeObject( a1 );
    output.writeObject( a2 );
    output.writeObject( a3 );
    output.writeObject( list );
    output.writeObject( (size == null - null :
        size.getStringValue()) );
    output.writeObject( timeline );
    output.writeObject( work );
    /**end writeExternal% [ ]writeExternal.body
}

// Example of a readExternal method
protected boolean readVersion( MyItem thisObject,
    ObjectInput input,
    long readSerialVersionUID, boolean passThrough,
    boolean superDone )
    throws IOException, ClassNotFoundException {
    /**begin readVersion% [ ]readVersion.body preserve=no

    boolean success = true;

    if ( readSerialVersionUID == EXTERNALIZATION_VERSION_UID ) {
        if ( !superDone )
            super.readExternal( input );

        a1 = (String)input.readObject();
        a2 = (Date)input.readObject();
        a3 = (Xyz)input.readObject();
        list = (Vector)input.readObject();
        String size_string_value = (String)input.readObject();
        try { size = (MySize)wt.fc.EnumeratedType.toEnumeratedType(
            size_string_value ); }
        // in case old format
        catch( wt.util.WTInvalidParameterException e ) {
            size = MySize.toMySize( size_string_value );
        }
        timeline = (Timeline)input.readObject();
        work = (MyAddress)input.readObject();
    }
    else
        success = readOldVersion( input, readSerialVersionUID,
            passThrough, superDone );
}
```

```

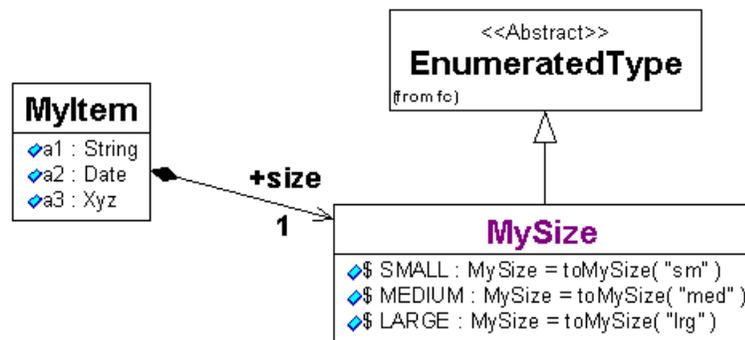
return success;
//##end readVersion% [ ]readVersion.body
}

```

Extending the EnumeratedType class

The EnumeratedType class is a wrapper for a string whose valid values are constrained to a defined set. A type, or class, can be created to define a specific set of values by extending the EnumeratedType class.

The following model provides an example of creating an enumerated type. MySize also shows the modeling of constants that reference certain values of the set's valid values.



Extending EnumeratedTypes

An example of the generated enumerated type follows:

```

public class MySize extends EnumeratedType {
    // define the name of the resource bundle
    private static final String CLASS_RESOURCE = MySizeRB;
    // initialization of programmatic constants
    public static final MySize SMALL = toMySize( "sm" );
    public static final MySize MEDIUM = toMySize( "med" );
    public static final MySize LARGE = toMySize( "lrg" );
    // Obtain a MySize value from an internal value
    public static MySize toMySize( String a_value )
    // Obtain the default MySize "value"
    public static MySize getMySizeDefault()
    // Obtain the set of valid MySize "values"
    public static MySize[] getMySizeSet()
    // Obtain the set of valid MySize "values" (polymorphic)
    public EnumeratedType[] getValueSet()
}

```

There should be no need for a developer to change any of the implementation of a generated enumerated type, but they are generated with preserve markers to allow for developer enhancements, if the need should arise. Note that the Windchill tab on Rose specification dialogs for classes and attributes provides property set

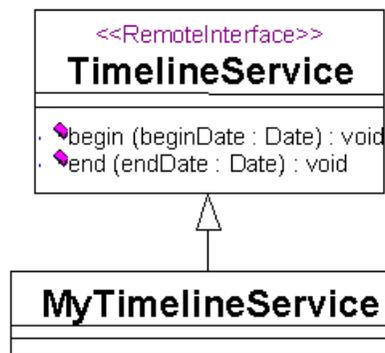
sheets specific to EnumeratedType usage. These properties are detailed in the model elements sections above.

For more information on working with enumerated types, see the *Windchill Customizer's Guide* appendix, Enumerated Types.

Stereotyping an interface as remote

When you model a service, you model interfaces and remote interfaces (see the chapter on developing server logic for more information). The <<RemoteInterface>> stereotype causes the code generator to generate a forwarder class that clients can use to invoke the method on the server. A forwarder class has the same name as the remote interface in the Rose model plus the suffix "Fwd" (for example, TimelineServiceFwd).

The following is a model of a business service, stereotyped as a RemoteInterface.



Business Service Model

The following is an example of a generated remote forwarder:

```
public class TimelineServiceFwd implements
    wt.method.RemoteAccess,
    TimelineService, Serializable {
    // constant that enables the object to know where it is
    static final boolean SERVER =
        RemoteMethodServer.ServerFlag;
    // identifies what type of service to forward to
    public Class getTypeClass() {
        return example.TimelineService.class;
    }
    public void begin( Date beginDate ) throws WTEException {
        if (SERVER)
            // invoke the service
        else {
            // remote call to the server
        }
    }
}
```

```
    public void end( Date endDate ) throws WTEException
  }
```

Services contain the complete business logic and are expected to run only on the method server, but forwarders go back and forth between the server and the client. From the client, they invoke the business service methods running on the server.

At runtime, the forwarder binds to a service which is determined by the registration of services that is done in the wt.properties file.

The forwarder classes are completely generated and provide no preserve markers for developer editing.

How Rose UML Maps to Info Objects

Introspection is part of the Java Beans specification. It is a standard method for providing metadata information about an object, such as its attributes and the methods it supports. Windchill extends that concept with Info objects.

Info objects contain class information from the Rose UML model that is needed by the runtime system. One Info object is generated for each modeled class that implements NetFactor, each modeled EnumeratedType subclass, and for each modeled interface. The system generation tool creates Info objects by capturing a subset of the class information stored in the Rose repository as serialized resource files that are read by the runtime system.

Info objects contain information for the following elements:

- Classes and interfaces
 - Descendents (that is, a list of subclasses, which allows you to determine what classes are inherited from a specific class)
 - Link membership
 - Property descriptors for attributes
 - Column descriptors containing attribute mapping information
 - Database table, package, and other parameters
- Links (When you have a link, you can also create a link between objects that are subtypes of the role A and role B objects. Therefore, you need to know which are valid and how many there can be.)
 - Role A and role B names
 - Valid role A and role B classes
 - Role A and role B cardinality
- Attributes of classes
 - Standard JavaBeans PropertyDescriptor with extended information

- Derived flag
- Persistent flag
- Updateable flag
- Query name (name of its column descriptor)
- StringCase (if a String type is forced to either uppercase or lowercase)
- Required flag
- Upper limit
- Method that returns set of valid values
- ConstrainedType (if type overridden in subclass)
- DefinedAs (class and attribute name as originally modeled)
- ModeledPersistentType (modeled column type)

For a complete description of the introspection information that is provided, see the Javadoc for the `wt.introspection` package.

Windchill provides a utility to print the content of a serialized Info object. You can run the utility as follows:

InfoReport *<fully.qualified.classname>*

The resulting report is in a file named *<qualified.classname>.out*, which is located in the directory specified by the `wt.temp` property in `wt.properties`.

To print the info contents for each class of an entire package, you can run the utility as follows:

InfoReport *<fully.qualified.*>*

Windchill also provides a utility that will verify the environment as it relates to introspection information. That is, it will verify the existence and ability to load all classes and serialized info objects that are referenced in the registry files. The utility simply writes its output to the console, with reported problems being preceded by three asterisks (***)

To verify the entire registry (the complete environment), enter the following command:

executeTool wt.introspection.VerifyEnvironment registry

To verify a single package, enter the following command:

executeTool wt.introspection.VerifyEnvironment *<fully.qualified.*>*

How Rose UML Maps to Database Schema

Table 8-1 shows how common Java types map to the JDBC specification standard SQL types and Oracle SQL types.

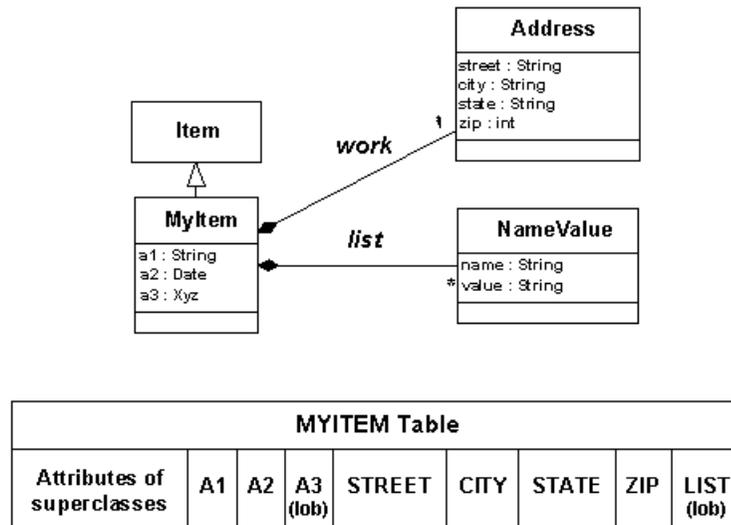
Java Type	SQL Type	Oracle SQL Type
java.lang.Integer(int)	INTEGER	NUMBER
java.lang.Long (long)	BIGINT	NUMBER
java.lang.Short (short)	SMALLINT	NUMBER
java.lang.Byte (byte)	TINYINT	NUMBER
java.lang..Float (float)	REAL	NUMBER
java.lang.Double (double)	DOUBLE	NUMBER
java.lang.Boolean (boolean)	BIT	NUMBER
java.lang.Char (char)	CHAR	CHAR
java.lang.String	VARCHAR	VARCHAR
java.math.BigDecimal	NUMERIC	NUMBER
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATE
java.sql.Timestamp	TIMESTAMP	DATE
java.sql.Blob	BLOB	BLOB
wt.fc.LobLocator	BLOB	BLOB
WTypes.SMALLBLOB	Not applicable	VARCHAR (encoded)
WTypes.INLINEBLOB	Not applicable	VARCHAR (encoded, if fits) BLOB (if size exceeds limit of VARCHAR column)
WTypes.SEQUENCE	NUMERIC	NUMBER

Java to SQL Mapping

Each persistable class maps to a database table. Each persistent attribute in the class is mapped to a column name in the table. Structured attributes, like the Address class shown earlier, are decomposed into their simple attributes. Attributes are stored as lobes when no SQL-supported mapping is found, or if the cardinality is greater than one (because SQL cannot allocate space for an indefinite number of columns and rows).

Tables contain all the attributes they inherit from classes above them. Because a class must be persistable to be in the database, all tables contain the persistInfo attributes, including the object identifier.

The figure below shows a representation (not the actual columns) of the table that is generated for the modeled class.



Mapping a Model to the Database

MyItem is persistable because it extends Item, which extends WToObject, which implements the Persistable interface. Therefore, a table is created in the database for the objects in MyItem.

The first element shown in the table, Attributes of superclasses, represents the persistent attributes (from persistInfo) that MyItem inherits from its superclasses. Each of the attributes would be mapped to a separate column in the actual table. One of those columns would be the object identifier.

Attribute a1 is a String, so column A1 is a VARCHAR. Attribute a2 is a Date (actually a java.sql.Date), so column A2 is a DATE. Attribute a3 has no SQL mapping support but is serializable so the value for a3 is stored in column A3 as a BLOB.

MyItem has a one-to-one composite aggregation relationship with Address, so the attributes of the Address class are mapped to the MyItem table. Windchill uses a naming algorithm to map attributes of an embedded structured attribute to Oracle table column names. To list the mapping, use the InfoReport utility mentioned in the section on how Rose UML maps to Info objects, earlier in this chapter.

MyItem has a one-to-many composite aggregation relationship with NameValue, so NameValue is stored as a BLOB.

By default, the class name is mapped to the table name. You can specify a different name with the Rose specification table name property, for example, if the name is a duplication of another name or a reserved word in Oracle (such as VIEW).

Some additional properties can be specified to influence Oracle storage options. By default, all Windchill tables and indexes are created in the Windchill user's default tablespace. A tablespace is a logical collection of tables and indexes. Data for the table can be stored elsewhere by specifying a TableName. If an IndexTableName is specified, it is used to store the indexes for the table. The TableSize (TINY, SMALL, MEDIUM, LARGE, or HUGE) is used together with the associated properties in wt.tools.properties to specify storage parameters for Oracle.

The default storage mechanism for BLOB attributes is to store them in their own tablespace. The name of the tablespace is a configurable property in the tools.properties file. The default name is BLOBS. For further information, see the section on creating large objects in the *Windchill Customizer's Guide*.

Each attribute of the table also has properties which can be specified for the DDL generation, such as whether to allow NULLS or whether to index the attribute. If more complex indexing is required, then the Rose Class specification Windchill index properties should be used. There are several index entries available for specifying composite indices. Composite unique indices are also supported. See the Rose Windchill Help documentation for more information. Typically indices are generated into the associated table creation script. However, a new utility, IndexGenerator, is also available that generates scripts containing only indices.

For complex DDL statements associated with a class that cannot be handled by the SQL generation support, the user additions script feature is available. It is recommended that this feature only be used if absolutely necessary. When the Rose modeling DDL generation features are used, this information can also be used by the upgrade tools to ensure that the DDL is migrated properly. Every generated sql script contains a line at the end to execute the statements in an associated sql file. If the <tablename>_UserAdditions script is located in the same directory as the table script, then this line is not commented out of the script that creates the table.

For a many-to-one aggregation implemented via a foreign key, the autoNavigate property can be used. When the autoNavigate property is set to true, the runtime query APIs treat the columns on the base table along with the columns of the

aggregated table in much the same way as a database view. The two tables are implicitly joined via the foreign key.

There are two types of sequence data types available. For simple sequences, you can make the attribute type Long and select SEQUENCE from the WT_Oracle tab for the column type. This causes the DDL generation to create a sequence (if it does not already exist) named `_seq`, and increment the Oracle sequence value to set the value of the attribute when the object is inserted into the database.

If you prefer to assign the sequence number at a later time, you can use `PersistenceManager.getNextSequence()` to obtain the next available sequence number for the named sequence. That interface returns a `String`, so you would either model the attribute as a `String` or convert it to whatever you need. You must ensure that the sequence is created using either the Oracle `CREATE SEQUENCE` command or the stored procedure provided by Windchill:

```
exec wtpk.createSequence(MySequence',1,1)'
```

For further information, see `getNextSequence` in chapter 6, `Persistence Management`.

All the database scripts necessary for a given Java package are generated to an associated subdirectory of the database generation directory. In addition, "make" scripts are generated for the entire package. The "Make_<package name>" script calls all the scripts for creating tables. The "make" scripts will also call the associated make scripts in any nested packages.

How Rose UML Maps to Localizable Resource Info Files

Since Windchill is localized to run in a variety of locale specific languages, default display names are generated into localizable resource info (`.rbInfo`) files, for each model element name that might need to appear on a user interface. Language translators can then create localized versions of these files, so that modeled elements can be displayed in a locale specific language.

The `Localizable` property is available on the Windchill tab of the Package, Class, Attribute and Association Role specifications in Rose. The purpose of this property is to limit the entries in the resource info files to those that need to be translated.

The `Class Localizable` property is an enumerated type, and has three possible values: `<Default>`, `True` and `False`. The default value is `<Default>`, which implies that the code generator will inspect the Package level value to determine the value at the Class level. The Class will be considered `Localizable` by the localizable display generator under the following circumstances:

- The `Class Localizable` property is explicitly set to `True`
- The `Class Localizable` property is set to `<Default>`, and the Class implements `ObjectMappable` and the Package level value is `True`

The Localizable property for Attributes and Roles is also an enumerated type, and thus has possible values: <Default>, True and False. The default value is <Default>, which implies that the code generator will inspect the Class level value to determine the value at this level. These elements will be considered Localizable by the localizable display generator under either of the following circumstances:

- The element's Localizable property is explicitly set to True
- The element's Localizable property is set to <Default>, and its containing Class is considered Localizable.
- If the resource info file contains an entry for a particular model element, the generator will not overwrite it, but it can remove it if the modeled element no longer exists, or if it is no longer considered localizable. The removal of elements is controlled by the wt.generation.cleanupDisplayNames property in tools.properties. For more information on creating localizations, see the Internationalization and Localization.

The generated resource info files should only be edited by the owner of the package. The resource info tool set provides a means of customization, where the customers of the package can define customizations in a separate file. For more information on customizing localizable resource entries, see the Customizing Modeled Elements chapter of the Windchill Customizer's Guide.

The name of the file is "ModelRB.rbInfo", with the simple package name prepended, for example, partModelRB.rbInfo. The following sections describe the generated resource info files.

Header

Each resource info file contains the following lines that define certain file level information.

```
ResourceInfo.class=wt.tools.resource.MetadataResourceInfo
```

```
ResourceInfo.customizable=true
```

```
ResourceInfo.deprecated=false
```

Resource Entry Format

Entry Format (values equal to default value are not included)

```
#<key.value=
```

```
#<key.comment=
```

```
#<key.argComment<n>=
```

```
#<key.constant=
```

```
#<key.customizable=
```

```
#<key.deprecated=
```

```
#<key>.abbreviatedDisplay=  
#<key>.fullDisplay=  
#<key>.shortDescription=  
#<key>.longDescription=
```

The <key>.value is the only key that is required to have a value defined. This one, along with the two Display values, and the two Description values are the only ones that are localizable. The <key>.constant value is unused for these metadata display name entries, but is used for other types of resource info files.

Resource Entry Contents

Below are examples of the default entries that are generated.

```
WTPart.value=Part  
WTPart.partType.value=Part Type
```

The first entry is the display name for the WTPart class, and the second is the display name for partType attribute of the WTPart class. These values will only be generated once, so that the package owner can override the default values, if desired. For example, the value for the partType attribute could be changed as follows:

```
WTPart.partType.value=Type
```

In addition, any of the keys referred to, in the format section above, could be added for an element. For example, the following information could be added for the partType attribute.

```
WTPart.partType.fullDisplay=Part Type  
WTPart.partType.longDescription=Classification of parts according to  
predefined types.
```

Building Runtime Resources

Windchill provides a utility to build the runtime resource, for the .rbInfo files. To build the runtime resources into the codebase for a particular modelRB file:

```
ResourceBuild <absolute_directory>  
e.g. ResourceBuild %wt_home%\src\wt\part
```

To build the runtime resource into the codebase for all the resource info files for a particular directory:

```
ResourceBuild <directory\relative\to\src>  
e.g. ResourceBuild wt\part
```

The resulting resource file is named <name> RB.ser, which is a serialized instance of SerializedResourceBundle. For example, src\wt\part\partModelRB.rbInfo will build to codebase\wt\part\partModelRB.RB.ser.

To print the contents of the serialized resource bundle, you can run the following utility:

```
java wt.util.resource.ResourceBundleUtil <fully.qualified.name [<locale>]
```

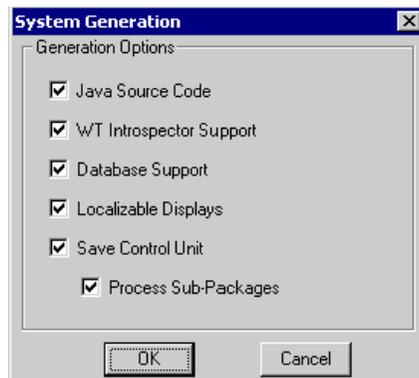
```
e.g. java wt.util.resource.ResourceBundleUtil wt.part.partModelRB en_GB
```

```
output goes to: $(wt.temp)part.partModelRB_en_GB.lst
```

Using the Windchill System Generation Tool

When you are ready to generate code, the packages being generated must be loaded in the model. Any model dependencies will automatically be loaded as a result of generating. Therefore, it is not necessary to load the entire model, or even to manually load dependent packages. Also, ensure that your WT_WORK Rose Edit Path variable specifies the correct location for the generated mData (model data) files.

To use the Windchill System Generation Tool from Rose, select the classes and packages for which you would like to generate code. Go to the Tools menu, select the Windchill menu item, and then select the System Generation option. A popup window allows you to select the desired system generation options. WT Introspector Support creates the Info objects. Database Support creates the DDL scripts for the database. Localizable Displays updates the resource info (rbInfo) files for the generated packages.



System Generation Dialog

As mentioned in the overview at the beginning of this chapter, the system generator creates mData files which, in turn, are used to create Java code, Info files, SQL files and Resource Info files. The system generator also updates the classRegistry, descendentRegistry and associationRegistry, properties files.

Registry Files

The classRegistry, descendentRegistry, and associationRegistry files are located in the directory specified by the wt.generation.bin.dir entry in wt.tools.properties.

The classRegistry file contains all the information the code generator needs to create import statements for the generated Java classes. When classRegistry is initialized, it goes through all the directories in your CLASSPATH variable and makes an entry for every class it finds. The classRegistry must be deleted and regenerated if you change your CLASSPATH to include new packages that contain classes for which the code generator will need to generate imports.

As the number of CLASSPATH packages increases, the potential for conflicts increases. These conflicts are best resolved by explicitly modeling package level dependencies in Rose. The generator will then use this information to resolve any ambiguities that occur due to having multiple classRegistry entries for a particular class name.

The descendentRegistry file contains genealogy information about which classes extend the classes above them. It is required to support subclass queries. The associationRegistry contains information about Persistable associations.

Classes that are renamed or removed from a package will be automatically removed from the registries upon a regeneration of that package. A complete package can be removed from the registries (uninstalled), by using the modelUnInstall script.

Generating mData Files

An mData file is an ASCII file in non-proprietary format that contains all the information the Windchill system generation tool needs. It is an intermediate file format that allows for support of other model specification tools in the future.

The mData files contain all of the information about the models. The export tool creates one mData file for each package in Rose and puts them in the WT_WORK directory.

The mData files have a directory structure that maps to the package structure inside Rose. For example, because there is an fc package in Rose, there is a corresponding fc directory in the WT_WORK directory. In that fc directory is one mData with the name of the package (for example, fc.mData).

Using mData files, the system generator creates Java classes and Info files. Java classes are source code that is generated into the source tree directory. Info files, because they are serialized resource files, are generated into the directory where Java classes are located.

Generating Java Code

As described earlier in this chapter, one Java class is generated for each modeled class. In addition, classes modeled as remote interfaces (such as

TimelineService.java) will have forward classes generated for the service (such as TimelineServiceFwd.java).

All the generated Java classes are put into directories, specified by the wt.generation.source.dir entry in wt.tools.properties, which follow the package structure. All the classes generated for a particular package are placed in the corresponding directory.

Generating Info Files

Info files contain all the metadata needed by the run time system. They are generated from the corresponding mData files, one Info file for every interface, and one for each class that implements NetFactor. Info files are put into directories, specified by the wt.generation.bin entry in wt.tools.properties, using a directory structure that follows the package structure. The generated file name is the same as the class name plus the suffix ClassInfo.ser (such as MyItem.ClassInfo.ser).

Generating SQL Files

SQL files contain directives for generating Oracle tables for persistable classes. On the System Generation window, you can choose Database Support and the system generator creates the SQL files for you. If you do not choose that option, the system generator creates the information necessary for you to generate the SQL files yourself. Input always comes from the corresponding Info files.

A utility is provided which allows you to generate SQL yourself. For each persistent class from your model, enter the following command:

```
JavaGen "<your package .your classname>" F F True
```

Or, to generate the SQL for an entire package, enter the following command:

```
JavaGen "<your package >.*" F F true
```

When specifying your class, include the fully qualified name of the class, which includes the parent package names.

If you want to change the default storage parameters in the tools.properties file and recreate all the SQL, Windchill DDL can be regenerated for all modeled and registered classes by entering the following command:

```
JavaGen registry F F true
```

DDL is generated into the sql subdirectory named by the wt.generation.sql.dir property you specified in the wt.tools.properties file. File names are of the form **create_ <tablename >.sql**.

Regardless of how you generated the SQL files, you must create the Oracle tables using SQLPLUS. Ensure that your SQLPATH environment variable is correct or, in the command you enter next, specify the full path to the SQL file. Use the following command:

sqlplus <user/password >serviceName > @create_ <your table name >

If you have classes and tables, it is useful to create a single SQLPLUS script that can execute each @create script. (The create_ddl_wt.sql script is an example.)

The following SQL sample shows the directives for creating the MyItem table:

```
exec WTPK.dropTable(MyItem)'  
set echo on  
REM Creating table MyItem for example.MyItem  
set echo off  
CREATE TABLE MyItem (  
    a1 VARCHAR2(200),  
    a2 DATE,  
    a3 BLOB,  
    classnameA2domainRef VARCHAR2(60),  
    idA2domainRef NUMBER,  
    list BLOB NOT NULL,  
    size VARCHAR2(200) NOT NULL,  
    createStampA2 DATE,  
    modifyStampA2 DATE,  
    classnameA2A2 VARCHAR2(60),  
    idA2A2 NUMBER,  
    updateCountA2 NUMBER,  
    updateStampA2 DATE,  
    effectiveDatetimeline DATE,  
    terminationDatetimeline DATE,  
    cityA5 VARCHAR2(200),  
    stateA5 VARCHAR2(200),  
    streetA5 VARCHAR2(200),  
    zipA5 NUMBER,  
    CONSTRAINT PK_MyItem PRIMARY KEY (idA2A2))  
    STORAGE ( INITIAL 20k NEXT 20k PCTINCREASE 0 )  
LOB ( a3 ) STORE AS  
    (TABLESPACE blobs  
    STORAGE (INITIAL 100k NEXT 100k PCTINCREASE 0)  
    CHUNK 32K)  
LOB ( list ) STORE AS  
    (TABLESPACE blobs  
    STORAGE (INITIAL 100k NEXT 100k PCTINCREASE 0)  
    CHUNK 32K)  
/  
COMMENT ON TABLE MyItem IS 'Table MyItem created for  
example.MyItem on 17-Jun-98 at 9:09:31 AM'
```

The first line, which executes the dropTable procedure of the WTPK package, drops the MyItem table if it already exists. The next line is an informational message that is displayed within SQL*Plus as the table is being created. Then the table is created, with a primary key constraint on the object identifier. The storage clause indicates that an initial extent of size 20K will be allocated by Oracle. This corresponds to a SMALL size table. The last two lines insert a comment into Oracle's data dictionary to document when and why the table was created.

These comments can be viewed using SQL*Plus by entering the following command:

```
select comments from user_tab_comments where table_name = 'MYITEM';'
```

Using Windchill System Generation in a Build Environment

Using the Windchill System Generation tools in an integrated build environment requires an understanding of the dependencies that need to be managed. The following table shows the input and output relationships between the various artifacts of system generation.

Process	Input(s)	Outputs
Export Model Data	Rose model (.mdl and/or .cat)	mData file
Populate Model Registries	mData files for all dependent packages	modelRegistry.properties – descendentRegistry.properties ¹ associationRegistry.properties ¹
Populate Class Registry	tools.properties – wt.classRegistry.search.path tools.properties – wt.classRegistry.search.pattern	classRegistry.properties
Generate Java Source	classRegistry.properties – modelRegistry.properties registered mData files – mData file for target package	.java files modelRegistry.properties descendentRegistry.properties associationRegistry.properties
Generate Info Objects	modelRegistry.properties registered mData files mData file for target package	.ClassInfo.ser files .modelRegistry.properties descendentRegistry.propertiesN associationRegistry.properties
Generate SQL Scripts	.ClassInfo.ser files for target package and dependents	.sql files
Generate Resource Info	.mData file for target package	.rbInfo files
Build Resource Bundles	.rbInfo files for target package	.RB.ser files

1. Accumulated during generation, but used to define the modeled relationships for the installed runtime application.

Management of the mData File

An mData (model data) file is generated from the Rose .mdl and/or .cat file by using the Windchill Generation option on the Tools menu in Rose. If the mData

will be used to generate code in an integrated build environment, the mData file must be kept synchronized with the cat file. This is best accomplished by always checking in and checking out the two files in tandem, and ensuring that the generated mData reflects the saved version of the cat file.

Build Sequence

The following table describes the steps and commands used in a build sequence.

Step	Description	Command
1	Create classRegistry, based on search path and pattern defined in tools.properties. ¹	RegistryGen
2	Register all models that will be integrated	ModelInstall "<package>.*"
3	Generate Java source code, info objects, SQL and resource bundle files scripts for target packages. ²	JavaGen "<package>.*"
4	Build the resource bundles for target directory.	ResourceBuild "<dir\sub>"
5	Compile Java source code.	javac ...

1. The system generation tool will create a classRegistry if one does not exist.
2. Remember to update database table structures after generating SQL scripts that have structural changes.

Command Line Utilities

RegistryGen

Creates a class registry from scratch, using the following two properties from tools.properties, to determine what to include in the registry:

- wt.classRegistry.search.path
- wt.classRegistry.search.pattern

ModelInstall

Updates the model registries with all the modeled packages specified by Arg1. Arg1 can be either of the following:

"<package>.*"

Registers the single package specified.

"<package>.*"

Registers the package for each mData found recursively under the <package> directory.

ModelUnInstall

Removes the modeled packages specified by Arg1 from the model registries. Arg1 can be either of the following:

"<package>.*"

Removes the single package specified.

"<package>.*"

Removes the package for each mData found recursively under the <package> directory.

JavaGen

Generates the outputs specified by the arguments for the classes/packages specified by Arg1. Arg1 can be any of the following:

"<package>.<Class>"

Generates outputs for the single class specified.

"<package>.*"

Generates outputs for the single package specified.

"<package>.*"

Generates outputs for each mData found recursively under the <package> directory.

registry

Generates outputs for all registered packages.

The following additional arguments can be specified:

Arg2

Specifies if Java Source will be generated.

Arg3

Specifies if Introspector Info Objects will be generated.

Arg4

Specifies if SQL scripts will be generated.

Arg5

Specifies if Model Check will be performed.

Values for arguments not passed will be obtained from tools.properties.

Arg6

Specifies if Display Names will be generated into resource info files.

ResourceBuild

Builds the runtime resource bundles for the packages/directories specified by Arg1. Arg1 can be any of the following:

“<package>.<resource>”

Builds resource bundle for the single resource info.

"<absolute_dir\sub>"

Builds resource bundles for each resource info found under in the directory. (“<absolute_dir\sub>” is an absolute directory.)

The following additional arguments can be specified:

Arg2

Specifies if locale specific files will be built. (wt.locale.set in wt.properties define valid locales)

InfoReport

Reports the contents of info objects to a text file for classes/packages specified by Arg1. Arg1 can be any of the following:

“<package>.<Class>”

Reports for the single class specified.

“<package>.*”

Reports for the single package specified.

registry

Reports for all registered packages.

VerifyEnvironment

Verifies the existence and ability to load all classes and serialized info objects that are referenced in the registry files. Arg1 can be any of the following:

executeTool wt.introspection.VerifyEnvironment "<package>.<Class>"

Verifies the single class specified.

executeTool wt.introspection.VerifyEnvironment "<package>.*"

Verifies the single package specified.

executeTool wt.introspection.VerifyEnvironment registry

Verifies all registered packages.

IndexGenerator

Generates index creation (and drop) DDL to a standalone SQL script. The utility supports both a single package and all indices for all modeled packages.

executeApp wt.tools.generation.sql.OracleIndexGenerator "*" *"create file name" "drop file name"*

Generates index creation DDL for all modeled packages to the specified files. Files names must be absolute. The "*" parameter must be specified in quotes.

executeApp wt.tools.generation.sql.OracleIndexGenerator
"<package>.*" *"create file name" "drop file name"*

Generates index creation DDL for the package to the specified files. If a file path is not absolute, then it is treated as path relative to the wt.generation.sql.dir directory.

10

Developing Client Logic

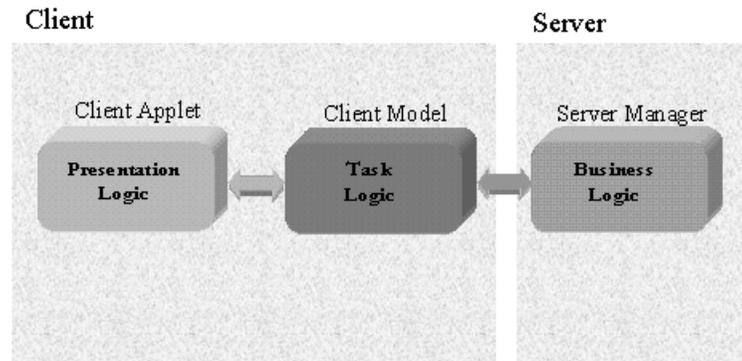
The process used to design Windchill applications is described in appendix E, GUI Design Process. You are not required to follow this process to write your own applications. It is included as an example and to give some background as to how Windchill applications, especially the GUI components, were designed. Likewise, the client programming model described in this section is included to show you how Windchill applications were developed.

In addition to the information presented in this chapter, see also the *Windchill Customizer's Guide* for descriptions of how to use and customize many of the Windchill components used in client development.

Topic	Page
The Client Programming Model.....	10-2
Presentation Logic	10-3
Integrated Development Environments (IDEs).....	10-3
Task Logic.....	10-3
Windchill Java Beans	10-6
HTTPUploadDownload Panel Bean	10-58
Clipboard Support	10-62
Programming Outside the Sandbox Using security.jar	10-63
Threading.....	10-65
Threading.....	10-65
Preferences	10-72
Online Help	10-69
Using Batch Containers.....	10-74
The Rbinfo File Format.....	10-86

The Client Programming Model

In the programming model we use, the client application is divided into two components: the presentation logic for the client applet and the task logic for the client model. The presentation logic implements the look and feel (that is, the user interface) of the application. The task logic implements the client model and interacts with the business logic of the application in the server.



The Client Programming Model

Separating the presentation and the task logic provides several benefits:

- Independence
Changes made to one of the components need not have a corresponding effect on the other component. If the logic needed to implement a task is changed, changing the model has no effect on the presentation. For example, suppose a change is made so the results of a database query are returned as an array of objects rather than a `QueryResult`. Because this change is contained in the implementation of the task logic, the presentation can be reused without modification. Similarly, changes made to the user interface of a client application do not affect the implementation of the task logic.
- Reuse
Multiple implementations of the presentation can be used with the same implementation of the model. For example, suppose a developer wants to create a user interface to support users with a keyboard but not a mouse. By partitioning the client application, the developer need only create another user interface to support keyboard input. The same model can be used for both applications.
- Extensibility
The model can be extended to support evolving business logic.

Presentation Logic

The presentation is the portion of the client application that implements its look and feel and provides the mechanism through which the end user interacts with the application. As a result of your GUI design process (the process used at Windchill is described in Appendix E, GUI Design Process), you should have identified how the application should look as well as some degree of how the user interacts with the application.

When considering the layout of your application, consider how the application will respond to resizing the window. Many visual development environments provide a default layout that makes creating user interfaces very straightforward. However, these default layouts often result in user interfaces in which the locations of the various widgets are hard-coded. Consequently, these user interfaces do not adjust appropriately when the window is resized.

Similarly, consider layout when you create applications that must be internationalized. The layout must support the changes that result from changing the locale of the application (for example, labels can become significantly longer, text could use fonts that are larger, and so on).

Another issue you must be conscious of is that of programming with threads. Event-handlers, which are invoked in response to the end user interacting with the application, are often run in a separate thread, allowing the end user to continue interacting with the application while the process is going on (printing, for example).

Integrated Development Environments (IDEs)

Windchill applications are developed using an Integrated Development Environment (IDE). If you want to customize portions of our GUI, or if you are creating your own GUI, you are free to use whatever IDE you choose.

Task Logic

The task logic implements the model. The model represents how the objects in the application domain interact with each other to perform tasks, completely independent of how the application looks or how the end user interacts with the application. The objects and tasks are expected to have been identified as the result of a design process (for example, the design process used to develop Windchill applications is described in appendix E, GUI Design Process).

Implementing the task logic that has been defined involves three activities:

- Communicating with the method server and the database.
- Identifying and supporting the data to be cached on the client.
- Providing an API (that is, a set of methods) that can be used by the presentation portion of the application to retrieve information.

Essentially, tasks can be built with any of the following:

- Invoking methods on the server manager objects that result in communication with the server and ultimately the database.
- Client-side validation.
- Interactions between business objects (business processing logic).

The following example shows how you can use a task scenario (a typical result from a design process) to decide how to create the task logic.

- An incident report is created by a particular customer and applies to a particular product. When a customer fills out an incident report, there are certain pieces of information that the customer is required to provide. When the customer has finished filling in the details of the incident report, the incident report is saved in a persistent data store.

The first sentence implies that the task will involve interactions between Incident Report, Customer, and Product business objects.

- The second sentence implies that the task will involve client-side validation of certain pieces of data.
- The third sentence implies that the task will involve communication with the method server, which will ultimately result in the incident report being saved in the database.

Following are some code examples of task logic.

Interacting with Business Objects

The following example shows interaction with business objects. In this example, interaction with the product object is required to determine if the product name, version, and release level correspond to an existing product.

```
public String createIncidentReport( String product_name,
    String product_version,
        String product_release, IncidentReport inc_report )
    throws InvalidAttributeException, WTEException {
    String track_number = new String("");
    if( inc_report == null ) {
        throw new WTEException( null, RESOURCE, "nullIncReport", null);
    }
    if( product_name == null ) {
        throw new WTEException( null, RESOURCE, "nullProductName", null);
    }
    int i = 0;
    boolean found = false;
    Product current_product = null;
    while( ( i < products.size() ) &&
        ( !found ) ) {
        current_product = (Product) products.elementAt(i);
```

```

        if( ( current_product.getName().equals( product_name ) ) &&
            (current_product.getVersionLevel().equals(product_version) )
            &&
            ( current_product.getReleaseLevel().equals(
                product_release ) ) ) {
            found = true;
        }

        i++;
    } // End of 'while' Loop

```

Client-Side Validation

The following example shows client-side validation. In this example, validation is performed to ensure that the user has completed both the incident summary and the incident detail. If the incident summary or incident detail is null or an empty string, an `InvalidAttributeException` is thrown.

```

while( ( i < products.size() ) &&
        ( !found ) ) {
    current_product = (Product) products.elementAt(i);

    if( ( current_product.getName().equals( product_name ) ) &&
        ( current_product.getVersionLevel().equals( product_version ) ) &&
        ( current_product.getReleaseLevel().equals( product_release ) ) ) {
        found = true;
    }

    i++;
} // End of while'Loop'
if( !found ) {
    Object params[] = { curCustomer.getCompanyName(),
                       product_name, product_version, product_release };
    throw new WTEException( null, RESOURCE, "productNotFound", params );
}

if( ( inc_report.getIncidentSummary().equals("") ) &&
    ( inc_report.getIncidentDetail().equals("") ) ) {
    throw new InvalidAttributeException( null, RESOURCE,
                                         "nullIncSummaryAndDetails", null);
}

```

Invoking Server Methods

The following example shows how server methods are invoked. In this example, the `submitIncidentReport` method is invoked on the `HelpdeskMgr` on the server manager. Initially, checking is done to ensure the fields are filled in correctly. Then the method is invoked on the server to create an incident report using the customer and product parameters that are passed. The `HelpdeskMgr` creates the necessary links between these business objects and handles the saving of this information in the database.

```

if( ( inc_report.getIncidentSummary().equals("") ) &&
    ( inc_report.getIncidentDetail().equals("") ) ) {

```

```

        throw new InvalidAttributeException( null, RESOURCE,
                                           "nullIncSummaryAndDetails", null);
    } else if( inc_report.getIncidentSummary().equals("") ) {
        throw new InvalidAttributeException( null, RESOURCE,
                                           "nullIncSummary", null);
    } else if( inc_report.getIncidentDetail().equals("") ) {
        throw new InvalidAttributeException( null, RESOURCE,
                                           "nullIncDetails", null);
    }
}

try {
inc_report=HelpdeskMgr.submitIncidentReport(inc_report,
                                           curCustomer,
                                           current_product );
    incidentReports.addElement( inc_report );
} catch (WTEException wte) {
    throw new WTEException( wte, RESOURCE, "submitIncReportFailed",
                           null );
}

track_number = inc_report.getTrackingNumber();

return track_number;

```

Windchill Java Beans

This section describes some of the Java bean components that have been developed for use with Windchill clients and how to use them. Examples or screen shots use Visual Cafe as an example of a supported IDE, but Windchill supports any Java IDE customers chooses to use.

Importing Windchill Beans to Visual Cafe

The Windchill beans are packaged in a jar file, wtbeans.jar, in your Windchill\lib directory. To use the beans within the Visual Cafe environment, you must first add the wtbeans.jar file to the Visual Cafe Component Library. To do this, perform the following steps:

1. On the Tools menu, select Environment Options > Internal VM.
2. Add your Windchill\codebase directory to the SC.INI CLASSPATH.
3. Click OK to close the Environment Options dialog box and a message stating the changes will go into effect the next time you start Visual Cafe appears. Exit and then restart Visual Cafe.
4. On the File menu, select Add Component to Library. When the Add Component to Library dialog box appears, navigate to the Windchill\lib directory and select wtbeans.jar.
5. After loading the wtbeans.jar file, Visual Cafe should display the message: Successfully added 'n' component(s) from wtbeans.jar to the Component Library.'

You can verify that the Windchill beans have been added to your component library by viewing the contents of the library (select the Component Library option in the View menu in Visual Cafe). You should see a folder named wtbeans.

Once the wtbeans folder is in your component library, you can open it, and drag and drop the individual beans onto the GUI you are creating.

Package Dependencies

The Windchill beans are dependent on the following packages which must be installed before using the beans:

- JDK 1.1.5 or higher
- symantec.itools.awt
- Windchill/codebase

WTContentHolder Bean

The WTContentHolder bean (wt.clients.contentholder.WTContentHolder) provides support for manipulating the content of objects that implement the wt.content.ContentHolder interface. WTContentHolder is useful for creating user interfaces to create, view, and update such content-holding objects. This bean provides support for the following manipulations on a ContentHolder object:

- Adding content files.
- Adding URLs.
- Removing content files.
- Removing URLs.
- Updating the attributes of a file.
- Replacing an existing content file with a new content file.
- Updating the properties of a URL.
- Viewing the properties of a file.
- Viewing the properties of a URL.
- Downloading the file contents.
- Opening a URL.

Currently, only a few properties of the WTContentHolder can be configured at design time. From the Visual Café Property List, in the figure below, you can see the mode of the WTContentHolder.



Property List

The WTContentHolder bean supports the following modes:

- 0** Create: supports adding and removing files and URLs, getting files and URLs, and viewing the properties of files and URLs.
- 1** Update: supports adding and removing files and URLs, updating files and URLs, getting files and URLs, and viewing the properties of files and URLs.
- 2** View: supports getting files and URLs, and viewing the properties of files and URLs.

Building the GUI is complete. What remains, then, is tying in the WTContentHolder to the task logic of updating a Report. To be able to manipulate files and URLs with the WTContentHolder, the WTContentHolder must be initialized with an object that implements the ContentHolder interface. Further, this ContentHolder object must be persistent.

```

public class UpdateReportFrame extends java.awt.Frame {
    ...
    public void setReport( Report report ) {
        if( report != null ) {
            this.report = report;

            try {
                myWTContentHolder.setContentHolder( report );
            } catch (WTPROPERTYVETOException wtpve) {
                wtpve.printStackTrace();
            }
        }
    }
    ...
} // End class 'UpdateReportFrame'

```

Notice that after invoking `setReport`, both the `UpdateReportFrame` and the `WTContentHolder` used within this frame have a reference to the `ContentHolder` object. The `UpdateReportFrame` has a reference in its local variable — `report` — and the `WTContentHolder` bean has a reference which is set in the call to `setContentHolder(report)`. With both the frame and the bean having different references to the same `ContentHolder` object, the programmer of the frame must take care to ensure that both references remain current. For example, if a content change is made using the `WTContentHolder`, and the `ContentHolder` is updated, the copy of the `ContentHolder` maintained by the `UpdateReportFrame` has become out of date.

The `WTContentHolder` bean provides an API that helps guard against references to the same object getting out of sync. In addition to having an API that directly manipulates a `ContentHolder` object, the `WTContentHolder` also has an API that manipulates a `ContentHolder` via a `wt.clients.util.ReferenceHolder`. By manipulating a `ReferenceHolder`, both the `UpdateReportFrame` and the contained `WTContentHolder` manipulate the same `ContentHolder` object:

```

public class UpdateReportFrame extends java.awt.Frame {
    ...
    public void setReport( Report report ) {
        if( report != null ) {
            this.reportHandle = new ReportHandle( report );

            try {
                myWTContentHolder.setContentReference(
                    reportHandle );
            } catch (WTPROPERTYVETOException wtpve) {
                wtpve.printStackTrace();
            }
        }
    }
    ...
}

```

```

class ReportHandle implements ReferenceHolder {
    Report report = null;

    public ReportHandle( Report report ) {
        this.report = report;
    }

    public Object getObject() {
        return report;
    }

    public void setObject( Object obj ) {
        report = obj;
    }

} // End of inner class 'ReportHandle'
} // End class 'UpdateReportFrame'

```

Essentially, these are the only two actions needed to use the WTContentHolder bean: adding the WTContentHolder to the parenting GUI screen, and initializing the WTContentHolder with an appropriate ContentHolder object.

The WTContentHolder provides additional methods for enhancing the interaction between the WTContentHolder and the container in which it exists. For example, while the WTContentHolder bean provides a Save button which, when invoked, causes all content changes to be saved, the WTContentHolder also provides a method to programmatically save any content changes. Consider the Report example. If the user clicks on the Cancel button in the UpdateReportFrame, and changes to the content have been made that have not yet been saved, you may want to prompt the user to save these changes:

```

public class UpdateReportFrame extends java.awt.Frame {

    ...

    void cancelButton_Action(java.awt.event.ActionEvent event) {

        // Check for unsaved content changes
        if( myWTContentHolder.isDirty() ) {

            if( promptUserToSaveChanges() ) {
                try {
                    myWTContentHolder.persistContentChanges();
                } catch (PropertyVetoException pve) {
                    pve.printStackTrace();

                } catch (WTEException wte) {
                    wte.printStackTrace();
                }
            }
        }

        setVisible( false );
    }

    ...

} // End class 'UpdateReportFrame'

```

For more details on using WtContentHolder, see the javadoc for the `wt.clients.beans.contentholder` package.

WTE Explorer Bean

The WTE Explorer bean is an "Explorer" type browser for displaying items and relationships.

The WTE Explorer is a composite object which contains a tree view, a list view, several status bars, and a splitter panel. Users can select nodes in the tree view and display information about the node in the list view. Tree nodes can be expanded to show structures.

Nodes can be created and added to the tree with objects which implement the `wt.clients.beans.explorer.Explorables` interface. A sample adapter class, `wt.clients.beans.explorer.WTBusinessObject`, is provided which implements the `Explorables` interface and can be easily subclassed. The WTE Explorer invokes the `getUses()` method on the contained object to display its children in the tree view. It invokes the `getContents()` method on the object to display its contents in the list view.

API

The WTE Explorer bean has a large number of attributes and methods. Methods are provided to perform the following functions:

- Setting the foreground and background colors.
- Setting the preferred fonts.
- Adding, deleting, and refreshing nodes in the tree.
- Clearing the WTE Explorer of all objects.
- Setting the column headings for the list view.
- Setting the methods to invoke on the contained objects.
- Setting the column alignments and sizes.
- Controlling caching of detail information for a node.
- Setting the text displayed in the status bars.

For more details on using WTE Explorer, see the javadoc.

Sample Code

The following code sample shows an instance of the WTE Explorer bean being created and added to a component. The instance of the WTE Explorer is initialized with the specified font and color preferences, the desired column headings, the methods to invoke on the contained objects, and the desired toolbar buttons.

The container is added as a listener for WTE Explorer events so it is notified when the user selects nodes in the tree or clicks on a toolbar button. When the container receives an event from the WTE Explorer, it invokes an appropriate method to act on the selected object in the WTE Explorer.

```
import wt.clients.beans.explorer.WTE Explorer;
import wt.clients.beans.explorer.WTE ExplorerEvent;
import wt.clients.beans.explorer.WTE ExplorerListener;
import wt.clients.beans.explorer.WTNode;
import wt.clients.beans.explorer.Explorable;

public class PartExplorer extends Container implements
    WTE ExplorerListener
{
    // instance variable for the WTE Explorer
    protected wt.clients.beans.explorer.WTE Explorer myExplorer;

    // Create a default PartExplorer
    public PartExplorer()
    {

        GridBagLayout gridBagLayout;
        gridBagLayout = new GridBagLayout();
        setLayout(gridBagLayout);
        setSize(730,423);
        myExplorer = new wt.clients.beans.explorer.WTE Explorer();

        // Set the column headings in the list view.
        try {
            java.lang.String[] tempString = new java.lang.String[8];
            tempString[0] = new java.lang.String("Number");
            tempString[1] = new java.lang.String("Name");
            tempString[2] = new java.lang.String("Version");
            tempString[3] = new java.lang.String("View");
            tempString[4] = new java.lang.String("Qty");
            tempString[5] = new java.lang.String("Units");
            tempString[6] = new java.lang.String("Type");
            tempString[7] = new java.lang.String("State");
            myExplorer.setListHeadings(tempString);
        }
        catch(java.beans.PropertyVetoException e) { }
        myExplorer.setDisplayUsesAsContents(true);

        // Set the fonts to use in the Explorer

        myExplorer.setTreeFont(new java.awt.Font("Dialog",
            java.awt.Font.PLAIN,11));
        try {
            myExplorer.setListHeadingFont(new java.awt.Font("Dialog",
                java.awt.Font.PLAIN,11));
        }
        catch(java.beans.PropertyVetoException e) { }

        // Set the column alignment in the list view.

        try {
            java.lang.String[] tempString = new java.lang.String[8];
            tempString[0] = new java.lang.String("Left");
```

```

        tempString[1] = new java.lang.String("Left");
        tempString[2] = new java.lang.String("Left");
        tempString[3] = new java.lang.String("Left");
        tempString[4] = new java.lang.String("Right");
        tempString[5] = new java.lang.String("Left");
        tempString[6] = new java.lang.String("Left");
        tempString[7] = new java.lang.String("Left");
        myExplorer.setListColumnAlignments(tempString);
    }
catch(java.beans.PropertyVetoException e) { }

// Set the methods to invoke for each column.
try {
    java.lang.String[] tempString = new java.lang.String[8];
    tempString[0] = new java.lang.String("getNumber");
    tempString[1] = new java.lang.String("getName");
    tempString[2] = new java.lang.String("getVersion");
    tempString[3] = new java.lang.String("getViewName");
    tempString[4] = new java.lang.String("getQuantity");
    tempString[5] = new java.lang.String("getUnitName");
    tempString[6] = new java.lang.String("getType");
    tempString[7] = new java.lang.String("getState");
    myExplorer.setListMethods(tempString);
}
catch(java.beans.PropertyVetoException e) { }
myExplorer.setRootNodeText("Product Structure");

// Set the icons for the toolbar
{
    java.lang.String[] tempString = new java.lang.String[20];
    tempString[0] = new java.lang.String("new");
    tempString[1] = new java.lang.String("update");
    tempString[2] = new java.lang.String("view");
    tempString[3] = new java.lang.String("delete");
    tempString[4] = new java.lang.String("Spacer");
    tempString[5] = new java.lang.String("revise");
    tempString[6] = new java.lang.String("syscfg");
    tempString[7] = new java.lang.String("Spacer");
    tempString[8] = new java.lang.String("refresh");
    tempString[9] = new java.lang.String("clear");
    tempString[10] = new java.lang.String("Spacer");
    tempString[11] = new java.lang.String("localsearch");
    tempString[12] = new java.lang.String("enterprisesearch");
    tempString[13] = new java.lang.String("Spacer");
    tempString[14] = new java.lang.String("wexplr");
    tempString[15] = new java.lang.String("Spacer");
    tempString[16] = new java.lang.String("help");
    tempString[17] = new java.lang.String("Spacer");
    tempString[18] = new java.lang.String("Spacer");
    tempString[19] = new java.lang.String("Spacer");

    myExplorer.setTools(tempString);
}

// Set the column sizes for the list view.
try {
    java.lang.String[] tempString = new java.lang.String[8];
    tempString[0] = new java.lang.String("90");

```

```

        tempString[1] = new java.lang.String("150");
        tempString[2] = new java.lang.String("50");
        tempString[3] = new java.lang.String("40");
        tempString[4] = new java.lang.String("40");
        tempString[5] = new java.lang.String("40");
        tempString[6] = new java.lang.String("60");
        tempString[7] = new java.lang.String("60");
        myExplorer.setListColumnSizes(tempString);
    }
    catch(java.beans.PropertyVetoException e) { }

    // Set some more properties for the explorer.

    try {
        myExplorer.setTreeStatusBarText("All Parts");
    }
    catch(java.beans.PropertyVetoException e) { }
    try {
        myExplorer.setListCellFont(new java.awt.Font("Dialog",
        java.awt.Font.PLAIN,11));
    }
    catch(java.beans.PropertyVetoException e) { }
    try {
        myExplorer.setListFont(new java.awt.Font("Dialog",
        java.awt.Font.PLAIN,11));
    }
    catch(java.beans.PropertyVetoException e) { }
    myExplorer.setBounds(0,0,730,423);
    myExplorer.setForeground(java.awt.Color.black);
    myExplorer.setBackground(java.awt.Color.lightGray);

    // Add the explorer instance to the container.
    add(myExplorer);
    setBackground(java.awt.Color.lightGray);

    // --- Callbacks
    myExplorer.addListener(this);
    // add as listener to refresh service

    addRefreshListener();

    }

/**
 * Handle events from the WTEexplorer.
 *
 * @param e the WTEexplorerEvent
 */

public void explorerEvent(WTEexplorerEvent e)
{
    if( e.getType().equals(WTEexplorerEvent.COMMAND ) )
    {
        processCommand(e);
    }
    else if( e.getType().equals( WTEexplorerEvent.OPEN ) )
    {
        handleDoubleClickEvent( e );
    }
}

```

```

}
}

/**
 * Handle double-click events from the WTE Explorer. The
 * "view" task for the selected object will be launched.
 *
 * @param e the WTE ExplorerEvent
 */

protected void handleDoubleClickEvent( WTE ExplorerEvent e )
{
    getParentFrame().setCursor(Cursor.getPredefinedCursor(
        Cursor.WAIT_CURSOR));
    myExplorer.setCursor(Cursor.getPredefinedCursor(
        Cursor.WAIT_CURSOR));
    processViewCommand();
    getParentFrame().setCursor(Cursor.getDefaultCursor());
    myExplorer.setCursor(Cursor.getDefaultCursor());
}

/**
 * Process the command from the WTE Explorer. The WTE Explorer
 * will send commands when the user selects a toolbar
 * button. The command will equal the name of the
 * image on the toolbar button.
 *
 * @param e the WTE ExplorerEvent
 */

protected void processCommand( WTE ExplorerEvent e )
{
    getParentFrame().setCursor(Cursor.getPredefinedCursor(
        Cursor.WAIT_CURSOR));

    if ( e.getCommand().equals("update") )
    {
        processEditCommand();
    }
    else if ( e.getCommand().equals("view") )
    {
        processViewCommand();
    }
}

/**
 * Process the "Edit" command.
 *
 */

public void processEditCommand()
{
    Object selected_obj = getSelectedObject();
    if (selected_obj != null )
    {
        if ( isUpdateAllowed((Part)selected_obj))
        {

```

```

TaskDelegate delegate =
    TaskDelegateFactory.instantiateTaskDelegate(
        selected_obj );
if( delegate != null )
{
    delegate.setParentApplet( getApplet() );
    delegate.setParentFrame( getParentFrame() );
    delegate.setObject( selected_obj );
    try
    {
        delegate.launchUpdateTask();
    }
    catch (TaskDelegateException tde)
    {
        System.out.println(tde);
    }
    catch (WTEException wte)
    {
        System.out.println(wte);
    }
}
else
{
    System.out.println("Could not instantiate Task
        Delegate for " + selected_obj);
}
}
}

/**
 * Process the "View" command.
 *
 */
public void processViewCommand()
{
    try
    {
        Object selected_obj = getSelectedObject();

        if (selected_obj != null )
        {
            TaskDelegate delegate =
                TaskDelegateFactory.instantiateTaskDelegate(
                    selected_obj );
            if( delegate != null )
            {
                delegate.setParentApplet( getApplet() );
                delegate.setParentFrame( getParentFrame() );
                delegate.setObject( selected_obj );
                delegate.launchViewTask();
            }
        }
        else
        {
            System.out.println("Could not instantiate Task
                Delegate for " + selected_obj);
        }
    }
}

```

```

        }
    }
}
catch (TaskDelegateException tde)
{
    System.out.println(tde.getMessage());
}
catch (WTEException wte)
{
    System.out.println(wte);
}
}
}

/**
 * Get the object currently selected in the explorer.
 * If a list object is selected, use that object.
 * Otherwise use the selected tree node.
 *
 * @return the selected object or null if no object is
 * selected
 */

protected Object getSelectedObject()
{
    Explorable busobj = null;
    Object obj = null;
    WTNode node = null;

    // If a list object is selected, use that object.
    // Otherwise, use the selected tree node.

    busobj = myExplorer.getSelectedDetail();
    if (busobj == null)
    {
        node = myExplorer.getSelectedNode();
        if ( node != null)
        {
            busobj = node.getObj();
        }
    }

    if (busobj == null )
    {
        return null;
    }
    return busobj.getObject();
}

public static void main(String[] args)
{
    Frame f = new Frame("PartExplorer test");

    PartExplorer exp = new PartExplorer();
    f.setSize(700,700);
    f.setLayout(new BorderLayout());
    f.add("Center",exp);
    f.show();
}
}

```

```
}
```

PartAttributesPanel Bean

Overview

PartAttributesPanel is a Java Bean component for manipulating `wt.clients.prodmgmt.PartItem` objects. It is used in the Create, Update, and View Part windows in the Product Information Manager client, and can be used by developers who are customizing the Product Information Management applications. The source code for the bean is contained in the `wt.clients.prodmgmt.PartAttributesPanel.java` file.

The PartAttributesPanel bean contains properties to specify the class of the object being manipulated and the attributes to be displayed. A label, maximum length, and edit/view can be specified for each attribute.

The PartAttributesPanel bean dynamically constructs a user interface based on the contained class and the specified information about the attributes. Boolean attributes are represented as check boxes, enumerated types as choice lists, and string and integer values are shown in text fields. The bean uses a grid-bag layout to display the attributes in a tabular format. Labels will be displayed to the left of each attribute. The layout has two attributes on each row. The labels have a grid width of 1 and attribute elements normally have a grid width of 3 (TextAreas and other large components will have a larger area). The attributes will be displayed in the order they were specified in the attributes property for the bean.

After you modify values using the constructed user interface, the program can invoke a method on the instance of the PartAttributesPanel bean to transfer the modified values to the contained object.

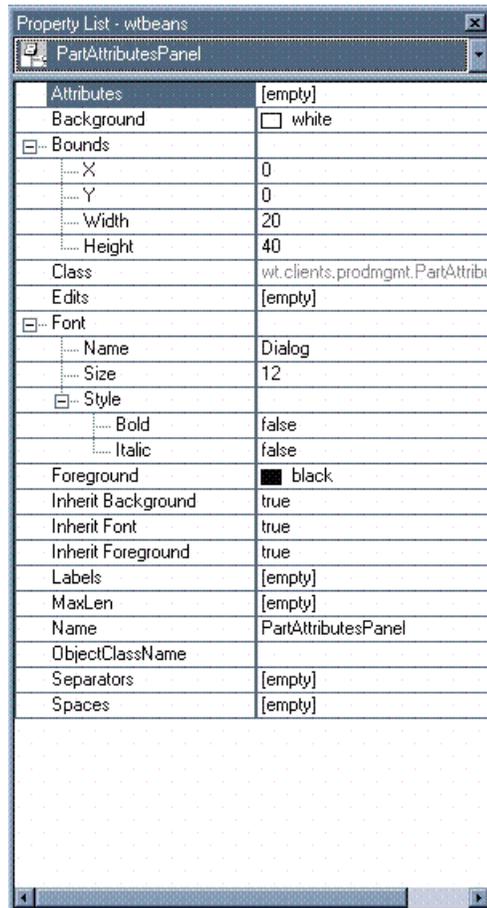
API

The PartAttributesPanel bean contains methods to specify the following items:

- The class name of the object being manipulated.
- The attributes to display.
- The labels for each attribute.
- Whether an attribute is editable or view only.
- The maximum length for each attribute. (This property is currently used only to determine if a string should be displayed in a TextArea instead of a TextField element.)
- Where to position separators (horizontal lines) on the form.
- Where to position blank spaces on the form.

For more details on using the PartAttributesPanel bean, see the Javadoc for the wt.clients.prodmgmt package.

Many properties of the PartAttributesPanel bean can be configured at development time. The following figure shows the Property List displayed in Visual Cafe for the PartAttributesPanel.



Attributes	[empty]
Background	<input type="checkbox"/> white
Bounds	
X	0
Y	0
Width	20
Height	40
Class	wt.clients.prodmgmt.PartAttribu
Edits	[empty]
Font	
Name	Dialog
Size	12
Style	
Bold	false
Italic	false
Foreground	<input checked="" type="checkbox"/> black
Inherit Background	true
Inherit Font	true
Inherit Foreground	true
Labels	[empty]
MaxLen	[empty]
Name	PartAttributesPanel
ObjectClassName	
Separators	[empty]
Spaces	[empty]

Property List for PartAttributesPanel

Initially, only a blank panel is displayed in the Visual Cafe environment. To construct the user interface, the bean must have valid values for the following properties:

ObjectClassName

The fully qualified class name of the object (such as, wt.clients.prodmgmt.PartItem)

Attributes

An array of String values specifying the attributes to display (such as, Name, Number, and so on).

Labels

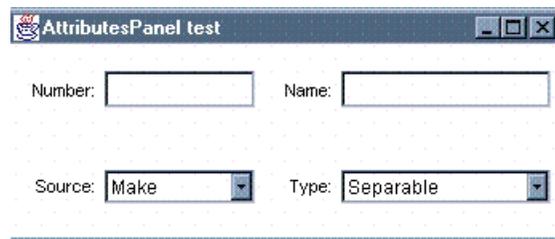
An array of String values specifying the labels for the attributes (such as, Name:, Number:, and so on).

Edits

An array of String values specifying whether an attribute is view only (false) or editable (true).

The number of values specified for the Attributes, Labels, and Edits properties must be the same for each property. If you specify four Attributes, you must also specify four Labels and four Edits.

Once the preceding properties for the bean have been specified, the panel should display a form with the specified attributes in the Visual Cafe environment. The following figure shows what should appear in the form designer:



Form Designer

Sample Code

The following code demonstrates a possible use of this class:

```
Frame f = new Frame("AttributesPanel test");
PartAttributesPanel attributeBean = new PartAttributesPanel();
f.setSize(700,600);
f.setLayout(new BorderLayout());
try
{
    // Set the class name of the object to be manipulated
    attributeBean.setObjectClassName("wt.clients.prodmgmt.PartItem");
    // Set the attributes to display in the panel
    {
        java.lang.String[] tempString = new java.lang.String[4];
        tempString[0] = new java.lang.String("Number");
        tempString[1] = new java.lang.String("Name");
        tempString[2] = new java.lang.String("Source");
        tempString[3] = new java.lang.String("Type");
        attributeBean.setAttributes(tempString);
    }
    // Set the labels for the attributes
    {
        java.lang.String[] tempString = new java.lang.String[4];
        tempString[0] = new java.lang.String("Number:");
        tempString[1] = new java.lang.String("Name:");
        tempString[2] = new java.lang.String("Source:");
        tempString[3] = new java.lang.String("Type:");
```

```

        attributeBean.setLabels(tempString);
    }
    // Make all the attributes editable
    {
        java.lang.String[] tempString = new java.lang.String[4];
        tempString[0] = "true";
        tempString[1] = "true";
        tempString[2] = "true";
        tempString[3] = "true";
        attributeBean.setEdits(tempString);
    }
}
catch ( WTPROPERTYVETOException wte)
{
    wte.printStackTrace();
}
// Add the Bean to the panel
f.add("Center",attributeBean);
f.pack();
f.show();

```

WTQuery Bean

WTQuery provides a tool that allows you to include in your application the ability to search for objects in the local Windchill database. WTQuery is comprised of an optional title, a Tab panel containing an input area for search criteria, three required buttons, three optional buttons, a check box indicating whether to append to or replace the search results, and a list area for displaying search results.

The search results list area presents the results of the database query. The user can clear this area by pressing the Clear button. A check box is available to the user that determines whether to clear the list area between invocations of pressing the Find button. By default, the list area allows multiple objects to be selected. To restrict the list area to allow only single selection, use the `setMultipleMode()` method. To obtain the list of selected objects, use either the `getSelectedDetails()` or `getSelectedDetail()` method, depending on whether multiple or single selection mode is in use.

The buttons are displayed in two columns. The required column contains the Find, Stop, and Clear buttons. The optional column contains the OK, Close, and Help buttons. By default, you get all six buttons. The functionality for the required buttons and the Help button is handled by WTQuery (described later in this section). The calling application can register a listener to be notified when the OK or Close buttons are pressed. To register a listener, create a `WTQueryListener` object and call the `addListener()` method using the `WTQueryListener` object as input.

When creating a WTQuery object, you can specify a search filter string that is passed to the method server when performing the database query. The search filter is used to restrict the query. For more information about the possible values for the search filter string, see the javadoc for `wt.query.SearchTask`.

A WTQuery object provides the ability to search for objects of a specified class. A WTSchema object is used to configure what attributes and classes are available to WTQuery. When calling WTQuery, you must specify a WTSchema object.

A WTSchema object stores the class name and attributes used by a WTQuery object. You must create this object with a properly formatted string that contains a fully qualified class name as well as attributes that will be displayed in the Tab panel and results list area of the WTQuery object.

The format of the string used to create a WTSchema object is broken up into records. A one-character code followed by a colon begins each record; a semi-colon and space combination are used as a separator between records. The one-character codes are C, G, A and D:

- C** Specifies a fully qualified class name to query on. This must be the first record in the format string.
- G** Specifies text for the search criteria tabs. All attributes that follow will be grouped on that tab until the next G record.
- A** Specifies an attribute used in both the search criteria Tab panel and the results list area. Attributes are displayed in the order they appear in the string.
- D** Specifies an attribute used in only the results list area.

The following string is an example:

```
"C:wt.doc.WTDocument; G:Search Criteria; A:name;
D:versionIdentity; G:More Search Criteria; A:description;"
```

This string designates wt.doc.WTDocument as the class that will be queried. The tab panel will contain two tabs labeled Search Criteria and More Search Criteria. The Search Criteria tab will contain an input field for name. The More Search Criteria tab will contain an input field for description. The results list area will be set up with columns for name, versionIdentity, and description.

The following example shows how to add WTQuery to a panel:

```
Panel queryPanel;

try {
    WTQuery myQuery = new WTQuery("My Query Panel",
        SearchTask.ALL_VERSIONS, false);

    myQuery.setSchema(new WTSchema("C:wt.doc.WTDocument; G:Search
        Criteria; A:name; D:versionIdentifier; G:More Search
        Criteria; A:description;"));
    myQuery.setMultipleMode(false);
    myQuery.setDialogButtonsVisible(false);
    this.setLayout(new BorderLayout());
    this.add("Center", myQuery);
    myQuery.addListener(new WTQueryListener() {
```

```

        public void queryEvent(WTQueryEvent e) {
            if (e.getType().equals(WTQueryEvent.STATUS)) {
                String s = e.getStatus();
                System.out.println("Status change: " + s);
            }
        }
    });
} catch (Exception e) {}

```

WTChooser bean

WTChooser is a subclass of WTQuery. WTChooser provides a tool that allows you to include in your application the ability to search for objects in the local Windchill database.

The WTChooser bean is not currently included in the wtbeans.jar file but the API for incorporating WTChooser into an application is available.

WTChooser is comprised of an optional title, a tab panel containing an input area for search criteria, six buttons, a check box indicating whether to append to or replace the search results, and a list area for displaying search results.

The WTChooser object provides the ability to search for objects of a specified class. The ChooserOptions class is distributed as source code and is used to configure the attributes and classes available to WTChooser.

The ChooserOptions object consists of strings that associate a fully-qualified class name with attributes to be used for restricting the query and attributes to be displayed in the results list area. When calling WTChooser, you must specify a fully-qualified class name that is defined appropriately in the ChooserOptions object.

The format string is broken up into records. A one-character code followed by a colon begins each record; a semi-colon and space combination are used as a separator between records. The one-character codes are C, G, A and D:

C

Specifies a fully-qualified class name to query on. This must be the first record in the format string.

G

Specifies text for the search criteria tabs. All attributes that follow will be grouped on that tab until the next G record.

A

Specifies an attribute used in both the search criteria Tab panel and the results list area. Attributes are displayed in the order they appear in the string.

D

Specifies an attribute used in only the results list area.

The search results list area presents the results of the database query. The user can clear this area by pressing the Clear button. A check box is available to the user that determines whether to clear the list area between invocations of pressing the

Find button. By default, the list area allows multiple objects to be selected. To restrict the list area to allow only a single selection, use the `setMultipleMode()` method. To obtain the list of selected objects, use either the `getSelectedDetails()` or `getSelectedDetail()` method, depending on whether multiple or single selection mode is in use.

The buttons are displayed in two columns. One column contains the Find, Stop, and Clear buttons. The other contains the OK, Close, and Help buttons. For the most part, the functionality for these buttons is handled by `WTChooser`. The calling application can register a listener to be notified when the OK or Close buttons are pressed. To register a listener, create a `WTQueryListener` object and call the `addListener()` method using the `WTQueryListener` object as input.

The `WTChooser` panel can either be embedded in an existing frame or you can instruct `WTChooser` to create a dialog window. If a dialog window is used, you must specify a `java.awt.Component` as the parent. The dialog will be centered over the parent.

The following example shows how to create a `WTChooser` in a dialog:

```
Frame frame = getParentFrame();

WTChooser chooser = new WTChooser("wt.part.WTPartMaster",
    "Find Part", frame);

chooser.addListener ( new WTQueryListener() {
    public void queryEvent(WTQueryEvent e)
    {
        if (e.getType().equals(WTQueryEvent.COMMAND ))
        {
            if (e.getCommand().equals(WTQuery.OkCMD) )
            {
```

EnumeratedChoice Bean

Overview

The `EnumeratedChoice` bean is a simple extension of the `java.awt.Choice` class. It allows you to specify a subclass of `wt.fc.EnumeratedType`. The display values of the specified `EnumeratedType` subclass are used to populate the list of selectable values.

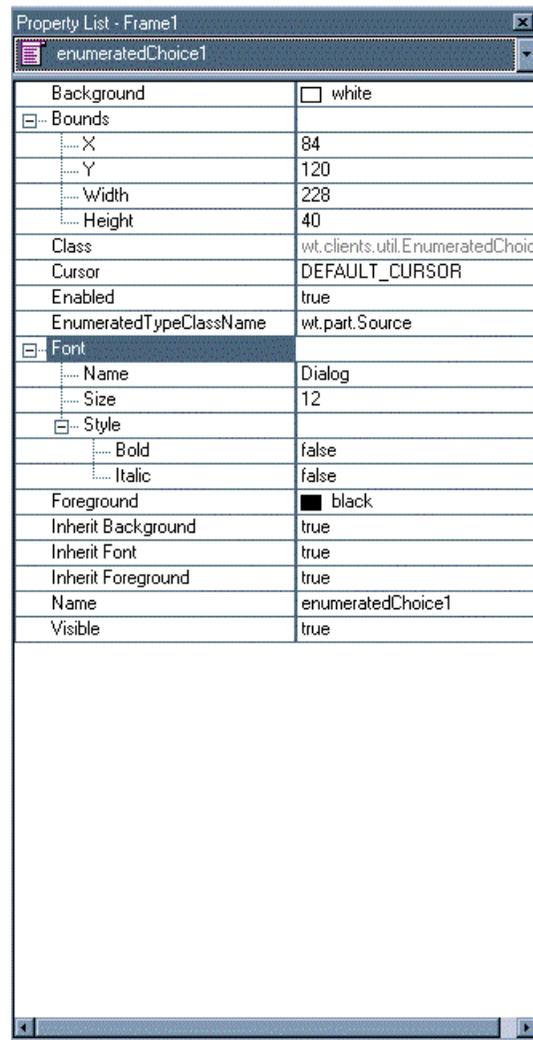
API

The `EnumeratedChoice` bean contains methods to perform the following operations:

- Specify the class name of the `EnumeratedType` object being manipulated.
- Set the selected `EnumeratedType` value in the list.
- Get the selected `EnumeratedType` value from the list.
- Allow the display of an empty (blank) choice.

For more details on using the EnumeratedChoice bean, see the Javadoc for the wt.clients.util package.

Many properties of the EnumeratedChoice bean can be configured at development time. The following figure shows the Property List displayed in Visual Cafe for the EnumeratedChoice bean.



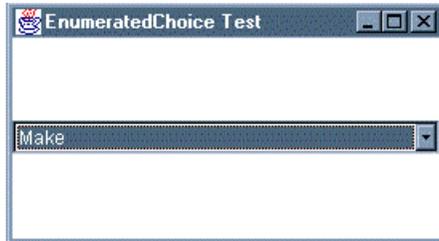
Property List for EnumeratedChoice Bean

Initially, only an empty Choice will be displayed in the Visual Cafe environment. To populate the list of possible values, the bean must have valid values for the following property:

EnumeratedTypeClassName

The fully qualified class name of the subclass of wt.fc.EnumeratedType (such as, wt.part.Source).

Once the preceding property for the bean has been specified, the panel should display a form with the specified attributes in the Visual Cafe environment. The following figure shows what should appear in the form designer for an EnumeratedChoice with the EnumeratedTypeClassName initialized to wt.part.source.



Form Designer for EnumeratedChoice Bean

Sample Code

The following code demonstrates a possible use of this class:

```
static public void main(String args[])
{
    if (args.length == 0 )
    {
        System.out.println("supply enumeration class");
        return;
    }
    String classname = args[0];
    class DriverFrame extends java.awt.Frame
    {
        EnumeratedChoice choice = null;
        public DriverFrame(String title)
        {
            super(title);
            addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent event)
            {
                dispose(); // free the system resources
                System.exit(0); // close the application
            }
            });
            setLayout(new java.awt.BorderLayout());
            setSize(300,300);
            choice = new EnumeratedChoice();
            add(choice, "Center");

            choice.addItemListener( new java.awt.event.ItemListener() {
                public void itemStateChanged(ItemEvent ie)
                {
                    EnumeratedType value = choice.getSelectedEnumeratedType();
                    System.out.println("Selected value is now " + value);
                }
            });
        }
    }
    public void setClass(String name)
```

```

{
  try {
    choice.setEnumeratedTypeClassName(name);
  }
  catch(PropertyVetoException pve)
  {
    pve.printStackTrace();
  }
}
}
DriverFrame df;
df = new DriverFrame("EnumeratedChoice Test");
df.setClass(classname);
df.show();
}

```

Life Cycle Beans

Three Windchill beans are supplied for manipulating `LifeCycleManaged` objects: `LifeCyclePanel`, `ProjectPanel`, and `LifeCycleInfo`. Each bean has a create and view mode.

The create mode of `LifeCyclePanel` is comprised of a life cycle label, choice box, and browse button for finding and selecting a life cycle template. The view mode of `LifeCyclePanel` consists of a life cycle label, current life cycle template name, life cycle state label, and current life cycle state value.

The create mode of `ProjectPanel` is comprised of a project label, choice box, and browse button for finding and selecting a project. The view mode of `ProjectPanel` consists of a project label and the current project value.

`LifeCycleInfo` combines `LifeCyclePanel` and `ProjectPanel` into one bean. The create mode is comprised of a life cycle label, choice box, and browse button for finding and selecting a life cycle template, and a project label, choice box, and browse button for finding and selecting a project. The view mode of `LifeCycleInfo` consists of a life cycle label, current life cycle template name, life cycle state label, and current life cycle state value, project label, and current project value.

You can use the `LifeCycleInfo` panel to view an existing life cycle template and project associated with a `LifeCycleManaged` object or to assign a life cycle template and a project to a `LifeCycleManaged` object. In create mode, the `LifeCycleInfo` object will populate a choice menu of candidate life cycle templates and a choice menu of candidate projects for the user to select from. The template candidates are selected based on the type of object being created. The project candidates are projects residing in the same domain as the object is being created in, plus any projects defined in the system domain.

If an existing `LifeCycleManaged` object has been set in the `LifeCycleInfo` object, the values from the `LifeCycleManaged` object for both the life cycle template and project are included in the appropriate choice menus and selected as the current

menu item. The user can browse the system to select other templates or projects if desired.

The `LifeCycleInfo` panel could be used as shown in the following example:

```
import wt.clients.lifecycle.beans.LifeCycleInfo
.
. // Create a dialog or frame for adding a lifecycle info panel
.

// Create the LifeCycleInfo object and add to the dialog
LifeCycleInfo myLifeCycleInfo = new LifeCycleInfo();
myDialog.add( myLifeCycleInfo );

// Set the mode to create
lifeCycleInfo1.setMode( LifeCycleInfo.CREATE_MODE );

// Set the class of the object being created
lifeCycleInfo1.setClass( wt.doc.WTDocument.class );

myDialog.show();
.
. //create a LifeCycleManaged object
.
//myObject is the lifecycle managed object being created
myLifeCycleInfo.assign( myObject );
// Optionally at this point you could switch to view mode
myLifeCycleInfo.setMode( LifeCycleInfo.VIEW_MODE );
```

For more information, see the javadoc in `wt.clients.beans.lifecycle.LifeCyclePanel`, `wt.clients.beans.lifecycle.LifeCycleInfo`, and `wt.clients.beans.lifecycle.ProjectPanel`.

Spinner bean

The `Spinner` bean (`wt.clients.prodmgmt.Spinner`) is a subclass of the `symantec.itools.awt.util.spinner.Spinner` class. It adds a new method, `addKeyListener()`, to enable listeners to detect if users have modified the value in the text field associated with the spinner. It also adds a new property, `CurrentDoubleValue`, to allow the spinner to handle double values instead of integer values.

For more information, see the javadoc in the `symantec.itools.awt.util.spinner` and the `wt.clients.prodmgmt` packages.

WTMultiList bean

The `WTMultiList` bean (`wt.clients.util.WTMultiList`) is a subclass of the `symantec.itools.awt.MultiList` class. It overrides several methods to improve the display appearance of the bean on UNIX clients.

`WTMultiList` can also be configured to display attributes of modeled Windchill business objects, such as `Parts`, `Documents`, and so on in a multicolumn list. When taking advantage of this feature of `WTMultiList`, it is necessary to specify the

class and attributes that will be used by the display methods. A `wt.clients.beans.query.WTSchema` object is used to specify the class and attributes used by `WTMultiList` for displaying Windchill business objects. The attributes defined by the `WTSchema` object are used to obtain the column headings and values for the cells within a column. Methods that allow adding/obtaining modeled Windchill business objects are included in this class. An icon representing the object associated with a particular row is always displayed in the first column.

An example of using the `WTMultiList` to display modeled Windchill business objects is presenting the results of a database search to the user. After performing the database search, the objects obtained can be added to a `WTMultiList` object for selection by the user. `WTMultiList` can be configured in single-select or multiselect mode.

The user cannot type or edit a selection in a `WTMultiList`. The user can resize a column at run time by dragging the column boundary to a new position.

The following code demonstrates creating a `WTMultiList` in a frame.

```
Frame f = new Frame();
f.setSize(400,200);
WTMultiList multi_list = new WTMultiList();

// Create a schema to display Name, Version and Description of
// objects with class wt.doc.WTDocument
WTSchema schema = new WTSchema("C:wt.doc.WTDocument; D:name;
    D:versionIdentifier; D:description;");
multi_list.setSchema(schema);
f.add(multi_list);
f.show();

// Add objects .....
```

For more information, see the javadoc in the `symantec.itools.awt` and the `wt.clients.util` packages.

AssociationsPanel bean

Overview

The `AssociationsPanel` bean (`wt.clients.beans.AssociationsPanel`) has a dev time component and a runtime component. At dev time, you can drag this component from the component library in Visual Café and drop it onto a container. The properties can also be edited by using Visual Café's Property Editor. At runtime, this class retrieves the links of the object and displays them in a multilist.

The bean has an Edit and View mode. In edit mode, the user can add new relationships, remove existing relationships, and update existing relationships by editing the attributes on the link. In view mode, the user can view the relationships and view the attributes on the link. The bean keeps track of the create, update, and removes operations, and is able to persist the changes when `save()` is invoked.

This bean contains a `WTMultiList` that displays the results of a navigate, add, or remove. The user can select a row in the multilist and press a button to view that object or remove that object. The user can press the Add button to search for a new object to create a link to. Also, when the user selects a row, the attributes on the link of that row are displayed in an `AttributesForm`. If the `AttributesForm` is editable, the user can edit the attributes on the link in those fields.

API

The `AssociationsPanel` bean contains methods to specify the following items at dev time:

- `objectClassName` - The class name of the object being manipulated.
- `role` - The string representing the Role to navigate.
- `otherSideClassName` - The class name of the `OtherSideObject` (a `Persistable`) for viewing and displaying the attributes.
- `otherSideAttributes` - The attributes of the `otherSideObject` to be displayed in the multilist.
- `otherSideSchema` - The developer can use a GUI at dev time to set the `otherSideClassName` and the `otherSideAttributes`.
- `labels` - The labels for the columns of the multilist.
- `multiListLinkAttributes` - The attributes of the link to be displayed in the multilist.
- `linkClassName` - The class name of the `BinaryLink` for creating new links.
- `linkAttributes` - The attributes of the link to be set in the `AttributesForm` bean below the multilist.
- `linkSchema` - The developer can use a GUI at dev time to set the `linkClass` name and the `linkAttributes`.
- `mode` - Edit or View. Edit will have Add and Remove buttons and the attributes form is editable.
- `chooserOptions` - Must be a valid key in the `wt.clients.beans.query.ChooserOptions` resource bundle. The string value associated with the key sets the list of class names that the user can search for upon pressing the Add button.
- `relativeColumnWidths` - Sets the relative widths of the columns in the `WTMultiList`.

The `AssociationsPanel` bean contains the following methods to be used at runtime:

- `setObject(Persistable myObject)` - Sets the object to be manipulated.
- `addHelpListener(helpListener)` - Adds a listener to the bean's help system.

- `setEnabled(boolean myBoolean)` - Sets all fields to editable or not editable.
- `save(Persistable myObject)` - Saves the new links for that object, deletes the removed links, and updates the links that have been modified by the user.
- `getHelpContext()` - Returns the `HelpContext` currently being used.

Following is an example of how to use the bean.

```

associationsPanell = new wt.clients.beans.AssociationsPanel();
associationsPanell.setRole(
    "theChangeable");
associationsPanell.setChooserOptions(wt.clients.beans.query.
    ChooserOptions.CHANGEABLE_SEARCH_LIST);
try {
    java.lang.String[] tempString = new java.lang.String[5];
    tempString[0] = display("nameColumn");
    tempString[1] = display("numberColumn");
    tempString[2] = display(ChangeMgmtRB.TITLE_COLUMN);
    tempString[3] = display(ChangeMgmtRB.PART_TYPE_COLUMN);
    tempString[4] = display(ChangeMgmtRB.SOURCE_COLUMN);
    associationsPanell.setLabels(tempString);
}
catch(java.beans.PropertyVetoException e) { }
try {
    associationsPanell.setLinkClassName("wt.change2.
        RelevantRequestData");
}
catch(wt.util.WTPropertyVetoException e) { }
try {
    associationsPanell.setObjectClassName(
        "wt.change2.WTChangeRequest2");
}
catch(java.lang.ClassNotFoundException e) { }
{
    java.lang.String[] tempString = new java.lang.String[5];
    tempString[0] = new java.lang.String("name");
    tempString[1] = new java.lang.String("number");
    tempString[2] = new java.lang.String("title");
    tempString[3] = new java.lang.String("partType");
    tempString[4] = new java.lang.String("source");
    associationsPanell.setOtherSideAttributes(tempString);
}
try {
    associationsPanell.setOtherSideClassName(
        "wt.change2.Changeable");
}
catch(java.lang.ClassNotFoundException e) { }
associationsPanell.setMode("Edit");
try {
    java.lang.String[] tempString = new java.lang.String[1];
    tempString[0] = new java.lang.String("description");
    associationsPanell.setLinkAttributes(tempString);
}
catch(wt.util.WTPropertyVetoException e) { }
associationsPanell.setBounds(0,0,530,62);
associationsPanell.setBackground(new Color(12632256));

```

```

gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 0;
gbc.weightx = 1.0;
gbc.weighty = 1.0;
gbc.fill = GridBagConstraints.BOTH;
gbc.insets = new Insets(0,0,0,0);
((GridBagLayout)panel3.getLayout()).setConstraints(
    associationsPanell, gbc);
panel3.add(associationsPanell);
associationsPanell.addHelpListener(helpListener);
associationsPanell.setObject(taskLogic.getCurrentChangeRequest());
associationsPanell.setEnabled(true);
associationsPanell.save(taskLogic.getCurrentChangeRequest());
associationsPanell.getHelpContext().stopIt();

```

Using the AssociationsPanel usually requires adding new entries in the wt.clients.beans.query.ChooserOptions resource bundle. The chooserOptions property on the AssociationsPanel must be a valid key in the wt.clients.beans.query.ChooserOptions resource bundle. The string value associated with the key sets the list of class names that the user can search for upon pressing the Add button.

The following wt.clients.beans.query.ChooserOptions resource bundle shows the entries — highlighted in bold — that were added for use by the AssociationsPanel for the preceding example:

```

public class ChooserOptions extends ListResourceBundle{
    public static final String CHANGEABLE_SEARCH_LIST = "1";

    public Object getContents()[][] {
        return contents;
    }

    static final Object[][]contents = {
        {"UsrSCM", "C:wt.org.WTUser; G:Search Criteria; A:name;
        A:fullName; " +
        "A:authenticationName"},
        {"CRSCM", "C:wt.change2.WTChangeRequest2; G:Search Criteria;
        A:number; " +
        "A:name; A:requestType; A:productPriority; " +
        "G:More Search Criteria; A:lifeCycleState;
        A:projectId; " +
        "A:cabinet; A:modifyTimestamp; A:description"},
        {"COSCM", "C:wt.change2.WTChangeOrder2; G:Search Criteria;
        A:number; " +
        "A:name; A:orderType; A:customerNeedDate; " +
        "G:More Search Criteria; A:lifeCycleState;
        A:projectId; " +
        "A:cabinet; A:modifyTimestamp; A:description"},
        {"CASCM", "C:wt.change2.WTChangeActivity2;
        G:Search Criteria; " +
        "A:number; A:name; A:lifeCycleState; A:projectId;
        G:More Search Criteria; " +
        "A:cabinet; A:modifyTimestamp; A:description"},

```

```

{"DocSCM", "C:wt.doc.WTDocument; G:Search Criteria;
  A:number; " +
  "A:name; A:docType; A:versionIdentifier;
  A:lifecycleState; A:projectId;
  G:More Search Criteria; " +
  "A:cabinet; A:format; A:modifyTimestamp; "},

{"DocMasterSCM", "C:wt.doc.WTDocumentMaster;
  G:Search Criteria; " +
  "A:name; A:number;"},

{"CabSCM", "C:wt.folder.CabinetReference;
  G:Search Criteria; A:name"},

{"PartSCM", "C:wt.part.WTPart; G:Search Criteria; A:number;
  A:name; A:view; " +
  "A:versionIdentifier; A:partType; A:source;
  G:More Search Criteria; " +
  "A:lifecycleState; A:projectId; A:cabinet;
  A:modifyTimestamp; "},

{"PartMasterSCM", "C:wt.part.WTPartMaster;
  G:Search Criteria; " +
  "A:name; A:number"},

{"ConfigItemSCM", "C:wt.effectivity.ConfigurationItem;
  G:Search Criteria; " +
  "A:name; A:effectivityType; A:lifecycleName;
  A:lifecycleState; G:More Search Criteria; " +
  "A:lifecycleAtGate; " +
  "A:modifyTimestamp; A:createTimestamp; A:description"},
{CHANGEABLE_SEARCH_LIST, "wt.doc.WTDocument
  wt.part.WTPart"},

{"BaselineSCM", "C:wt.vc.baseline.ManagedBaseline;
  G:Search Criteria; " +
  "A:number; A:name; A:lifecycleState; A:projectId;
  G:More Search Criteria; " +
  "A:cabinet; A:modifyTimestamp; A:description"},

};

```

EffectivityPanel Bean

Overview

The EffectivityPanel bean (wt.clients.beans.EffectivityPanel) is used to create, update, or view effectivity for an EffectivityManageable object. It has dev time and runtime usage. At dev time, you can drag and drop the bean from the Component Library in Visual Cafe onto a frame, applet, or panel. Properties, such as background and mode, can be set at dev time. At run time, the user can view the effectivity for the effectivity manageable object, or set/update the effectivity.

API

The EffectivityPanel bean contains methods to specify the following items at dev time:

- mode - The mode for the bean; either Edit or View.
- background - The color of the background.
- foreground - The color of the foreground.
- font - The font.

The EffectivityPanel bean contains the following methods to be used at run time:

- addHelpListener(helpListener) - Adds a listener to the bean's help system.
- setObject(EffectivityManageable myObject) - Sets the object whose effectivity is to be viewed or modified.
- isDirty() - Tests whether the user has modified any of the fields.
- save() - Saves the created/modified Effectivity object for the EffectivityManageable object.

Following is an example of how to use the EffectivityPanel.

```
effectivityPanell = new wt.clients.beans.EffectivityPanel();
try {
    effectivityPanell.setMode("Edit");
}
catch(java.beans.PropertyVetoException e) { }
effectivityPanell.setBounds(0,0,415,75);
effectivityPanell.setFont(new Font("Dialog", Font.PLAIN, 11));
effectivityPanell.setForeground(new Color(0));
effectivityPanell.setBackground(new Color(12632256));
gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 0;
gbc.gridwidth = 2;
gbc.weightx = 1.0;
gbc.anchor = GridBagConstraints.NORTHWEST;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.insets = new Insets(0,0,0,0);
((GridBagLayout)panell.getLayout()).setConstraints(
    effectivityPanell, gbc);
panell.add(effectivityPanell);
effectivityPanell.addHelpListener(helpListener);
effectivityPanell.setObject(em);
if (effectivityPanell.isDirty()) {
effectivityPanell.save();
}
```

FolderPanel Bean

Overview

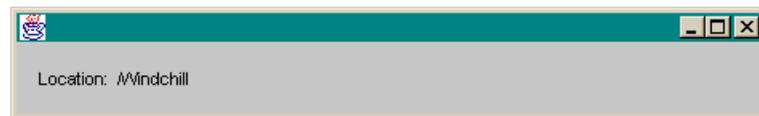
The FolderPanel bean provides support for objects that implement the `wt.folder.FolderEntry` interface. This bean supports assigning the folder of a new object, changing the folder of an existing object, and viewing the folder of an existing object.

When not in view-only mode, the FolderPanel bean contains a label, a text field for entering the location of a folder, and a "Browse..." button that launches the `WTFolderBrowserDialog` to allow the user to browse for a folder.



FolderPanel Bean Not in View-Only Mode

When in view-only mode, the FolderPanel bean contains a label and text that displays the location.



FolderPanel Bean in View-Only Mode

API

The FolderPanel bean has APIs that support the following actions:

- Setting the mode as view-only.
- Setting the list of cabinets from which the user can choose a folder.
- Restricting the list of cabinets from which the user can choose a folder to the user's personal cabinet.
- Restricting the list of cabinets and folders from which the user can choose to those for which the user has a given permission.
- Assigning the folder of a foldered object.
- Changing the folder of a foldered object.
- Viewing the folder location of a foldered object.

For a complete list of the FolderPanel APIs, see the javadoc.

Sample Code

The following example illustrates using the FolderPanel bean in a GUI Frame to create a document.

```
import wt.clients.beans.FolderPanel;
public class CreateDocumentFrame extends java.awt.Frame implements
    java.beans.PropertyChangeListener {

    // Default constructor for CreateDocumentFrame
    public CreateDocumentFrame() {

        ...

        // Code generated by Symantec Visual Cafe when FolderPanel is
        // dragged
        // onto Form Designer and configured using Property Editor.
        // This
        // FolderPanel is configured to only allow the user to choose
        // from
        // folders in his/her personal cabinet.

        folderPanell = new wt.clients.beans.FolderPanel();
        folderPanell.setRequired(true);
        folderPanell.setInPersonalCabinet(true);
        gridBagLayout = new GridBagLayout();
        folderPanell.setLayout(gridBagLayout);
        folderPanell.setBounds(26,151,603,29);
        folderPanell.setFont(new Font("Dialog", Font.PLAIN, 11));
        folderPanell.setForeground(new Color(0));
        folderPanell.setBackground(new Color(12632256));
        gbc = new GridBagConstraints();
        gbc.gridy = 4;
        gbc.gridwidth = 4;
        gbc.anchor = GridBagConstraints.NORTHWEST;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.insets = new Insets(0,26,0,20);
        ((GridBagLayout)getLayout()).setConstraints(folderPanell, gbc);
        add(folderPanell);

        ...

    }

    // Instantiate a new WTDocument and assign it to a
    // Folder.
    public WTDocument createDocument() throws WTEException {

        doc = WTDocument.newWTDocument( numberTextField.getText(),
                                         nameTextField.getText(),
                                         getDocumentType() );

        ...

        folderPanell.setFolderedObject( new_doc );
        new_doc = (WTDocument)folderPanell.assignFolder();
        return new_doc;
    }
}
```

```

        // Initialize the help system used to display online help
        // and status messages. Subscribe as listener to help messages
        // broadcast by FolderPanel
public void initializeHelp() {

    if( helpContext == null ) {
        initializeHelpContext();
    }

    ...

    folderPanel1.addHelpListener( this );
}

// Once the document is created and saved, transition into
// update mode. The location can no longer be changed in
// update mode.
private void setUpdateMode() {
    ...
    folderPanel1.setViewOnly( true );
}

```

AttributesForm Bean

Overview

The AttributesForm is a Java bean component for manipulating modeled Windchill objects that implement the `wt.fc.Persistable` interface.

The AttributesForm bean contains properties to specify the class of the object being manipulated and the attributes to be displayed. Alternatively, the schema property of the bean can be set. The schema property specifies both the object class name and the list of attributes to display for the class. Windchill introspection mechanisms are then used to obtain information about each attribute, including the display name, the maximum length, and whether the attribute is required or updateable.

Setting the schema property in the Visual Café environment at design time launches a custom property editor that allows browsing a list of all possible Windchill modeled classes that implement the `wt.fc.Persistable` interface. The developer can graphically select the class name and the attributes to display in the form.

The AttributesForm dynamically constructs a user interface based on the contained class and the specified information about the attributes. Boolean attributes are represented as checkboxes; Enumerated types are represented as choice lists; strings are shown in text fields or text areas; and integer and decimal values are shown in Spinners. The bean uses a grid-bag layout to display the attributes in a tabular format. Labels will be displayed to the left of each attribute. The layout has two attributes on each row. The labels have a grid width of 1; attribute elements normally have a grid width of 3 (TextAreas and other large components will have a larger area). The attributes will be displayed in the order they were specified in the attributes property for the bean.

After a user modifies values using the constructed user interface, the program can invoke a method on the instance of the `AttributesForm` bean to transfer the modified values to the contained object.

The `AttributesForm` depends on a Factory mechanism to construct the components for each data type. See the `wt.clients.beans.selectors` package for details. The following data types for attributes are currently supported:

- `int`
- `short`
- `long`
- `boolean`
- `Integer`
- `Double`
- `Float`
- `String`
- `java.sql.Timestamp`
- `java.sql.Date`
- `java.util.Date`
- `wt.fc.EnumeratedType`
- `wt.vc.views.ViewReference`
- `wt.project.ProjectReference`
- `wt.lifecycle.LifeCycleState`
- `wt.folder.FolderingInfo`
- `wt.org.WTPrincipalReference`
- `wt.part.Quantity`

API

The `AttributesForm` bean contains methods to specify the following items:

- The class name of the object being manipulated.
- The attributes to display.
- The labels for each attribute.
- Whether an attribute is editable or view only.
- Where to position separators (horizontal lines) on the form.

- Where to position blank spaces on the form.

For more details on using the `AttributesForm`, see the Javadoc for the `wt.clients.beans.AttributesForm` class.

Package Dependencies

The bean is dependent on base classes from the Windchill packages. Ensure that `c:\ptc\Windchill\codebase` is specified in the classpath for Visual Café.

Visual Manipulation

Many properties of the `AttributesForm` can be configured at development time. See the following diagram for the Property List displayed in Visual Café for the `AttributesForm`.

Property List - wtbeans	
AttributesForm	
Attributes	[empty]
Background	<input type="checkbox"/> lightGray
Bounds	
X	0
Y	0
Width	20
Height	40
Class	wt.clients.beans.AttributesForm
Edits	[empty]
Font	
Name	Dialog
Size	11
Style	
Bold	false
Italic	false
Foreground	<input checked="" type="checkbox"/> black
Inherit Background	false
Inherit Font	false
Inherit Foreground	false
Labels	[empty]
MaxLen	[empty]
Mode	VIEW_MODE
Name	AttributesForm
ObjectClassName	
Schema	..
Separators	[empty]
Spaces	[empty]

AttributesForm

Initially, only a blank panel will be displayed in the Visual Cafe environment. To construct the user interface, the bean must have valid values for the following properties:

- `ObjectClassName` - the fully qualified class name of the object (such as, `wt.part.WTPart`)
- `Attributes` - an array of `String` values specifying the attributes to display (such as, `name`, `number`, and so on).

By default, the `AttributesForm` will use Windchill introspection mechanisms to determine the correct display values to use for the labels for each attribute. Required attributes will be displayed with labels containing a preceding asterisk (*). The label values can be overridden by explicitly setting the `Labels` property with an array of `String` values specifying the labels for each attribute (such as, `Name:`, `Number:`, and so on).

By default, the `AttributesForm` shows all attributes as view only. This can be changed by setting either the `Mode` property or the `Edits` property of the bean.

If the `Mode` is set to `CREATE_MODE`, the bean will display editable components for all attributes that have a valid public setter.

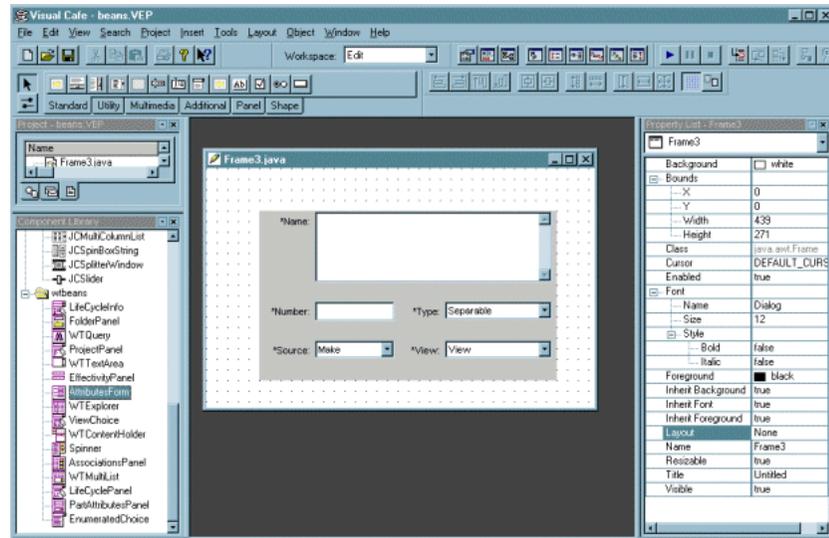
If the `Mode` is set to `UPDATE_MODE`, the bean will display editable components for all attributes that have a valid public setter and for which the attribute is updateable (the attribute can be modified after the object is persisted). If the attribute has a public setter but is modeled to be not updateable, a noneditable component is displayed for the attribute. To model an attribute as nonupdateable, the attribute should have its Windchill `Changeable` property set to `ViaOtherMeans`.

If the `Mode` is set to `VIEW_MODE`, the bean will display non-editable components for all attributes.

The `Edits` property provides more explicit control to individually specify if each attribute is editable or not. It consists of an array of `String` values specifying whether an attribute is view only ("false"), or editable ("true").

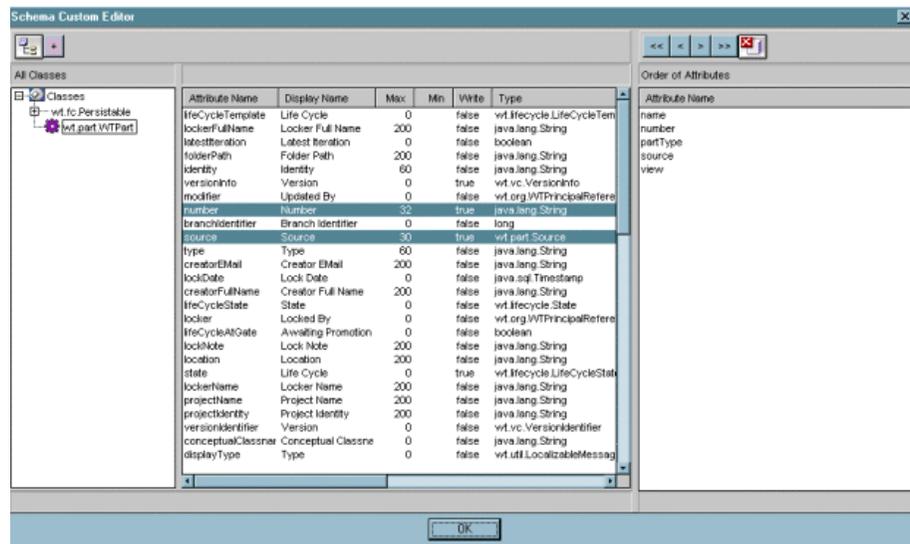
If the `Attributes`, `Labels`, and `Edits` properties all have values, the number of values specified must be the same for each property. For example, if four `Attributes` are specified, four `Labels` and four `Edits` must also be specified.

Once the preceding properties for the bean have been specified, the panel should display a form with the specified attributes in the Visual Cafe. The following diagram shows what should appear in the form designer:



Form Designer for AttributesForm

Clicking on the schema property in Visual Café will launch a custom property editor to allow the developer to visually select the class of the object and the attributes of the class to display in the form:



Custom Property Editor

The Schema Editor is meant to be used as an alternative to explicitly setting the ObjectClassName and the Attributes properties of the bean. In general, to

initialize the `AttributesForm` bean, use either the Schema Editor or set the `ObjectClassName` and the `Attributes` properties; do not use both methods to initialize the same bean instance.

The Schema Editor displays a Class Explorer which shows all the Windchill classes and interfaces that implement the `wt.fc.Persistable` interface. Selecting a class in the treeview on the left-hand side of the Schema Editor will show the attributes for the class in the listview in the middle of the Schema Editor. Selecting attributes by clicking on the listview causes the attributes to appear in the list of selected attributes in the right-hand side of the Schema Editor. Multiple attributes can be selected by using Shift-click or Control-click. Once the selected attributes are shown in the list on the right-hand side, they can be reordered using the toolbar buttons at the top of the Schema Editor. The button labeled "<<" moves an attribute to the beginning of the list. The button labeled "<" moves an attribute up one position in the list. The button labeled ">>" moves an attribute to the end of the list. The button labeled ">" moves an attribute down one position in the list.

Sample Code

The following code demonstrates a possible use of this class:

```
final Frame f = new Frame("AttributesPanel test");
AttributesForm attributeBean = new AttributesForm();
f.setSize(700,600);
f.setLayout(new BorderLayout());
f.addWindowListener(new java.awt.event.WindowAdapter() {
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        f.dispose(); // free the system resources
        System.exit(0); // close the application
    }
});
try
{
    attributeBean.setObjectClassName("wt.part.WTPart");
    {
        java.lang.String[] tempString = new java.lang.String[5];
        tempString[0] = new java.lang.String("number");
        tempString[1] = new java.lang.String("name");
        tempString[2] = new java.lang.String("view");
        tempString[3] = new java.lang.String("partType");
        tempString[4] = new java.lang.String("projectId");

        attributeBean.setAttributes(tempString);
    }
    {
        java.lang.String[] tempString = new java.lang.String[5];
        tempString[0] = "true";
        tempString[1] = "true";
        tempString[2] = "true";
        tempString[3] = "true";
        tempString[4] = "true";
    }
}
```

```

        attributeBean.setEdits(tempString);
    }
}
catch ( WTPPropertyVetoException wte)
{
    wte.printStackTrace();
}
f.add("Center",attributeBean);

try
{
    wt.part.WTPart part = wt.part.WTPart.newWTPart("airplane", "747");
    part.setPartType(wt.part.PartType.toPartType("separable"));
    wt.vc.views.View view = wt.vc.views.ViewHelper.service.getView(
        "Engineering");
    wt.vc.views.ViewHelper.assignToView(part, view);
    attributeBean.setObject(part);
}
catch (wt.util.WTException wte)
{
    wte.printStackTrace();
}
f.pack();
f.show();

```

ViewChoice Bean

Overview

The ViewChoice bean is a simple extension of the java.awt.Choice class. It allows specifying an instance of wt.vc.views.View. The display values of the available Views in the Windchill database are used to populate the list of selectable values.

API

The ViewChoice bean contains methods to perform the following operations:

- Set the selected View value in the list.
- Get the selected View value from the list.
- Allow the display of an empty (blank) choice.

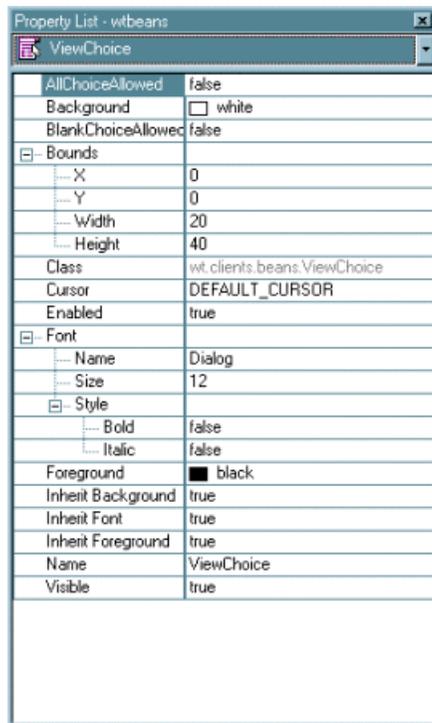
For more details on using the ViewChoice bean, see the Javadoc for the wt.clients.beans package.

Package Dependencies

The bean is dependent upon base classes from the Windchill packages. Ensure that c:\ptc\Windchill\codebase is specified in the classpath for Visual Café.

Visual Manipulation

Several properties of the ViewChoice bean can be configured at development time. See the following diagram for the Property List for the ViewChoice bean displayed in Visual Café.



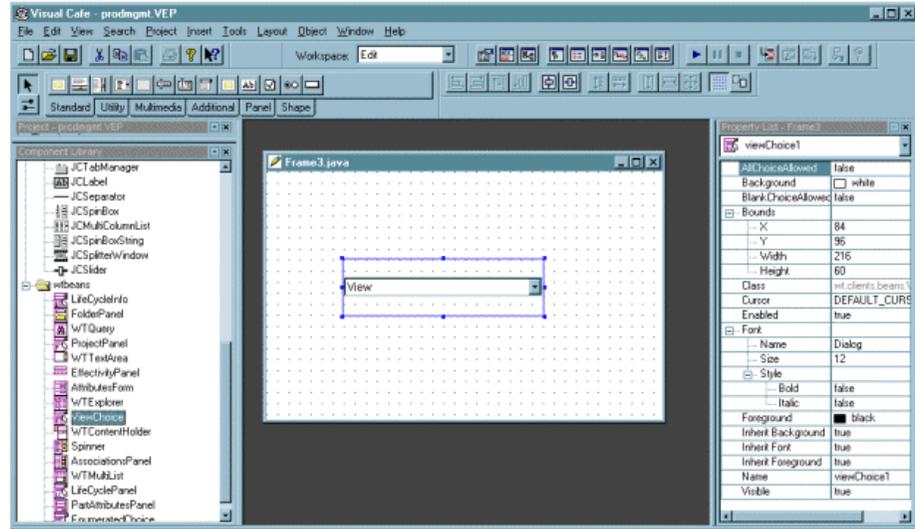
The screenshot shows a window titled "Property List - wtbeans" with a dropdown menu set to "ViewChoice". The table below lists the properties and their values for this bean.

AllChoiceAllowed	false
Background	<input type="checkbox"/> white
BlankChoiceAllowed	false
Bounds	
X	0
Y	0
Width	20
Height	40
Class	wt.clients.beans.ViewChoice
Cursor	DEFAULT_CURSOR
Enabled	true
Font	
Name	Dialog
Size	12
Style	
Bold	false
Italic	false
Foreground	<input checked="" type="checkbox"/> black
Inherit Background	true
Inherit Font	true
Inherit Foreground	true
Name	ViewChoice
Visible	true

ViewChoice

At design time, the Choice will show with a single possible value, "View" in the Visual Café development environment. At runtime, the choice will be populated

with the actual values for all the Views stored in the database. The following diagram shows what should appear in the form designer for a ViewChoice:



Form Designer for ViewChoice

Sample Code

The following code demonstrates a possible use of this class:

```
static public void main(String args[])
{
    class DriverFrame extends java.awt.Frame
    {
        ViewChoice choice = null;
        public DriverFrame(String title)
        {
            super(title);
            addWindowListener(new java.awt.event.WindowAdapter() {
                public void windowClosing(java.awt.event.WindowEvent event)
                {
                    dispose(); // free the system resources
                    System.exit(0); // close the application
                }
            });
            setLayout(new java.awt.BorderLayout());
            setSize(300,300);
            choice = new ViewChoice();
            choice.setBlankChoiceAllowed(true);
            add(choice, "Center");

            choice.addItemListener( new java.awt.event.ItemListener() {
                public void itemStateChanged(ItemEvent ie)
                {
                    View value = choice.getSelectedView();
                    String viewname;
                    if ( value == null)

```

```

        {
            viewname = " none selected";
        }
        else
        {
            viewname = value.getName();
        }
        System.out.println("Selected value is now " + viewname);
    }
}
});
}
}
}
DriverFrame df;
df = new DriverFrame("ViewChoice Test");
df.show();
}
}

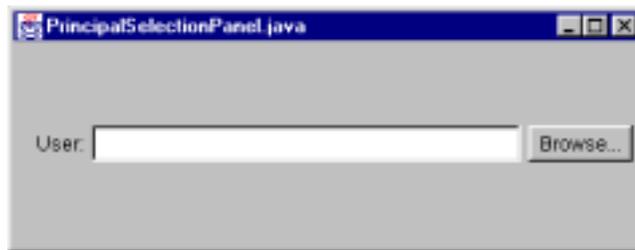
```

PrincipalSelectionPanel Bean

Overview

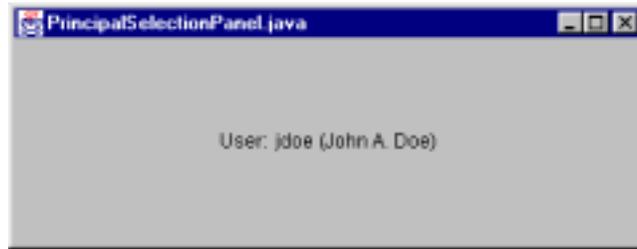
The PrincipalSelectionPanel bean allows the user to select a WTPrincipal (WTUser or WTGroup) by either entering the name of a WTPrincipal directly or by browsing lists of the available WTGroups or WTUsers. The bean can also simply display a WTPrincipal. This bean contains a label, a text field, and a browse button when used for selecting a principal. When the browse button is clicked, a dialog box opens that allows the user to browse a list or lists for a desired principal.

The bean has a selection mode (DISPLAY_SELECTION_MODE) and a view-only mode (DISPLAY_VIEW_ONLY_MODE). In selection mode, the user is allowed to select a WTPrincipal either by directly entering the name or by browsing for it. The following figure shows the PrincipalSelectionPanel bean in selection mode.



PrincipalSelectionPanel Bean in Selection Mode

In view-only mode, the WTPrincipal is simply displayed. The following figure shows the PrincipalSelectionPanel bean in view-only mode.



PrincipalSelectionPanel Bean in View-only Mode

The bean has a visible label mode (LABEL_VISIBLE_MODE) and an invisible label mode (LABEL_INVISIBLE_MODE). In visible label mode, the label for the principal is displayed. In invisible label mode, the label is not displayed.

The bean has three principal modes: user mode (USER_MODE), group mode (GROUP_MODE), and principal mode (PRINCIPAL_MODE). In user mode, only a WTPrincipal of type WTUser can be selected. In group mode, only a WTPrincipal of type WTGroup can be selected. In principal mode, either a WTUser or a WTGroup can be selected.

API

The PrincipalSelectionPanel bean contains methods to specify the following items at dev time:

- `displayMode` - Allows the user to set the display mode for the panel (DISPLAY_SELECTION_MODE or DISPLAY_VIEW_ONLY_MODE).
- `labelMode` - Allows the user to toggle between whether the principal label is visible or not (LABEL_VISIBLE_MODE or LABEL_INVISIBLE_MODE).
- `principalMode` - Allows the user to set the principal selection mode for the panel (PRINCIPAL_MODE, USER_MODE, or GROUP_MODE).
- `browseButtonLabel` - The label displayed for the browse button. A localized label is supplied in the resource bundle; changing the `browseButtonLabel` in the Property List will override the label from the resource bundle.
- `principalLabel` - The label displayed for the principal. A localized label is supplied in the resource bundle; changing the `principalLabel` in the Property List will override the label from the resource bundle.

The PrincipalSelectionPanel bean contains the following methods to be used at runtime:

- `getSelectedParticipant()` - Returns the selected WTPrincipal object.

- `addPropertyChangeListener(PropertyChangeListener l)` - Adds a `PropertyChangeListener` to the panel to receive a `PropertyChangeEvent` when a principal is selected.
- `removePropertyChangeListener(PropertyChangeListener l)` - Removes the specified `PropertyChangeListener`.
- `setEnabled(boolean b)` - Enables or disables both the browse button and the text field based on the value of `b`.
- `addHelpListener(PropertyChangeListener)` - Adds a listener to the bean's help system.
- `removeHelpListener(PropertyChangeListener)` - Removes a listener from the bean's help system.
- `getHelpContext()` - Returns the `HelpContext` currently being used.

Sample Code

Following is an example using the `PrincipalSelectionPanel` bean:

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

import wt.clients.beans.PrincipalSelectionPanel;
import wt.help.*;
import wt.org.WTPrincipal;

import symantec.itools.awt.StatusBar;

public class PSPTestFrame extends Frame
{
    public PSPTestFrame()
    {
        // This code is automatically generated by Visual Cafe
        // when you add
        // components to the visual environment. It instantiates
        // and initializes
        // the components. To modify the code, only use code syntax
        // that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable
        // to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        GridBagLayout gridBagLayout;
        gridBagLayout = new GridBagLayout();
        setLayout(gridBagLayout);
        setVisible(false);
        setSize(497,211);
        setBackground(new Color(12632256));
        psp = new wt.clients.beans.PrincipalSelectionPanel();
        psp.setBounds(0,0,495,143);
        psp.setBackground(new Color(12632256));
        GridBagConstraints gbc;
        gbc = new GridBagConstraints();
        gbc.gridx = 0;
        gbc.gridy = 0;
        gbc.weightx = 1.0;
        gbc.weighty = 1.0;
        gbc.fill = GridBagConstraints.BOTH;
        gbc.insets = new Insets(0,0,0,2);
        ((GridBagLayout)getLayout()).setConstraints(psp, gbc);
        add(psp);
        testButton = new java.awt.Button();
        testButton.setLabel("Test Button");
        testButton.setBounds(414,143,76,23);
        testButton.setBackground(new Color(12632256));
        gbc = new GridBagConstraints();
        gbc.gridx = 0;
        gbc.gridy = 1;
        gbc.weightx = 1.0;
        gbc.anchor = GridBagConstraints.EAST;
        gbc.fill = GridBagConstraints.NONE;
        gbc.insets = new Insets(0,0,5,7);
        ((GridBagLayout)getLayout()).setConstraints(testButton, gbc);
```

```

add(testButton);
statusBar = new symantec.itools.awt.StatusBar();
try {
statusBar.setBevelStyle(
    symantec.itools.awt.StatusBar.BEVEL_LOWERED);
}
catch(java.beans.PropertyVetoException e) { }
statusBar.setBounds(0,171,147,40);
gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 2;
gbc.weightx = 1.0;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.insets = new Insets(0,0,0,0);
((GridBagLayout)getLayout()).setConstraints(statusBar, gbc);
add(statusBar);
setTitle("PSP Test Frame");
//}}

//{{{INIT_MENUS
//}}}

//{{{REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
testButton.addActionListener(lSymAction);
//}}}

psp.setPrincipalMode(PrincipalSelectionPanel.PRINCIPAL_MODE);
psp.setDisplayMode(PrincipalSelectionPanel.DISPLAY_SELECTION_
    MODE);
psp.setLabelMode(PrincipalSelectionPanel.LABEL_VISIBLE_MODE);
psp.setBrowseButtonLabel("Custom Browse...");
psp.setPrincipalLabel("Custom Principal:");

HelpListener helpListener = new HelpListener();
psp.addHelpListener(helpListener);
}

public PSPTestFrame(String title)
{
    this();
    setTitle(title);
}

public void setVisible(boolean b)
{
    if(b)
    {
        setLocation(50, 50);
    }

    super.setVisible(b);
}

static public void main(String args[])
{
    (new PSPTestFrame()).setVisible(true);
}

```

```

}
public void addNotify()
{
    Dimension d = getSize();

    super.addNotify();

    if (fComponentsAdjusted)
        return;

    setSize(insets().left + insets().right + d.width,
            insets().top + insets().bottom + d.height);
    Component components[] = getComponents();
    for (int i = 0; i < components.length; i++)
    {
        Point p = components[i].getLocation();
        p.translate(insets().left, insets().top);
        components[i].setLocation(p);
    }
    fComponentsAdjusted = true;
}

boolean fComponentsAdjusted = false;

//{{{DECLARE_CONTROLS
wt.clients.beans.PrincipalSelectionPanel psp;
java.awt.Button testButton;
symantec.itools.awt.StatusBar statusBar;
//}}}

//{{{DECLARE_MENUS
//}}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == PSPTestFrame.this)
            Frame1_WindowClosing(event);
    }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    dispose(); // free the system resources
    System.exit(0); // close the application
}

class HelpListener implements PropertyChangeListener
{
    public void propertyChange(PropertyChangeEvent pce)
    {
        try
        {
            if (pce.getPropertyName().equals
                (HelpContext.TOOL_DESCRIPTION))
            {
                try

```

```

        {
            statusBar.setStatusText((String)
                pce.getNewValue());
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

catch (Exception e) {}
}

}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == testButton)
            testButton_ActionPerformed(event);
    }
}

void testButton_ActionPerformed(java.awt.event.ActionEvent event)
{
    WTPPrincipal test = psp.getSelectedParticipant();
    System.out.println(test);
}
}
}

```

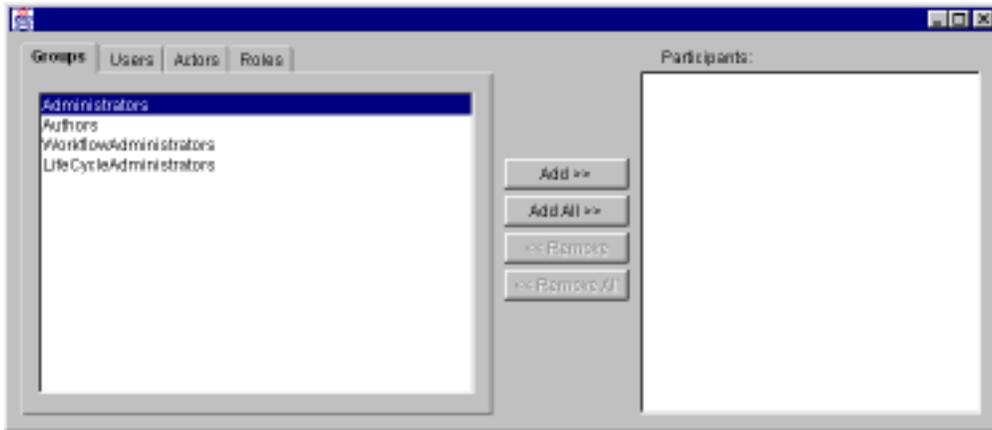
PrincipalSelectionBrowser Bean

Overview

The PrincipalSelectionBrowser bean allows the user to select a WTPPrincipal (WTUser or WTGroup), an Actor, or a Role. This is done by selecting one of the tabs on the left (Groups, Users, Actors, or Roles) and then selecting the desired participants from the list displayed on the selected tab. This bean contains a tab panel containing lists of the available participants, an Add button, an Add All button, a Remove button, a Remove All button, and a list to display all the selected participants.

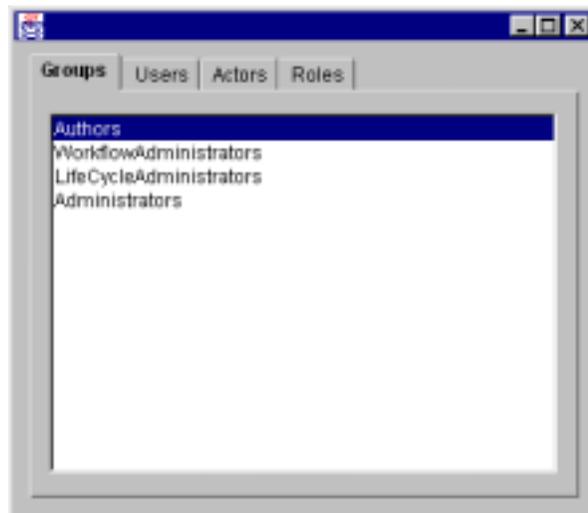
The bean has a multiple selection flag. If the multiple selection flag is set to true, the user can select multiple participants, and the buttons and selected participants

list are displayed. The following figure shows the PrincipalSelectionBrowser bean in multiple selection mode.



PrincipalSelectionBrowser Bean in Multiple Selection Mode

If the multiple selection flag is set to false, the user can select only a single participant and only the tabs are displayed. The following figure shows the PrincipalSelectionBrowser bean in single selection mode.



PrincipalSelectionBrowser Bean in Single Selection Mode

The bean allows each of the four tabs to be made invisible. For example, if you want to allow the user to browse for only users, then the groups, actors, and roles tabs could be made invisible and only the users tab would be displayed.

API

The `PrincipalSelectionBrowser` bean contains methods to specify the following items at development time:

- `multipleSelection` - Allows the user to set whether multiple participants can be selected. If the flag is set to true, multiple participants can be selected; if the flag is set to false, only a single participant can be selected.
- `Visibility flags` - The following flags can be set to control the visibility of tabs: `groupSelectionTabVisible`, `userSelectionTabVisible`, `actorSelectionTabVisible`, and `roleSelectionTabVisible`. If a flag is set to true, its corresponding tab is displayed; if it is set to false, the tab is not displayed.
- `deletedPrincipalsVisible` - Allows the user to set whether deleted principals are displayed for selection. If the flag is set to true, deleted principals are displayed and can be selected; if the flag is set to false, only active principals are displayed for selection.

The `PrincipalSelectionBrowser` bean contains the following methods to be used at runtime:

- `getSelectedParticipants()` - Returns a vector containing the selected participants.
- `addHelpListener(PropertyChangeListener)` - Adds a listener to the bean's help system.
- `removeHelpListener(PropertyChangeListener)` - Removes a listener from the bean's help system.
- `getHelpContext()` - Returns the `HelpContext` currently being used.

Sample Code

Following is a small example using the `PrincipalSelectionBrowser` bean:

```
//-----  
// java imports  
//-----  
  
import java.awt.*;  
import java.beans.PropertyChangeListener;  
import java.beans.PropertyChangeEvent;  
import java.util.Vector;  
  
//-----  
// wt imports  
//-----  
  
import wt.clients.beans.PrincipalSelectionBrowser;  
import wt.help.*;
```

```

//-----
// symantec imports
//-----

import symantec.itools.awt.StatusBar;

public class PSBTestFrame extends Frame
{
    public PSBTestFrame()
    {
        // This code is automatically generated by Visual Cafe
        // when you add
        // components to the visual environment. It
        // instantiates and initializes
        // the components. To modify the code, only use code
        // syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may
        // be unable to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        GridBagLayout gridBagLayout;
        gridBagLayout = new GridBagLayout();
        setLayout(gridBagLayout);
        setVisible(false);
        setSize(773,322);
        setBackground(new Color(12632256));
        psb = new wt.clients.beans.PrincipalSelectionBrowser();
        psb.setBounds(0,0,773,259);
        psb.setBackground(new Color(12632256));
        GridBagConstraints gbc;
        gbc = new GridBagConstraints();
        gbc.gridx = 0;
        gbc.gridy = 0;
        gbc.weightx = 1.0;
        gbc.weighty = 1.0;
        gbc.fill = GridBagConstraints.BOTH;
        gbc.insets = new Insets(0,0,0,0);
        ((GridBagLayout)getLayout()).setConstraints(psb, gbc);
        add(psb);
        TestButton = new java.awt.Button();
        TestButton.setLabel("Test Button");
        TestButton.setBounds(687,259,76,23);
        TestButton.setBackground(new Color(12632256));
        gbc = new GridBagConstraints();
        gbc.gridx = 0;
        gbc.gridy = 1;
        gbc.weightx = 1.0;
        gbc.anchor = GridBagConstraints.EAST;
        gbc.fill = GridBagConstraints.NONE;
        gbc.insets = new Insets(0,0,0,10);
        ((GridBagLayout)getLayout()).setConstraints(TestButton,
            gbc);
        add(TestButton);
        statusBar = new symantec.itools.awt.StatusBar();
        try {
            statusBar.setBevelStyle(
                symantec.itools.awt.StatusBar.BEVEL_LOWERED);
        }
    }
}

```

```

    }
    catch(java.beans.PropertyVetoException e) { }
    statusBar.setBounds(2,282,769,40);
    gbc = new GridBagConstraints();
    gbc.gridx = 0;
    gbc.gridy = 2;
    gbc.weightx = 1.0;
    gbc.fill = GridBagConstraints.HORIZONTAL;
    gbc.insets = new Insets(0,2,0,2);
    ((GridBagLayout)getLayout()).setConstraints(statusBar,
        gbc);
    add(statusBar);
    setTitle("PSB Test Frame");
    //}}

    psb.setMultipleSelection(true);
    psb.setGroupSelectionTabVisible(true);
    psb.setUserSelectionTabVisible(true);
    psb.setActorSelectionTabVisible(true);
    psb.setRoleSelectionTabVisible(true);

    HelpListener helpListener = new HelpListener();
    psb.addHelpListener(helpListener);

    //{{INIT_MENUS
    //}}

    //{{REGISTER_LISTENERS
    SymWindow aSymWindow = new SymWindow();
    this.addWindowListener(aSymWindow);
    SymAction lSymAction = new SymAction();
    TestButton.addActionListener(lSymAction);
    //}}
}
public PSBTestFrame(String title)
{
    this();
    setTitle(title);
}
public void setVisible(boolean b)
{
    if(b)
    {
        setLocation(50, 50);
    }
    super.setVisible(b);
}

static public void main(String args[])
{
    (new PSBTestFrame()).setVisible(true);
}

public void addNotify()
{
    Dimension d = getSize();

    super.addNotify();
}

```

```

        if (fComponentsAdjusted)
            return;

        setSize(insets().left + insets().right + d.width,
            insets().top + insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
            Point p = components[i].getLocation();
            p.translate(insets().left, insets().top);
            components[i].setLocation(p);
        }
        fComponentsAdjusted = true;
    }

    boolean fComponentsAdjusted = false;

    //{{DECLARE_CONTROLS
    wt.clients.beans.PrincipalSelectionBrowser psb;
    java.awt.Button TestButton;
    symantec.itools.awt.StatusBar statusBar;
    //}}

    //{{DECLARE_MENUS
    //}}

    class SymWindow extends java.awt.event.WindowAdapter
    {
        public void windowClosing(java.awt.event.WindowEvent event)
        {
            Object object = event.getSource();
            if (object == PSBTestFrame.this)
                Frame1_WindowClosing(event);
        }
    }

    void Frame1_WindowClosing(java.awt.event.WindowEvent event)
    {
        dispose(); // free the system resources
        System.exit(0); // close the application
    }

    class HelpListener implements PropertyChangeListener
    {
        public void propertyChange(PropertyChangeEvent pce)
        {
            try
            {
                if (pce.getPropertyName().equals
                    (HelpContext.TOOL_DESCRIPTION))
                {
                    try
                    {
                        statusBar.setStatusText((String)
                            pce.getNewValue());
                    }
                }
                catch (Exception e)
                {

```

```

        e.printStackTrace();
    }
}

catch (Exception e) {}
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent
        event)
    {
        Object object = event.getSource();
        if (object == TestButton)
            TestButton_ActionPerformed(event);
    }
}

void TestButton_ActionPerformed(java.awt.event.ActionEvent
    event)
{
    Vector v = new Vector();

    v = psb.getSelectedParticipants();
    System.out.println(v);
}
}

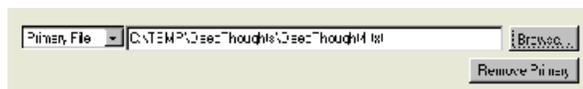
```

HTTPUploadDownload Panel Bean

Overview

The HTTPUploadDownloadPanel bean is used to specify the primary content of a FormatContentHolder object, such as a WTDocument. It allows the user to select whether the primary content is a file or a URL, enter the file path or URL, and remove an existing primary content entry. When the save() method is called, the primary content's file path or URL will be persisted, and if the primary content is a file then that file will be uploaded. If the file already uploaded is identical to the file in the local file path, a message will be displayed telling the user that the file wasn't uploaded because it hadn't changed.

When Primary File is selected in the drop down menu, the user can hit the Browse button to launch a FileLocator dialog to browse for the file path.



Primary File

When Primary URL is selected in the dropdown menu, the Browse button is disabled and the user directly types the URL into the text field.



Primary URL

To remove a primary content item, the user can either click the Remove Primary button or delete the contents of the text field. The Remove Primary button will become disabled and the text field will display a localized message of "No Primary".



No Primary

For situations where only files are valid as primary content, the ChooseFileOrURL dropdown can be hidden.

There is also a Get button which is typically hidden when this bean is used in the Windchill client. If displayed, the Get button is disabled when Primary File is selected and enabled when Primary URL is selected. Clicking the Get button will launch a browser window displaying the URL.

API

The HTTPUploadDownloadPanel bean contains methods to get/set the following items:

- The FormatContentHolder object being manipulated (get/setDocumentHandle).
- The target type (File or URL), target path, default filename (used by FileLocator), mode, and hostURL values (get/setTarget, get/setTargetType, get/setDefaultFilename, get/setMode, get/setHostURL).
- Whether or not the applet containing the bean is embedded in an HTML page which controls the applet via Javascript (is/setEmbeddedApplet).
- Whether or not the Get button, ChooseFileOrURL menu, and/or Remove Primary button are visible (is/setGetButtonVisible, setChooseFileOrURLVisible, is/setRemovable).

Once the bean is displayed, there are also methods available to:

- See whether the file path entered is valid on the local system (isValidTarget).
- Determine whether the file on the local system is identical to the file in the database (checksumsMatch).

- Check whether the panel values are different than the persisted values (isDirty).
- Persist the target information, including uploading if a primary file (save).

Sample Code

See `wt.clients.doc.CreateDocumentFrame` for use of this class in the Java client. The following code demonstrates a possible use of this class within an applet embedded in an HTML page. Initializing the `HTTPUploadDownloadPanel`:

```

/**
 * Upload mode indicator
 */
public static final int UPLOAD = HTTPUploadDownloadPanel.UPLOAD;

/**
 * Download mode indicator.
 */
public static final int DOWNLOAD = HTTPUploadDownloadPanel.DOWNLOAD;

public void init()
{
    ... other applet initializing code ...

    HTTPUploadDownloadPanel1 =
    new wt.clients.util.http.HTTPUploadDownloadPanel();
    HTTPUploadDownloadPanel1.setLayout(null);
    HTTPUploadDownloadPanel1.setBounds(0,0,998,452);
    HTTPUploadDownloadPanel1.setFont(new Font("Dialog", Font.PLAIN, 11));

    // Note: elsewhere in this class is a two-argument getParameter method which uses
    // the second argument as a default value if there is no parameter value for the
    // key given as first argument.

    // set panel color from parameter, otherwise default to grey
    HTTPUploadDownloadPanel1.setBackground(new Color(Integer.parseInt
    (getParameter
    ( "bgcolor", "12632256" ))));

    // hide getButton
    HTTPUploadDownloadPanel1.setGetButtonVisible(false);
    add(HTTPUploadDownloadPanel1);
    //}}

    // Initialize the panel.
    HTTPUploadDownloadPanel1.init( this );

    // Initialize from parameters.
    if ( getParameter( "debug", "FALSE" ).equalsIgnoreCase( "TRUE" ) ) {
        HTTPUploadDownloadPanel1.setDebugOption( true );
        HTTPUploadDownloadPanel1.setDebug( true );
    }

    HTTPUploadDownloadPanel1.setRemovable
    ( getParameter( "removable", "false" ).equalsIgnoreCase("true"));
    HTTPUploadDownloadPanel1.setHostURL ( getParameter( "hosturl", "" ) );
    HTTPUploadDownloadPanel1.setTargetType( getParameter( "type", "FILE" ) );

```

```

HTTPUploadDownloadPanell.setTarget( getParameter( "target", "" ) );
HTTPUploadDownloadPanell.setEmbeddedApplet(true);

// Note: elsewhere in the class, UPLOAD
if ( getParameter( "mode", "UPLOAD" ).equalsIgnoreCase( "UPLOAD" ) )
    HTTPUploadDownloadPanell.setMode( UPLOAD );
else
    HTTPUploadDownloadPanell.setMode( DOWNLOAD );

HTTPUploadDownloadPanell.setDefaultFilename
( getParameter( "defaultfile", " *.*" ) );

... other applet initializing code ...

}

Committing the save - be sure to check isDirty() before calling save():
private String saveContents() {
    String error_msg = "";

    try {
        if ( ( HTTPUploadDownloadPanell != null ) &&
            ( HTTPUploadDownloadPanell.isDirty() ) ) {
            if (!HTTPUploadDownloadPanell.save()) {
                Object[] params = {getDocumentIdentity(getDocument()
,getContext().getLocale())};
                error_msg = WTMesssage.getLocalizedMessage( RESOURCES,
                    DocRB.CONTENT_NOT_UPLOADED,
                    params,getContext().getLocale());
            }
        }
    }

    ... other applet logic dealing with the status of the save ...

} catch ( WTEException e ) {
    String identity = "";
    LocalizableMessage message_id = getDocument().getDisplayIdentity();

    if( message_id != null ) {
        identity=message_id.getLocalizableMessage
(getContext().getLocale());
    } else {
        identity = getDocument().getName();
    }

    Object[] params = { identity,
        e.getLocalizableMessage(getContext().getLocale()) };
    error_msg = WTMesssage.getLocalizableMessage( RESOURCES,
        DocRB.CONTENT_SAVE_FAILED, params,getContext().getLocale());
    } // catch

    return error_msg;
} // end saveContents()

```

When the bean is in an applet, embedded in an HTML page, `setEmbeddedApplet(true)` causes the pop-up Java message dialogs (such as error messages) to be disabled, in order to avoid potential browser-hanging problems. These error messages (like `error_msg` in the above example) should be displayed by other means, such as an HTML page response header.

Clipboard Support

Windchill clipboard support provides cut, copy, and paste functionality for Windchill applications. It is located in the `wt.clients.util` package and modeled after the classes in the `java.awt.datatransfer` package.

To gain access to a clipboard, use the `wt.clients.util.WTClipboard.getClipboard()` method. This is a static method that returns an instance of `java.awt.datatransfer.Clipboard`. All Windchill applets/applications loaded from the same codebase have access to a common clipboard. The `Clipboard` class contains `setContents` and `getContents` methods to set and get information to and from the clipboard. The objects that are placed on the clipboard must implement the `java.awt.datatransfer.Transferable` interface.

A `wt.clients.WTObjectSelection` class implements both the `Transferable` interface and `ClipboardOwner` interface. This class has two constructors that allow either a `WTObject` or an array of `WTObject` as input. The type of data being transferred is represented by the `java.awt.datatransfer.DataFlavor` class. A flavor named `WT_OBJECT_ARRAY_FLAVOR` has been created for Windchill objects; it is currently being statically stored in the `WTClipboard` class. The `WTObjectSelection` provides access to Windchill data in three different flavors (formats): `WTClipboard.WT_OBJECT_ARRAY_FLAVOR`, `DataFlavor.stringFlavor` and `DataFlavor.plainTextFlavor`. The two standard data flavors provide a Windchill object in the form of a URL to the properties page for the object. If the array of `WTObject`s being maintained in the clipboard is greater than 1, the URLs for each object are concatenated into one string separated by a space.

Copying a WTObject to the Windchill Clipboard

The following example shows how to copy a `WTObject` to the Windchill clipboard.

```
Clipboard clipboard = WTClipboard.getClipboard();
WTObjectSelection clipboard_contents = new
WTObjectSelection(wt_object);
// Clipboard.setContents requires a Transferable and
ClipboardOwner object.
// For convenience the WTObjectSelection implements both
interfaces.
clipboard.setContents(clipboard_contents, clipboard_contents);
```

Retrieving a WTObject From the Windchill Clipboard

The following example shows how to retrieve a `WTObject` from the Windchill clipboard.

```
Clipboard clipboard = WTClipboard.getClipboard();
Transferable clipboard_contents = clipboard.getContents();
WTObject wt_object_array[];
// Check to see if the windchill object array flavor is supported.
// An array will be returned.
```

```
if (clipboard_contents.isDataFlavorSupported
    (WTClipboard.WT_OBJECT_ARRAY_FLAVOR))
    wt_object_array = clipboard_contents.getTransferData
        (WTClipboard.WT_OBJECT_ARRAY_FLAVOR);
```

Accessing the System Clipboard

To enable clipboard support in Netscape, you must enable principal support. This can be done using the `prefwrangler.html` page from within the Netscape Browser and enabling the "signed.applets.codebase principal support" option. The `prefwrangler.html` file can be found in `wt/src/clients/prefwrangler` directory. Once the option has been enabled, when Windchill tries to access the system clipboard, the user is presented with a Java Security window requesting access to the system clipboard.

To enable clipboard support in Internet Explorer, you must modify the Internet Options security tab. You must add the Windchill Web site to the Trusted sites zone and make the following selections: Trusted sites zone > Custom > settings > java permissions, custom > JavaCustom Settings... > Permissions Given to Unsigned Content > edit Permissions > In RunUnsigned Content, select enable.

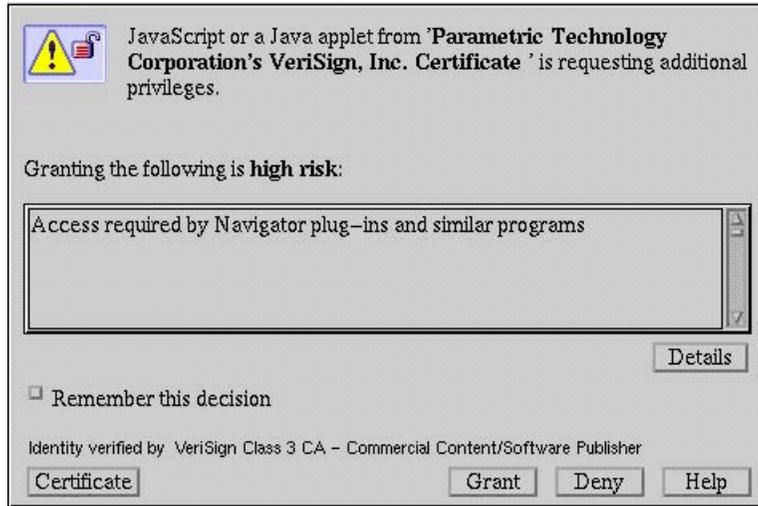
If you gain access to the system clipboard, the copy/paste works as follows. When an object is set in the Windchill clipboard, a string representation of the object is set in the system clipboard and Windchill "owns" the System clipboard at this point. If Windchill loses ownership of the system clipboard (due to an outside application setting contents in the system clipboard) and a paste occurs within the Windchill application, the contents from the system clipboard will be available to the Windchill application, rather than any contents that may have been stored in the Windchill clipboard. The fact that Windchill objects are being stored in a separate clipboard from the system clipboard is masked.

Programming Outside the Sandbox Using `security.jar`

Browsers restrict untrusted applet code by making it run within a *sandbox*. This practice separates the applet code from the rest of the system and denies access to privileged operations such as file access, network access, and system properties. For an applet to achieve access outside the sandbox, code invoking these sensitive operations must be signed, enabled, and granted by the user. This process has not been standardized among browsers, and obtaining a code-signing certificate is costly and time-consuming. To help applet programmers obtain access to operations outside the sandbox, without having to sign their individual code, the `security.jar` package has been developed. This package contains wrappers for commonly requested operations and handles all the browser and security issues.

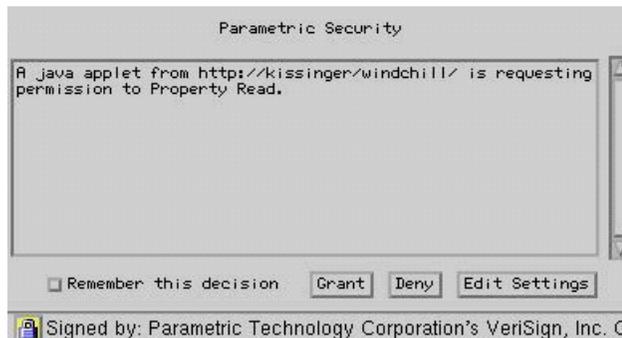
The `security.jar` package (and `security.cab` for Internet Explorer) is a package of classes that is signed with Parametric Technology Corporation's code-signing certificates. After a user is granted access outside the sandbox to the PTC

certificate (see the following figure), the browser will trust code from this package.



PTC Certification Dialog Box

Because any applet code could potentially use this package to gain access to privileged operations, another layer of security is maintained by the security package. This allows the user to grant or deny usage of security.jar by codebase (see the following figure). For example, code from www.widgets.com/windchill may be granted by the user; an new applet from www.rogueapplets.com would prompt the user to either grant or deny privileges to the given operation.



Applet Permission Dialog Box

To load the security package, the applet tag must specify the location of the signed code. Netscape loads signed classes specified in the ARCHIVE tag while Internet Explorer loads signed classes specified by the CABINETS parameter. The following tag will load the security package for either browser:

```
<applet code="mypkg/myapplet.class"
codebase="/windchill" archive="wt/security/security.jar">
```

```
<param name="cabinets" value="wt/security/security.cab">
</applet>
```

The following tag can also be used with the bootstrap loader:

```
<applet code="wt/boot/BootstrapApplet.class"
  codebase="/windchill" archive="wt/security/security.jar">
<param name="cabinets" value="wt/security/security.cab">
<param name="boot_jar" value="wt.jar">
<param name="boot_class" value="mypkg.myapplet">
</applet>
```

When the security package has been loaded, the applet may use the Access classes to perform operations outside the sandbox. Currently, the security.jar package provides FileAccess, NetAccess, PropAccess and RuntimeAccess. The basic template for using these operations is as follows:

```
File input = new File(FILENAME);
FileAccess fa = FileAccess.getFileAccess();
FileInputStream fis = fa.getFileInputStream(input);
. . .
fa.delete(input);
```

Threading

The following example shows a way to create a frame and populate lists within it. This example uses threading to process database events in parallel rather than sequentially and, thus, populates the frame more quickly. Three threads are started: one to get user attributes, one to get group membership, and one to get all available groups. When all three threads complete, the frame is activated.

```
private static final int ALL_GROUPS = 3;
private static final int MY_GROUPS = 2;
private static final int ATTRIBUTES = 1;

class ActionThread implements java.lang.Runnable {
    int action;

    ActionThread(int action) {
        this.action = action;
    }

    public void run() {
        try {
            switch (action) {
                case ATTRIBUTES:
                    userAttributes();
                    break;
                case MY_GROUPS:
                    populateBelongsToList();
                    break;
                case ALL_GROUPS:
                    populateGroupList();
                    break;
            }
            threadStatus(true, "");
        }
    }
}
```

```

        catch (Exception e) {
            threadStatus(false,e.toString());
        }
    }
}

private String thread_msg = "";
private boolean thread_status = true;
private int threads_needed = 0;
private int threads_complete = 0;

/** ...threadStatus
 * threadStatus is used for tracking when all the threads
 * used to create & populate the user frame have completed.
 * When each thread is run, it calls threadStatus to report
 * its completion status and any error message it captured.
 * When all the threads have completed, if no errors occurred
 * the frame is activated. If something failed a msgbox shows
 * the captured err message, then the frame is destroyed.
 */

private void threadStatus(boolean my_status, String ErrMsg) {
    threads_complete = threads_complete + 1;
    thread_status = thread_status & my_status;

    if (!my_status)
        //Save any error information
        thread_msg = thread_msg + ErrMsg + " ";

    if (threads_complete == threads_needed) {

        this.setCursor(Cursor.getDefaultCursor());
        if (thread_status) {
            //activate
            this.setEnabled(true);
        }
        else {
            //all threads are complete but someone failed
            DiaMessage m = new DiaMessage(this,
this.getTitle(),true);
            thread_msg = "Action failed. The following error
occurred: " + thread_msg;
            m.show(thread_msg);
            this.dispose();
        }
    } //end if (threads_complete)
}

public synchronized void show() {
    this.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    move(50, 50);
    super.show();

    //deactivate stuff
    this.setEnabled(false);

    threads_needed = 3;
    protectID();
    setTitle("Update User");
}

```

```

//Get user attributes in a new thread
ActionThread a = new ActionThread(ATTRIBUTES);
Thread t2 = new Thread(a);
t2.start();

//Get user group membership in a new thread
ActionThread b = new ActionThread(MY_GROUPS);
Thread t3 = new Thread(b);
t3.start();

//Get all available groups in a new thread
ActionThread g = new ActionThread(ALL_GROUPS);
Thread t1 = new Thread(g);
t1.start();
}

```

Refreshing Client Data

The following example shows how to refresh client data. It uses the following classes from the wt.clients.util package:

- RefreshEvent
- RefreshListener
- RefreshService

Assume the application in this example contains a browser that displays documents and the browser should be refreshed when new documents are created. New documents are created in a separate Create Document dialog box, so the main browser does not know when new documents are created; it must be told. To do this, the Create Document dialog dispatches a Refresh Event after creating a document. The main browser window registers (in the source for the main browser) as a listener to Refresh Events.

Registering as a Listener for the Event

Windows interested in receiving RefreshEvents must register with the RefreshService as a Listener:

```

// class DocBrowserFrame
RefreshListener aRefreshListener;
// In the DocBrowserFrame() constructor, the following code is
// added to register the DocBrowserFrame as a listener to RefreshEvents.
// DocBrowserFrame is registered by using inner classes (SymRefresh()).

RefreshService refresh_serv = RefreshService.getRefreshService();
RefreshListener aRefreshListener = new SymRefresh();
refresh_serv.addRefreshListener( aRefreshListener);

```

Listening for the Event

When a window hears an interesting event, it can update its own display:

```

// Inner class, 'SymRefresh', definition:

```

```

class SymRefresh implements RefreshListener {
    public void refreshObject( RefreshEvent evt ) {
        Object obj = evt.getSource();
        int action = evt.getAction();

        switch ( action ) {
            case RefreshEvent.CREATE:

                if( obj instanceof Document ) {
                    refreshDetailTable( (Document)obj );
                }
                break;

            case RefreshEvent.UPDATE:
                break;

            case RefreshEvent.DELETE:
                break;

            default:
                break;
        }
    }
}

```

Broadcasting the Event

When an object is created, updated, or deleted, an event is broadcast. When `CreateDocumentDialog` is invoked, and a new document is created, `CreateDocumentDialog` will broadcast a `RefreshEvent` to all `RefreshListeners`:

```

// In CreateDocumentDialog

    // In method, 'createDocument'

    .
    .
    .
    if( object != null ) {
        RefreshEvent evt = new RefreshEvent( object,
            RefreshEvent.CREATE );
        RefreshService.getRefreshService().dispatchRefresh( evt );
    }
    .
    .
    .

```

Broadcasting of `RefreshEvents` is actually being done in a separate thread, so the code is as follows:

```

// In 'createDocument'

    if( newDocument != null ) {
        RefreshThread refresh = new RefreshThread();
        refresh.setObject( newDocument );
        WThread wt_refresh = new WThread( refresh );
        wt_refresh.start();
    }

```

where the definition of the inner class, RefreshThread, is:

```
class RefreshThread implements Runnable {
    Object object;

    public void run() {
        if( object != null ) {
            RefreshEvent evt = new RefreshEvent( object,
                RefreshEvent.CREATE );
            RefreshService.getRefreshService().dispatchRefresh( evt );
        }
    }

    public void setObject(Object obj) {
        object = obj;
    }
}
```

Unregistering as a Listener to Events

When a window is closing or an application is exiting, any listeners to the RefreshService should be removed. If listeners do not unsubscribe with the RefreshService, the RefreshService will maintain a reference to the listener, potentially preventing garbage collection from occurring. In the docBrowserFrame example, the following code must be called prior to closing the window:

```
RefreshService refresh_serv = RefreshService.getRefreshService();
refresh_serv.removeRefreshListener(aRefreshListener);
```

Online Help

Windchill provides help classes that allow you to register an application with the online help system and create keys that are associated with the online help that is to be displayed. The following example links frames to help classes to provide online help for those frames. For example, under the comment line:

```
//Add help topics
```

the line:

```
helpContext.addComponentHelp(btnUsers, "UserAdmin");
```

associates the key "UserAdmin" with the Users button. The resource bundle at the end of the example, AdminHelpResources, links the key with the help that should be displayed for that button.

```
//These packages are needed for online Help
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyVetoException;
import wt.help.*;
//End of online help imports
```

```

private HelpSystem helpSystem;
private HelpContext helpContext;

public void addNotify() {

<Automatically-generated code was removed at this point to
improve readability.>

    //Set up Online Help
try {
    //Create the HelpContext.
    helpContext = new NFHelpContext(this, helpSystem, "UserAdmin");

    //Add help topics
helpContext.addComponentHelp(btnUsers, "UserAdmin");
helpContext.addComponentHelp(btnGroups, "GroupAdmin");
helpContext.addComponentHelp(btnDomains, "DomainAdmin");
helpContext.addComponentHelp(btnNotification, "NotifyAdmin");
helpContext.addComponentHelp(btnAccessControl, "AccessAdmin");
helpContext.addComponentHelp(btnIndexing, "IndexAdmin");
helpContext.addComponentHelp(txtCriteria, "Criteria");
helpContext.addComponentHelp(btnSearch, "Search");
helpContext.addComponentHelp(lstUsers, "UserList");
helpContext.addComponentHelp(btnCreate, "Create");
helpContext.addComponentHelp(btnAddUserToGroup, "AddToGroup");
helpContext.addComponentHelp(btnUpdate, "Update");
helpContext.addComponentHelp(btnView, "View");
helpContext.addComponentHelp(btnDelete, "Delete");
helpContext.addComponentHelp(btnClose, "Close");

    //Setup the status bar to display the tool description.
helpContext.addPropertyChangeListener(
    new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent event) {
            if (event.getPropertyName().equals
(HelpContext.TOOL_DESCRIPTION)) {
                try {
                    sbrStatusBar.setStatusText((String)
event.getNewValue());
                }
                catch (Exception e) {
                    System.out.println("Failure setting StatusText
while initializing Help listener: " + e);
                }
            } //if HelpContext.Tool_Description
        } //propertyChange()
    } //new PropertyChangeListener()
); //helpContext.addPropertyChangeListener(...)
} //try
catch (Exception e) {
    System.out.println
("Failure initializing context sensitive help: " + e);
}

//Localize
localize();

}

```

```

//{{{DECLARE_CONTROLS
.
.
.
//}}}

//{{{DECLARE_MENUS
//}}}

}

package wt.clients.administrator;

import java.util.*;
import java.lang.*;
public class AdminHelpResources extends java.util.
    ListResourceBundle
{
    public Object getContents()[][]
    {
        return contents;
    }

    static final Object [][] contents = {
//Localize this
//-----Administrator Defaults-----
{"Contents/Administrator", "contents.html"},
{"Help/Administrator", "BethHelp.html"},
{"Tip/Administrator//UserAdmin", "Administer user accounts"},
{"Tip/Administrator//GroupAdmin", "Administer user groups"},
{"Tip/Administrator//DomainAdmin", "Administer domains"},
{"Tip/Administrator//NotifyAdmin", "Define notification policy"},
{"Tip/Administrator//AccessAdmin", "Define access control policy"},
{"Tip/Administrator//IndexAdmin", "Define indexing policy"},

//-----User Administration-----
{"Contents/Administrator/UserAdmin", "contents.html"},
{"Help/Administrator/UserAdmin", "FAQ.html"},
{"Desc/Administrator/UserAdmin", "User Admin default"},

{"Desc/Administrator/UserAdmin/Criteria", "Specify search criteria;
press F9 for more information"},
{"Desc/Administrator/UserAdmin/Search", "Initiate search"},
{"Desc/Administrator/UserAdmin/UserList", "Use Search to populate
list of user accounts; click on entry to select it"},
{"Desc/Administrator/UserAdmin/Create", "Create a new user
account"},
{"Desc/Administrator/UserAdmin/AddToGroup", "Update group
membership for selected user"},
{"Desc/Administrator/UserAdmin/Update", "Change attributes for
selected user"},
{"Desc/Administrator/UserAdmin/View", "Examine attributes for
selected user"},
{"Desc/Administrator/UserAdmin/Delete", "Delete selected user
entry"},
{"Desc/Administrator/UserAdmin/Close", "Close this window"},

}
}

```

Preferences

The Preferences Framework is based on the principal that a unique preference consists of the following attributes:

- Parent Node (or root node if at the top of the hierarchy)
- Preference Node (usually associated as a group of similar preferences)
- Preference Key

Together these attributes form a unique key structure of parent/node/key. This unique key structure will be referred to as the fully qualified preference key. To separate individual user and group preferences for the same fully qualified preference key, a context is applied to the preference.

The context consists of the following elements:

- Macro — a constant defining the type of context (see below)
- (optionally) Descriptor — text defining the name of the context.

These elements are placed together with a ':' to form the Preference Context.'

The fully qualified preference key when placed together with a context will form a unique row in the database table, allowing users, and other divisions to have individual preferences.

Preference Macros

The `wt.prefs.WTPreferences` class defines the following types of Preference Context Macros:

- `USER_CONTEXT` - the context for individual users.
- `DEFAULT_CONTEXT` - the context for the system default (shipping) values.
- `CONTAINER_CONTEXT` - a context used in the container hierarchy.
- `CONTAINER_POLICY_CONTEXT` - a container context that is enforced as a policy.
- `DIVISION_CONTEXT` - the context used for any scopes defined in addition to the default, container, and user scopes.
- `DIVISION_POLICY_CONTEXT` - a division context that is enforced as a policy

Setting the Hierarchy

The `delegates.properties` value `wt.prefs.delegates.DelegateOrder` controls the hierarchy in which delegates are called. For each level in the hierarchy there should be an entry in this property. The customized entries should appear as `DIVISION_CONTEXT`. For example, in the out-of-the-box hierarchy, there is a

division scope called Windchill Enterprise, and the out-of-the-box wt.prefs.delegates.DelegateOrder property value is:

```
$DEFAULT,$CONTAINER,$DIVISION:WindchillEnterprise,$USER
```

In this value, there is no DIVISION_POLICY_CONTEXT defined since DIVISION_POLICY_CONTEXT and DIVISION_CONTEXT are related and are at the same level in the preference hierarchy. Similarly, the CONTAINER_POLICY_CONTEXT need not be included. Entries are designated differently only when storing and retrieving preferences internally. For more details on correctly naming delegates, see the delegates.properties file.

If wt.prefs.delegates.DelegateOrder has been removed from the delegates.properties file, Windchill uses the following:

```
$DEFAULT,$CONTAINER,$USERSetting Preferences
```

Edit the file Windchill/loadFiles/preferences.txt. This file is used to put the system values into the database. Note that you don't put quotes around the strings unless you actually want quotes persisted as part of the preference.

Syntax:

```
PrefEntry~keyName~default value~fullyQualifiedNodePath
```

Example:

```
PrefEntry~fileOperationType~ASK~/wt/content
```

Getting Preferences

By first navigating the preferences tree to the proper node, then setting the context for that particular user, then getting the value for that key.

Example:

```
// returns an instance of the top node in the Windchill preference
"tree"
Preferences root = WTPreferences.root();
// returns the preference node at that path
Preferences myPrefs = root.node( "/wt/content" );
((WTPreferences)myPrefs).setContextMask
(PreferenceHelper.createContextMask() );
// get( ), gets the value for that
// preference key
String prefValue = myPrefs.get( "fileOperationType", "SAVE" );
```

Clearing a Preference

There is a big difference between "clear" and "remove". Assuming there are no division-level defaults or policies, if you "clear" a user preference by setting the value to be the empty string "", then the value returned will be ""; but if you "remove" the user-level preference, then the value returned would be system default value. In most cases you will want to remove the user-level preference and not clear it, giving the user the upper hierarchal preference as their default.

Example:

```
Preferences root = WTPreferences.root();
    Preferences myPrefs = root.node( "/wt/content" );
    ((WTPreferences)myPrefs).setEditContext
(PreferenceHelper.createEditMask());
    ((WTPreferences)myPrefs).setContextMask
(PreferenceHelper.createContextMask());
    String prevValue = myPrefs.remove("fileOperationType");
```

The Preference Registry

The registry is a way to take a cryptic name like a preference and provide a localized series of data about it. This registry is in the form of rbInfo files. Anyone adding preferences to the system will have the option of adding this localized information to the Preference Registry. For more information, see the Windchill Customizer's Guide.

Adding a preference to the preference registry

To add a Preference to the Preference Registry, you need to edit the file /Windchill/src/wt/prefs/registry/prefRegistry.rbInfo and for the Preference you want to add, add at least:

```
DISPLAY_NAME
DESCRIPTION
DEFAULT
```

The format is as follows:

```
/node-name/key-name% [ ]tag.value=
```

Where /node-name is the name of the node (for example /wt/workflow), /key-name is the name of the key under the node (SortOrder) and % []tag is one of the tags mentioned above (% []DISPLAY_NAME).

Using Batch Containers

The batch container package resides in the package wt.container.batch. It provides the ability for a client application to gather a group of create, modify, and delete assertions and submit them to a service method for processing in one transaction.

Design Overview

The batch container package is generic and does not require the target objects to be of any particular class. Within Windchill applications, batch containers are generally used to batch groups of Persistable objects for server-side processing.

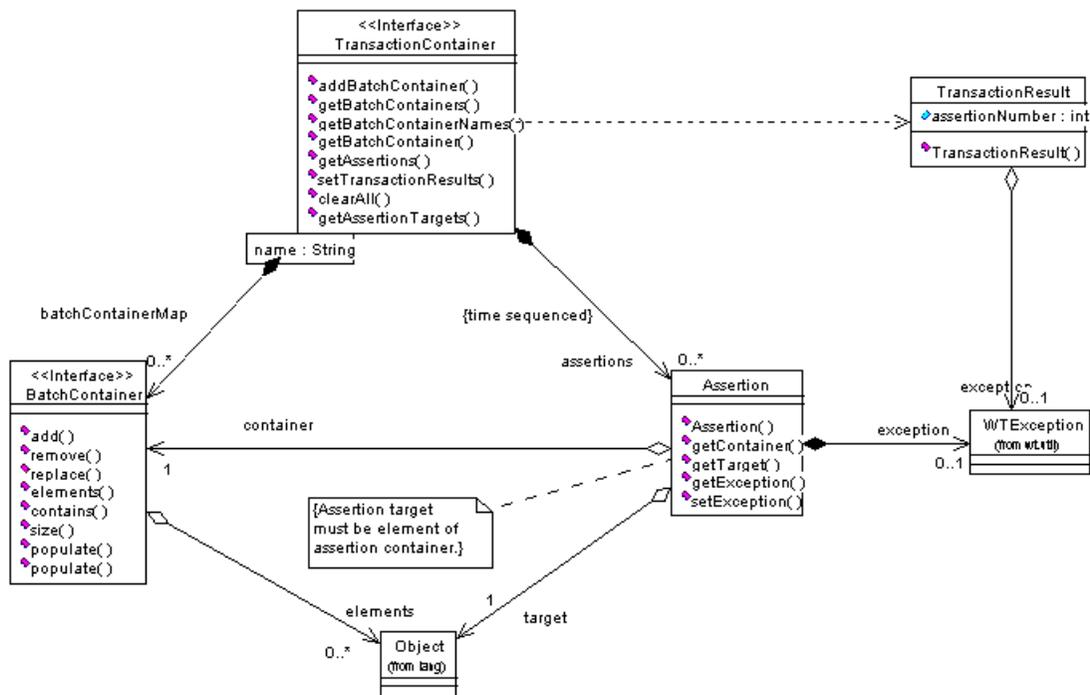
The batch container package provides no default server-side capability. The developer must implement the service required for their particular purpose. The

batch container package consists of two primary types of objects: BatchContainers and TransactionContainers.

A BatchContainer is a collection that represents the result of a series of object creation, modification and deletion assertions.

TransactionContainers are used to link one or more BatchContainers that are being used in a single update transaction. TransactionContainers keep a time-ordered record of BatchContainer changes by recording each BatchContainer change as an Assertion.

When processing the assertions carried by a TransactionContainer, the server-side service may encounter an error and refuse the requested changes. When this occurs, the service returns a TransactionResult which is accessible by the application through the TransactionContainer.



Batch Container Model

External Interface

In addition to the types already mentioned, the container.batch package includes the following types.

The RoleBatchContainer interface is a special purpose BatchContainer used to accumulate associative assertions. In addition to the functionality provided by type BatchContainer, the RoleBatchContainer carries information about the

association role being manipulated. This role information includes the association name and the role name.

The `BatchContainerFactoryIfc` specifies a type that manufactures `BatchContainers` and `TransactionContainers`.

The `Assertion` class is the abstract base class for assertions.

There are three specializations of the `Assertion` class: `AddAssertion`, `RemoveAssertion` and `ReplaceAssertion`. These classes represent assertions for adding, removing and modifying target objects.

The URLFactory

The `URLFactory` is a utility class provided to generate relative HREFs and support the Windchill Single-Point-Of-Contact (SPOC) environment. The `URLFactory` has been designed to take in a web-resource at a particular state in the Windchill system (from a certain request, host etc.) and generate an appropriate String HREF or URL object corresponding to that resource.

The `URLFactory` is meant to replace the functionality provided by the `GatewayURL` class. The `GatewayURL` class has several drawbacks including no support for relative HREFs, as all values returned are in the form of an URL which must be constructed and there is no support for remapping resources. As well, in complex SPOC configurations the hostname of the Windchill server may not be the hostname of the machine which sends the final page to the client, and thus currently links generated through firewalls can only work if IP masquerading and other techniques are used to fool the servers into redirecting the request properly.

The `URLFactory` is an instance based toolkit which may be used in either, java code directly, JSP Pages or HTML templates. For Java code directly, there are two constructors defined as can be seen in the javadoc. The most commonly utilized one will be:

```
URLFactory aURLFactory = new URLFactory( );
```

This will utilize the current server's codebase and properties for HREF construction.

Using the URLFactory with HTML

HTML Templates are not going to be using the `URLFactory` externally. Internally, the `GatewayURL` class will be re-written to utilize `URLFactory`, however this will not support relative links, but will support remapping of resources from one location to another. For highly stressed areas of Windchill such as local search result pages, these may be moved to utilize `URLFactory` code via Template processing eventually.

Using the URLFactory with JSP

See the following section, URLFactory in the JSP Environment for details on how to setup a JSP page to utilize the URLFactory.

Utilizing Windchill Gateways and Servlets with the URLFactory

In order to utilize the gateways and servlets with URLFactory a Helper class needs to be written like the `wt.httpgw.GatewayServletHelper` class. These helper classes will provide a series of methods to build properly formed links through the gateway or servlet. For example, using the `GatewayServletHelper` class a call to `buildAuthenticatedHREF()` will return a HREF utilizing the URLFactory with a call similar to the one below (for jsp)

```
wt.httpgw.GatewayServletHelper.buildAuthenticatedHREF  
(url_factory, "wt.httpgw.HTTPServer", "echo", "");
```

If writing a helper class, the `GatewayServletHelper` provides a good basis for developing further helper classes. Helper classes should not be extended but implemented on a package or servlet specific level and if possible should be declared final for jit compilation and optimization.

Utilizing relative links in the URLFactory

URLFactory is simply a processor for taking a resource from an arbitrary location located under the codebase and determining if several techniques such as relative links may be utilized for the desired resource. Relative links will automatically be returned if the RequestURI is set, and the desired resource can be resolved into a relative link (for example an external link to another site will always be returned as an external link).

Setting the URLFactory to Output Fully Qualified HREFs

In order to generate fully qualified HREFs, the easiest method is to set the request URI to null. For example,

```
URLFactory aFactory = new URLFactory( );  
aFactory.setRequestURI( null );  
String aHREF =  
aFactory.getHREF( "wt/clients/login/login.html" );
```

Would return the HREF to be

```
"http://hostname/Windchill-Codebase/wt/clients/login/  
login.html".
```

Writing a Mapping File

A mapping file is basically a flat-text property file. Empty lines, or lines starting with a # are ignored. The mapping filename for Windchill is `urlmap.properties` and should be located in the Windchill codebase. The URLFactory is smart enough to find it and utilize it automatically. However, currently a method server restart is necessary if you edit the contents of the file. This is being addressed

before R6.0 is released. The mappings should be key/value pairs and following the mapping schemes defined below.

Valid Mapping Schemes

There are many different mapping schemes and formats that may be used. However, to show all the values in this document would be too long. Please see the Javadoc for `wt.httpgw.URLFactory` mappings. Following are three common mappings that may be used, however altogether there are over twenty such mappings.

```
# Will map the folder wt/clients to a folder on newhost.com,  
in the Windchill6 codebase  
wt/clients/=http://newhost.com/Windchill6/wt/clients/  
  
# Will map the wt/clients folder to another folder  
wt/clients/=wt/customization/clients/  
  
# will map the resource index.html to the same  
protocol on a  
new host and codebase  
wt/clients/index.html=://newhost.com/Windchill6  
/wt/  
clients/index.html
```

Mapping a Particular File Type

Currently there is no functionality in the mapping scheme to map a particular file type to an alternate location/hostname. In future releases of Windchill this may be addressed if there is a need for this capability.

Mapping Components of Core Windchill

For Windchill Development, the mapping scheme should NOT be used to map any components of the core Windchill. This capability has been added to support multiple customer requirements, and should not be utilized as a way to make classes/documents of Windchill re-direct to other classes/documents. The mapping file may be used during development to remap to a new prototype or program, however proper HREFs should be implemented in any relevant files before the program is placed into production. There is a performance decrease using the URLFactory, and it should only be used to support customizations.

Capturing Errors From the URLFactory

Currently the only exceptions thrown by the URLFactory are at URLFactory instantiation. A `WTEException` will be thrown by URLFactory if there is a problem reading the `urlmap.properties` file at construction.

Character Encoding in the URLFactory

Character encoding in the URLFactory will be handled internally by the URLFactory. All HREFs and URLs will be properly encoded before being returned to the calling program. As well, any passed in parameters are assumed to

be in their decoded form and stored in the appropriate HashMap. Please see the methods `parseFormData` or `parseQueryString` in the javadoc.

URLFactory in the JSP Environment

There are several discrete steps required to utilize the URLFactory in the JSP environment.

1. Create the WtContextBean if not created.
2. Create a URLFactory Bean to be used for a given scope.

Setup the reference point for all HREFs (from a request, Base Tags, or fully qualified).

3. Create HREFs.

Creating an URLFactory

All Windchill Java client Server Pages or Servlets should set the context by using the WtContextBean using the code below:

```
<% [ ] /** The WtContextBean is a JavaBean for use in
Java Server
Pages or Servlets that wish to use Windchill Java client
or server
APIs ***/ % [ ]> <jsp:useBean id="wtcontext"
class="wt.httpgw.WtContextBean" scope="request"
<jsp:setProperty name="wtcontext" property="request"
value="<% [ ]=request% [ ]>" /> </jsp:useBean>
```

Next, an URLFactory instance may be created for the current codebase by using the useBean JSP command and setting the scope of the URLFactory (in this case for the request, which supports JSP include commands).

```
<% [ ] /** The URLFactory is a JavaBean for use in the
generation of HREFs used in Windchill HTML clients ***/ % [ ]>
<jsp:useBean id="url_factory" class="wt.httpgw.URLFactory"
scope="request" />
```

If a remote codebase needs to be accessed (note the request from the server) an URLFactory object can be instantiated by first creating the bean as above, and then recreating the URLFactory pointing it to the new codebase. (NOTE: The codebase ends with a '/' marking it as a directory.)

```
<% [ ] /** The URLFactory is a JavaBean for use in the
generation of HREFs used in Windchill HTML clients ***/ % [ ]>
<jsp:useBean id="url_factory" class="wt.httpgw.URLFactory"
scope="request" > <% [ ] url_factory = new wt.httpgw.URLFactory
( SomeURLToRemoteWindchill );
% [ ]> </jsp:useBean>
```

Setting the JSP Reference Point

The URLFactory needs to be informed of the current location, or way in which HREFs are to be generated. This can be done using four methods;

1. Setting the context to the current request (Recommended)
2. Setting the BaseTag in the page so that all HREFs are relative to the BaseTag
3. Forcing all HREFs to be fully qualified

Current Request Context

This is the recommended approach for developers. What needs to be done is to set the RequestURI of the URLFactory to be the request URI that was sent to the WEBServer. This will properly support reverse proxy configurations and firewalls. To configure URLFactory for this, utilize the setRequestURL method.

```
<% [ ]
    url_factory.setRequestURL(request.getScheme()
, request.getHeader("HOST"), request.getRequestURI());
% [ ]>
```

Setting a BaseTag

This configuration has the advantage of allowing a user to reload a page after it has been persisted (say to a local file system) and still have all the links work properly. If the base is to be the Windchill Codebase then a call to setRequestURIToBase() will suffice.

```
<BASE HREF="<% [ ]= url_factory.setRequestURIToBase() % [ ]>">
```

However in many situations, you may wish to set the base tag relative to some point in the Windchill codebase. An example is you want to generate a search page and have all the links be relative to some starting position. In this case, a little more is involved. If you can obtain the current request context (i.e., you are developing either a jsp or servlet page) then first you should set the request information using the setRequestURL() method described above.

Setting the BaseTag Within JSP pages and Servlets

```
<% [ ]
    // First set the request URI to be relative to the
request (thus maintaining
    // the protocol, port and host information.
    url_factory.setRequestURL(request.getScheme(),
request.getHeader("HOST"),
    request.getRequestURI());
    // Next set the request URI to be null
(ie. to the Windchill codebase)
    url_factory.setRequestURI(null);
    // Now set the request URI to be relative
to the resource you desire.
    url_factory.setRequestURI("wt/clients/login/
Login.html");
```

```

// Now the Base tag can be set to this resource
% [ ]> <BASE HREF="<%= [ ]=url_factory.getFullyQualifiedRequestURI
()">

```

Setting the BaseTag Within a Non-JSP Page or Servlet (i.e, Java code)

We will not have access to the request object in Java code that is not a servlet. Therefore we have to set the RequestURI to be relative to initially the Windchill codebase based on the configured host, protocol and port information and then set the requestURI to desired resource.

```

...
// Set the request URI to the Windchill codebase.
url_factory.setRequestURIToBase();
// Now set the request URI to the desired resource.
url_factory.setRequestURI("wt/clients/login/Login.html");
// Now we can obtain the string for the Base Tag
String baseTag = url_factory.getFullyQualifiedRequestURI();

```

Forcing all HREFs to be Fully-Qualified

If there is a need to force fully qualified HREFs (such as the current configuration) this can be accomplished using the following;

```

<%= [ ] url_factory.setRequestURI( (java.lang.String)null ); % [ ]>

```

Generating Links on the Page

There are five types of HREFs used in the Windchill Environment.

1. Standard resources located under the codebase.
2. Access to a Windchill Servlet via a Gateway (for example Authenticated Gateway)
3. Access to an external source
4. Forcing a fully qualified link with a URLFactory that has a non-null request URI
5. Creating a link from the request.getRequestURI() String

Standard Resources Under the Codebase

In order to generate links on the page, simply calling getHREF(resource) from URLFactory will create relative HREF Strings for the desired resource base on the RequestURI that was set earlier. The resources are all designated relative to the Windchill Codebase. For example, to obtain a link for Login.html located in wt/clients/login, the following line could be used:

```

<A HREF="<%= [ ]= url_factory.getHREF( "wt/clients/login/Login.html"
)"> Link to Login.html </A>

```

Depending on the Request Context setup above, the correct HREF String will be returned and inserted into the output page at compile time.

Access Through a Servlet

In order to generate links via a Windchill Servlet the appropriate ServletHelper class should be referenced. At initial unveiling, a ServletHelper is available for the Anonymous/Authenticated Gateways and helpers for the UIServlet and System Administrator will follow in builds after Build R6.0SYS_v0.15. Please see the javadoc for GatewayServletHelper for the proper method API.

```
<A HREF="<% [ ]= wt.httpgw.GatewayServletHelper.  
buildAnonymousHREF( url_factory,  
                    "wt.httpgw.HTTPServer", "echo", (java.lang.String)null )  
% [ ]">Windchill Echo</A>
```

Access to an External Source

To generate a link to an external source, simply make a call to getHREF as in example 1, except explicitly define the HREF to the resource you want. Whenever possible calls for external HREFs should be made through URLFactory to take advantage of possible future features.

```
<A HREF="<% [ ]= url_factory.getHREF( "http://www.ptc.com" )  
% [ ]"> Link to PTC's  
Website</A>
```

Forcing a Fully Qualified Link With a URLFactory that has a Non-null Request URI

A small caveat has been discovered with usage of the URLFactory. If the developer wants to create a link that opens a file in a new window (such as through the Javascript.open() method) the string must be fully qualified. However, the rest of the links on the page may be relative. How can this be achieved- The usage of the getHREF() methods that include the boolean switch will do this by optionally setting the URLFactory into fully-qualified mode for the duration of the method. This allows a single HREF to be generated without affecting the rest of the URLFactory. Like all getHREF() methods there are forms of the methods which take query strings and arguments'

```
<A HREF="<% [ ]= url_factory.getHREF( "wt/clients/login/Login.html"  
, true ) % [ ]">Fully Qualified Link</A>
```

Creating a Link From the Request.getRequestURI() String

This only applies to servlets and JSP Pages. The request object has a method getRequestURI() which returns the path to the web resource (usually starting after the hostname/port onward). This String includes a leading '/' which by definition in the URLFactory would redefine the resource and not create a relative link. However, there is a chance to create a relative link if the getRequestURI() path and the Windchill codebase are common. This is where usage of the determineResource() method may be used.

```
<A HREF="<% [ ]= url_factory.getHref(
    url_factory.determineResource(request.getRequestURI()))% [ ]>"
> Resource Link</A>
```

Internationalizing JSP Pages in Windchill

With the introduction of JSP and new browser technology, it is no longer necessary to use separate encodings for each language. Instead, a single encoding: UTF-8 can be utilized. All JSP Pages in the Windchill environment should be written to utilize UTF-8 as their character set. This will allow a single JSP Page to be deployed for any language variant. (Previous versions implemented separate language versions for all HTML templates.)

International Encoding on JSP Pages

There are two steps that are needed to develop pages properly using UTF-8.

1. To set the content type of the current page that is being processed within the JSP Parser. The JSP page directive needs to be called with the content type being set.

```
<% [ ]@ page contentType="text/html; charset=UTF-8"% [ ]>
```

2. Set the response content type. This will set the encoding of the HTML page returned to the client. It should be set near the start of the JSP page (after Bean declarations). In order to do this the following needs to be added:

```
<% [ ] response.setContentType( "text/html; charset=UTF-8" ); %
[ ]>
```

Note: The response content type method must be called before any call to request.getParameter() is made.

Decoding UTF-8 Encoded Text

If you set the encoding as explained above, the text your code receives will be encoded using the UTF-8 encoding. However, the Java String class uses 16-bit integers to represent Unicode characters. You need to do the conversion from UTF-8 to Java String in your code.

EncodingConverter

Note: To find out more about the EncodingConverter class, please refer to the Windchill Javadoc.

The EncodingConverter class (in the wt.httpgw package) contains a series of decode() methods that may be used to efficiently decode text that has been encoded in UTF-8 format. This must be called on ALL text that is read from parameters that were submitted from a FORM element. There are two ways to decode text:

The first method is to use the decode() methods of wt.httpgw.EncodingConverter to efficiently decode text that has been encoded in UTF-8 format. This must be called for every text entry that was submitted from a FORM element, or any parameters that contain encoded text. This methodology supports both request query strings and FORM data. For example:

```
// Create an instance of the encoding converter for the page
//
EncodingConverter encoder = new EncodingConverter
();
// Using the EncodingConverter with the default
encoding (UTF-8 )
//
String tag = encoder.decode( request.getParameter
("tag") );
// Using the EncodingConverter with a specific
encoding
//
String tag = encoder.decode( request.getParameter
("tag"), "UTF-8");
```

The second method that can be used is the parseQueryString() method of the URLFactory() class. Usage of the parseQueryString() method takes an encoded query string and decodes it. The result is placed into a HashMap. The HashMap values may then be queried using the HashMap 'get' method. This method will only work with Request Parameters and not form elements. For example:

```
// Use the URLFactory to parse the Query String
//
java.util.HashMap query_map =
url_factory.parseQueryString(
request.getQueryString() );
// Retrieve the (already decoded) string
from the hash map
//
String tag = query_map.get("tag");
```

Deprecation of WTURLEncoder

As of Windchill release 6.0 WTURLEncoder SHOULD NOT be used for text encoding or decoding. This class may be removed post-R 6.0.

Encoding of Forms

By default POST submitted forms are submitted using the encoding application/x-www-form-urlencoded. The methods provided above for decoding the text will allow these form's data to be written properly.

All HREFs and URLs should be generated using the URLFactory class. The URLFactory provides methods to automatically encode any non-ASCII characters in the query string, if a HashMap containing the parameters is provided to the URLFactory. If a string is being passed in for a query string that has been created within your code, you must encode it first, with a call to EncodingConverter.encode()

Note: See the URLFactory javadoc for details.

Interfacing JSP Pages and Templates

Currently, since the template files are written with language specific encodings, you can not submit a form from a JSP page to a Template file for processing. It is planned for Windchill R6.0 to move some of the Template Processing to JSP format or UTF-8 encoding, however developers working with Windchill R6.0 are advised to assume this may not be available. Please check the current situation with the appropriate client development group.

Sample JSP Page

Below is a sample JSP Page that illustrates how to utilize the URLFactory, and proper UTF-8 encoding with a HTML form. If the file is saved into your Windchill codebase directory as sample.jsp, you can execute this sample. To see the power of this sample try to paste different languages into the text field and then click the button.

```
<HTML>
<% [ ] /** WTContest bean for Windchill Client */ % [ ] >
<jsp:useBean id="wtcontext" class="wt.httpgw.WTContextBean" scope="request">
<jsp:setProperty name="wtcontext" property="request" value="<% [ ]=request% [ ]>" />
</jsp:useBean>
<jsp:useBean id="url_factory" class="wt.httpgw.URLFactory" scope="request" >
<% [ ] // Set the URLFactory to the current page request
    url_factory.setRequestURL(request.getScheme(), request.getHeader("HOST"),
        request.getRequestURI());
% [ ] >
</jsp:useBean>
<% [ ] @ page contentType="text/html; charset=UTF-8" % [ ] >
<HEAD>

<% [ ]
// Set the content type for the response
response.setContentType("text/html; charset=UTF-8");
// Get the current locale for the browser that is supported by Windchill
// This to used with WTMessage.getLocalizedMessage(RESOURCE,"tagname",locale)
java.util.Locale locale = wt.httpgw.LanguagePreference.getLocale(
    request.getHeader("Accept-Language") );
% [ ] >

</HEAD>

<BODY>
<h1>Sample Form</H1>
<% [ ]
    String text = request.getParameter("sample_text");
    if ( text != null )
    {
% [ ] >

Here was the text that was submitted: <i>
<% [ ]= url_factory.decode(text) % [ ] > </I>
<BR><% [ ]= new String(text.getBytes("ISO8859_1"),"UTF-8")% [ ] >
<% [ ] } % [ ] >
```

```

<P>
<FORM METHOD="POST"
  ACTION="<% [ ]=url_factory.getHref("sample.jsp",request.
getQueryString
() ) % [ ]>" >
  <% [ ]
    if ( request.getParameter("sample_text") != null )
    {
  % [ ]>
  <INPUT TYPE="text" NAME="sample_text"
    VALUE="<% [ ]= url_factory.decode( text) % [ ]>">
  <% [ ] }
    else {
  % [ ]>
  <INPUT TYPE="text" NAME="sample_text" >
  <% [ ] } % [ ]>
  <INPUT TYPE="submit" NAME="button" VALUE="click here">
</FORM>
</BODY>
</HTML>

```

Localizable text on JSP pages

Any displayed text should be obtained from ResourceBundles or RbInfo files that are language specific as defined by the Internationalization/Localization techniques located here. **IMPORTANT:** you must not hard-code localizable text directly on the JSP page! This would ruin the whole concept of having language-independent JSP pages. Whenever text needs to be displayed on a JSP page, it should be pulled in during runtime from an RbInfo file.

The Rbinfo File Format

Resource info (or rbInfo for short) files are resource files used to store localizable strings of Java programs. The primary purpose of the rbInfo files is to provide an easier and more manageable way to handle localizable strings than resource bundles. RbInfo files offer a number of advantages over resource bundles:

- Resource bundle files are Java source files, so that a single misplaced curly bracket, missing double quote or extra comma will cause a syntax error and break the compile and integration process. RbInfo files have much simpler format, it is easier to localize and more difficult to introduce syntax errors.
- Because of the simpler format of the rbInfo files, it is easier to handle them with localization tools; perform change tracking, change propagation and so on.
- It is more difficult to abuse the rbInfo file format and introduce 'tricky' resource types. Java resource bundles can hold any type of objects, but rbInfo files can handle strings only. (This may appear to be a limitation, but it is not. It makes localization easier.)

RbInfo files are converted to compiled Java class files in the integration process, so that the same naming convention rules apply to rbInfos as resource bundles.

(Localized versions are kept in separate files; there is one resource file per language, the name of the locale is appended to the name of the localized files.)

The format of the rbInfo files is PTC-specific. It was designed primarily for Windchill 6.0, but can be used in other Java-based products as well. The migration from resource bundles to rbInfo files is seamless; there is no need to change the source code. Old resource bundles can be converted to rbInfo format using a relatively straightforward process. To find out more about the migration, refer to the *Windchill Upgrade and Migration Guide*.

11

Internationalization and Localization

Internationalization is the process of designing and developing an application that can be easily adapted to the cultural and language differences of locales other than the one in which the application was developed. Localization is the process of adapting an application to fit the culture and language of a specific locale.

All Windchill applications are fully internationalized and ready to be localized. Windchill applications are delivered with a default locale of US English (en_US).

This chapter describes how to localize text visible to the user by using resource bundles. For additional information on localizing HTML templates, see the *Windchill Customizer's Guide*.

Topic	Page
Background	11-2
The Windchill Approach	11-2
Localizing Text Visible to the User	11-4

Background

Changing an application for use in another country or culture is often thought of as merely translating the language that appears in the user interface. There are many other aspects, however, that you should consider when developing a global application.

- How will you identify the preferred language and geographic location of the user-

You may want to design into the application (or underlying product architecture) the ability to determine the locale and present the appropriate version from a collection of different localized versions.

- What data used within your application is sensitive to locale-

Consider the use of decimals within numbers, currency symbols, date formats, address styles, and system of measurement.

- How should data be formatted-

Consider the order in which text and numbers are read by different audiences. Languages that display numbers from left to right and text from right to left affect the layout of menu bars and text entry fields. The grammar of a language may dictate different placement of variables in error messages.

- Collation of sortable lists

Consider how different alphabets affect the collation sequence and how collation of typical list elements is done in the locales of potential users of your application.

- Non-Roman alphabets

Your application must be able to accommodate different fonts and different sizes of fonts. This again can affect the layout of menu bars and text entry fields.

- What are the cultural sensitivities toward graphics and use of color-

When designing icons or other graphics, and deciding on background and other colors, consider whether they may be objectionable in another culture

Both client and server developers need to be aware of these factors. You must be able to localize not only the GUI, but also feedback messages and exceptions that might be displayed to the user.

The Windchill Approach

Rather than try to keep all these preceding factors in mind and accommodate them individually as you develop an application, the best approach is to isolate any language- or locale-dependent code from the language-independent code (that is, the application's executable code). Windchill is designed to allow you to do this.

Windchill takes advantage of many Java features that support international applications:

- Locale class

Each locale-sensitive object maintains its own locale-specific information. The initial default for locale is specified in the system but users can specify a preference in the Web browser.

- Resource bundles

In a resource bundle, you define pairs of keys and values, where the values are strings and other language-dependent objects for a specific locale. Within code, you use the key to indicate where the corresponding string or object should be inserted. For example, Windchill uses resource bundles in its online help and to identify button names, field names, and other elements of graphic user interfaces. The default or preferred locale specifies which resource bundle to use and, therefore, determines which strings and objects to display. (An example is shown later in this chapter.)

Windchill uses a structured properties file format to manage much of the localizable text. Unlike the `java.util.PropertyResourceBundle` properties files, these resource info files are not used at runtime. They are more like `java.util.ListResourceBundle` java files, where they are used to manage the information, and runtime resource bundles are built from them. These resource info files have a `.rbInfo` file extension. This format is required for managing the localizable information for `EnumeratedTypes` and display names for metadata, since these localizable resources are updated by generation tools. The resource info format can be used for storing other localizable text, but it is not mandatory.

- Unicode

This is a 16-bit international character-encoding standard. A character encoding is a numeric representation of alphanumeric and special text characters. A multi-byte encoding is necessary to represent characters such as those used in Asian countries. The intent of Unicode is to be able to represent all written languages in the world today.

- Localized text manipulation

The Java classes `java.io.inputStreamReader` and `java.io.OutputStreamWriter` provide the mechanism to convert standard character encodings to Unicode and back, thus enabling the translation of characters to and from platform and locale-dependent encoding.

- Handling local customs

The `java.text` package provides classes that convert dates and numbers to a format that conforms to the local conventions. This package also handles sorting of strings.

- `java.text.NumberFormat`. formats numbers, monetary amounts, and percentages.
- `java.text.DateFormat` contains the names of the months in the language of the locale and formats the data according to the local convention. This class is used with the `TimeZone` and `Calendar` classes of the `java.util` package. `TimeZone` tells `DateFormat` the time zone in which the date should be interpreted and `Calendar` separates the date into days, weeks, months, and years. All Windchill dates are stored by the server in the database based on a conversion to Greenwich Mean Time.

To display Timestamps in the correct Timezone, the application programmer should use `wt.util.WTContext` to set the Timezone in the `DateFormat` as follows:

```
DateFormat df = DateFormat.getDateTimeInstance(
    DateFormat.SHORT,
    DateFormat.SHORT, WTContext.getContext().getLocale()
);
df.setTimeZone(WTContext.getContext().getTimeZone());
System.out.println("The current time is: " +
    df.format(new Timestamp(current_time_millis)));
```

- `java.text.Collator` can compare, sort, and search strings in a locale-dependent way.

Localizing Text Visible to the User

Windchill provides internationalized applications with US English (`en_US`) as the default locale. We recommend that you provide a localized resource bundle for every other locale that you support.

Resource bundles are used to hold information, generally text, that you may want to modify based on your locale. A resource bundle is a hash table of key/value pairs, where the values are specific to the locale. Every package should have a resource bundle. The Windchill naming convention is as follows:

<your package name >.<pkg>Resource.class

Implementation classes have a generated constant, `RESOURCE`, to identify their fully qualified resource bundle class.

Resource bundles are loaded at runtime based on the system setting or user-specified preference for locale. To load the resource bundle, a Java program calls `java.util.ResourceBundle.getBundle`, specifying the base name of the desired `ResourceBundle`. For example, the algorithm to find a `ResourceBundle` named `fc.fcResource` is as follows:

1. Search for a class with the name `fc.fcResource_language_country_variant`.
2. Search for a class with the name `fc.fcResource_language_country`.
3. Search for a class with the name `fc.fcResource_language`.

4. Search for a class with the name fc.fcResource.

All Windchill resource bundles are provided for the default locale en_US. Because these resource bundles are specified by the base name, they have no extension.

Because IDEs, such as Visual Café, generate code to handle graphical components and interactions, do not put references to resource bundles in sections that have been generated. If you make any changes and regenerate the code, those references will be lost. Instead, create a localize method that overrides the hard-coded label with the appropriate label from a resource bundle and put it outside the generated code area. (Visual Café, for example, indicates generated code with markers. Code you put outside those markers is retained across regenerations.)

The following example shows how to make visible text locale dependent. For example, within the localize method, the line:

```
lblUser.setText(RB.getString("lblUser") + ":");
```

associates the label defined internally as lblUser with the string found in the resource bundle that corresponds to the lblUser key; that is,

```
{"lblUser", "User"},
```

The string "User" is then displayed in this label.

```
static ResourceBundle RB;

public void addNotify() {
    //Localize
    localize();
}

//{{{DECLARE_CONTROLS
//}}}

//{{{DECLARE_MENUS
//}}}

}

private void localize() {
    RB=ResourceBundle.getBundle("wt.clients.administrator.LabelsRB",getLocale());

    lblUser.setText(RB.getString("lblUser") + ":");
    btnSearch.setLabel(RB.getString("btnSearch"));
    btnCreate.setLabel(RB.getString("btnCreate"));
    btnUpdate.setLabel(RB.getString("btnUpdate"));
    btnAddUserstoGroup.setLabel(RB.getString
"btnAddUserstoGroup"));
    btnView.setLabel(RB.getString("btnView"));
}
```

```

        btnDelete.setLabel(RB.getString("btnDelete"));
        btnClose.setLabel(RB.getString("btnClose"));
        try {
            //MultiList column headings
            java.lang.String[] tempString = new java.lang.
String[4];
            tempString[0] = RB.getString("Full Name");
            tempString[1] = RB.getString("UserID");
            tempString[2] = RB.getString("Web Server ID");
            tempString[3] = RB.getString("E-Mail");
            lstUsers.setHeadings(tempString);
        }
        catch (PropertyVetoException e) {}
    }
}

```

(If using rbInfo files, See Resource Info section below.)

```

package wt.clients.administrator;

import java.util.ListResourceBundle;

public class LabelsRB extends java.util.ListResourceBundle
{
    public Object getContents()[][] {
        return contents;
    }

    static final Object[][]contents = {
        //Labels
        {"lblAdministrative", "Administrative"},
        {"lblAllGroups", "All Groups"},
        {"lblAttach", "Attach"},
        {"lblAuthorization", "*Web Server ID"},
        {"lblBelongs", "Groups User Belongs to"},
        {"lblCity", "City"},
        {"lblCountry", "Country"},
        {"lblCreate", "Create"},
        {"lblCreated", "Created"},
        {"lblDelete", "Delete"},
        {"lblDescription", "Description"},
        {"lblEmail", "E-Mail"},
        {"lblFullName", "Full Name"},
        {"lblGroup", "Group"},
        {"lblGroupName", "Group Name"},
        {"lblID", "*ID"},
        {"lblLocale", "Locale"},
        {"lblModify", "Modify"},
        {"lblName", "Name"},
        {"lblRead", "Read"},
        {"lblState", "State"},
        {"lblStreet1", "Street1"},
        {"lblStreet2", "Street2"},
        {"lblTitle", "Title"},
        {"lblUse", "Use"},
        {"lblUser", "User"},
        {"lblUserName", "User Name"},
        {"lblZip", "Zip"},
    }
}

```

```
//Button Labels
{"btnAdd", "Add>>"},
{"btnAddAll", "Add All>>"},
{"btnAddRemove", "Add/Remove Members"},
{"btnAddUserstoGroup", "Add User to Group"},
{"btnApply", "Apply"},
{"btnCancel", "Cancel"},
{"btnClear", "Clear"},
{"btnClose", "Close"},
{"btnCreate", "Create"},
{"btnDelete", "Delete"},
{"btnGenerate", "Generate Now"},
{"btnNewGroup", "New Group..."},
{"btnNewUser", "New User..."},
{"btnOK", "OK"},
{"btnRefresh", "Refresh"},
{"btnRegenerate", "Regenerate"},
{"btnRemove", "<"},
{"btnRemove", "<"}
```

To create a different localization for this resource bundle, for example, French, you would create a new class in the `wt.clients.administrator` package called `LabelsRB_fr`. This class would contain the same label keys, such as `"lblAdministrative"` but its value would be `"administratif"` rather than `"Administrative"`. All the other values would likewise be changed to their French counterparts. You would compile the new class; then the Java runtime would be able to find a French resource bundle for the Administrator client.

An example of resource bundles being used in online help is given in the preceding chapter on developing client logic.

Resource Info (.rbInfo) Files

wt.L10N.complete

Resource Info files are an alternative to storing localizable text in `ListResourceBundle` source code files. They are structured properties files that facilitate easy manipulation by automated tools.

General Resource Info File Usage Rules

- A line beginning with `'#'` is considered a freeform comment.
- Each file must contain a header line that categorizes the file.
- Only String values are supported.
- Since values are assumed to be Strings, they should not be in quotes.
- Each entry must exist on a single line, and the following escaped characters are supported: `\\`, `\n`, `\r`, `\t`, `\f`, `\"`.
- Key cannot contain `'='`, since it is the key/value separator.

- Key cannot contain "#", since it is a comment character, but the character is allowed in the value.

Localizable text is considered to be in one four categories for the purpose of resource info file usage.

Resource Type	Source File	Run-Time File
Message Text	*RB.rbInfo *Resource.rbInfo	*RB.class *Resource.class
Modeled Metadata (Display Names)	ModelRB.rbInfo	ModelRB.RB.ser
EnumtaedType Options Definition	<EnumType>RB.rbInfo	<EnumType>RB.RB.ser

Message Text

The Message Text category most commonly contains error messages and labels for user interface actions, but is the general category for any localizable text that does not fall into one of the other categories. The Message Text files are completely user maintained, while the maintenance of the entries in the other three categories is automated via various generation tools. Since this category is not maintained by automated tools, and since the resulting run-time bundle is the same ListResourceBundle subclass that it would be if the information were stored in a ListResourceBundle source code file, the use of .rbInfo file format is optional for Message Text.

Note: The following sections describe the resource info files for Message Text.

Message Text Resource Info Header

Each resource info file contains the following lines that define certain file level information.

```
ResourceInfo.class=wt.tools.resource.StringResourceInfo
ResourceInfo.customizable=false
ResourceInfo.deprecated=false
```

The first line classifies the resource info and should never be changed. The values of the second to lines can be changed by the owner of the package, if the file can be customized, and/or the file is deprecated.

Message Text Resource Entry Format

The following keys define the structure of a Message Text rbInfo file.

Key	Description
<key>.value	The localizable text that will be displayed. (required)
<key>.constant	A string that will be used to generate a constant field into the runtime resource bundle, which can be used by code that does resource lookups.
<key>.comment	A comment describing the entry. (optional)
<key>.argComment <n>	A comment for each substitution argument of the value string. (optional)

Message Text Resource Entry Examples

```
//Labels

lblAdministrative.value=Administrative
lblAdministrative.constant=LBL_ADMIN
lblAdministrative.comment=administrative ui label

lblAllGroups.value=All Groups
lblAllGroups.constant=LBL_ALL_GROUPS

//Button Labels
btnAdd.value=Add>>
btnAdd.constant=BTN_ADD
btnAddAll.value=Add All>>
btnAddAll.constant=BTN_ADD_ALL

//MultiList column headings
Class.value=Class
Created On.value=Created On
```

Building Runtime Resource Bundles for Resource Info Files

Since the information is not stored in Java source code files, a tool other than the Java compiler is needed to build the runtime resource bundles. This tool can be executed by using the ResourceBuild script. For more information, see the Command Line Utilities section of the System Generation chapter of the *Windchill Application Developer's Guide*.

Generated Resource Info Files

The two categories of Resource Info files that are generated are discussed in detail in other areas.

Resource Type	Documented
Modeled Metadata (Display Names)	Windchill Application Developer's Guide — System Generation chapter
EnumeratedType Options Definition	Windchill Customizer's Guide — EnumeratedType Appendix A

12

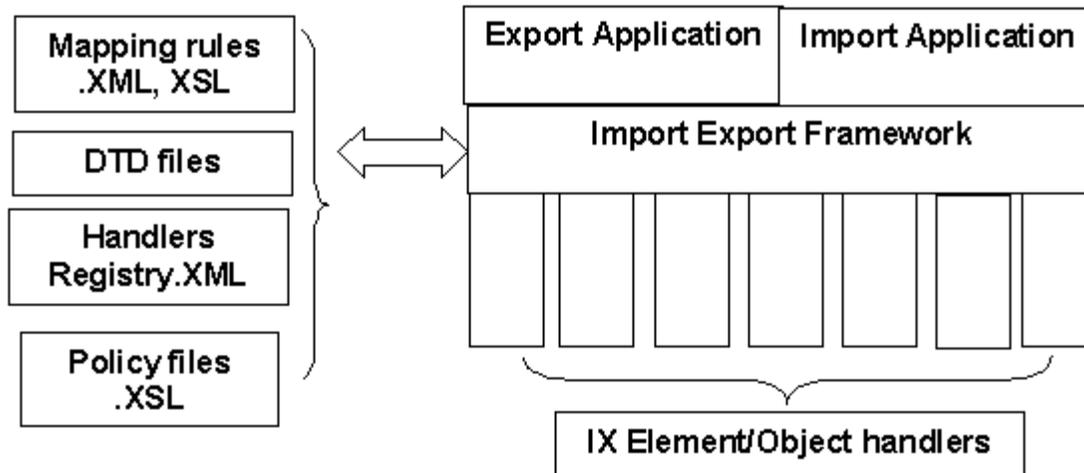
Import Export Framework

The Import Export Chapter describes the IX Framework and explains how to customize and use it for various solutions.

Topic	Page
Overview	12-2
How to Write an IX Application	12-2
Exporter Class	12-4
Using the Exporter to Export Objects	12-7
How Import Works.....	12-8
Importer class	12-11
Use Importer to import object from XML files.....	12-13
Writing Export Handlers for the Export Process.....	12-16
How to write Exp/Imp Handlers	12-16
DTD Files	12-18
How to Write Handlers for the Import Process	12-21
How to Write a Class (Element) Import Handler.....	12-22
Navigating Through an Object's Structure with ObjectSet Application.....	12-28
List of Existing Generators and Filters.....	12-33
Examples	12-33
Simple Export Handler Code:	12-39

Overview

The basic unit of job for the framework is importing or exporting a single object. The framework understands the transactional nature of import and encapsulates a session of individual imports into one database transaction.



How to Write an IX Application

The export starts with Export application. The code of the application that invokes export should look as follows:

```
ExportHandler appHandler = new ExportHandler ();
Exporter exporter = IxbHelper.newExporter(handle,
                                         IxbHelper.STANDARD_DTD,
                                         clientSettingsElement,
                                         policyFile==null?(File)null:policyFile.getFile());

Iterator iter = objectSet.iterator();
while (iter.hasNext()) {
    Persistable ob = (Persistable)iter.next();
    exporter.doExport(ob);
}
exporter.finalizeExport();
appHandler.cleanUp ();
```

Create an Application Export Handler 'appHandler'. This is an instance of a class either implementing ApplicationExportHandler interface or extending the abstract class ApplicationExportHandlerTemplate. In the export application in StandardIXBService, the appHandler extends ApplicationExportHandlerForJar, a subclass of ApplicationExportHandlerTemplate

The job of the 'appHandler' is:

- To create a file to store exported objects (e.g. a JAR file).
- To store logs to be sent back to the client (optional).
- To clean up temporary files and do other clean-up jobs (advised, but optional).

To create, the following methods must be implemented in 'appHandler':

- `storeLogMessage(...)` methods are used to send logs back to clients. It is up to the developer how to implement the mechanism to send the logs back. If you do not want to send any log messages, make your Export Handler extend `ApplicationExportHandlerTemplate`. This class has the default `storeLogMessage()` (empty method).
- It is optional to have clean up and other concluding tasks here, and these jobs must be called explicitly after `exporter.finalizeExport()`.

The Application Export Handler may also contain methods to perform tasks of transforming the output if the application needs to modify the exported XML. For example, `PDXExportHandler` has methods for XSL transformation to PDX format. These methods must be called explicitly after `exporter.finalizeExport()`.

The existing implementations of the Application Export Handler are:

- `WCXMLExportHandler` extends `ApplicationExportHandlerTemplate`. This is a public class designed for creating export jar file on server. (It is mostly a copy of the inner class `ExportHandler` in `StandardIXBService`.) This handler will be removed in 7.0, since now we have `ApplicationExportHandlerForJar`.
- `PDXExportHandler` extends `ApplicationExportHandlerTemplate`. This class performs specific tasks connected with export to PDX format. This includes creating additional XML attributes/elements and XSL transformation to PDX format.

Create an instance of the `Exporter` class, and use it to export objects by calling `exporter.doExport(obj)`, where `obj` runs through all WT objects collected for export.

After this, call `exporter.finalizeExport()`, perform any additional tasks (for example, transformation to another format), call methods of `appHandler` to clean up after the export process, send log messages to client.

The methods `doExport(...)`, `doExportImpl(...)` and the inner class `ExportHandler` in `StandardIXBService` are examples of one export application. Please see the details in the section, "Example of an Export Application" later in this chapter.

Exporter Class

The details of the exporter class are as follows:

Definition:

```
public class Exporter extends ExpImporter{...};
```

Constructor:

```
Exporter (ApplicationExportHandler _applicationExportHandler,  
         String targetDTD,  
         IxbElement localMappingRules,  
         File policyRuleFile) throws WTEException {  
    super ("export", localMappingRules);  
  
    // -- init expActionTuner --  
    applicationExportHandler = _applicationExportHandler;  
    dtd                      = targetDTD;  
    // -- init expActionTuner --  
    expActionTuner = new ExportActionTuner (policyRuleFile);  
}
```

An explanation of the arguments follows:

`_applicationExportHandler` - an instance of any class that either implements the interface `ApplicationExportHandler`, extends the abstract class `ApplicationExportHandlerTemplate` or extends the abstract class `ApplicationExportHandlerForJar`.

The class `ApplicationExportHandlerForJar` extends the class `ApplicationExportHandlerTemplate`. The class `ApplicationExportHandlerForJar` provides methods for storing XML and content files in export jar file. It handles both `ApplicationData` content and content residing in local file system.

`_applicationExportHandler` has a job of creating a Jar file (or any other way of storing) of the resulting collection of XML pieces (exported objects). It must implement two methods:

```
storeContent (ApplicationDataobj);  
storeDocument (IxbElement elem);
```

`targetDTD`: string that specifies what DTD must be used for export process. The IX framework will find appropriate handlers for objects and Document Type Definition for objects based on this DTD string. Currently, there are two DTD strings that are used in Windchill 6.2.2:

`standard.dtd`: DTD for Windchill 6.0

`standard62.dtd`: DTD for Windchill 6.2

Generally the intent was to be able to export objects in any DTD. As you will see below the class export handlers are resolved using the DTD identifier.

The string `targetDTD` is also written to the XML file of exported objects, so the import process could know what DTD should be used to import objects. (This feature is available only from Windchill R6.2.2 and beyond).

`localMappingRules`: XML file or XSL file that is used to override, change or exclude certain attributes objects when the export process takes place.

The following XML rule file overrides the Team Template attribute, and no matter what team an object belonged to when it was exported, its team template attribute will be “Change Team” in the “/System” domain.

```
<?xml version="1.0" encoding="UTF-8"?>
<userSettings>
  <mappingRules>
    <COPY_AS>
      <tag>teamIdentity</tag>
      <value>*</value>
      <newValue>Change Team (/System)</newValue>
    </COPY_AS>
  </mappingRules>
</userSettings>
```

The XSL rule file tests if the exported object has the name of “part_c”, it will override the Team Template attribute and version information, and tests if the exported object has the name of “PART_B”, it will override the Team Template attribute.

If you don’t want to override anything, just pass “null” for the argument `localMappingRules`.

`policyRuleFile`: XSL file that is used to override, change or exclude certain attributes objects when the export process takes place.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="WTPart">
    <xsl:choose>
      <xsl:when test="name='part_c'">
        <newInfo>
          <teamIdentity>Default (/System)</teamIdentity>
          <folderPath>/Design</folderPath>
          <versionInfo>
            <versionId>B</versionId>
            <iterationId>2</iterationId>
            <versionLevel>1</versionLevel>
          </versionInfo>
        </newInfo>
      </xsl:when>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

```

        </newInfo>
    </xsl:when>
    <xsl:when test="number='PART_B'">
        <newInfo>
            <teamIdentity>Default (/System)</teamIdentity>
            <folderPath>/Design</folderPath>
        </newInfo>
    </xsl:when>
    <xsl:otherwise>
    </xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>

```

For example the policy rule file specifies that check out exported WTPart objects in the database after exporting them. If an exported WTDocument object has the number of “TESTDOC-1”, check it out after exporting it. With all other exported WTDocuments, lock them in the database after exporting them.

```

<?xml version="1.0"?>
  <xsl:stylesheet xmlns:xsl=http://www.w3.org/1999/XSL/Transform
    version="1.0">
    <xsl:template match='*'>
      <xsl:apply-templates select='WTPart' />
      <xsl:apply-templates select='WTDocument' />
    </xsl:template>

    <xsl:template match='WTPart'>
      <ACTION>Checkout</ACTION>
    </xsl:template>

    <xsl:template match='WTDocument'>
      <xsl:choose>
        <xsl:when test="number='TESTDOC-1'">
          <ACTION>Checkout</ACTION>
        </xsl:when>
        <xsl:otherwise>
          <ACTION>Lock</ACTION>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>

  </xsl:stylesheet>

```

If you don’t want to override anything, just pass “null” for the argument policyRuleFile.

An instance of the Exporter must be created via the factory method newExporter() of the class IxbHelper. For example:

```

Exporter exporter = IxbHelper.newExporter (
    appHandler,
    IxbHelper.STANDARD_DTD,
    localMappingRules,
    policyRuleFile);

```

Using the Exporter to Export Objects

After you create an instance of the class `Exporter`, for example `exporter`, you can use it to export top-level objects. The call to export the object 'obj' would be `exporter.doExport(obj);`

This is actually making a call to the method `doExport (Object obj, String targetDTD, String targetElem)` in the class `Exporter`.

In this method, the real Export handler for the object `obj` will be created.

Note: A list of available export handlers is created based on XML files in the folder `<WC_home>\registry\ixb\handlers`. If you pass a wrong DTD in the constructor of `Exporter` (a DTD that is not available in the system), you will not get the handler, so you cannot export the object. Please refer to the part "How to write Export Handler" for information how to add entry for an Export handler to XML files.

If you have more than one object, you have to pass them to the exporter one by one. Let's assume that those objects are in a set called `res`, then you can export them like this:

```
Iterator iter = res.iterator();
while (iter.hasNext()) {
    Persistable obj = (Persistable)iter.next();
    exporter.doExport(obj);
}
```

After you export all objects, you must call `exporter.finalizeExport();`

You can call clean-up methods of `appHandler` (if there are any). Now the export is finished.

Note: A sample file with comments is distributed along with installation information.

How Import Works

To import objects from XML files in a jar file, import application must do the following:

1. Create a tool to read the imported jar file and extract the list of XML files that are contained in the jar file. Please see the `doImportImp(...)` method in the `StandardIXBService` and the class `IXBJarReader` for an implementation example.
2. Prepare the String `ruleFileName` to be passed into an instance of the class `Importer`. The String `ruleFileName` can be obtained from an `IxbStreamer`, from the user, assigned the value `null` or obtained from somewhere else, depending on the structure of the import application.
3. Process `policyFile` (if it is not `null`) to create an XSL `StreamSource` that contains the import policy.
4. Create an Application Import Handler `appHandler`. This is an instance of any class that either implements the interface `ApplicationImportHandler` or extends the abstract class `ApplicationImportHandlerTemplate`.
5. Create an instance of the class `Importer` (`importer`).
6. Get the list of XML files in the jar file.
7. Create `IxbDocuments` from those XML files.
8. With each `IxbDocument`, do the following:
 - If there is an action name passed to the application and the `policyFile` is `null`, apply the action name into the `IxbDocument`. If the `policyFile` is not `null`, apply action name and action information in the `policyFile` into the `IxbDocument`.
 - Feed them to the `importer` one by one by calling the import process:
 - `importer.doImport(IxbDocument Doc);`
 - `importer.finalizeImport();`
9. Clean up (if needed).
10. Send log messages to client.

The methods `doImport(...)`, `doImportImpl(...)` and the inner class `ImportHandler` in the `StandardIXBService` are an example of one import application. Please see the details in the part “Example of an Import Application”.

From Windchill 6.2.6, versioned objects can be imported in any of 10 different manners decided by action name and action information that are written to the IxbDocument fileXML of each importing object. Developers who write import applications must know about action names and their meanings to apply them correctly, but object handlers don't have to worry about the Actor classes. A list of all available action names can be found in the file Windchill\src\wt\ixb\registry\ixb\handlers\actor.xml.

All of the actions are different from each other in 3 crucial methods: previewObject, createObject and storeObject. In the class ExpImpForVersionedObject, based on action name and action information that are passed into the IxbDocument fileXML, appropriate actor will be created and this actor's methods will be called to serve the purpose of previewing, creating and storing versioned objects.

Here is the list by actor names for information.

1. PickExistingObject: Find if an object with the same ufid or same (name, number, version, iteration) with the object in XML file exists in database. If such an object exists, do nothing. Otherwise, import the object in XML file.
2. NewIteration: Import object in XML file as the next available iteration in the database.
 - For example: If there is no version/iteration in the database for the object which is in the XML file, the imported object will get the version / iteration specified in the XML file. If the latest version / iteration of the object in the database is B.2, the imported object will be B.3.
3. NewVersion: Import objects from the XML file as the next available version in the database.
 - For example: If there is no version / iteration in the database for the object which is in the XML file, the imported object will get the version / iteration specified in the XML file. If the latest version / iteration of the object in the database is B.2, the imported object will be C.1.
4. CheckOut: Find any version/iteration of the object in the XML file (Check the existence of the master object in the database). If there is no version of the object in the XML file, throw an error. Otherwise, find an instance of the object in the database that has the same version (iteration can be different) as the object in the XML file. If such an object exists, check out the latest iteration of the object in the database, update it with information from the XML file. I agree Otherwise, throw an error. No, we don't check it in
5. ImportNonVersionedAttr: Find an object with the same ufid or same (name, number, version, iteration) with the object in the XML file. If such an object exists, update it with information from the XML file. Otherwise, throw an error.

6. `UpdateInPlace`: Find an object with the same `ufid` or same (name, number, version, iteration) with the object in XML file exists in database. If such an object exists AND it is checked out, update it with information from the XML file. Otherwise, throw an error.
7. `UnlockAndIterate`: Find an object in the database with the same `ufid` or same (name, number, version, iteration) as the object in the XML file. If such an object exists AND it is locked, unlock and iterate it, then update it with information from the XML file. Otherwise, throw an error.
8. `CreateNewObject`: Create a brand new object with new name, new number, new version, new iteration provided in Import Policy file. Other information will be extracted from the XML file. This functionality cannot be used alone,

Note: This option cannot work without a policy file to specify the new object identities.

The format of new information that must be provided in ImportPolicy file is:

```
<actionInfo>
  <xsl:choose>
    <xsl:when test="criteria='value'">
      <action>CreateNewObject</action>
      <actionParams>
        <newName>New Name</newName>
        <newNumber>New Number</newNumber>
        <newVersion>New Version</newVersion>
        <newIteration>New Iteration</newIteration>
      </actionParams>
    </xsl:when>
    <xsl:otherwise>
      <action>Some other action</action>
    </xsl:otherwise>
  </xsl:choose>
</actionInfo>
```

Note: `<actionInfo>` must always exist.

- Criteria can be any valid attribute of the object in XML file.
 - Between `<xsl:choose>`, there can be many `<xsl: when test>` with different criteria and different action names.
 - Only `CreateNewObject` and `SubstituteObject` can have action params, and there are only four action params `<newName>`, `<newNumber>`, `<newVersion>`, `<newIteration>`, all of them must be provided.
9. `SubstituteObject`: Substitute the object in the XML file for an object in the database that has the name, number, version, and iteration provided in the ImportPolicy file. If such an object doesn't exist, throw an exception. Format of tag and params for this case is exactly the same with `CreateNewObject`, but the `<action>` is `SubstituteObject`.

10. Ignore: Do not import the object in the XML file. This action doesn't require any actor.

Importer class

Definition: public class Importer extends ExpImporter

Constructor:

```
Importer (ApplicationImportHandler _applicationImportHandler,  
         String _dtd,  
         String _ruleFileName,  
         Boolean _overrideConflicts,  
         Boolean _validate  
        ) throws WTEException
```

Parameters explanation:

- `applicationImportHandler`: an instance of a class that either implements the interface `ApplicationImportHandler` or extends the abstract class `ApplicationImportHandlerTemplate`
- `applicationImportHandler` has a job of extracting from the Jar file that stores XML, and its class must implement 2 methods:

```
getContentAsStream (String contentId);  
getContentAsApplicationData (String contentId);
```

The later method may always return null to indicate that the file does not exist in Windchill DB.

Note: Please see the inner class `ImportHandler` of the class `StandardIXBService` for an example of implementation of an application import handler.

- `targetDTD`: string that specifies what DTD must be used for import process. The IX framework will find appropriate handlers for objects and Document Type Definition for objects based on this DTD string if the imported file does not specify any. Currently, there are two DTD strings that are used in Windchill 6.2.2:

```
standard.dtd: DTD for Windchill 6.0
```

```
standard62.dtd: DTD for Windchill 6.2
```

- `ruleFileName`: From windchill 6.2.6, mapping rule file can be XML file (like in previous versions) or XSL file, so this parameter is String. The constructor that uses `IxbElement _localMappingRules` is deprecated. In the

case you do not have mapping rule file and want to put it to null, please do not put the “null” value directly in the constructor, because it will cause one ambiguous reference error. Instead of doing that, you should use a string, assign null value to it, and pass it as ruleFileName. Mapping rule file is used to change, override or exclude certain attributes objects when the import process takes place.

For example, the rule file overrides the Team Template attribute, and no matter what team an object belonged to when it was exported, its team template attribute is replaced by “Change” in the “/System” Domain on import.

```
<?xml version="1.0" encoding="UTF-8"?>
<userSettings>
  <mappingRules>
    <COPY_AS>
      <tag>teamIdentity</tag>
      <value>*</value>
      <newValue>Change Team (/System)</newValue>
    </COPY_AS>
  </mappingRules>
</userSettings>
```

An example for XSL mapping rule file:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="WTPart">
    <xsl:choose>
      <xsl:when test="name='part_c'">
        <newInfo>
          <teamIdentity>Default (/System)</teamIdentity>
          <folderPath>/Design</folderPath>
          <versionInfo>
            <versionId>B</versionId>
            <iterationId>2</iterationId>
            <versionLevel>1</versionLevel>
          </versionInfo>
        </newInfo>
      </xsl:when>
      <xsl:when test="number='PART_B'">
        <newInfo>
          <teamIdentity>Default (/System)</teamIdentity>
          <folderPath>/Design</folderPath>
        </newInfo>
      </xsl:when>
      <xsl:otherwise>
        </xsl:otherwise>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

This XSL file says that whenever the import process meet a WTPart named part_c, then change its team identity template to Default (/System), change its folder part to /Design, and change its version to B.2, whenever the import process meet a WTPart with the number PART_B, then change its team identity template to Default (/System), change its folder part to /Design

If you don't want to override anything, just pass "null" for the argument localMappingRules.

- `_overrideConflicts`: boolean value specifies whether the import process should override "overridable" conflicts.
- `_validate`: boolean value specifies whether the import process should validate the XML file of the imported object against the DTD.

An instance of the class Importer must be created via the method `newImporter()` of the class `IxbHelper`. For example:

```
Importer importer = IxbHelper.newImporter(  
    handler,  
    IxbHelper.STANDARD_DTD,  
        ruleFileName,  
    overrideConflicts,  
    null);
```

Use Importer to import object from XML files

After you create an instance of the class Importer (`importer`) you can use it to import objects from XML files. If you have more than one XML files to import, you have to give `IxbDocument`-s that created from XML files to the importer one by one.

As mentioned above, the Import Application server must call two methods to do the import:

```
importer.doImport(IxbDocument doc);  
  
importer.finalizeImport();
```

doImport (doc)

This method doesn't really import the object, but inserts the XML document that represents the object in to a list to be imported later. After all XML documents representing all the imported objects are inserted in the import list, the real import process starts with the call to `finalizeImport()`.

- `finalizeImport()`: The import process is actually performed in this method. It will call to:

```
doImport_CheckConflicts() - check conflicts for imported objects.
```

`doImport_doRealImport ()` - do the real import for the list of objects by calling `importElement (IxbElement doc)` for each XML document representing each object in the list.

In the method `importElement (...)` the import handler for a particular type of the object is created by calling `getImportHandler (tag, dtdFromXML)`.

The method `getImportHandler (...)` finds the appropriate handler for the imported object as follows.

1. Try to get an import handler by using the DTD string in the `<dtd>` tag of the XML file of the imported object.
2. If the handler is null, try again using current DTD in the `importer`. This current DTD is calculated using version of the current Windchill. For Windchill R6.0 it is `standard.dtd`. For Windchill R6.2 it is `standard62.dtd`.

After getting the handler for the element, the `importElement (...)` calls the following methods:

`handler.importElement (...)` to do the import task.

`handler.outputLog (...)` to send log to user.

For Windchill 6.2.6, all handlers for non-versioned objects (for example links, ReportTemplate ...) extend the class `ClassExporterImporterTemplate`, and all handlers for versioned objects (for example Parts, Documents, EPMDocuments ...) extend the class `ExpImpForVersionedObject`.

Note: Handlers for non-versioned objects act like the previous versions.

If the real handler doesn't implement the method `importElement (...)`, then the call invokes the default method `importElement (...)` of the class `ClassExporterImporterTemplate`. In this class, the `importElement (...)` method calls to

`findAmongExistingObjects (fileXML, importer);`

If it finds that the object in the XML file currently exists in the database, it will not import the object. Otherwise, it will call the following methods:

```
createObject (fileXML, importer);  
importObjectAttributes (ob, fileXML, importer);  
storeObject (ob, fileXML, importer);  
importObjectAttributesAfterStore (ob, fileXML, importer);
```

Some of these methods should be implemented in the handler, and it is how and when the real handler comes to do its job.

Note: Handlers for versioned objects act different.

If the real handler doesn't implement the method `importElement(...)`, then the call invokes the default method `importElement(...)` of the class `ExpImpForVersionedObject`. In this class, despite the object in the XML file exists in the database or not, the `importElement(...)` method always calls to:

```
createObject (fileXML, importer);
importObjectAttributes (ob, fileXML,
importer);
storeObject (ob, fileXML, importer);
importObjectAttributesAfterStore (ob,
fileXML, importer);
```

Then, the Import Application can do a clean-up, and send messages to the client. The import process is finished.

How to write Exp/Imp Handlers

Writing Export Handlers for the Export Process

To create an export handler for a class, one needs to know three things:

1. How to use XML files that contain handlers information.
2. How to update DTD files that contain information about objects that will be exported/imported.
3. How to write a handler.

XML files that contain handlers information

These XML files reside in the folder Windchill\codebase\registry\ixb\handlers. The whole folder is processed as the root element of XML tags, and the contents of XML files in this folder are processed as child elements. So, the XML files in this folder are not well formed, and they have contents similar to the following:

```
<classExporter>
  <class>
    [class name of the object that will be exported with full path
name]
  </class>
  <dtd>
    [String DTD, specifies where DTD for the class is stored]
  </dtd>
  <targetTag>default</targetTag>
  <handler>
    [class name of the handler that is used to export the object]
  </handler>
</classExporter>
```

For example:

```
<classExporter>
  <class>wt.part.WTPart</class>
  <dtd>standard62.dtd</dtd>
  <targetTag>default</targetTag>
  <handler>wt.ixb.handlers.forclasses.ExpImpForWTPart</handler>
</classExporter>
```

All handlers with the <dtd> tag “standard.dtd” are handlers for export object in R6.0. All handlers with the <dtd> tag “standard62.dtd” are handlers for export object in R6.2.

For example, to be able to export data in R6.0 legacy format, we have tags:

```
<classExporter>
  <class>wt.part.WTPart</class>
  <dtd>standard.dtd</dtd>
  <targetTag>default</targetTag>
  <handler>wt.ixb.handlers.forclasses.ExpImpForWTPart60</handler>
</classExporter>
```

For R6.2 we have tags:

```
<classExporter>
  <class>wt.part.WTPart</class>
  <dtd>standard62.dtd</dtd>
  <targetTag>default</targetTag>
  <handler>wt.ixb.handlers.forclasses.ExpImpForWTPart</handler>
</classExporter>
```

So we know that the class

wt.ixb.handlers.forclasses.ExpImpForWTPart60 is handler for export of the class wt.part.WTPart in R6.0 legacy format, and the class wt.ixb.handlers.forclasses.ExpImpForWTPart is a handler for exporting the class wt.part.WTPart in R6.2 format.

It is possible that one handler can be used for export in both R6.0 and R6.2. For example, in Windchill\src\wt\ixb\registry\ixb\handlers\core.xml, there are tags:

```
<classExporter>
  <class>wt.part.WTPartUsageLink</class>
  <dtd>standard.dtd</dtd>
  <targetTag>default</targetTag>
  <handler>
    wt.ixb.handlers.forclasses.ExpImpForWTPartUsageLink
  </handler>
</classExporter>
```

and in Windchill\src\wt\ixb\registry\ixb\handlers\core62.xml, there are tags:

```
<classExporter>
  <class>wt.part.WTPartUsageLink</class>
  <dtd>standard62.dtd</dtd>
  <targetTag>default</targetTag>
  <handler>
    wt.ixb.handlers.forclasses.ExpImpForWTPartUsageLink
  </handler>
</classExporter>
```

Thus we know that the handler `wt.ixb.handlers.forclasses.ExpImpForWTPartUsageLink` will be used for export of the class `wt.part.WTPartUsageLink` in both R6.0 and R6.2.

The class `IxbHandlersManager` contains all necessary methods to manipulate handlers.

DTD Files

For backward compatible support, in the folder `Windchill\src\wt\ixb\registry\ixb\dtds`, there are 2 folders:

`standard.dtd` contains DTD files for Windchill 6.0

`standard62.dtd` contains DTD files for Windchill 6.2

In each folders there is a file named `coreobjects.dtd` that is the DTD file for all objects that will be exported/imported. For example, in the file `coreobjects.dtd` in the folder `standard.dtd`, we can see the definition for `EPMDocument` in R6.0:

```
<!ELEMENT EPMDocument (
  ObjectID,
  (ownerApplication , authoringApplication, description?) ,
  (number, name ) , epmDocType , genericFamilyInstance? ,
  (extentsValid , EPMBBoxExtents)? ,
  folderPath , versionInfo? , lifecycleInfo? , projectIdentity ,
  contentItem*, iba* ) >
```

In the file `coreobjects.dtd` in the folder `standard62.dtd`, we can see the definition for `EPMDocument` in R6.2:

```
<!ELEMENT EPMDocument (
  ObjectID,
  (ownerApplication , authoringApplication, description?) ,
  (number, name ,docType, docSubType, CADName) ,
  (extentsValid , EPMBBoxExtents)? ,
  folderPath , versionInfo? , lifecycleInfo? , teamIdentity ,
  contentItem*, iba* ) >
```

How to Write a Class Export Handler

- a. Create a Class that extends `ClassExporterImporterTemplate`
- b. Implement `exportAttributes(Object obj, Exporter exp)` method, which retrieves the data from the object and adds it to the XML DOM Document. The following is an example of this method for the object of "MyClass":

```
public void exportAttributes (Object object, Exporter exporter)
throws WTEException {
    try {

        MyClass ob = (MyClass)object;
        // export the local id
        IxbHndHelper.exportAttribute(
            ExpImpForLocalIdAttr.class, ob, fileXML, exporter);
        // export other attributes that are specific to
        // MyObject; e.g. name, number
        IxbHndHelper.exportAttribute(
            ExpImpForMyObjectAttr.class, ob, fileXML, exporter);
        // export version information
        IxbHndHelper.exportAttribute(
            ExpImpForVersionAttr.class, ob, fileXML, exporter);
        // export content
        IxbHndHelper.exportAttribute(
            ExpImpForContentAttr.class, ob, fileXML, exporter);

    }
    catch (Exception e) {
        LogHelper.devExc ( e,
            "exportAttributes: could not export
object=<"+object+">");
    }
}
```

- c. Add an entry in the handlers XML file (`<WC_home>\registry\ixb\handlers\core.xml` or `core62.xml`) that specifies the class being exported (`com.mycompany.MyObject`), the XML DTD for core Windchill objects (`standard.dtd` or `standard62.dtd`), and the handler for the class (`wt.ixb.handlers.forclasses.ExpImpForMyObject`). An example entry follows:

```
<classExporter>
  <class>com.mycompany.MyObject</class>
  <dtd>standard.dtd</dtd>
  <targetTag>default</targetTag>
  <handler>wt.ixb.handlers.forclasses.ExpImpForMyObject</handler>
</classExporter>
```

or

```
<classExporter>
  <class>com.mycompany.MyObject</class>
  <dtd>standard62.dtd</dtd>
```

```

<targetTag>default</targetTag>
<handler>wt.ixb.handlers.forclasses.ExpImpForMyObject</handler>
</classExporter>

```

How to Write an Attribute Export Handler

If there is an attribute that is required to be exported the same way for different classes or if you simply decide to handle it is a separate handler, you can create an attribute handler. The steps to follow are:

- a. Create a Java class extending `AttrExporterImporterTemplate`.
- b. Implement `exportAttribute(Object ob, IxbElement fileXML, Exporter exporter)` method, which retrieves the attribute data from the object and adds it to the XML DOM Document. The following is an example of this method for the object "MyObject". This method gets the part type and the part source of the object.

```

c. public void exportAttribute (
    Object obj,
    IxbElement fileXML,
    Exporter exporter) throws WTEException{
try {
    MyClass ob = (MyClass) obj;
    LocalizableMessage localMessage1 = ob.getDisplayType();
    Locale locale1 = new Locale("English", "US");
    String dispType = localMessage1.getLocalizedMessage(locale1);
    fileXML.addValue(IxbHndHelper.XML_ATTR_PARTTYPE, dispType);
    fileXML.addValue(
        IxbHndHelper.XML_ATTR_SOURCE, ob.getSource ().toString() );
}
catch (Exception e){
    LogHelper.devExc (e,
        "Exception in ExpImpForLTPartAttr, ob=<"+obj+">");
}
}

```

After adding this, the export handler for class may call this method to have the part attribute exported.

How to Write Handlers for the Import Process

To create import handlers for a class, there are two things to know:

1. XML files that contain handlers information.
2. How to write a handler.

XML files that contain handlers information

These XML files are in the folder `Windchill\codebase\registry\ixb\handlers`.

The whole folder is processed as the root element of XML tags, and the contents of XML files in this folder are processed as child elements. So the XML files in this folder are not well formed, and their contents are similar to the following:

```
<elementImporter>
  <tag>
    [class name of the object that will be imported without full
    path name ]
  </tag>
  <dtd>
    [String DTD, specifies where DTD for the class is stored]
  </dtd>
  <handler>
    [class name of the handler that is used to import the object]
  </handler>
</elementImporter>
```

For example:

```
<elementImporter>
  <tag>WTPart</tag>
  <dtd>standard62.dtd</dtd>
  <handler>wt.ixb.handlers.forclasses.ExpImpForWTPart</handler>
</elementImporter>
```

All handlers with the `<dtd>` tag “`standard.dtd`” are handlers for import object in R6.0. All handlers with the `<dtd>` tag “`standard62.dtd`” are handlers for import object in R6.2.

For example, to import parts from XML that came from R6.0, we have tags:

```
<elementImporter>
  <tag>WTPart</tag>
  <dtd>standard.dtd</dtd>
  <handler>wt.ixb.handlers.forclasses.ExpImpForWTPart60</handler>
</elementImporter>
```

For R6.2 format we have tags:

```
<elementImporter>
  <tag>WTPart</tag>
  <dtd>standard62.dtd</dtd>
```

```
<handler>wt.ixb.handlers.forclasses.ExpImpForWTPart</handler>
</elementImporter>
```

So we know that the class

`wt.ixb.handlers.forclasses.ExpImpForWTPart60` is handler for import of the class `wt.part.WTPart` that comes in R6.0 format, and the class `wt.ixb.handlers.forclasses.ExpImpForWTPart` is handler for import of the class `wt.part.WTPart` in R6.2.

It is possible that one handler can be used for import in both R6.0 and R6.2 formatted data. For example, in `Windchill\codebase\registry\ixb\handlers\core.xml`, there are the following tags:

```
<elementImporter>
  <tag>WTPartUsageLink</tag>
  <dtd>standard.dtd</dtd>
  <handler>
    wt.ixb.handlers.forclasses.ExpImpForWTPartUsageLink
  </handler>
</elementImporter>
```

In `Windchill\codebase\registry\ixb\handlers\core62.xml`, there are the following tags:

```
<elementImporter>
  <tag>WTPartUsageLink</tag>
  <dtd>standard62.dtd</dtd>
  <handler>
    wt.ixb.handlers.forclasses.ExpImpForWTPartUsageLink
  </handler>
</elementImporter>
```

We know that the handler

`wt.ixb.handlers.forclasses.ExpImpForWTPartUsageLink` will be used for import of the class `wt.part.WTPartUsageLink` in both R6.0 and R6.2 formats.

The class `IxbHandlersManager` contains all methods to manipulate handlers.

How to Write a Class (Element) Import Handler

Import handlers for classes can be divided in to two types:

- Import handlers for versioned objects.
- Import handlers for non-versioned objects.

Handler for Non-Versioned Object

- a. Create a Java class that extends `ClassExporterImporterTemplate`
- b. Implement method:

```
public Object createObject (IxbElement fileXML, Importer importer)
```

- c. Override the following methods if necessary:

```
public Object importObjectAttributes (Object ob,
                                     IxbElement fileXML,
                                     Importer importer);

public Object storeObject (Object object,
                           IxbElement fileXML,
                           Importer importer);

public Object importObjectAttributesAfterStore(Object object,
                                               IxbElement fileXML,
                                               Importer importer);

public Object findAmongExistingObjects (IxbElement fileXML,
                                       Importer importer);
```

Import handlers for non-versioned objects are quite straightforward, and any of the following classes can be referred as example:

```
ExpImpForWTPartDescribeLink
ExpImpForWTPartReferenceLink
ExpImpForWTPartUsageLink
```

Note: An object is imported by the following sequence: `createObject()`, `importObjectAttributes()`, `storeObject()`, `importObjectAttributesAfterStore()`.

`createObject()`: if the object doesn't exist in database (is not found by `findAmongExistingObjects`), it will be created, and all the Ufid attribute of the object is imported.

`importObjectAttributes()`: import all attributes of the object

`storeObject()`: the method `storeObject()` will call to `PersistenceHelper.manager.store()`.

`importObjectAttributesAfterStore()` imports all attributes that must be imported after the object is stored.

- d. Add an entry to the handlers registry file (`<WC_home>\registry\ixb\handlers\core.xml` or `core62.xml`). The entry specifies the class being imported (`MyObject`), XML DTD for core Windchill objects (`standard.dtd` or `standard62.dtd`), and the handler for the class

`com.ptc.mypackage.ExpImpForMyObject`. An example entry follows:

```
<elementImporter>
  <tag>MyObject</tag>
  <dtd>standard.dtd</dtd>
  <handler>com.ptc.mypackage.ExpImpForMyObject</handler>
</ elementImporter >
or
<elementImporter>
  <tag>MyObject</tag>
  <dtd>standard62.dtd</dtd>
  <handler>com.ptc.mypackage.ExpImpForMyObject</handler>
</ elementImporter >
```

Handler for Versioned Object

- a. Create a Java class that extends `ExpImpVersionedObject`
- b. Implement the following methods:
 - `public Mastered getMaster (IxbElement fileXML):` returns the Object Master if there is any version of the importing object in the database, otherwise returns null.
 - `public Versioned createNewObject (IxbElement fileXML,Importer importer):` create a new object with information from the XML file.

From Windchill 6.2.6, import handlers for versioned objects don't have to implement the method `public Object createObject(...)` anymore. It is implemented in the class `ExpImpForVersionedObject`.

- c. Override the following methods if necessary:

```
public Object importObjectAttributes (Object ob,
                                     IxbElement fileXML,
                                     Importer importer);

public Object importObjectAttributesAfterStore(Object object,
                                               IxbElement fileXML,
                                               Importer importer);

public Object findAmongExistingObjects (IxbElement fileXML,
                                       Importer importer);
```

For Windchill 6.2.6, import handlers for versioned objects don't have to implement the method `public Object storeObject (...)` anymore. It is implemented in the class `ExpImpForVersionedObject`.

Note: An object is imported by the following sequence: `createObject()`, `importObjectAttributes()`, `storeObject()`, `importObjectAttributesAfterStore()`.

createObject(): From version 6.2.6, this method is implemented in the class `ExpImpForVersionedObject`. The creation of objects will be delegated to Actor classes, depends on the actor name in the XML file (The actor name is written into the XML file by import application). Particular object handlers don't have to worry about `createObject()` and Actor.

importObjectAttributes(): import all attributes of the object that can be imported before the object is stored.

storeObject(): From version 6.2.6, this method is implemented in the class `ExpImpForVersionedObject`. The store of objects will be delegated to Actor classes, depends on the actor name in the XML file (The actor name is written into the XML file by import application). Particular object handlers don't have to worry about `storeObject()` and Actor.

importObjectAttributesAfterStore(): import all attributes of the object that must be imported after the object is stored.

`ExpImpForWTPart`, `ExpImpForDocument`, `ExpImpForEPMDocument` are examples of import handlers for versioned objects.

- d. Add an entry to the handlers registry file (`<WC_home>\registry\ixb\handlers\core.xml` or `core62.xml`). The entry specifies the class being imported (`MyObject`), XML DTD for core Windchill objects (`standard.dtd` or `standard62.dtd`), and the handler for the class `com.ptc.mypackage.ExpImpForMyObject`. An example entry follows:

```
<elementImporter>
  <tag>MyObject</tag>
  <dtd>standard.dtd</dtd>
  <handler>com.ptc.mypackage.ExpImpForMyObject</handler>
</ elementImporter >
```

or

```
<elementImporter>
  <tag>MyObject</tag>
  <dtd>standard62.dtd</dtd>
  <handler>com.ptc.mypackage.ExpImpForMyObject</handler>
</ elementImporter >
```

How to Write an Attribute Import Handler

If there is an attribute that is required to be imported the same way for different classes or if you simply decide to handle it as a separate handler, you can create an attribute handler. The steps to follow are:

- a. Create a Java class that extends `AttrExporterImporterTemplate`.
- b. Override the following methods if needed:

`prepareForCheckConflicts(Importer importer)`: prepares for conflict checking. It is likely never be implemented by a particular handler.

`checkConflictForAttribute(Object existingOb,`

`IxbElement fileXML,`

`Importer importer)` :

This method does the conflict checking for particular attribute, so if the imported attribute can potentially have conflicts with an attribute that exists in the database, this method must be overridden.

```
importAttribute (Object object,  
IxbElement fileXML,  
Importer importer):
```

Retrieves the attribute data from the XML DOM Document and set it to the imported object. This method must be overridden to suit particular attribute.

Here is an example for `importAttribute()` for the attribute `MyAttr` to the object `MyObject`:

```
public Object importAttribute (Object object,
                               IxbElement fileXML,
                               Importer importer)
    throws WTEException{
    String myAttr;
    try{
        myAttr = fileXML.getValue(IxbHndHelper.XML_ATTR_MYATTR);
        // XML_ATTR_MYATTR tag must be defined in IxbHndHelper
    }
    catch (Exception exc){
        // The paragraph bellows allows the import process continue,
        // even when the import of MyAttr fails. If the programmer
        // wants the import process to stop when the import of
        // MyAttr fails, please assign ob=null and throw exception
        System.out.println(
            "Exception when getting MyAttr in importAttribute");
        System.out.println("MyAttr attribute is not imported");
        return object;
    }

    MyObject ob = (MyObject) object;
    try {
        MyObjectHelper.service.setMyAttr(ob, myAttr);
    }
    catch (Exception e) {
        if (! importer.
            attributeExporterImporterManager.
                overrideConflicts) {
            ob = null;
            throw e;
        }
        else{
            // override the conflict by doing something here...
        }
    }
    finally{
        return ob;
    }
}
```

Navigating Through an Object's Structure with ObjectSet Application

When an object is given to the export process, ObjectSet application will do the job of navigating through the object's structure and collecting all its related objects. For example, when one WTPart is given to the export process, the ObjectSet application will navigate through the product structure to see all parts that are used by it, all "referenced" documents and the "described-by" document.

The definition of the navigation is taken from a set of XML files known as navigation rule files. (Don't confuse with the mapping rule files that are used to override/change/exclude object's attributes in Export and Import processes). Additionally the application uses Java classes known as generators and filters to collect a set of objects that will be exported when simple navigation is not enough and some programming logic needs to be applied. The navigation rule files reside in the folder `Windchill\codebase\registry\ixb\object_set_handlers`.

There are two types of navigating rules: Generator and Filter.

- Generators are rules that are used by application to traverse through the object's structure and get all of its objects to be exported, such as "uses", "described by", "reference", etc.
- Filters are rules that will be applied to objects to be exported to exclude certain objects from the export process. For example, with "Filter By Time", we can choose to export objects that modified during a specified period; with "Filter By Object Number", we can choose not to export objects with specific numbers.

Combining these Generators and Filters together, the application that wants to export object will be able to create a set of objects that will be exported based on given top-level object.

Available GeneratorIds are defined in XML files in the folder `Windchill\codebase\registry\ixb\object_set_handlers` with the tag `<setGenerator>`. A list of these Generator Ids can be obtained by calling to `IXBHelper.service.getAllAvailableGenerators()`. The call returns all available Generators in the system.

`IXBHelper.service.getGeneratorList()` returns a list of Generators that will be displayed to the GUI for user selection. This helps to hide Generators that you don't want the end-user to see. To hide such Generators, the XML files of these Generators, should have the `<display>` tag set to false.

For example: a paragraph of XML file for WTPart:

(From “<Windchill>\codebase\registry\ixb\object_set_handlers\
product_struct.xml”)

```
<setGenerator>
  <id>productStructureNavigator</id>
  <handler>
wt.ixb.objectset.handlers.navigator.ProductStructureNavigator
  </handler>
  <dialogClassName>
wt.clients.ixb.exp.NavigatorSearchDialog
  </dialogClassName>
  <localizedName>
  <localizedString>
  <class>wt.ixb.objectset.objectSetResource</class>
  <key>PRODUCT_STRUCTURE_NAME</key>
  </localizedString>
  </localizedName>
```

Tags

<id>: Generator Id

<handler>: Navigator – Java class that helps navigating through the object structure. In the example, to navigate through WTPart structure.

<dialogClassName> : Java class of the dialog that must be called from the GUI to search the top-level object of this class in database (in this example, to search WTPart).

<localizedName> and its sub tags are for internationalization purpose. The string resource for displaying the Generator to the GUI will be defined in the .rbInfo file specified by localizedName/localizedString/class and its key in the .rbInfo file is localizedName/localizedString/key.

If you don't want this GeneratorId to be displayed to the GUI, but only to be used programmatically in your application, add the tag <display> like this:

```
<setGenerator>
  <id>productStructureNavigator</id>
  <display>false</display>
  <handler>
wt.ixb.objectset.handlers.navigator.ProductStructureNavigator
  </handler>
  <dialogClassName>
wt.clients.ixb.exp.NavigatorSearchDialog
  </dialogClassName>
  <localizedName>
  <localizedString>
  <class>wt.ixb.objectset.objectSetResource</class>
  <key>PRODUCT_STRUCTURE_NAME</key>
```

```
</localizedString>
</localizedName>
```

Available Filter Id-s are defined in XML files in the folder Windchill\codebase\registry\ixb\object_set_handlers with the tag <setFilter>. A list of these Filter Ids can be obtained by calling to IXBHelper.service.getAllAvaivableFilters(). It returns all available Filters in the system.

IXBHelper.service.getFilterList() returns a list of Filters that will be displayed to the GUI for user selection. This function help to hide filters which you don't want the end user to see. To hide such Filters, set the value of the <display> tag to false in the XML files of these Filters.

If the tag <display> is not specified, or set to true, the generator will be included in the result of the method ObjectSetHelper.getListOfObjectSetGenerators() and it will be displayed in the GUI. If the tag <display> is false, the generator will not be included in the result of the method ObjectSetHelper.getListOfObjectSetGenerators(), and it will not be displayed to the GUI. To get all generators in the system, please use the method ObjectSetHelper.getAllAvaivableGenerators().

All the methods get...Generators and get...Filters return Vectors that contain an element list of the type IXBHandlerDescription. Use getId() of those elements to get lists of Generators or Filters that are passed as arguments generatorIds and filterIds for the method doExport() in the StandardIXBService.

Object Navigation

The mechanism is an XML-rule-driven “navigator” of Windchill objects. Given a seed (top level) object the mechanism uses specified rules to navigate from object to object. The navigation can be performed through a DB link, a foreign key, or a specified method. Here is an example of rule definition of WTPart. The seed is a Folder object.

(From “<Windchill>\codebase\registry\ixb\object_set_handlers\product_struct.xml”)

```
<handler>
wt.ixb.objectset.handlers.navigator.ProductStructureNavigator
</handler>
...
<schema>
...
<rule>
  <for>wt.part.WTPart</for>
  <go>
    <byMethod>
```

```

        <method>navigateFromWTPartToDescribeByDoc</method>
    </byMethod>
</go>
</rule>

<rule>
    <for>wt.part.WTPartDescribeLink</for>
    <go>
        <fromForeignKey>
            <targetClass>wt.doc.WTDocument</targetClass>
            <methodName>getDescribedBy</methodName>
        </fromForeignKey>
    </go>
</rule>
...
</schema>

```

The example above shows both possible types of navigation: From `WTPart` it instructs to navigate to the `wt.part.WTPartDescribeLink` by a `navigate` method and from there using method `getDescribedBy` to get the `WTDocument` that the `WTPart` is described by. Then, non-trivial semantic steps can be made.

After collecting, the objects can be filtered out by a set of defined filters. The filter definition is stored in the same object set registry. Here is an example of a date/time filter:

(File: "`<Windchill>\codebase\registry\ixb\object_set_handlers\filter_by_time.xml`")

```

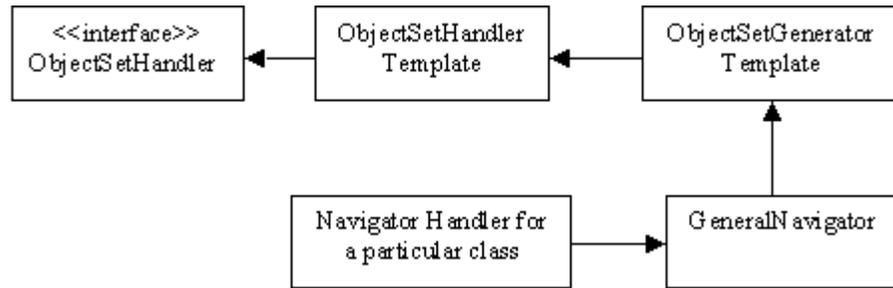
<setFilter>
    <id>filterByTime</id>
    <handler>wt.ixb.objectset.handlers.FilterByTime</handler>
    <dialogClassName>
wt.clients.ixb.exp.FilterByTimeDialog
    </dialogClassName>
    <localizedName>
        <localizedString>
            <class>wt.ixb.objectset.objectSetResource</class>
            <key>FILTER_BY_TIME_NAME</key>
        </localizedString>
    </localizedName>
    <parameters>
    </parameters>
</setFilter>

```

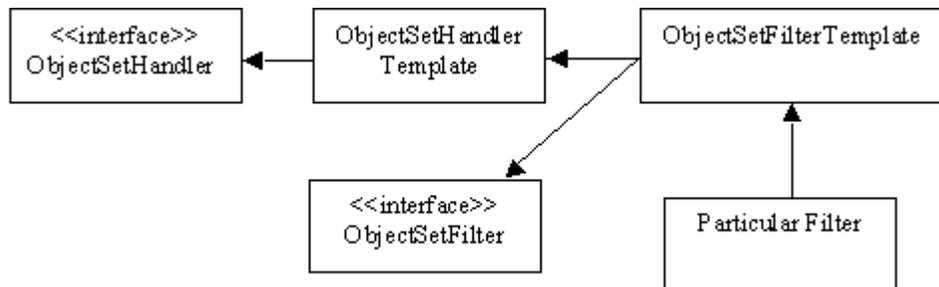
Adding New Navigators and Filters

Most of methods that manage generators and filters are implemented in the class `wt.ixb.objectset.ObjectSetHelper`.

The hierarchy of Navigator classes:



The hierarchy of Filter classes:



List of Existing Generators and Filters

This appendix provides help for GUI developer who will be using IX Object Collection mechanism by calling

```
ObjectSetHelper.computeObjectSetForGivenGeneratorsAndFilters(generatorIds,  
generatorParams, filterIds, filterParams);
```

Examples

Part With all Children

```
genId[0] = "productStructureNavigator";  
genParams[ 0] = "wt.part.WTPart:6789";  
WTHashSet objects = (WTHashSet) ObjectSetHelper.  
computeObjectSetForGivenGeneratorsAndFilters(genIds,  
genParams,  
new String [0 ],  
new String [0 ]);
```

All Objects in the Cabinet and Folder (Including Subfolders)

```
genId[0] = genId[ 1] = "folderContent";  
genParams[ 0] = "wt.folder.Cabinet:1865";  
genParams[ 1] = "wt.folder.Subfolder:5674";  
WTHashSet objects = (WTHashSet) ObjectSetHelper.  
computeObjectSetForGivenGeneratorsAndFilters(genIds,  
genParams,  
new String [0 ],  
new String [0 ]);
```

Note: Warning, if there are no filters, you can pass `new String[0]` for `filterIds` and `filterParams` (Don't pass null, an exception is thrown)

To get String constants for GUI, see `Windchill\codebase\wt\ixb\objectset\ObjectSetResource.rbInfo`.

Generators list

String id	Description	Localized name – En (for GUI)	Parameters as String
folderContent	Collects all objects in a given Cabinet/Folder (including subfolders)	Cabinet and Folder	<class name: oid>, like: wt.folder.Subfolder:1234 or wt.folder.Cabinet:1234
productStructureNavigator	Collects all children of a given Part (e.g. Parts, which it uses and Documents which describe it)	Product Structure (built with active Config Spec)	<class name: oid>, like: wt.part.WTPart:1234 for the top-level object. This object must be instance of WTPart
productStructureNavigatorEPM	Collects all children of a given CAD Document	CAD Document Structure (built with active config spec)	<class name: oid>, like: wt.epm.EPMDocument:1234 for the top-level object. This object must be instance of EPMDocument
productStructureNavigatorWithEPM	Collects all children of a given Part including related CAD Documents	Product Structure with CAD documents (built with active Config Spec)	<class name:oid>, like: wt.part.WTPart:1234 for the top-level object. This object must be instance of WTPart
singleDocument	Takes only given document	Document	<class name: oid>, like: wt.doc.WTDocument:1234

Note: actually <class_name:oid> is object local id

Filters list

String id	Description	Localized name - En (for GUI)	Parameters as String
filterByTime	Filters out objects with modification time before/after the given interval	Filter based on modification time	<timeFrom#timeTo>, where timeFrom and timeTo = "null" or Timestamp.toString()

Examples about Exp/Imp Application:

Export Application

The current Windchill Export OOTB GUI and the StandardIXBService is an Export Application.

The OOTB Windchill Export GUI is Export Application (client) that calls export process in StandardIXBService (Export Application (server)) via IXBHelper.

There are two ways to use it:

With GUI:

```
IXBExpImpStatus status = IXBHelper.service.doExport(  
    boolean previewOnly,  
    String[] generatorIds,  
    String[] generatorParams,  
    String[] filterIds,  
    String[] filterParams,  
    IXBStreamer ruleFile,  
    String guiId,  
    boolean detailedLog);
```

Without GUI:

```
IXBExpImpStatus status = IXBHelper.service.doExport(  
    boolean previewOnly,  
    String[] generatorIds,  
    String[] generatorParams,  
    String[] filterIds,  
    String[] filterParams,  
    IXBStreamer ruleFile,  
    boolean detailedLog,  
    String stDtd);
```

IXBHelper is a class in wt.ixb.clientAccess.

It calls methods `doExport(...)` of the class `StandardIXBService` to do export process.

`IXBExpImpStatus` is a class in `wt.ixb.clientsAccess` containing information about the Exp/Imp process and is used to pass Exp/Imp status between the server and client.

`generatorIds` – see definition above.

`generatorParams` is an array of Object Ids of top-level objects that will be exported. From the current Exp GUI, those objects will be chosen by using `NavigatorSearchDialog`. After the selection is done, this dialog will return a list of `IXBSelectedNavInfo` with Navigator Id and Generator Id, and seed object as an `objectId`. Given an object `obj`, we can get the Object Id by using `IXBHelper.service.getObjectId(obj)`.

`filterIds` – see definition above

`filterParams` is an array of objects attributes to set the objects to be excluded from export process, or to be included in export process, depends on the type of filters.

`ruleFile` is the rule file for export process. This file is provided to Exporter to create a tuner for export process.

`guiId` is the id of the GUI from which the export process is called. See the method `recordGuiIdInContext()` of the class `wt.clients.ixb.util.ExpImpServerRequest` for an example how to create the GUIid.

`detailLog` indicates whether the status message should be in details or not.

`stDtd` specifies which version of the Exp/Imp handlers will be used. This is used to support backward compatible. If `stDtd` is null or empty (""), the `STRING_DTD` will be calculated based on the current Windchill.

When the method `IXBHelper.service.doExport(...)` is called, it will call to the method `doExportImpl(...)` in the `StandardIXBService`.

This method:

- Creates a general export handler `ExportHandler` handler. This is an inner class of `StandardIXBService`.

```
Gets a list of objects that will be exported by calling
ObjectSetHelper.computeObjectSetForGivenGeneratorsAndFilters
(
generatorIds,
generatorParams,
filterIds,
filterParams);
```

- Creates an instance of `Exporter`, the class that does the export.
- Depending on `isPreview` (`true/false`) the exporter will do a preview or real export by calling methods of `Exporter` class mention in the section `Exporter` class of this document.
- Calls clean-up methods of the `ExportHandler` handler.

Import Application

The current Windchill Import GUI and `StandardIXBService` are the Import Application. The current Windchill Import OOTB GUI is Import Application client that calls import process in `StandardIXBService` (Import Application server) via `IXBHelper`.

There are two ways to use it:

With GUI:

```
IXBExpImpStatus status = IXBHelper.service.doImport(
    IXBStreamer ruleFile,
    IXBStreamer dataFile,
    boolean overrideConflicts,
    String guiId,
    boolean isPreview,
    boolean detailedLog,
    String creatorName);
```

Without GUI:

```
IXBExpImpStatus status = IXBHelper.service.doImport(
    IXBStreamer ruleFile,
    IXBStreamer dataFile,
    boolean overrideConflicts,
    boolean isPreview,
    boolean detailedLog,
    String creatorName,
    String stDtd);
```

`IXBHelper` is a class in `wt.ixb.clientAccess`.

It calls methods `doImport(...)` of the class `StandardIXBService` to do the import process.

`IXBExpImpStatus` is a class in `wt.ixb.clientsAccess` containing information about Exp/Imp process and used to pass Exp/Imp status between server and client.

`ruleFile` is the rule file for export process. This file is provided to Importer to create a tuner for import process.

`dataFile` is the jar file that contains XML files of objects that will be imported.

`overrideConflicts` specifies whether overridable conflicts must be overridden or not.

`isPreview` specifies whether the process should do real import, or check conflicts and display what objects will be imported.

`guiId` is the id of the GUI from which the export process is called. See the method `recordGuiIdInContext()` of the class `wt.clients.ixb.util.ExpImpServerRequest` for an example how to create the `GUIid`.

`detailLog` indicates whether the status message should be in details or not.

`creatorName` specifies how top-level imported objects (for example `EPMDocument`, `WTDocument`, `WTPart`) are created.

`stDtd` specifies which version of `Exp/Imp` handlers will be used. This is used to support backward compatible. If `stDtd` is null or empty (""), the `STRING_DTD` will be calculated based on version of current Windchill system.

When the method `IXBHelper.service.doImport(...)` is called, it will call to the method `doImportImpl(...)` in `StandardIXBService`.

This method:

- Puts creator name in `WTContext` to be used by import handler.
- Creates a general import handler `ImporterHandler` handler.
- Gets a list of XML files from the Jar file to be imported by calling `jar.getFileNamesByExtension("xml");`
- Creates an instance of `Importer`, the class that does the import job.
- Depending on `isPreview(true/false)`, the method `doImportImpl(...)` calls the appropriate methods of `Importer` to do a preview or the real import:
 - `importer.doImport(stream);`
 - `importer.doPreview(stream);`
- The others (`importer.doImport(fn, tag)` and `importer.doPreview(fn, tag)`) are for optimization, and they depend on how XML files are named. This feature is just for a particular `Exp/Imp Application` (`wt.clients.ixb` and `StandardIXBService`).
- Sends log messages back to client.

Simple Export Handler Code:

```
import java.io.File;
import java.io.PrintStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileOutputStream;
import java.io.FileInputStream;

import java.util.HashSet;
import java.util.Set;
import java.util.Iterator;

import wt.pom.Transaction;
import wt.content.ApplicationData;
import wt.content.ContentItem;
import wt.content.Streamed;

import wt.ixb.publicforapps.ApplicationExportHandlerTemplate;

import wt.ixb.publicforhandlers.IxbElement;
import wt.ixb.publicforapps.Exporter;
import wt.ixb.publicforapps.IxbHelper;

import wt.ixb.objectset.ObjectSetHelper;

import wt.util.WTException;
import wt.util.WTMessage;

import wt.ixb.clientAccess.IXBJarWriter;

import wt.fc.Persistable;

public class SimpleApplicationExportHandler extends
ApplicationExportHandlerTemplate{

    private File targetDir = null;
    private PrintStream log = null;

    private IXBJarWriter jw = null;
    private File resJar = null;
    private int fileNum = 0; //counter for exported content files
    private HashSet contentFileNames = new HashSet(); //to handle
content files with the same name

    public static final String NAME_IS_TAG = "TAG";
    public final static String CONTENT_SUBDIR = "CONTENTS";

    public SimpleApplicationExportHandler(File tmp_storeDir,
PrintStream a_log)
        throws WTException{
        if (!tmp_storeDir.exists()){
            tmp_storeDir.mkdirs();
        }
    }
}
```

```

        targetDir = tmp_storeDir;
        log = a_log;
    }

    public String storeContent (ApplicationData ob)
        throws WTEException{
        String fileName = ob.getFileName();
        String storeName = null;
        try{
            storeName = this.computeUniqueFileName(fileName);
            Streamed sd = (Streamed)ob.getStreamData().getObject();
            InputStream is = sd.retrieveStream();

            jw.addEntry(is, storeName);
        }
        catch (IOException ioe){
            throw new WTEException(ioe);
        }
        return storeName;
    }
    public void storeDocument (IxbElement elem)
        throws WTEException{
        try {
            String tag = elem.getTag();
            String fn = NAME_IS_TAG+"-"+tag+"-"+(fileNum++)+".xml";
            File file = new File(targetDir,fn);
            FileOutputStream stream= new FileOutputStream (file);
            elem.store(stream, IxbHelper.STANDARD_DTD);
            stream.close();

            jw.addEntry(file);
            file.delete();
        }
        catch (IOException ioe){
            throw new WTEException(ioe);
        }
    }

    public void storeLogMessage(String resourceBundle, String
messageKey, Object[] textInserts)
        throws WTEException{
        WTMMessage m = new WTMMessage(resourceBundle, messageKey,
textInserts);
        String s = m.getLocalizedMessage();
        log.println(s);
    }
    public void storeLogMessage(String resourceBundle, String
messageKey, Object[] textInserts, int importanceLevel)
        throws WTEException{
        storeLogMessage (resourceBundle, messageKey, textInserts);
    }

    public void exportObjectContent (Object obj, Exporter exporter,
ContentItem item, String exportFileName)
        throws WTEException {
        if (item instanceof ApplicationData) {
            ApplicationData ad = (ApplicationData) item;

```

```

        Streamed streamedIntfc = (Streamed)
ad.getStreamData().getObject();
        try{
            InputStream is = streamedIntfc.retrieveStream();
            jw.addEntry(is, exportFileName);
        }
        catch (IOException ioe){
            throw new WTEException(ioe);
        }
    }
}

private String computeUniqueFileName (String fn) throws
IOException {
    //compute file name in jar (should be unique)
    if (contentFileNames.contains(fn)) {
        // if simple content's name already has been used then
look for
        // name in form  name-123.ext
        // As a result will have names like:  design.doc,
design-23.doc, design-57.doc, ...
        int    i    = fn.lastIndexOf('.');
        String  fn_n = ( i>0 ? fn.substring(0, i) : fn);
        String  fn_t = ( i>0 ? fn.substring(i+1)  : "" );
        while (true) {
            fn = (i>0 ?
                fn_n + '-' + (fileNum++) + '.' + fn_t :
                fn_n + '-' + (fileNum++)
            );
            if (!contentFileNames.contains(fn)) break;
        }
    }
    contentFileNames.add(fn);
    String fnInJar = CONTENT_SUBDIR + "/" + fn;
    return fnInJar;
}

public File doExport(    String[] generatorIds, String[]
generatorParams,
                        String[] filterIds, String[] filterParams,
                        File ruleFile, File policyFile,
                        String actionName, String stDtd, File
resultingJar)
    throws WTEException{

    //init jar file
    resJar = resultingJar;
    try{
        jw = new IXBJarWriter(resultingJar);
    }
    catch (IOException ioe){
        throw new WTEException(ioe);
    }

    //adopt incoming rule file
    IxbElement clientSettingsElement = null;
    if (ruleFile!=null) {

```

```

        try{
            InputStream clientSettingsStream = new
FileInputStream(ruleFile);
            clientSettingsElement =
IxbHelper.newIxbDocument(clientSettingsStream, false);
        }
        catch(IOException ioe){
            throw new WTEException(ioe);
        }
    }

    //create exporter
    Exporter exporter = null;
    if ( policyFile==null ) { // policy rule is null; export
action is expected ...
        exporter = IxbHelper.newExporter (this,
IxbHelper.STANDARD_DTD, clientSettingsElement, null, actionName );
    }
    else{
        exporter = IxbHelper.newExporter (this,
IxbHelper.STANDARD_DTD, clientSettingsElement, policyFile, null );
    }

    //gen set of items to export
    Set res =
ObjectSetHelper.computeObjectSetForGivenGeneratorsAndFilters
(generatorIds, generatorParams, filterIds, filterParams);

    Iterator iter = res.iterator();

    Transaction trx = new Transaction();
    try {
        if ( !(actionName != null &&
actionName.equals(wt.ixb.tuner.ExportActionHelper.NO_ACTION_CMD)
))){
            trx.start();
        }
        while (iter.hasNext()) {
            Persistable ob = (Persistable)iter.next();
            exporter.doExport(ob);
        }
        exporter.finalizeExport();
        if ( !(actionName != null &&
actionName.equals(wt.ixb.tuner.ExportActionHelper.NO_ACTION_CMD)
))){
            trx.commit();
        }
        trx = null;
    }
    finally {
        if (trx != null) {
            if ( !(actionName != null &&
actionName.equals(wt.ixb.tuner.ExportActionHelper.NO_ACTION_CMD)
))){
                trx.rollback();
            }
        }
    }
}

```

```

        trx = null;
    }
}
try{
    jw.finalizeJar();
}
catch(IOException ioe){
    throw new WTEException (ioe);
}

return resJar;

}
}

```

Simple Import Handler Code:

```

import java.io.File;
import java.io.PrintStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileOutputStream;
import java.io.FileInputStream;

import java.util.HashSet;
import java.util.Set;
import java.util.Iterator;

import wt.pom.Transaction;

import wt.content.ApplicationData;
import wt.content.ContentItem;
import wt.content.Streamed;

import wt.ixb.publicforapps.ApplicationImportHandlerTemplate;

import wt.ixb.publicforhandlers.IxbElement;
import wt.ixb.publicforhandlers.IxbHndHelper;
import wt.ixb.publicforapps.IxbDocument;
import wt.ixb.publicforapps.Importer;
import wt.ixb.publicforapps.IxbHelper;

import wt.ixb.objectset.ObjectSetHelper;

import wt.ixb.actor.actions.IxbActionsHelper;

import wt.util.WTEException;
import wt.util.WTMessage;

import wt.ixb.clientAccess.IXBJarReader;

import wt.fc.Persistable;

import javax.xml.transform.stream.StreamSource;

```

```

public class SimpleApplicationImportHandler extends
ApplicationImportHandlerTemplate{

    private IXBJarReader jr = null;
    private PrintStream log = null;

    public SimpleApplicationImportHandler(PrintStream a_log){
        log = a_log;
    }

    public InputStream getContentAsStream (String contentId)
        throws WTEException {
        try{
            return jr.getStreamByName (contentId);
        }
        catch(IOException ioe){
            throw new WTEException(ioe);
        }
    }

    public void storeLogMessage(String resourceBundle, String
messageKey, Object[] textInserts)
        throws WTEException{
        WTMesssage m = new WTMesssage(resourceBundle, messageKey,
textInserts);
        log.println(m.getLocalizedMessage());
    }

    public void doImport( File ruleFile, File dataFile, boolean
_overrideConflicts,
                        String actorName, File policyFile)
        throws WTEException{
        try{
            jr = new IXBJarReader(dataFile);
        }
        catch(IOException ioe){
            throw new WTEException (ioe);
        }
        //prepare rule file
        String ruleFileName = (ruleFile!=null)?
ruleFile.getAbsolutePath(): null;

        //prepare policy file
        StreamSource xslPolicyFile = null;
        if (policyFile!=null) {
            xslPolicyFile = new
StreamSource(policyFile.getAbsolutePath());
        }
        Boolean overrideConflicts = new Boolean (_overrideConflicts);

        Importer importer = IxbHelper.newImporter(this,
IxbHelper.STANDARD_DTD,ruleFileName,
overrideConflicts, null /*validate*/);

        String [] fns = jr.getNonContentFileNamesByExtension ("xml");

```

```

        boolean validate =
IxbHndHelper.getIxbProperty("import.parser.validate", false);

        for (int i=0; i<fns.length; i++) {

            String fn = fns[i];
            InputStream stream = null;
            try{
                stream = jr.getStreamByName (fns[i]);
            }
            catch (IOException ioe){
                throw new WTEException (ioe);
            }
            IxbDocument doc = IxbHelper.newIxbDocument(stream,
validate);

                //if policyFile == null, apply actorName to XML Document
before pass it to importer
                if (policyFile == null){
                    IxbElement rootElem = doc.getRootElement();
                    //XML_ACTION_KEY = "actionInfo/action"
                    IxbElement actionElement =
doc.getElement(IxbActionsHelper.XML_ACTION_KEY);
                    if (actionElement == null){
                        rootElem.addValue(IxbActionsHelper.XML_ACTION_KEY,
actorName);
                    }
                    else {
                        rootElem.removeChildElement( actionElement);
                        rootElem.addValue(IxbActionsHelper.XML_ACTION_KEY,
actorName);
                    }
                }
                else { //apply policy file
                    doc =
IxbActionsHelper.writeActionAndParametersToFileXML(doc,xslPolicyFi
le);
                }
                //set elem ready for import
                importer.doImport(doc);
            }

            //perform actual import
            importer.finalizeImport();
        }
    }
}

```


13

Xconfmanager Utility

This chapter contains information about the xconfmanager utility.

Topic	Page
About the xconfmanager Utility.....	13-2
About the Windchill Command	13-5

About the xconfmanager Utility

The xconfmanager is a command line utility that you can run to add, remove, or modify properties in any Windchill property file. With one exception, the following files are managed by Windchill Information Modeler and should not be edited manually or edited with the xconfmanager:

- associationRegistry.properties
- classRegistry.properties
- descendentRegistry.properties
- modelRegistry.properties

The xconfmanager utility saves your changes in the site.xconf file and provides an option to generate updated property files using the updates in the site.xconf file. The site.xconf file contains the changes made to Windchill property files starting with the installation and continuing with each use of the xconfmanager utility or the System Configurator. The xconfmanager utility is located in the *<Windchill>/bin* directory.

This chapter describes only the information and instructions necessary to modify specific Windchill properties. A full description of the xconfmanager utility and management of the Windchill property files is documented in the *Windchill System Administrator's Guide* in the Administering Runtime Services chapter.

Anyone with write access to the XCONF and property files under the Windchill installation directory can successfully run the xconfmanager utility. The xconfmanager is executed from the command line from within a windchill shell. See About the windchill Command chapter for more information about the windchill shell.

The syntax of xconfmanager command is as follows:

```
xconfmanager {-FhuwvV} {-r <product_root>} {-s <property_pair>
{-t <property_file>}} {--reset <property_names>}
{--undefine <property_names>} {-d <property_names>} {-p}
```

For the purposes of modifying Windchill properties, you will primarily use the set (s), targetFile (t), and propagate (p) parameters.

- set is used to define the property and new property value. See the Formatting Property Value Guidelines section (below) for information about formatting the *<property_pair>* value.
- targetFile is used to specify the directory location of the property file. If the file name or path contains spaces, you must enclose the *<property_file>* value in double quotes (" "). It is recommended to use a fully-qualified file name to ensure an accurate reference to the file is made.
- propagate is used to propagate the changes made to the XCONF files into the property file being modified in order to keep the XCONF and the property files in synch with one another.

- help is used to view the help for xconfmanager.

Some examples are as follows:

- xconfmanager is run from the windchill shell. To open a windchill shell, at a command prompt, execute the following command:

```
windchill shell
```

- To display xconfmanager help, execute the following command from the windchill shell:

```
xconfmanager -h
```

- To display the current settings for a property, execute the following command from the windchill shell:

```
xconfmanager -d <property_names>
```

- To change a property value, execute the following command from the windchill shell:

```
xconfmanager -s <property_pair>=<property_value>  
-t <property_file> -p
```

It is recommended to use the fully-qualified name of the property file to ensure an accurate reference to the file is made.

Formatting Property Value Guidelines

The property values you set must conform with the specification for `java.util.Properties`. The following guidelines will help ensure that you set properties correctly:

Use forward slashes (/) in file paths so that the platform designation is not an issue.

To specify a property whose value contains characters that might be interpreted by your shell, escape them using the appropriate technique for the shell you are using.

For example, on a Windows system you can include spaces in a value by enclosing the argument with double quotes. For example, use the following:

```
-s "wt.inf.container.SiteOrganization.name=ACME Corporation"
```

On a UNIX system, you can use double quotes or you can escape the space character with a backslash. For example, use the following:

```
-s wt.inf.container.SiteOrganization.name=ACME\ Corporation"
```

On UNIX, dollar signs are usually interpreted by shells as variable prefixes. To set a property value that has a dollar symbol in it, use single quotes around the argument so that it is not interpreted by the shell or use backslash to escape the dollar symbols. For example, use either of the following:

```
-s 'wt.homepage.jsp=$(wt.server.codebase)/wtcore/jsp/wt/portal/  
index.jsp'
```

or

```
-s wt.homepage.jsp=  
`\$(wt.server.codebase)/wtcore/jsp/wt/portal/index.jsp
```

Other than escaping arguments so that the command line shell does not misinterpret them, the values should not need to be escaped any further to be compatible with XML or property file syntaxes. The xconfmanager escapes property names and values automatically if necessary.

About the Windchill Command

PTC has provided a command, `windchill`, to invoke Windchill actions. For example, the command can be used to stop and start Windchill, check the status of the Windchill server, and create a new shell and set the environment variables. It can also be used as a Java wrapper. In that regard, it can accept a Class file as an argument, just like Java, and execute it without a predefined environment (Windchill classes in CLASSPATH, Java in PATH, and so on).

The `windchill` command should be used to execute any server-side Windchill Java code. This will insure that the environment that the command is executed in is properly setup. The environment that actions are executed within, including the `windchill` shell action, is defined by the `wt.env` properties in the `wt.properties` file. For example, the `wt.env.CLASSPATH` property will set the CLASSPATH environment variable for the action that is being invoked.

The `windchill` command is a Perl script that has also been compiled into a Windows binary executable. For UNIX systems, Perl 5.0 or greater must be installed. The `windchill` script assumes that Perl is installed in the standard install location of `/usr/bin/perl`. If Perl is not installed at this location, you can either create a symbolic link (recommended method) to the Perl install location or edit the `windchill` script to reference the Perl install location. To modify the `windchill` script, edit the `<Windchill>/bin/windchill` file. Locate the `#!` entry (for example, `#!/usr/bin/perl -w`) and change the Perl directory to the location where Perl is installed.

The `windchill` command is located in the `<Windchill>/bin` directory. If you receive a command not found message when you execute the `windchill` command, add the `<Windchill>/bin` directory to your PATH environment variable. The syntax of the `windchill` command is:

```
windchill [args] action
```

You can display the help for the `windchill` command by executing `windchill` with the `-h` argument or with no argument.

The following tables list some of the arguments and actions applicable to the `windchill` command. To see a complete list of the arguments, use the report generated from the help (argument).

windchill Arguments:

Arguments (optional)	Description
- h, --help	Displays help and exits.
-v, --[no]verbose	Explains what is being done when a command is executed. Default is noverbose.
-w, --wthome=DIR	Sets the Windchill home directory. Default is the parent directory containing the windchill script.
--java=JAVA_EXE	The Java executable. Default is the wt.java.cmd variable value specified in the \$WT_HOME/code-base/wt.properties file.
-cp, --classpath=PATH	Java classpath. Default is the wt.java.classpath variable value specified in the \$WT_HOME/code-base/wt.properties file.
--javaargs=JAVAARGS	Java command line arguments.

windchill Actions

Action	Description
shell	Sets up a Windchill environment in a new instance of the currently running shell.
start	Starts the Windchill server.
stop	Stops the Windchill server.
status	Retrieves the status of the Windchill server.
version	Displays the Windchill install version.

Action	Description
properties <i><resource>[,...][?key[&key2]...]</i>	Displays the properties as seen by Windchill for the given resource with substitution, etc. executed. It can be limited to a given set of keys. For example: windchill properties wt.properties — lists all wt.properties windchill properties wt.properties?wt.server.codebase — lists server codebase windchill properties wt.properties?wt.env.* — lists all the environment variables use by windchill shell windchill properties — with no arguments generates the help report
CLASS [CLASS_ARGS]	Run a Windchill class with optional class arguments. For example: windchill wt.load.Developer -UAOps

About the windchill shell

The windchill shell brings up a new command shell, from the parent shell that is setup for the Windchill environment. This includes setting all environment variables defined in wt.env property in the wt.properties file.

To execute the windchill shell, at the command prompt enter the following command:

```
windchill shell
```

When you are finished using the windchill shell, you can exit the shell and return to the parent shell.

PTC recommends running all server-side Windchill applications, tools, and utilities from the windchill shell. Also, you can use the windchill shell to set up your development environment to use javac or Java directly.

A

Windchill User Authentication

In order for Windchill method servers to enforce access control at an application level, they must be able to securely and reliably determine the identity of a calling client. Recognizing that the mechanics behind user authentication may need to be customized for different environments, Windchill's user authentication framework is designed with extensibility and replacement in mind. The framework itself and the reference implementation based on HTTP authentication are described separately in this appendix. If you are interested in implementing new authentication techniques, you should read about the framework (described next). If you are interested only in understanding the out-of-the-box authentication handler based on HTTP authentication, you can go to the reference implementation section.

Topic	Page
User Authentication Framework	A-2
Reference Implementation (HTTP authentication)	A-5
Null Authentication	A-7

User Authentication Framework

The primary goal of the authentication framework is to allow each client-to-server RMI call to be reliably, securely, and efficiently associated with an authenticated user name by the server. A secondary goal is to be simple and extensible so that a number of implementations are possible, including the reference implementation described later in this appendix. To achieve these goals, the framework consists of three key interfaces that partition responsibilities as follows:

- Endorsing individual RMI calls (the `wt.method.MethodAuthenticator` interface).
- Securely and reliably determining user name (the `wt.auth.AuthenticationHandler` interface).
- Validating endorsed calls, associating them to a user name (the `wt.auth.Authenticator` interface).

Each of these interfaces is described below. The `wt.method` package contains the Windchill method server application main class, and it owns the basic client-to-server calling mechanism. The `wt.auth` package builds on the basic framework provided by the `wt.method` package to add more authentication specific support classes.

`wt.method.MethodAuthenticator`

The Windchill method servers provide a dynamic method invocation interface that is accessed through a non remote class called `wt.method.RemoteMethodServer`. This class exposes an `invoke` method but encapsulates the details of accessing a remote method server. It hides the actual RMI operations in order to add network failure recovery and authentication support. The invocation target and all the call arguments are passed in an argument container class called `wt.method.MethodArgs`. This argument container contains a field for arbitrary authentication information.

The basic method invocation framework does nothing with this field. Instead, it allows for an arbitrary server-supplied object (a `wt.method.MethodAuthenticator`) to endorse outgoing calls. This object can set the extra field or wrapper the entire `wt.method.MethodArgs` object such that deserialization on the server-side actually initializes the extra field. The requirement is simply that when argument deserialization is complete, a server-side authenticator object can use the credentials in this extra field to reliably determine the caller's user name.

The client receives a `wt.method.MethodAuthenticator` object inside a special Throwable, `wt.method.AuthenticationException`, sent from the server when authentication is required. The client automatically handles this exception by initializing the received `wt.method.MethodAuthenticator` and using it to endorse a retry of the failed call. Once installed, it continues to endorse all calls from that client.

The `wt.method.MethodAuthenticator` interface consists of the following methods:

public MethodArgs **newMethodArgs** ()

Creates a new wt.method.MethodArgs object or a subclass used by this authenticator. The server calling mechanism delegates construction of the argument-containing object to the installed authenticator object so that it can optionally instantiate a special extension of wt.method.MethodArgs that performs encryption or message digest computation and signing during serialization. In these cases, the endorse method called prior to serialization just guarantees that the argument container is of the desired type.

public boolean **init** ()

Initializes a newly received authenticator. IT can return false or throw a runtime exception to prevent the installation of this authenticator and subsequent retry of the failed method invocation.

public MethodArgs **endorse (MethodArgs args)**

Endorses the current message in some arbitrary way. This can be as simple as setting the auth field of the object, or replacing the passed object with a subclass that performs encryption or message digest computation and signing during serialization. A newly installed authenticator was not given a chance to construct the original argument container; this gives it a chance to guarantee that it is of the right class or to construct a replacement.

public boolean **failure (MethodArgs args, AuthenticationException e)**

Called to report failure of an endorsed call due to a wt.method.AuthenticationException. The previous argument container and the received exception are passed in. The method should return true if the existing authenticator object should be deinstalled, possibly to be replaced by a new one contained in the exception object. If it returns false, the current authenticator will remain installed and given another change to endorse the call.

This simple interface does not assume much about how client-to-server calls will be secured, but allows for many interesting implementations that can be dynamically loaded from the server as authenticator objects are thrown back from the server.

wt.auth.AuthenticationHandler

The method server does not hard-code what authentication scheme will be used to securely and reliably determine user names. Instead, the wt.auth package supports a configuration property, wt.auth.handlers, that specifies a list of classes that implement an authentication handler interface, wt.auth.AuthenticationHandler.

The class wt.auth.Authentication is responsible for making the client's authenticated user name available for the higher-level application layers to take advantage of. It has a static method called getUsername that is used to access the authenticated user name for the current thread. If the client call being processed by the current thread did not pass any authentication information, it asks the first

wt.auth.AuthenticationHandler in the configured list to construct a bootstrapping wt.method.MethodAuthenticator object and sends it back to the client inside a wt.method.AuthenticationException as described earlier.

It is called a bootstrapping authenticator because it is usually responsible only for performing some kind of secure user login that can reliably identify the authenticated user name and communicate that to the method server. Once this initial login processing is performed, it is likely to be replaced by an authenticator that simply associates each client call to that authenticated user name.

The wt.auth.AuthenticationHandler interface consists of the following methods:

```
public MethodAuthenticator getBootstrapAuthenticator ()
```

Gets an initial wt.method.MethodAuthenticator object capable of reliably identifying the user. If this handler is not appropriate for the current client host (available via wt.method.MethodContext), this method should return null to give another handler in the handler list a chance.

```
public MethodAuthenticator getBootstrapAuthenticator (String session_id)
```

Gets a new wt.method.MethodAuthenticator object capable of reliably re-identifying the user of an existing session. This is used to implement re-authentication to change the user name associated with an existing session. The session ID should be available later when replacing the bootstrapping authenticator. If this handler is not appropriate for the current client host (available via wt.method.MethodContext), this method should return null to give another handler in the handler list a chance.

```
public MethodAuthenticator bootstrap (MethodAuthenticator authenticator)
```

Initialization method called from remote bootstrapping authenticator's init method. It can be used to convert the bootstrapping authenticator into a real one, usually by calling the wt.auth.Authentication class which delegates to a wt.auth.Authenticator class (described later in this section). If the passed authenticator is invalid, it throws a wt.method.AuthenticationException to allow another handler in the handler list a chance to perform authentication. This method should return null if the passed authenticator is not an instance used by this handler implementation.

wt.auth.Authenticator

In order to keep the initial (bootstrap) login authentication separate from subsequent per-call authentication, a third interface, wt.auth.Authenticator, and another configurable property, wt.auth.authenticator, are used to validate endorsed calls. This allows several login techniques to share the same per-call endorsement technique and allow them to be developed independently.

When a bootstrapping authenticator has determined the authenticated user name of a client, the bootstrapping authenticator can be replaced by one returned by the wt.auth.Authentication class's newMethodAuthenticator method. This method

will delegate to the configured authenticator class to construct a new wt.method.MethodAuthenticator for the given authenticated user name.

When application code later asks the wt.auth.Authentication class's getUsername method to identify the client's authenticated user name, it will again delegate to the configured authenticator to associate the call's endorsement with the authenticated user name that was determined by the bootstrapping authenticator.

The wt.auth.Authenticator interface consists of the following methods:

public MethodAuthenticator **newMethodAuthenticator (String user)**

Returns a new wt.method.MethodAuthenticator object that will associate the given authenticated user name to endorsed method calls.

public MethodAuthenticator **newMethodAuthenticator (String user, String session_id)**

Returns a new wt.method.MethodAuthenticator object that will associate the given authenticated user name to endorsed method calls for an existing session. This is to support re-authentication for an existing session initiated by the reauthenticateUser method. The session ID will be a string originally created by this class in the reauthenticateUser method.

public String **getUserName (Object auth)**

Uses the passed authentication information (usually added to the method calls by the previously returned wt.method.MethodAuthenticator object) to determine the authenticated user name.

public Object **setUserName (String user)**

Returns an object that, when passed to getUsername will cause it to return the given user name. This is used within the server to associate non-RMI method invocation contexts (such as, HTTP or CORBA-based services) with an authenticated user name.

public void **reauthenticateUser (Object auth)**

Uses the passed authentication information to initiate re-authentication for the identified session. For session-based authenticators, this method can prime the current session for re-authentication and throw back a new bootstrapping authenticator. It can pass an arbitrary session ID string to the bootstrapping authenticator which will eventually return to the newMethodAuthenticator method so this class can associate the new authentication to the existing session.

Reference Implementation (HTTP authentication)

The framework described in the preceding section shows that Windchill user authentication is open to customization to meet site or application needs. This section describes what is available out-of-the-box. The Windchill base product

contains a set of classes that work within the preceding framework to perform user authentication based on HTTP Authentication.

Windchill applications are intended to be Web-based, and the Web infrastructure already has standards to support user authentication between browsers and Web servers. Applications deployed on the Web should be able to leverage the existing user authentication infrastructure provided by browsers and Web servers, allowing it to be shared across multiple Web-based applications, as well as other resources accessed via the same Web servers.

Outsourcing user authentication to an interaction between the browser and Web server allows sites more flexibility in administering user authentication because they are no longer limited to solutions provided directly by Windchill. For example, one site might deploy Windchill using Netscape Enterprise server in conjunction with a central enterprise Certificate Server and Directory Server that manages users. Such a configuration can support enterprise-wide, digital certificate-based user authentication shared by all Web applications, not just Windchill.

Another site might choose to use Microsoft's Internet Information Server combined with Windows NT Domain controllers to let users running Internet Explorer authenticate transparently using NT Challenge/Response authentication. Such a site achieves single sign-on support for their NT users using the same Windchill software as the Netscape site.

The Windchill user authentication framework is used to accomplish user authentication while allowing the actual HTTP authentication to be carried out between the browser and Web server by implementing objects that satisfy the interfaces expected by the Windchill framework as described next:

- `wt.httpgw.HTTPAuthentication` implements `wt.auth.AuthenticationHandler`

First, a concrete `wt.auth.AuthenticationHandler` class is provided as part of the `wt.httpgw` package. This package contains the HTTP gateway into Windchill method servers. One of the gateway URLs is required to be for authenticated access only. This allows Windchill to use the `REMOTE_USER` value passed by the Web server as an authenticated user name.

When asked to create a bootstrapping authenticator object, the `wt.httpgw.HTTPAuthentication` class constructs and returns a new `wt.httpgw.HTTPLogin` object.

- `wt.httpgw.HTTPLogin` implements `wt.method.MethodAuthenticator`

The `wt.httpgw.HTTPLogin` object's `init` method opens a URL connection to the authenticated Windchill HTTP Gateway URL. Specifically, it attempts to invoke a `login` method in the `wt.httpgw.HTTPAuthentication` class. Because the authenticated Windchill Gateway is subject to access control by the Web server, the call is not received by the Windchill method server until after the client browser and the Web server carry out HTTP authentication.

When it finds itself outside of a Web browser, such as a stand-alone Java application, the `wt.httpgw.HTTPLogin` class is limited in its ability to carry out vendor specific authentication schemes (such as, NT Challenge/Response). It does, however, support HTTP 1.0 Basic and HTTP 1.1 Digest authentication using its own login dialog to prompt for a user name and password.

Once authenticated by the Web server, the login method is reached and is responsible for returning a new `wt.method.MethodAuthenticator` in the body of the HTTP reply that can be used to authenticate RMI-based calls to the server. It returns the configured `wt.auth.Authenticator` class, which defaults to `wt.session.SessionAuthenticator`.

- `wt.session.SessionAuthenticator` implements `wt.auth.Authenticator`

The `wt.session` package is responsible for maintaining session-based state about clients. The `wt.session.SessionAuthenticator` class uses this server-maintained session state to associate unique session IDs with authenticated user names. It returns instances of `wt.session.SimpleSessionAuthenticator` to the client.

- `wt.session.SimpleSessionAuthenticator` implements `wt.auth.MethodAuthenticator`

The `wt.session.SimpleSessionAuthenticator` uses a unique session ID to endorse each call.

Null Authentication

The `wt.auth.NullAuthentication` class, another authentication handler provided in the Windchill base product class allows standalone applications running on the same host as the Windchill method server to authenticate using their `user.name` Java system property. It can be used to allow standalone administrative applications, such as install/load tools, to execute even if the built-in authentication schemes supported by the `wt.httpgw.HTTPLogin` class are not sufficient for the local Web server environment.

It checks against the client's IP address to determine if it is a local client. If so, it sends a `wt.auth.NullLogin` object to the client which accesses the `user.name` Java system property. It will not access the Java system properties if a security manager is in effect. In other words, it does not try to authenticate browser-based clients, even if they are running on the local host. It is intended only to authenticate stand-alone administrative applications.

B

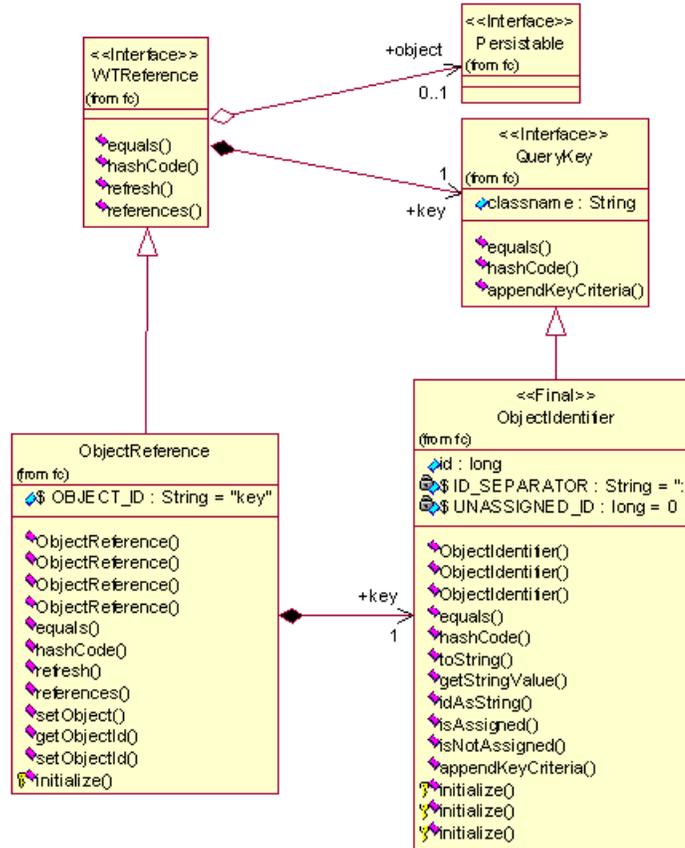
Windchill Design Patterns

This section describes design patterns that represent Windchill's current best practices regarding development of server logic, most notably the design pattern on how to develop business services. These patterns have emerged during the development of the Windchill services and should be used as standards and guidelines when developing new server logic.

Topic	Page
The Object Reference Design Pattern	B-2
The Business Service Design Pattern	B-3
The Master-iteration Design Pattern	B-6

The Object Reference Design Pattern

One of the most basic design patterns is the object reference design pattern.



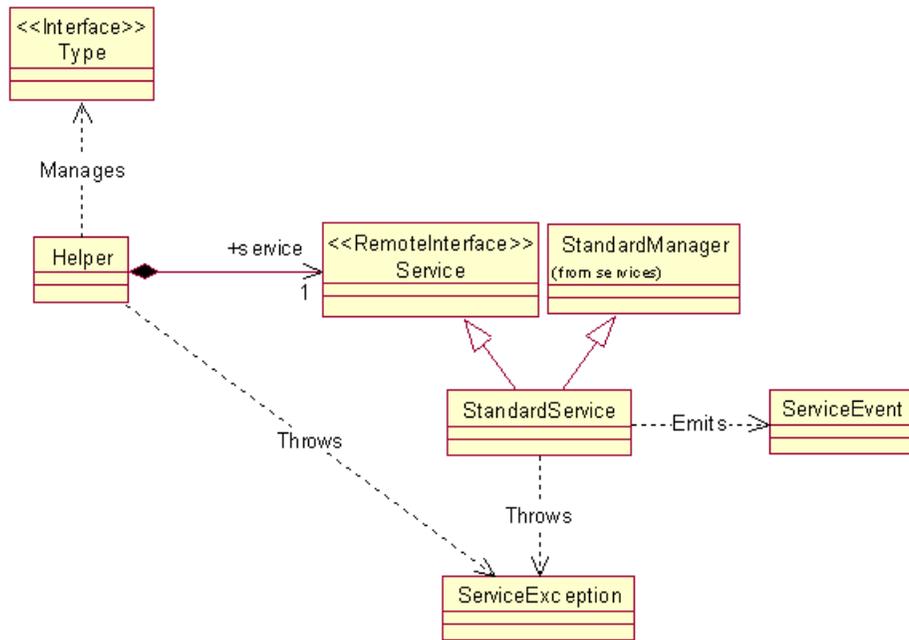
Object Reference Pattern

This pattern essentially encapsulates details concerning persistable objects and their unique database query key. The pattern asserts that an object is a derived attribute aggregated by reference and is not persisted. The object's unique database query key is aggregated by value, is persisted, and is write-protected against the attempt of any other package class to set its value.

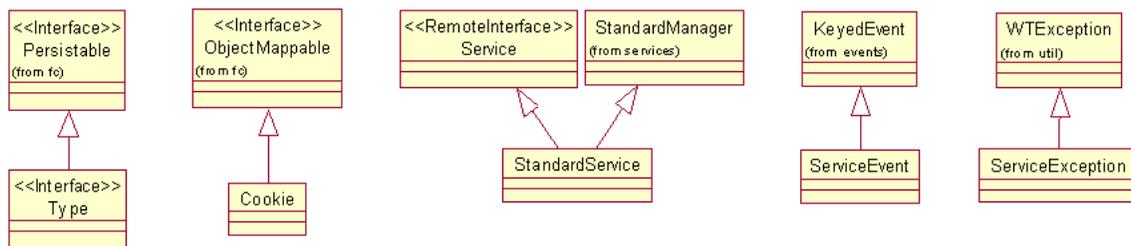
In cases where database performance and storage are issues, object references persist only their object identifiers, and can be used in place of actual objects and acted upon via their identifiers. However, when the actual object is required, it can be gotten from the object reference which may or may not be currently holding the object. If the object reference does not hold the object and the object is asked for, the object is refreshed via its query key from the database.

The Business Service Design Pattern

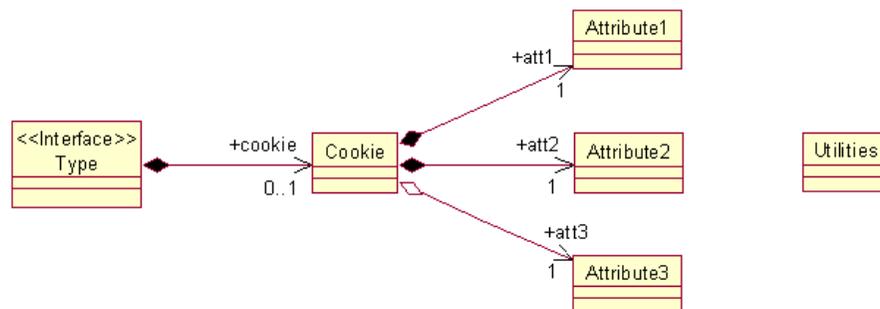
The most prevalent design pattern referenced to build Windchill services is the business service design pattern (figures B-2, B-3, and B-4).



Business Service Interface



Business Service Classification



Business Service Information

This pattern has the following major kinds of abstractions:

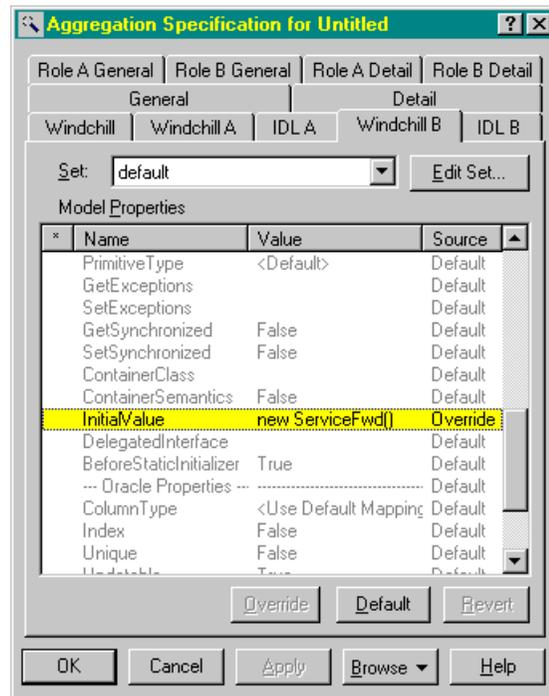
- Type
- Cookie
- Helper
- Service
- ServiceEvent
- ServiceException

The Type abstraction provides an interface for means to type an object as being of a particular kind. This interface is what the service expects to deal with in terms of input and output, other than additional information. An object that does not specify it is of a certain type cannot statically be used by the service and thus is rejected at compile-time. In general, a Type is a kind of persistable object.

The Cookie abstraction provides a class that is used to specify the information to be associated with and stored as a part of the typed object. When an object asserts itself as being a Type, the Cookie and its attributes, including all nested attributes, are code generated into the object along with applicable accessors. If a Cookie's cardinality is 0..1, the Cookie and all its nested attributes can be stored as null if none of the Cookie's attributes are required. If any of the simple, or structured, attributes of the Cookie are constrained to be non-null in the database, the Cookie is forced to be non-null.

The Helper abstraction provides a class representing the service's external interface from which all visible functionality can be invoked. The helper is intended to specify only static methods and attributes which any other class can access without having to create any instances. The static methods are typically Cookie accessors. The static attribute is a remote reference to the service's server-

side functionality by means of initializing the "service" attribute as shown in the following example.



Initializing the Service Attribute

Notice the property in bold type named "InitialValue" and its value, "new ServiceFwd()." This property setting directs the code generator to make an initializer for the "service" instance to what is specified as the value. The service name appended with "Fwd" is the name of the class generated for a class stereotype as being a "RemoteInterface."

The Service abstraction provides an interface that specifies the main functionality of the service itself, which may or may not be invoked remotely if the interface is stereotyped as a "RemoteInterface." Otherwise, the service's interface will be available only locally in the server. This interface must be adhered to and implemented for the service to function properly. Additionally, a standard implementation of the service's methods exists. This standard implementation is a singleton executing on the server and is the default for all Windchill services.

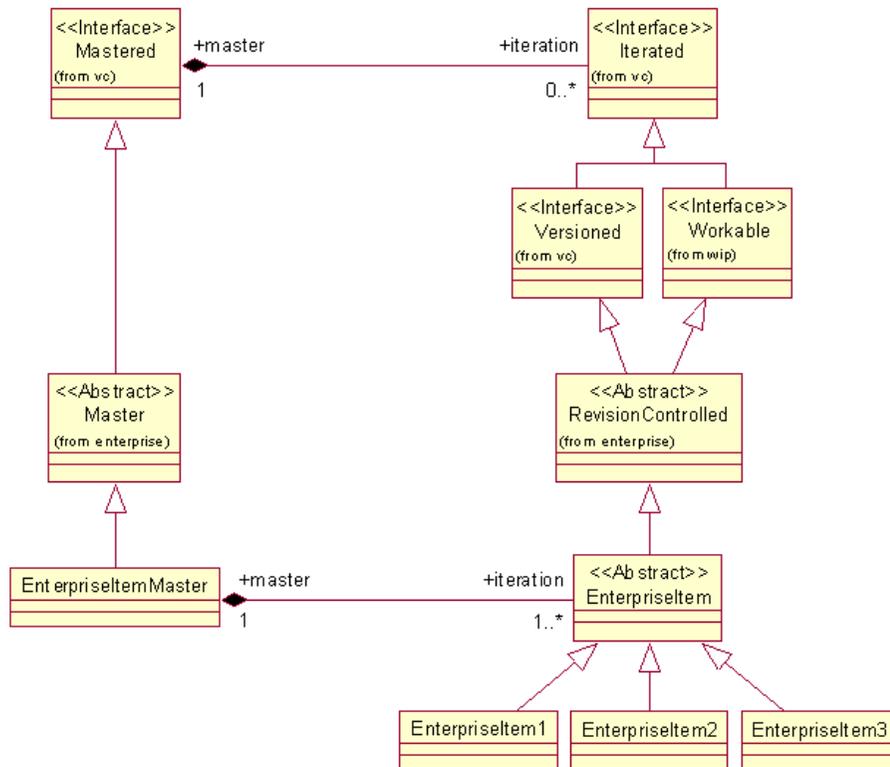
The ServiceEvent abstraction provides a common definition of an event that can be emitted from the service and cause another service to be notified of the event. This event specifies one or more kinds of occurrences that are used to generate keys for listeners. Because these specific kinds of occurrences are extremely simple in nature, only one event per service that defines all occurrences is specified.

The ServiceException abstraction provides a common definition of an exceptional condition that may occur as a result of abnormal behavior in the service. This

exception, along with the service's resource bundle, can be used exclusively to throw any and all kinds of errors. However, it may be appropriate, but not necessary, to specialize this exception for more explicit and applicable errors.

The Master-iteration Design Pattern

The design pattern that you must adhere to for all versioned data is the master-iteration design pattern.



Master-Iteration Pattern

This pattern typically establishes two objects that work in concert with one another. Without one, the other should not exist and is certainly invalid. At the root are the basic abstractions:

- Mastered
- Iterated

The Mastered interface provides an abstraction of a plug-and-play component in conjunction with the Iterated interface. The intent is that, in a business model, an object would assert that it is a master by inheriting the Mastered interface. With this assertion, the business object can then be mastered through the version control service's API. The business object must assert itself as being a kind of mastered object in order for its instance to be iterated.

The Iterated interface provides an abstraction of a plug-and-play component in conjunction with the Mastered interface. The intent is that, in a business model, an object would assert that it is an iteration (instance) by inheriting the Iterated interface. With this assertion, the business object can then be incrementally superseded, rolled back, and rolled up through the version control service's API, provided it has a master. The business object must assert itself as being a kind of Iterated object in order for it to be incrementally changed.

The next level of master-iteration pairs defines abstract entities that start pulling together (that is, assert) all applicable capabilities from a general [virtual] enterprise perspective. The level below starts becoming more concrete where the EnterpriseItemMaster is concrete but the EnterpriseItem is not. It is at this level where the association between master and iteration is overridden with the exact named roles. However, it should be noted that cardinality of the iterations within a master can be specialized to be further constrained. Also, this association again specifies itself as a foreign key and the master can be auto-navigated from the iteration. Thus, when an iteration is fetched from the database, its master is fetched as well in one SQL statement via a database view.

Note that the iteration at this level need not be concrete for an association of this kind with the foreign key, and auto-navigation on the concrete class can have the other side as an abstract class.

At the very bottom, all of the concrete specializations of the EnterpriseItem exist. All of these specializations inherit the foreign key, auto-navigate association from EnterpriseItem. And thus, each is generated with a specific database view such that three database views are generated for EnterpriseItem1, EnterpriseItem2, and EnterpriseItem3.

C

Advanced Query Capabilities

This appendix describes advanced query capabilities supported in the wt.query package. These capabilities support advanced SQL queries and you are assumed to be familiar with the functionality and behavior of advanced SQL statements.

QuerySpec

The QuerySpec contains the following attributes and APIs for building complex query expressions.

Descendant Query Attribute

By default, when a query contains a class, all concrete, persistable subclasses (that is, all classes that have an associated table in the database) are queried. Use the boolean descendant query attribute to control this behavior. When this attribute is false, only the specified class in the query is queried in the database. Note that an exception is thrown if the class is not concrete and persistable. Use this feature only when it is necessary to perform a query against a single table.

Single Column Expression in SELECT Clause

When a class is selected in the query result set, every column of that class is included in the SELECT clause so that a full object can be built and returned in the QueryResult. In some cases, only single columns are needed in a result set. When possible, single columns should be used since this is much more efficient and offers better performance. The following API supports single column expressions in the SELECT clause:

```
appendSelect(ColumnExpression a_column, int[] a_fromIndices, boolean  
a_selectOnly)
```

The fromIndices parameter is used to associate the column expression to classes in the query, if applicable. In general, a ColumnExpression may be associated

with zero or more classes in the From clause. See the following table to determine the size of the `a_fromIndices` array based on the type of `ColumnExpression`. For example, a single `ClassAttribute` would require one from index. A `SQLFunction` with two `ClassAttribute` arguments and a `ConstantExpression` argument would require two from indices. If no fromIndices are required, a null value can be passed as the argument. The `selectOnly` parameter controls whether the column expression should be returned as a result object. If true, the column expression is included only in the select and is not returned as a result.

The `ColumnExpression` parameter specifies the query column to append to the select clause. The following are concrete `ColumnExpression` implementations:

Column Expression	Description	Required From Indices
ClassAttribute	This class represents a class attribute that can be used in a SQL statement. Introspection information is used to determine the associated table and column.	1
SQLFunction	This class represents a SQL function within a SQL statement.	0 or more. This number is based on the sum of the required from indices of all arguments.
ConstantExpression	This class represents a constant in a SQL statement.	0
KeywordExpression	This class represents an expression that evaluates to a SQL keyword that can be used in a SQL statement.	0
TableColumn	This class represents a table column that can be used in a SQL statement. The exact table and column name specified are used directly in the SQL statement.	0

The following example builds a query for part numbers:

```
QuerySpec qs = new QuerySpec();
int classIndex = qs.appendClassList(wt.part.WTPart.class, false);
ClassAttribute ca = new ClassAttribute(
    wt.part.WTPart.class, wt.part.WTPart.NUMBER);
qs.appendSelect(ca, new int[] { classIndex }, false);
```

Note that when the `WTPart` class is appended to the query, the `selectable` parameter is false. The full object is not returned; only the number column is returned.

Results are still returned in the `QueryResult` object. Each element of the `QueryResult` corresponds to a row and is an `Object` array (that is, `Object[]`). In this example, the number column is at index 0 for each element. The actual Java type

for each result is based on the table column and the JDBC SQL-to-Java type mapping.

The behavior of queries for parent classes (that is, classes that have one or more persistable, concrete subclasses) is to execute SQL for each table. When only ColumnExpressions are included in the SELECT clause, all of these SQL statements are implicitly executed as a single UNION statement instead of multiple separate database queries.

Queries that include only column expressions still have Access Control applied. Internally, columns are added to the query to retrieve information needed for Access Control. When queries containing only columns are used as sub-selects as part of an ObjectReference query (described later in this appendix), then the Access Control columns are not added. This behavior is important to understand when using aggregate SQL functions. When these are used, the SELECT clause must contain only expressions with aggregate SQL functions (or the expression must be included in the GROUP BY clause. If Access Control is applied to such a statement, then it will result in invalid SQL.

Table Expression in FROM Clause

When a class is added to a query, the generated SQL includes the associated table for the class in the FROM clause of the query. It is also possible to append a TableExpression to the FROM clause as shown below:

```
appendFrom( TableExpression a_tableExpression )
```

The following are concrete TableExpression implementations:

ClassTableExpression	This class represents a class specification of a table that can be used in a SQL FROM clause. Introspection information is used to determine the associated table.
ClassViewExpression	This class represents a "view" of a class table that can be used in a SQL FROM clause. All descendent classes are part of this view (however, no additional sub-class columns are included). This class is useful for queries involving outer joins or group by because all sub-classes are treated as a single "table".
SubSelectExpression	This class represents a subselect that can be used in a SQL statement. The subselect is specified via a StatementSpec attribute.
ExternalTableExpression	This class represents a table that can be used in a SQL FROM clause. The exact table name specified is used directly in the SQL statement.

The following example builds a query against a non-modeled table named dual:

```
QuerySpec qs = new QuerySpec();
int fromIndex = qs.appendFrom(new
ExternalTableExpression("dual"));
TableColumn dummyColumn = new TableColumn("dual", "dummy");
SQLFunction currentDate = SQLFunction.new
SQLFunction(SQLFunction.SYSDATE);
qs.appendSelect(dummyColumn, new int[] { fromIndex }, false);
qs.appendSelect(currentDate, null, false);
```

Expression in WHERE Clause

The most common type of WHERE clause expression is a SearchCondition. However, other expressions can also be used. The following APIs support expressions in the WHERE clause:

```
appendWhere(WhereExpression a_expression, int[] a_fromIndices)
appendWhere(WhereExpression a_expression, TableExpression[]
a_tableExpressions, String[] a_aliases)
```

The following are concrete WhereExpression implementations:

SearchCondition	This class represents a search condition on a query. When appended to a QuerySpec, the values will be used in the SQL WHERE clause.
ExistsExpression	This class represents an EXISTS expression in a WHERE clause. A StatementSpec instance is used for the subselect.
CompositeWhereExpression	This class represents a number of WHERE expressions connected using a logical operator (i.e. AND/OR).
NegatedExpression	This class represents a negation of an expression in a WHERE clause. This class contains an aggregated WhereExpression that is preceded with a NOT when this expression is evaluated.

The fromIndices parameter is used to associate the WHERE expression operands with tables in the FROM clause. Similar to the appendSelect() method, the fromIndices array is based on the types of WhereExpression and ColumnExpressions used in those WhereExpressions. For example, a SearchCondition with a ClassAttribute and a ConstantExpression would require a

single from index. A CompositeWhereExpression containing three SearchConditions would require fromIndices array with size equal to the sum of the size needed for each SearchCondition.

The following example demonstrates the proper usage of the fromIndices. This code queries for parts and their associated alternate parts. A composite where expression is used with several criteria: the second through fourth characters of the associated part numbers are equivalent, the part name begins with "E", or the alternate part name begins with "E". This first section of code sets up the classes in the query, the select items, and the joins between the classes.

```
QuerySpec qs = new QuerySpec();
int partIndex = qs.appendClassList(wt.part.WTPartMaster.class, false);
int alternatePartIndex = qs.appendClassList(wt.part.WTPartMaster.class, false);
int linkIndex = qs.appendClassList(wt.part.WTPartAlternateLink.class, false);

// Define the attributes in the query
ClassAttribute partName =
    new ClassAttribute(wt.part.WTPartMaster.class, wt.part.WTPartMaster.NAME);
ClassAttribute alternatePartName =
    new ClassAttribute(wt.part.WTPartMaster.class, wt.part.WTPartMaster.NAME);
ClassAttribute partNumber =
    new ClassAttribute(wt.part.WTPartMaster.class, wt.part.WTPartMaster.NUMBER);
ClassAttribute alternatePartNumber =
    new ClassAttribute(wt.part.WTPartMaster.class, wt.part.WTPartMaster.NUMBER);

// Define constants used in the criteria
ConstantExpression subStringStart = new ConstantExpression(new Long(2));
ConstantExpression subStringEnd = new ConstantExpression(new Long(4));
ConstantExpression wildcardExpression = new ConstantExpression("E% [ ]");

// Add items to the select and join the classes
qs.appendSelect(partName, new int[] { 0 }, false);
qs.appendSelect(alternatePartName, new int[] { 1 }, false);
qs.appendJoin(linkIndex, wt.part.WTPartAlternateLink.ALTERNATES_ROLE, partIndex);
qs.appendJoin(linkIndex, wt.part.WTPartAlternateLink.ALTERNATE_FOR_ROLE,
    alternatePartIndex);
```

In this next section, the criteria are constructed and appended to the query. Note that the first SearchCondition uses two ClassAttribute instances. The corresponding indices must be added to the fromIndices array that is used in the appendWhere. Likewise, the second SearchCondition references the part class and the third SearchCondition references the alternate part class. Therefore, four fromIndices are required and each array element must correspond to the appropriate SearchCondition.

```
CompositeWhereExpression orExpression =
    new CompositeWhereExpression(LogicalOperator.OR);
orExpression.append(new SearchCondition(
    SQLFunction.newSQLFunction(SQLFunction.SUB_STRING,
        partNumber, subStringStart, subStringEnd),
        SearchCondition.EQUAL,
        SQLFunction.newSQLFunction(SQLFunction.SUB_STRING,
            alternatePartNumber, subStringStart, subStringEnd)));
```

```

orExpression.append(new SearchCondition(
    partName, SearchCondition.LIKE, wildcardExpression));
orExpression.append(new SearchCondition(
    alternatePartName, SearchCondition.LIKE, wildcardExpression));

qs.appendWhere(orExpression, new int[] {
    partIndex, alternatePartIndex, partIndex, alternatePartIndex });

```

The last API explicitly specifies table expressions and aliases for the WHERE expression operands. This API is used for correlated subselects. When using subselects, it is common to use correlated columns (that is, a join between a column in the outer select and a column in the subselect). This is supported using the appendWhere() API in which TableExpressions and aliases are passed explicitly. For WhereExpressions that do not involve a subselect, the TableExpressions and aliases are derived implicitly using the QuerySpec FROM clause and the specified indices.

The following example builds a query using an EXISTS clause and a correlated subselect. The query will return all PartMasters for which an alternate PartMaster does not exist. An alternate is represented by the WTPartAlternateLink class, which is a many-to-many association between PartMasters. The role A of the WTPartAlternateLink class specifies the current PartMaster and the role B specifies the alternate PartMaster. Following is the SQL for this query:

```

SELECT A0.*
FROM WTPartMaster A0
WHERE NOT (EXISTS (SELECT B0.ida2a2
                  FROM WTPartAlternateLink B0
                  WHERE (A0.ida2a2 = B0.ida3a5)))

```

The following code constructs the query specification. The outer select will return PartMaster objects.

```

QuerySpec select = new QuerySpec();
int partIndex = select.appendClassList(wt.part.WTPartMaster.class,
true);

```

The following code constructs the subselect. The alias prefix is changed to avoid conflicts with the outer select.

```

QuerySpec subSelect = new QuerySpec();
subSelect.getFromClause().setAliasPrefix("B");
int altIndex =
subSelect.appendClassList(wt.part.WTPartAlternateLink.class,
false);

subSelect.appendSelect(new ClassAttribute(
wt.part.WTPartAlternateLink.class, WTAttributeNameIfc.ID_NAME),
new int[] { altIndex }, true);

```

The following code explicitly sets up the TableExpressions and aliases, which are passed as arrays. The join will be from the outer select to the subselect so the outer

select values are placed in the arrays at index 0 and the subselect values are placed in the array at index 1. The arrays are then used to append the SearchCondition.

```
TableExpression[] tables = new TableExpression[2];
String[] aliases = new String[2];
tables[0] = select.getFromClause().getTableExpressionAt(partIndex);
aliases[0] = select.getFromClause().getAliasAt(partIndex);
tables[1] = subSelect.getFromClause().getTableExpressionAt(altIndex);
aliases[1] = subSelect.getFromClause().getAliasAt(altIndex);

SearchCondition correlatedJoin = new SearchCondition(
    wt.part.WTPartMaster.class, WTAttributeNameIfc.ID_NAME,
    wt.part.WTPartAlternateLink.class, WTAttributeNameIfc.ROLEA_OBJECT_ID);
subSelect.appendWhere(correlatedJoin, tables, aliases);
```

Finally, the negated EXISTS clause is appended to the outer select.

```
select.appendWhere(new NegatedExpression(new
    ExistsExpression(subSelect)), null);
```

Bind Parameters

Bind parameters are a database/JDBC feature to take advantage of database statement parsing and optimization. Bind parameters are a mechanism for replacing constants in a SQL statement with replacement parameters at execution time. For example, the following WHERE clause expression uses the constant 'Engine':

```
WTPartMaster.name = 'Engine'
```

This expression can be replaced with the following in the static SQL:

```
WTPartMaster.name = ?
```

and the value 'Engine' can be bound to the parameter ? at execution time.

On a subsequent execution, a new value, such as Cylinder, can be bound to that same parameter. If these two statements had used the constant value directly in the static SQL, each statement would have been parsed, optimized, and precompiled separately. When bind parameters are used, a single static SQL statement can be reused multiple times.

This bind parameter feature is implicitly supported when using the QuerySpec, SearchCondition, and other query classes. However, the bind parameters can also be explicitly accessed using the following APIs:

```
getBindParameterCount()
getBindParameterAt(int a_index)
setBindParameterAt(Object a_value, int a_index)
```

Query Limit

A QuerySpec attribute, "queryLimit", can be used to limit the results returned from a query. As the database results are processed, a count is kept for each item

in the result set. This count includes items filtered out due to Access Control. If the limit is reached, then a `PartialResultException` will be thrown. This exception will contain a `QueryResult` with the items that have been processed. This could be used in a situation where a client may choose to display these results after issuing a message that a query limit was reached.

SearchCondition

A `SearchCondition` represents a SQL WHERE clause expression of the following form:

<left side operand> <operator> <right side operand>

The following are examples:

```
MyTable.Column1 = 5
MyTable.Column2 LIKE "E%"
MyTable.Column3 = JoinTable.Column1
```

Operands can also be more complex, such as SQL functions or subselects. `SearchCondition` can use arbitrary `RelationalExpression` operands. The operands can be specified using the `SearchCondition` constructor or setter methods. The following are concrete `ColumnExpression` implementations:

ClassAttribute	This class represents a class attribute that can be used in a SQL statement. Introspection information is used to determine the associated table and column.
SQLFunction	This class represents a SQL function within a SQL statement.
SubSelectExpression	This class represents a subselect that can be used in a SQL statement. The subselect is specified via a <code>StatementSpec</code> attribute.
ConstantExpression	This class represents a constant in a SQL statement.
KeywordExpression	This class represents an expression that evaluates to a SQL keyword that can be used in a SQL statement.
RangeExpression	This class represents a range in a SQL WHERE clause.
DateExpression	This class represents a date constant in a SQL statement. This subclass of <code>ConstantExpression</code> is necessary to provide the special handling for date values.

ArrayExpression	This class represents an array of constants in a SQL IN clause.
TableColumn	This class represents a table column that can be used in a SQL statement. The exact table and column name specified are used directly in the SQL statement.

The following example builds a complex query to determine the WTPartMaster object with the oldest modify timestamp after a specified date cutoff. Following is the SQL for this query:

```
SELECT A0.*
FROM WTPartMaster A0
WHERE (A0.modifyStampA2 IN (SELECT MIN(B0.modifyStampA2)
FROM WTPartMaster B0
WHERE B0.modifyStampA2 > 'cutoff' ) )
```

The following code constructs the query specification:

```
Class targetClass = wt.part.WTPartMaster.class;
QuerySpec subSelect = new QuerySpec();
subSelect.getFromClause().setAliasPrefix("B");
int subIndex = subSelect.appendClassList(targetClass, false);
int[] fromIndicies = { subIndex };
ClassAttribute subModifyStamp =
    new ClassAttribute(targetClass, WTAttributeNameIfc.MODIFY_STAMP_NAME);
SQLFunction minFunction = SQLFunction.new SQLFunction(SQLFunction.
MINIMUM, subModifyStamp);
subSelect.appendSelect(minFunction, fromIndicies, false);
subSelect.appendWhere(new SearchCondition(subModifyStamp,
    SearchCondition.GREATER_THAN, DateExpression.newExpression(cutoff)),
    fromIndicies);

QuerySpec select = new QuerySpec();
int index = select.appendClassList(targetClass, true);
select.appendWhere(new SearchCondition(modifyStamp, SearchCondition.IN,
    new SubSelectExpression(subSelect)), new int[] { index });
```

Compound Query

A compound query is a SQL statement that combines more than one component query into a single SQL statement via a set operator. Set operators include UNION, UNION ALL, INTERSECT, and MINUS. A compound query is composed by specifying a set operator and adding component queries. The component queries are StatementSpec objects so nesting of compound queries is also supported.

Note: The current version of the Oracle JDBC driver contains a bug that prohibits using parentheses around component statements in a nested compound query. The setting of the wt.pom.allowCompoundParentheses property in the db.properties file controls whether parentheses are used. By default, this setting is false to avoid

the Oracle JDBC driver bug. This workaround could lead to unexpected results if the set operator precedence is significant.

The following example builds a compound query to return a specific PartMaster number and the numbers of all of its alternates. Note that only numbers are selected, not full objects. This is necessary because, if all subclasses are considered, the compound query statement must include all subclass tables. These subclass tables may contain additional columns that would make the select list for each statement incompatible with other component statements. SQL requires that each component query in a compound statement must have the same number and corresponding type in the select list. Following is the SQL for this query:

```
SELECT A0.number
FROM WTPartMaster A0
WHERE (A0.name = 'ENGINE')
UNION
SELECT A2.number
FROM WTPartMaster A0,WTPartAlternateLink A1,WTPartMaster A2
WHERE (A0.name = 'ENGINE') AND
      (A0.idA2A2 = A1.idA3A5) AND (A2.idA2A2 = A1.idA3B5)
```

The following code constructs the query specification. The first select constructed is for PartMasters with the name ENGINE.

```
QuerySpec partSelect = new QuerySpec();
int partIndex = partSelect.appendClassList(wt.part.WTPartMaster.class, false);
partSelect.appendWhere(new SearchCondition(wt.part.WTPartMaster.class,
    WTPartMaster.NAME, SearchCondition.EQUAL, "ENGINE"), new int[]
{ partIndex });
```

The next select is constructed for returning PartMaster alternates. An alternate is represented by the WTPartAlternateLink class, which is a many-to-many association between PartMasters. A join must be specified across this association from the original part to its alternates.

```
QuerySpec altSelect = new QuerySpec();
partIndex = altSelect.appendClassList(wt.part.WTPartMaster.class,
false);
int altIndex = altSelect.appendClassList(W
wt.part.WTPartAlternateLink.class,
false);
int altPartIndex =
altSelect.appendClassList(wt.part.WTPartMaster.class,
false);

altSelect.appendSelect(new ClassAttribute(
    wt.part.WTPartMaster.class, wt.part.WTPartMaster.NUMBER),
    new int[] { altPartIndex }, false);

altSelect.appendWhere(new
SearchCondition(wt.part.WTPartMaster.class,
    WTPartMaster.NAME, SearchCondition.EQUAL, "ENGINE"), new int[]
{ partIndex });
```

```

altSelect.appendJoin(altIndex,
wt.part.WTPartAlternateLink.ALTERNATES_ROLE,
partIndex);
altSelect.appendJoin(altIndex,
wt.part.WTPartAlternateLink.ALTERNATE_FOR_ROLE,
altPartIndex);

```

Finally, the compound statement is constructed using the two previous queries and the UNION set operator.

```

CompoundQuerySpec compound = new CompoundQuerySpec();
compound.setSetOperator(SetOperator.UNION);
compound.addComponent(partSelect);
compound.addComponent(altSelect);

```

Access Control Consideration

The use of some advanced SQL APIs can bypass Access Control. These situations are detected and an `AdvancedQueryAccessException` will be thrown. The following advanced query uses will cause this exception:

- Sub-selects (`wt.query.SubSelectExpression`)
- MINUS or INTERSECT Compound Statements (`wt.query.CompoundQuerySpec`)
- External Tables (`wt.query.ExternalTableExpression`)
- Aggregate Functions (`wt.query.SQLFunction`)

AVERAGE

MAXIMUM

MINIMUM

SUM

COUNT

- ROWNUM keyword (`wt.query.KeywordExpression`)

This is done to ensure that Access Control is not bypassed unknowingly. In some cases, the use of these advanced SQL features that bypass Access Control is legitimate. For these cases, the advanced checking can be disabled at runtime. Query specification classes support an "advancedQueryEnabled" attribute that can only be set from server side code. If applicable, the attribute should be set to true on the query instance that is passed to the PersistenceManager query/find API to allow these queries to be executed without throwing an exception.

```

// Use advanced APIs to build query.

// Disable checking of advance features
statement.setAdvancedQueryEnabled(true);

```

```
// Execute query with access control
```

```
PersistenceHelper.manager.find(statement);
```

The `find()` method executes the statement with access control. Therefore, Access Control related columns may be implicitly added to the select. For some advanced features, such as aggregate functions, INTERSECT, and MINUS, the addition of these columns can affect the expected results or cause a SQL exception. In these cases, to successfully execute the query, the server side, non-access controlled `query()` method should be used.

```
PersistenceServerHelper.manager.query(statement);
```

Sorting

Queries can be used to sort the result data at the database level. However, in general, database sorting should only be applied to paging queries and queries that involve only `ColumnExpressions`. Other types of queries may be implemented as several separate SQL statements so the sorting is only applied to the individual statements and not the complete query. Any `ColumnExpression` can be used as a sort column. The `OrderBy` item is used to pass the `ColumnExpression` to the `StatementSpec`. The `OrderBy` also indicates the sort order (ascending or descending) and optionally a `Locale`. If a `Locale` is specified, then any character based attributes are sorted with respect to that `Locale` using the database language support. For Oracle, this is the National Language Support (NLS) (see Oracle documentation for more information). Java `Locale` values are mapped to Oracle NLS linguistic sort names via `dbservice.properties` entries.

Sorting is supported for standard and compound queries via `QuerySpec` and `CompoundQuerySpec` methods.

```
QuerySpec.appendOrderBy(OrderBy a_orderBy, int[] a_fromIndicies)
```

```
CompoundQuerySpec.appendOrderBy(OrderBy a_orderBy)
```

The `QuerySpec` method validates the `ColumnExpression` contained in the `OrderBy` against the `QuerySpec`'s `FROM` clause. The `CompoundQuerySpec` method does not validate. Note that neither method handles appending the `ColumnExpression` to the `SELECT` clause of the statement. This still must be done via the `appendSelect()` method. In both cases, it is recommended that a column alias be set for each `ColumnExpression` that is contained in an `OrderBy`. Depending on the type of query and the number of subclasses involved, the actual SQL statements may not be valid if a column alias is not used. Note that the column alias must not be a SQL reserved word (e.g., "number").

The following example builds a compound query using sorting. The names of parts and documents are returned sorted by name. Following is the SQL for this query:

```
SELECT A0.bname sortName
FROM WTPart A0
UNION
SELECT A0.bname sortName
```

```
FROM WTDocument A0
ORDER BY sortName DESC
```

The following code constructs the query specification. The first component query is for Parts. Note the setting of the column alias.

```
String sortName = "sortName";

QuerySpec partQuery = new QuerySpec();
int classIndex = partQuery.appendClassList(wt.part.WTPart.class,
false);
ClassAttribute partName = new ClassAttribute(wt.part.WTPart.class,
wt.part.WTPart.NAME);
partName.setColumnAlias(sortName);
partQuery.appendSelect(partName, new int[] { classIndex }, false);
```

This next section constructs the Document portion of the query. The same column alias is used.

```
QuerySpec docQuery = new QuerySpec();
classIndex = docQuery.appendClassList(wt.doc.WTDocument.class,
false);
ClassAttribute docName =
new ClassAttribute(wt.doc.WTDocument.class,
wt.doc.WTDocument.NAME);
docName.setColumnAlias(sortName);
docQuery.appendSelect(docName, new int[] { classIndex }, false);
```

Finally, the compound query is constructed using these two component queries. The OrderBy is appended to the overall query. The default locale is used to sort the names with respect to the user's language.

```
CompoundQuerySpec query = new CompoundQuerySpec();
query.setSetOperator(SetOperator.UNION);
query.addComponent(partQuery);
query.addComponent(docQuery);
query.appendOrderBy(new OrderBy(partName, true
));
```

Join Support

Query joins are used for associating data contained in separate tables. Joins can be accomplished by using the PersistenceManager navigate methods or through adhoc WhereExpressions. The QuerySpec class also provides explicit support for appending a join to a query using link classes and roles defined in the Rose model. This offers the flexibility of the QuerySpec along with the simplicity of specifying query joins using model information. The following QuerySpec methods can be used.

```
appendJoin(int a_linkIndex, String a_role, Persistable a_source)
appendJoin(int a_linkIndex, String a_role, int a_targetIndex)
```

The following example builds a query that joins together the SubFolder and Part classes via the FolderMembership link. The query returns all folders and all of the

associated parts that are contained in the folder. The following code constructs the query specification. The first section adds the classes and the attributes that should be returned. The final two lines of code join together the classes using the modeled roles for the FolderMembership link class.

```
QuerySpec query = new QuerySpec();

int folderIndex = query.appendClassList(wt.folder.SubFolder.class, false);
int linkIndex = query.appendClassList(wt.folder.FolderMembership.class, false);
int partIndex = query.appendClassList(wt.part.WTPart.class, false);
query.appendSelect(new ClassAttribute(wt.folder.SubFolder.class,
wt.folder.SubFolder.NAME),
    new int[] { folderIndex } , false);
query.appendSelect(new ClassAttribute(wt.part.WTPart.class, wt.part.WTPart.NAME),
    new int[] { partIndex } , false);
query.appendJoin(linkIndex, wt.folder.FolderMembership.FOLDER_ROLE, folderIndex);
query.appendJoin(linkIndex, wt.folder.FolderMembership.MEMBER_ROLE, partIndex);
```

D

Evolvable Classes

Topic	Page
Background Information	D-2
General Externalization Guidelines.....	D-3
Hand-coded Externalization Guidelines.....	D-3
Migration Guidelines for Classes with Hand-coded Externalization.....	D-4
Examples of Generated Externalization Code for Evolvable Classes.....	D-4

Externalizable classes that implement the Evolvable interface are the Windchill classes that can be serialized into BLOB columns in the database. As the persistent structure of these classes changes, action may be required to maintain compatibility with previous versions that have been serialized into the database.

During the migration period (that is, at Windchill Release 4.0), all Externalizable classes have the methods necessary to manage class compatibility but, in the future, only Evolvable classes will have these features. Any modeled class that is intended to support being serialized into a BLOB database column must implement the Evolvable interface. Once Evolvable is implemented, the owner of the class must manage its compatibility from version to version.

The Persistent Data Service (PDS) will report any classes being serialized into the database that implement the NetFactor interface and not the Evolvable interface. This allows third party classes, such as Vectors, Hashtables, and so on, to be serialized into the database. This also allows modeled classes that do not implement NetFactor to be serialized into the database; however, we do not recommend this practice because it leaves the class exposed to serious data migration problems.

The best way to specify that a modeled class will implement the Evolvable interface is to set the *Serializable* property for the class to *Evolvable*. This property is on the Windchill tab of the class specification in Rose.

Background Information

As of Release 4.0, the generated externalization code reads and writes a data stream according to the following order, with field values for each class ordered alphabetically:

1. BottomClass.ID
2. MiddleClass.ID
3. TopClass.ID
4. TopClass field value1
5. TopClass field value2
6. TopClass field value...N
7. MiddleClass field value1
8. MiddleClass field value2
9. MiddleClass field value...N
10. BottomClass field value1
11. BottomClass field value2
12. BottomClass field value...N

To maintain externalization compatibility of a class from version to version, it is most important to understand the resulting stream layout for each version of a class. When the persistent signature of a class is being changed significantly, it may be helpful to map out the stream format for each version, in order to understand clearly what is necessary to read a previous version of the stream.

Beyond the fundamental understanding of the stream format, points in the following sections provide guidance concerning when and what kind of manual intervention is necessary to maintain compatibility.

General Externalization Guidelines

The following are general externalization guidelines:

- Inserting a parent into the hierarchy is handled automatically.
- Removing a parent from the hierarchy requires implementation of `readOldVersion`.
 - The old ID and fields must be read (removed) from the stream.
- Changing the persistent signature of a class requires implementation of `readOldVersion`.
 - Adding or removing fields.
 - Changing the name or type of a field.

Hand-coded Externalization Guidelines

The following are guidelines for hand-coded externalization:

- Use the version ID that is generated as the `EXTERNALIZATION_VERSION_UID` constant. This allows subclasses to use it for their own comparisons.
- If you want to control the `EXTERNALIZATION_VERSION_UID`, you must model it and specify its initial value in the model. A modeled attribute supersedes the one provided by the code generator.
 - To calculate this version UID, the code generator uses the name of the parent class, and the names and types of all the non-transient, persistent fields.
 - It is prudent to manually control this value if the hand-coded externalization is not related to the signature, or footprint, of the non-transient, persistent fields of the class. (This occurrence should be rare.)
- Do not model a `serialVersionUID` attribute with a value other than 1 as this disallows the ability to read old versions of the class.

Migration Guidelines for Classes with Hand-coded Externalization

The following are guidelines for the migration of classes with hand-coded externalization:

- First, check whether the code generated into the `readVersion` method looks the same as the code that is preserved in `readExternal`. If so, you should turn off preservation and just let it all be generated.
- If the class had modeled a `serialVersionUID` attribute, remove it from the model.
- Set the *Serializable* property on the Windchill tab of the class specification in Rose to *Evolvable*.
- If the class already implemented `readOldVersion`, multiple old versions will now need to be supported.
- If the class is not final, the following guidelines apply:
 - If the class was using something other than the generated version ID number, you must change the read/write to use the `EXTERNALIZATION_VERSION_UID` constant.

In addition, `OLD_FORMAT_VERSION_UID` should be ignored because it is incorrect for your class. To read in instances externalized in the old format, reference the version UID that was actually used.

You must also add support for the old ID to `readOldVersion()`.

- You must move code from `readExternal()` to `readVersion()`, and set `preserve=no` for `readExternal()`.

Examples of Generated Externalization Code for Evolvable Classes

This section contains examples of generated externalization code for evolvable classes.

Example of Generated Constants

```
static final long serialVersionUID = 1;
public static final long EXTERNALIZATION_VERSION_UID = 4445336297727427925L;
protected static final long OLD_FORMAT_VERSION_UID = 2252240915753719640L

// OLD_FORMAT_VERSION_UID is only valid for R4 instances of the class
```

Example of a writeExternal Method

```
public void writeExternal( ObjectOutputStream output )
    throws IOException {
```

```

    ///##begin writeExternal% [ ]writeExternal.body preserve=no
    output.writeLong( EXTERNALIZATION_VERSION_UID );
    super.writeExternal( output );

    output.writeObject( a1 );
    output.writeObject( a2 );
    output.writeObject( a3 );
    output.writeObject( list );
    output.writeObject((size==null - null:size.getStringValue()) );
    output.writeObject( timeline );
    output.writeObject( work );

    ///##end writeExternal% [ ]writeExternal.body
}

```

Example of a readVersion Method

```

protected boolean readVersion( MyItem thisObject, ObjectInput input,
    long readSerialVersionUID, boolean passThrough, boolean superDone )
    throws IOException, ClassNotFoundException {
    ///##begin readVersion% [ ]readVersion.body preserve=no
    boolean success = true;

    // current version UID
    if ( readSerialVersionUID == EXTERNALIZATION_VERSION_UID )
        if ( !superDone ) // if not doing backward compatibility
            super.readExternal( input ); // handle super class

        a1 = (String)input.readObject();
        a2 = (Date)input.readObject();
        a3 = (Xyz)input.readObject();
        list = (Vector)input.readObject();
        String size_string_value = (String)input.readObject();

        try { size = (MySize)wt.fc.EnumeratedType.toEnumeratedType(
            size_string_value ); }

        // in case it was old format
        catch( wt.util.WTInvalidParameterException e ) {
            size = MySize.toMySize( size_string_value );
        }
        timeline = (Timeline)input.readObject();
        work = (MyAddress)input.readObject();
    }
    else
        success = readOldVersion( input, readSerialVersionUID, passThrough,
            superDone );

    return success;

    ///##end readVersion% [ ]readVersion.body
}

```

Example of a readOldVersion Method

This method is generated virtually the same as readVersion to support backward compatibility with the previous stream format. To support backward compatibility for multiple releases, the method should include a conditional block for each old version that is being supported. Each condition should check for the version UID that was used in that version of the class, and the block that reads the fields should be the same set of read calls that they were used for that version. The generated OLD_FORMAT_VERSION_UID constant is only valid for R4 instances of the class. Fields that no longer exist can be read and discarded (not assigned). Fields that didn't exist for the version can be initialized in the manner necessary to put the object into a valid state.

```
private boolean readOldVersion( ObjectInput input,
    long readSerialVersionUID, boolean passThrough, boolean superDone )
    throws IOException, ClassNotFoundException {

    //##begin readOldVersion% [ ]readOldVersion.body preserve=no

    boolean success = true;

    // handle previous version
    if ( readSerialVersionUID == OLD_FORMAT_VERSION_UID ) {
        a1 = (String)input.readObject();
        a2 = (Date)input.readObject();
        a3 = (Xyz)input.readObject();
        list = (Vector)input.readObject();
        String size_string_value = (String)input.readObject();

        try { size = (MySize)wt.fc.EnumeratedType.toEnumeratedType(
            size_string_value ); }

        // in case it was old format
        catch( wt.util.WTInvalidParameterException e ) {
            size = MySize.toMySize( size_string_value );
        }

        timeline = (Timeline)input.readObject();
        work = (MyAddress)input.readObject();
    }

    else if ( !superDone ) {
        success = super.readVersion( this, input,
            readSerialVersionUID, false, false ); // reformatted stream-

        if ( success &&& !passThrough &&& // forced pass through to skip me
            // I have been inserted into hierarchy
            readSerialVersionUID != super.EXTERNALIZATION_VERSION_UID )
            // try mine again
            readVersion( this, input, input.readLong(), false, true );
    }
    else
        throw new java.io.InvalidClassException( CLASSNAME,
            "Local class not compatible:"
            + " stream classdesc externalizationVersionUID="
            + readSerialVersionUID
            + " local class externalizationVersionUID="
            + EXTERNALIZATION_VERSION_UID );
}
```

```
return success;  
//##end readOldVersion% [ ]readOldVersion.body  
}
```


E

GUI Design Process

This appendix describes the GUI design process used to create the user interface of Windchill applications. Although you are not required to follow this process to create applications with Windchill, we recommend that you use some formal design method. Even if you have already started design, you can use this process to validate your work.

Topic	Page
Overview of the Design Process	E-2
Gather User Requirements	E-4
Analyze Tasks	E-6
Design the GUI.....	E-13

Overview of the Design Process

There are three major phases in the design process, each of which contains specialized tasks that are described in more detail later in this appendix.

Phase 1: Gather User Requirements

- Define user profiles
- Define user requirements

During this phase, you determine who the users of the system are and what they need in order to perform their work. As you progress, continually validate your decisions by using this information about your users.

Phase 2: Analyze Tasks

- Define use cases
- Model task structure and sequences
- Develop task scenarios
- Develop storyboards

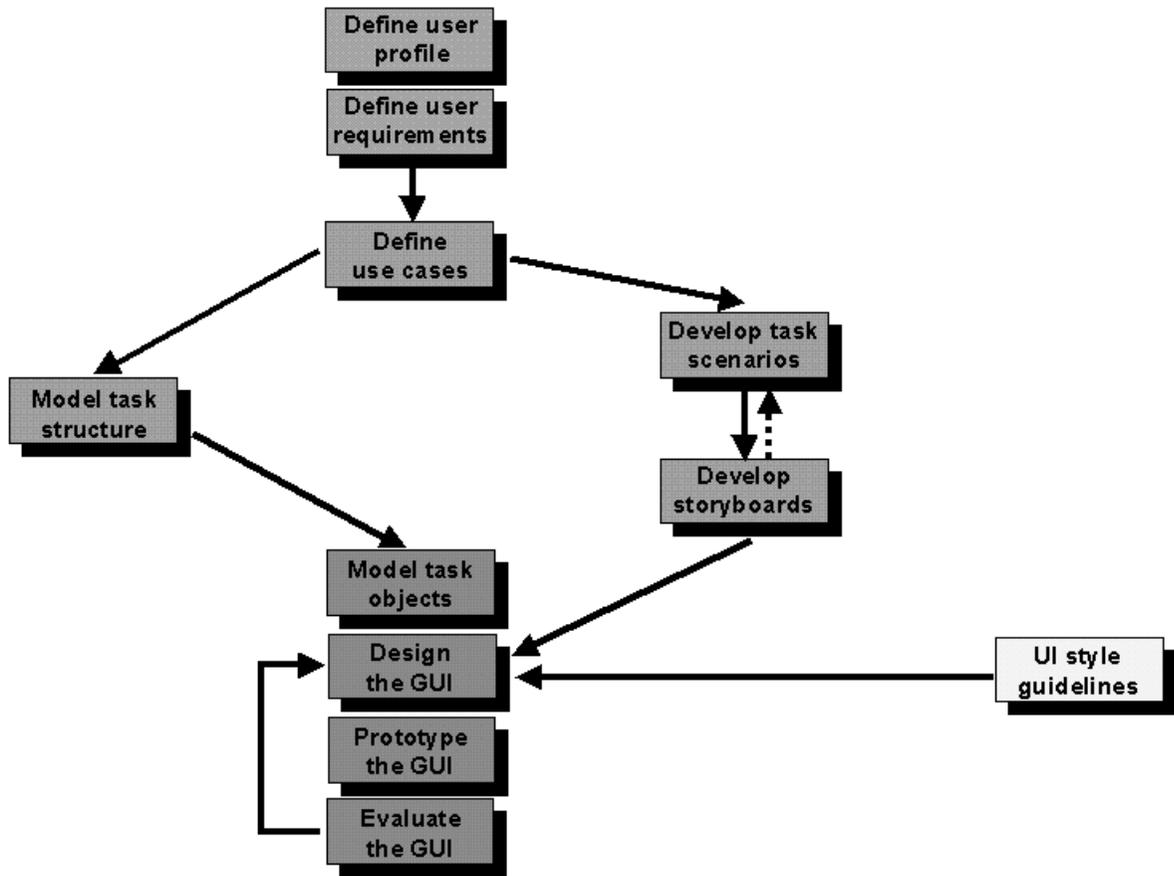
During this phase, you define user tasks at a high level and prioritize them. Do not think about design at this point. If design issues arise naturally, record them and go back to the high-level activities.

Phase 3: Design the GUI

- Model task objects
- Design the GUI
- Prototype the GUI
- Evaluate the GUI

The last three steps of this phase (designing, prototyping, and evaluating the GUI) are iterative steps and are typically repeated at least twice.

The following figure illustrates the process.



Overview of the GUI Design Process

The first steps in the GUI design process are to define the users and their requirements. Profiles of the users and their requirements are used as input when modeling user tasks and GUI objects. They are also used during the GUI evaluation step to validate the prototype.

Task structure models and task object models can be used as input to each other. Task structure models are used to identify objects within the tasks, and the resulting objects can be used to validate the task structure models. Both are then used as input to the GUI design step.

A highly desirable input to the GUI design is a UI style guide, which also addresses internationalization concerns. Use (or develop) your own corporate style guide, use one already written for the platform for which you are designing, or use one of the many available commercially.

The last phase, actual development of the GUI, is an iterative process of designing the GUI, developing a prototype, evaluating the prototype, and then redesigning

based on the evaluation. These steps overlap in that they are sometimes done in parallel and may merge with each other.

Gather User Requirements

Define User Profiles

A user profile describes a type of user, for example, an administrator. An application can have more than one type of user. For example, in a document management application, you could have an administrator of documents, an author of documents, and a reader of documents. Each would use the application differently, have different characteristics (with some overlap possible, of course), and, therefore, have a different user profile.

A typical user profile should include the following information:

- Name of the user class (type)
- The breadth or depth of product knowledge
- Tasks performed by the user
- User's experience and skills in the task domain
- User's experience and skills with Windchill
- Frequency of the user's use of the system
- Other applications used by the user
- Relevant internationalization issues

The process to define user profiles is as follows:

1. To determine who will use the system, talk with users and domain experts. Ask the following questions:
 - Who are the end users of the system-
 - What characteristics and knowledge do these users have-
 - What is their pattern of use-
 - What is their work environment-
 - What is their hardware environment-
 - What other applications do they interact with-
2. Group the users into types based on their patterns of use and other relevant characteristics.
3. Write user profiles including the information given earlier in the section.
4. Determine what percentage of the total user base each type of user represents.

As an example, assume a Help Desk has two types of users: customers and product support analysts. Following is a typical user profile of customers:

- Registered users of Windchill products.
- Are using the system to report product defects, make a product-related inquiry, retrieve information about new or upcoming product releases, or obtain marketing and sales information.
- Have a wide range of expertise with Windchill products, from first-time users to infrequent users to "power" users.
- Have a wide range of expertise with computer applications in general, from novice to expert. (It would be even more helpful to explicitly define novice and expert at this point.)
- Are experiencing varying levels of frustration, from not frustrated to extremely frustrated.

A GUI designer might look at the fourth bullet, for example, and start to think of designing a very simple GUI for novice users while building in shortcuts and other accelerators for expert users.

Define User Requirements

User requirements are often developed in parallel with system requirements, but they are not the same. User requirements are written from the perspective of how users get their work done. They do not address how the system will operate in order to support that work.

However, when you define user requirements early in the process, they can be used very effectively as input for defining system requirements.

The process to define user requirements is as follows:

1. Define the requirements for each type of user in your user profiles. To determine these requirements, talk with users and domain experts. Whenever possible, speak with the actual users to understand their needs and perspective.
2. Clarify and expand the list of requirements. Continually validate and re-validate the list with users and domain experts as you add and further refine the requirements.
3. Categorize the list of requirements according to types of users (if more than one).
4. Prioritize the list considering the following questions:
 - Which requirements are critical to success-
 - What requirements will increase usability-
 - Which requirements are achievable for this release or a future release-

- Are the target levels correct-
- What are the minimum requirements-

The following tables show a subset of the user requirements for the customer user type.

For incident reports

Requirement	Priority
Report an incident via the Web	High
View attributes of an incident report	High
Query incident reports	Medium
Report an incident via e-mail	Low

For product information

Requirement	Priority
Make inquiry about product customer is registered for	Medium
Make inquiry about any existing product	Medium
Download answers to local file system	Low

To create user requirements, first list the requirements with no organization or priority assigned. Validate them with typical users of the system. Remember that user requirements can be included for usability as well as functionality.

Next organize the requirements loosely in terms of function. In this example, a number of requirements are related only to incident reports, with a smaller number related only to obtaining product information.

Finally, prioritize the requirements within each category.

Analyze Tasks

A task is a human activity that achieves a specific goal. Examples of tasks are: creating a document, finding a file, generating a bill of materials, or updating user information.

Modeling user tasks starts with very simple, brief descriptions of individual tasks and evolves to detailed descriptions of how the user interacts with the system.

Modeling tasks involves the following steps:

- Define use cases
- Model task structure and sequence
- Develop task scenarios
- Develop storyboards

Define Use Cases

A use case is a short, often one-sentence description of a discrete user task.

The process to define a use case is as follows:

1. To identify use cases, use as input the user requirements defined earlier. If you do a complete set of use cases, every user requirement should be accounted for. They need not map one-to-one, however. A use case can incorporate more than one user requirement.
2. Categorize use cases by user type. Within the set of use cases for each user type, you can categorize further by the type of task.
3. Prioritize use cases to determine which ones to expand into task models and scenarios. To prioritize, consider the following factors:
 - What are the priorities of the contributing user requirements
 - What is the breadth of the use case- Does it encompass more than one user requirement
 - How frequently will the task described by this use case be performed- How critical is it to users' work-
4. Validate the completeness of your set of use cases by going back to user requirements.

Again, this example shows a subset of the use cases for the customer user type.

Use Case	Priority
<i>General</i>	
Log on to Help Desk via Web	High
<i>Incident Reports</i>	
Create an incident report via the Web	High
Download copy of incident report	Medium
Print incident report	Low
<i>Product Information</i>	
Make inquiry about a product	Medium
Print answers	Low

Every user requirement previously defined should be accounted for in the use cases (although this slide is a subset and may not show every entry actually in the use cases).

User requirements, however, do not have to map to use cases one-to-one. For example, the two user requirements shown earlier:

- Make inquiry about product user is registered for.
- Make inquiry about any existing product.

Model Task Structure and Sequence

A task structure model is an abstract model of the structure of a task that shows the subtasks a person performs to achieve the task's goal. For example, updating a report consists of the subtasks of retrieving, modifying, and then saving the report. The task structure model also shows the sequence of the subtasks performed by the user and the system's response to those actions.

A task structure model helps provide an understanding of what the user is trying to achieve (the goal), what the user actually does to achieve that goal (the subtasks), and the environment in which the user works (the task context). Task structure models are where you begin to identify contingencies, such as alternative actions and error handling. They also help identify task objects in the design phase.

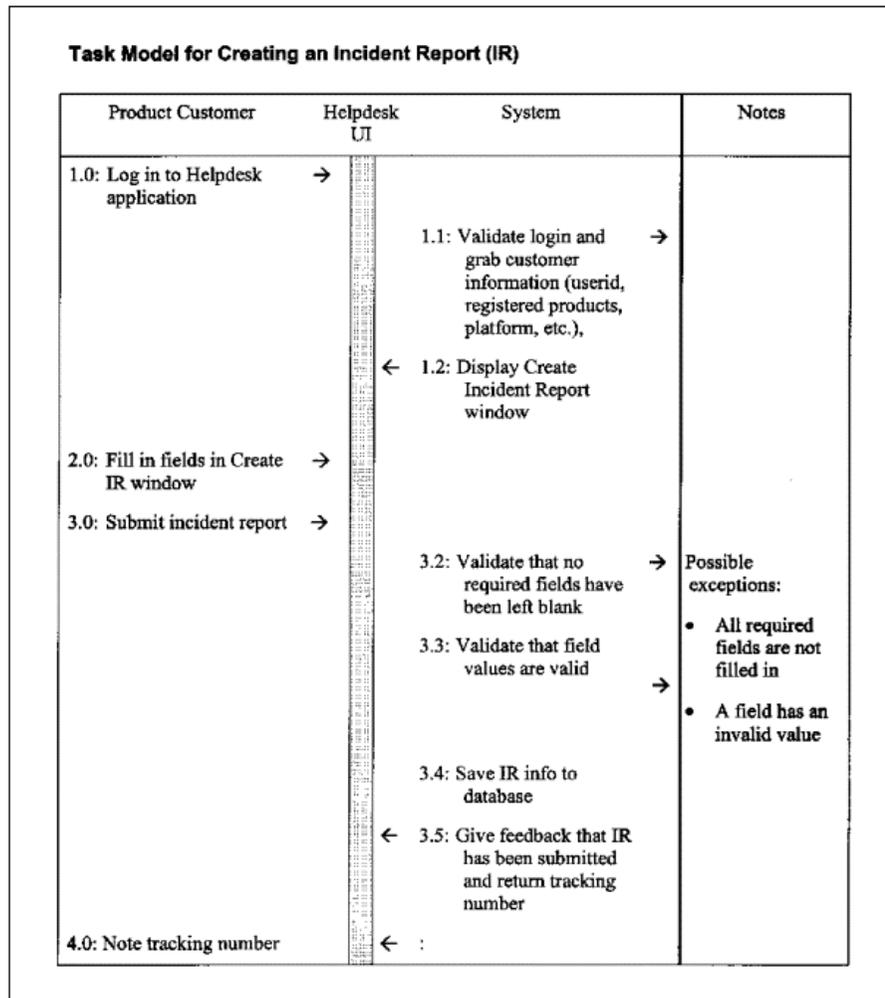
Keep user actions and system responses in the task structure model at a high level. For example, you might say the system "presents action choices" rather than the system "presents a menu of allowable actions". Do not start making implementation decisions at this point or let potential implementation problems

influence your model. If issues come up, you can note them for future discussion, but continue identifying high-level activities.

The process for modeling a task structure is as follows:

1. Determine which use cases are critical and should be carried forward in 4 to 8 task structure models.
2. Analyze each task into a sequence of subtasks, each with a well-defined endpoint. Determine whether there are any dependencies among the subtasks.
3. Develop the task diagram. Decompose subtasks only to the level of logical actions on objects in the system (for example, find a document or create a user). Do not model details of the user interface.
4. Enrich the task structure model by considering contingencies. For example, what happens if the user makes an error- What happens if information is missing- Are there alternative actions- Does the task ever vary-
5. Validate the task structure model by getting feedback from users or domain experts other than those who provided the input. Observe task scenarios being performed and ensure that they map onto the task structure model. Consider the following questions:
 - Is the set of task models complete- That is, have all significant user activities been modeled-
 - Do users consider the models realistic representations of their tasks-
 - Do the models account for errors, alternative actions, task variants, and other contingencies-

You can model tasks in several different ways. The following figure shows an interaction sequence model.



Task Model as an Interaction Sequence Model

The goals of this activity are to capture:

- Operations associated with a task, including user-initiated actions and system responses.
- The sequence of operations
- Points where exceptions and contingencies are likely to occur.

Develop Task Scenarios

A task scenario starts with a use case and expands it to create a real-world example, complete with named users and values for objects with which they want

to interact. As a very simple example, a use case of "Create a file" could become a scenario of "Pat wants to create a file named data1."

The purpose of task scenarios is to describe the situation or context for a task, including the business context, the current state of the system, and values for any user input. They also describe how a user performs the task in this scenario in terms of a sequence of user actions. In this sense, a task scenario represents an abstract interaction design: specific in terms of the interactions but abstract in terms of the actual user interface. It defines how the user interacts with objects in the system without referring to the design of the user interface.

You use information about the frequency of actions to identify how and how often users need to navigate from one interface element to another, from one screen to another, or from one application to another. You can use information about the sequence of actions as input to design decisions about the order of a set of screens, or even the layout of a set of controls, where the layout is affected by the sequence of actions.

The process to develop task scenarios is as follows:

1. Choose which task structure models to develop as scenarios. Typically you develop 4 to 8 scenarios. Choose those that have the highest priority, represent the greatest breadth of user requirements, and are the most critical and frequently used.
2. Make the task structure models more complete and realistic by adding dummy data, such as user names and specific values.
3. Include interaction sequences; that is, add user actions and system responses.
4. Account for contingencies, such as incomplete information or an error by the user.

This process may seem similar to modeling task structure but includes specific values and real-world situations. If, in the interest of time, only one can be done, task scenarios are usually preferable.

The following is an example of a task scenario for creating an incident report.

Audrey Carmen is a senior software developer in the software development group at Acme, Inc. After repeated attempts, she was still having trouble with the new release of their debugger, ProgramPerfector 4.0, v2. She had run her code through ProgramPerfector five times, and it still had bugs. She decided to file an incident report. Although some developers in Audrey's organization run ProgramPerfector on Solaris, Audrey and her team run it on Windows NT.

Audrey logs into the Help Desk, creates an incident report, enters all required information, and submits the report. She notes the tracking number so she can follow the status of her report.

As demonstrated in this example, a task scenario preserves the business context of earlier steps and adds specific values for attributes. For example, the user in this

scenario has a name (Audrey Carmen), and she has encountered a problem that will be submitted via an incident report.

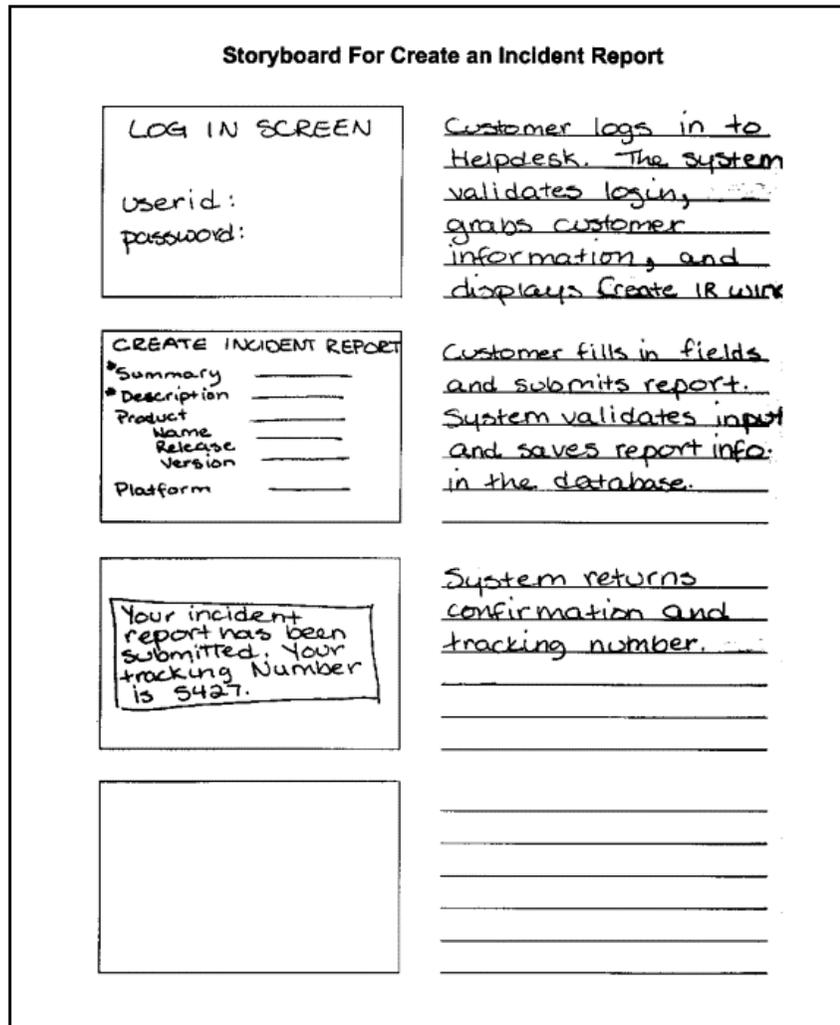
Develop Storyboards

A storyboard is based on one or more task scenarios. It is a sequence of images, usually drawn by hand on paper, that show interactions with the system and a sequence of screens. For example, the first image may be the action "log on" and the next image could be a screen labeled "Initial logon screen".

The sequence of screens conveys information about structure, functionality, and navigation. This information can be used in developing prototypes and validating user requirements.

Typically a storyboard is done in two columns with the images in the first column and explanatory text in the second column. The following figure is an example of

a typical storyboard. Note that it is a simple, handwritten document. The explanatory text is taken directly from task scenarios.



Storyboard Example

Design the GUI

Model Task Objects

Task objects represent objects with which the user will interact while performing a task. They form the basis of the user's mental model. That is, they represent how users conceptualize the objects in the system. They may or may not map directly to actual objects in the system.

A task object model models the business objects that users perceive when they interact through the GUI. Task object modeling helps ensure that the system behavior is understandable and intuitive to the user.

You can use the task structure models developed earlier as input to identify task objects, and then use the task objects to validate the task structure models.

The process to design a GUI is as follows:

1. Assess whether you need a separate user model for every user profile. For example, in a manufacturing environment, design engineers may be concerned with part structures while suppliers might perceive the system to be inventory. Both models can be represented in the UI.
2. Identify objects from discussions with users, the task structure models and scenarios created earlier, and the data model (if available). Discuss the following questions:
 - What is created or changed by this task-
 - What objects are used or referred to in performing this task-
 - Do users need to see and interact with the object to perform their tasks or should it be invisible to users-
 - Does the object group related information in a way that helps the user perform a specific task-
3. List the potential task objects. Analyze the relationships among the objects, and diagram the objects and relationships.
4. Define attributes for each of the task objects identified by determining what information the user can know about the object.
5. Define actions performed by the task objects.
6. Create a matrix mapping objects to the possible actions that can be performed on them.
7. Check for dynamic behavior by asking the following questions:
 - Can the actions be invalid depending on the prior state of the object-
 - Are there any constraints on the sequence in which the actions can occur-
8. Validate the task object model using methods similar to those used in validating the task structure model.

The following figure is an example of a typical task object model. Again, it is a simple, handwritten document.

Task Object Model for Create an Incident Report

Object	Incident Report	Product	Platform
Description	Customer-generated report containing questions, problems, or suggestions about a Netfactor product registered to the customer.	The Netfactor application registered to the customer.	Operating system on which customer runs the product.
Attributes	Summary Description	Name Release Version	OS
Actions	Create Submit Clear	Select	Select
Containment Hierarchy			
I'm in:	_____	Incident Report	Incident Report
In me:	Product Platform	_____	_____

Task Object Model

The objects described in the model must be represented on the screen. Creating a task object model helps you create objects that map directly to user tasks.

Creating a task object model is also an important step in streamlining the user interface. Following the model will result in an interface that contains every object needed to accomplish user tasks-and nothing more.

Design the GUI

In designing the initial GUI, consider the following questions:

- What views of objects are required for tasks-
- How should these views be represented in windows-
- What layout should be used- (This refers to layout guidelines such as the position of a company logo; it is not as fine-grained as designing the layout you would use in an IDE.)
- How does the user interact with objects-

- How does the user navigate through the windows-
- What controls are needed-
- How do the controls behave-
- What terminology should be used (for example, "update" or "modify")-

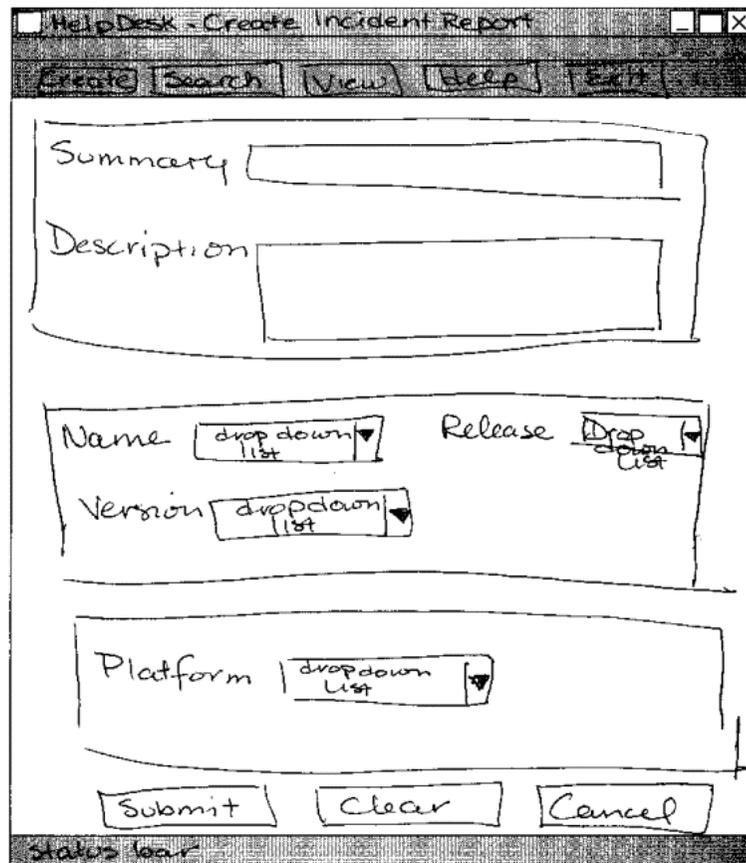
A style guide is very useful at this point. If you have a corporate style guide, use it or consider developing one. Otherwise, use one of those written for the platform on which you are designing.

Results of this step are window designs (including specification of interactive behavior) and window navigation design.

Prototype the GUI

The prototype is not intended to represent the final design of the GUI but is one of many iterations in its design and development.

We recommend that you start with low fidelity prototyping on paper as shown in the following figure.



Low Fidelity Prototype

Users feel much more comfortable suggesting changes if they do not feel you have to change code, and you can mimic sequences in the system much more easily. Show your prototype to users as often as possible without trying to make it perfect. This allows you to go through many iterations and make constant improvements and adjustments before creating real screens and including some functionality in a high fidelity prototype.

During prototyping, consider the following questions:

- How can the user perform the task scenarios using the GUI-
- What problems does the user encounter-
- Are additional views of objects required-
- Should the windows be restructured to support tasks better-
- Does the navigation between windows support tasks-
- How can the user's actions be simplified and streamlined-
- What improvements does the user suggest-

The following figure shows the high fidelity prototype that might result from this iterative process.

The image shows a window titled "WTHelpdeskApplet - Form Designer" with a toolbar at the top containing buttons for "Create", "Search", "View", "Help", and "Exit". The main content area is divided into sections:

- Customer:** A label above the main form area.
- Incident Details:** A section containing a "Summary:" label followed by a text input field, and a "Description:" label followed by a large text area with scrollbars.
- Product:** A section containing three dropdown menus: "Name:", "Version:", and "Release:".
- Operating System:** A section containing a "Platform:" dropdown menu.

At the bottom of the form are three buttons: "Submit", "Clear", and "Cancel".

High Fidelity Prototype

Evaluate the GUI

The best way to evaluate your GUI is to have one or more typical end users (as defined in your user profiles) attempt to use the prototype to perform task scenarios while designers observe.

The GUI should satisfy all the user requirements defined earlier and adequately support all tasks that were identified and modeled.

In evaluating the GUI, consider the following questions

- How usable is the GUI by the end users, in terms of user requirements previously specified-

- What usability problems did users encounter-
- Does the GUI provide adequate support to all types of users performing their full range of tasks-

Following evaluation, use the results as input back into the GUI design and prototype steps.

Use the following guidelines when designing a GUI:

- Does the GUI provide adequate support to all types of users performing their full range of tasks-
- Provide frequent task closure. That is, provide feedback that a task has been completed, for example in a status bar or popup message.
- Design for recognition rather than recall. Actions and objects that can be understood intuitively are preferable to those that must be learned.
- Support recovery from errors by using the following methods:
 - Prevent users from making errors.
 - Allow users to undo operations when possible.
 - Allow users to cancel operations when possible.
 - Provide error messages that tell the user what went wrong and how to recover.
- Support exploration by using the following methods:
 - Make the user feel sufficiently safe (that is, from catastrophic failures) and confident to explore.
 - Provide recognizable objects and actions that correspond to the user's mental model of the task, or are similar to objects and actions in other applications the user uses.

Index

A

- Access control
 - Client-side, 10-64
 - Property, 1-9
- Access control package, 6-4
- Accessor methods
 - Overriding, 8-9
- AssociationsPanel bean, 10-29
- AttributesForm bean, 10-37
- Authentication, A-1

B

- Background queuing package, 6-50
- Batch containers, 10-73
- Batch scripts, 1-2
- Bean
 - see Java beans
- bin directory, 1-2
- Business classes
 - Revision controlled business class, 5-6
- Business data types
 - Implementing, 8-7
- Business objects
 - Modeling, 3-1
- Business services
 - Implementing, 8-12

C

- cat files, 1-6
 - For sharing code, 1-6
- checkAttribute method, 8-10
- Class files, 1-4
- Class path environment variable, 1-7
- Classes
 - Document, 5-10
 - Folder resident business class, 5-4
 - Windchill Foundation classes, 3-10
 - Item, 3-10
 - Link, 3-11
 - Managed business class, 5-5
 - Part, 5-13

- Simple business class, 5-2
- WTOBJECT, 3-10
- ClassInfo.ser files
 - Location, 1-10
 - Location of, 1-5
- CLASSPATH environment variable, 1-7
- Client
 - Assigning life cycle template, 10-27
 - Assigning project, 10-27
 - Batch containers, 10-73
 - Client-side validation, 10-5
 - Clipboard support, 10-61
 - Development, 10-1
 - IDEs, 10-3
 - Invoking server methods, 10-5
 - Online help, 10-68
 - Refreshing data, 10-66
 - Threading, 10-64
- Client development
 - GUI design process, E-1
- Clipboard support, 10-61
- Code
 - Sharing, 1-6
- Code generation
 - Files, 1-6
 - See system generation tools.properties file, 1-10
- codebase directory
 - Details of, 1-4
 - Location, 1-2
- Configuration specification package, 6-73
- Content handling package, 6-8
- Content replication package, 6-14
- Control units, 1-6

D

- Database
 - Access set by properties, 1-10
 - db.properties.file, 1-10
 - Default table size, 1-10
 - Properties file, 1-2
 - Property file
 - See db.properties file

- db directory, 1-2
- db.properties file
 - Database access properties, 1-10
 - General description, 1-8
 - wt.pom.dbPassword property, 1-11
 - wt.pom.dbUser property, 1-11
 - wt.pom.serviceName property, 1-11
- Debug tracing, 1-9
- debug.properties file
 - General description, 1-8
- Development environment
 - Directory structure, 1-2
 - Environment variables, 1-7
 - Files, 1-2, 1-5
 - Property files, 1-8
 - Source code control, 1-6
- Directory
 - bin, 1-2
 - codebase
 - Details of, 1-4
 - Location, 1-2
 - codebase/wt, 1-5
 - db, 1-2
 - docs, 1-2
 - loadFiles, 1-2
 - logs, 1-2
 - RoseExtensions, 1-2, 1-7
 - search, 1-2
 - src
 - Details of, 1-5
 - Location, 1-2
 - src/wt, 1-6
 - Structure after installation, 1-2
- Doc package, 5-10
- docs directory, 1-2
- Document class, 5-10
- Document package, 5-10
- Domain administration package, 6-7

E

- EffectivityPanel bean, 10-33
- Enterprise package, 5-2
- EnumeratedChoice bean, 10-24
- Environment variables
 - Class path, 1-7
 - Rational Rose virtual path map, 1-7
 - SQL path, 1-7
- Events
 - Managing, 8-3
- Examples
 - Development process, 2-1

- Executable class files, 1-4

F

- Federation package, 6-21
- Folder resident business class, 5-4
- Folding package, 6-24
- FolderPanel bean, 10-35
- Windchill Foundation classes
 - Definitions, 3-10

G

- GUI
 - Design process, E-1

H

- HTML files
 - Location of, 1-5
- HTTP authentication, A-5
- HTTPUploadDownloadPanel bean, 10-57

I

- IDEs, 10-3
- Indexing package, 6-34
- Integrated development environment
 - See IDEs
- Internationalization, 11-1
- Item class, 3-10

J

- Java beans
 - AssociationsPanel, 10-29
 - AttributesForm, 10-37
 - EffectivityPanel, 10-33
 - EnumeratedChoice, 10-24
 - FolderPanel, 10-35
 - HTTPUploadDownloadPanel bean, 10-57
 - PartAttributesPanel, 10-18
 - PrincipalSelectionBrowser, 10-51
 - PrincipalSelectionPanel, 10-46
 - Spinner, 10-28
 - ViewChoice, 10-43
 - WTChooser, 10-23
 - WTContentHolder, 10-7
 - WTExplorer, 10-11
 - WTMultiList, 10-28
 - WTQuery, 10-21
- java.rmi.server.hostname property, 1-9
- JSP

L

- Life cycle
 - Assigning template, 10-27
 - Beans, 10-27
- Life cycle management package, 6-36
- Link class, 3-11
- loadFiles directory, 1-2
- Loading
 - Initial data, 1-2
- Localization, 11-1
 - Location of required files, 1-5
- Localizing
 - Text, 11-4
- Locking package, 6-41
- Logging
 - Default location for trace logs, 1-2
 - Enable/disable logging, 1-9
 - Trace messages
 - From method server, 1-9
 - From server manager, 1-9
- logs directory, 1-2

M

- Managed business class, 5-5
- Manuals
 - Location, 1-2
- mData files, 1-6
 - Location, 1-10
- mdl files, 1-6
- Method server
 - Logging trace messages, 1-9
- Model files, 1-6
- Modeling
 - Business objects, 3-1

N

- Notification package, 6-43
- Null authentication, A-7

O

- Online help, 10-68
- Oracle password property, 1-11
- Oracle service name property, 1-11
- Oracle user name property, 1-11
- Organization package, 6-46
- Ownership package, 6-47

P

- Packages
 - Access control, 6-4
 - Background queuing, 6-50
 - Batch container, 10-73
 - Configuration specification, 6-73
 - Content handling, 6-8
 - Content replication, 6-14
 - Control units, 1-6
 - Doc, 5-10
 - Document, 5-10
 - Domain administration, 6-7
 - Enterprise, 5-2
 - Federation service, 6-21
 - Foldering service, 6-24
 - Indexing, 6-34
 - Life cycle management, 6-36
 - Location of, 1-5
 - Locking service, 6-41
 - Notification, 6-43
 - Organization, 6-46
 - Overview, 6-2
 - Ownership, 6-47
 - Part, 5-13
 - Query, 7-12
 - Session management, 6-57
 - Version control, 6-64
 - Work in progress, 6-84
 - Workflow, 6-87
- Part class, 5-13
- Part package, 5-13
- PartAttributesPanel bean, 10-18
- PATH environment variable, 1-7
- Path environment variable, 1-7
- Persistence
 - Management, 7-1
 - Manager, 7-2
 - Query, 7-12
- Presentation logic, 10-3
- PrincipalSelectionBrowser bean, 10-51
- PrincipalSelectionPanel bean, 10-46
- Project
 - Assigning, 10-27
- Property files
 - db.properties file, 1-8
 - debug.properties file, 1-8
 - Editing, 1-8
 - service.properties file, 1-8
 - System Configurator, 1-8
 - tools.properties file, 1-8
 - wt.properties file, 1-8

Q

QueryResult, 7-15
QuerySpec, 7-13

R

Rational Rose, 3-2
 Virtual path map, 1-7
 Windchill extensions, 1-2, 1-7
 WT_EXTENSIONS entry, 1-7
 WT_STD_PACKAGES entry, 1-7
 WT_WORK entry, 1-7, 1-10
RB.java files
 Location of, 1-5
Refreshing client data, 10-66
Resource bundles
 Localizing, 11-4
 Location of, 1-5
 Online help, 10-68
Revision control, 5-6
Revision controlled business class, 5-6
Rose model components, 1-6
RoseExtensions directory, 1-2, 1-7
Runtime environment
 Files, 1-2, 1-4

S

search directory, 1-2
SearchCondition, 7-14
Server logic
 Overview of developing, 8-1
Server manager
 Logging trace messages, 1-9
service.properties file
 General description, 1-8
Services
 Access control, 6-4
 Background queuing, 6-50
 Batch containers, 10-73
 Configuration specification, 6-73
 Content handling, 6-8
 Content replication, 6-14
 Domain administration, 6-7
 Event management, 8-3
 Federation, 6-21
 Foldering, 6-24
 Indexing, 6-34
 Life cycle management, 6-36
 Locking, 6-41
 Managing, 8-2

Notification, 6-43
Organization, 6-46
Overview, 6-1
Ownership, 6-47
Query, 7-12
Session management, 6-57
Version control, 6-64
Work in progress, 6-84
Workflow, 6-87
Session management package, 6-57
Sharing code, 1-6
Signed authentication, A-7
Simple business class, 5-2
Source code management, 1-6
Source files, 1-2, 1-5, 1-6
Spinner bean, 10-28
SQL path environment variable, 1-7
SQL scripts, 1-2
 Location, 1-10
SQLPATH environment variable, 1-7
src directory, 1-2
 Details of, 1-5
System generation
 Overview, 9-1
 Using, 9-41

T

Task logic, 10-3
Threading, 10-64
tools.properties file
 General description, 1-8
 Use by code generator, 1-10
 wt.classRegistry.search.path property, 1-10
 wt.classRegistry.search.pattern property, 1-10
 wt.generation.bin.dir property, 1-10
 wt.generation.source.dir property, 1-10
 wt.generation.sql.dir property, 1-10
 wt.generation.sql.xxxTablesSize property, 1-10
ToolsSetup.bat, 1-2
Trace logs
 Default location, 1-2
Trace messages
 Logging
 From method server, 1-9
 From server manager, 1-9
Training materials
 Location, 1-2
Transactions, 7-16

U

User authentication, A-1

V

Validation

Client-side, 10-5

Verity Search, 1-2

Version control

Structuring, 6-80

Viewing, 6-82

Version control package, 6-64

ViewChoice bean, 10-43

Virtual path map

Rational Rose

Purpose, 1-7

WT_EXTENSIONS entry, 1-7

WT_STD_PACKAGES entry, 1-7

WT_WORK entry, 1-7, 1-10

W

Windchill extensions, 1-2

Work in progress package, 6-84

Workflow package, 6-87

wt directory

See also Packages

wt.access.enforce property, 1-9

wt.classRegistry.search.path property, 1-10

wt.classRegistry.search.pattern property, 1-10

wt.generation.bin.dir property, 1-10

wt.generation.source.dir property, 1-10

wt.generation.sql.dir property, 1-10

wt.generation.sql.xxtablesSize property, 1-10

wt.home property, 1-8

wt.logs.enabled property, 1-9

wt.manager.verboseClient property, 1-9

wt.manager.verboseServer property, 1-9

wt.method.verboseClient property, 1-9

wt.method.verboseServer property, 1-9

wt.pom.dbPassword property, 1-11

wt.pom.dbUser property, 1-11

wt.pom.properties property, 1-10

wt.pom.serviceName property, 1-11

wt.properties file

Double back slashes in path names, 1-8

Format of path names, 1-8

General description, 1-8

java.rmi.server.hostname property, 1-9

wt.access.enforce property, 1-9

wt.home property, 1-8

wt.logs.enabled property, 1-9

wt.manager.verboseClient entry, 1-9

wt.manager.verboseServer entry, 1-9

wt.method.verboseClient entry, 1-9

wt.method.verboseServer entry, 1-9

wt.pom.properties property, 1-10

wt.server.codebase entry, 1-9

wt.server.codebase property, 1-9

WT_EXTENSIONS entry, 1-7

WT_STD_PACKAGES entry, 1-7

WT_WORK entry, 1-7, 1-10

WTChooser bean, 10-23

WTContentHolder bean, 10-7

WTExplorer bean, 10-11

WTMultiList bean, 10-28

WTObject class, 3-10

WTQuery bean, 10-21