PTC®

PTC Global Services

Windchill 9 - Client Customization - PTC Student Guide

TS-GS-011

IP Policy

PTC IP Policy

The PTC IP usage and protection policy for PTC internal employees is still being defined.

Table of Contents

Windchill 9 - Client Customization - PTC Student Guide

Preface

This course explains how to customize the Windchill 9.0 user interface, including practical template customization which can be used immediately on an implementation. It reviews standard user interface technologies such as JavaScript, Ajax, JSTL and JavaServer Pages, then covers modifying existing Windchill pages and adding new pages, all using the Windchill client architecture. Finally, it covers configuring the Product Structure Explorer applet.

Course Objectives

- Confirm attendee proficiency with JSP, JavaScript, the Java Expression Language, Ajax and JSTL.
- Configure and customize the Windchill Client UI.
- Understand file structure and key files used in the Windchill client architecture.
- Identify, describe and use Windchill client architectural components.

Prerequisites

- Basic implementation experience using JSP, JavaScript, the Java Expression Language, Ajax and JSTL.

Agenda

- Introduction
- Generic UI Customization
- Standard UI Technologies
- Windchill Client Architecture
- Adding Actions
- Display a Basic Table
- Attributes
- Trees and Tables
- Advanced Component Configuration
- Constructing Wizards
- Information Pages
- Incorporating Pickers
- Customizing the Product Structure Explorer (PSE)

Provided Course Materials

- PTC Student Guide
- PTC Slides

Audience

- Global Services Technical Community
- VAR Technical Community

Module

1

Introduction

Welcome to class! This module is a discussion about course logistics and expectations. Please review them with your instructor.

Objectives

Upon successful completion of this module, you will be able to:

- Describe classroom logistics.
- Describe proper classroom expectations.

Lecture Notes

You may use the space below to take your own notes.

Class Introduction and Logistics

Take some time to meet each other and find out more about everyone's backgrounds and history.

Introduce yourselves by stating your:

- name
- home office
- title/role
- experience
- experience with PTC products (roles, products used, types of projects)

Classroom Logistics

Be sure that you are familiar with this building and its surroundings.

- Accessing the building

Note: If you have been assigned a badge, you are responsible for returning it on the last day of class.

- Toilets
- Printers and office supplies
- Snacks/Beverages
- Lunch
- Breaks
  - There will be frequent breaks.
  - If necessary, remind the instructor to give you time for a break.
  - Take advantage of the break- get up and walk around, let your mind relax, at least for a few minutes.
  - Depending on the pace of the class, there should be time available for completing project work or other necessary conference calls and appointments.
  - If there are any special breaks required, inform the instructor beforehand.

Classroom Rules

Because we are taking the time to come together as a group for this training, it is important to review some proper classroom etiquette and expectations before getting started.

Classroom Expectations

Question: What are some annoying or rude behaviors you have experienced in a class or meeting?

- Sidebar conversations
- Cell phone ringing or taking a call during class
- Questions unrelated to the topic
- Arguments instead of discussions
- Interrupting
- Inappropriate language
- Distracted by a computer (E-mail or Instant Messages)
- Starting late
- Running long

Conduct and Behavior Expectations

We are all representatives of PTC and should act professional at all times. Following is a list of some expectations for the duration of this class.

- Attendance and Attentiveness
  - Be ready to begin class on time.
  - No non-class tasks during classroom activities.
  - Notify the instructor (or class) in advance of absences.
- Show respect and avoid interruptions.
  - Save questions for the end of the topic, section or module.
  - Respect others in the class and in the office.

Remember, classroom rules optimize learning and will effectively get us home ASAP.

Commit to investing in your personal growth. Take the time to ask questions and help each other. When you are offering feedback to the instructor or other students, be honest and constructive and always show respect.

PTC guarantees a passing score. But the guarantee is voided if you:

- use cell phones during class.
- miss large portions of the training.
- use your laptop for tasks other than training.

Summary

After completing this module, you should be able to:

- Describe classroom logistics.
- Describe proper classroom expectations.

Module

2

Generic UI Customization

In this module, we investigate, some common UI fundamentals and best practices such as: creating icons, resource bundles, soft-attributes, role based UI functions, working with preferences, online help, display identity, and constructing URLs.

Objectives

Upon successful completion of this module, you will be able to:

- Customize UI branding.
- Localize UI text using resource bundles.
- Access Windchill preferences.
- Change logos.
- Define icons for business objects.
- Change styles and colors on pages.

Lecture Notes

You may use the space below to take your own notes.

UI Branding

The “welcome” message displayed under the second-level tabs is defined in *WT_HOME/src/com/ptc/core/ui/navigationRB.rbInfo*:

```
WELCOME.value=Welcome, {0}
WELCOME.constant=WELCOME
```

“Powered By” message and URL at the bottom of the UI is defined in *WT_HOME/src/com/ptc/netmarkets/util/utilResource.rbInfo*:

```
4.value=Powered by
4.constant=PWRD_BY
4b.value=http://www.ptc.com/appserver/mkt/products/home.jsp?&k=37
4b.constant=PWRD_BY_WC_WEBSITE
4c.value=Windchill
4c.constant=PWRD_BY_WC
```

Logo Icons

The top-right corner logo images are based on the installed products and are defined in *WT_HOME/codebase/netmarkets/css/nmstyles.css*:

```
/**** Application Logos for the header area ****/
.applLogo {background-repeat: no-repeat; height: 55px; background-position:34px;}
.wncApplLogo {background-image: url(../../netmarkets/images/logoWC.gif);}
.pjlApplLogo {background-image: url(../../netmarkets/images/logoPJL.gif);}
.pdmlApplLogo {background-image: url(../../netmarkets/images/logoPDML.gif);}
.proIApplLogo {background-image: url(../../netmarkets/images/logoPROI.gif);}
.atcmApplLogo {background-image: url(../../netmarkets/images/logoATCM.gif);}
/****/
```

Either edit *nmstyles.css* or replace the images themselves.

Colors and Fonts

Colors and fonts are defined in *WT_HOME/codebase/netmarkets/css/nmstyles.css*.

Changing Displayed Text in rbInfo Files

Most displayed text in Windchill PDMLink is controlled via locale-specific *rbInfo* (resource bundle) files.

-
- There is a standard *rbInfo* file, plus one for each installed locale.
- Although the OOTB *rbInfo* files are stored under the *WT_HOME/src* directory, do not edit the files there:
  1. It becomes difficult to identify changes made to the OOTB values.
  2. These files can be overwritten during a maintenance installation, thus losing modifications.
- Changes to Windchill text values should be stored in *rbInfo* files under the *WT_HOME/wtCustom* directory structure.
- Never delete OOTB values; instead, they can be made “unselectable”.
-
- To use text tailoring requires installing the “Displayed Text Tailoring” capability.

rbInfo files contain two main pieces of data.

1. a “key” - this is how data is stored in the database
2. a “value” - for each key, this is the value shown to the user

Note:

- It is necessary to have an internationalized Java SDK installed so that local-specific data can be compiled.
- Editing resource bundle files requires a **Custom** installation type for Windchill Services, Windchill PDMLink, Windchill ProjectLink, or Pro/INTRALINK.

Types of Resource Bundles

There are three kinds of text stored in *rbInfo* files.

1. text for enumerations (lists of values, known as EnumResourceInfo)
2. text for many messages, error messages, UI buttons or labels (known as StringResourceInfo)
3. display text for **modeled** business information such as classes, attributes or relationships (known as MetadataResourceInfo)

Note:

- Changes to displayed text may have an effect on resources that are incorporated in client *JAR* files.

Enumeration Resource Bundles

Examples of enumeration resource bundles include:

- Effectivity names in *wt/effectivity/EffectivityTypeRB.rbInfo*
- Meeting types in *wt/meeting/MeetingTypeRB.rbInfo*
- Part types in *wt/part/PartTypeRB.rbInfo*
- Quantity units for BOMs in *wt/part/QuantityUnitRB.rbInfo*
- Roles in *wt/project/RoleRB.rbInfo*
- Sources in *wt/part/SourceRB.rbInfo*
- Life cycle state names in *wt/lifecycle/StateRB.rbInfo*

Partial content of *WT_HOME/src/wt/project/RoleRB.rbInfo*:

```
ResourceInfo.class=wt.tools.resource.EnumResourceInfo
ResourceInfo.customizable=true
ResourceInfo.deprecated=false
ResourceInfo.UUID=8db111cc-5428-4d8b-b2e9-ed3a5b20baf0

# Entry Format (values equal to default value are not included)
# <key>.value=
# <key>.category=
# <key>.comment=
# <key>.argComment<n>=
# <key>.constant=
# <key>.customizable=
# <key>.deprecated=
# <key>.abbreviatedDisplay=
# <key>.fullDisplay=
# <key>.shortDescription=
# <key>.longDescription=
# <key>.order=
# <key>.defaultValue=
# <key>.selectable=

# Entry Contents

APPROVER.value=Approver
APPROVER.shortDescription=Workflow Approver role
APPROVER.order=10

APPROVER_LEVEL_1.value=Approver - Level I
APPROVER_LEVEL_1.shortDescription=Workflow Approver role for level one approval in OOTB template
APPROVER_LEVEL_1.order=11

APPROVER_LEVEL_2.value=Approver - Level II
APPROVER_LEVEL_2.shortDescription=Workflow Approver role for level two approval in OOTB template
APPROVER_LEVEL_2.order=12

ASSIGNEE.value=Assignee
ASSIGNEE.shortDescription=Workflow Asignee role
ASSIGNEE.order=40

AUDITOR.value=Auditor
AUDITOR.shortDescription=Workflow Auditor role
AUDITOR.order=20
```

Enumeration resource bundles can be found by searching the *WT_HOME/src* directory for *.rbInfo* files that contain the string “ResourceInfo.class=wt.tools.resource.EnumResourceInfo”.

Modifying Enumeration Resource Bundles

To modify enumeration resource bundles, use the WT_HOME/bin/enumCustomize tool.

enumCustomize creates the required files in the *wtCustom* directory and stores the text values replaced or added.

Note:

- Text changes to be used by the running product must be compiled into *WT_HOME/codebase*. Running the enumCustomize tool does this automatically, but only for enumeration resource bundles.
- There is also a command-line utility to compile resource bundles. Start a Windchill shell and execute the following command: ant -f bin/tools.xml bundle_custom -Dbundle.input=registry.
- For text changes that are to be used by any applets, additionally execute the following command: ant -f WT_HOME/codebase/MakeJar.xml.

When enumCustomize is used, *rbInfo* data from both *WT_HOME/src* (OOTB values) and *WT_HOME/wtCustom* (custom values) are read. After changes are made, all changes are written

to *WT_HOME/wtCustom*, and a compiled version of all OOTB and custom data is generated into *WT_HOME/codebase*.

Resource Bundle File Names

rbInfo files have the same name and extension in *WT_HOME/src* and *WT_HOME/wtCustom*. The compiled file in *WT_HOME/codebase* has the same name but the extension *RB.ser*.

Take two enumeration resource bundles as an example:

OOTB Values	Custom Values	Compiled Output
src/**wt/part/Part-TypeRB**.rbInfo	wtCustom/**wt/part/Part-TypeRB**.rbInfo	codebase/**wt/part/Part-TypeRB**.RB.ser
src/**wt/effectivity/Effectivity-TypeRB**.rbInfo	wtCustom/**wt/effectivity/EffectivityTypeRB**.rbInfo	codebase/**wt/effectivity/EffectivityTypeRB**.RB.ser

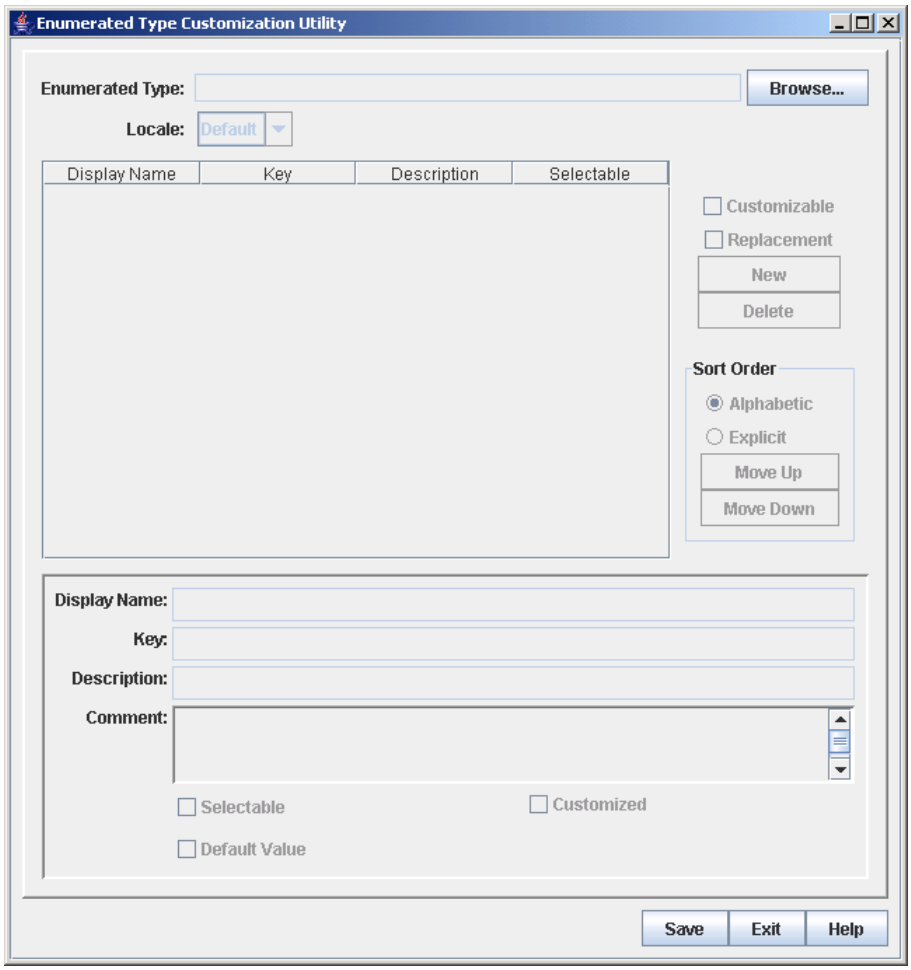


Figure 2-1: enumCustomize

enumCustomize works by selecting the compiled file under *WT_HOME/codebase*.

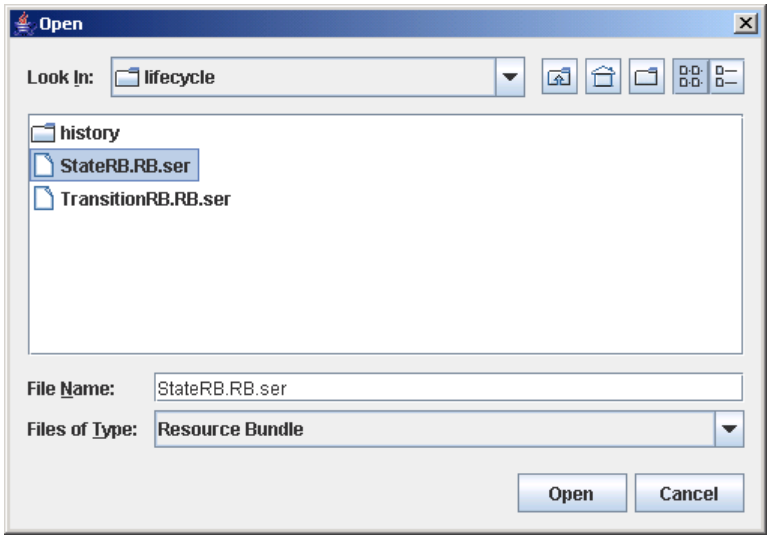


Figure 2-2: Selecting an Enumerated Resource Bundle

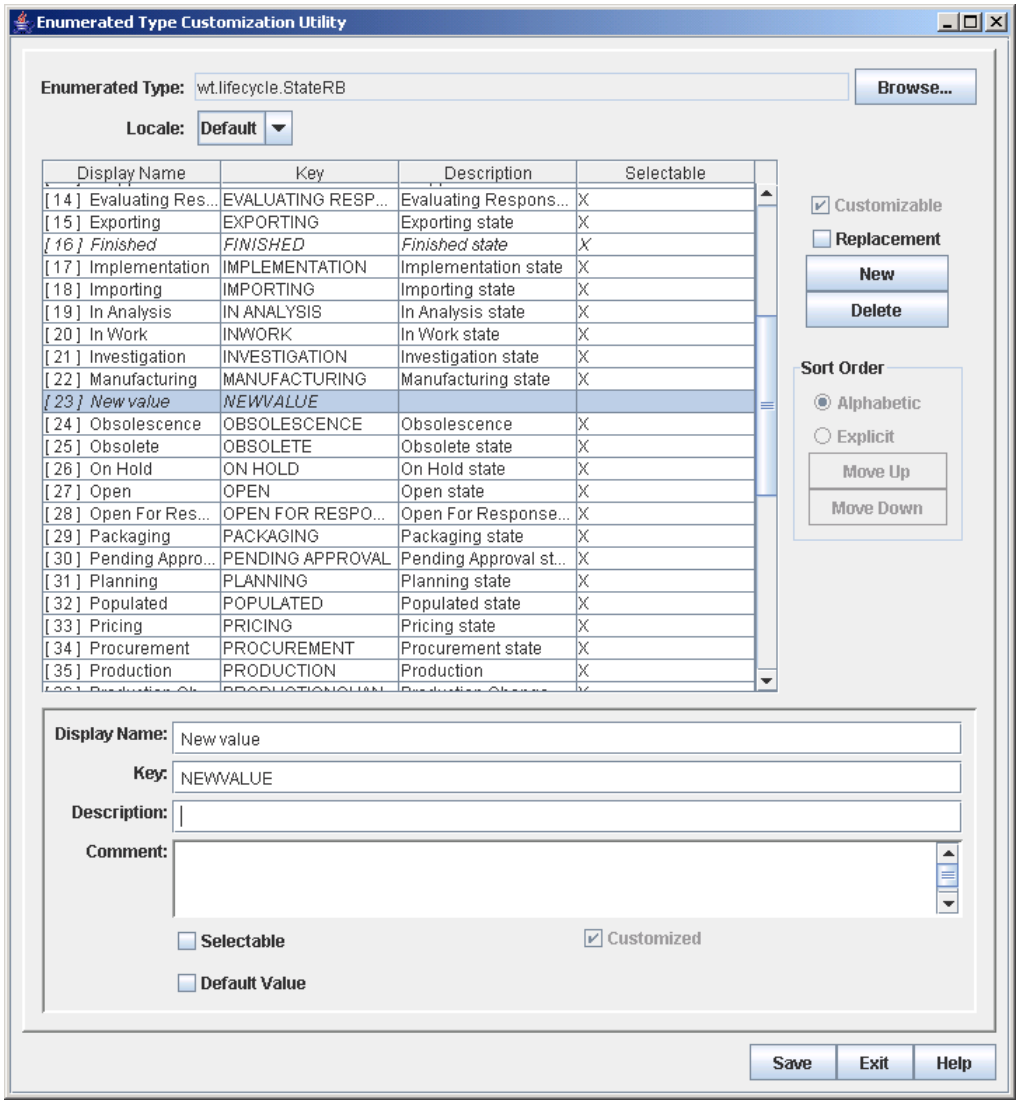


Figure 2-3: Adding a New Value with enumCustomize

String Resource Bundles

Examples of string resource bundles include:

- Change object data in *wt/clients/change2/Change2RB.rbInfo* or *wt/clients/change2/change2Resource.rbInfo*
- Document object data in *wt/clients/doc/DocRB.rbInfo* or *wt/doc/docResource.rbInfo*
- CAD Document object data in *com/ptc/windchill/enterprise/epmdoc/config/epmdocResource.rbInfo*
- Part object data in *wt/part/partResource.rbInfo* or *com/ptc/windchill/enterprise/part/partResource.rbInfo*

Partial content of *WT_HOME/src/wt/part/partResource.rbInfo*:

```
ResourceInfo.class=wt.tools.resource.StringResourceInfo
ResourceInfo.customizable=true
ResourceInfo.deprecated=false
ResourceInfo.UUID=7aaba588-6fa2-4b25-9bf7-c221f2066d21

# Entry Format (values equal to default value are not included)
# <key>.value=
```

```
# <key>.comment=
# <key>.argComment<n>=
# <key>.constant=
# <key>.customizable=
# <key>.deprecated=

# Entry Contents

0.value=The value for the argument "{0}" can not be null
0.constant=NULL_ARGUMENT

1.value=The view specified is not valid:  it must be persistent
1.constant=VIEW_NOT_PERSISTENT

2.value=More than one wt.part.WTPartConfigSpec returned, when only one was expected
2.constant=QUERY_WTPARTCONFIGSPEC_QTY

3.value=The number attribute must have a value
3.constant=NULL_NUMBER

19.value=The name attribute must have a value
19.constant=NULL_NAME
```

String resource bundles can be found by searching the *WT_HOME/src* directory for *.rbInfo* files that contain the string “ResourceInfo.class=wt.tools.resource.StringResourceInfo”.

Metadata Resource Bundles

Examples of metadata resource bundles include:

- Change model data in *wt/change2/change2ModelRB.rbInfo*
- Document model data in *wt/doc/docModelRB.rbInfo*
- CAD Document model data in *wt/epm/epmModelRB.rbInfo*
- Part model data in *wt/part/partModelRB.rbInfo*

Partial content of *WT_HOME/src/wt/change2/change2ModelRB.rbInfo*:

```
ResourceInfo.class=wt.tools.resource.MetadataResourceInfo
ResourceInfo.customizable=true
ResourceInfo.deprecated=false
ResourceInfo.UUID=4770a6b1-2041-4ad3-a393-c758d5d63c01


# Entry Format (values equal to default value are not included)
# <key>.value=
# <key>.category=
# <key>.comment=
# <key>.argComment<n>=
# <key>.constant=
# <key>.customizable=
# <key>.deprecated=
# <key>.abbreviatedDisplay=
# <key>.fullDisplay=
# <key>.shortDescription=
# <key>.longDescription=

# Entry Contents

AcceptedStrategy.value=Accepted Strategy

AcceptedStrategy.theChangeOrder2.value=Change Notice

AcceptedStrategy.theChangeProposal.value=Change Proposal

AddressedBy2.value=Addressed By

AddressedBy2.theChangeOrder2.value=Change Notice

AddressedBy2.theChangeRequest2.value=Change Request
```

Modifying either String or Metadata Resource Bundles

- Place custom values in the appropriate *rbInfo* files in the *wtCustom* directory.
- Use the corresponding OOTB *rbInfo* file in the *WT_HOME/src* directory as a template for the content.
- To modify string resource bundles, use a text editor, and enter only the values to override or add.
- Values that you are not changing should not be included.

Note:

- Text changes to be used by the running product must be compiled into *WT_HOME/codebase*.
- For string resource bundles, start a Windchill shell and execute the following command: ant -f bin/tools.xml bundle_custom -Dbundle.input=registry.
- For text changes that are to be used by any applets, additionally execute the following command: ant -f WT_HOME/codebase/MakeJar.xml.

Exercise 2-1: Create a New Life Cycle State

Objectives

- Create a lifecycle state using enumCustomize.
- Run ant -f bin/tools.xml bundle_custom -Dbundle.input=registry

Scenario

A customer has a life cycle state “Finished”, which does not exist OOTB. Technically, existing states such as “Approved” or “Completed” are similar, but for this customer “Finished” is a term long used and known within the company.

Detailed Description

Life cycle state names are enumerated resource bundle data, so enumCustomize will be used.

Step 1. Check for any existing edits to life cycle state names.

a. Using Windows Explorer, browse for *WT_HOME/src/wt/lifecycle/StateRB.rbInfo*. It should exist, but **not** contain “Finished”.
b. Using Windows Explorer, browse for *WT_HOME/wtCustom/wt/lifecycle/StateRB.rbInfo*. Depending on any previous customization, the directory and file may or may not exist, but the file should definitely not contain “Finished”.
c. Note the syntax of the file - the “key” is in upper case, and the value is mixed case. For consistency, follow the syntax of each file being edited.

Step 2. Start WT_HOME/bin/enumCustomize.

Step 3. Open the resource bundle file.

a. Browse to *wt/lifecycle* and open *StateRB.RB.ser*.
b. Note that this is the compiled version of the *StateRB.rbInfo* text file, not an *rbInfo* text file in *src* or *wtCustom*.
c. enumCustomize will open the appropriate *rbInfo* files in *src* and *wtCustom*.

Step 4. Add the new life cycle state value.

a. Select **New**.
b. For **Display Name**:, enter **`Finished`**.
c. For **Key:**, enter **`FINISHED`**.
d. For **Descritpion:**, enter **`Finished state representing the completion of a product`**.
e. For **Comment:**, enter **`Added on <TODAYS_DATE> by <YOUR_NAME>`**.
f. Select the **`Selectable`** check box.

Step 5. Compile the files into the *WT_HOME/codebase*.

a. Select **`Save`**.
b. In the **Build?** window, select **`Continue`**.
c. In the **Success** window, select **`OK`**.
d. Select **`Exit`** . In the **Exit without Saving?** window, select **`Yes`**. Note that changes were actually saved previously by selecting the **Save** button.

Step 6. Confirm the changes in the *rbInfo* file.

a. Using Windows Explorer, open the file*WT_HOME/wtCustom/wt/lifecycle/StateRB.rbInfo*.
b. This file should contain an entry for “Finished”.

Step 7. Rebuild *JAR* files.

a. If Windchill PDMLink is running, stop it.
b. Open a Windchill shell.
c. Execute ant -f WT_HOME/codebase/MakeJar.xml.

Step 8. View the changes in Windchill PDMLink.

a. Restart Windchill PDMLink.
b. Open the **Site > Lifecycle Administrator** and create a new life cycle.
c. The “Finished” state should be available as a life cycle state name.

Identifying Modeled Localizable Text

Some localizable text defined in resource bundles can be identified on a class-by-class basis using InfoReport located in *WT_HOME/bin*.

The syntax for the command is InfoReport [-x] <CLASS_NAME> where:

- the “-x” option causes XML reports to be produced instead of older-style formatted text reports
- “<CLASS_NAME>” is a fully qualified class, such as wt.part.WTPart

The output is a file <CLASS_NAME>.out in the directory defined by the property *wt.temp*.

The default value for *wt.temp* is *WT_HOME/temp*.

- For the command InfoReport wt.part.WTPart, the output would be written to *WT_HOME/temp/wt.part.WTPart.out*.
- For the command InfoReport -x wt.doc.WTDocument, the output would be written to *WT_HOME/temp/wt.doc.WTDocument.xml*.

Note: Be wary of the Master to Version relationship. For an object like Parts, some information is stored in the WTPartMaster object and some will be stored in the WTPart object.

Interpreting InfoReport Output

The output is separated into blocks.

The first block corresponds to elements directly defined on the input class (the argument supplied to InfoReport).

- If this first block contains “getClassname()”, the value of this line can be used to identify the resource bundle.
  - The “getClassname()” line will contain a class and package.
  - The package specifies the directory of the resource bundle.
  - The package name plus the suffix “ModelRB.rbInfo” identifies the resource bundle file.
  - The text to be edited is on the line beginning with the class plus the suffix “.value”.

Modeled Class Data Example

For example, the following is the beginning of the output file for InfoReport wt.part.WTPart:

```
Class                          : wt.introspection.ClassInfo
getDisplayName()               : Part
getStandardIcon()              : wtcore/images/part.gif
getOpenIcon()                  : wtcore/images/part.gif
getClassname()                 : wt.part.WTPart
...
```

The text “Part” can be modified in *WT_HOME/src/wt/part/partModelRB.rbInfo* in the line beginning “WTPart.value”.

- Subsequent blocks represent modeled elements related to the definition the input class. Each of these blocks begin with “getName()”.
- The “getName()” value identifies the modeled name of the element.
- For each block, the display text shown to a user is on the line “getDisplayName()”.

Related Element Blocks in InfoReport Output

There are two scenarios for identifying resource bundle files in “related element” blocks. If this “related element” block contains:

1. “getLinkInfo()” — this block refers to linked or related data
2. “getValue(WTIntrospector.DEFINED_AS)” — this block refers to a message

Note: Not all modeled elements are shown in the UI, so not all blocks have a display name.

Localized Link Text

- The “getLinkInfo()” line will contain a class and package.
- The package specifies the directory of the resource bundle.
- The package name plus the suffix “ModelRB.rbInfo” identifies the resource bundle file.
- The text to be edited is on the line beginning with the class plus “.” plus the value of “getName()” plus the suffix “.value”.

Note: To support other locales, make locale-specific copies of the files and add the localized values.

Example: Localized Link Text

For example, the following is a portion of a block of the output file for InfoReport wt.part.WTPart:

```
getName()                 : originalCopy
  getDisplayName()        : Original Copy
  getLinkInfo()           : wt.vc.wip.CheckoutLink
  ...
```

The text “Original Copy” can be modified in *WT_HOME/src/wt/vc/wip/wipModelRB.rbInfo* in the line beginning “CheckoutLink.originalCopy.value”.

Localized Text Message

- The “getValue(WTIntrospector.DEFINED_AS)” line will contain a package, class and modeled element name.
- The package specifies the directory of the resource bundle.
- The package name plus the suffix “ModelRB.rbInfo” identifies the resource bundle file.
- The text to be edited is on the line beginning with the class plus “.” plus the value of “getName()” plus the suffix “.value”.

Note: To support other locales, make locale-specific copies of the files and add the localized values.

Example: Localized Text Message

For example, the following is a portion of a block of the output file for InfoReport wt.part.WTPart:

```
getName()                                  : teamTemplateName
  getValue( WTIntrospector.DISPLAY_ENTRY ) : true
  getDisplayName()                         : Team Template Name
  ...
  getValue( WTIntrospector.DEFINED_AS )    : wt.team.TeamManaged.teamTemplateName
  ...
```

The text “Team Template Name” can be modified in *WT_HOME/src/wt/team/teamModelRB.rbInfo* in the line beginning “TeamManaged.teamTemplateName.value”.

Simple Method for Finding Localized Text

- Using the UI, identify the desired UI text to be modified.
- Search for **.rbInfo* files in the directory *WT_HOME/src* for files that contain the desired text string with a prefix of either “.value=” or “=”.

Exercise 2-2: Modify Localized Text Using InfoReport

Objectives

- Use InfoReport to determine the appropriate resource bundle file to modify.
- Modify localized text.

Scenario

A customer uses the word “Version” differently than in the Windchill UI; they want to change “Version” to “Revision”. Since the word “Revision” in Windchill refers to the combined version and iteration, the word “Revision” must also be changed. The customer wants to use “WTRevision”.

Step 1. Determine the appropriate attributes by running an InfoReport wt.part.WTPart.

a. Use of the “-x” option is up to the student.

Step 2. Find the affected model files for “Version”.

a. *getDisplayName()* equals *Version*
b. *getName()* equals *versionDisplayIdentifier*
c. *getValue(WTIntrospector.DEFINED_AS)* equals *wt.enterprise.RevisionControlled.versionDisplayIdentifier*
d. Edit *WT_HOME/src/wt.enterprise.enterpriseModelRB.rbInfo*, “RevisionControlled.versionDisplayIdentifier.value”.
e. Are there other blocks that contain the display text “Version”? For this demonstration exercise, the other instances can be ignored; in production, they should be updated, as well.

Step 3. Find the affected model files for “Revision”.

a. The *wt.part.WTPart.out* file may not be useful; try InfoReport wt.part.WTPartMaster.
b. The *wt.part.WTPartMaster.out* file may not be useful; try the next relevant super class/interface InfoReport wt.enterprise.RevisionControlled.
c. The *wt.part.WTPartMaster.out* file may not be useful; search for **.rbInfo* files in *WT_HOME/src* for “.value+Revision”.
d. The result should be *WT_HOME/src/wt/enterprise/enterpriseResource.rbInfo*, on the line **281.value=Revision**.

Step 4. Edit resource bundle files.

a. Use the files relate to *WT_HOME/src* as a template.
b. Create a file in the correct directory under *WT_HOME/wtCustom* with only the changed values.

Step 5. Deploy the customization.

a. Compile the changes.
b. Rebuild *JAR* files.
c. Restart Windchill.

Handling Icons for Business Objects

Icons for a modeled class are defined by properties in Rational Rose.

- **StandardIcon** is the icon when viewing the object in a list or a Properties page.
- **OpenIcon** is seldom used, but represents situations where the icon is a different visual key when an object is in an expanded or open state.
- For a modeled class, select the **Windchill** tab and set the **StandardIcon** and **OpenIcon** accordingly.

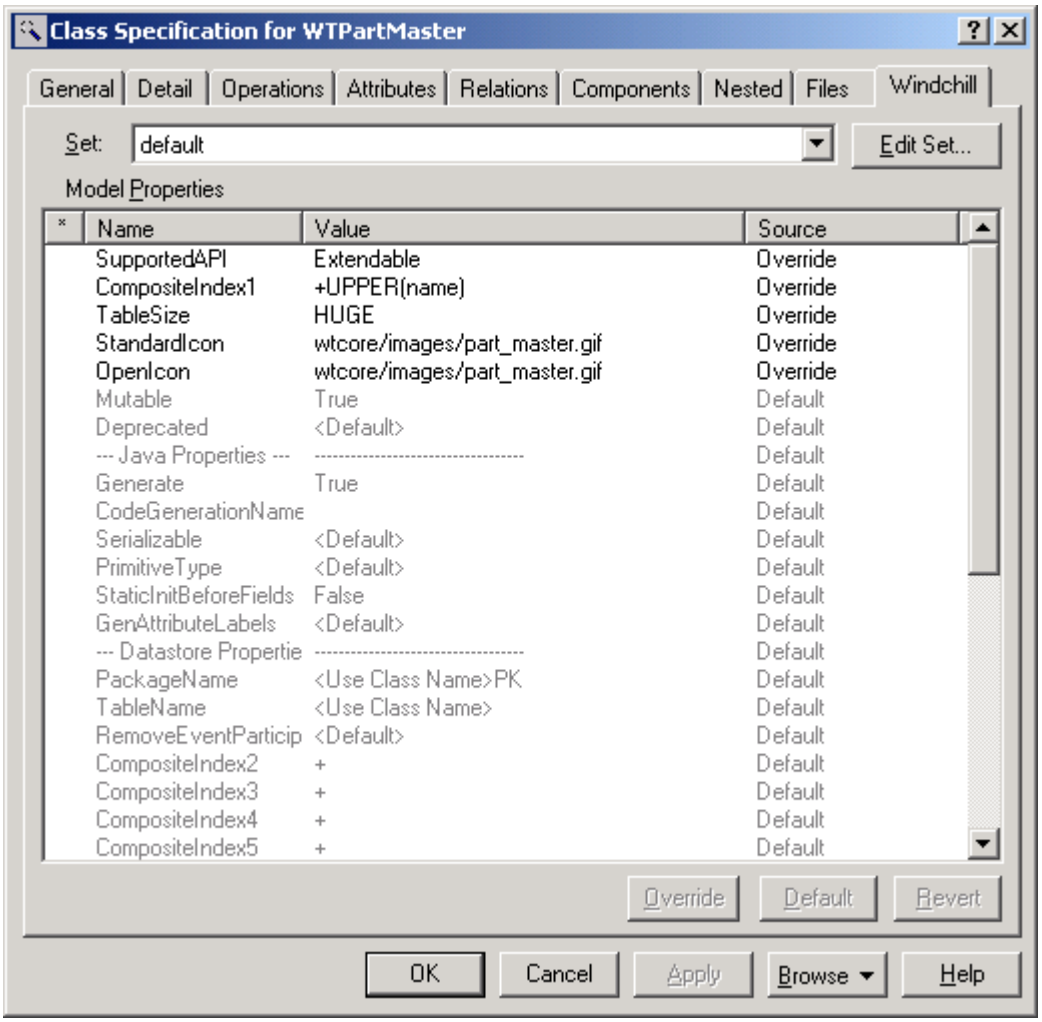


Figure 2-4: Icon definition for Parts in Rational Rose

Note: Values are relative to *WT_HOME/codebase*.

Icons for a soft type are defined by the **Icon** property of the soft type, as seen in the **Type and Attribute Manager**.

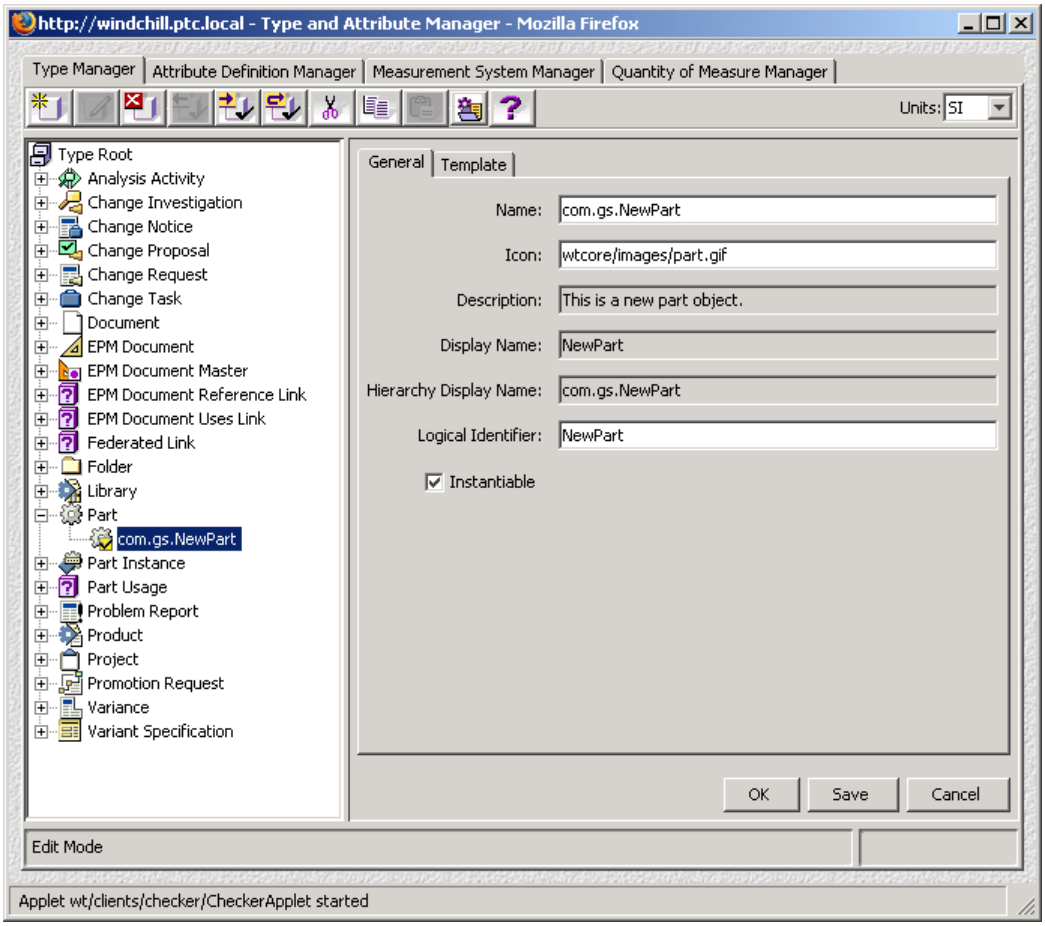


Figure 2-5: Attribute and Type Manager

Images should be *GIF* files that are 16 by 16 pixels.

The wt.clients.util.IconCache class is used to access icons:

```
IconCache icons = new IconCache( someWTContextObj );
Image img1 = icons.getStandardIcon( someWTObject );
Image img2 = icons.getOpenIcon( someWTObject );
```

Glyphs

Glyphs overlay object icons and indicate status such as:

- modified
- new
- checked out
- shared to Windchill ProjectLink project

Class inheritance determines what glyphs are overlaid on an object icon. For example, subclasses of wt.doc.Document are displayed based on their primary format and will also inherit the glyph overlays of RevisionControlled.

Preferences

Client behavior can be customized using **Preferences**.

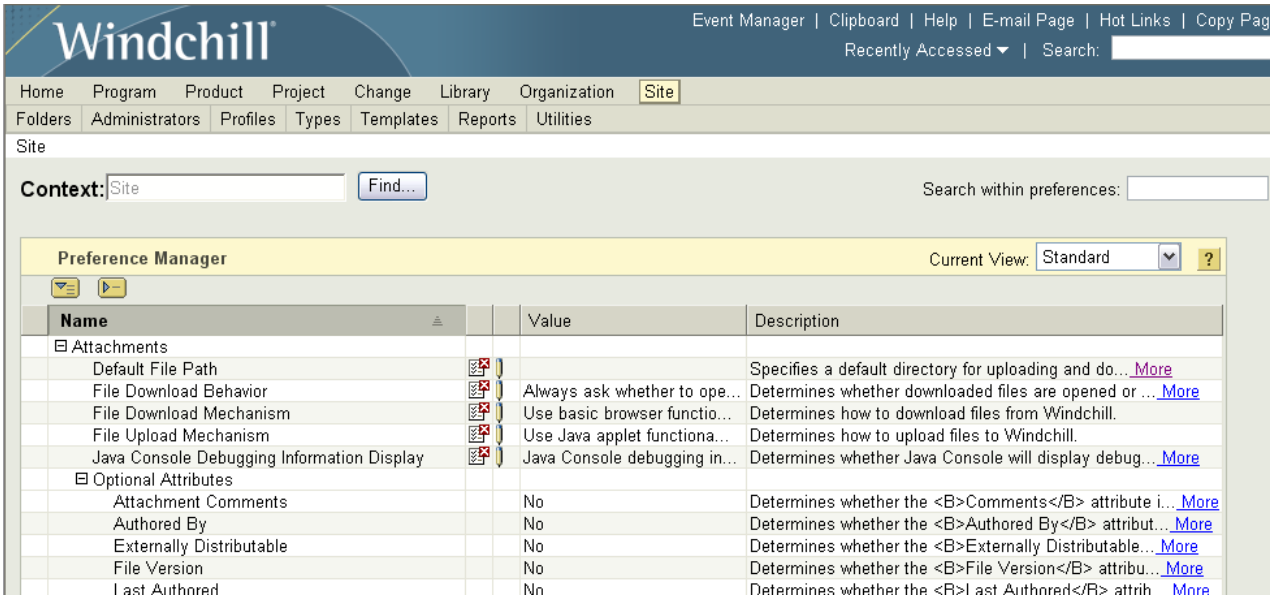


Figure 2-6: Preferences

A preference is uniquely identified by 3 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

Preference Context

Preferences require a context in order to separate individual and group preferences. The context is defined by the following attributes separated by a colon)

- Macro defined in wt.prefs.WTPreferences to be one of:
  - 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
- Descriptor - optional text defining the name of the context (e.g. WindchillEnterprise)

WT_HOME/codebase/wt/prefs/delegates/delegate.properties
controls the hierarchy in which delegates are called.
wt.prefs.delegates.DelegateOrder=$DEFAULT,$CONTAINER,$DIVISION:WindchillEnterprise,$USER
Preferences are initially loaded using *WT_HOME/loadFiles/preferences.xml*

Getting a Preference

You can get a preference

```
// 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() method gets the value for that preference key
String prefValue = myPrefs.get( "fileOperationType", "SAVE" );
```

Clearing a Preference

Clear a preference as follows:

```
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");
```

URL Factory

The URLFactory is a utility class provided to generate relative HREF objects.

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.

In order to generate fully qualified HREF objects, the easiest method is to set the request URI to null.

Example Code

```
URLFactory aFactory = new URLFactory( );
aFactory.setRequestURI( null );
String aHREF = aFactory.getHREF("wt/clients/login/login.html");
```

The above code would return the HREF “http://<hostname>/<web-app>/wt/clients/login/login.html”.

Summary

After completing this module, you should be able to:

- Customize UI branding.
- Localize UI text using resource bundles.
- Access Windchill preferences.
- Change logos.
- Define icons for business objects.
- Change styles and colors on pages.

Module

3

Standard UI Technologies

In this module you learn the basic technology behind the JSP Framework, the terminology, and the tools that can be used to query the framework.

Objectives

Upon successful completion of this module, you will be able to:

- Implement basic pages with JSP.
- Implement basic pages with JavaScript.
- Implement basic pages with the Java Expression Language.
- Implement basic pages with Ajax.
- Implement basic pages with JSTL.

Lecture Notes

You may use the space below to take your own notes.

Exercise 3-1: Create a Custom Simple Tag

Objectives

- Create a Java class that extends SimpleTag.
- Use a custom simple tag in a JSP page.

Scenario

Create a “Hello World of Tags” application for Simple Tags.

Step 1. Create the Java class by extending SimpleTagSupport.
Example:

```
package com.gsdev.client;

import javax.servlet.jsp.tagext.SimpleTagSupport;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;

public class HelloWorld extends SimpleTagSupport {

 public HelloWorld () {
  // TODO Auto-generated constructor stub
 }

 public void doTag() throws JspException {
  PageContext pageContext = (PageContext)getJspContext();
  JspWriter out = pageContext.getOut();
  try {
   out.println("Hello World of Tags");
  } catch (Exception e) {
   // Ignore.
  }
 }

}
```

Step 2. Create a“ Tag Lib Directive” file and place it in the *WT_HOME/WEB-INFf/tlds* directory.

a. The Tag Lib Directive creates the mapping from the tag to the class file including the code.

b. This is the content of *WEB-INF/tlds/demo.tld*.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0"
 xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd">

 <tlib-version>1.0</tlib-version>
 <uri>tlduri</uri>

 <tag>
  <name>tagname</name>
  <tag-class>com.gsdev.client.HelloWorld</tag-class>
  <body-content>empty</body-content>
 </tag>

</taglib>
```

Step 3. Create a JSP that references the tag created.

a. This is the content of *WT_HOME/codebase/wtcore/jsp/com/gsdev/HelloWorldTag.jsp*.

Example:

```
<%@taglib prefix="aprefix" uri="tlduri"%>

<html>
<head><title>JSP Page</title></head>
<body>

<aprefix:tagname/>

</body>
</html>
```

b. In the **taglib** directive line, the prefix could be anything as long as it matches the call to the prefix in the body of the JSP page.

c. In the call to the tag name, this text must match the name in the *TLD* file.

JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many Web applications, as simple tags.

The JSTL 1.2 Maintenance Release aligns with the Unified Expression Language (EL) that is being delivered as part of the Java Server Pages (JSP) 2.1 specification.

JSTL has support for common, structural tasks such as:

- tags for looping, conditionals, includes, passing parameters, URL rewriting, setting values, printing values
- tags for manipulating XML documents
- internationalization tags
- SQL tags

Before you write your own tag (or rely on scriptlets) check for an already existing tag in JSTL.

Note:

- JSTL 1.1 is not part of the JSP 2.0 spec
- JSTL core tags, by convention, are prefixed by a c
- Framework for integrating existing customization

```
<table>
<c:forEach var="movie" items="${movieList}">
<tr><td>${movie}</td></tr>
</c:forEach>
</table>
```

JSTL is exposed as multiple tag libraries:

	URIs	Recommended Prefix
Core	http://java.sun.com/jsp/jstl/core	c
XML	http://java.sun.com/jsp/jstl/xml	x
Internationalization	http://java.sun.com/jsp/jstl/fmt	fmt
SQL	http://java.sun.com/jsp/jstl/sql	sql
Functions	http://java.sun.com/jsp/jstl/functions	fn

For example, to use the JSTL “core” library in a JSP pages, use this taglib directive: <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>.

Custom Tags

What happens when standard JSP actions, EL, or JSTL doesn't have a function that is needed?

- Before writing scriptlet code, consider a custom tag.
- Custom tags use Java code in class files that can be called from JSP pages.
- The tag handler API has both a "Classic" and a "SimpleTag" versions.
  - Classic tags are harder to write and are more complex than needed in typical Windchill applications.
  - SimpleTag is a simpler implementation than Classic tags, and was created to address some of the issues when using Classic tags.
  - Basically implement doTag() in a class file; add information to a *TLD*, and imbed the tag in JSP pages.

To use a tag library in a JSP page, add **taglib** directives for each library, such as:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="APrefix" uri="ATaglibURI"%>
```

URI is any name that matches the **uri** element in the TLD file.

Note: With JSP 2.0, there is no need to include a **<taglib>** entry in the web application *web.xml* file.

Simple Tag Lifecycle

When a tag is invoked, a simple process occurs

- The most important piece of this process is the running of the doTag() method

Note: The Student guide contains the full process.

When a tag is invoked, the following occurs:

1. The web application container creates a new tag handler instance by calling the provided zero argument constructor
2. setJspContext() and setParent() methods are called by the container
   - setParent() method is only called if the element is nested within another tag invocation
3. The setters for each attribute defined for this tag are called by the container
4. If a body exists, the setJspBody() method is called by the container to set the body of this tag, as a "JspFragment". If the action element is empty in the page, this method is not called at all.
5. The doTag() method is called by the container
   - All tag logic, iteration, body evaluations, etc., **occur in this** doTag() **method.**
6. The doTag() method returns and all variables are synchronized.

SimpleTag Interface

The SimpleTag interface (http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/jsp/tagext/SimpleTag.html) provides the contract between the simple tag and the JSP page

```
public void doTag() throws JspException;
public JspTag getParent();
public void setJspBody(JspFragment body);
public void setJspContext(JspContext context);
public void setParent(JspTag parent);
```

SimpleTagSupport

To simplify the task of creating a simple tag, Java provides a SimpleTagSupport class.

- The javax.servlet.jsp.tagext.SimpleTagSupport class provides a default implementation of all methods in the SimpleTag interface.
- Extend this class and override it's doTag() method to build your own custom tag.

```
public class ACustomTag extends SimpleTagSupport {
  ...

  public void doTag() throws JspException {
  ...
 }

}
```

Tag Lib Directive

The Tag Lib Directive provides a reference to call a tag from a JSP page.

- The TLD file and attributes are created the same way as with Classic tags.

```
<tag>
   <name>newTag</name>
    <tag-class>com.ptc.NewTagClass</tag-class>
    <body-content>JSP</body-content>
    <attribute>
   <name>attrib</name>
   <required>true</required>
   <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

Tag Files

In place of TLD's, Windchill at times uses tags.

- Tags are similar to TLD's but they are referenced differently on the JSP

```
<%@ taglib prefix="wctags" tagdir="/WEB-INF/tags" %>
```

Tag Files are an additional option for implementing the custom tag function.

- If a component is included in JSP pages regularly, place the with *.tag* extension.
- Must be located in tags directory under *WT_HOME/codebase/WEB-INF*.
- Add a taglib directive (using tagdir).
- Use in JSP pages like a regular tag.

In essence, it's a special type of JSP fragment.

Tag File Example

The following is the code in *WT_HOME/codebase/WEB-INF/externalFormData.tag*.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test="${param.externalFormData != null}">
   <input type="hidden" name="externalFormData" value="${param.externalFormData}"/>
</c:if>
<c:if test="${externalFormData != null}">
   <input type="hidden" name="externalFormData" value="${externalFormData}"/>
</c:if>
```

This code is called from JSP pages by the following

```
<%@ taglib prefix="wctags" tagdir="/WEB-INF/tags" %>
...
<wctags:externalFormData/>
```

Summary

- If standard JSP actions, EL, or JSTL doesn't have a function that is needed, consider a custom tag.

- When writing a custom tag, there is probably no need to use the “classic” tag as the basis for the tag; use a simple tag (SimpleTagSupport).
- Using a tag file is an additional option for the custom function encapsulated in theSimpleTagSupport extension.

Java Expression Language

The JSP 2.0 spec includes a new tool, Java Expression Language (EL or JEL).

- EL makes it easy to print nested properties ${person.dog.name}
- EL expressions always begin with a $ and are between {}
- EL provides a number of implicit objects (different from those available in JSP scripting)
  - pageScope, requestScope, sessionScope, applicationScope, param, paramValues, header, headerValues, cookie and initParam
  - All of the above are “Map” objects (so scopes aren’t actual scope objects)
  - pageContext is not a Map – it references the actual pageContext object

Properties can be accessed via the “.” or “[ ]” operator

“[ ]” is more powerful – allows the “thing on the left” to be a Map, bean, List, or array

EL Example

In the Servlet layer:

```
java.util.Map musicMap = new java.util.HashMap();
musicMap.put("Ambient", "Zero 7");
musicMap.put("Surf", "Tahiti");
musicMap.put("DJ", "BT");
request.setAttribute("musicMap", musicMap);
String[] musicTypes = {"Ambient", "Surf", "DJ"};
request.setAttribute("musicType", musicTypes);
```

In the JSP:

```
Music is ${musicMap["Ambient"]}
Music is ${musicMap[musicType[1]]}
```

Results in:

```
Music is Zero 7
Music is Tahiti
```

Exercise 3-2: Retrieve Resource Bundle Data and Parse with JSTL and EL

Objectives

- Retrieve messages from a resource bundle.
- Practice JSTL and EL.

Scenario

A customization requires retrieving all Document error messages into a HashMap for retrieval later in the page, using JSTL and EL.

Step 1. Create a JSP file.

a. Add a standard JSP page directive; the JSTL core taglib directive, and HTML and BODY tags.

Example:

```
<%@page language="java"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html><body>

</body></html>
```

Step 2. Retrieve all Document error messages using scriptlets.

a. This will required adding appropriate import statements.

b. Loop through the resource bundle keys and print the messages to the a server log. Which log file?

Example:

```
<%@page language="java"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@page import="java.util.ResourceBundle,
        java.util.Enumeration, java.util.Locale, wt.util.WTContext"%>

<%
  Locale locale = WTContext.getContext().getLocale();
  ResourceBundle docRB = ResourceBundle.getBundle("wt.doc.docResource", locale);
  Enumeration enum1 = docRB.getKeys();
  String key;
  while ( enum1.hasMoreElements() ) {
    key = (enum1.nextElement()).toString();
    System.out.println(key + " " + docRB.getString(key));
  }
%>

<html><body>

</body></html>
```

c. Save the file and test.

d. Inside the scriptlet, modify the loop to read the messages into a map of objects.

Example:

```
<%@page language="java"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@page import="java.util.HashMap, java.util.ResourceBundle,
        java.util.Enumeration, java.util.Locale, wt.util.WTContext"%>
<%
  Locale locale = WTContext.getContext().getLocale();
  ResourceBundle docRB = ResourceBundle.getBundle("wt.doc.docResource", locale);
  Enumeration enum1 = docRB.getKeys();
  String key;
  HashMap map = new HashMap();
  while ( enum1.hasMoreElements() ) {
    key = (enum1.nextElement()).toString();
    map.put(key, docRB.getString(key));
    System.out.println(key + " " + docRB.getString(key));
  }
%>

<html><body>

</body></html>
```

Step 3. Add JSTL and EL to loop through the map and display to the screen.

a. In this example, the output has been wrapped in a table.

b. A <c:forEach> JSTL tag loops.

c. A <c:out> JSTL tag prints output.

d. The input being processed is the map and is designated in EL format — ${map}, {item.key} and {item.value}.

e. The *item* variable is output from the JSTL loop which is processing *map*.

Example:

```
<%@page language="java"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@page import="java.util.HashMap, java.util.ResourceBundle,
        java.util.Enumeration, java.util.Locale, wt.util.WTContext"%>

<%
  Locale locale = WTContext.getContext().getLocale();
  ResourceBundle docRB = ResourceBundle.getBundle("wt.doc.docResource", locale);
  Enumeration enum1 = docRB.getKeys();
  String key;
  HashMap map = new HashMap();
  while ( enum1.hasMoreElements() ) {
    key = (enum1.nextElement()).toString();
    map.put(key, docRB.getString(key));
    System.out.println(key + " " + docRB.getString(key));
  }
%>

<html><body>

<table border=1><c:forEach var='item' items='${map}'>
   <tr><td><c:out value="${item.key}"/></td><td><c:out value="${item.value}"/></td></tr>
</c:forEach>

</body></html>
```

f. If the file is tested, there is no output to the screen. Note that EL parameters need a scope.

Example:

```
<%@page language="java"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@page import="java.util.HashMap, java.util.ResourceBundle,
        java.util.Enumeration, java.util.Locale, wt.util.WTContext"%>

<%
  Locale locale = WTContext.getContext().getLocale();
  ResourceBundle docRB = ResourceBundle.getBundle("wt.doc.docResource", locale);
  Enumeration enum1 = docRB.getKeys();
  String key;
  HashMap map = new HashMap();
  while ( enum1.hasMoreElements() ) {
    key = (enum1.nextElement()).toString();
    map.put(key, docRB.getString(key));
    System.out.println(key + " " + docRB.getString(key));
  }
  pageContext.setAttribute("map", map);
%>

<html><body>

<table border=1><c:forEach var='item' items='${map}'>
   <tr><td><c:out value="${item.key}"/></td><td><c:out value="${item.value}"/></td></tr>
</c:forEach>

</body></html>
```

g. Save the file.

Step 4. Test.

Step 5. Challenge exercise: add a form which prompts user for a key and displays the message correlating to the key.

JavaScript Overview

JavaScript is a prerequisite for this class. This is a brief overview of JavaScript.

- JavaScript is a scripting language typically for client-side development embedded in HTML pages.
- JavaScript is mainly used to write functions inside or included in HTML pages.
- JavaScript is essentially unrelated to the Java programming language, although the tools have some common history.
- There is a standardized version named “ECMAScript”.

The following HTML code defines a JavaScript method (SomeJavaScriptMethod). Later in the page, an HTML form calls the method based on an event (onkeyup).

```
<html>
<script type="text/javascript">
function SomeJavaScriptMethod( key ) {
  document.aForm.typedKey.value=key;
}

</script>

<body>
<form name=aForm method=get onkeyup="SomeJavaScriptMethod(document.aForm.key.value);">
  Key:<input type="text" name="key"/>
  Typed Key:<input type="text" name="typedKey"/>
</form>

</body></html>
```

JavaScript Events

JavaScript can key off of the following events:

Event Name	The event occurs when:
onabort	Loading of an image is interrupted
onblur	An element loses focus
onchange	The user changes the content of a field
onclick	Mouse clicks an object
ondblclick	Mouse double-clicks an object
onerror	An error occurs when loading a document or an image
onfocus	An element gets focus
onkeydown	A keyboard key is pressed
onkeypress	A keyboard key is pressed or held down
onkeyup	A keyboard key is released
onload	A page or an image is finished loading
onmousedown	A mouse button is pressed
onmousemove	The mouse is moved
onmouseout	The mouse is moved off an element
onmouseover	The mouse is moved over an element
onmouseup	A mouse button is released
onreset	The reset button is clicked
onresize	A window or frame is resized
onselect	Text is selected
onsubmit	The submit button is clicked
onunload	The user exits the page

Asynchronous JavaScript and XML (Ajax)

- Ajax is **not** a new programming language
- Ajax is a technique for creating better, faster, and more interactive web applications

With Ajax, JavaScript can communicate directly with the server, using the JavaScript XMLHttpRequest object to:

- trade data with a web server, without reloading the page
- provide asynchronous data transfer (using HTTP requests) between the browser and the web server

Benefits:

- Web pages will request small bits of information from the server instead of whole pages.
- The Ajax technique makes Internet applications smaller, faster and more user-friendly.
- Ajax is browser-independent (although it may be implemented differently for different browsers)
- Ajax can dynamically refresh a portion of a page when some event occurs.

Ajax Example

The following is an Ajax page which runs Info*Engine tasks as soon as they are typed in the entry panel.

```
<html>
<body>

<script type="text/javascript">
function ajaxFunction() {
  var xmlHttp;
  try {
    // Firefox, Opera 8.0+, Safari
    xmlHttp=new XMLHttpRequest();
  } catch (e) {
    // Internet Explorer
    try {
      xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
    } catch (e) {
      try {
        xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
      } catch (e) {
        alert("Your browser does not support Ajax");
        return false;
      }
    }
  }
  xmlHttp.onreadystatechange=function() {
    if(xmlHttp.readyState==4) {
      document.myForm.comments.value=xmlHttp.responseText;
    }
  }

  xmlHttp.open("GET", "http://localhost/Windchill/servlet/IE/tasks/" +
                                  document.myForm.taskname.value, true);

  xmlHttp.send(null);
}

</script><form name="myForm">
<form name="myForm">
Info*Engine Task name (infoengine/examples/CreateGroup.xml):
<input type="text" onkeyup="ajaxFunction();" name="taskname" />
<p>
<textarea name="comments" value="time" cols="60" rows="20" >
</textarea>
</form>
```

```
</body>
</html>
```

The example works as follows:

1. A form is displayed with an entry field for the task name.
2. When anything is entered in the text area — because of the JavaScript onkeyup — the ajaxFunction is called to run the function as it is being typed.
3. To run the function, XMLHTTPRequest() is instantiated.
4. onreadystatechange tells the function to run asynchronously with the browser window.
5. When the data is received, a call back is initiated by the system.
6. When the call back is received, the component within the page is updated.

Note: The method that runs is still executing against the web server. There may still be a performance issue if the server is slow to respond or if the response contains a large amount of data

Implementing Ajax in Windchill

- To make the Windchill product faster and more scalable, some Ajax-based operations were included to cut down on the refresh times of pages.
- All actions should be Ajax-based where possible.
- The only actions that should not be Ajax-based are those that update the entire page or the majority of the page.
- Ajax actions only update, for example, one row of the table, or refreshes all rows in a table leaving the rest of the page and table unchanged.
- Steps:
  1. Define the action in an XML-based file
  2. Create a delegate (essentially a Java program) that performs the database transaction.
  3. A JSP file based on the Windchill client architecture to partially refresh the page (row, table or portion of page).

Before implementing Ajax in Windchill:

1. Practice with Ajax.
2. Define actions and create JSP files based on the Windchill client architecture to display basic pages.
3. Implement Ajax in Windchill.

Exercise 3-3: Dynamic UI to Search for Parts

Objectives

- Create an interactive web page.
- Practice Ajax, JavaScript and EL.

Scenario

A user wants to see Windchill Part information based on entering a number, like a search. However, the Part numbers are long and hard to remember, so that while entering the number, the user wants a prompt of valid numbers that match the criteria entered so far. Once the user has narrowed down to one Part number, there is a **Retrieve** button to return all the properties for the Part number entered.

Step 1. Create the form with the UI Elements.

a. Create a JSP page *WT_HOME/codebase/wtcore/jsp/com/gsdev/client/GetPartNumberForm.jsp* with an HTML form like the following:

Example:

```
<form name="aForm" method=POST
      action="/Windchill/servlet/IE/tasks/com/gsdev/client/QueryParts.xml">
 Suggested Parts:<textarea name="parts" cols="20" rows="50"></textarea>
 Enter Part Number:<input type="text" name="number" onkeyup="ajaxFunction();"></input>
 <input type="submit" value="Retrieve"></input>
</form>
```

b. Add the appropriate page directives.

c. Add HTML and BODY elements.

d. When selecting **Retrieve**, the form will call a second JSP page, *WT_HOME/codebase/wtcore/jsp/com/gsdev/client/GetPartNumbers.jsp*, plus the argument entered in the form.

Step 2. Pass the name of the file using Expression Language

Step 3. Edit *WT_HOME/codebase/wtcore/jsp/com/gsdev/client/GetPartNumberForm.jsp* to add Ajax components that update the **Suggested Parts** textarea.

a.

Example:

```
<script type="text/javascript">
 function ajaxFunction() {
  var xmlHttp;
  try {
   // Firefox, Opera 8.0+, Safari
   xmlHttp=new XMLHttpRequest();
  } catch (e) {
   // Internet Explorer
   try {
     xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
   } catch (e) {
    try {
     xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
    } catch (e) {
     alert("Your browser does not support Ajax!");
     return false;
    }
   }
  }
  xmlHttp.onreadystatechange=function() {
   if(xmlHttp.readyState==4) {
     document.aForm.parts.value=xmlHttp.responseText;
     document.aForm.partsel.value=xmlHttp.responseText;
   }
  }
  xmlHttp.open("GET","GetPartNumbers.jsp?number=" + "*" +
                  document.aForm.number.value + "*",true);
  xmlHttp.send(null);
 }
</script>
```

b. Place the Ajax code in the HEAD area of the HTML element.

Step 4. Create an Info*Engine task to retrieve part numbers.

a. Create *WT_HOME/tasks/com/gsdev/client/QueryParts.xml* with the following content:

Example:

```
<%@page language="java"%>
<%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<ie:webject name="Get-Properties" type="MGT">
    <ie:param name="ATTRIBUTE" data="wt.federation.ie.VMName"/>
    <ie:param name="GROUP_OUT" data="properties"/>
</ie:webject>

<ie:webject name="Query-Objects" type="OBJ">
 <ie:param name="INSTANCE"  data="${properties[0]wt.federation.ie.VMName[0]}"
                              default="local.ptc.windchill.Windchill"/>
 <ie:param name="WHERE"     data="number=${@form[0]number[0]}" default="*"/>
 <ie:param name="TYPE"      data="wt.part.WTPart"/>
 <ie:param name="GROUP_OUT" data="parts"/>
</ie:webject>
```

b. The task will take the request object *number* parameter as input, and return an Info*Engine "VDB" group named *parts*.

c. Update the default value of *INSTANCE*.

Step 5. Create a JSP page to call the Info*Engine task.

a. Create *WT_HOME/tasks/com/gsdev/client/QueryParts.xml* with the following content:

Example:

```
<%@page language="java" isELIgnored="true"%>
<%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>
<%@page import="com.infoengine.object.factory.Group, java.util.HashMap"%>
<ie:task uri="com/gsdev/client/QueryParts.xml">
<ie:param name="number" data="${@form[0]number[0]}"/>
</ie:task><ie:getService varName="ie"/>
<%
Group parts = ie.getGroup("parts");
for (int i = 0; i < parts.getElementCount(); i++) {
 try {
  out.println( (String)parts.getAttributeValue(i, "number") );
 } catch (java.lang.Exception e) {
  //e.printStackTrace();
 }
}
%>
```

b. Due to an issue with "JSP white space", this code will produce a number of carriage returns in the output. Unusual formatting can remove that white space.

Example:

```
<%@page language="java" isELIgnored="true"
%><%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"
%><%@page import="com.infoengine.object.factory.Group, java.util.HashMap"
%><ie:task uri="com/gsdev/client/QueryParts.xml">
<ie:param name="number" data="${@form[0]number[0]}"/>
</ie:task><ie:getService varName="ie"/><%
Group parts = ie.getGroup("parts");
for (int i = 0; i < parts.getElementCount(); i++) {
 try {
  out.println( (String)parts.getAttributeValue(i, "number") );
 } catch (java.lang.Exception e) {
  //e.printStackTrace();
 }
}
%>
```

Summary

After completing this module, you should be able to:

- Implement basic pages with JSP.
- Implement basic pages with JavaScript.
- Implement basic pages with the Java Expression Language.
- Implement basic pages with Ajax.
- Implement basic pages with JSTL.

Module

4

Windchill Client Architecture

This module introduces the Windchill client architecture (also known as “Java Client Architecture”) and defines the Windchill user interface components.

Objectives

Upon successful completion of this module, you will be able to:

- Identify and describe the elements of the Windchill client architecture.
- Provide an overview of the OOTB tools and examples available.

Lecture Notes

You may use the space below to take your own notes.

Windchill Client Technologies

- JSP framework pages
- Info*Engine tasks with JSP
- Dynamic Client Architecture (DCA)
- Template processors
- Java applets
- Standalone Java applications
- Non-Java clients
- JSP Client Architecture (JCA)

As of 9.0:

- Most of the Windchill 9 UI is JCA.
- Template processing still exists on the **Document Structure** and **Product Structure** pages.
- DCA exists in ESI components, **Queue Manager** and **Principal Administrator** pages.
- The main search is JCA.

UI Elements

The term “UI Element” is simply meant to represent something that appears in the UI.

Note: In many cases, the same UI element can have multiple names.

- Global Navigation and Tabs
- First Level Tab
- Second Level Tab
- Context Bar
- Info Page
- Top Attributes Panel
- Third-level Navigation Bar
- Third-level Content Area
- Table
- Tree
- Toolbar
- Filtered View
- Filtered Views List Manager
- Filtered View Create Client
- Object Icon
- Status Glyph
- Wizard
- Picker
- Action
- Action Model
- Action Icon
- Actions Menu
- See Actions Menu
- Action Service
- Attribute
- UI Component Validation
- UI Component Validation Service
- UI Component Validator Icon Delegate
- NmCommandBean
- NmSessionBean
- begin.jspf
- Object Reference
- Version Reference
- NmOid
- NmSimpleOid
- Descriptor
- Data Utility
- GUI Component
- getModel & getIEModel (tags)
- Model Command

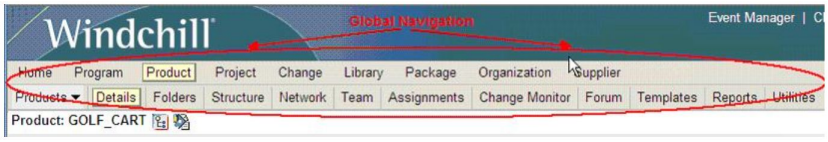


Figure 4-1: Global Navigation

Global Navigation

- Set of links at the top of a Windchill page
- Allow an end user to *navigate* throughout a Windchill installation
- May vary from one user to another
- May vary from one installation to another
- Specifically, composed of the First Level Tabs and the Second Level Tabs

Tab

- Individual link in the Global Navigation
- Resembled a tab on a file folder in previous releases

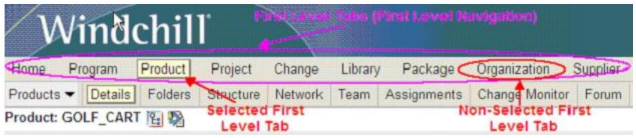


Figure 4-2: First Level Tab

- Individual link in the first row of links in the Global Navigation
- Available tabs vary from user to user based on administration privileges, etc.
- Available tabs vary from one Windchill installation to another based on the set of installed Windchill solutions (e.g., PDMLink, ProjectLink, SuMa, etc.)
- Examples include **Home**, **Product**, **Project**, **Organization**, etc.
- Collective set of First Level Tabs is sometimes referred to as *First Level Navigation.*

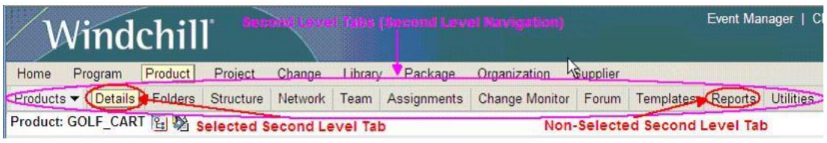


Figure 4-3: Second Level Tab

- Also called a “Sub Tab”
- Individual link in the second row of links in the Global Navigation
- One set of Second Level tabs for each First Level Tab
- Only one set of Second Level Tabs will be displayed at a given time, depending on the First Level Tab selected
- Examples include **Details**, **Folders**, **Team**, and **Utilities**, etc.
- Collective set of Second Level Tabs is sometimes referred to as *Second Level Navigation*

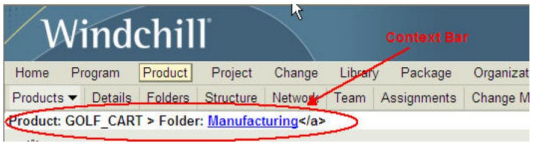


Figure 4-4: Context Bar

- Also called a “White Bar”
- Tells the user “where they are” in the UI (container, folder, etc.)
- Located directly below second-level navigation

- May also contain action icons

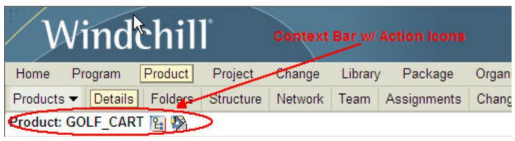


Figure 4-5: Another View of the Context Bar

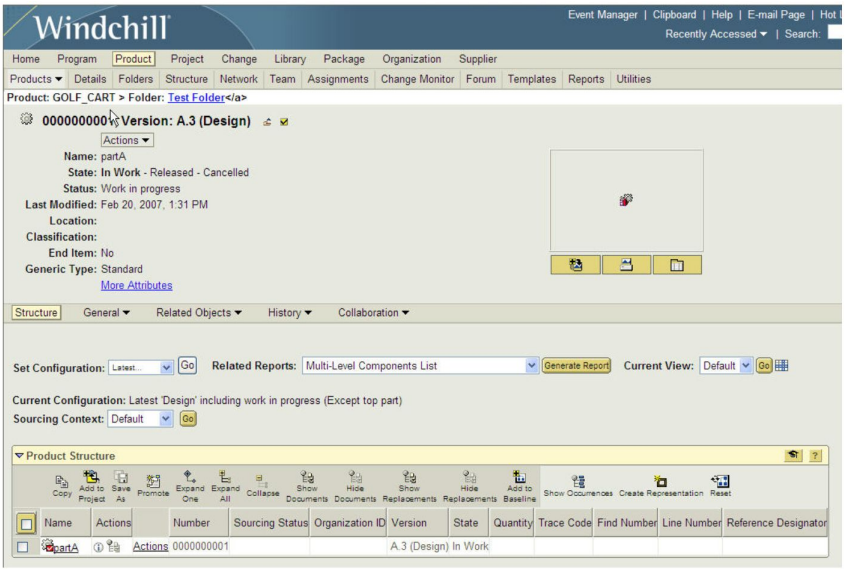


Figure 4-6: Info Page

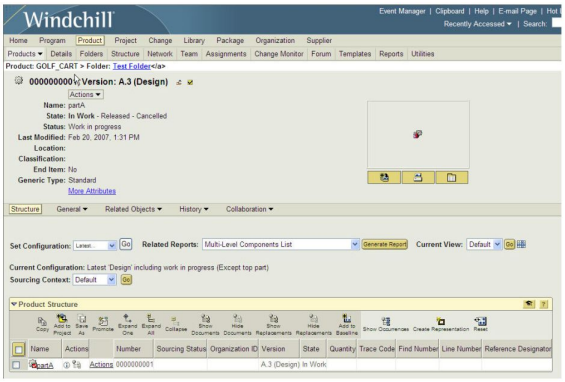


Figure 4-7: Another View of the Info Page

- Contains data specific to a single Windchill business object
- Common Component exists to aid in application development
- Composed of **Top Attributes Panel** and **Third-level Content**

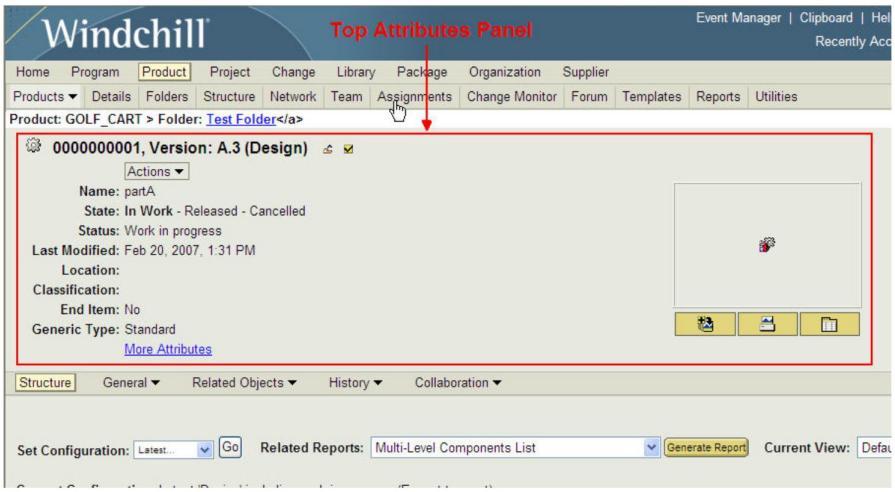


Figure 4-8: Top Attributes Panel

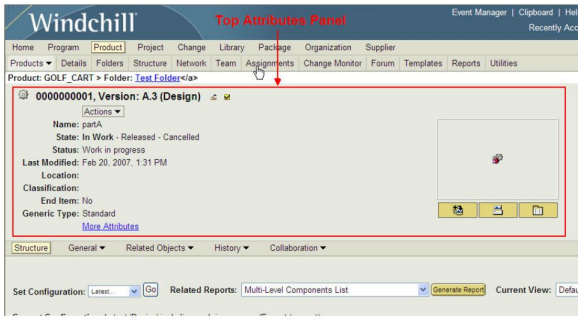


Figure 4-9: Another View of the Top Attributes Panel

- Top portion of an info page
- Contains:
  - context object identification info (name, number, icon, etc.)
  - values for several of the context object's main attributes
  - list of actions that can be performed on the context object
  - link to view additional attributes of the context object

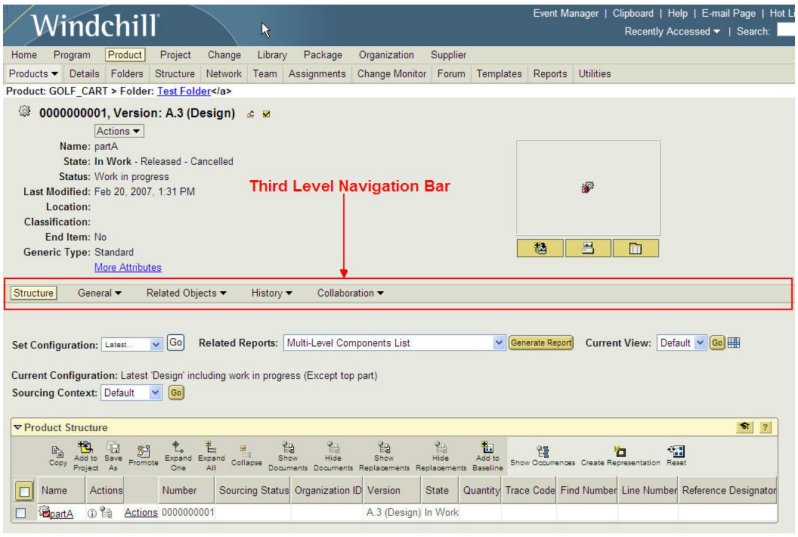


Figure 4-10: Third-level Navigation Bar

Figure 4-11: Another View of the Third-level Navigation Bar

- Located in center section of an Info Page
  - Separates Top Attributes Panel from Third-level Content Area
- Composed of multiple action menus (definition forthcoming)
- Selected menu items determine what is displayed in Third-level Content Area
- Third-level of navigation in the Windchill UI (First Level Tabs, Second Level Tabs)

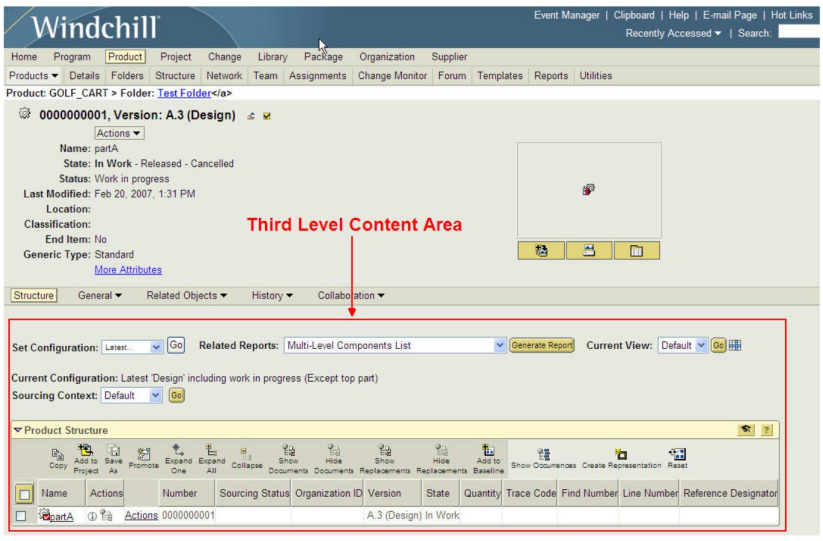


Figure 4-12: Third-level Content Area

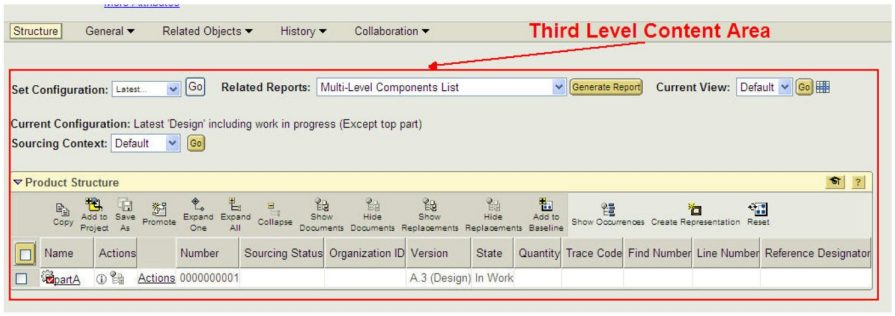


Figure 4-13: Another View of the Third-level Content Area

- located at the bottom of a Windchill Info Page
- contents determined by selected Third-level Navigation menu item

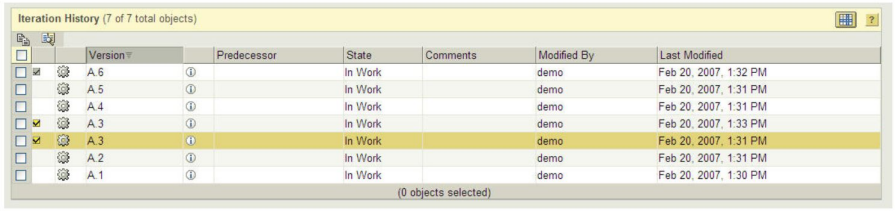


Figure 4-14: Table

- useful for displaying many results of some related data
- consists of columns and rows
- Excel-like

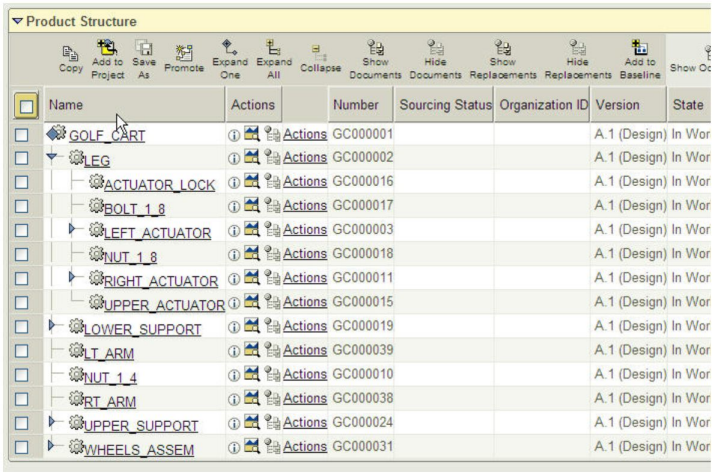


Figure 4-15: Tree View

- Similar to table, plus hierarchy
- Rows can be expanded or collapsed if they have children
- Windows Explorer-like

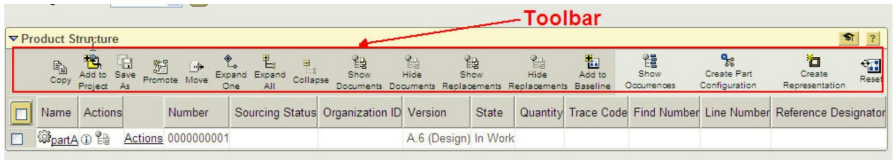


Figure 4-16: Toolbar

- Located at the top of a table or tree
- Contains action icons (definition forthcoming)
  - Actions in left/dark-shaded area can be applied to one or more selected row objects
  - Actions in right/light-shaded area are general in nature and do not apply to any specific row object(s)

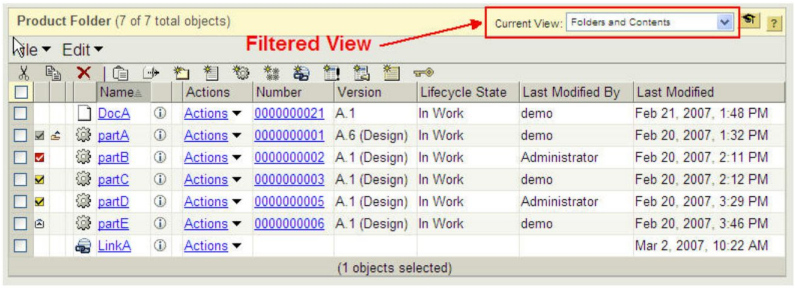


Figure 4-17: A Filtered View

- Defines a representation of a table's or tree's data:
  - Columns to display
  - Column order
  - Column sort order
  - Filter criteria
- Tables and trees can be configured to include/exclude filtered views
- Views included can be OOTB or custom user-defined views

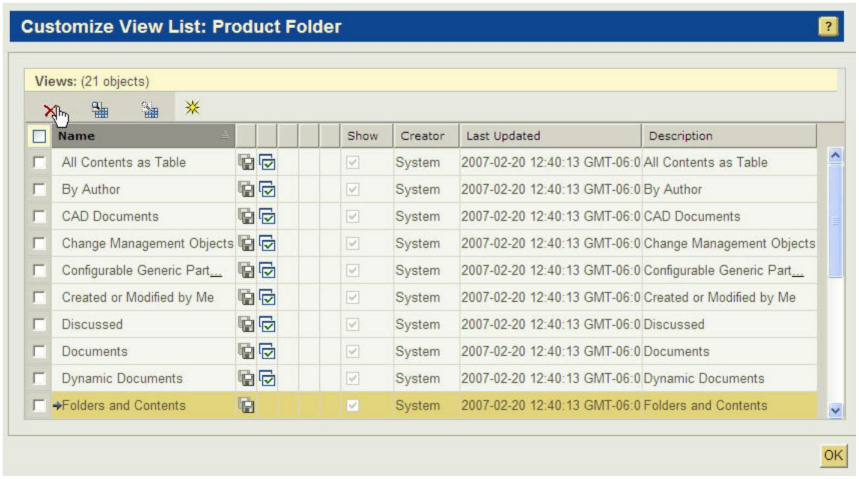


Figure 4-18: Filtered Views List Manager View

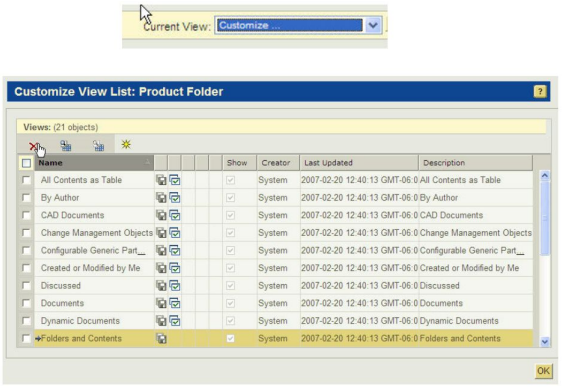


Figure 4-19: Another View of the Filtered Views List Manager

- Allows users to create, edit and delete filtered views
- Also allows user to change active/default views
- Launched by selecting **Customize...** in the table view drop down

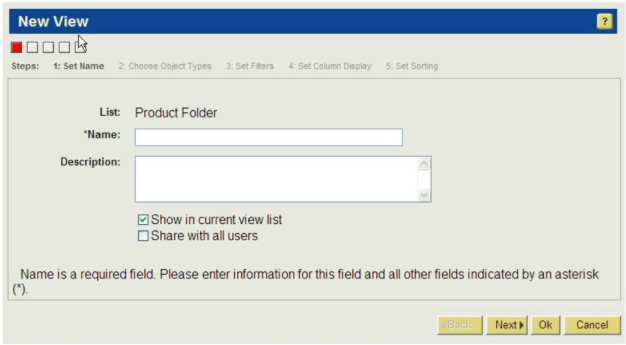


Figure 4-20: Filtered View Create Client Wizard

- Wizard used to create filtered views
- Launched from Filtered Views List Manager

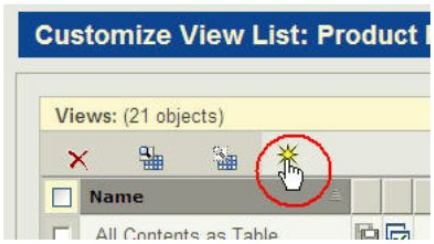


Figure 4-21: Filtered Views List Manager

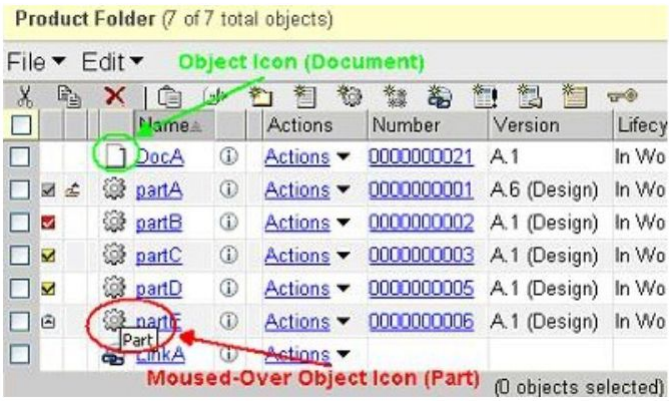


Figure 4-22: Object Icon

- Identifies type of Windchill object being displayed
- Tooltip provides the name of the represented object
- Appear in tables, trees, Top Attributes Panels, etc.

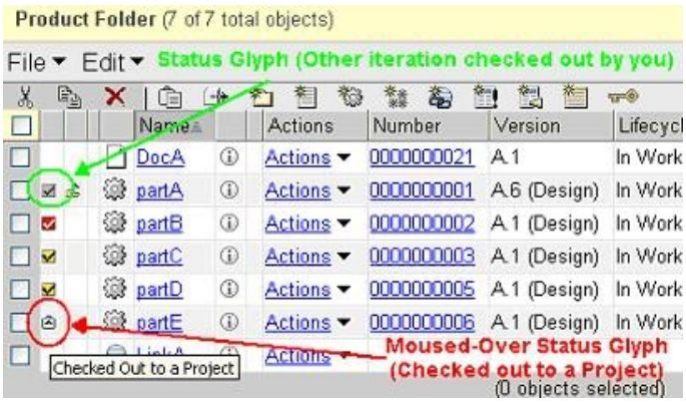


Figure 4-23: Status Glyph

- Provides information about the state of a Windchill object
- Tooltip indicates actual state indicated by the glyph
- Appear in tables, trees, Top Attributes Panels, etc.

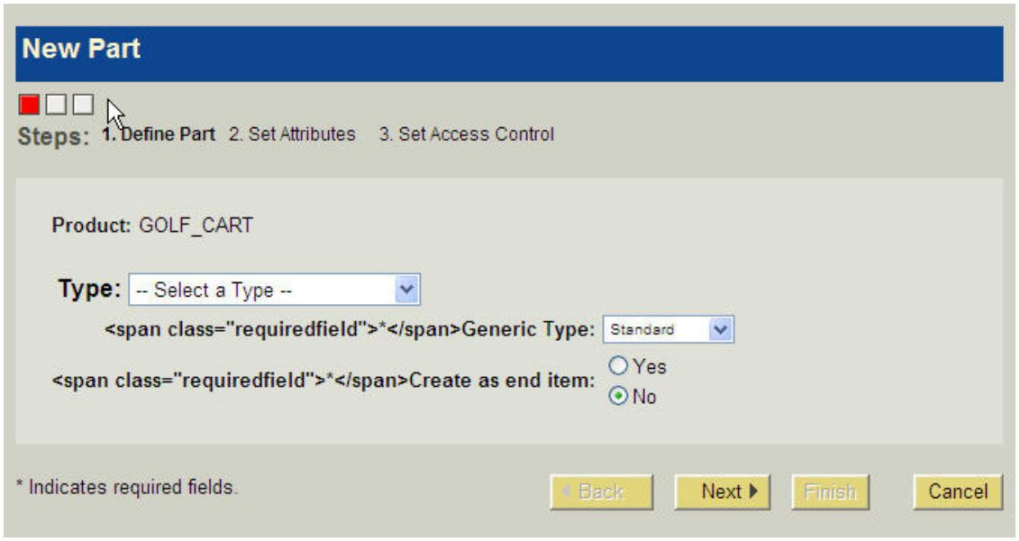


Figure 4-24: Wizard

- Separate window with one or more steps
- Navigate between steps using Next and Back buttons
- Once all (required) steps are completed by user, Finish button closes the wizard and performs some action
- Wizard Common Components are available for application development
- Similar Windows application installers

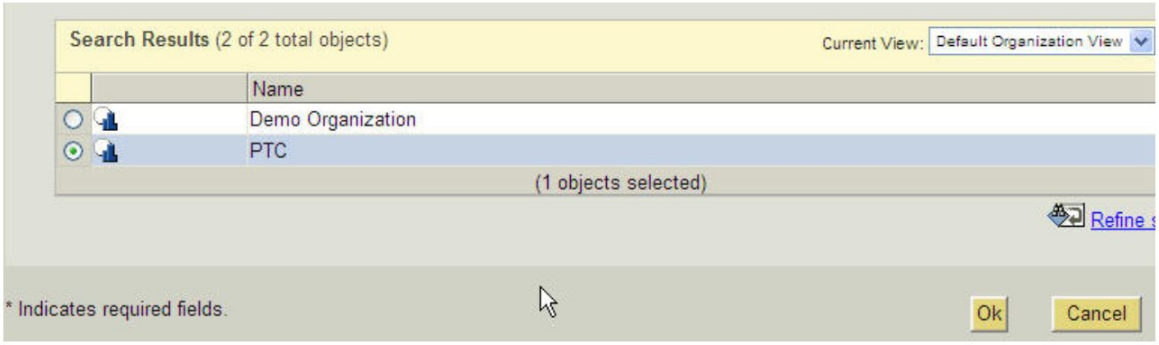


Figure 4-25: Picker

- Allows a user to search for and select items (Windchill business objects)
- Typically found in a separate or pop up window
- Picker Common Components available for application development

Action Icon

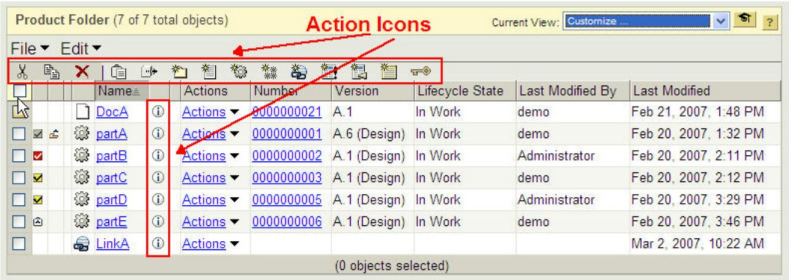


Figure 4-26: Action Icons

- Action rendered as an icon in the UI
- Can appear in tool bars, table/tree rows, etc.

Action Menu

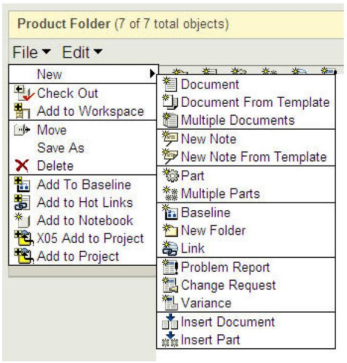


Figure 4-27: Action Menu

- Collection of actions displayed as text links in the UI
- Organized in hierarchical tree structure
- Each branch in the tree is an action model
- Therefore, the action menu is just a set of nested action models

Figure 4-28: See Actions Menu or Actions Drop Down

- Also called “Actions drop down”
- Used to display actions available for a row object in a table/tree
- Rendered using Ajax for performance reasons
  - Don’t have to calculate available actions for every row in a table/tree when the table/tree is rendered.
- Called “See Actions” because in previous releases there was a link with the text “See Actions” that would launch a popup window with available actions

JSP Client Architecture Terminology

The following is a list of terms related to the JSP Client Architecture (JCA) UI technology.

JSP Client Architecture:

- A collection of building blocks that can be used to create a (HTML) Windchill client:
  - JSP pages/fragments
  - Java Beans
  - Custom tags/taglibs
  - Java services
  - JavaScript functions

JCA is an Entire Framework

The entire architecture (classes, methods, property files, customization points) needed to be able to do the following:

- Support retrieving and rendering data in a common fashion.
- Support customization easily.
- Keep customization separated from OOTB files.
- Support resource bundles and localization.
- Validate data and forms.

JCA is an adaptation or extension of the “Netmarkets” infrastructure:

- Introduced for Windchill ProjectLink clients in 6.0 and has been evolving ever since.
- Hence many of the JCA bjects and classes include an “Nm” prefix.

Action

- Combination of a link displayed on a Windchill page and logic that is executed when link is invoked
- Can be rendered in the UI as text, icon, navigation tab, menu item, etc.
- JCA object representing an action is NmAction
- JCA actions are defined in files ending with *actions.xml*
  - (e.g., *ChangeManagement-actions.xml*)

Action Model

- Collection of logically-related actions
- Contents of an action model may be actions or action models
  - (i.e., action models may be nested)
- NmHTMLActionModel objects represent action models in JCA
- JCA action models are defined in files ending in *actionmodels.xml*
  - (e.g., *ChangeManagement-actionModels.xml*)

Action Service

- Service used to retrieve actions or action models in JCA
- Given an action ID (from *actions.xml*), returns an action (NmAction)
- Given an action model ID (from *actionmodels.xml*), returns an action model (NmHTMLActionModel)
- Runs on Windchill Method Server
- StandardNmActionService is actual class

Attribute

- Characteristic of a Windchill business object

- Examples include *name* and *number*

UI Component Validation

- Also called “UI validation”, “action validation”, or just “validation”
- Determines whether or not something should appear or be allowed in the UI
- Three main types of validation:
  1. “Pre-validation”:
     - determines whether or not something should appear in the UI
     - can be performed on actions or attributes
  2. “Post-Select Validation”:
     - determines whether or not an action should be allowed to be performed after it’s selected
  3. “Post-Submit Validation”:
     - determines whether or not user-entered data in a form is legitimate
- Validation can be performed on the client using JavaScript, or on the Method Server using the validation service and validators.

UI Component Validation Service

- Also called “UI validation service”, “action validation service”, or just “validation service”
- Given a UI element (action or attribute), will return a result indicating whether or not the usage of that UI element is valid in a given context
- Delegates to Validators to perform the actual logic of determining validity
- Can be called to perform Pre-Validation, Post-Select Validation, and Post-Submit Validation
- Runs on Windchill Method Server
- Actual Class is StandardUIComponentValidationService

UI Component Validator

- Also called “action validator” or just “validator”
- Delegate of the UI Component Validation Service
- Contains business logic to perform validation on one or more UI Components (actions or attributes)
- A single validator can contain logic for pre-validation, post-select validation, and post-submit validation
- Should be a subclass of DefaultUIComponentValidator

Icon Delegate

- Mechanism for type specific retrieval of object icons.
- If no type-specific icon delegate exists, the default delegate will retrieve the icon from the Rational Rose model.

NmCommandBean

- aka, “Command Bean”
- Contains HTTP request and all the JCA state information
- All JCA delegates and components can use this data to execute their logic
- Primary goal is to give some level of MVC to different components
- Javadoc has improved to have more description for some of the methods

Object Reference

- Lightweight handle to a Persistable (super class for persisted Windchill business objects)
- Represents a row in the database
- The two key attributes are the Table name and the Row id

Version Reference

- Lightweight handle to a Versioned Persistable.
- Represents a row a branch of an object.
- Each branch can have may objects associated with it much like a ClearCase branch.
- The two key attributes are the *table name* and the *branch id*.
- There can be only one object with the latest flag on each branch
  - So getting a Persistable from the Version Reference often will inflate the latest iteration of the object.

begin.jspf

- JSP fragment that adds all the JCA state data to the HTML page and to the request
- Initializes all JCA beans (e.g., NmCommandBean, NmSessionBean)
- Includes JCA JavaScript files
- Responsible for rendering the Windchill header and Global Navigation
- Contains logic for executing actions

NmSessionBean

- aka, “Session Bean”
- Session state wrapper for JCA
- Contains helper APIs to get and set preferences
- Used to carry information between requests or to do some caching to reduce RMI trips

NmOid

- JCA wrapper for the Windchill WTReference (which may be an Object Reference, Version Reference, etc.)
- Lightweight handle to a Persistable in Windchill.
- Tries to hide some of the details of the different kinds of WTReferences like VersionReference and cachedObjectReference to improve performance and make developing easier.
- Example of a string: “oid=OR%3Awt.org.WTUser%3A8897”

NmSimpleOid

- Special NmOid class that lets developers control the String used to represent some object in the page.
- For non-persisted objects, NmSimpleOids are the only way to represent the in-work object.
- Often its just a time stamp with a string type attached.
  - Example of a string: “oid=WTPart_3433048930”
- Developer is responsible for the string format so must be able to parse the string and reconstruct the object.
- NmSimpleOids are often used so that more information can be carried around with the row.

Descriptor

- aka, “Component Descriptor”
- Object that contains properties about a JCA component.
- A bit like a properties object, but can have children descriptors.
- These properties are then used by the other tags to control their behavior.

DataUtility

- Creates table cell data from row objects.
- It is mapped to a column id through *service.properties* file.
- Responsible for getting the display value of any object.

- One can view all the data utilities that a table uses by adding “JCADebug=1” to the URL.
- DataUtility delegates should be able to fetch a display value for at least Persistables, Type Instances, and Info*Engine elements.

GUI Component

- UI Bean objects that represent the attributes of what to display in each table cell.
- These are the things that Data Utilities should be returning.
- For example, to show an icon about the row object, the IconComponent (extends from GuiComponent) should be used.
- It has an HTML renderer mapped to it that knows how to convert the IconComponent into an <img> HTML tag.
- One day there might be multiple renderers mapped to GUI Components if we ever support some other type of display.

getModel & getIEModel Tags

- The tag that gets the business data for a table.
- This tag is the one that makes the RMI trip to the Method Server.
- The properties are which method to call (getModel) or which Info*Engine Task (getIEModel).
- These tags serve as input in a tableDescriptor tag (which is used to represent a Component Descriptor for a table).

ModelCommand

- Is the object that the getModel tag eventually executes in order to get business data for the table.
- There are many subclasses of this that each gather up the row-data for a table in its own way.
- ModelCommand has hard-coded logic about which command Delegate subclass to use depending on the parameters specified in the JSP.
- For example, it knows if the JSP specifies a service class and method name, to construct a ServiceModelCommand.

Tools and References

JSP, JSTL:

- http://java.sun.com/products/jsp
- http://jsptut.com
- http://java.sun.com/products/jsp/jstl/
- http://www.onjava.com/pub/a/onjava/2002/08/14/jstl1.html
- http://www-128.ibm.com/developerworks/java/library/j-jstl0211.html
- http://www-128.ibm.com/developerworks/java/library/j-jstl0318
- http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html

Internal Links

- A table component design pattern R&D Wiki page: http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=JCA_Table_Component:
  - "The actionModel property indicates the name of the action model (from *actionmodels.xml*) that should be rendered in the table's toolbar. "
- DTD or action/action model information:https://pds.ptc.com/Windchill/netmarkets/jsp/folder/view.jsp?oid=folder~wt.folder.SubFolder:194247766
- More information on the renderTableTree tag can be found at http://windchillweb.ptcnet.ptc.com/javadoc/Windchill_x-10/docs/api/TagLibraryDoc/wnc/CommonComponents-web/components/renderTableTree.html.
- Additional information about JCA actions: http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=JCA_Actions.
- "Tables and Trees" design pattern on the R&D Wiki: http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=JCA_Table_Component
- http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=Main_Page_JCA_Infrastructure
- http://windchillweb/wiki/index.php?title=JCA_Resources_For_Developers
- DataUtilities Wiki page: http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=Data_Utilities
- The R&D Wiki has some more detail the execution flow: http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=Table_Execution_Flow
- The R&D Wiki also contains a tree example: http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=JCA_Tree_Component
- Details on validator registration: http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=See_if_your_validator_is_registered
- For more information on UIComponentClientGroup, see http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=Client-based_validation page.
- For more information on the Role Based Service, see this Wiki page.
- For more information on UIComponentSolutionGroup, see this Wiki page.
- Latest validation updates and examples. It can be found here.
- Common component Owners
  - http://rdwiki.ptcnet.ptc.com/wiki/index.php?title=X-10_Common_Component_Owners_and_Supporting_Documentation
- OOTB Windchill Tags
  - http://windchillweb.ptcnet.ptc.com/javadoc/Windchill_x-10/docs/api/TagLibraryDoc/
  - This documentation was generated using tools from https://taglibrarydoc.dev.java.net/. This tool can also be used on custom code

Property Report

- Similar to the **Attribute Report** is the **Property Report**.

- A property is different from an attribute in the way that it must be displayed. Namely, the two column display aligned at the division with a semicolon as the separator.
- Can be accessed from a running Windchill installation.
- The URL is: http://localhost/Windchill/netmarkets/jsp/property/propertyReport. jsp.

Debugging

Add “&jcaDebug=1” to the end of a URL request and the UI presents the name of the elements embedded with the content.

Although the presentation of the page is skewed, it provides detailed information on the elements.

JCA Tools and Examples

Some JCA tools and examples are found in *WT_HOME/codebase/netmarkets/jsp/carambola*

URL	What it Demonstrates
carambola/createtable-view/availableAttributesReport.jsp	Provides a full list of attributes for a type/class
carambola/database/remove_configurable_views.jsp	The configurable view information is kept in two tables. Testing changes to configurable view information often requires deleting these tables to force the recreation of the views and thus incorporate any changes. This page allows the deletion of those two tables.
carambola/database/strider.jsp	This utility allows navigating through a Windchill database to examine the contents.
carambola/ie/jcaSearch.jsp	Use the table view manager to add search criteria and to filter by object type.
carambola/lwc/table.jsp	Custom table example
carambola/propertyPanel/examples.jsp	Example property panel
carambola/suggest/suggest.jsp	Example of suggestion in a text field
carambola/svc/report.jsp	Utility to report Windchill property-based service configuration, e.g., for a given class, report the DataUtility
carambola/svc/tree.jsp	Example of a tree
carambola/svc/treeWithNodeColumn.jsp	Example of a tree with expandable nodes
carambola/tags/format.jsp	Formatting example
carambola/tags/formatNoLocale.jsp	Formatting example
carambola/tags/formatNoRB.jsp	Formatting example
carambola/tags/formatOptional.jsp	Formatting example
carambola/tools/actionReport/action.jsp	Lists UI action information

Available Attributes Report

The **Available Attributes Report** provides a full list of attributes for a type/class.

- The URL is: http://localhost/Windchill/netmarkets/jsp/carambola/createtableview/availableAttributesReport.jsp.
- ID used to reference the attribute in JSP pages
- Label used for that attribute in the UI
- Data utility mapped to that component

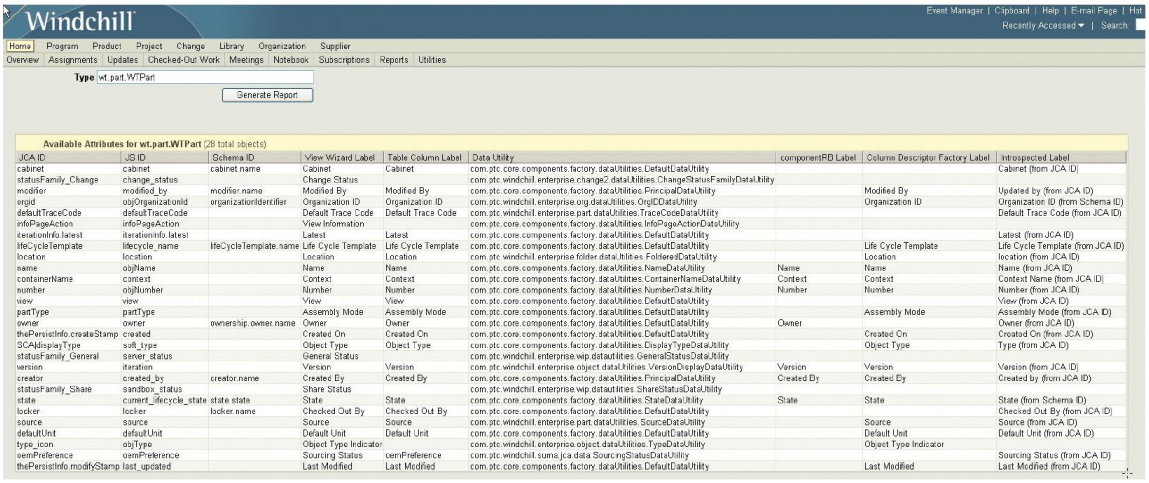


Figure 4-29: Available Attributes Report

Action Report

The Action Report is a search page that returns the details of actions matching the search criteria.

This report can be accessed via the following URL: http://localhost/Windchill/netmarkets/jsp/carambola/tools/actionReport/action.jsp

Figure 4-30: Running the Action Report

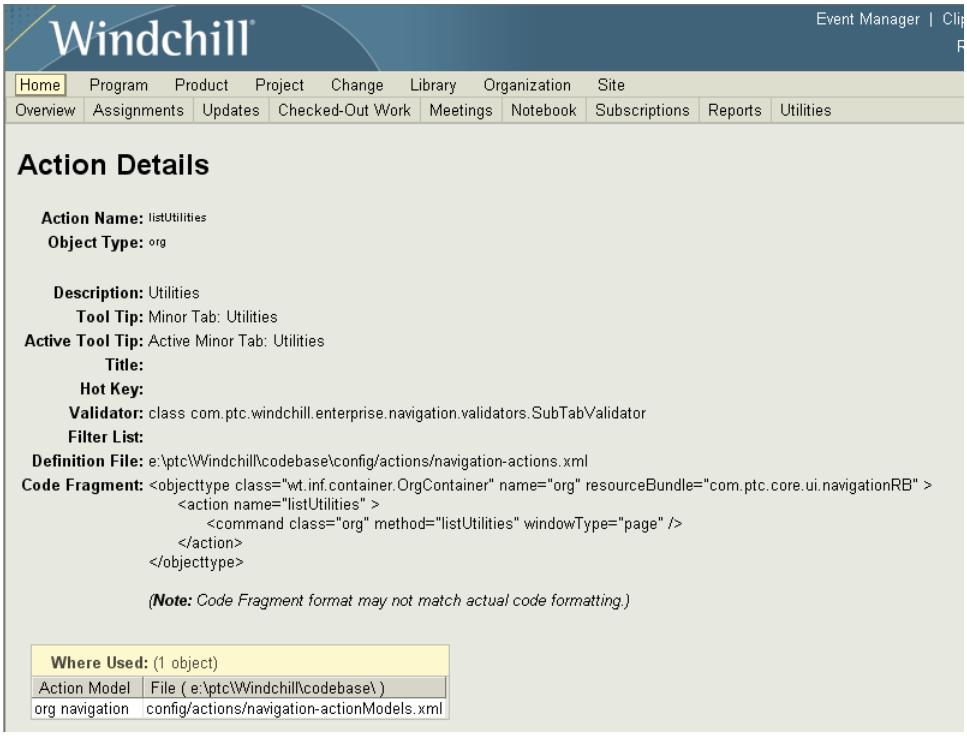


Figure 4-31: Action Details from Action Report

Exercise 4-1: Use the JCA Tools and Examples

Objectives

- Navigate the database with a JCA example.
- Run reports on actions, models and services.
- View available component examples.

Scenario

Practice with existing JCA tools and examples — available OOTB.

Step 1. All of the following steps begin from a browser opened to `http://localhost/Windchill/netmarkets/jsp/carambola`.

Step 2. Browse to **database > remove_configurable_views.jsp**.

Step 3. Browse to **database > strider.jsp**.

Step 4. Browse to **ie > jcaSearch.jsp**.

Step 5. Browse to **lwc > table.jsp**.

Step 6. Browse to **propertyPanel > examples.jsp**.

Step 7. Browse to **svc > report.jsp**.

Step 8. Browse to **svc > tree.jsp**.

Step 9. Browse to **svc > treeWithNodeColumn.jsp**.

Step 10. Browse to **createtableview > availableAttributesReport.jsp**.

Step 11. Browse to **tags > tags.jsp**.

Step 12. Search for actions and action models.

a. Browse to **tools > actionReport > action.jsp**.

b. For **Action Model Name:**, enter `home navigation` and select `Search`. Review the actions listed; what do you think this list represents?

c. For **Action Model Name:**, enter `main navigation` and select `Search`. Review the actions listed; what do you think this list represents?

Step 13. Run a Property Report.

a. Browse to `http://localhost/Windchill/netmarkets/jsp/property`, then browse to `propertyReport.jsp`.

b. For **Type:**, enter wt.part.WTPart. Select **Report**.

Summary

After completing this module, you should be able to:

- Identify and describe the elements of the Windchill client architecture.
- Provide an overview of the OOTB tools and examples available.

Module

5

Adding Actions

In this module, you will define new actions, alter the tab model and modify the visibility of actions.

Objectives

Upon successful completion of this module, you will be able to:

- Add a new action that will be exposed in the user interface.
- Add a new action model to represent multiple actions (e.g., a drop down of actions).
- Remove an OOTB action from the user interface.
- Add and remove tabs or sub tabs.
- Customize the set of UI components (actions or other UI elements) that the administrators of the site, organization or containers can manage using the role-based visibility feature
- Add new action for given type to search result table.

Lecture Notes

You may use the space below to take your own notes.

Action Framework

Uses “Model-View-Controller (MVC) Model 1” architecture for page generation.

- – MVC is a design pattern to separate presentation from content from persistent data.

A request is made to a JSP or servlet. That JSP or servlet handles all responsibilities for the request, including processing the request, validating data, handling the business logic, and generating a response.

In general, the page requests data and renders it using RMI calls for data acquisition.

- Either directly from the JSP or via a bean
- Historically, call will query the data, builds NmObjects
- NmObjects are returned to the JSP
- NmObjects contain the information to be displayed on the page

Renderers encapsulate the HTML that gets generated given the set of NmObjects.

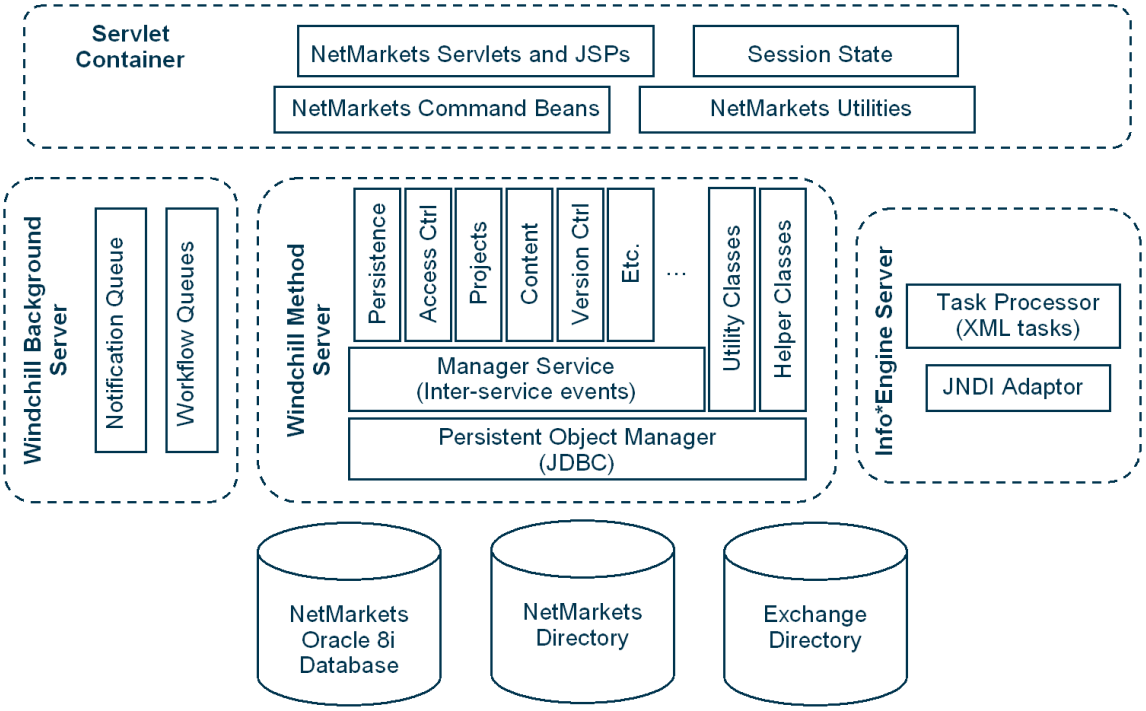


Figure 5-1: Architecture

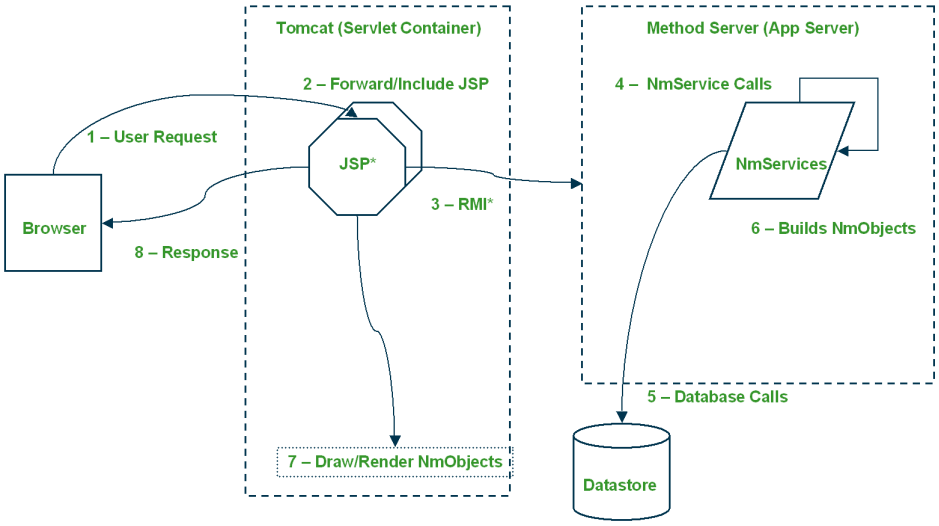


Figure 5-2: General Overview

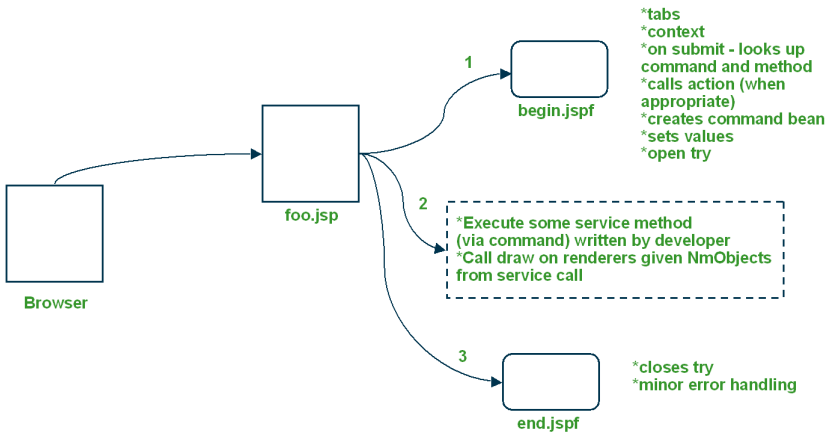


Figure 5-3: Overview of a Standard Page

Action Configuration

JCA has a number of configuration files:

- *WT_HOME/codebase/config/actons/*-actions.xml*
  - Maps an action to the class and method to execute
- *WT_HOME/codebase/config/actons/*-actionmodels.xml*
  - Models are grouped collections of actions.
  - Examples include tool bars, tabs, drop down menus.
- *WT_HOME/codebase/action_*.properties*
  - Properties for actions: icon, window size, description/label, tool tip, etc.
  - This is a localized file

Action Registry

The action registry is a Windchill service/cache that:

- Reads configuration files
- Builds individual actions.

- Builds action models:
  - These are grouped collections of actions.
  - Examples include tool bars, tabs, drop down menus.
- It is initialized when the Method Server starts.
- It can be reloaded.
  - From a Windchill shell, execute: windchill com.ptc.netmarkets.util.misc.NmActionServiceHelper.
  - This command only reloads actions and action models; not JSP pages, or Method Server properties

.

Actions

```
<objecttype name="customPart" class="com.ptc.netmarkets.part.NmPart">
...
<action name="view" enabledwhensuspended="true">
 <command class="com.ptc.netmarkets.part.NmPartCommands"
           method="view" windowType="page"/>
 <access name="read"/>
</action>
...
<action name="checkin" checkaccess="true">
 <command class="com.ptc.netmarkets.part.NmPartCommands"
           method="checkIn"
           windowType="popup"/>
 <access name="update"/>
</action>
...
<action name="part_create"
        checkaccess="true"
        uicomponent="PROJECT_PART_CAPABILITY">
 <command class="com.ptc.netmarkets.part.NmPartCommands"
           method="wizardCreate"
           windowType="popup"/>
 <access name="update"/>
</action>
</objecttype>
```

Object Types

Object types are logical packaging of actions

- Loosely mapped to business model
- Should make sense to users (i.e., part not wtpart)

Action Types

When declaring an action, the action definition has several properties, including windowType:

- Page

```
<command class="com.ptc.netmarkets.part.NmPartCommands"
         method="view"
         windowType="page"/>
```

  - Displays an entirely new page
  - This will be a link located according to the action type and name
- Popup

```
<command class="com.ptc.netmarkets.part.NmPartCommands"
         method="checkIn"
         windowType="popup"/>
```

  - JS Function that launches a new window/wizard
  - Typically a wizard is rendered
  - Sets some form variables, generates a URL, and opens a new window

Action Models

- Models are grouped collections of actions.
- Examples include tool bars, tabs, drop down menus.
- Stored is *-actionmodels.xml

```
<model
   name="customActionModel">
 <action
    name="view"
     type="customPart"/>
 <action
    name="checkin"
    type="customPart"/>
 <action
    name="part_create"
    type="customPart"/>
</model>
```

NmObjects Details

Netmarkets objects are Java objects used in the UI framework.

There are two versions of Netmarkets objects:

1. Lightweight representation of a Persistable
2. Simple representation of an HTML Element

1. Lightweight representation of a Persistable
   - NmProject is a lightweight version of a Project2 container object
2. The other kind of Netmarkets object encapsulates information about an HTML element.
   - NmString wraps a string in an object

NmString also does the following:

- Includes the stylesheet class
- Determines whether to handle URLs embedded in the string in some special way, etc
- Handles String truncation

A Call to the Action Service

This is an example of JSP code making a direct call to NmObjects (e.g., NmCommandBean, NmHTMLActionModel, etc.). This is now all encapsulated by the Windchill client architecture.

In 9.0, developers will need to know the OOTB Windchill TagLibs and use the TagLib tags in custom JSP pages, not the APIs.

This is only presented as reference as exposure to the APIs represented in the classes represented by the TagLib tags that developers will use.

```
<%@ include file="/netmarkets/jsp/util/begin.jsp‟ %>
NmCommandBean cb = new NmCommandBean();
// Setup various attributes and configuration in cb
// Get action model via some RMI call
NmHTMLActionModel nmm = null;
nmm = NmActionServiceHelper.service.getActionModel("customActionModel ", null);

modelBean.setModel(nmm);

// table - draw
// The following 3 lines are 1 unbroken line:
NmTableRenderer.draw(modelBean, objectBean,  sessionBean, localeBean,  urlFactoryBean,
     actionBean,  stringBean,  linkBean,  nmcontext,  checkBoxBean,textBoxBean,
     radioButtonBean,  textAreaBean,  comboBoxBean,dateBean,"true", null, out, request, response);

<%@ include file="/netmarkets/jsp/util/end.jsp"%>
```

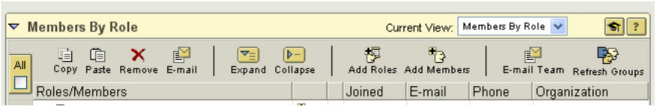


Figure 5-4: Pseudo Screen Shot

How does the Framework Represent Those Actions?

All the JSPs are relative to netmarkets/jsp.

Pages call each other via embedded JavaScript:

```
javascript:popupAction('nmpartpart_create',
   '/netmarkets/jsp/customPart/part_create.jsp',
   …
   'com.ptc.netmarkets.part.NmPartCommands',
   'wizardCreate',
   'height=375,width=485',…'P107')
```

- Window title is nmpartpart_create – generated (also has a timestamp when clicked)
- We're using a relative URL here
- Type is customPart (from actions.xml)
- part_create is the action (from actions.xml)
- In codebase/netmarkets/jsp there has to be a package called customPart
- Package name maps to object type
- Codebase/netmarkets/jsp/customPart/part_create.jsp: takes object type and actionfrom actionmodel
- NmPartCommands is the class we're calling
- wizardCreate is the method
- Height/width come from action.properties though it can be overridden via cookie

JCA Actions

Represent a user action in the UI:

- Can appear in the UI as a:
  - tab
  - icon
  - link
  - menu item

Action Examples

Figure 5-5: Navigation Tabs

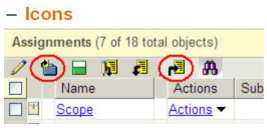


Figure 5-6: Icons

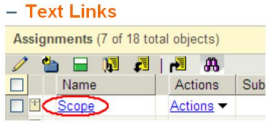


Figure 5-7: Text Links

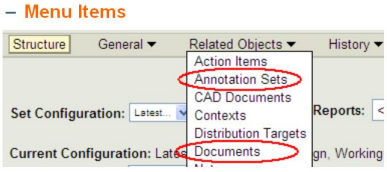


Figure 5-8: Menu Items

JCA Actions

- Implemented by com.ptc.netmarkets.util.misc.NmAction
- Properties of NmAction include:
  - *actionClass* - class implementing a command
  - *action method* - method implementing a command
  - *type* - type of the object an action is intended for
  - *action* - name of the action - not a user visible name
  - *windowType* – page, popup, etc
  - *desc* – the user visible name for the action in the UI.
  - *renderType* – what mechanism to use to determine the url
  - *contextObject* – the object for which this action should apply

What is an Action Model?

- A collection of actions

- Implemented by com.ptc.netmarkets.util.misc.NmHTMLActionModel
- Used to logically group actions that appear in the UI, e.g., part-related actions or tool bar actions.

Note: In X-10, action models can be nested.

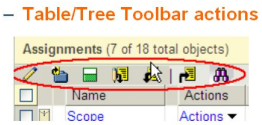


Figure 5-9: Table/Tree Toolbar Actions

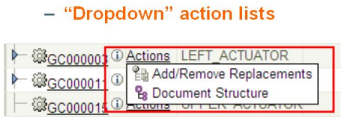


Figure 5-10: Dropdown action lists

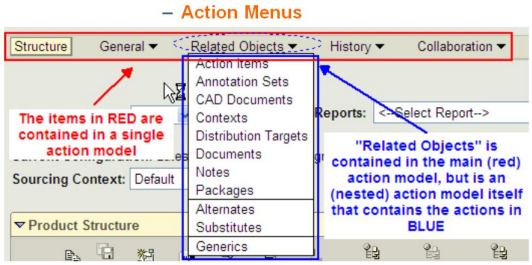


Figure 5-11: Action Menus

What is the JCA Action Service?

- Loads and caches NmActions and NmActionModels
- Uses the action registry files to initialize actions and action models
- Implemented by com.ptc.netmarkets.util.misc.StandardNmActionService
- Calls a "Validation service" to determine whether or not actions are available
- Returns NmActions and NmActionModels requested by clients
- If a request is for something that doesn't exist in the registry, an exception is returned
- Developers need to know the IDs (names) of the actions and action models to be included in customization JSP pages.
- "Common components" will typically make the call to the service

Configuring JCA Actions

- The JCA action registry is composed of two types of XML files:
  - *actions.xml*
    - Maps an action to a class and method to execute
    - Yields an NmAction object
  - *actionmodels.xml*
    - For a given model, identifies which actions can be performed (from *actions.xml*)
    - Yields an NmHTMLActionModel (which contains 1 to many NmAction objects)

Each functional module should contain its own version of each of these files

- Source files should be located in *<ModuleDir>/src_web/config/actions*, e.g., */wcEnterprise/ChangeManagement/src_web/config/actions*
- Files should be named *<ModuleName>-actions.xml* or *<ModuleName>-actionModels.xml*, e.g., *ChangeManagement-actions.xml & ChangeManagement-actionModels.xml*.

Configuring Actions using *actions.xml*

Here is a sample block of XML in an *actions.xml* file:

```
<listofactions>
  <objecttype name="problemReport" class="wt.change2.WTChangeIssue"
              resourceBundle="[...]changeManagementActionsRB">
    <action name="create" >
      <command class="[...]CreateProblemReportFormProcessor" method="execute" windowType="popup"/>
    </action>
    <action name="editModifyContentOnly">
      <command class="[...]EditObjectFormProcessor" method="execute" windowType="popup"
               url="[...]editContentWizard.jsp"/>
    </action>
  </objecttype>
</listofactions>
```

Actions are grouped together using object-types.

These object-types are declared inside of the **listofactions** element of the **actions.xml* file and they are referenced by the **name** attribute, which should be unique to each object type.

Note:

1. Actions need not be in any particular order. One action - doCreation - will invoke the other actions according which will be defined in the JSPs later
2. Actions can point directly to OOTB components.
3. If the same object/action combination is defined in another *actions.xml* file, the one that is read last will be the one used. The object types are actually merged together to create the super set of actions for a given object type.

actions.xml: objecttype Element

Possible objecttype Attributes

- name (required): The name used to reference this object type
- class (required): Object class for the enclosed actions
- resourceBundle (optional): Class name for the default resource bundle to use for the properties of the actions that are to be localized

```
<objecttype name="problemReport" class="wt.change2.WTChangeIssue"
            resourceBundle="[...]changeManagementActionsRB">
```

actions.xml: action Element

Actions are defined inside of the *objecttype* element and use the *name* attribute to identify the action.

```
<objecttype name="problemReport" class="wt.change2.WTChangeIssue"
            resourceBundle="changeManagementActionsRB">
  <action name="create" >
   <command class="CreateProblemReportFormProcessor" method="execute" windowType="popup"/>
  </action>
  <action name="editModifyContentOnly">
   <command class="EditObjectFormProcessor" method="execute" windowType="popup"
             url="editContentWizard.jsp"/>
  </action>
</objecttype>
```

The action elements are placed inside the objecttype element.

The only required attribute for an action is the *name* attribute.

Note: For the other possible attributes of the action element, see the DTD at *ClientArchitecture/CommonComponents/src_web/actions.dtd*

How is the URL for an Action Determined?

URL associated with an action is determined in one of several ways, depending on how that action is registered in the *actions.xml* file.

Default is to map to a URL of the form:

- "http://<hostname>/<webapp>/netmarkets/jsp/<type>/<action>.jsp"
- where "<type>" is the names of the objecttype and "<action>" is the action specified in *actions.xml*.

For example, given the following action definition from actions.xml:

```
<objecttype name="customPart" class="com.ptc.netmarkets.part.NmPart">
  <action name="view" enabledwhensuspended="true">
   <command class="com.ptc.netmarkets.part.NmPartCommands" method="view" windowType="page"/>
   <access name="read"/>
  </action>

</objecttype>
```

Since a renderType attribute is not specified in the action element, the URL assigned to this action is http://<hostname>/<webapp>/netmarkets/jsp/customPart/view.jsp.

The corresponding JSP file is *WT_HOME/codebase/netmarkets/jsp/customPart/view.jsp*.

ActionModels.xml

As we mentioned before, individual actions can be organized into collections called "action models".

Models are declared inside of the *actionmodels* element of the **actionModels.xml* file and they are referenced by the name attribute, which should be unique.

```
<model name="myNewActionModel">
  <action name="myNewAction" type="myobject"/>
  <action name="yourNewAction"  type="myobject"/>
</model>
```

Actions are defined inside of the <model> start and end tags.

The Action name and type must match something in one of the *actions.xml files.

Both the name and type are required within the <action> tag.

Action Models can be nested, so that Action Models can be a collection of Action Models.

Recommended grouping and ordering of actions

If you are not sure where an action or separator fits within your action model, see the spreadsheet on Action Information

It contains a worksheet, called Action Grouping for Menus, that contains a list of the commonly known actions and where they fit in.

If you cannot determine which group an action may fall into, talk with your UI designer or Product Definition person.

Also, add your actions to this spreadsheet for future tracking, if you do not see them there.

Best Practices

Actions can and will behave differently when launched from different launch points. When creating a new action, be sure that it works from various launch-points such as:

- Folder browser

- Menu bar
- Template Processor page
- DCA Page
- Details Page

Exercise 5-1: Modify Default List of Part Actions

Objectives

- Research action model definition.
- Update the action model.

Scenario

The customer has requested for actions to be removed from the Part Property Page **Actions** menu.

Step 1. Find the right action model.

a. Look at the available action model files (**actionmodels.xml*) in *WT_HOME/codebase/config/actions*.

b. The best fit to modify part actions is *PartManagement-actionmodels.xml*.

c. There are several sections labelled as “information page Actions”; each section is for a different class (WTPart, WTProduct, WTLibrary, etc.). The correct section is “<model name="more parts actions" menufor="wt.part.WTPart">”.

Step 2. Change an action model.

a. Edit *WT_HOME/codebase/config/actions/PartManagement-actionmodels.xml*, under the appropriate section for WTParts:

Example:

```
<!-- Part information page Actions list -->
   <model name="more parts actions" menufor="wt.part.WTPart">
```

b. Locate the line “<model name="more parts actions" menufor="wt.part.WTPart">”.

c. After that model line, remove the following lines:

Although the example below is truncated, you will be able to find the correct lines to remove.

Example:

```
<action name="viewDefaultRepresentationPV"      type="wvs"/>         <!-- Open in ProductV
<action name="viewDefaultRepresentationPVLite"  type="wvs"/>         <!-- Open in ProductV
<action name="launchPSE"                        type="explorer" />   <!-- Open in Product
<action name="launchABE"                        type="explorer" />   <!-- Open in Manufauc
<action name="separator"                        type="separator"/>   <!-- ================
<action name="WFDOWNLOAD"                       type="epmdocument"/> <!-- Add to Workspace
<action name="checkin"                          type="wip"/>         <!-- Check In -->
<action name="WFCHECKIN"                        type="pdmObject"/>   <!-- Check In for new
                                                                          checked out
<action name="checkout"                         type="wip"/>         <!-- Check Out -->
<action name="checkoutAndEdit"                  type="part"/>        <!-- Check Out and Ed
<action name="undocheckout"                     type="object"/>      <!-- Undo Checkout --
<action name="WFCANCELCHECKOUT"                 type="pdmObject"/>   <!-- Undo Checkout fo
<action name="edit"                             type="part"/>        <!-- Edit -->
<action name="editNewPartInWorkspace"           type="part"/>        <!-- Edit newly creat
<action name="editCheckedOutPartInWorkspace"    type="part"/>        <!-- Edit part checke
<action name="editPartCommonAttrsWizard"        type="part"/>        <!-- Edit Common Attr
<action name="tabular_input"                    type="part"/>        <!-- Edit Structure -
<action name="separator"                        type="separator"/>   <!-- ================
```

a. Save *PartManagement-actionmodels.xml*.

Step 3. Reload the configuration.

a. There are two methods for reloading action/tab configuration while Windchill PDMLink is running. One is to restart Windchill PDMLink; the second is to execute the following command from a Windchill shell:

Example:

```
windchill com.ptc.netmarkets.util.misc.NmActionServiceHelper
```

Editing Tabs and Sub Tabs

The Windchill 9.0 UI Framework is similar to the 8.0 Netmarkets UI, and the 9.0 process is slightly different, as well.

The Files that Control Windchill Navigation

- *codebase/config/actions/navigation-actions.xml*
  - Action definitions for the actions that represent the tabs and sub tabs in Windchill.
- *codebase/config/actions/navigation-actionModels.xml*
  - Action model definitions used in the navigation. This includes the main tab list, the sub tab list, recent lists and the header actions lists.
- *codebase/action.properties* or an *rbinfo* file:
  - Labels for the action
- *codebase/netmarkets/css/nmstyles.css*
  - Style classes used to create the look and feel of the Windchill header and footer
- *codebase/netmarkets/jsp/<TYPE>/<ACTION>.jsp*
  - For the default action *renderType*, this is the JSP file to display when an action is performed.
  - For other renderTypes, other pages/classes/methods can be executed.

Adding Tabs in Windchill

Updates to the models and actions are in separate files from the OOTB files.

- Custom actions belong in *WT_HOME/codebase/confg/actions/custom-actions.xml*.
- Custom models/navigation belong in *WT_HOME/codebase/confg/actions/custom-actionModels.xml*.

To add tabs and sub tabs:

1. Add actions to *custom-actions.xml*, one for the top tab and each sub tab.
2. Copy the OOTB “home navigation” model in *navigation-actions.xml* into *custom-actions.xml*.
3. Add an element to the (copied) “home navigation” model in *custom-actionModels.xml*. This element will represent the top tab.
4. Add a new model to *custom-actionModels.xml* that will represent the sub tab model.
5. Edit *actions.properties* to provide labels for the tabs and sub tabs.
   - There is no custom *action.properties* file.
   - For each locale, also edit *actions_LOCALE.properties*.
   - The alternative is to use resource bundles.
6. Create JSP files for each sub tab — there is no need to create a JSP file for a top tab. When opening a top tab, a sub tab is opened automatically, by default.
   - The JSP file will include the *begin.jspf* and *end.jspf* fragments to initialize the page, generate the header and close the JSP page.

Debugging Tab Customization

Appending *jcaDebug=1* to a URL shows the page in debug mode.

Instead of showing the Label "My Tab", it shows the name of the action, the *actionModel* and the Validator for the action.

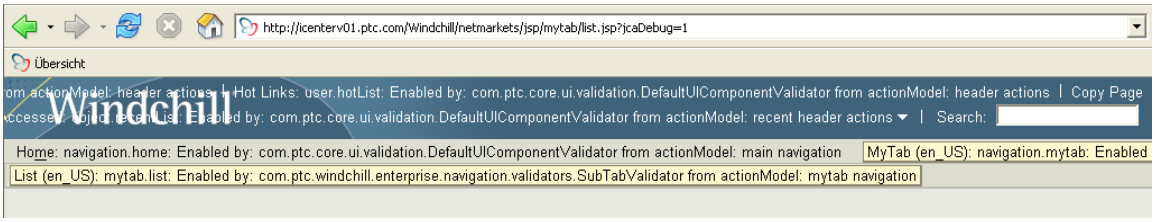


Figure 5-12: Debugging Tab Customization

Displayed Text

- Tomcat and the Method Server must be restarted to reload *action.properties*.
- The preferred method for updating labels is to use resource bundles.

Note: Eventually, updating *action.properties* will not be allowed.

To use a resource bundle instead of editing *action.properties*:

1. Update *custom-actions.xml* to reference a custom Resource Bundle

```
<objecttype
name="navigation" class="" resourceBundle="com.gsdev.client.NavigationRB">
 <action name="home" renderType="GENERAL">
    <command class="netmarkets" method="servlet/Navigation?tab=home" windowType="page"/>
   </action>
...
</objecttype>
```

2. Create the Resource Bundle nested under *WT_HOME/src*. Technically, new entries could be added to a new resource bundle in *WT_HOME/wtCustom*.
3. Add entries to the *rbInfo* files, in a similar but different format than *action.properties*:

```
navigation.newTab.description.value=My New Tab
newTab.list1.description.value=A New List
newTab.list2.description.value=A New JCA Table
```

4. Execute ResourceBuild <JAVA_FORMAT_PATH_TO_RB_FILE> from a Windchill shell.

Exercise 5-2: Incorporate Custom JSP Pages into the UI

Objectives

- Add actions.
- Update resource bundles for action labels.
- Use *jcaDebug* to display properties.

Scenario

A customer wants to create a first-level tab; 5 sub tabs, and open custom JSP pages for the sub tabs.

Step 1. Add actions to *custom-actions.xml*, one for the top tab and each sub tab.

a. There will be 5 actions, total.

b. Create or open *WT_HOME/codebase/config/actions/custom-actions.xml*.

c. To be a part of the top tabs, the action created must have *objecttype* equal to “navigation”, but the action name can be anything that does not conflict with existing “navigation” action names. The class attribute is a reference to a modeled class for which the tab applies, but is optional; navigation will work even if there is not a concrete class association for the actions. Add the following to *custom-actions.xml* for the top tab action:

Example:

```
<objecttype name="navigation" class="">
  <action name="newTab" renderType="GENERAL">
    <command class="netmarkets" method="servlet/Navigation?tab=newTab" windowType="page"/>
  </action>
</objecttype>
```

d. Custom actions can have any *objecttype* name and action name as long as the *objecttype* name does not conflict with an existing *objecttype* name. Add the following to *custom-actions.xml* for the sub tab actions:

Example:

```
<objecttype name="newTab" class="">
  <action name="list1" >
    <command windowType="page"/>
  </action>
  <action name="list2" >
    <command windowType="page"/>
  </action>
  <action name="list3" >
    <command windowType="page"/>
  </action>
  <action name="list4" >
    <command windowType="page"/>
  </action>
  <action name="list5" >
    <command windowType="page"/>
  </action>
</objecttype>
```

Step 2. Define the initial action model.

a. Copy the OOTB “main navigation” model.

b. Open *navigation-actionModels.xml* as a reference file.

c. Open or create *custom-actionModels.xml*.

d. From *navigation-actions.xml*, copy the *model* element with the name “main navigation” and paste into *custom-actionModels.xml*.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE actionmodels SYSTEM "actionmodels.dtd">
<actionmodels>

<!-- Main navigation -->
<model name="main navigation">
  <action name="home" type="navigation"/>
  <action name="product" type="navigation"/>
  <action name="project" type="navigation"/>
  <action name="change" type="navigation"/>
  <action name="library" type="navigation"/>
  <action name="org" type="navigation"/>
  <action name="site" type="navigation"/>
</model>

</actionmodels>
```

Step 3. Insert an element to represent the top tab.

a. The custom action model is now the same as the OOTB model. After making changes, the edits will override the OOTB model, thereby adding custom tabs.

b. Add a line to the “main navigation” model to insert the new top tab:

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE actionmodels SYSTEM "actionmodels.dtd">
<actionmodels>

<!-- Main navigation -->
<model name="main navigation">
  <action name="home" type="navigation"/>
  <action name="newTab" type="navigation"/>
  <action name="product" type="navigation"/>
  <action name="project" type="navigation"/>
  <action name="change" type="navigation"/>
  <action name="library" type="navigation"/>
  <action name="org" type="navigation"/>
  <action name="site" type="navigation"/>
</model>

</actionmodels>
```

Step 4. Add a new model to *custom-actionModels.xml* that will represent the sub tab model.

a. The name of the sub tab model is a concatenation of the *name* and *type* elements added to the “main navigation” model.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE actionmodels SYSTEM "actionmodels.dtd">
<actionmodels>

<!-- Main navigation -->
<model name="main navigation">
  <action name="home" type="navigation"/>
  <action name="newTab" type="navigation"/>
  <action name="product" type="navigation"/>
  <action name="project" type="navigation"/>
  <action name="change" type="navigation"/>
  <action name="library" type="navigation"/>
  <action name="org" type="navigation"/>
  <action name="site" type="navigation"/>
</model>

<model name="newTab navigation" defaultActionType="newTab" defaultActionName="list1">
  <action name="list1" type="newTab"/>
  <action name="list2" type="newTab"/>
  <action name="list3" type="newTab"/>
  <action name="list4" type="newTab"/>
  <action name="list5" type="newTab"/>
</model>

</actionmodels>
```

b. *defaultActionType* and *defaultActionName* specify the name and type of an action to open when the top tab is selected.

Step 5. Add labels for the tabs and sub tabs.

a. Edit *action.properties*.

Example:

```
navigation.newTab.description=Top Tab
newTab.list1.description=Sub Tab 1
newTab.list2.description=Sub Tab 2
newTab.list3.description=Sub Tab 3
newTab.list4.description=Sub Tab 4
newTab.list5.description=Sub Tab 5
#object.actionsMenu.description=Actions
# according to current UI standard there is no tooltip for the actionsMenu

object.create.icon=newobj.gif
object.create.disabled_icon=export_collapse.gif
object.create.description=Create
...
```

b. Further, edit all *actions_LOCALE.properties* files in a similar, but localized, fashion.

Step 6. Restart the Method Server and Tomcat in order to reload the *action.properties* changes.

Step 7. Create JSP files for each sub tab.

a. There is no need to create a JSP file for a top tab. When opening a top tab, a sub tab is opened automatically, by default, as defined in *custom-actionModels.xml*

b. Create *WT_HOME/netmarkets/jsp/newTab/list1.jsp*:

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<H1>This is WT_HOME/codebase/netmarkets/jsp/newTab/list1.jsp</H1>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

c. Create the other sub tab JSP files. The pages will have similar content, but with a different text string inside the files to uniquely identify each JSP page.

Step 8. Test.

a. There should be a top tab.

b. There should be several sub tabs.

c. Each sub tab should open a separate JSP page.

Step 9. Once the labels are working with *action.properties*, switch to resource bundles.

a. Update *custom-actions.xml*

Example:

```
<objecttype
name="navigation" class="" resourceBundle="com.gsdev.client.navigationRB">
  <action name="newTab" renderType="GENERAL">
    <command class="netmarkets" method="servlet/Navigation?tab=newTab" windowType="page"/>
  </action>
</objecttype>

<objecttype name="newTab" class="" resourceBundle="com.gsdev.client.navigationRB">
  <action name="list1" >
    <command windowType="page"/>
  </action>
  <action name="list2" >
    <command windowType="page"/>
  </action>
  <action name="list3" >
    <command windowType="page"/>
  </action>
  <action name="list4" >
    <command windowType="page"/>
  </action>
  <action name="list5" >
      <command windowType="page"/>
  </action>
</objecttype>
```

b. Create a resource bundle file *WT_HOME/src/com/gsdev/client/navigationRB.rbInfo*. Add the following text:

Example:

```
ResourceInfo.class=wt.tools.resource.StringResourceInfo
ResourceInfo.customizable=true
ResourceInfo.deprecated=false

navigation.newTab.description.value=Client
newTab.list1.description.value=Basic Table
newTab.list2.description.value=I*E Search
newTab.list3.description.value=Table
newTab.list4.description.value=Tree
newTab.list5.description.value=Custom Obj
```

c. xconfmanager -s wt.resource.locales.fromDefault="en,en_US,en_GB" -t codebase/user.properties -p.

d. Execute ResourceBuild com.gsdev.client.navigationRB from a Windchill shell.

e. Restart the Method Server and Tomcat in order to reload the resource bundle changes.

Step 10. Test.

a. The top tab and sub tab labels should be from resource bundles.

Step 11. Add “&jcaDebug=1” to the end of the URL.

a. The custom tabs and sub tabs should open, but also display detailed information about the actions and action models.

Validators

"Validation" encompasses activities performed to determine what a user can see or do.

Examples of Windchill validation include:

- Should the **Create Part** action be displayed?
- If a user selects **Check Out** for an object, should the action be allowed to happen?
- Is the data entered by a user in a create wizard acceptable?

In the past, each client architecture (HTML Template Processor, JCA, DCA) had its own mechanism for validating actions, which led to duplicate code and inconsistent behavior throughout the product. (e.g., a given action was available on a JCA page, but that same action was not available on an HTML Template Processor page).

- In 9.0, there is a new service can be called by client applications to perform validation activities on the Method Server.
- This service is available to all client architectures.
- This service allows server-side validation code to be written once and be called by any client architecture.

What does the Validation Service do?

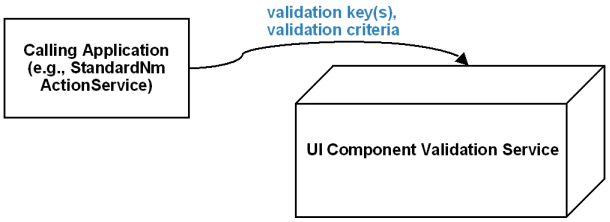
The service is a Java-based API to perform three types of validation:

1. Pre-validation
   - Attempts to answer the questions: "**Should something appear to the user in the UI? And, if so, should it be editable/selectable?"**
   - For example, "should we display and enable the "Create Part" action for user A in in the folder browser toolbar inside container B?"
   - Pre-Validation can be performed for actions or other UI components (status glyphs, attributes, tables, etc.)
2. Post-select validation
   - Attempts to answer the question, "should the operation that was just selected in the UI be allowed to proceed?"
   - For example, "can we allow the checkout of parts A, B, and C?"
3. Post-submit validation
   - Attempts to answer the question, "is the data the user just entered valid?"
   - For example, "when a user clicks **Next** in the **Create Part** wizard, can the user continue to the next step — or do they need to modify some data (e.g., name, number) in the current step?"

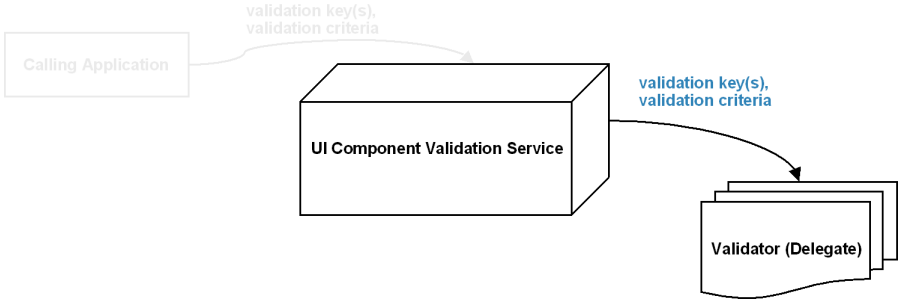
Service Details

At a high level, a caller invokes an API on the service, passing:

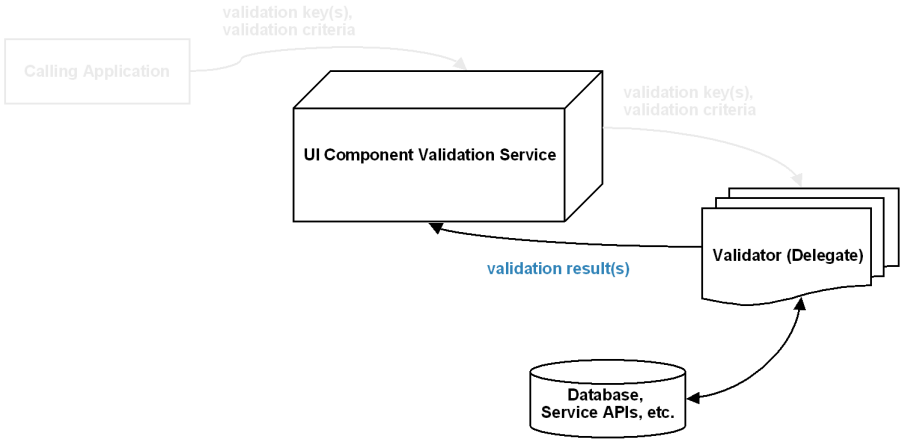
1. A UIValidationKey or Collection of UIValidationKey objects identifying the action(s) or UI Component(s) to be validated.
   - A UIValidationKey has two attributes:
     a. an action/component name (e.g., "create")
     b. an object type (e.g., "part").
2. A UIValidationCriteria object, which is a bean containing data required to perform validation (e.g., current user, working container, context object(s), etc.)



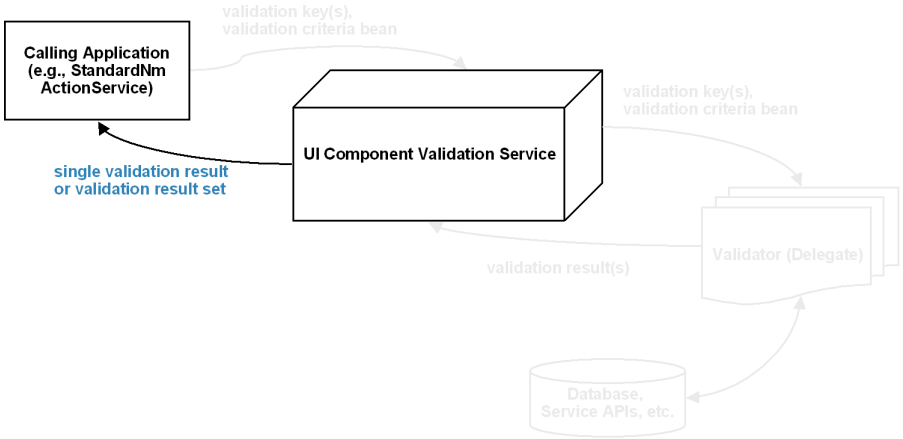
For each key passed in, the service finds a corresponding delegate (a.k.a. “Validator”) and passes the client-provided bean to each Validator.



The Validator performs the necessary logic to determine the status of the identified action or UI Component, and returns the results to the service.



The service bundles the result(s) returned by the Validator(s) and returns them to the client.



In most cases, the common components should make the necessary calls to the validation service, and interpret the results.

Creating a Validator

To create a Validator, create a subclass of com.ptc.core.ui.validation.DefaultUIComponentValidator.

There are currently five public methods defined in DefaultUIComponentValidator. A subclass can override from one to all five methods:

- performFullPreValidation()
- performLimitedPreValidation()
- validateFormSubmission()
- validateSelectedAction()
- validateSelectedMultiSelectAction()

These five methods return com.ptc.core.ui.validation.UIValidationResultSet objects.

Note: For methods not overridden, the default behavior of “always enable” and “always permit” will be inherited from DefaultUIComponentValidator.

Registering a Validator

To associate a Validator extension class with a particular validation key (action), a *service.properties* entry is required.

The validation service uses these entries to find the right Validator for a given validation id (action name).

- e.g., wcEnterprise/EnterpriseUI/src/com/ptc/windchill/enterprise/enterpriseui-service.properties.xconf

Add a custom *service.properties* file by adding a custom file to the property *wt.services.applicationcontext.WTServiceProviderFromProperties.defaultPropertyFiles*.

The xconf entry should have this format:

```
<Service context="default"
         name="com.ptc.core.ui.validation.UIComponentValidator">
  <Option selector="<component ID>"
          requestor="null"
          serviceClass="<validatorClass>"/>
</Service>
```

- The selector value is the component ID that gets passed into the validation service.
  - For actions, the component ID will be the action name from *actions.xml*:

```
<objecttype name="changeable" class="wt.change2.Changeab
   <actio name="addAffectedData" ajax="row" renderType='
   <command windowType="popup"/>
   </action>
```

Figure 5-13: Action Name

  - For attributes (or table columns), the component ID will be the component descriptor.
- The *serviceClass* value is the fully-qualified name of your Validator subclass
- So to map the addAffectedData action to com.ptc.windchill.TrainingExampleValidator, we would create the following xconf entry:

```
<Service context="default"
         name="com.ptc.core.ui.validation.UIComponentValidator">
  <Option selector="addAffectedData"
          requestor="null"
          serviceClass="com.ptc.windchll.TrainingExampleValidator"/>
</Service>
```

requestor Attribute

Most of the time, the *requestor* attribute is NOT necessary.

- In this case, *requestor* can either be set to null or omitted altogether.

In some cases, the action name alone is not specific enough:

- For example, there are many actions named create in the system, and we may not want them to all use the same validator.
- *requestor* attribute can be used to indicate a particular validator to apply only to actions of a given object type.

Type-based Lookup Using the *requestor* Attribute

Consider a Validator used for an action called *create*, but only if the object type being created is a problem report.

The create:problemReport action is defined in *actions.xml*:

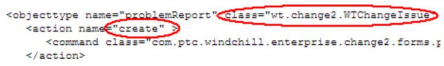


Figure 5-14: Defining create:problemReport

Assuming the validator is called com.ptc.windchill.TrainingExmpleValidator2.

Use the *class* attribute from the actions.xml entry as our *requestor* attribute in the validator registry xconf file.

```
<Service
   context="default"
   name="com.ptc.core.ui.validation.UIComponentValidator">
  <Option
    selector="create"
    requestor="wt.change2.WTChangeIssue"
    serviceClass="com.ptc.windchll.TrainingExampleValidator2"/>
</Service>
```

Note: Any action named create that is a subclass of wt.change2.WTChangeIssue will also be mapped to this validator, unless there is a different validator registered for that subclass.

Verifying a Validator is Registered

There is a utility (UIComponentValidatorFactory) that can be run to verify whether or not a custom validator has been registered.

```
V:\>java com.ptc.core.ui.validation.UIComponentValidatorFactory pasteAsCopy
Registered validators:
pasteAsCopy -> com.ptc.core.foundation.saveas.validators.PasteValidator
```

Figure 5-15: Verifying a Validator is Registered

Details on Implementing A Validator

1. Gather information from the UIValidationCriteria (user, container context, etc.)
2. Perform some business logic, which may include calling other services or APIs
3. Return a result (or set of results in some cases) indicating whether or not the desired action/component is valid.

validateSelectedAction

Called by a client application wanting to know whether or not an action selected by a user from the UI can be performed.

- e.g.: A user selects “revise” from a list of actions for a document. Should the revise action proceed?

The method signature for validateSelectedAction looks like this:

```
public UIValidationResult validateSelectedAction (
```

```
    UIValidationKey key, UIValidationCriteria criteria,
    Locale locale) throws WTException
```

The method takes three arguments:

1. UIValidationKey indicating the key that was used to identify the action being validated (think of this as the action name and object type)
2. UIValidationCriteria object, and
3. Locale

The first and third arguments should be fairly self-explanatory. The UIValidationCriteria object is the validation criteria bean that was previously mentioned. It holds data that the client set, and passed to the validation service (e.g., user, container, context object(s), etc.)

The return type is UIValidationResult with attributes including a status indicating whether or not the action should be permitted:

```
// If you want to enable the action/component, do this:
resultSet.addResult(new UIValidationResult(validationKey, UIValidationStatus.ENABLED));
// If you want to disable the action/component, do this:
resultSet.addResult(new UIValidationResult(validationKey, UIValidationStatus.DISABLED));
// If you want to hide the action/component, do this:
resultSet.addResult(new UIValidationResult(validationKey, UIValidationStatus.HIDDEN));
```

validateSelectedMultiSelectAction

Similar to the validateSelectedAction.

- Only difference is that this method will be called to validate a multi-select action, and will therefore return multiple results
- e.g.: A user selects “checkout” from a table header to perform the checkout of several selected parts in the table. Should we allow all of the parts to be checked out?

The method signature for validateSelectedMultiSelectAction looks like this:

```
public UIValidationResultSet validateSelectedMultiSelectAction (
   UIValidationKey key, UIValidationCriteria criteria,
   Locale locale) throws WTException
```

Arguments are identical to those passed to validateSelectedAction

This method returns an object of type UIValidationResultSet

- Set of UIValidationResult objects

.

validateFormSubmission

Invoked on the service by a client wanting to validate whether or not the user-entered data in a form is valid.

- e.g.: A user clicks “next” after entering data on the details step of a “create part” wizard. Should we allow them to go to the next step, based on what they’ve entered?

Arguments and return type for *validateFormSubmission* are identical to those for *validateSelectedAction:*

```
public UIValidationResult validateFormSubmission (
   UIValidationKey key, UIValidationCriteria criteria,
   Locale locale) throws WTException
```

Pre-Validation methods

The last two methods you can override in your Validator implementation are:

- performFullPreValidation
- performLimitedPreValidation

The methods provide similar functionality, as they are both called by clients wanting to know whether an action (or other UI component) should be enabled, disabled, or hidden in the UI.

The intended usage of performLimitedPreValidation is that a client would call it when they want to do a quick check of whether or not an action should appear in the UI. A client calling this method should know that they are not guaranteed that actions will be accurately hidden in all cases – we are sacrificing accuracy in favor of performance.

On the other hand, a client would call performFullPreValidation in cases where they want accurate results, at the potential cost of decreased performance.

In general, the only time performLimitedPreValidation is called by JCA clients is for table row icon actions

- If there are several actions in each row and several rows in a table, performance can degrade quickly
- If your action appears as an icon in a table row, and you want it validated, you must implement the performLimitedPreValidation method in your validator
- Actions generated as part of the “Actions” drop-down link in table rows are validated via performFullPreValidation, since only one row’s actions gets validated at a time

e.g.: Use performLimitedPreValidation when rendering a table in which each row could have action icons for check-out, check-in, revise, or delete.

e.g.: Use performFullPreValidation when rendering a dropdown list of actions on a doc details page.

The argument list and return types for each Pre-Validation method are identical:

```
public UIValidationResultSet performFullPreValidation (
   UIValidationKey key, UIValidationCriteria criteria,
   Locale locale) throws WTException
public UIValidationResultSet performLimitedPreValidation (
   UIValidationKey key, UIValidationCriteria criteria,
   Locale locale) throws WTException
```

The validation service actually does some additional up-front checks for the pre-validation methods, before calling your validators:

1. Check to see if the component should ever be available based on the set of Windchill solutions installed.
2. Checks to see if the component should ever be available based on the client that the user is browsing in (PSE, Wildfire, DTI, etc.)
3. Checks to see if an administrator has used their privileges to hide the component from the active user by calling the RoleBasedUIService
4. If the above checks passed, call the Validator registered for the component

Solution Groups

The first check performed by the validation service in pre-validation is to verify that the component is valid for the installed set of solutions.

To accomplish this, the validation service creates a cache of invalid components at startup

The cache is created by calling all of the UIComponentSolutionGroups registered in the system and getting a list of invalid components from each group

Each application team is responsible for writing a UIComponentSolutionGroup with solution-based logic for the actions they own

Client Groups

If a component being pre-validated passes the solution group check, the validation service will check to see if it’s valid for the client that the user is browsing from.

This is accomplished by calling a UIComponentClientGroup

Role-Based Checking

If the pre-validation passes the solution-based checks and the client-based checks, the validation service will then call the RoleBasedUIService to verify that an administrator has not configured the system to hide the component from the given user.

Pre-Validation Statuses

Finally, if the pre-validation checks the previous three slides pass, the validation service calls your validator to see if the given component should be displayed.

There are three UIValidationStatus values that you can potentially return from a pre-validation method:

1. ENABLED – should be returned if the action should be visible and the user may invoke it
2. DISABLED – should be returned if the action can not be invoked by the user, but the same action could potentially be enabled for the same user under different circumstances
3. HIDDEN – should be returned if the action would never be available for the user under any circumstances

Exercise 5-3: Create a Tab Visible only to Administrator

Objectives

- Create a validator which checks the user name and then determines whether to display or hide a tab based on user name.

Scenario

Create a first-level tab only visible to the “wcadmin” Administrative user.

Detailed Description

From previous exercises, there should be an action named “newTab” associated with a first-level tab. The focus of this exercise will be to control the display of this tab/action.

Step 1. From previous exercises, there should be an action named “newTab” associated with a first-level tab.

Step 2. Create the validator class.

a. Create a class com.gsdev.client.UIComponentValidator that extendscom.ptc.core.ui.validation.DefaultUIComponentValidator.

b. Add a public method performFullPreValidation () with arguments com.ptc.core.ui.validation.UIValidationKey, com.ptc.core.ui.validation.UIValidationCriteria and java.util.Locale, return object com.ptc.core.ui.validation.UIValidationResultSet and throws WTException.

c.

Example:

```
WTPrincipal wtp = SessionHelper.manager.getPrincipal();
if ( (wtp.getName()).equals( "Administrator" ) ) {
  resultSet.addResult(new UIValidationResult(validationKey, UIValidationStatus.ENABLED));
} else {
  resultSet.addResult(new UIValidationResult(validationKey, UIValidationStatus.HIDDEN));
}
return resultSet;
```

d. Compile the class into *WT_HOME/codebase*.

Step 3. Register the validator.

a. The action to be controlled is **newTab** and this validation applies to any object type (**null**), so the property that must be set is *wt.services/svc/default/com.ptc.core.ui.validation.UIComponentValidator/newTab/null/0*.

b. The validator class is com.gsdev.client.UIComponentValidator, so the value of *wt.services/svc/default/com.ptc.core.ui.validation.UIComponentValidator/newTab/null/0* is *com.gsdev.client.UIComponentValidator/duplicate*.

c. Use xconfmanager to set *wt.services/svc/default/com.ptc.core.ui.validation.UIComponentValidator/newTab/null/0* equal to *com.gsdev.client.UIComponentValidator/duplicate* with a target file *codebase/service.properties*.

d. Confirm that both *site.xconf* and *codebase/service.properties* have been updated.

Step 4. Restart the Method Server and Tomcat in order to process the changes to Windchill properties.

Step 5. Test the custom validation.

a. Open a browser and login as the user “demo”.

b. Verify that the tab does not appear.

c. Close the browser.

d. Open a browser and login as the user “wcadmin”.

e. Verify that the tab appears.

Summary

After completing this module, you should be able to:

- Add a new action that will be exposed in the user interface.
- Add a new action model to represent multiple actions (e.g., a drop down of actions).
- Remove an OOTB action from the user interface.
- Add and remove tabs or sub tabs.
- Customize the set of UI components (actions or other UI elements) that the administrators of the site, organization or containers can manage using the role-based visibility feature
- Add new action for given type to search result table.

Module

6

Display a Basic Table

This module addresses the basics of displaying tables based on the Windchill Client Architecture. Further, the various mechanisms for gathering data is explained and practiced.

Objectives

Upon successful completion of this module, you will be able to:

- Display a basic table using the Windchill client architecture.
- Configure how data is retrieved for a Windchill client architecture table.
- Use an Info*Engine Task to retrieve data for a Windchill client architecture table component.

Lecture Notes

You may use the space below to take your own notes.

JCA Common Components

JCA includes a series of reusable, configurable, pre-built “common components” to build UIs.

- These common components are taglib tags.
- taglib documentation is available from **www.ptc.com > Support > Download Reference Documents**.
- Windchill TLDs are located in *WT_HOME/codebase/WEB-INF/tlds*

Windchill supports:

- configuring existing components
- using common components to build new UIs
- creating new components

Building a JCA Component

The basic steps to use any JCA component (table, tree, property panel, wizard, etc.) are:

1. Describe the component
2. Retrieve the necessary data
3. Render the component

Note: The concepts illustrated are for tables but apply to many other UI elements (wizards, info pages, etc.).

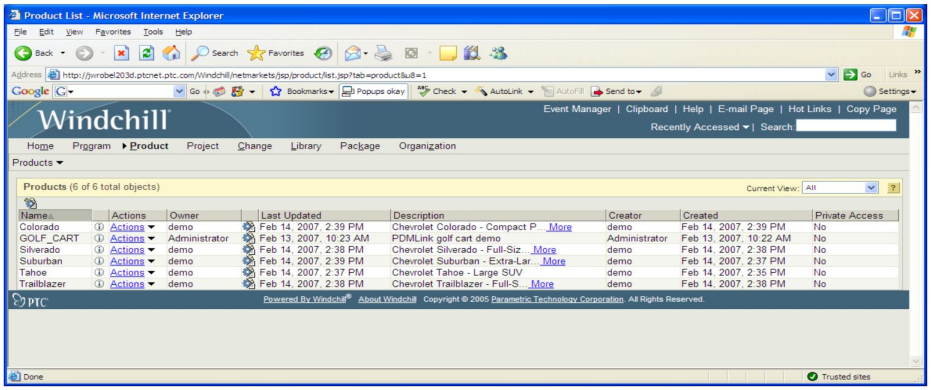
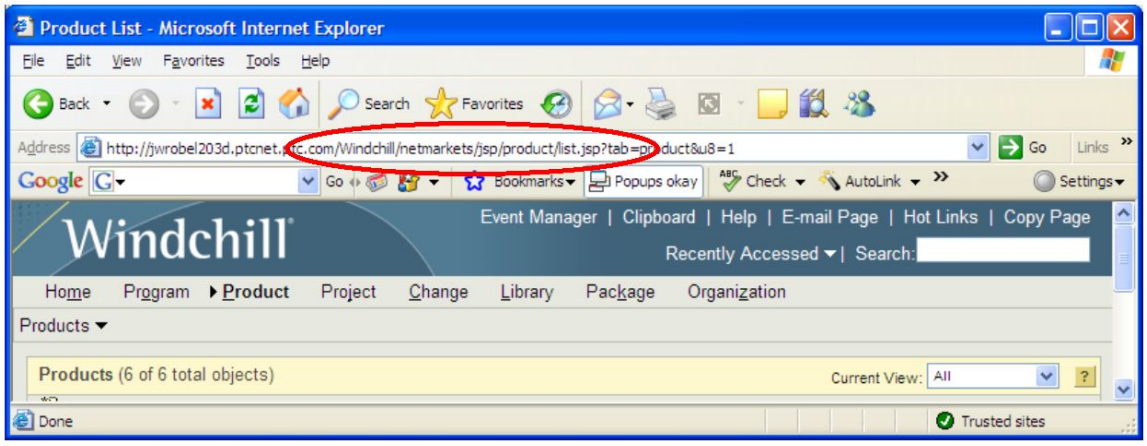


Figure 6-1: Windchill 9.0 Products List

Finding the Source JSP File Locations

- In this case, the URL includes *netmarkets/jsp/product/list.jsp*
- Look in *WT_HOME/codebase/netmarkets/jsp/product* for *list.jsp*



WT_HOME/netmarkets/src_web/netmarkets/jsp/product/list.jsp

```
<%@page import="com.ptc.core.ui.navigationRB" %>
<% request.setAttribute("browserWinTitleConst", navigationRB.WIN_TITLE_PROD_TAB_LIST); %>
<% request.setAttribute(NmAction.SHOW_CONTEXT_INFO, "false"); %>
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib tagdir="/WEB-INF/tags" prefix="tags"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/containercomponents" prefix="containercomponent"%>

<%-->Get preference value to decide on no of rows to be displayed in a table<--%>
<containercomponent:getContextListTablesPagingSize pagingSize="noOfRows"/>

<%@ page import="com.ptc.windchill.pdmlink.product.server.processors.productResource" %>

<fmt:setLocale value="${localeBean.locale}"/>
<fmt:setBundle basename="com.ptc.windchill.pdmlink.product.server.processors.productResource" />
<fmt:message var="productListTableHeader" key="<%= productResource.PRODUCT_TABLE_TITLE %>" />
<fmt:message var="orgIDLabel" key="<%= productResource.ORG_ID_LABEL %>" />

<%-->Build a table descriptor and assign it to page variable td<--%>
<jca:describeTable var="tableDescriptor" id="netmarkets.product.list"
                   type="wt.pdmlink.PDMLinkProduct" label="${productListTableHeader}"
                   configurable="true">
   <jca:setComponentProperty key="actionModel" value="product list"/>
   <jca:describeColumn id="name" isInfoPageLink="true" sortable="true"/>
   <jca:describeColumn id="infoPageAction" sortable="false"/>
   <jca:describeColumn id="nmActions" sortable="false"/>
   <jca:describeColumn id="containerInfo.owner" sortable="true"/>
   <jca:describeColumn id="orgid" sortable="true"/>
   <jca:describeColumn id="type" sortable="false"/>
   <jca:describeColumn id="thePersistInfo.modifyStamp" sortable="true"/>
   <jca:describeColumn id="containerInfo.description" sortable="true"/>
   <jca:describeColumn id="containerInfo.creator" sortable="true"/>
   <jca:describeColumn id="thePersistInfo.createStamp" sortable="true"/>
   <jca:describeColumn id="containerInfo.privateAccess" sortable="true"/>
</jca:describeTable>

<%-->Get a component model for our table<--%>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
              serviceName="com.ptc.windchill.enterprise.product.ProductListCommand"
              methodName="getProducts">
          <jca:addServiceArgument value="${tableDescriptor.id}"/>
</jca:getModel>

<c:set var="currentClass" value="<%=wt.pdmlink.PDMLinkProduct.class%>" />
<tags:msgStatus model="${tableModel}" containerClass="${currentClass}"/>

<%-->Get the NmHTMLTable from the command<--%>
```

```
<jca:renderTable showCustomViewLink="false" helpContext="ProductsHelp" model="${tableModel}"
                 showCount="true" showPagingLinks="true" pageLimit="${noOfRows}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Initialization

From *product/list.jsp*:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
```

- Include *begin.jspf*
- Include directive to include a JSP Fragment (.jspf) that is included in most JCA pages – begin.jspf
- Renders Windchill header and 1st and 2nd level navigation tabs
- Takes care of much of the JCA "housekeeping"
  - Instantiates the beans (context, clipboard, etc)
  - Executes a command (for actions only)
  - Includes some JS functions
  - Includes the common HTML pieces (<html>...<body>...etc.)
  - Creates ONE form for the page data (called mainform)
  - Provides support for closing pop ups and refreshing the main page
- If it ends up being included multiple times on the same page, the infrastructure is smart enough to only use it once.

From *product/list.jsp*:

```
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
```

- JSP taglib directive indicating that we want to use the components taglib and prefix the tags with jca.
- By convention, use the jca prefix to reference tags in the components taglib.
- taglib directive to include the tags defined a the TLD with the URI
- A prefix of "jca" will be used to reference the tags

Use Common Components

The following tags use common components to build and render a table:

- jca:describeTable
- jca:getModel
- jca:renderTable

```
<%-->Build a table descriptor and assign it to page variable td<--%>
<jca:describeTable var="tableDescriptor" id="netmarkets.product.list"
                            type="wt.pdmlink.PDMLinkProduct"
                            label="${productListTableHeader}" configurable="true">
</jca:describeTable>

<%-->Get a component model for our table<--%>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
               serviceName="com.ptc.windchill.enterprise.product.ProductListCommand"
               methodName="getProducts">
          <jca:addServiceArgument value="${tableDescriptor.id}"/>
</jca:getModel>

<%-->Get the NmHTMLTable from the command<--%>
<jca:renderTable showCustomViewLink="false" helpContext="ProductsHelp"
                         model="${tableModel}" showCount="true" showPagingLinks="true"
                         pageLimit="${noOfRows}"/>
```

describeTable Example

```
<%-->Build a table descriptor and assign it to page variable td<--%>
<jca:describeTable var="tableDescriptor" id="netmarkets.product.list"
```

```
                          type="wt.pdmlink.PDMLinkProduct"
                          label="${productListTableHeader}" configurable="true">
   <jca:setComponentProperty key="actionModel" value="product list"/>
   <jca:describeColumn id="name" isInfoPageLink="true" sortable="true"/>
   <jca:describeColumn id="infoPageAction" sortable="false"/>
   <jca:describeColumn id="nmActions" sortable="false"/>
   <jca:describeColumn id="containerInfo.owner" sortable="true"/>
   <jca:describeColumn id="orgid" sortable="true"/>
   <jca:describeColumn id="type" sortable="false"/>
   <jca:describeColumn id="thePersistInfo.modifyStamp" sortable="true"/>
   <jca:describeColumn id="containerInfo.description" sortable="true"/>
   <jca:describeColumn id="containerInfo.creator" sortable="true"/>
   <jca:describeColumn id="thePersistInfo.createStamp" sortable="true"/>
   <jca:describeColumn id="containerInfo.privateAccess" sortable="true"/>
</jca:describeTable>
```

getModel Example

```
<%-->Get a component model for our table<--%>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
              serviceName="com.ptc.windchill.enterprise.product.ProductListCommand"
              methodName="getProducts">
          <jca:addServiceArgument value="${tableDescriptor.id}"/>
</jca:getModel>
```

renderTable Example

```
<%-->Get the NmHTMLTable from the command<--%>
<jca:renderTable showCustomViewLink="false" helpContext="ProductsHelp"
                         model="${tableModel}" showCount="true" showPagingLinks="true"
                         pageLimit="${noOfRows}"/>
```

end.jspf

```
<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Describing a Component

The basic steps to produce any JCA component (table, tree, property panel, wizard, etc.) are:

1. **Describe the component**
2. Retrieve the necessary data
3. Render the component

Define Table to be Rendered

From *product/list.jsp*:

```
<%-->Build a table descriptor and assign it to page variable td<--%>
<jca:describeTable var="tableDescriptor" id="netmarkets.product.list"
                   type="wt.pdmlink.PDMLinkProduct" label="${productListTableHeader}"
                   configurable="true">
   <jca:setComponentProperty key="actionModel" value="product list"/>
   <jca:describeColumn id="name" isInfoPageLink="true" sortable="true"/>
   <jca:describeColumn id="infoPageAction" sortable="false"/>
   <jca:describeColumn id="nmActions" sortable="false"/>
   <jca:describeColumn id="containerInfo.owner" sortable="true"/>
   <jca:describeColumn id="orgid" sortable="true"/>
   <jca:describeColumn id="type" sortable="false"/>
   <jca:describeColumn id="thePersistInfo.modifyStamp" sortable="true"/>
   <jca:describeColumn id="containerInfo.description" sortable="true"/>
   <jca:describeColumn id="containerInfo.creator" sortable="true"/>
   <jca:describeColumn id="thePersistInfo.createStamp" sortable="true"/>
   <jca:describeColumn id="containerInfo.privateAccess" sortable="true"/>
</jca:describeTable>
```

describeTable tag

```
13 <%-->Build a table descriptor and assign it to a page variable named “tableDescriptor”<--%>
14 <jca:describeTable var="tableDescriptor" id="netmarkets.product.list" type="wt.pdmlink.PDMLinkProduct"
15             label="${productListTableHeader}" configurable="true">
   ...
28 </jca:describeTable>
```

Figure 6-2: The describeTable tag

- The *describeTable* tag defines a table:
  - Creates a table descriptor object
  - Assigns it to the page variable specified by the var attribute (in this case, *tableDescriptor*)
  - This allows the table descriptor to be referenced in the JSP using the Expression Language (EL).
- Table descriptor:
  - An instance of a ComponentDescriptor object that describes a table component.

describeTable tag: ComponentDescriptor

A *ComponentDescriptor* encapsulates metadata about a UI component.

- *ComponentDescriptor* objects can be nested.
  - By convention, a table, tree, info page, attribute panel component descriptor has children
  - In this case child *ComponentDescriptors* correspond to columns or properties

- Some of the primary properties of a *ComponentDescriptor* are:

id	unique key
label	Display text associated with the UI element
componentMode	(VIEW, EDIT, SEARCH, CREATE)
componentType	(INFO, PICKER, SEARCH, TABLE, WIZARD, WIZARD_TABLE)

describeTable tag: TableDescriptor

A *table* descriptor is really just an instance of a *ComponentDescriptor* where:

- componentType = TABLE
- componentMode = VIEW

The *describeTable* tag creates a table descriptor

- Creates a table descriptor
- Allows us to reference that table descriptor object in our JSP using the handle we specified (“tableDescriptor”) via the var attribute.

The describeTable tag: Other Attributes

```
13 <%-->Build a table descriptor and assign it to a page variable named “tableDescriptor”<--%>
14 <jca:describeTable var="tableDescriptor" id="netmarkets.product.list" type="wt.pdmlink.PDMLinkProduct"
15              label="${productListTableHeader}" configurable="true">
```

Figure 6-3: The describeTable tag

- Look for the *describeTable* tag in *components.tld*.
  - The id attribute is the identifier for this table descriptor. If this table supports configurable views, then this id may be used to look up the view configurations, unless it is overridden by the *configurableTableId* attribute.
  - The type attribute is the name of the main type of object displayed in this table. This can be either a modeled or soft type. When specified, the infrastructure will use the type to look up labels and other default values when it can’t find them otherwise.

describeTable tag: More Attributes

More Attributes

```
13 <%-->Build a table descriptor and assign it to a page variable named “tableDescriptor”<--%>
14 <jca:describeTable var="tableDescriptor" id="netmarkets.product.list" type="wt.pdmlink.PDMLinkProduct"
15              label="${productListTableHeader}" configurable="true">
```

Figure 6-4: Other Attributes

- From *components.tld*:
  - The label attribute is the localized label for this component.
  - The configurable attribute indicates whether or not this table supports configurable views. Defaults to false.

setComponentProperty Tag

```
14 <jca:describeTable var="tableDescriptor" id="netmarkets.product.list" type="wt.pdmlink.PDMLinkProduct"
15              label="${productListTableHeader}" configurable="true">
16   <jca:setComponentProperty key="actionModel" value="product list"/>
```

Figure 6-5: The setComponentProperty tag

- From *components.tld*, *setComponentProperty* is:
  - A helper tag that sets properties on the component descriptor managed by a parent describe tag. Alternatively, the component descriptor to configure can be specified using the target attribute.
- Sets additional properties on the table descriptor we created with our *describeTable* tag.
- We'll discuss actions and action models in more detail later.

The describeColumn tag(s)

The next several tags defined under the *describeTable* tag are a series of *describeColumn* tags.

- Describe the columns in the table
- In our example, all of the *describeColumn* tags had two attributes: an id attribute and a sortable attribute
  - The id attribute is "the identifier for this column descriptor. This is used to look up default properties for the column. If the parent table supports configurable views, then this id is used to look up the view configurations."
  - The sortable attribute: "When true, the column should be rendered as sortable. Defaults to true."
    - So actually, we don't need to include all of the sortable="true" attributes in our tags

```
13 <%-->Build a table descriptor and assign it to a page variable named "tableDescriptor"<--%>
14 <jca:describeTable var="tableDescriptor" id="netmarkets.product.list" type="wt.pdmlink.PDMLinkProduct"
15                 label="${productListTableHeader}" configurable="true">
16   <jca:setComponentProperty key="actionModel" value="product list"/>
17   <jca:describeColumn id="name" sortable="true"/>
18   <jca:describeColumn id="infoPageAction" sortable="false"/>
19   <jca:describeColumn id="nmActions" sortable="false"/>
20   <jca:describeColumn id="containerInfo.owner" sortable="true"/>
21   <jca:describeColumn id="orgid" sortable="true"/>
22   <jca:describeColumn id="type" sortable="false"/>
23   <jca:describeColumn id="thePersistInfo.modifyStamp" sortable="true"/>
24   <jca:describeColumn id="containerInfo.description" sortable="true"/>
25   <jca:describeColumn id="containerInfo.creator" sortable="true"/>
26   <jca:describeColumn id="thePersistInfo.createStamp" sortable="true"/>
27   <jca:describeColumn id="containerInfo.privateAccess" sortable="true"/>
28 </jca:describeTable>
```

Figure 6-6: describeColumn Tags

The describeColumn tag(s): ids

How would we have known what to use as an id attribute for each of the *describeColumn* tags?

- The carambola property report.
- Find a column in an OOTB table and view that table with the jcaDebug=1 parameter appended to the URL.

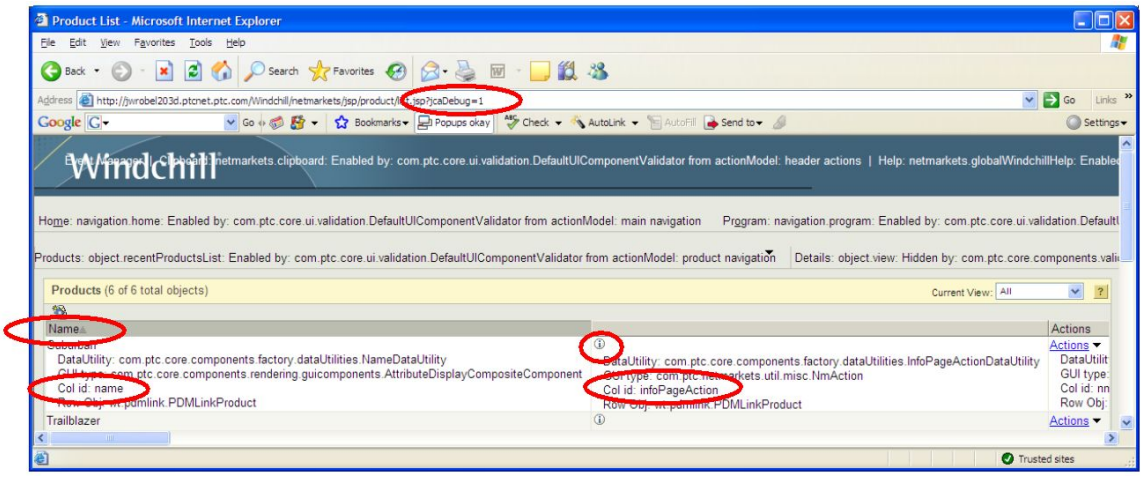


Figure 6-7: jcaDebug Example

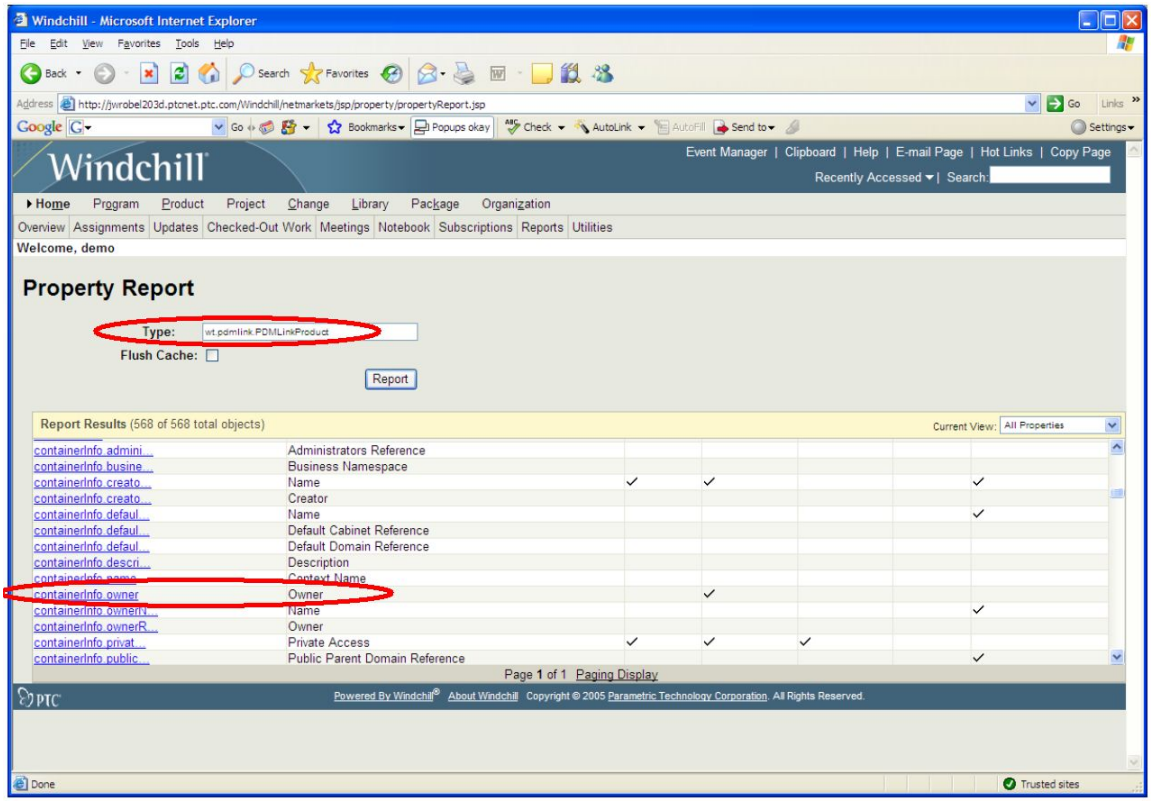


Figure 6-8: Carambola Property Report to Find Owner Column Id

Retrieving Data

The basic steps to produce any JCA component (table, tree, property panel, wizard, etc.) are:

1. Describe the component
2. **Retrieve the necessary data**
3. Render the component

getModel Tag

From *product/list.jsp*:

```
<%-->Get a component model for our table<--%>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
              serviceName="com.ptc.windchill.enterprise.product.ProductListCommand"
              methodName="getProducts">
          <jca:addServiceArgument value="${tableDescriptor.id}"/>
</jca:getModel>
```

getModel Tag

The getModel tag takes your table descriptor plus a way to get table data and returns a ComponentModel object that can be used to render the table.

- Like the describeTable tag, the result of the getModel tag is placed in the page-scoped variable defined by the var attribute.

The getModel tag provides several ways to get data:

1. By specifying a class name and method name.
   - The class must either be a Windchill service class, or must implement RemoteAccess.
   - Arguments to the method can be specified using the addServiceArgument tag.
   - The result of the method must be a QueryResult, Collection, or array of the row objects that you wish to display
2. By specifying a StatementSpec. (StatementSpec is the parent interface of QuerySpec)
3. By specifying a query command. (The command infrastructure used to get TypeInstance objects)

In this case, getModel uses the getProducts() method on the ProductListCommand class

```
30 <%-->Get a component model for our table<--%>
31 <jca:getModel var="tableModel" descriptor="${tableDescriptor}"
32           serviceName="com.ptc.windchill.enterprise.product.ProductListCommand"
33           methodName="getProducts">
34   <jca:addServiceArgument value="${tableDescriptor.id}"/>
35 </jca:getModel>
```

Figure 6-9: Component Model for Table

ComponentModel Objects

getModel tag will produce a ComponentModel object

- A ComponentModel encapsulates three other models:
  1. An *NmHTMLActionModel*
     - Contains the actions for this component (toolbar for a table, third-level navigation actions for an info page)
  2. A ComponentViewModel
     - Defines the view-related information for the component, if any
  3. A data model
     - Internally represented by an Info*Engine Group
     - Defines that actual data to be displayed in the component

ComponentModelBean

The getModel tag then delegates to the ComponentModelBean.

- A ComponentModelBean actually does the work of building a ComponentModel
- It constructs a BatchCommand with 3 subcommands to get each of the ComponentModel's 3 parts
  - ActionCommand
  - ComponentViewCommand
  - ModelCommand
- has hard-coded logic that creates one of the known ModelCommand implementations depending on the parameters specified in the JSP.
  - For example, it knows if the JSP specifies a service class and method name, to construct a ServiceModelCommand

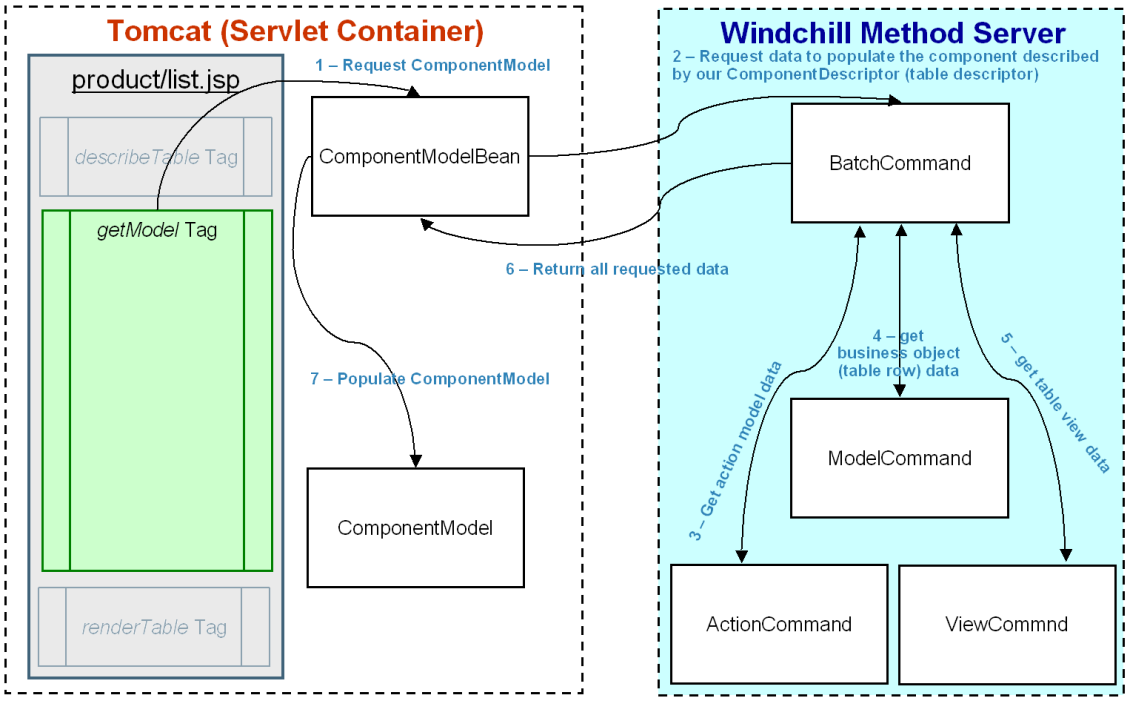


Figure 6-10: Data Acquisition (Part 1)

Acquiring JCA Data

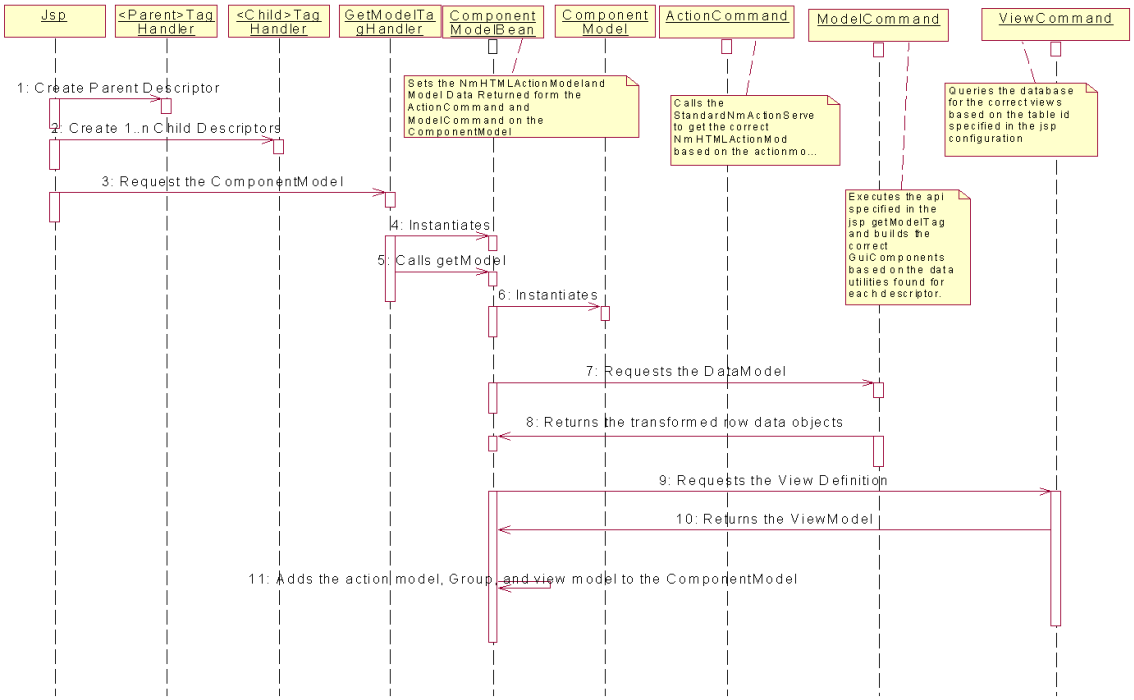


Figure 6-11: Acquiring JCA Data

Acquiring Business Object (Table Row) Data

Acquisition of the business object (table row) data

- Recall that ModelCommand is created and executed to perform business object data acquisition.
- ComponentModelBean constructs the right type of ModelCommand, based on the arguments specified in the getModel tag.
  - In our case, we included serviceName and methodName parameters in our getModel tag
  - Which indicates that a ServiceModelCommand is the type of ModelCommand that should be constructed and executed.
- The ServiceModelCommand, in turn, delegates to a ServiceModelCommandDelegate.

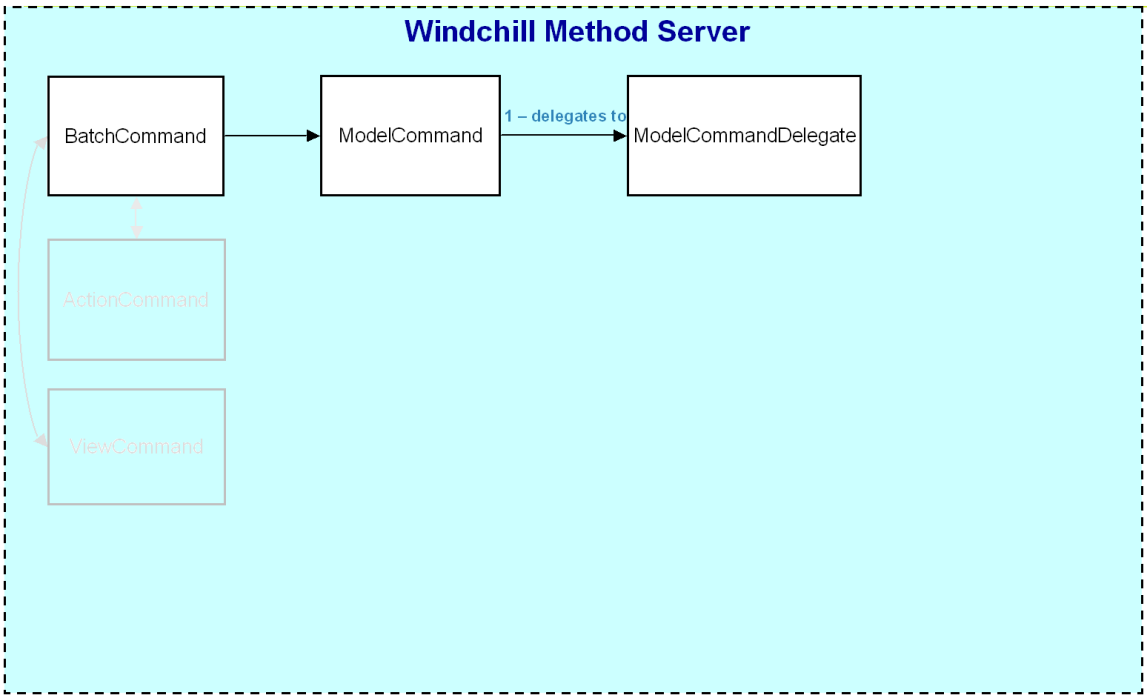


Figure 6-12: Data Acquisition (Part 2)

ServiceModelCommandDelegate

What happens in the ServiceModelCommandDelegate?

- The ServiceModelCommand(Delegate) is responsible for retrieving the business data and returning the “transformed” row data objects.
  - When we say “transformed” row data, we mean that we have done any massaging of the data in order for it to be displayed.
- To retrieve the data, the ServiceModelCommandDelegate invokes a service API using the method and class specified on the y tag.
- Once the row data has been retrieved, the ServiceModelCommandDelegate creates an instance of a DataUtilityBean, which will assist in the “transformation” of the data.
- Takes an input List of row objects and a ComponentDescriptor, and builds a resulting data model by running the row objects through the data utilities that map to the component descriptor’s child descriptors.
  - In our example, the ComponentDescriptor passed to the DataUtilityBean is the table descriptor for our table, and it’s child descriptors are its column descriptors.

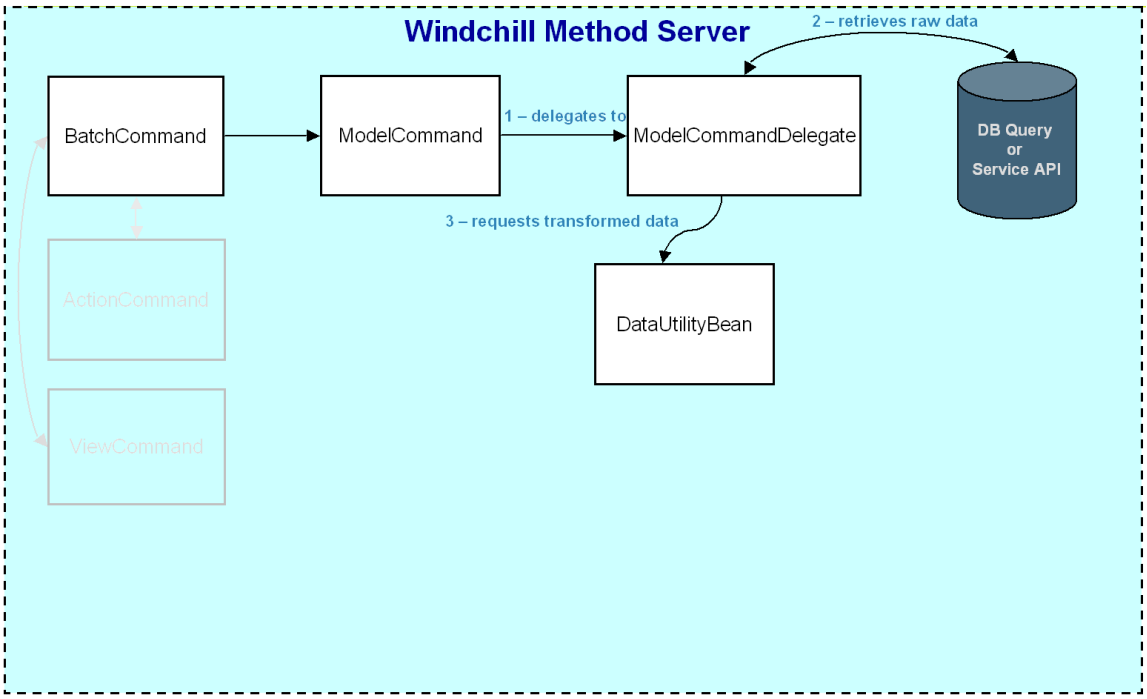


Figure 6-13: Data Acquisition (Part 2)

DataUtilityBean, DataUtilityHelper, and ModelContext

DataUtilityBean

- Uses the DataUtilityHelper to find the DataUtilities that should be used for each of the properties specified in the column descriptors.
- Passes a ModelContext to each DataUtility to communicate information about the column.
- Once the DataUtilityBean runs the row data through the DataUtilities, it consolidates the data to be returned

ModelContext

- Helper object that is passed to each DataUtility
- Gives the implementation access to the current ComponentDescriptor as well as the NmCommandBean, if present.
- Can be thought of as an abstraction of the request. It should contain everything you need to build a table.

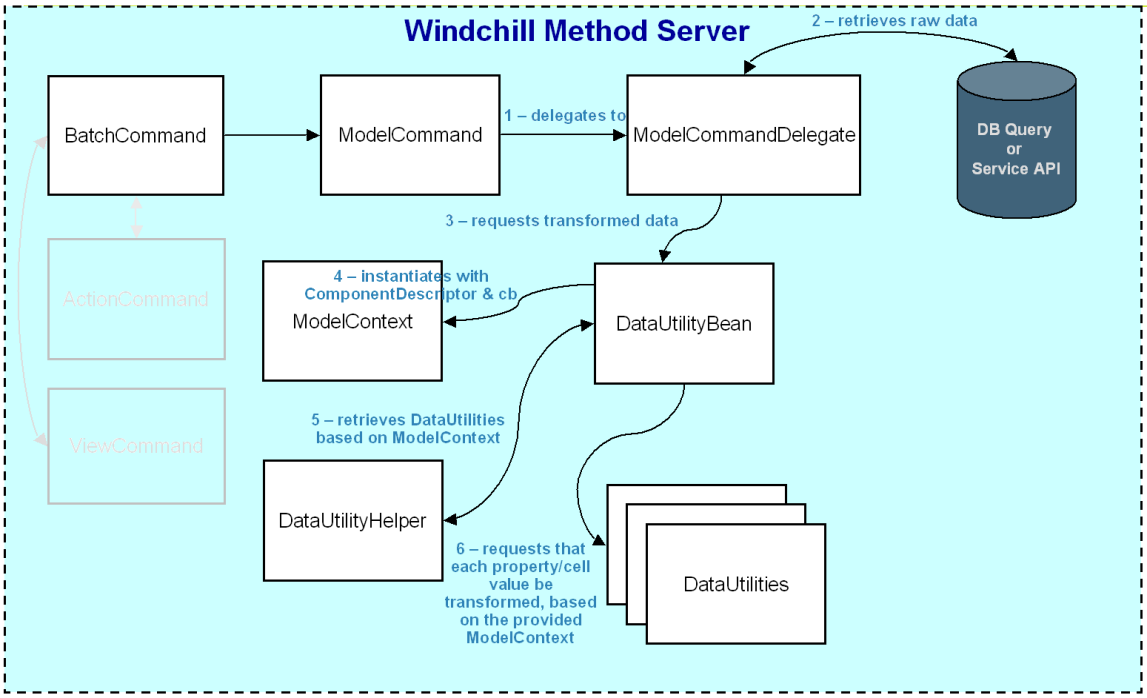


Figure 6-14: Data Acquisition (Part 2)

DataUtility

Constructs a GUIComponent for a given object and component id.

-
- Takes raw query data and converts it into the model data that a component can be built out of.
- In the case of tables, the data utilities are responsible for constructing particular cell values.
- NEVER return HTML from a DataUtility.
  - The GUIComponent's HTMLRenderer will convert the output to HTML.

The DataUtility interface has two core methods:

- void setModelData(String component_id, List objects, ModelContext mc) throws WTException;
  - allows the data utility to pre-fetch data for all the row objects that will be rendered. The method is called before getDataValue is called for any row, and is supplied a List of all the objects that will be processed.
- Object getDataValue(String component_id, Object datum, ModelContext mc) throws WTException;
  - gets the value that should be placed within a particular table cell. Typically, the object that is returned by this method should be an instance of GUIComponent

GUIComponents

A GUIComponent is an object that:

- Carries information about a UI Element that is necessary to appropriately render the element on a page.
- Includes the default renderer class that knows how to generate the appropriate display, based on the information in the GUI component.
- Example: Location picker

Figure 6-15:

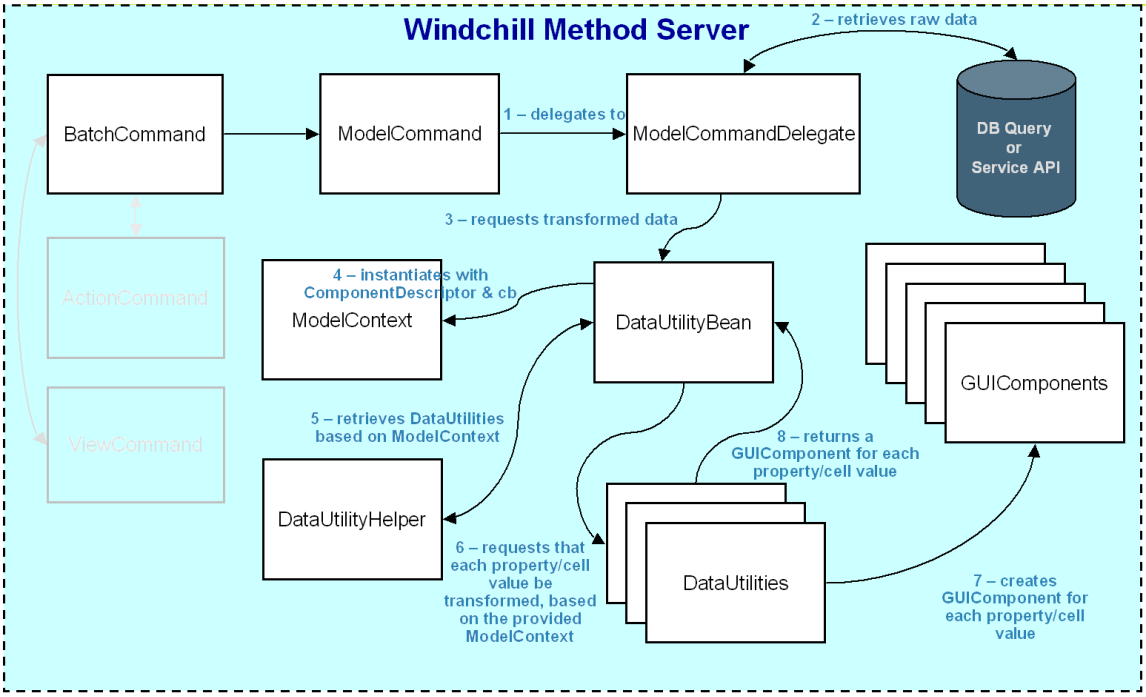


Figure 6-16: Data Acquisition (Part 2)

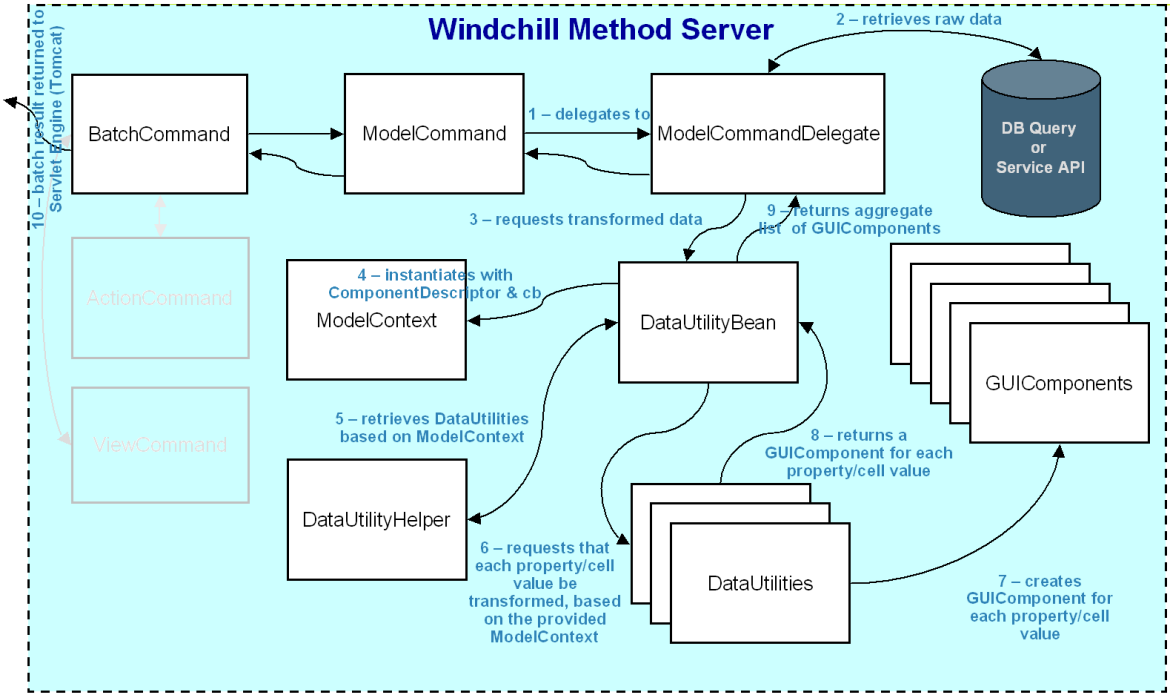


Figure 6-17: Data Acquisition (Part 2)

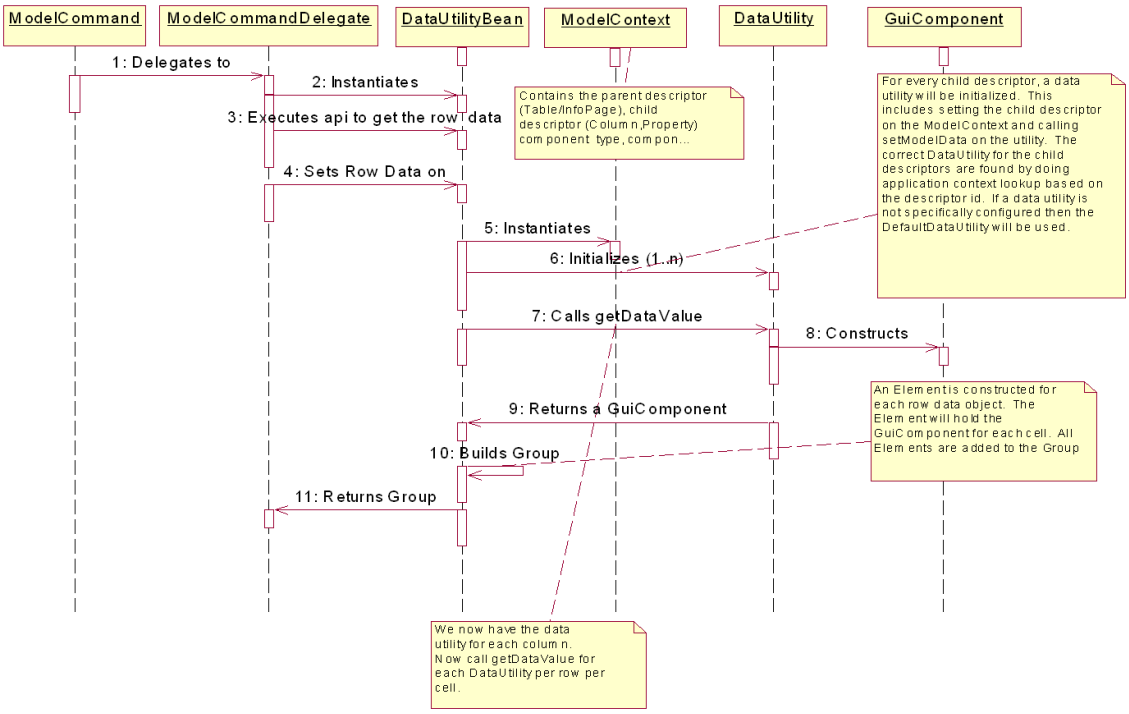


Figure 6-18: Transforming the data into GuiComponents

Additional Details for Acquiring Data

Within the framework, it is possible to retrieve and process data, without writing framework-specific Java code.

getModel tag	JSP tag Configures a Method Server data source
getIeModel tag	JSP tag Configures an Info*Engine data source
describeColumn tag	JSP tag Configures a column to display in a table
describeProperty tag	JSP tag Configures a property to display in a property panel. Supports the same API as describeColumn.
targetObject attribute	JSP tag attribute that allows you to change the “backing object”
Property report	A reporting tool to find what existing Windchill Client Architecture extension points are in the system
batch tag	JSP tag Batches together multiple getModel calls

Use getModel or getIeModel?

getModel will:

- Enable coding to Persistable and other plug and play interfaces
- Facilitate unit testing and more POJO-style development

getIeModel will:

- Facilitate federation and customization
- Limit the attributes available to you to whatever is specifically requested, rather than the entire java object
  - Performance trade-off
- Require the definition of logical attributes to return associations

getModel

The first step in using getModel is to specify how to extract each property that will be displayed in the component from the resulting data elements. Windchill Client Architecture can automatically figure out how to extract these properties. There are three scenarios:

1. Bean properties are handled automatically
   - Using API that returns a Java bean
   - Property in the UI corresponds to a property of the bean
2. Soft attributes are handled automatically
   - Soft attributes for your business objects,
   - Even if the objects you returned do not know about the soft attributes (logical id mssing)
3. Existing Windchill Client Architecture extensions
   - Logical "properties" can be used to augment the business object

Note: In order for non-Persistable objects to be selectable in the interface some additional configuration is necessary

getModel using Bean Properties

Windchill Client Architecture handles Java beans in the same way that the JSP EL does:

- Refer to a property name and the infrastructure will look for a corresponding getter on your object
- By default the UI components just render a blank in this case

For example, consider a table with a column like the following: <jca:describeColumn id="foo" .../>

- The infrastructure will look for a getFoo() method on each result object that it processes when it executes your getModel API
- If the method isn't found, the infrastructure will return a null value, for the UI component to process in whatever way it chooses

You are not limited to simple properties of the bean.

- Anything you can do with the JSP EL can also be accomplished with Windchill Client Architecture
  - bar[0]: get the first value of the bar property
- Dot notation can be used to refer to nested properties
  - foo.bar: get the nested *bar* property of a *foo* property
- Map objects are treated as beans, and can be referred to as if it had getter methods
  - If getFoo() returns a Map, then this would correspond to getFoo().get("bar")

Windchill Client Architecture uses beanutils from Apache commons to process bean properties: http://commons.apache.org/beanutils/index.html .

getModel using Soft Attributes

Windchill Client Architecture can automatically look up the soft attributes for your Typed business objects. Consider a part with a soft attribute named "baz", By adding the column descriptor as follows: <jca:describeColumn id="baz" .../> Windchill Client Architecture will work in the following order:

1. Initially fail to find a bean property for "baz" on your business object,
2. Check if "baz" could be a soft attribute of the object by checking to see if
   a. It corresponds to a logical attribute
   b. The component id is a soft attribute in external form

If Windchill Client Architecture is able to find a soft attribute definition for the object, then it will go and fetch the object's value.

Note: Retrieval of soft attributes can be expensive, so if Windchill Client Architecture determines a soft attribute retrieval is in order, it will get the soft attributes for all objects in the data set at once

So called "convenience attributes may also be used:

```
<describeColumn id="ALL_SOFT_ATTRIBUTES" .../>
```

Here is a list of convenience attributes:

- ALL_SOFT_ATTRIBUTES
- ALL_SOFT_CLASSIFICATION_ATTR IBUTES
- ALL_SOFT_NON_SCHEMA_ATTRIB UTES
- ALL_SOFT_SCHEMA_ATTRIBUTES
- ALL_SOFT_SCHEMA_ATTRIBUTES_ FOR_INPUT_TYPE
- ALL_SOFT_NON_CLASSIFICATION_ SCHEMA_ATTRIBUTES
- ALL_CUSTOM_HARD_ATTRIBUTES _FOR_INPUT_TYPE

getModel using targetObject

targetObject is used to change the result so that the infrastructure treats a property of each result object as the "row" object rather than the result object itself.

```
<jca:describeColumn id="bar" targetObject="foo" .../>
```

In the above example the attribute bar is displayed, but the backing object is foo. Whereas in the following example,

```
<jca:describeColumn id="foo.bar" .../>
```

bar is both display and target object. This may not have been the intended behavior. Explicitly specifying a targetObject leaves no doubt what the backing object is meant to be and is preferred.

targetObject must be one of the following:

- Persistable
- WTReference
- ObjectIdentifier
- NmObject
- NmOid
- NmSimpleOid

targetObject can be used with non-persistables by wrapping the object in an NmSimpleOid as follows:

```
// Construct an NmObject from a Persistable
public NmObject getNmObject(Object datum, ModelContext mc) {
 // getStringKeyFromDatum is the method to find the unique String key
 String key = getStringKeyFromDatum (datum);
 NmSimpleOid oid = new NmSimpleOid();
 oid.setInternalName(key);
 NmObject result = new NmObject();
 result.setOid(oid);
 return result;
}
```

Batching getModel calls

The batch tag can be used retrieve multiple models at once. This cuts down on interprocess communication between the servlet container and method server. To use the batch tag include the wc tag library as follows

```
<%@ taglib uri="http://www.ptc.com/windchill/taglib/core"
prefix="wc"%>
```

Use the batch tag as follows:

```
<wc:batch>
 <jca:getModel ..../>
 <jca:getModel ..../>
</wc:batch>
```

Render Component

The basic steps to produce any JCA component (table, tree, property panel, wizard, etc.) are:

1. Describe the component
2. Retrieve the necessary data
3. **Render the component**

From *product/list.jsp*:

```
<%-->Get the NmHTMLTable from the command<--%>
<jca:renderTable showCustomViewLink="false" helpContext="ProductsHelp" model="${tableModel}"
                 showCount="true" showPagingLinks="true" pageLimit="${noOfRows}"/>
```

The renderTable tag

- The rendering portion of the table is simply done by passing your model object (defined by the *getModel* tag) into the renderTable tag
- The model attribute of the *renderTable* tag uses the *var* attribute from the *tableModel* tag to indicate the model to be rendered.
- Look in *components.tld* to get the information on the other attributes specified in the tag:
  - *helpContext*: The help context for this table
  - *showCount*: optionally show the count of objects in the table title area
  - *showPagingLinks*: Determines whether or not the table is paged
- Takes an input *ComponentModel* and renders an *NmHTMLTable* out of it.

Exercise 6-1: Implement a Life Cycle List

Objectives

- Use JCA components in JSP pages to render tables that look like the Windchill PDMLink UI.
- Create a helper Java class to search for data that will be rendered in the JSP page.

Scenario

For an existing second-level tab, the second-level tab should use JCA to display all existing life cycles in a Windchill PDMLink-style table.

Detailed Description

From previous exercises, there should be a custom first-level tab and several custom second-level tabs. The focus of this exercise will be to display a table in one of the a second-level tabs.

Step 1. Create a helper.

a. Create a class com.gsdev.client.CustomerJCAHelper.

b. The helper must implement wt.methd.RemoteAccess. Any other extend statements, implement statements or thrown exceptions depend on the purpose of the class.

Example:

```
package com.gsdev.client;

import wt.method.RemoteAccess;

public class CustomerJCAHelper implements RemoteAccess {

 public CustomerJCAHelper() {
  // TODO Auto-generated constructor stub
 }

}
```

c. Add a static method getLifeCycles() with no arguments and returns wt.fc.QueryResult.

Example:

```
package com.gsdev.client;

import wt.fc.QueryResult;
import wt.method.RemoteAccess;

public class CustomerJCAHelper implements RemoteAccess {

 public CustomerJCAHelper() {
  // TODO Auto-generated constructor stub
 }

 public static QueryResult getLifeCycles () {
  return qr;
 }
}
```

d. getLifeCycles() will perform an API-based search for life cycles.

Example:

```
package com.gsdev.client;

import wt.fc.PersistenceHelper;
import wt.fc.QueryResult;
import wt.lifecycle.LifeCycleTemplate;
import wt.method.RemoteAccess;
import wt.pds.StatementSpec;
import wt.query.QuerySpec;

public class CustomerJCAHelper implements RemoteAccess {

 public CustomerJCAHelper() {
  // TODO Auto-generated constructor stub
 }

 public static QueryResult getLifeCycles () {
  QueryResult qr = null;
  try {
   QuerySpec qs=new QuerySpec( LifeCycleTemplate.class );
   qr = PersistenceHelper.manager.find((StatementSpec)qs);
  } catch (Exception e) {
   // Do nothing
  }
  return qr;
 }
}
```

a. Compile the class.

Step 2. Edit the JSP page associated with a second-level tab.

a. Use the existing second-level page *WT_HOME/codebase/netmarkets/jsp/newTab/list.jsp*.

b. Open *list.jsp* in a text editor.

c. Add the “include begin.jspf”, JCA taglib and “include end.jspf” statements.

Example:

```
<!-- begin.jspf is required to draw the header -->
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<!-- Your content should occur between begin.jspf and end.jspf -->
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

d. Save the file.

Step 3. Describe the table component.

a.

Example:

```
<!-- begin.jspf is required to draw the header -->
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<!-- Your content should occur between begin.jspf and end.jspf -->
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%-->Build a table descriptor<--%>
<jca:describeTable var="tableDescriptor" id="com.gsdev.client.list_lc" label="A Table">
   <jca:describeColumn id="name" sortable="true" label="column label; default is column id"/>
</jca:describeTable>

<!-- end.jspf is required to draw the footer -->
<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

b. Save the file.

Step 4. Acquire data with a getModel element.

a. The *serviceName* is the name of the helper class — com.gsdev.client.CustomerJCAHelper.

b. the *methodName* is the method to retrieve a list of life cycles — getLifeCycles.

Example:

```
<!-- begin.jspf is required to draw the header -->
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<!-- Your content should occur between begin.jspf and end.jspf -->
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%-->Build a table descriptor<--%>
<jca:describeTable var="tableDescriptor" id="com.gsdev.client.list_lc" label="A Table">
   <jca:describeColumn id="name" sortable="true" label="Column Label"/>
</jca:describeTable>

<%-->Get a component model for our table<--%>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
              serviceName="com.gsdev.client.CustomerJCAHelper"
              methodName="getLifeCycles">
</jca:getModel>

<!-- end.jspf is required to draw the footer -->
<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

c. Save the file.

Step 5. Render the table.

a. Add a renderTable element.

Example:

```
<!-- begin.jspf is required to draw the header -->
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<!-- Your content should occur between begin.jspf and end.jspf -->
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%-->Build a table descriptor<--%>
<jca:describeTable var="tableDescriptor" id="com.gsdev.client.list_lc" label="A Table">
   <jca:describeColumn id="name" sortable="true" label="Column Label"/>
</jca:describeTable>

<%-->Get a component model for our table<--%>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
              serviceName="com.gsdev.client.CustomerJCAHelper"
              methodName="getLifeCycles">
</jca:getModel>

<%-->Get the NmHTMLTable from the command<--%>
<jca:renderTable showCustomViewLink="false" model="${tableModel}" showCount="false"/>

<!-- end.jspf is required to draw the footer -->
<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

b. Save the file.

Step 6. Test.

a. Restart the Method Server and Tomcat.

b. Open the second-level tab and a list of life cycles should be shown in a table.

Acquire Data with Info*Engine

Instead of using Java Beans, one may choose to use an Info*Engine task to get the data for a Windchill Client Architecture table component.

In order to accomplish this, the following elements must be used:

getIeModel tag	JSP tag Configures the task to use to get the data
ie tag library	The Info*Engine JSP tag library
Param tag	JSP tag from the ie tag library that supplies parameters to the getIeModel tag

Implementing the getIeModel tag

getIeModel is used in a JSP page similar to getModel. The only required configuration for the getIeModel tag (beyond descriptor and var) is an action pointing to the task that should be executed to get the model data. The following code snippet demonstrates the use of the dca-Search

```
<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}"
action="dca-search"/>
```

getIeModel works like the tags included in the Info*Engine JSP library, and can interact with them. Since it is an Info*Engine tag, getIeModel uses the page VDB and is embeddable. In addition, you supply parameters to the tag using Info*Engine's param tag. To use the param tag, you need to include the Info*Engine's tag library in your page, as follows:

```
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>
```

The param tag is nested in your getIeModel tag declaration as follows:

```
<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}" action="dca-search">
<ie:param name="GROUP_OUT" data="groupOutName"/>
</jca:getIeModel>
```

Configuring Task Dispatching

Internally, the getIeModel tag uses Info*Engine's Dispatch-Tasks webject to look up the task implementation for the action task name that is configured in the JSP. getIeModel exposes several parameters which are passed on to Dispatch-Tasks that it uses to choose the right implementation task.

```
<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}" action="dca-search">
<!--These parameters are used by Dispatch-Tasks-->
<ie:param name="GROUP_IN" data="groupInName"/>
<ie:param name="TYPE" data="typeName"/>
<ie:param name="CLIMBER" data="climberName"/>
</jca:getIeModel>
```

In this case, TYPE isn't required since the type of the objects are always persistable.

Supplying Form Data

Any request parameters in your JSP page will be supplied as part of the form data to your task.

- If the page's URL is /somePage.jsp?foo=bar, then in your task implementation, *@FORM[]foo[]* will map to *bar*

Form data can be explicitly configured for a task:

1. Creating the form group in the JSP
   - Use Info*Engine's JSP tag library
2. Specify the name of the group as an attribute of the getIeModel tag

```
<jca:getIeModel
var="tableModel" descriptor="${tableDescriptor}" form="formGroupName" action="dca-search">
```

Supplying Parameters for Configurable Views

getIeModel supports configurable table views.

- Sort and filter criteria from the current table view are passed along to your task using

```
<ie:param name="Parameter" data="ParameterValue"/>
```

Windchill Client Architecture supplies information to your task about the attributes displayed by the requesting component and current table view

- Component uses configurable tables
- Task can then use this information to query for the right resulting data

The additional parameters supplied to your task map to a subset of those accepted by the Query-Objects webject:

- ATTRIBUTE: This contains the list of attributes the component wants to display
- SORTBY: What attributes to sort by • SORTED: The sort direction for each attribute
- WHERE: Filter criteria
- TYPE: The type or types to search for.
- VERSION: Whether LATEST or ALL versions should be returned.
- ITERATION: Whether LATEST or ALL iterations should be returned
- ATTRIBUTE_TYPE_CONTEXT: When multiple types are supplied, the type that should be used as a context to look up attribute definitions
- PAGE_LIMIT: The number of results per page
- PAGE_OFFSET: The first result row to return from the paging session, if any
- PAGING_SESSION_ID: The current paging session id, if any

Exercise 6-2: Acquire Data with Info*Engine

Objectives

- Create an Info*Engine task to search for data.
- Create a Task Delegate for the Info*Engine task.
- Create a JSP page with JCA tags to search for data using the Task Delegate.

Scenario

For an existing second-level tab, the second-level tab should use JCA and getIeModel to search for Windchill PDMLink data.

Detailed Description

From previous exercises, there should be a custom first-level tab and several custom second-level tabs. The focus of this exercise will be to display a table in one of the existing a second-level tabs.

Step 1. Create a demo-search Info*Engine Task that returns three attributes: *name*, *number* and *version*.

a. Create a file *WT_HOME/tasks/com/gsdev/client/demo-SearchLifeCycles.xml*.
b. Add a Query-Objects webject.
c. Hard-code the *INSTANCE* parameter, and search for all wt.lifecycle.LifeCycleTemplates.

Example:

```
<?xml version="1.0" standalone="yes"?>
<%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<ie:webject name="Query-Objects" type="OBJ">
 <ie:param name="ATTRIBUTE" data="name,number,version" delim=","/>
 <ie:param name="INSTANCE"  data="local.ptc.windchill.Windchill"/>
 <ie:param name="WHERE"     data="()"/>
 <ie:param name="TYPE"      data="wt.lifecycle.LifeCycleTemplate"/>
</ie:webject>
```

d. Save the task.
e. Test the task in a web browser.

Step 2. Implement a Task Delegate.

a. Select **Site > Utilities > Task Delegate Administrator**.
b. Enter the LDAP Administrator credentials. Ask the instructor for help, if necessary.
c. Select **Create Delegate**.
d. For **Repository Type**, select **com.ptc.windchill**.
e. For **Name**, enter **`demo-SearchLifeCycles`**.
f. For **Source URL**, enter **`com/gsdev/client/demo-SearchLifeCycles.xml`**.
g. For **Type Identifier**, select **WCTYPE|wt.fc.Persistable**.
h. Select **Create**.

Step 3. Create the JSP page.

a. Use the existing second-level page *WT_HOME/codebase/netmarkets/jsp/newTab/list2.jsp*.
b. Open *list2.jsp*.
c. Add the begin.jspf and end.jspf statements.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

d. Include a taglib directive for JCA components.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

e. Add a describeTable tag with 3 columns: *name*, *number* and *version*.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
</jca:describeTable>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

f. Add a getIeModel tag, with Info*Engine core tags for WHERE and TYPE.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
</jca:describeTable>

<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}"
                    action="demo-SearchLifeCycles"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

g. Add a renderTable tag.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
</jca:describeTable>

<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}"
                action="demo-SearchLifeCycles"/>

<jca:renderTable model="${tableModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

h. Save the JSP file.

Step 4. Test by opening the appropriate second-level tab associated with *list2.jsp*.

a. A table of life cycles should be displayed. This table is retrieving data via a custom Info*Engine Task.

Step 5. Challenge: add a localized resource bundle for the table label (“Search Results”).

Acquire Data with NmObjectUtility

NmObjectUtility provides:

- a way to uniquely identify objects
- a mechanism for handling data that:
  - is not a Persistable object
  - does not map to Persistable in an obvious way

Note:

1. This is particularly useful for external data as external data does not have an *oid* to uniquely distinguish each object in a row of objects.
2. If there is no need to perform an action against a row object, or the row does not have a link relative to the object shown (e.g., open an object's detailed properties, parent/child information, etc.), there is no need to use NmObjectUtility.
3. If the row objects expose an API that returns a supported object type, use targetObject.

Implementing NmObjectUtility

In order to implement NmObjectUtility, use the following:

- Implement the NmObjectUtility interface
- DefaultNmObjectUtility is a default implementation that can be extended
- Via *xconf* configuration files, register any new NmObjectUtility implementations or DefaultNmObjectUtility extensions

Constructing an NmObject for a Persisted Object

Obtain one of the following:

- Persistable
- WTReference
- ObjectIdentifier

Construct an NmOid as follows from the Persistable:

```
public NmObject getNmObject(Object datum, ModelContext mc) {
  // getPersistableFromDatum is the method to find the Persistable object
  Persistable p = getPersistableFromDatum(datum);
  NmOid oid = new NmOid(p);
  NmObject result = new NmObject();
  result.setOid(oid);
  return result;
}
```

Constructing an NmObject for a Non-Persisted Object

1. Create a unique String key for the object
2. Use the key to construct an NmSimpleOid object
3. Construct a new NmObject
4. Set the *id* of the object

```
public NmObject getNmObject(Object datum, ModelContext mc) {
  // getStringKeyFromDatum is a method to find the unique String key
  String key = getStringKeyFromDatum (datum);
  NmSimpleOid oid = new NmSimpleOid();
  oid.setInternalName(key);
  NmObject result = new NmObject();
```

```
    result.setOid(oid);
    return result;
  }
```

Incorporate the NmObjectUtility into the Windchill Client Architecture

Register the NmObjectUtility in the Windchill Client Architecture infrastructure as follows:

- For a table id *customTable*

```
<Service name="com.ptc.core.components.descriptor.NmObjectUtility">
  <Option serviceClass="com.myco.MyCoNmObjectUtility"
    selector="customTable"
    requestor="java.lang.Object"
    cardinality="singleton"/>
</Service>
```

When customTable is used by a descriptor, the elements in the resulting table will be given an *oid* by the NmObjectUtility.

Summary

After completing this module, you should be able to:

- Display a basic table using the Windchill client architecture.
- Configure how data is retrieved for a Windchill client architecture table.
- Use an Info*Engine Task to retrieve data for a Windchill client architecture table component.

Module

7

Attributes

This module goes into detail about handling and displaying attributes, including removing attributes from the OOTB user interface.

Objectives

Upon successful completion of this module, you will be able to:

- Modify the attributes shown in an **Attribute Panel**.
- Create a custom **Attribute Panel**.
- Customize and handle soft attributes using *AllClients.xml*.
- Customize attribute rendering.
- Control the display of attributes in a JSP Page.
- Control how attributes are handled (how data is retrieved for attributes).
- Implement a data utilities.
- Implement a UI component renderer.
- Show Soft Attributes and SCAs.
- Add customized attribute to criteria drop downs.

Lecture Notes

You may use the space below to take your own notes.

Modify the Top Attributes Panel

The Top Attributes Panel is at the top of an **Info** page:

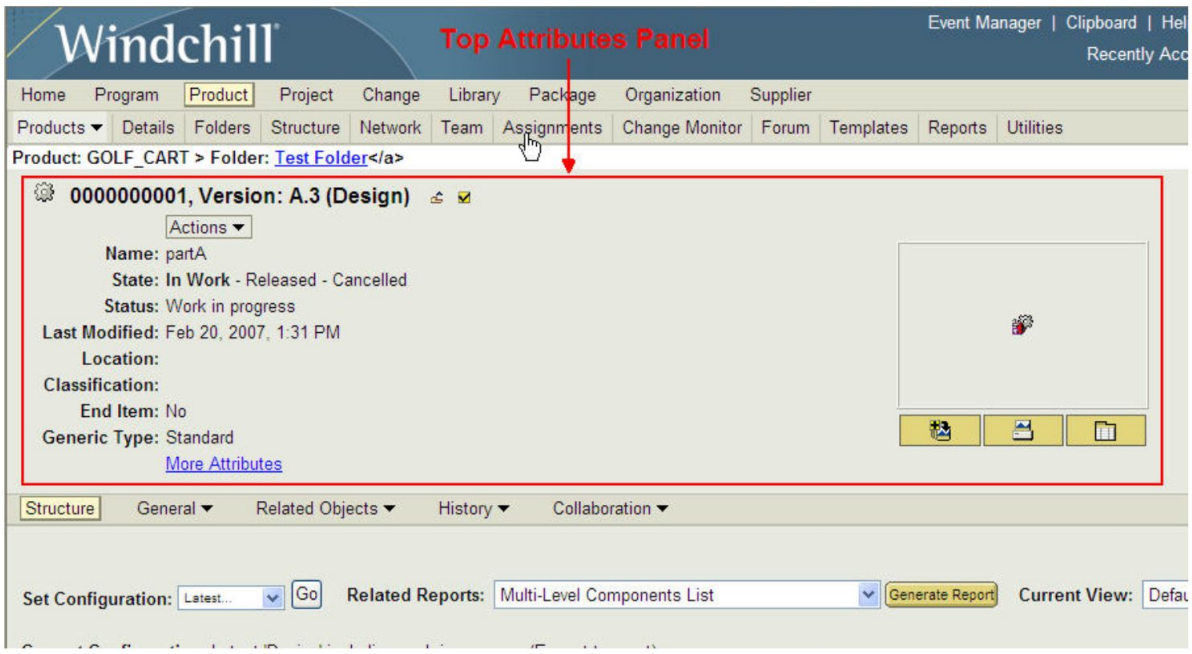


Figure 7-1: Top Attributes Panel

Modifying the Attributes Panel

1. Identify the OOTB **Attribute Panel** for the desired object.
   - Search *WT_HOME/codebase/typedservices.properties* for "wt.services/rsc/default/com.ptc.netmarkets.util.misc.FilePathFactory/Attributes/<OBJECT_TYPE>" (e.g., "wt.services/rsc/default/com.ptc.netmarkets.util.misc.FilePathFactory/Attributes/wt.doc.WTDocument/").
   - If the exact class of object is not found, search for inherited classes.
   - The value found specifies a path, relative to *WT_HOME/codebase*, of the JSP page that defines the OOTB **Attribute Panel**. For *wt.doc.WTDocument*, the file is *netmarkets/jsp/document/attributes.jsp*.
   - Use this page found as a template for a new **Attribute Panel**.

2. Create a custom JSP page to be used to display the desired attributes and insert tags into that JSP page.
   - The custom JSP page will have only the describeAttributesTable tag and list of attribute names as id in describeProperty.
   - Based on the configuration in the describeAttributesTable, optionally, a fragment (*showSoftAttribute.jspf*) component can render the attributes table panel and its content.
   - The describeAttributesTable tag adds a table
   - describeProperty; describeProperty tag adds the attributes:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<jca:describeAttributesTable var="attributesTableDescriptor" mode="VIEW" id="view.setAttribute"
                             label="Attributes" scope="request">
  <jca:describeProperty id="containerName" />
  <jca:describeProperty id="name"/>
  <jca:describeProperty id="number"/>
  <jca:describeProperty id="creatorName" />
  <jca:describeProperty id="currentPrincipal" />
  <jca:describeProperty id="ALL_SOFT_SCHEMA_ATTRIBUTES"/>*
</jca:describeAttributesTable>
```

3. Define custom property entries.
   - Create an *xconf* file to hold custom property settings.
     - Create a file *WT_HOME/codebase/com/gsdev/client/custom-service.properties.xconf*.
     - Add the following to *custom-service.properties* (note the value of *targetFile*):

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE Configuration
  SYSTEM "xconf.dtd">
<Configuration targetFile="codebase/com/gsdev/client/custom-service.properties">

</Configuration>
```

   - Install the custom file (xconfmanager -i custom-service.properties.xconf).
   - Add property overrides to the *custom-typedservices.properties*.

```
<Property default="com/gsdev/client/custom-service.properties"
     name="wt.services.applicationcontext.WTServiceProviderFromProperties.customPropertyFiles"/>

<Resource context="default" name="com.ptc.netmarkets.util.misc.FilePathFactory">
  <Option requestor="wt.doc.WTDocument"
          resource="/netmarkets/jsp/com/gsdev/client/document/attributes.jsp"
          selector="Attributes"/>
  <Option requestor="wt.doc.WTDocument"
          resource="/netmarkets/jsp/com/gsdev/client/document/info.jsp"
          selector="InfoPage"/>
</Resource>
```

     - Resource is the path to a JSP fragment that contains the attribute information
     - Requestor is the type of object
     - Selector is the UI element key

Default Attribute Display

Default attribute display can be overridden using *windchill/codebase/config/logicalrepository/xml/AllClients.xml*

For example review the following code

```
<ElementGroup>
  <LogicContext dataType="ext.part.CustomModeledPart"/>
  <ObjectAttributes id="ObjectAttributes">
    <AttributeEditField id="MBA|MyBooleanModeledAttribute" defaultValueDisplayMode="button" />
```

```
    <AttributeEditField id="MBA|MyStringModeledAttribute" defaultValueDisplayMode="prePopulate"
                        inputFieldType="singleLine"/>
    <AttributeEditField id="IBA|MyStringSoftAttribute" inputFieldType="multiLine"/>
  </ObjectAttributes>
</ElementGroup>
```

Another example

```
<LogicRepository>
  <ElementGroup>
    <LogicContext dataType="wt.folder.SubFolder"/>
    <ObjectAttributes id="ObjectAttributes">
      <AttributeEditField id="IBA|PickOne" selectionListStyle="buttons"/>
      <AttributeEditField id="IBA|Description" defaultValueDisplayMode="prePopulate" />
    </ObjectAttributes>
  </ElementGroup>
</LogicRepository>
```

The syntax for this xml file is governed by:

```
<!DOCTYPE LogicRepository SYSTEM
"/config/logicrepository/dtd/LogicRepository.dtd">
```

which in turn references ***AllClients.dtd***

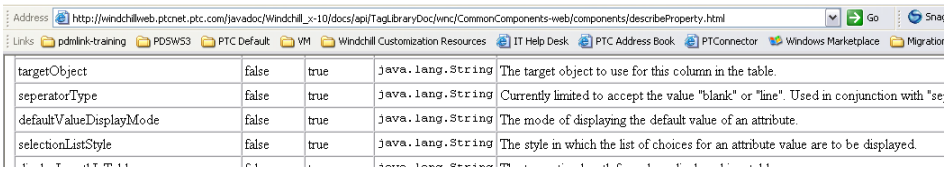
AllClients.dtd looks like this:

```
<!ELEMENT AttributeEditField (DateValueDisplayFormat?) >
<!ATTLIST AttributeEditField id CDATA #REQUIRED>
<!ATTLIST AttributeEditField inputFieldType
   (singleLine | multiLine) "multiLine">
<!ATTLIST AttributeEditField defaultValueDisplayMode
   (none | prePopulate | button)  #IMPLIED>
<!ATTLIST AttributeEditField selectionListStyle
   (buttons | dropdown) #IMPLIED>
<!ATTLIST AttributeEditField stringLengthThreshholdForMultilineInput
   CDATA #IMPLIED>
<!ATTLIST AttributeEditField dateInputFieldType
   (dateOnly | dateAndTime | timeOnly) #IMPLIED>
<!ATTLIST AttributeEditField percent (true | false )
   #IMPLIED>
<!ATTLIST AttributeEditField currency (true | false )
   #IMPLIED>
<!ATTLIST AttributeEditField ignoreUrls (true | false )
    #IMPLIED>
<!ATTLIST AttributeEditField displayLengthInTables CDATA
   #IMPLIED>
<!ATTLIST AttributeEditField displayLengthInInfoPage CDATA
   #IMPLIED>
```

Overriding Default Attribute Display

Attribute display can also be controlled via the JCA components in the JSP itself

- Referencing the TagLib documentation, we see that the same attributes that were used to alter the default display are also available on the *describePropety* tag.



An example of this customization is:

```
<jca:describeAttributesTable var="attributesTableDescriptor"
                             scope="request"
                             id="attributesTableDescriptor"
                             componentType="WIZARD_ATTRIBUTES_TABLE"
                             mode="CREATE"
```

```
                            type="wt.folder.SubFolder"
                            label="${attributesTableHeader}">
      <jca:describeProperty id="name"/>
<jca:describeProperty id="Desc" label="Description"/>
<jca:describeProperty id="PickOne" label="Pick One" selectionListStyle="buttons"/>
          <jca:describeProperty id="folder.id"/>
     </jca:describeAttributesTable>
  <c:remove var="isSharedFolder" scope="session"/>
```

Exercise 7-1: Modify the Top Attributes Panel

Objectives

- Remove an attribute from an **attribute Panel**.
- Add an IBA to an **Attribute Panel**.

Scenario

A customer wants to edit the Attributes Panel for Parts by removing **Modified By** and adding a soft attribute.

Step 1. Create a soft attribute for Parts.

a. Access **Site > Utilities > Type and Attribute Manager**.
b. Access the **Attribute Definition Manager** tab.
c. If an Organizer named “Business” does not exist, create it.
d. Under the **Business** attribute organizer, create a soft attribute of **Type** `String` and named after your first name.
e. The *logical id* should be the same as the attribute name.
f. Select **OK**.

Step 2. Add a soft attribute to Parts.

a. Access **Site > Utilities > Type and Attribute Manager**.
b. Access the **Type Manager** tab.
c. Highlight **Part** and select the edit icon. This function will both check out the Part type as well as enter edit mode.
d. Select the **Template** tab.
e. Highlight **Attribute Root** and select the **Add Attribute** button.
f. In the **Select Attribute** pop up window, expand **Business** and select the attribute just created. Select the **Select** button.
g. Select **OK**.
h. Highlight **Part** and select the check in icon.
i. Close **Type and Attribute Manager**.

Step 3. Create a Part with a value for the added IBA.

Step 4. Find the appropriate definition for an Attribute Panel for Parts.

a. The **Attribute Panel** is nested in the entire **Info** page, so the **Info** page needs to be identified.
b. Search *WT_HOME/codebase/typedservices.properties* for the string “wt.services/rsc/default/com.ptc.netmarkets.util.misc.FilePathFactory/InfoPage/wt.part.WTPart/”.
c. The definition file found should be */netmarkets/jsp/part/info.jsp*.
d. Copy */netmarkets/jsp/part/info.jsp* to */netmarkets/jsp/com/gsdev/client/part/info.jsp*.

Step 5. Edit */netmarkets/jsp/com/gsdev/client/part/info.jsp*.

a. Find the “<jca:describePropertyPanel var="propertyPanelDescriptor">” section. This is the Attribute Panel. Add the following code near the end of the element.

Example:

```
<jca:describeProperty id="genericType" />
<jca:describeProperty id="templateOfGeneric" />

 <jca:describeProperty id="IBA|com.gs.Cost"/>
 <jca:describeProperty id="Cost"/>

</jca:describePropertyPanel>
```

b. Save the file.

Step 6. Register the new **Info** page for wt.part.WTPart.

a. Although only the **Attribute Panel** was changed, the changes were made to the **Info** page, so a new **Info** page must be registered.
b. Create a file *WT_HOME/codebase/com/gsdev/client/custom-service.properties.xconf*.

c. Add the following default content for new *xconf* files:

Example:

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE Configuration
  SYSTEM "xconf.dtd">
<Configuration targetFile="codebase/com/gsdev/client/custom-service.properties">

</Configuration>
```

d. Add the following property to recognize this file as containing service properties:

Example:

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE Configuration
  SYSTEM "xconf.dtd">
<Configuration targetFile="codebase/com/gsdev/client/custom-service.properties">

<Property default="com/gsdev/client/custom-service.properties"
     name="wt.services.applicationcontext.WTServiceProviderFromProperties.customPropertyFiles"/>

</Configuration>
```

e. Add a reference to the new Info page:

Example:

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE Configuration
  SYSTEM "xconf.dtd">
<Configuration targetFile="codebase/com/gsdev/client/custom-service.properties">

<Property default="com/gsdev/client/custom-service.properties"
     name="wt.services.applicationcontext.WTServiceProviderFromProperties.customPropertyFiles"/>

<Resource context="default" name="com.ptc.netmarkets.util.misc.FilePathFactory">
  <Option requestor="wt.part.WTPart"
          resource="/netmarkets/jsp/com/gsdev/client/part/info.jsp" selector="InfoPage"/>
</Resource>

</Configuration>
```

f. Install the new *xconf* file by executing xconfmanager -i codebase/com/gsdev/client/custom-service.properties

g. Propagate the changes.

h. Restart the Method Server and Tomcat.

Step 7. Test.

a. Open the properties page for a Part.

Step 8. Remove **Modified By**.

a. Edit */netmarkets/jsp/com/gsdev/client/part/info.jsp*.

b. Find the tag “<jca:describePropertyPanel var="propertyPanelDescriptor">”.

c. Comment out “lifeCycleState”:

Example:

```
<jca:describePropertyPanel var="propertyPanelDescriptor">
   <jca:describeProperty id="name" />
   <jca:describeProperty id="supplierId" />
   <!jca:describeProperty id="lifeCycleState" />
   <jca:describeProperty id="checkoutInfo" />
```

d. Note the format for the comment line. The comment technique (e.g., “!” or “<!– –>”) can vary. This is a custom file, and the original file is intact, so technically this line could be deleted entirely.

Step 9. Test.

Attribute Handling and Data Utilities

Attributes of Windchill objects are rendered in two modes:

1. an “input” mode for create and edit functions
2. a “view” mode for non-editable situations

- A data utility is used to create a GUI component for an attribute.
- The data used to create a GUI component is a combination of the attribute and constraint definitions
- The renderer for a component generates HTML for displaying the component
- The GUI components and their renderers can also be used to represent attributes that are not Windchill attributes like drawing a simple check-box on a page to capture some input from the user
- GUI components can be simple components such as a text-box or composite GUI components built using the simple components
- Components generated by this framework also have some built-in client-side validations, based on the constraint definitions and the configurations
- Windchill JCA framework supported types are one of three types:
    1. Attributes whose type is a valid Windchill soft attribute type
    2. Attributes whose type is a native Java type
    3. Special Windchill attributes

Supported Attribute Types

Valid Soft Attribute	Native Java Types	Special Windchill Attributes
Boolean	primitive boolean java.lang.Boolean	Name
Integer	primitive byte short, integer long java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long	Number
Floating point number	primitive float, double java.lang.Float java.lang.Double	Location
Floating point number with Units		Revision label
String	java.lang.String	Principal
Date & Time	java.sql.Timestamp	Status (work in progress state)
wt.fc.EnumeratedType		Lifecycle state
URL		Lifecycle template

Valid Soft Attribute	Native Java Types	Special Windchill Attributes
WTOrgRef		Team template
Classification Node		

Modifying Attribute Handling

Modifying the way Windchill handles attribute can involve one or more of the following updates:

- Windchill Info Modeler to define the modeled attributes
- **Attribute Manager and Type Manager** to define the instance based attributes
- **Object Initialization Rules Administrator**
- **Preference Manager**
- *AllClients.xml* to customize some of the attribute rendering behavior
- JSP files to customize the attribute rendering behavior
- Resource bundles
- Data utility delegate to implements the methods to create concrete GUI component objects for a given attribute
- *service.properties* to configuration point for mapping components to data utilities
- An existing or new GUI component
- GUI component renderer to implement the rendering of a GUI component
- *components.service.properties* to map GUI components to renderers
- Form processing delegates to process form data of GUI components

OOTB Attribute Handling Behavior

Attribute display can be modified without customization, as follows:

- Add/remove attributes from a properties panel or table
- Change the default attribute behavior based on Windchill preferences
- Change the default attribute behavior based on the contents of *AllClients.xml*

Note: Refer to p11-16 of the *WCCustomizersGuide.pdf*.

Relationship between GUI Components, Renderers and Utilities

GUI Components are the elements displayed on pages.

How components are rendered on a page:

- Framework requests an attribute and then in turn requests the Component Model
- Framework looks up the data utility for the attribute by interrogating the properties
- Data Utility for the attribute is invoked to get the data
- Data Utility instantiates the component sets it's data to the values returned in the previous step
- GUI Component uses the draw routine to instantiate it's renderer and then calls the renderer's draw method

Details on Rendering Components

- The Data Utility
  - Gets the data using either getDataValue or in special cases setModelData for returning data for multiple components in one call to the Method Server
  - Instantiates the GUI Component
  - Calls the setValue method (or similar) of the GUI Component to set that data that the component will draw
- GUI Component has a draw() method which instantiates and calls the Renderer

- The Renderer in turn has it's draw method which completes the drawing of visual components

When to Implement a New Data Utility

Implement a new data utility when:

- Attribute type is not a Windchill JCA framework supported type
- A different display is required (e.g., E-mail addresses — which are normally string text — are displayed as hyper links)
- Data needs to come from a different source
- Need to post-process the raw data returned by the core APIs before displaying (e.g., roll up cost or reformat dates/times)
  - e.g., generate a hidden field to store some data that may be required in conjunction with the data from the GUI component in order to derive the actual value for an attribute
  - requires a form processing delegate

Note:

- Data utilities can augment the data returned by a core API with additional information.
- The additional information can be returned by an additional query, a call to a service, or anything else that can be retrieved or generated at via Java code.

OOTB Rendering Example

Consider the **Folder List** page. When part objects are displayed, the name column is rendered as a hyper link.

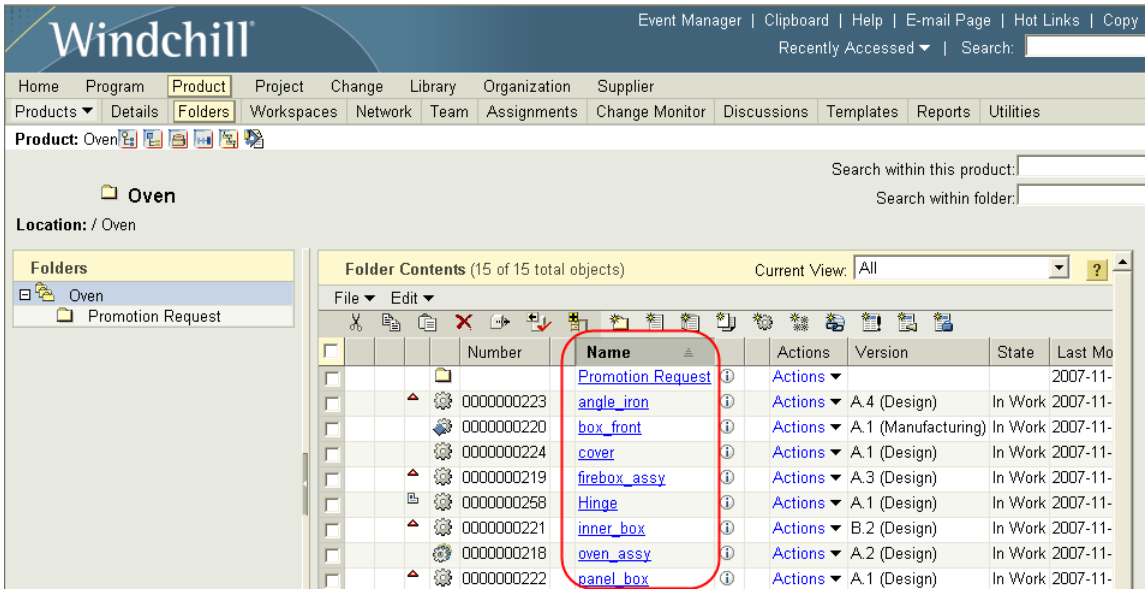
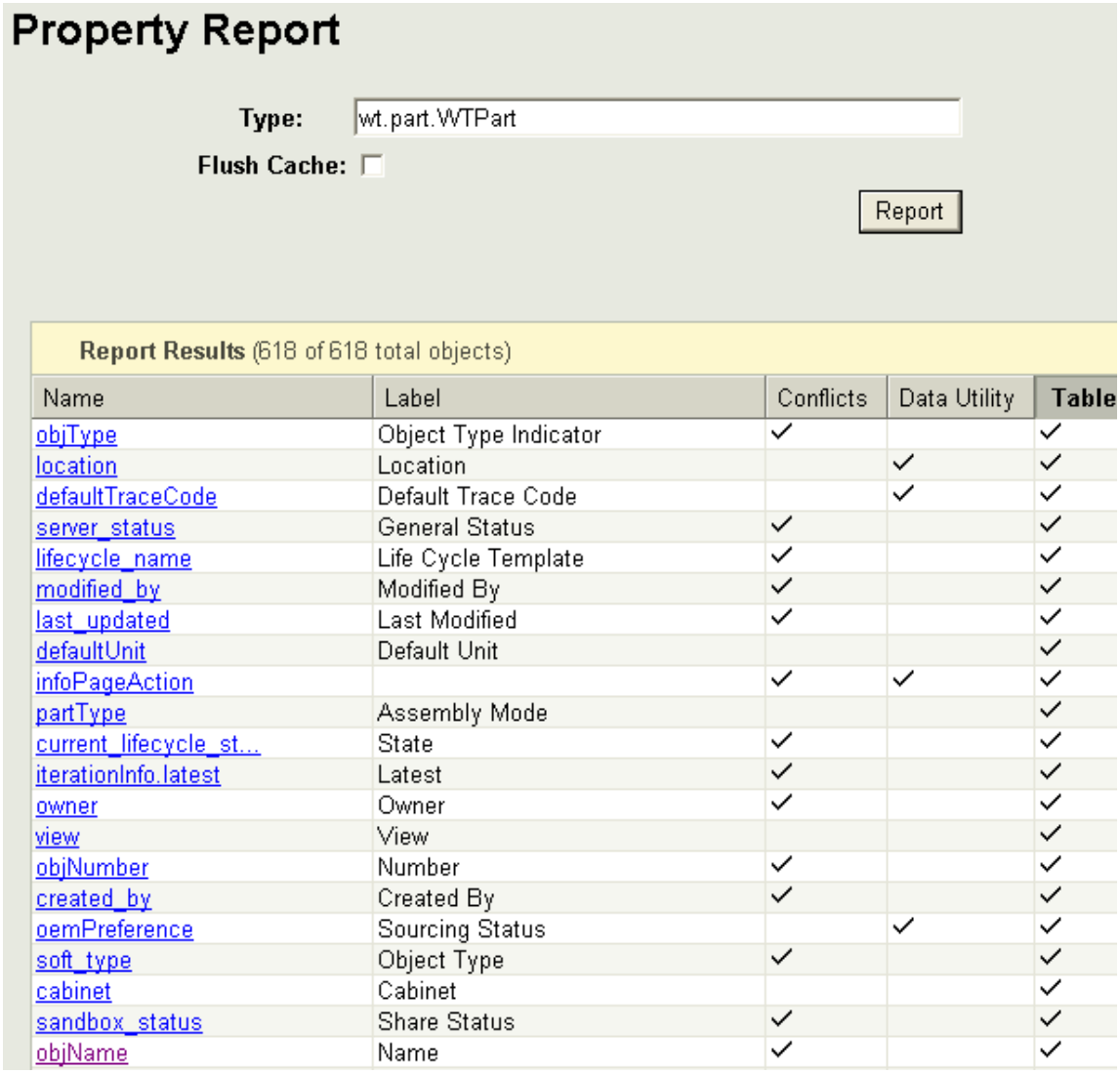


Figure 7-2: Folder List Page

The Data Utility can be found by looking at the **Property Report**



Property Report

Type: wt.part.WTPart

Flush Cache:

Report

Report Results (618 of 618 total objects)

Name	Label	Conflicts	Data Utility	Table
objType	Object Type Indicator	✓		✓
location	Location		✓	✓
defaultTraceCode	Default Trace Code		✓	✓
server_status	General Status	✓		✓
lifecycle_name	Life Cycle Template	✓		✓
modified_by	Modified By	✓		✓
last_updated	Last Modified	✓		✓
defaultUnit	Default Unit			✓
infoPageAction		✓	✓	✓
partType	Assembly Mode			✓
current_lifecycle_st...	State	✓		✓
iterationInfo.latest	Latest	✓		✓
owner	Owner	✓		✓
view	View			✓
objNumber	Number	✓		✓
created_by	Created By	✓		✓
oemPreference	Sourcing Status		✓	✓
soft_type	Object Type	✓		✓
cabinet	Cabinet			✓
sandbox_status	Share Status	✓		✓
objName	Name	✓		✓

Figure 7-3: propertyReport on WTPart

AttributesMap.xml connects it to a logical attribute of “name”.

xref: /x-12/PdmLink/PdmLink/src/com/ptc/windchill/pdmlink/part/attributeMap.properties

Home | History | Annotate | Download | Search only in part

```
1  objName=name
2  objNumber=number
3  description=description
4  current_lifecycle_state=state
5  iteration=iterationDisplayIdentifier
6  context=containerName
7  objOrganizationId=organizationUniqueIdentifier
8  referenceDesignator=referenceDesignator
9  MBA|lineNumber.value=lineNumber
10 MBA|quantity.unit=quantity
11 MBA|findNumber=findNumber
```

Figure 7-4: AttributeMap for PDMLink Parts

The DataUtility for this attribute is defined in the following file on line 138 of *D:\Ptc\Windchill\codebase\com\ptc\core\components\components.dataUtilities.properties*

```
wt.services/svc/default/com.ptc.core.components.descriptor.DataUtility/name/java.lang.Object/0=
                    com.ptc.core.components.factory.dataUtilities.NameDataUtility/duplicate
```

NameDataUtility is responsible for creating the HTML element that is returned to the web page. The structure of the code is slightly different from the recommended structure.

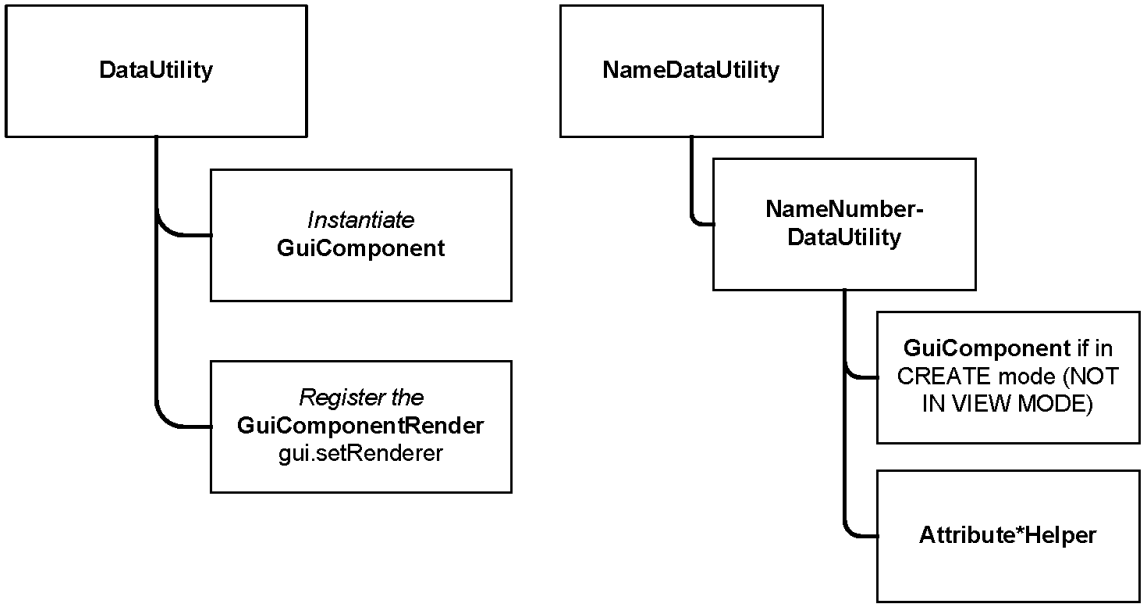


Figure 7-5: Comparison to Recommended Structure

The recommendation is closest to being followed in create mode

Figure 7-6:

How to Implement a Data Utility

Implement a Data Utility as follows:

1. Extend AbstractDataUtility
2. Update the class to have a *public*, and have a *public constructor* with no arguments

3. Implement the getDataValue or setModelData for multiple objects
   - Gets the value
   - Returns an instance of the GUI Component

```
Object getDataValue(
    String component_id,
    Object datum,
    ModelContext mc)
   throws WTException {

...

 String rawVal = myObject.get(component_id);
 String displayVal = rawVal + "My extra text";
 TextDisplayComponent gui =
   new TextDisplayComponent(…);
 gui.setValue(displayVal);

...

return gui;
```

4. Register the Data Utility (here are two examples):

```
<Service name="com.ptc.core.components.descriptor.DataUtility">
<Option serviceClass="com.ptc.core.components.factory.dataUtilities.ProjectDataUtility"
        requestor="java.lang.Object" selector="projectHealthStatus" cardinality="singleton"/>
</Service>

<Service name="com.ptc.core.components.descriptor.DataUtility">
<Option serviceClass="com.ptc.core.components.factory.dataUtilities.FolderedDataUtility"
        requestor="java.lang.Object" selector="location" cardinality="duplicate"/>
</Service>
```

Note:

1. Any java.lang.Object that is looking for a location attribute will use the FolderDataUtility.
2. Currently, only java.lang.Object is supported. If the DataUtility should only act against objects of a certain type, the DataUtility will need the incoming object based on type.

Exercise 7-2: Implement a Data Utility

Objectives

- Create a custom Data Utility.
- Register the Data Utility.

Scenario

The customer has requested that a suffix be added to the display of every Part name.

Step 1. Create a custom DataUtility.

a. Create a custom class *WT_HOME/codebase/com/gsdev/client/CustomNameDataUtility.java* which extends com.ptc.core.components.factory.dataUtilities.NameNumberDataUtility:

Example:

```
package com.gsdev.client;

import com.ptc.core.components.factory.dataUtilities.NameNumberDataUtility;

public class CustomNameDataUtility extends NameNumberDataUtility {

 public CustomNameDataUtility() {
  // TODO Auto-generated constructor stub
 }

}
```

Step 2. Add import statements.

Example:

```
import wt.util.WTException;
import com.ptc.core.components.descriptor.ModelContext;
import com.ptc.core.components.factory.dataUtilities.AttributeDataUtilityHelper;
import com.ptc.core.components.factory.dataUtilities.NameNumberDataUtility;
import com.ptc.core.components.factory.dataUtilities.StringDataUtility;
import com.ptc.core.components.rendering.GuiComponent;
import com.ptc.core.components.rendering.guicomponents.GuiComponentUtil;
import com.ptc.core.components.rendering.guicomponents.StringInputComponent;
import com.ptc.core.meta.container.common.AttributeTypeSummary;
import com.ptc.core.meta.container.common.State;
import com.ptc.core.meta.type.common.TypeInstance;
import com.ptc.core.ui.resources.ComponentMode;
```

Step 3. Override getDataValue().

a. Edit *WT_HOME/codebase/com/gsdev/client/CustomNameDataUtility.java* and add the method getDataValue():

Example:

```
public Object getDataValue(String component_id, Object obj, ModelContext mc)
                throws WTException {
 Object obj1 = null;
 if(mc.getDescriptorMode()==ComponentMode.CREATE||mc.getDescriptorMode()==ComponentMode.EDI
  GuiComponent guicomponent = createInputComponent(component_id, obj, mc);
  obj1 = guicomponent;
 } else {
  String s1 = AttributeDataUtilityHelper.getValue(mc, "name");
  if(s1 == null) {
  // The following 2 lines are one line
  System.out.println(
(new StringBuilder()).append("Can't find comp value: ").append(component_id).toString());
  }
  obj1 = super.createDisplayComponent(component_id, obj, mc, s1 + "extra text");
  System.out.println("Inside CustomNameDataUtility");
 }
 return obj1;
}
```

Step 4. Override createInputComponent().
Example:

```
protected GuiComponent createInputComponent(String s, Object obj, ModelContext mc)
          throws WTException {
 int i = GuiComponentUtil.MAX_STRING_LENGTH;
 AttributeTypeSummary attributetypesummary = mc.getATS();
 if(attributetypesummary != null && attributetypesummary.getMaxStringLength() > 0) {
  i = attributetypesummary.getMaxStringLength();
 }
 int j = StringDataUtility.getMaxStringLength(mc);
 if(j > 0) {
  i = j;
 }
 if(i > 150) {
  i = 150;
 }
 StringInputComponent stringinputcomponent = new StringInputComponent(s, i, i + 1);
 String s1 = mc.getDescriptor().getModelAttributeString();
 if(s1 == null || s1.length() == 0) {
  s1 = s;
 }
 stringinputcomponent.setColumnName(AttributeDataUtilityHelper.getColumnName(s1, obj, mc));
 stringinputcomponent.setRequired(AttributeDataUtilityHelper.isInputRequired(mc));
 State state = null;
 TypeInstance typeinstance = mc.getTypeInstance();
 if(typeinstance != null) {
  state = AttributeDataUtilityHelper.getFieldState(mc);
 }
 if(mc.getDescriptorMode() == ComponentMode.EDIT || state.compareTo(State.DEFAULT) == 0) {
  Object obj1 = mc.getRawValue();
  // The following 2 lines are one line
  com.ptc.core.meta.common.AttributeIdentifier aattributeidentifier[] = typeinstance == null
     ? null : typeinstance.getAttributeIdentifiers(s);
  // The following 2 lines are one line
  obj1 = obj1 != null || aattributeidentifier == null || aattributeidentifier.length <= 0
     ? obj1 : typeinstance.get(aattributeidentifier[0]);
  if(obj1 != null && (obj1 instanceof String)) {
   stringinputcomponent.setValue((String)obj1);
  }
 }
 // The following 2 lines are one line
 stringinputcomponent.setId((new StringBuilder()).append(AttributeDataUtilityHelper.getHtmlId(
       mc.getDescriptor()))).append(System.currentTimeMillis()).toString());
 return stringinputcomponent;
}
```

Step 5. Save and compile the class.

Step 6. Register the Data Utility.

a. Data utilities are, currently, not type-specific. If class-specific or type-specific behavior is required, it must be coded into the Data Utility class.

b. Edit *WT_HOME/codebase/com/gsdev/client/custom-service.properties.xconf*.

c. Add the following:

Example:

```
<Service context="default" name="com.ptc.core.components.descriptor.DataUtility">
  <Option requestor="java.lang.Object" serviceClass="com.gsdev.client.CustomNameDataUtility"
          selector="name" cardinality="duplicate"/>
</Service>
```

d. Propagate the changes.

e. Restart the Method Server and Tomcat.

Exercise 7-3: Add Non-Persisted Attribute to the GUI

Objectives

- Add a custom attribute.

Scenario

The customer needs to display an attribute on a Property Panel which is an amalgamation of existing Windchill attributes. This attribute is the number of days since an object was created.

Step 1. The attribute will be added to the custom life cycle list being retrieved by Info*Engine.

a. Modify the *demo-SearchLifeCycles* Task to also return the attribute *thePersistInfo.createStamp*.

b. Test the Task in a browser.

Step 2. Modify the JSP associated with the custom table retrieving data via Info*Engine.

a. Edit *WT_HOME/codebase/newTab/list2.jsp*.

b. To the *describeTable* tag, add a *describeColumn* tags to retrieve **`thePersistInfo.createStamp`**.

Example:

```
<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
   <jca:describeColumn id="thePersistInfo.createStamp"/>
</jca:describeTable>
```

c. To the *describeTable* tag, add a *describeColumn* tags to retrieve **`daysSinceCreation`**. The tag needs to know the source of this attribute. Add a **`need`** tag set equal to **`thePersistInfo.createStamp`**.

Example:

```
<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
   <jca:describeColumn id="thePersistInfo.createStamp"/>
   <jca:describeColumn id="daysSinceCreation" need="thePersistInfo.createStamp"/>
</jca:describeTable>
```

Step 3. Register a DataUtility for *daysSinceCreation*.

a. Edit *WT_HOME/codebase/com/gsdev/client/custom-service.properties.xconf.*

b. Add the following:

Example:

```
<Service context="default" name="com.ptc.core.components.descriptor.DataUtility">
  <Option requestor="java.lang.Object"
          serviceClass="com.gsdev.client.CustomDataUtility"
          selector="daysSinceCreation" cardinality="duplicate"/>
</Service>
```

c. Save the file.

d. Propagate the changes.

e. Restart Tomcat and the Method Server to read the property file changes.

Step 4. Create the custom Data Utility.

a. Create the class com.gsdev.client.CustomDataUtility, extending com.ptc.core.components.factory.AbstractDataUtility.

b. Add a package statement.

c. Add import statements.

Example:

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import com.ptc.core.components.factory.AbstractDataUtility;
import com.ptc.core.components.factory.dataUtilities.AttributeDataUtilityHelper;
import com.ptc.core.components.descriptor.ModelContext;
```

d. Override getDataValue().

Example:

```
 public Object getDataValue( String component_id, Object datum, ModelContext mc) throws wt.util.WT

   String createDate = "default value";
createDate = AttributeDataUtilityHelper.getValue(mc, "daysSinceCreation");
System.out.println( "Original value received: " + createDate );
SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss Z");

Date convertedDate = null;
try {
 convertedDate = format.parse( createDate );
} catch (ParseException pe) {
 // Do nothing
}
Date now = new Date();
System.out.println( "Now: " + now );
int days = (int) ((now.getTime() - convertedDate.getTime())/(1000*60*60*24));
  return Integer.toString( days ) ;
 }
```

e. Save the file.

f. Compile.

Step 5. Test it.

a. Open **Client > Basic I*E Table**.

b. Create a life cycle, then return to **Basic I*E Table** and refresh the page.

Step 6. The code added tocom.gsdev.client.CustomDataUtility.getDataValue() returned a String and not a GUIComponentobject. Why did this work?

Step 7. Challenge: add resource bundles for all labels, table and column.

GUI Component Renderer

How to Implement a GUI Component Renderer

1. Create a class that implements Renderer.
2. Implement the draw method using the template code provided in the *Customizer's Guide*.

```
public abstract class MyDateRenderer implements Renderer {

 public void draw(
    T o, Writer out,
    RenderingContext renderContext)
     throws RenderingException {
 …
 /*  render individual components
  *  similar to the old template processors
  */

 }
}
```

3. Register your custom renderer.
   - The renderer is registered in the data utility

```
DateInputComponent gui = new DateInputComponent();
...
gui.setRenderer(new MyDateRenderer());

...
```

How to Implement a New GUI Component

1. Implement the GUIComponent class.
2. Create a new renderer and data utility.
3. Call the draw method of the renderer.
4. Use the template provide in the*Customizer's Guide*.

```
public class MyTextBox implements GuiComponent {
private String value;
private intwidth = 10;
private int maxCharLimit = 20;
protected Renderer renderer;

  public void draw(
  Writer out,
  RenderingContext renderContext) throws RenderingException {
    if (renderer != null) {
      renderer.draw((Object)this,out,renderContext);
    }

  }
}
```

When to Create a GUI Component Renderer

Configuration on the standard GUI components will not be enough to get the UI behavior desired –>Implement a custom Renderer for the GUI component.

When to Create a GUI Component

When configuration on the standard GUI components and/or overriding the Renderer will not be enough to get the UI behavior required –> Implement a new GUI Component.

Note: A new GUI component should be considered when displaying attributes that are inherently different from the ones supported out of the box. Good examples of this are:

- Currency
- Cost
- Ratios
- Fractions
- Mathematical statistics and calculations

Exercise 7-4: Implement Custom Rendering

Objectives

- Create a custom Renderer.

Scenario

The customer wants to display the user that created the life cycle in uppercase.

Detailed Description

It would be possible to implement a new Data Utility and Renderer against existing attributes. However, recall that Data Utilities are not type-aware. Any logic written would affect any attribute by the same name. It would be easier and more prudent to create a new attribute (upperCaseUserName) entirely.

Step 1. Add the *creator* attribute to the *demo-SearchLifeCycles* Task.

Step 2. Modify the JSP associated with the custom table retrieving data via Info*Engine.

a. Add *upperCaseUserName* attribute which will depend on a data source named *creator* (retrieved from the Task).

Step 3. Register a DataUtility for *upperCaseUserName*.

a. Name the Data Utility com.gsdev.client.CustomUserDataUtility.

Step 4. Create the custom Data Utility.

a. The input will be *creator*; format it to be upper case.

Step 5. Test it.

Default Attribute Display

The default attribute display for PSE and the HTML UI can be overridden by editing *WT_HOME/codebase/config/logicrepository/xml/AllClients.xml*.

The file is empty by default.

For example, review the following code which could be added for a custom Part extension:

```
<ElementGroup>
  <LogicContext dataType="ext.part.CustomModeledPart"/>
  <ObjectAttributes id="ObjectAttributes">
     <AttributeEditField
       id="MBA|MyBooleanModeledAttribute"
       defaultValueDisplayMode="button" />
     <AttributeEditField
       id="MBA|MyStringModeledAttribute"
       defaultValueDisplayMode="prePopulate"
       inputFieldType="singleLine"/>
     <AttributeEditField
       id="IBA|MyStringSoftAttribute"
       inputFieldType="multiLine"/>
  </ObjectAttributes>
</ElementGroup>
```

The syntax for this xml file is governed by: *WT_HOME/codebase/config/logicrepository/dtd/LogicRepository.dtd* which, in turn, references *WT_HOME/codebase/config/logicrepository/dtd/AllClients.dtd*.

AllClients.dtd sample:

```
<!ELEMENT AttributeEditField (DateValueDisplayFormat?) >
<!ATTLIST AttributeEditField id CDATA #REQUIRED>
<!ATTLIST AttributeEditField inputFieldType
   (singleLine | multiLine) "multiLine">
<!ATTLIST AttributeEditField defaultValueDisplayMode
   (none | prePopulate | button)  #IMPLIED>
<!ATTLIST AttributeEditField selectionListStyle
   (buttons | dropdown) #IMPLIED>
<!ATTLIST AttributeEditField stringLengthThreshholdForMultilineInput
   CDATA #IMPLIED>
<!ATTLIST AttributeEditField dateInputFieldType
   (dateOnly | dateAndTime | timeOnly) #IMPLIED>
<!ATTLIST AttributeEditField percent (true | false )
   #IMPLIED>
<!ATTLIST AttributeEditField currency (true | false )
   #IMPLIED>
<!ATTLIST AttributeEditField ignoreUrls (true | false )
    #IMPLIED>
<!ATTLIST AttributeEditField displayLengthInTables CDATA
   #IMPLIED>
<!ATTLIST AttributeEditField displayLengthInInfoPage CDATA
   #IMPLIED>
```

AllClients Options

Other options that can be used in *AllClients.xml* to control attribute behavior:

Option	Options	Default
currency	truefalse	
dateInputFieldType	dateOnly | dateAndTime | timeOnly	
defaultValueDisplayMode	noneprePopulatebutton	
displayLengthInInfoPage		
displayLengthInTables		
ignoreUrls	truefalse	
inputFieldType	singleLinemultiLine	multiLine
percent	truefalse	
selectionListStyle	buttonsdropdown	
showDefaultValue	truefalse	
stringLengthThreshholdForMultilineInput		

Note:

- Edits to *WT_HOME/codebase/config/xml/AllClients.xml* are reloaded by restarting the Method Server.
- This command validates the files before doing the reload. If it finds an error in any of the files, the reload is aborted.

Overriding Default Attribute Display

Attribute display can also be controlled via the JCA components in the JSP itself.

- Referencing the TagLib documentation, the *describePropety* tag can modify elements of attribute display as well as *AllClients.xml*.

An example of this customization is:

```
<jca:describeAttributesTable var="attributesTableDescriptor"
                             scope="request"
                             id="attributesTableDescriptor"
                             componentType="WIZARD_ATTRIBUTES_TABLE"
                             mode="CREATE"
                             type="wt.folder.SubFolder"
                             label="${attributesTableHeader}">
      <jca:describeProperty id="name"/>
  <jca:describeProperty id="Desc" label="Description"/>
  <jca:describeProperty id="PickOne" label="Pick One" selectionListStyle="buttons"/>
           <jca:describeProperty id="folder.id"/>
     </jca:describeAttributesTable>
   <c:remove var="isSharedFolder" scope="session"/>
```

To add/remove columns in the table, insert/remove the describeColumn tag from the describeTable tag.

- The column order is the order in which the columns are defined using the describeColumn tag in describeTable tag in a JSP file.
- To hide a column, use hidden attribute in the describeColumn tag. Sorting on this column is still possible even though hidden.

Each column has a guiComponent corresponding to the value to be shown in the column.

- Each guiComponent has a reference to renderer object that is responsible for generating the HTML
- To change the default renderer we need to change the guiComponent or the renderer reference for that guiComponent. GuiComponent are created using the dataUtility.

.

Other Table Components

Setting sortable columns:

```
<describeColumn id="name" sortable="true" defaultSort="true" />
```

To add a Toolbar, add the following element in the table descriptor

```
<setComponentProperty key="actionModel" value="customToolbar"/>
```

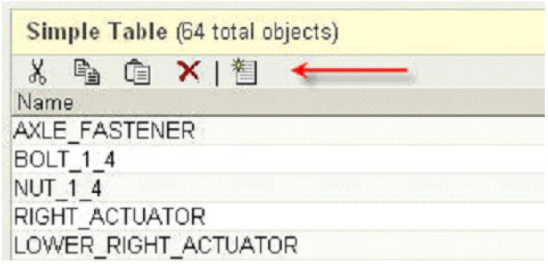


Figure 7-7: Toolbar

Adding row-level actions is similar to the Toolbar for the table descriptor. Add the following to the column descriptor

```
<setComponentProperty
   key="actionModel"
   value="<string mapping in actions-model.xml>" />
```

To add a Menu Bar use the following table descriptor entry:

```
<describeTable
   var="tableDescriptor"
   type="wt.part.WTPart"
   id="tableId"
   menubarName = "customMenubar"
   label="${tableName}"
   configurable="true">
```

Notice that the menu bar must be defined in **action-models.xml* as follows

```
<model name="customMenubar">
 <submodel name="fileMenu">
 <submodel name="editMenu">
</model>
<model name="fileMenu">
 <action name="list_cut" type="object"/>
 <action name="list_copy" type="object"/>
 <action name="fbpast" type="object"/>
 <action name="list_delete" type="object"/>
</model>
<model name="editMenu">
 <action name="create" type="document"/>
</model>
```

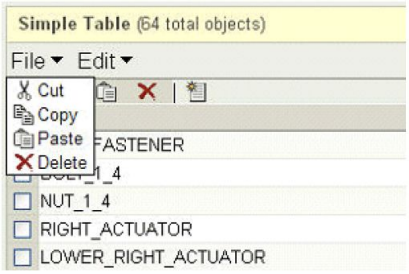


Figure 7-8: Menu Bar

Enabling row selection in the table descriptor as follows

```
<setComponentProperty key="selectable" value="true"/>
```

Enable Single/Multiple Row selection as follows in the renderTable tag:

```
<renderTable
  model="${tableModel}"
  showPagingLinks
  singleSelect="true"/>
```

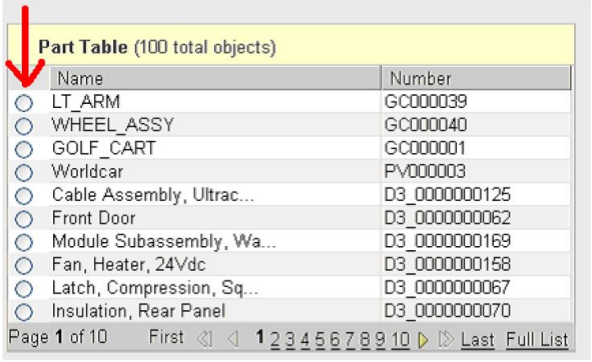


Figure 7-9: Single Select

Adding pagination and page limit

```
<renderTable
  model="${tableModel}"
  showPagingLinks ="true"
  pageLimit="10" />
```

Figure 7-10: Pagination of Results

Specify Different Data Utility for columns

```
<describeColumn
  id="iconType"
  label="${columnName}"
dataUtilityId="typePicker.iconType" />
```

Adding hyperlink to a column can be accomplished by entering the following

```
<describeColumn id="name" isInfoPageLink="true" />
```

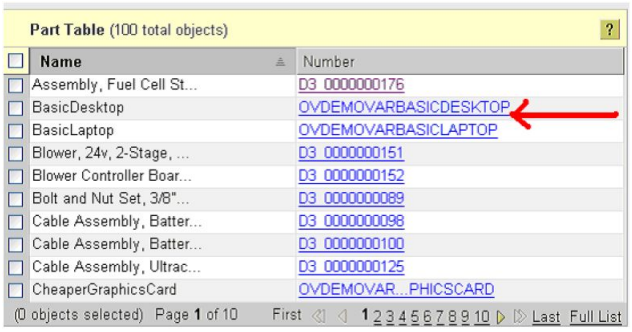


Figure 7-11:

Non-Persistent Attributes

At times, it might be necessary to display attributes that are not stored in Windchill.

Examples of this may be:

- Cost from an ERP system
- Roll-up of quantitative attributes in an assembly: cost, weight, etc.
- Derived properties: area, volume, etc.

In the case where the UI displays an attribute which does not exist in the system (either modelled or IBA), attributes must be defined in a configuration file and the configuration file must be registered.

The process has 2 steps:

1. Create a custom *AvailableAttributes.xml* file called *custom-AvailableAttributes.xml* in the *WT_HOME/codebase/config/attributes* directory.
2. Register the file by adding *congif/attributes/custom-AvailableAttributes.xml* to the multi-valued property *com.ptc.core.htmlcomp.createtableview.AvailableAttributesDigester.fileLocation*.

Exercise 7-5: Modifying Default Soft Attribute Display

Objectives

- Modify the display of Document attributes.

Scenario

The customer wants to modify a *cost* attribute to be displayed as currency.

Detailed Description

Note: This exercise does not work in 9.0 F000.

Step 1. Create *Cost* and *PercentComplete* attributes.

a. Both are floating numbers.

Step 2. Add *Cost* and *PercentComplete* to Document.

Step 3. Modify *AllClients.xml*.

a. Edit *WT_HOME/codebase/config/logicrepository/xml/AllClients.xml*.

b. Between the **LogicRepository** tags, add

Example:

```
<ElementGroup>
 <LogicContext dataType="wt.doc.WTDocument"/>
 <ObjectAttributes id="ObjectAttributes">
  <AttributeEditField id="IBA|Cost" currency="true" />
  <AttributeEditField id="IBA|PercentComplete" percent="true"/>
 </ObjectAttributes>
</ElementGroup>
```

c. Save the file.

d. Re the configuration.

Step 4. Restart Tomcat and the Method Server.

Step 5. Test.

a. Create a Document with *Cost* and *PercentComplete* values.

b. View the properties page for the Document.

Summary

After completing this module, you should be able to:

- Modify the attributes shown in an **Attribute Panel**.
- Create a custom **Attribute Panel**.
- Customize and handle soft attributes using *AllClients.xml*.
- Customize attribute rendering.
- Control the display of attributes in a JSP Page.
- Control how attributes are handled (how data is retrieved for attributes).
- Implement a data utilities.
- Implement a UI component renderer.
- Show Soft Attributes and SCAs.
- Add customized attribute to criteria drop downs.

Module

8

Trees and Tables

This module goes into detail about the “tree” and “table” components.

Objectives

Upon successful completion of this module, you will be able to:

- Add a custom table to the user interface.
- Add a custom tree to the user interface.
- Implement a client architecture tree for the display of business objects.
- Control how a table will appear using configuration properties.

Lecture Notes

You may use the space below to take your own notes.

Constructing a Table

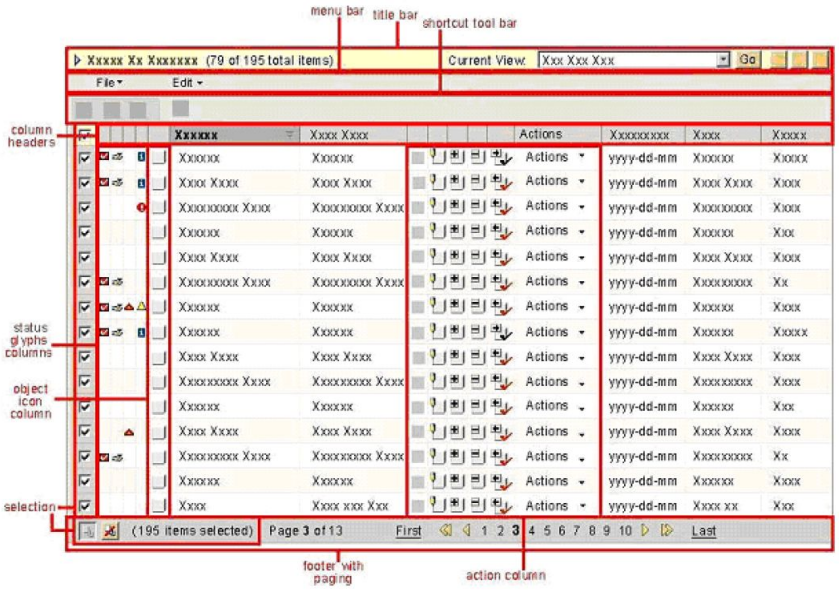


Figure 8-1: Table Components

Taglibs

“c” is the core JavaServer Pages Standard Tag Library (JSTL):

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

“jca” is to the JCA components specific to Windchill PDMLink:

```
<%@ taglib
uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
```

“wc” is the core Windchill library:

```
<%@ taglib uri="http://www.ptc.com/windchill/taglib/core" prefix="wc"%>
```

“cmb” is for the “Carambola” example tags:

```
<%@ taglib uri="http://www.ptc.com/windchill/taglib/carambola" prefix="cmb"%>
```

Note: Although the prefixes can be anything, stay with the standard designations for consistency.

describeTable Details

```
<jca:describeTable var="tableDescriptor" id="netmarkets.project.list" configurable="true"
                   type="wt.projmgmt.admin.Project2" label="Project List">
</jca:describeTable>
```

- The page variable, specified by the *var* attribute, allows the descriptor to be referenced in JSP using EL.
- The *id* specifies a unique name for the table. The component has an id in case it should be refreshed or identified by name by other tags.
- The parameter *configurable* is optional. A value of *true* adds the ability to configure Table Views on this table.

- The parameter *type* specifies the class of objects to be displayed.
  - This is optional as tables sometimes display more than one class of data.
  - For tables that only display one class of data, this is still optional unless subsequent tags depend on the type of data displayed in the table to make decisions. One example is during data retrieval when the data acquisition tag (*getModel*) will return data specific to the type displayed in the table.
- The parameter *label* is the table label.

setComponentProperty Details

setComponentProperty adds more information on the parent element. In this example, setComponentProperty adds more information about the table:

```
<jca:describeTable var="tableDescriptor" id="netmarkets.project.list" configurable="true"
                   type="wt.projmgmt.admin.Project2" label="Project List">
  <jca:setComponentProperty key="actionModel" value="projectlist"/>
</jca:describeTable>
```

In this case, the parent element is *tableDescriptor* and it is given an *actionModel*.

The value of *actionModel* is *projectlist*, which corresponds to an action name.

describeColumn Details

describeColumn is used to add columns to the table. Also note that setComponentProperty can be used within describeColumn to further define the column

```
<jca:describeTable var="tableDescriptor" id="netmarkets.project.list" configurable="true"
                   type="wt.projmgmt.admin.Project2" label="Project List">
  <jca:setComponentProperty key="actionModel" value="projectlist"/>
  <jca:describeColumn id="name" label="Name" sortable="true"/>
  <jca:describeColumn id="completionStatus">
    <jca:setComponentProperty key="percent" value="true"/>
  </jca:describeColumn>
  <jca:describeColumn id="container" label="Host" need="containerReference"/>
  <jca:describeColumn id="modifyStamp" need="thePersistInfo.modifyStamp" sortable="true"
                      defaultSort="true" />
</jca:describeTable>
```

The default sorted column is the column definition with *defaultSort* equal to *true*.

In this example:

- setComponentProperty on “completionStatus” has added more information about the property; specifically, it will be shown as a percentage. Other setComponentProperty keys are wrappable or selectable.
- For the property id equal to “container”
  - The data acquisition mechanism will return an attribute named *containerReference*.
  - If there was a default label associated with “container”, it is being overridden and set to “Host”.

Note: To better understand the additional settings for the column “container”,

- A raw attribute name can have several meanings; “container” can be the container for an object like Part or Document, or refer to nested containers for objects like Projects (which are themselves already containers).
- In this case, for Projects, it refers to the nesting capability of Projects.
- Therefore, data retrieval of an attribute with multiple meanings may not always be able to bring the data back with the most convenient label.
- Setting the label may be required so as the meaning of “container” — in this context, a list of Project containers — is clear.
- For technical consultants, run an Info*Engine Task that returns all attributes for Parts, then for Products. There will be multiple “container” or “containerReference” values.

getModel Details

```
<wc:getCurrentUser var="user"/>
<jca:getModel var="tableModel"  descriptor="${tableDescriptor}"
              serviceName="com.ptc.netmarkets.project.NmProjectService"
              methodName="getProjects">
  <jca:addServiceArgument value="${user}"/>
</jca:getModel>
```

Once the table descriptor is set up, it is passed in to the getModel tag.

The getModel tag takes your table descriptor plus a way to get table data and returns a model object that can be used to render the table.

Like the describeTable tag, the result of the getModel tag is placed in the page-scoped variable defined by the *var* attribute.

From the API, com.ptc.netmarkets.project.NmProjectService.getProjects() returns a QueryResult, but requires an argument of the user.

getModel Details

```
<wc:getCurrentUser var="user"/>
<jca:getModel var="tableModel"  descriptor="${tableDescriptor}"
              serviceName="com.ptc.netmarkets.project.NmProjectService"
              methodName="getProjects">
  <jca:addServiceArgument value="${user}"/>
</jca:getModel>
```

In the example, serviceName and methodName attributes were used to return a QueryResults object.

Alternatively, a queryCommand can be used, which passes in an argument of type com.ptc.core.query.command.common.AbstractQueryCommand.

A good example of this is included in Carambola.

```
<cmb:getTypeInstanceCommand  var="queryCommand" descriptor="${tableDescriptor}"/>
<getModel var="tableModel" descriptor="${tableDescriptor}" queryCommand="${queryCommand}"/>
```

renderTable Details

```
<jca:renderTable model="${tableModel}"/>
```

The rendering of the table is simply done by passing the model object into the renderTable tag.

Footer

```
<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Entire Code Example

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%-->Build a table descriptor and assign it to page variable td<--%>
<jca:describeTable var="tableDescriptor" id="netmarkets.project.list" configurable="true"
                   type="wt.projmgmt.admin.Project2" label="Project List">
  <jca:setComponentProperty key="actionModel" value="project list"/>
  <jca:describeColumn id="name" label="Name" sortable="true"/>
  <jca:describeColumn id="nmActions" />
  <jca:describeColumn id="owner" need="containerInfo.ownerRef"/>
  <jca:describeColumn id="container" label="Host" need="containerReference"/>
  <jca:describeColumn id="category"/>
  <jca:describeColumn id="phase"/>
  <jca:describeColumn id="state" need="containerTeamManagedInfo.state"/>
  <jca:describeColumn id="projectHealthStatus" need="healthStatus"/>
  <jca:describeColumn id="completionStatus">
    <jca:setComponentProperty key="percent" value="true"/>
  </jca:describeColumn>
  <jca:describeColumn id="modifyStamp" need="thePersistInfo.modifyStamp" sortable="true"
                      defaultSort="true" />
</jca:describeTable>
```

```
<%-->Get a component model for our table<--%>
<wc:getCurrentUser var="user"/>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
              serviceName="com.ptc.netmarkets.project.NmProjectService"
              methodName="getProjects">
  <comp:addServiceArgument value="${user}"/>
</jca:getModel>

<%-->Get the NmHTMLTable from the command<--%>
<jca:renderTable model="${tableModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Exercise 8-1: Implement a Product List Table

Objectives

- Implement a JSP page which produces the Product list table.

Scenario

Create a JCA table containing a list of products.

Detailed Description

From previous exercises, there should be a custom first-level tab and several custom second-level tabs. The focus of this exercise will be to display a table in one of the existing a second-level tabs.

Step 1. Use the existing second-level page *WT_HOME/codebase/netmarkets/jsp/newTab/list3.jsp*.

Step 2. Open *list3.jsp*.

a. Begin with the standard lines for a table:

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%-->Build a descriptor and assign it to page variable tableDescriptor<--%>
<jca:describeTable var="tableDescriptor" type="wt.part.WTPart" id="test.table"
                   label="Table Label=${tableName}">
</jca:describeTable>

<%-->Get a query command and assign it to page variable queryCommand<--%>
<cmb:getTypeInstanceCommand var="queryCommand" descriptor="${tableDescriptor}"/>

<%-->Get a table model using our descriptor and query command,
                 assign model to page variable tableModel<--%>
<jca:getModel var="tableModel" descriptor="${tableDescriptor}"
              queryCommand="${queryCommand}"/>

<%-->Render the table model<--%>
<jca:renderTable model="${tableModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

b. Save the file.

Step 3. Test it.

Step 4. Challenge: review the TagLib documentation for describeTable and add pagination.

Exercise 8-2: Render an Attribute

Objectives

- Create a new Renderer.

Scenario

The customer has requested a new attribute that represents a “date difference” — the difference between when the object was created and today, in the format hh:mm:ss.

Step 1. Create a new attribute called Lifetime which is defined as the difference between the start date and today

Step 2. Display this value *x days, y minutes and z seconds*

A new GUI Component is required. Take a look at the Carambola example for tags located in *netmarkets/jsp/carambola/tags/tags.jsp*

Tree Component

The process for building a tree is as follows:

- Create a JSP page that renders the tree
- Define a method of retrieving data
- If needed, specify any special data utilities that support the data

Implementing a “tree” is similar to implementing a “table”.

Note:

1. In a tree, as opposed to specifying a service class and method (as in a table), a TreeHandler is specified to populate the content.
2. Existing TreeHandlers can be found with the **Service Report** http://localhost/Windchill/netmarkets/jsp/carambola/svc/report.jsp by searching for “com.ptc.core.components.beans.TreeHandler”.

The elements of this solution include the following:

- *<your_page>.jsp* a JSP implementation of a “tree” component
- *<your>service.properties.xconf* for the definition of Data Utilities and services
- *<your>action.rbInfo* for the definition of action attributes
- *<your>actions.xml* for the definition of actions
- *<your>actionModels.xml* for the definition of action models
- The TreeHandler responsible for populating the tree content

A Tree Component

Defining a tree is similar to describing a table.

- The describeTableTree tag is a table-tree specific version of describeTable.
- Instead of passing a service method to the getModel tag, getModel specifies a TreeHandler.
- Instead of using the renderTable tag, the component model gets passed into the renderTableTree tag to render the tree.
- The only other difference in this JSP page is that the command bean is assigned the root *oid* of an expandable object (e.g., a Part BOM or a complete container/folder/object listing).
  - This information is needed in order to retrieve the primary *oid* from the NmCommandBean
  - To determine the root nodes, the TreeHandler class has access to the ModelContext.
    - The ModelContext in turn has a reference to the NmCommandBean that can be used to get the request parameters.
  - When the user passes an *oid* in the request parameter, the URL for the JSP page should be something like:

    treeExample.jsp?oid=OR%3Awt.part.WTPart%3A0000
    - Where “%3A” represents a colon and “0000” is the oid of a Part.

Note: By convention, the page variable for the tree descriptor is typically named *treeDescriptor*, and the variable for the component model is typically named *treeModel*.

TreeHandler

A custom TreeHandler will implement com.ptc.core.components.beans.TreeHandler.

The TreeHandler has certain methods which must be implemented:

- ModelContext getModelContext() gets the handler’s model context

- void setModelContext(ModelContext mc) throws WTException sets the contextual information about the tree that is being built, like the descriptor, command bean etc. This should be initialized before data is requested from the handler.
- List getRootNodes() throws WTException gets the list of root nodes for the tree.
- Map<Object,List> getNodes(List parents) throws WTException returns a mapping of the child nodes for each of the parent nodes in the given list This is the only method that will be called for the expand action, so this method must be able to initialize the handler properly so that it can answer the question of what are the children of the parent.
- boolean isExpandNeeded(Object node, int level) throws WTException determines whether or not the given node needs to be expanded. The default implementation looks at the session state of the list of nodes already expanded or configured by the tag.
- boolean hasChildren(Object node) throws WTException determines whether or not the given node should show the expand icon if its collapsed. Override this method to improve performance or to get custom behavior. The default implementation calls the getNodes and sees if the size is larger than 0.
- void addExpandedNode(Object node) throws WTException adds a node to the expanded node list. This list is returned and used in session state to remember what the tree expansion is.

The TreeHandler could also extend a default implementation of com.ptc.core.components.beans.TreeHandler named com.ptc.core.components.beans.TreeHandlerAdapter. In this case, the following methods should be overridden:

- getRootNodes()
- getNodes(List)

Register the TreeHandler by adding the an entry in *site.xconf* or a services file:

```
<Service context="default" name="com.ptc.core.components.beans.TreeHandler">
  <Option requestor="object_type" selector="custom_TreeHandler_handler"
          serviceClass="custom_TreeHandler_class" cardinality="duplicate"/>
</Service>
```

Example getRootNodes() Method

This method returns a List of root nodes by based on a Part *oid*.

```
public List getRootNodes() throws WTException {
  NmCommandBean nmcommandbean = getModelContext().getNmCommandBean();
  if(nmcommandbean == null) {
    return null;  // Error
  }
  NmOid nmoid = nmcommandbean.getPrimaryOid();
  WTPart wtpart;
  if(nmoid == null) {
    return null;  // Error
  } else {
    if (!nmoid.isA( wt.part.WTPart.class )) {
      //Error
      //Following 2 lines are one line
      throw new ClassCastException(
       (new StringBuilder()).append("Expected part, but was: ").append(nmoid).toString());
    }
    wtpart = (WTPart)nmoid.getRef();
  }
  if(wtpart == null) {
    return null;  // Error
  } else {
    configSpec = ConfigHelper.service.getConfigSpecFor(wtpart);
    return Collections.singletonList(wtpart);
  }
}
```

- The configSpec variable is a class variable of type ConfigSpec and is used in both the getRootNodes method and getNodes method.

- The TreeHandler has access to the ModelContext, which can be used to access required information.
- In this case, TreeHandler extends TreeHandlerAdapter which defines getModelContext
- The method first retrieves the NmCommandBean from the model context.
  - If the bean is null, then the method returns null.
- Then, it gets the primary oid from the command bean.
  - If no oid was requested from the user, then this is an error.
  - If the user specified an oid in the request parameter, it checks to make sure the oid is for a WTPart.
    - Otherwise, it throws a ClassCastException.
- If the oid is for a Part, it continues by retrieving the part from the oid via the getRef() method.
  - This method inflates the referenced object which then can be cast into a WTPart..
- The configSpec variable gets assigned the ConfigSpec of the part using the ConfigHelper class.
- Lastly, an immutable List containing the part is returned.

Example getNodes() Method

This method gets a mapping of the child nodes for each of the parent nodes in the given List.

```
public Map getNodes(List list) throws WTException {
  if(configSpec == null) {
    configSpec = getDefaultConfigSpec();
  }
  HashMap hashmap = new HashMap();
  Persistable p[][][] = WTPartHelper.service.getUsesWTParts(new WTArrayList(list), configSpec);
  ListIterator listiterator = list.listIterator();
  do {
    if(!listiterator.hasNext()) {
      break;
    }
    WTPart wtpart = (WTPart)listiterator.next();
    wt.fc.Persistable p1[][] = p[listiterator.previousIndex()];
    if(p1 != null) {
      ArrayList arraylist = new ArrayList(p1.length);
      hashmap.put(wtpart, arraylist);
      wt.fc.Persistable p2[][] = p1;
      int i = p2.length;
      int j = 0;
      while(j < i) {
        wt.fc.Persistable p3[] = p2[j];
        arraylist.add(p3[1]);
        j++;
      }
    }
  } while(true);
  return hashmap;
}
```

- It can be called directly without calling the getRootNodes() method first (e.g., expanding a node already in a table).
  - This means that this method must be able to initialize the handler properly so that it can address which objects are the children of the parent.
- The method first checks to see if the configSpec is initialized from the previous method.
  - If it isn't, it calls the getDefaultConfigSpec method which gets the default ConfigSpec from the WTPart class via the ConfigHelper.
- A Map variable named result is then initialized.
  - This will store the mapping of the child nodes for each of the parent nodes and gets returned from this method.

- A three-dimensional array is also initialized using the getUsesWTParts method from the WTPartHelper class.
  - This method navigates from many *used-by* parts (parents) to their *uses* part masters (children) and applies a ConfigSpec to select the Iterations of the uses parts
  - It returns a three dimensional array of Persistable objects where:
    - The first dimension corresponds to the used-by parts passed in.
    - The second dimension corresponds to the part usage links from the used-by part.
    - The third dimension is an two element array where
      - [0] is the WTPartUsageLink
      - and [1] is the uses WTPart or WTPartMaster if the ConfigSpec did not select an iteration of the uses part.
- It continues by iterating through the List of parent nodes.
- A 2D array of Persistable named “branch” is initialized.
  - Stores the child nodes of the given parent, if any exist.
  - If no children exist, continue to the next iteration.
  - If children do exist, they are added to an ArrayList named children.
- Lastly, the parent node with its corresponding children are added to the Map, and the Map is returned.

Other Tree Components

The *WCCustomizersGuide.pdf* provides information on other components that can be configured on the tree. Here is a brief list:

- Defining Node Column
- Adding ToolBar
- Adding MenuBar
- Adding Row Level Actions
- Enabling selectable rows
- Enabling sortable columns: within the limits of preserving the structure
- Specify DataUtility for columns
- Enabling views
- Control Level of Tree Expansion
- Adding Help
- Hide Expand/Collapse Norgie
- Enable Scrolling
- Set single/multi select
- Enable Tree Count
- Having Tree rows pre-selected
- Enable the Pagination

Exercise 8-3: Implement a Tree

Objectives

- Implement a tree.

Scenario

A customer wants a custom sub tab to display the BOM for a Part.

Detailed Description

From previous exercises, there should be a custom first-level tab and several custom second-level tabs. The focus of this exercise will be to display a table in one of the a second-level tabs.

Step 1. Refer to the OOTB *tree.jsp* located in *WT_HOME/codebase/netmarkets/jsp/carambola/svc/tree.jsp*, as needed.

Step 2. Edit the JSP.

a. Backup *WT_HOME/codebase/netmarkets/jsp/newTab/list4.jsp*.

b. Remove the previous content from *list.jsp*.

c. Add the standard fragments and taglib directives, plus a taglib directive for the JSTL core tags:

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

d. Add a describeTableTree tag with two columns, name and number.

e. In this exercise, a sample “tree handler” from the JCA exmaples will be used. This is the “partTree” handler. Add a getModel tag with a treeHandler attribute with a value of **partTree**.

f. Tree Handlers can be found with the **http://localhost/Windchill/netmarkets/jsp/carambola/svc/report.jsp** searching for service “com.ptc.core.components.beans.TreeHandler”.

g. Add a renderTableTree tag.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<jca:describeTableTree var="treeDescriptor" type="wt.part.WTPart"
                       id="com.gsdev.client.tree" label="A Tree ${treeName}">
  <jca:describeColumn id="name"/>
  <jca:describeColumn id="number"/>
</jca:describeTableTree>

<jca:getModel var="treeModel" descriptor="${treeDescriptor}" treeHandler="partTree"/>

<jca:renderTableTree model="${treeModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

h. Add a JSTL core tag to set the value of commandBean (initialized in the *begin.jspf* fragment) to have a Part *oid*. This value is initialized from the request object parameter *oid*.

The commandBean is expecting a format “part$bom$$<oid>”, therefore the core tag added takes the request parameter *oid* and reformats it for input to the commandBean.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%-->Build a descriptor and assign it to page variable treeDescriptor<--%>
<jca:describeTableTree var="treeDescriptor" type="wt.part.WTPart"
                       id="com.gsdev.client.tree" label="A Tree ${treeName}">
  <jca:describeColumn id="name"/>
  <jca:describeColumn id="number"/>
</jca:describeTableTree>

<c:set target="${commandBean}" property="compContext" value="part$bom$${param.oid}"/>

<jca:getModel var="treeModel" descriptor="${treeDescriptor}" treeHandler="partTree"/>

<jca:renderTableTree model="${treeModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Step 3. Test.

a. Open the JSP page.

b. Specify an *oid* as request object data. If necessary, ask the instructor for help to find a valid oid and format for your system.

Exercise 8-4: Implement a Custom TreeHandler

Objectives

- Implement a custom TreeHandler.

Scenario

A customer wants a custom sub tab to display the BOM for a Part.

Detailed Description

1. From previous exercises, there should be a custom first-level tab and several custom second-level tabs. The focus of this exercise will be to display a tree in one of the a second-level tabs.
2. Instead of using the JCA carambola example, write a custom TreeHandler.

Step 1. Edit *list4.jsp*.

a. Edit *list4.jsp* and update the getModel tag with a treeHandler attribute with a value of **customTreeHandler**. This is the selector for the custom TreeHandler that will be written.

Example:

```
<jca:getModel var="treeModel" descriptor="${treeDescriptor}"
              treeHandler="customTreeHandler"/>
```

Step 2. Create a TreeHandler.

a. Create a class com.gsdev.client.CustomTreeHandler that extends TreeHandlerAdapter:

Example:

```
package com.gsdev.client;

public class CustomTreeHandler extends TreeHandlerAdapter {

 public CustomTreeHandler() {
 }

   private ConfigSpec configSpec;

}
```

b. Add the import statements:

Example:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;

import wt.fc.Persistable;
import wt.fc.QueryResult;
import wt.fc.collections.WTArrayList;
import wt.part.WTPart;
import wt.part.WTPartHelper;
import wt.query.QuerySpec;
import wt.query.SearchCondition;
import wt.util.WTException;
import wt.vc.config.ConfigHelper;
import wt.vc.config.ConfigSpec;

import com.ptc.core.components.beans.TreeHandlerAdapter;
import com.ptc.netmarkets.model.NmOid;
import com.ptc.netmarkets.util.beans.NmCommandBean;
```

c. Add the method getRootNodes:

Example:

```
public List getRootNodes() throws WTException {
 NmCommandBean nmcommandbean = getModelContext().getNmCommandBean();
 if(nmcommandbean == null) {
  return null;
 }
 NmOid nmoid = nmcommandbean.getPrimaryOid();
 WTPart wtpart;
 if(nmoid == null) {
  wtpart = getGolfCart();
 } else {
  if (!nmoid.isA( wt.part.WTPart.class )) {
   //The following 2 lines are one line
          throw new ClassCastException(
          (new StringBuilder()).append("Expected part, but was: ").append(nmoid).toString());
  }
  wtpart = (WTPart)nmoid.getRef();
 }
 if(wtpart == null) {
  return null;
 } else {
  configSpec = ConfigHelper.service.getConfigSpecFor(wtpart);
  return Collections.singletonList(wtpart);
 }
}
```

d. Add the method getNodes:

Example:

```
public Map getNodes(List list) throws WTException {
 if(configSpec == null) {
  configSpec = getDefaultConfigSpec();
 }
 HashMap hashmap = new HashMap();
  Persistable p[][][] = WTPartHelper.service.getUsesWTParts(new WTArrayList(list), configSpec);
  ListIterator listiterator = list.listIterator();
  do {
   if(!listiterator.hasNext()) {
    break;
   }
   WTPart wtpart = (WTPart)listiterator.next();
   wt.fc.Persistable p1[][] = p[listiterator.previousIndex()];
   if(p1 != null) {
    ArrayList arraylist = new ArrayList(p1.length);
    hashmap.put(wtpart, arraylist);
    wt.fc.Persistable p2[][] = p1;
    int i = p2.length;
    int j = 0;
    while(j < i) {
     wt.fc.Persistable p3[] = p2[j];
     arraylist.add(p3[1]);
     j++;
    }
   }
  } while(true);
 return hashmap;
}
```

e. Add the method getDefaultConfigSpec:

Example:

```
private ConfigSpec getDefaultConfigSpec() throws WTException {
  return ConfigHelper.service.getDefaultConfigSpecFor(wt.part.WTPart.class);
}
```

f. Save the file.

g. Compile the file.

Step 3. Register the custom TreeHandler.

a. Open *custom-service.properties.xconf*.

b. Add the following:

Example:

```
<Service context="default" name="com.ptc.core.components.beans.TreeHandler">
  <Option requestor="java.lang.Object"
          serviceClass="com.gsdev.client.CustomTreeHandler"
          selector="customTreeHandler"
          cardinality="duplicate"/>
</Service>
```

c. Propagate the changes.

d. Restart the Method Server and Tomcat.

Step 4. Test.

a. Open the JSP page.

b. Specify an *oid* as request object data. If necessary, ask the instructor for help to find a valid oid and format for your system.

Summary

After completing this module, you should be able to:

- Add a custom table to the user interface.
- Add a custom tree to the user interface.
- Implement a client architecture tree for the display of business objects.
- Control how a table will appear using configuration properties.

Module

9

Advanced Component Configuration

In this module, OOTB **Info Pages** will be customized ; third-level navigation will be customized, and an advanced search client will be added to the user interface.

Objectives

Upon successful completion of this module, you will be able to:

- Customize the **Online Help** for pages.
- Use UI component “wrappers” (to display multiple GUI Components).
- Implement an attributes table component for a given business object **Info Page**.
- Add and modify the ability to manipulate attachments on business object.

Lecture Notes

You may use the space below to take your own notes.

Exercise 9-1: Modify Third Level Attributes Panel

Objectives

- Identify the JSP page for Part attributes.
- Modify the Part attribute panel.

Scenario

A customer needs to display a soft attribute in the **General > Attributes** panel for Parts.

Step 1. Create a soft attribute for Parts.

a. Open **Site > Utilities > Type and Attribute Manager**.
b. Select the **Attribute Definition Manager** tab.
c. If an Organizer named “Business” does not exist, create it.
d. Under the **Business** attribute organizer, create a soft attribute named **`your first name`** and **Type** equal to **`String`**.
e. The *logical id* should be the same as the attribute name.
f. Select **OK**.

Step 2. Add a soft attribute to Parts.

a. Open **Site > Utilities > Type and Attribute Manager**.
b. Select the **Type Manager** tab.
c. Highlight **Part** and select the edit icon. This function will both check out the Part type as well as enter edit mode.
d. Select the **Template** tab.
e. Highlight **Attribute Root** and select the **Add Attribute** button.
f. In the **Select Attribute** pop up window, expand **Business** and select the attribute just created. Select the **Select** button.
g. Select **OK**.
h. Highlight **Part** and select the check in icon.
i. Close **Type and Attribute Manager**.

Step 3. Create a Part with a value for the added IBA.

Step 4. Find the appropriate definition for an Attribute Panel for Parts.

a. Search *WT_HOME/codebase/typedservices.properties* for the string “wt.services/rsc/default/com.ptc.netmarkets.util.misc.FilePathFactory/Attributes/wt.part.WTPart/”.
b. The definition file found should be */netmarkets/jsp/part/attributes.jsp*.
c. Copy */netmarkets/jsp/part/attributes.jsp* to */netmarkets/jsp/com/gsdev/client/part/attributes.jsp*.

Step 5. Edit */netmarkets/jsp/com/gsdev/client/part/attributes.jsp*.

a. Add the following code near the bottom of the file.

Example:

```
            <jca:describeProperty
                 id="ALL_SOFT_AND_SOFT_SCHEMA_ATTRIBUTES_MINUS_CLASSIFICATION_ATTRIBUTES"/>

            <jca:describeProperty id="IBA|com.gs.Cost"/>
            <jca:describeProperty id="Cost"/>

        </jca:describeAttributesTable>
```

b. The two acceptable forms for a soft attribute have been added: the long form (“IBA|com.gs.Cost”) and the shorter logical id alias (“Cost”).
c. Note that “ALL_SOFT_AND_SOFT_SCHEMA_ATTRIBUTES_MINUS_CLASSIFICATION_ATTRIBUTES” was not commented, so all IBAs will be displayed, and then Cost will be displayed twice.
d. Save the file.

Step 6. Register the new **Attribute Panel** file for wt.part.WTPart.

a. Edit *custom-service.properties.xconf* and add the following“<Option requestor="wt.part.WTPart" resource="/netmarkets/jsp/com/gsdev/client/part/attributes.jsp" selector="Attributes"/>” to and propagate the change.

b. Propagate the changes.
c. Restart the Method Server and Tomcat.

Step 7. Test.

a. Open the properties page for a Part.
b. How many times is “Cost” displayed for a Part that does not have a Cost value set? Does this make sense?

Exercise 9-2: Modify Third-level Navigation Bar

Objectives

- Remove third-level navigation.
- Modify third-level navigation.

Scenario

A customer requires edits to the Document third-level navigation area. The customer is considering removing the area or removing the actions **Structure** and **General**. Implement both to demonstrate the look and feel to the customer.

Step 1. Remove the third-level navigation bar for Documents.

a. Register a custom info page for wt.doc.WTDocument.

b. Restart the Method Server and Tomcat.

c. Edit the custom info page to remove the following:

Example:

```
<c:choose>
   <c:when test='${param.miniInfoPage == "true"}'>
      <c:set var="nav_bar_name" value="third_level_nav_dti" />
   </c:when>
   <c:otherwise>
      <c:set var="nav_bar_name" value="third_level_nav_doc" />
   </c:otherwise>
</c:choose>
```

d. Access a Document Properties Page. There should be no navigation bar and no third-level content area.

Step 2. Find the action model that controls the third-level navigation.

a. From the Document Info page snippet, the third-level navigation is controlled by the action model third_level_nav_doc.

b. Search the directory *WT_HOME/codebase/config/actions* for file names that contain "*actionmodels.xml" and the file content contains the string "third_level_nav_doc". The result should be *DocumentManagement-actionmodels.xml*.

Step 3. Modify a third-level navigation bar.

a. Edit *custom-actionmodels.xml*.

b. Copy the *third_level_nav_doc* element from*DocumentManagement-actionmodels.xml* into *custom-actionmodels.xml*.

c. Modify the element to comment out **General** and **Related Items**.

Example:

```
<model name="third_level_nav_doc" defaultActionName="iterationHistory"
       defaultActionType="history">
  <action name="documentStructure" type="object"/>

  <!-- COMMENT OUT
  <submodel name="general"/>
  <submodel name="relatedItems"/>
  -->

  <submodel name="history"/>
  <submodel name="collaboration"/>
</model>
```

d. What issue occurs if "<submodel name="history"/>" is commented out?

Step 4. Reload the action models by executing windchill com.ptc.netmarkets.util.misc.NmActionServiceHelper from a Windchill shell.

Step 5. Challenge: perform a similar edit to Parts.

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>

<%-->Build a table descriptor and assign it to page variable td<--%>
<%-->Note: In production, you may want to localize the labels<--%>
<jca:describeTable var="tableDescriptor" id="tableId" label="Simple I*E Table"
                   type="wt.fc.Persistable">
   <jca:describeColumn id="name"        label="Name" />
   <jca:describeColumn id="address"     label="Address" />
   <jca:describeColumn id="birthday"    label="Birthday" />
   <jca:describeColumn id="phone"       label="Phone" />
   <jca:describeColumn id="mailingList" label="Mailing List" />
</jca:describeTable>

<%-->Get a component model for our table<--%>
<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}" action="create-group">

</jca:getIeModel>

<%-->Get the NmHTMLTable from the command<--%>
<jca:renderTable model="${tableModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Figure 9-1: Table Component for Simple Info*Engine Table

The columns specified in this table are all custom which requires localized labels. In this example, the labels are localized to the JSP page using the “label” attribute.

Note: It is recommended to localize the labels using a resource bundle rather than locally to the page.

The create-group Task Delegate

```
<?xml version="1.0" standalone="yes"?>
<%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<ie:webject name="Create-Group" type="GRP" >
    <ie:param name="CLASS" data="sampleData"/>
    <ie:param name="ELEMENT" data="name=Joe Schmoe:address=100 Cherry Lane:birthday=12/11/77:phone=(555) 555-5555:mailingList=Yes"/>
    <ie:param name="ELEMENT" data="name=Bob Sled:address=123 Maple Drive:birthday=03/21/72:phone=(111) 222-3333:mailingList=No"/>
    <ie:param name="ELEMENT" data="name=Sam Which:address=500 East Main St:birthday=07/01/81:phone=(123) 456-7890:mailingList=No"/>
    <ie:param name="ELEMENT" data="name=Jane Doe:address=777 Sunset Blvd:birthday=04/18/74:phone=(000) 111-2222:mailingList=Yes"/>
    <ie:param name="ELEMENT" data="name=Mary Joe:address=987 Park Ave:birthday=10/03/68:phone=(987) 654-3210:mailingList=Yes"/>
    <ie:param name="GROUP_OUT" data="output"/>
</ie:webject>
```

Figure 9-2: Sample data used in the simple Info*Engine table:

This sample XML file is an Info*Engine task that creates and returns a simple set of data (Virtual Database Group) that can be rendered by the table component.

Creating the Delegate

Make sure the name of the delegate matches the name specified in the action attribute for the getIeModel tag.

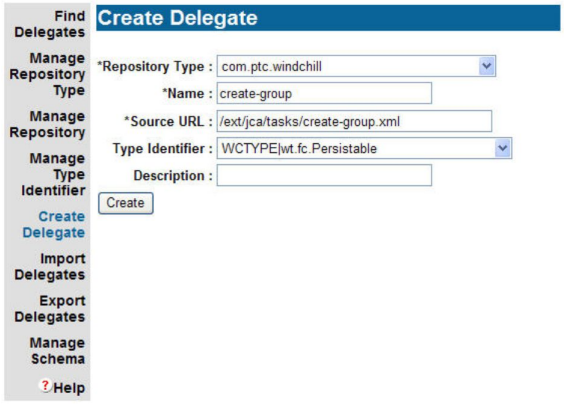


Figure 9-3:

Viewing the Simple Info*Engine Table

Figure 9-4: Viewing the Simple Info*Engine Table

Info*Engine Search Page

This JSP Page allows the user to search for object types using a simple form to enter query criteria

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core"      prefix="ie"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

 <div style="margin-left:10px">

 <h2>Search Page</h2>

 <table style="margin-left:100px" class="pp">
   <tr>
     <th scope="row" class="tablecolumnheaderfont">Type:</th>
     <td class="tabledatafont">
        <input type="text" name="search_type" size="50" maxlength="100"
               value="${param.search_type}"/></td>
   </tr>
   <tr>
     <th scope="row" class="tablecolumnheaderfont">Where:</th>
     <td class="tabledatafont">
        <input type="text" name="WHERE" size="50" maxlength="100" value="${param.WHERE}"/></td>
   </tr>
   <tr>
      <td colspan="2" align="right"><input type="submit" value="Search"/></td>
   </tr>
   <input type="hidden" name="PAGE_COUNT" value="15"/>
 </table>
```

Figure 9-5: Info*Engine Search Page

This image is the first half of the JSP page. It uses standard HTML code to set up a simple form where the user enters the object type and a where clause. The object type gets passed into a parameter called “search_type,” and the where clause gets passed into a parameter called “WHERE.”

The Info*Engine task requests the parameters using the getParam() method in scriptlet code.

The table component is rendered once the form is submitted

```
<c:choose>
<c:when test="${param.search_type != null}">

<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
   <jca:describeColumn id="name"/>
   <jca:describeColumn id="number"/>
   <jca:describeColumn id="version"/>
 </jca:describeTable>

<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}" action="psc-Search">
   <ie:param name="ATTRIBUTE_PARAM_NAME" data="dca_attribute"/>
</jca:getIeModel>

<jca:renderTable model="${tableModel}"/>

</c:when>
</c:choose>

</div>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Figure 9-6: Info*Engine Search Page Table Component

This image is the second half of the JSP page. The choose tag checks to see if the search_type parameter is not null. If it isn't, it continues by defining and rendering the table. Once again, the table descriptor gets passed into the getIeModel tag instead of getModel.

Running the Info*Engine Search Page

Upon launching the JSP page, the simple form will be displayed

Figure 9-7: Info*Engine Simple Form

Using the search criteria shown in the image, the task will search for all WTParts that begin with the letter L. The asterisk (*) is used as a wildcard in the where clause.

Once the form is submitted, a table with the search results is displayed

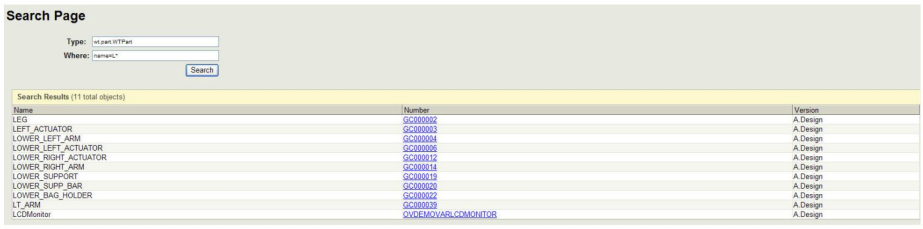


Figure 9-8: Search Results

Additional Properties to the Table Component

To make a table component more useful than just for simple display purposes, other properties can be added:

- Toolbar and menu bar
- Selectable rows
- Configurable views

Adding a Toolbar

Adding a toolbar to the table is achieved by using the setComponentProperty tag

- This is a helper tag that sets properties on the component descriptor managed by a parent describe tag

- The key attribute and the value attribute are the key and value for the properties map of the target ComponentDescriptor, respectively

In order to add a toolbar, the key attribute must be set to “actionModel” and the value attribute must be set to the name of the action model that contains the toolbar actions.

```
<jca:describeTable var="tableDescriptor" id="tableId" label="Simple Table">
   <jca:setComponentProperty key="actionModel" value="customToolbar" />
   <jca:describeColumn id="name"/>
   <jca:describeColumn id="number"/>
   <jca:describeColumn id="checkoutInfo.state"          label="State"/>
   <jca:describeColumn id="thePersistInfo.createStamp"  label="Created"/>
</jca:describeTable>
```

Figure 9-9: Adding a Toolbar

Both the key and value attributes are required for the setComponentProperty tag. One additional attribute to specify is the “target” attribute. This attribute specifies the component descriptor to set the property on. This attribute is only necessary if the setComponentProperty tag is not nested within a supported parent tag. The setComponetProperty tag for this example is nested in the describeTable tag, so the target attribute is not needed.

Toolbar Action Model

This example uses a toolbar named “customToolbar” that contains pre-existing folder actions

- If the table uses a new action model for the toolbar, it must be registered in the *actionModels.xml file

```
<model name="customToolbar">
   <action name="list_cut"            type="object"/>
   <action name="list_copy"           type="object"/>
   <action name="fbpaste"             type="object"/>
   <action name="list_delete"         type="object"/>
   <action name="separator"           type="separator"/>
   <action name="create"              type="document"/>
</model>
```

Figure 9-10: Toolbar Action Model

Formally, all action models were registered in *actionModels.xml*. Windchill provides several **actions.xml* and **actionModels.xml* named accordingly to their associated module or logical packaging. These files are located in *WT_HOME/codebase/config/actions*. There is also a *custom-actions.xml* and *custom-actionModels.xml* file available to register new actions and action models. The “customToolbar” action model is registered in the *custom-actionModels.xml* file.

Displaying the Toolbar

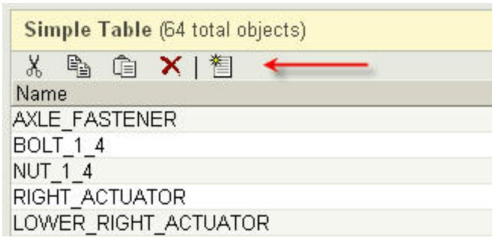


Figure 9-11: Simple table example with the custom toolbar added

The “separator” action is displayed as a vertical line between the action icons.

Adding a Menu Bar

To add a menu bar, the menubarName attribute from the describeTable tag must be included

- The name of action model to use for the menu bar actions is specified in this attribute

```
<jca:describeTable var="tableDescriptor" id="tableId"
   label="Simple Table" menubarName="customMenubar" >
   <jca:setComponentProperty key="actionModel" value="customToolbar" />
   <jca:describeColumn id="name"/>
   <jca:describeColumn id="number"/>
   <jca:describeColumn id="checkoutInfo.state"          label="State"/>
   <jca:describeColumn id="thePersistInfo.createStamp"  label="Created"/>
</jca:describeTable>
```

Figure 9-12: The menubarName attribute

Menu Bar Action Model

This example uses an action model named “customMenubar”

- The action model must also be registered in the *action-models.xml file
- Action models can be nested using the submodel tag

```
<model name="customMenubar">
   <submodel name="fileMenu"  />
   <submodel name="editMenu"  />
</model>

<model name="fileMenu">
   <action name="list_cut"              type="object"/>
   <action name="list_copy"             type="object"/>
   <action name="fbpaste"               type="object"/>
   <action name="list_delete"           type="object"/>
</model>

<model name="editMenu">
   <action name="create"                type="document"/>
</model>
```

Figure 9-13: Menu Bar Action Model

This custom menu bar with the File and Edit submodels are all registered in the *custom-actionModels.xml* file along with the toolbar.

Localizing the Menu Bar Action Properties

In order to label each menu, the description must be set for the File and Edit submodels

- The action properties must be included in either the *action.properties* file or in a resource bundle
- The properties depicted below are the entries that get added to the action.properties file

```
object.fileMenu.description=File
object.editMenu.description=Edit
```

The syntax for properties stored in resource bundles is similar to that of the *action.properties* file. The entries for the **File** and **Edit** action models would look like:

object.fileMenu.description.value=File

object.editMenu.description.value=Edit

Enabling Selectable Rows

Making the rows selectable can be easily done by adding another setComponentProperty tag

- The key attribute must be set to “selectable” and the value attribute must be set to *true*

```
<jca:describeTable var="tableDescriptor" id="tableId"
   label="Simple Table" menubarName="customMenubar" >
   <jca:setComponentProperty key="actionModel" value="customToolbar" />
   <jca:setComponentProperty key="selectable"  value="true" />
   <jca:describeColumn id="name"/>
   <jca:describeColumn id="number"/>
   <jca:describeColumn id="checkoutInfo.state"          label="State"/>
   <jca:describeColumn id="thePersistInfo.createStamp"  label="Created"/>
</jca:describeTable>
```

Figure 9-14: Enabling Selectable Rows

Displaying the New Table

Adding the menu bar and selectable rows will produce the following results:

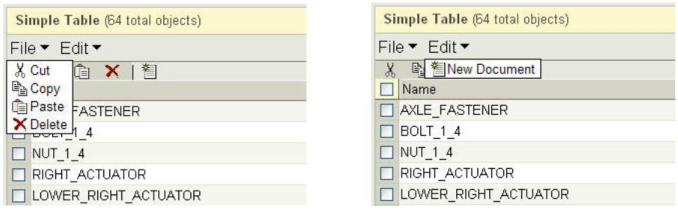


Figure 9-15: Displaying the New Table

The “separator” action can also be added to menu bar action models to organize any actions. The separator is displayed as a horizontal line in the drop-down menu between the actions.

Adding Configurable Views

Defining a configurable table requires these two main steps:

- Defining the columns, filter criteria, etc. that you want to show up in the configurable view wizard
- Registering the table and setting up the JSP page to enable configurable views

Defining a Configurable Table

In order to define a new configurable table is to implement the ConfigurableTable interface

- The interface has methods that tell the configurable view wizard which types to use, what columns should be available for selection, etc.

Table views intended for JCA are required to extend the JCAConfigurableTable class

- There are some differences in the way JCA handles configurable tables which are keyed off whether the configurable table instance is a JCAConfigurableTable

The JCAConfigurableTable class extends AbstractConfigurableTable which implements the ConfigurableTable interface.

Creating the Configurable Table Class

Most of the methods in the JCAConfigurableTable class are self-explanatory, once the configurable view wizard is executed to see the options it presents:

- getClassTypes()
- getSpecialTableColumnsAttrDefinition(Locale locale)
- getOOTBTableViews(String tableId, Locale locale)
- getDefaultSortColumn()
- getOOTBActiveViewName()
- getLabel(Locale locale)

The getClassTypes Method

This method describes the types of objects that can appear as rows on the table

- The information is used from the data type to automatically discover the set of possible columns that can be selected for the table
  - The sets of columns that are always available for a given data type can be configured in the AvailableAttributes.xml file
- This method can simply return “null,” in which case all of the columns will have to be determined using the following getSpecialTableColumnsAttrDefinition method

```
public Class[] getClassTypes() {
    return new Class[]{ WTPart.class };
}
```

When a data type like WTPart is included in this method like in the example, then the infrastructure knows that there a few columns that are always available for parts like name and number.

The *AvailableAttributes.xml* file is located in the *WT_HOME/codebase/com/ptc/core/htmlcomp/createtableview* directory

The getSpecialTableColumnsAttrDefinition Method

This method lets the configurable view wizard know about columns that it cannot find automatically

- Automatic columns are discovered by looking up the global columns that map to each data type returned by the getClassTypes method
- It returns a List of Attribute.AbstractAttribute objects in which each one represents a column that can be selected
- The advantage of using these attribute definitions is that it tells the configurable view wizard how to filter the criteria for the column

```
public List getSpecialTableColumnsAttrDefinition(Locale locale) {
    List result = new ArrayList();
    result.add(new Attribute.TextAttribute(/*id*/"foo", /*label*/"Foo", locale));
    result.add(new Attribute.TextAttribute(/*id*/"bar", /*label*/"Bar", locale));

    return result;
}
```

Figure 9-16: The getSpecialTableColumnsAttrDefinition Method

The example shown creates two special columns both of type Attribute.TextAttribute. Altogether, there are four concrete types of Attibute.AbstractAttribute that can be specified:

BooleanAttribute – A column that contains a boolean value

DateAttribute – A column that contains a Date value

TextAttribute – A column that contains a String value

ListAttribute – A column that contains a List of predefined values

A fifth type of Attibute.AbstractAttribute is CustomAttribute which is not used by JCA.

If a column is represented by a BooleanAttribute, for example, then the wizard will know how to filter the column which will be either by “true” or “false.”

The getOOTBTableViews Method

This method is where the configurable views are created that will appear in the current view drop-down by default

Each configurable view is represented by a TableViewDescriptor object in which a List of TableViewDescriptor objects gets returned

- Within each TableViewDescriptor contains the columns that will be displayed for that view
- These columns are represented by a Vector of TableColumnDefinition objects

If the configurable table implementation extends AbstractConfigurableTable (which JCAConfigurableTable does), then you can take advantage of some reusable helper methods to get standard columns added to your OOTB views.

The method shown below creates one OOTB view named “Sample View”

- Additional views can be added by creating another TableViewDescriptor and adding it to the List

```
public List getOOTBTableViews(String tableId, Locale locale) throws WTException {
    // The list of all ootb views
    List result = new ArrayList();

    // The columns that make up one ootb view
    Vector columns = new Vector();
    columns.add(TableColumnDefinition.newTableColumnDefinition(/*name*/ColumnIdentifiers.NAME, /*lockable*/false));
    columns.add(TableColumnDefinition.newTableColumnDefinition(/*name*/ColumnIdentifiers.NUMBER, /*lockable*/false));
    columns.add(TableColumnDefinition.newTableColumnDefinition(/*name*/ColumnIdentifiers.CREATED, /*lockable*/false));
    columns.add(TableColumnDefinition.newTableColumnDefinition(/*name*/ColumnIdentifiers.VERSION, /*lockable*/false));

    // Construct a descriptor based on columns
    TableViewDescriptor tvd = TableViewDescriptor.newTableViewDescriptor(/*name*/"Sample View",tableId, /*system*/ true,
            /*global*/true, columns, /*constraints*/null, /*match*/true, /*description*/"Sample View");

    // Add the descriptor to the list of all ootb views
    result.add(tvd);

    return result;
}
```

Figure 9-17: TableViewDescriptor

Column ids specified in the newTableColumnDefinition method call should use the ones in the DescriptorConstants.ColumnIdentifiers class when possible.

The Remaining ConfigurableTable Methods

The getDefaultSortColumn method chooses which column will be sorted when the table is viewed.

```
public String getDefaultSortColumn() {
    return ColumnIdentifiers.NAME;
}
```

Figure 9-18: getDefaultSortColumn

The getOOTBActiveViewName method lets you specify which one of the configurable views created in the getOOTBTableViews method will be active when you first view the table.

```
public String getOOTBActiveViewName() {
    return null;
}
```

Figure 9-19: getOOTBActiveViewName

The getLabel method is used to label the configurable view wizard.

```
public String getLabel(Locale locale) {
    return "Custom Config Table";
}
```

Figure 9-20: getLabel

The getDefaultSortColumn method may return “null” if no columns need to be Sorted. In this example, only one OOTB table view was made, so there was no need to specify a view in the getOOTBActiveViewName method.

Register the Configurable Table

Once the table is implemented, it needs to be registered in an application context in order for the infrastructure to find it.

- Add a service to the service.properties.xconf file
- The serviceClass attribute must match the location of the JCAConfigurableTable class

```
<Service context="default" name="com.ptc.core.htmlcomp.tableview.ConfigurableTable">
   <Option requestor="java.lang.Object" selector="tableId"
           serviceClass="com.ptc.carambola.refImpl.ConfigTable"/>
</Service>
```

Specifying Configurable Views in the JSP

In order for a table to be configurable, it must be specified in the JSP page.

- This is simply done by adding the configurable=“true” attribute to the describeTable tag
- Also make sure the id attribute corresponds to the selector attribute in the service entry
  - This ensures that the configurable table class is mapped to the JSP page

```
<jca:describeTable var="tableDescriptor" id="tableId"
   label="Config Table" configurable="true" menubarName="customMenubar" >
   <jca:setComponentProperty key="actionModel" value="customToolbar" />
   <jca:setComponentProperty key="selectable"  value="true" />
   <jca:describeColumn id="name"/>
   <jca:describeColumn id="number"/>
   <jca:describeColumn id="checkoutInfo.state"          label="State"/>
   <jca:describeColumn id="thePersistInfo.createStamp"  label="Created"/>
</jca:describeTable>
```

Figure 9-21: Specifying Configurable Views in the JSP

If you want to use a configurable table id that is different from the table id, you can specify this by adding the configurableTableId attribute.

Editing the Configurable Table Implementation

Though the initial configuration is via Java code, at runtime, the table view information is persisted in the database

- The first time that a configurable table is requested, the infrastructure reads in the ConfigurableTable class and persists this information in the database
- If any changes are made afterwards to the table class, the configurable table information must be cleared from the database in order to see the new changes made
  - The following SQL commands must be executed to clear the table information:

```
DELETE FROM TableViewDescriptor;
DELETE FROM ActiveViewLink;
```

Figure 9-22: Clearing the table information

Displaying the Configurable Table

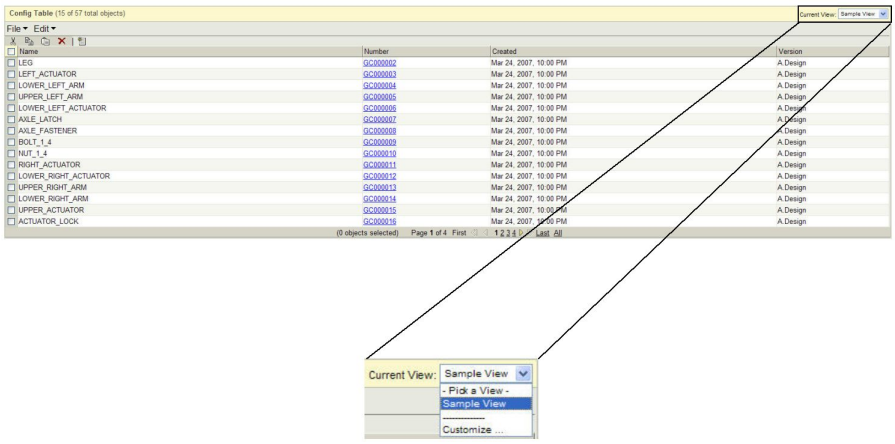


Figure 9-23: Displaying the Configurable Table

Setting configurable="true" adds the Current View drop-down to the top-right corner of the table. The table currently displays the "Sample View" created in the configurable table implementation. If more than one table view was implemented, then the current view displayed would be either the first view on the list (alphabetically) or the table view set in the getOOTBActiveViewName method.

The Configurable View Wizard

Selecting "Customize…" from the current view list launches a pop-up displaying the configurable table views.

- From here, additional table views can be added and removed using the toolbar commands

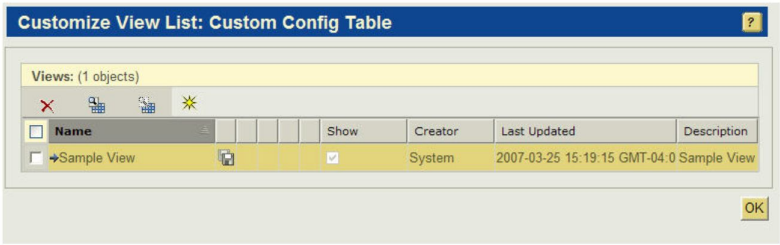


Figure 9-24: The Configurable View Wizard

Note: Table view created programmatically such as the “Sample View” cannot be edited or removed. Only table views created with the configurable view wizard may be modified.

Clicking on the “New View” icon will launch the configurable view wizard

The first step is to give the new view a name and description

- Also included is the option to show or hide the view in the current view list and to share the view with all users

Figure 9-25: Example Table View

For this example the table view name is called, “New” and description is left blank since it is not required.

The second step is to set any filters for a given object type.

- The “Object types” drop-down is populated with the object types specified in the getClassTypes method
- “Criteria” is populated with the attributes corresponding to the selected object type as well as any attributes specified in the getSpecialTableColumnsAttrDefinition method

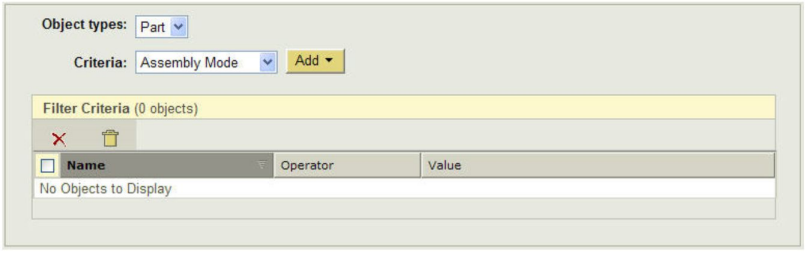


Figure 9-26: Creating a simple configurable view

Since this example is only creating a simple configurable view, no filter criteria is specified for the new view.

The third step is to set the columns to be displayed in the table

- The Top, Bottom, Up, and Down buttons also allow to specify the order in which the columns will appear

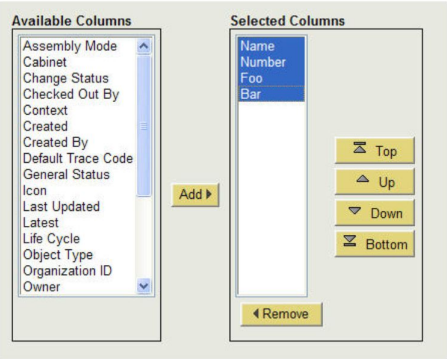
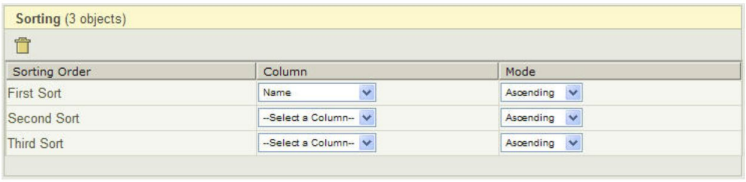


Figure 9-27: Order of Columns

The part name and number columns are added for this example as well as the special columns Foo and Bar created in the getSpecialTableColumnsAttrDefinition method.

The final step is to set the sorting of the columns specified in the previous step

- Up to three different sorts can be specified, but it is not required to sort any column
- The mode for the sort can be set to either ascending or descending



In this example, only the part name will be sorted.

Once the wizard is completed, the new view is then added to the customize view list

- From here, the new view can be edited, saved, set as the active view, hidden from view, or deleted

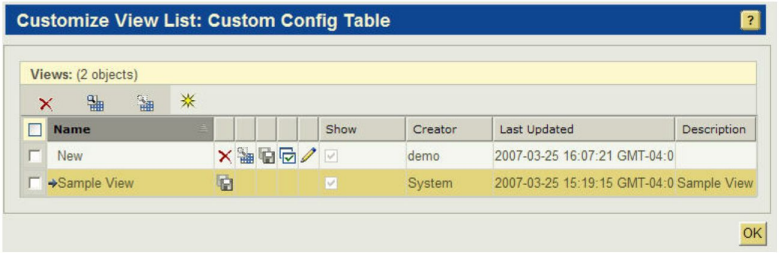


Figure 9-28: The Customize View List

Once “OK” is clicked to close this window, a confirmation box will appear asking to refresh to table in order to see the new changes. Clicking OK will update the table.

Displaying the New Table View

The “New” view will now be added to the current view drop-down

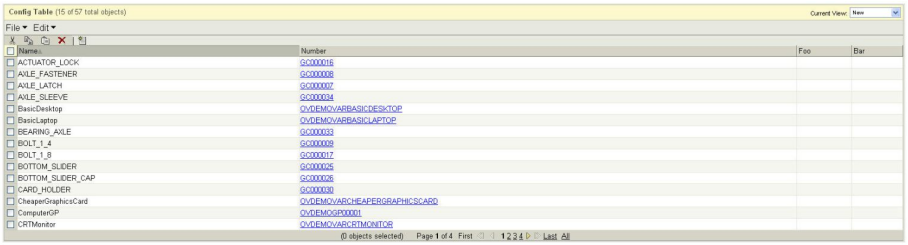


Figure 9-29: Displaying the New Table View

As seen from the image, the table displays the Name, Number, Foo, and Bar columns with the Name column sorted.

Exercise 9-3: Create an Advanced Search Client

Objectives

- Create an Info*Engine task to search for data.
- Create a Task Delegate for the Info*Engine task.
- Create a JSP page with JCA tags to search for data using the Task Delegate.

Scenario

For an existing second-level tab, the second-level tab should use JCA and getIeModel to search for Windchill PDMLink data.

Detailed Description

From previous exercises, there should be a custom first-level tab and several custom second-level tabs. The focus of this exercise will be to display a table in one of the existing a second-level tabs.

Step 1. Create a demo-search Info*Engine Task that returns three attributes: *name*, *number* and *version*.

a. Create a file *WT_HOME/tasks/com/gsdev/client/demo-search.xml*.
b. Add a Query-Objects or Search-Objects webject.
c. Hard-code the INSTANCE parameter, and accept request object input for *where* and *type*.
d. The following is a Search-Objects criteria. Note that Search-Objects webjects have a where criteria like “(attribute=’attributeValue’)”.

Example:

```
<?xml version="1.0" standalone="yes"?>
<%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<ie:webject name="Search-Objects" type="OBJ">
 <ie:param name="ATTRIBUTE" data="name,number,version" delim=","/>
 <ie:param name="INSTANCE"  data="local.ptc.windchill.Windchill"/>
 <ie:param name="WHERE"     data="${@FORM[]where[]}"/>
 <ie:param name="TYPE"      data="${@FORM[]type[]}"/>
</ie:webject>
```

e. Save the task.

Step 2. Implement a Task Delegate.

a. Select **Site > Utilities > Task Delegate Administrator**.
b. Enter the LDAP Administrator credentials. Ask the instructor for help, if necessary.
c. Select **Create Delegate**.
d. For **Repository Type**, select **com.ptc.windchill**.
e. For **Name**, enter **`demo-search`**.
f. For **Source URL**, enter **`com/gsdev/client/demo-search.xml`**.
g. For **Type Identifier**, select **WCTYPE|wt.fc.Persistable**.
h. Select **Create**.

Step 3. Create the JSP page.

a. Use the existing second-level page *WT_HOME/codebase/netmarkets/jsp/newTab/list5.jsp*.
b. Open *list5.jsp*.
c. Add the begin.jspf and end.jspf statements.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

d. Include a taglib directive for JCA components.
e. Include a taglib directive for Info*Engine core components.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

f. Add a form that submits back to itself and accept input for *type* and *where*.
Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<h2>Test Search</h2>
<b>Type:</b></td><td><input type="text" name="type" value="${param.type}"/>
<br>
<b>Criteria:</b><input type="text" name="where" value="${param.where}"/>
<br>
<input type="submit" value="Search"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

g. Add a describeTable tag with 3 columns: *name*, *number* and *version*.
Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<h2>Test Search</h2>
<b>Type:</b></td><td><input type="text" name="type" value="${param.type}"/>
<br>
<b>Criteria:</b><input type="text" name="where" value="${param.where}"/>
<br>
<input type="submit" value="Search"/>

<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
 <jca:describeColumn id="number"/>
 <jca:describeColumn id="version"/>
</jca:describeTable>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

h. Add a getIeModel tag, with Info*Engine core tags for WHERE and TYPE.
Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<h2>Test Search</h2>
<b>Type:</b></td><td><input type="text" name="type" value="${param.type}"/>
<br>
<b>Criteria:</b><input type="text" name="where" value="${param.where}"/>
<br>
<input type="submit" value="Search"/>

<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
 <jca:describeColumn id="number"/>
 <jca:describeColumn id="version"/>
</jca:describeTable>

<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}" action="demo-search">
 <ie:param name="WHERE" data="${param.where}"/>
 <ie:param name="TYPE" data="${param.type}"/>
</jca:getIeModel>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

i. Add a renderTable tag.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<h2>Test Search</h2>
<b>Type:</b></td><td><input type="text" name="type" value="${param.type}"/>
<br>
<b>Criteria:</b><input type="text" name="where" value="${param.where}"/>
<br>
<input type="submit" value="Search"/>
<p>

<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results">
 <jca:describeColumn id="name"/>
 <jca:describeColumn id="number"/>
 <jca:describeColumn id="version"/>
</jca:describeTable>

<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}" action="demo-search">
 <ie:param name="WHERE" data="${param.where}"/>
 <ie:param name="TYPE" data="${param.type}"/>
</jca:getIeModel>

<jca:renderTable model="${tableModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

j. Save the JSP file.

Step 4. Test by opening the appropriate second-level tab associated with *list5.jsp*.

a. For **Type**, enter **`wt.part.WTPart`**. For **Criteria**, enter **`(number='*')`**.

b. For **Type**, enter **`wt.doc.WTDocument`**. For **Criteria**, enter **`(number='*')`**.

Step 5. Add a custom menu bar.

a. Add custom actions.

Example:

```
<model name="customMenubar">
  <submodel name="fileMenu"/>
  <submodel name="fileMenu"/>
</model>

<model name="customMenubar">
  <action name="list_cut" type="object"/>
  <action name="list_copy" type="object"/>
  <action name="fb_paste" type="object"/>
  <action name="list_delete" type="object"/>
</model>

<model name="customMenubar">
  <action name="create" type="document"/>
</model>
```

b. Add a custom menu bar.
Example:

```
<model name="customMenubar">
  <submodel name="fileMenu"/>
  <submodel name="fileMenu"/>
</model>

<model name="customMenubar">
  <action name="list_cut" type="object"/>
  <action name="list_copy" type="object"/>
  <action name="fb_paste" type="object"/>
  <action name="list_delete" type="object"/>
</model>

<model name="customMenubar">
  <action name="create" type="document"/>
</model>
```

c. Add the menu bar to a table.
Example:

```
<jca:describeTable var="tableDescriptor" id="searchTable" label="Search Results"
                   menubarName="customMenubar">
```

Step 6. Make table rows selectable.
a.
Example:

```
<jca:setComponentProperty key="selectable" value="true"/>
```

b.
Step 7. Make table configurable.
a. configurable=“true” attribute to the describeTable

Summary

After completing this module, you should be able to:

- Customize the **Online Help** for pages.
- Use UI component “wrappers” (to display multiple GUI Components).
- Implement an attributes table component for a given business object **Info Page**.
- Add and modify the ability to manipulate attachments on business object.

Module

10

Constructing Wizards

In this module, students will learn the concept of a wizard for the purpose of creating an operation on object by collecting information in step-by-step fashion Students then implement this wizards.

Objectives

Upon successful completion of this module, you will be able to:

- Create wizards
- Add custom validation to wizards.

Lecture Notes

You may use the space below to take your own notes.

Wizards

The elements of this solution are as follows:

- *components.tld* Tag Library Descriptor (TLD) file
  - Wizard Tag and Wizard-Step Tag definition
  - Run time Location: *WT_HOME/codebase/WEB-INF/tlds/*
- *wizard.js* Javascript file
  - Contains all the necessary logic of how to move from one step to another and how to call methods defined for each of the steps.
  - Run time Location: *WT_HOME/codebase/netmarkets/tlds/javascript/components*
- *<your_wizard_page>.jsp* The JSP file in which your wizard implementation is defined.
- *<your_wizard_step>.jsp* The JSP file, which contains contents of the wizard step
- *<*-actions>.xml* The actions for the wizard as well as each wizard step are defined in this XML file
- *actionmodels.xml* The models for the list of buttons, to be displayed at the bottom of the wizard (i.e. the navigation area) are defined in this XML file.
- *<resourceBundle>.rbInfo* This is another option where you can specify locale specific Strings and properties for wizard step and wizard actions.
- *formProcessorController.java* Source for the Java class, which will be executed after the wizard, is submitted. The wizard framework will pass on the data / information to this Java class

Process for Configuring the Wizard:

- Create actions for the wizard and wizard steps
- Create the wizard JSP page
- Specify localized Strings and properties
- Create the wizard step pages

Create Actions

Each step of the wizard, and the wizard itself, need an action. In the example below:

- The create action is defined for the wizard page. Windchill will now expect *create.jsp* to be located in *WT_HOME/codebase/netmarkets/jsp/changeTask* folder. Since this action is for the wizard, you need to specify windowType as “popup” inside the <command> tag.
- The affectedAndResultingItemsStep action is defined for the wizard step. The corresponding JSP should be located at *WT_HOME/codebase/netmarkets/jsp/changeTask/affectedAndResultingItemsStep.jsp*. Since this action is for the wizard step, you need to specify windowType as “wizard_step”.

```
<objecttype name="changeTask" class="wt.change2.WTChangeActivity2">
  <action name="create">
    <command windowType="popup"/>
 </action>

  <action name="affectedAndResultingItemsStep">
    <command windowType="wizard_step" />
  </action>
</objecttype>
```

The Wizard JSP Page

1. Declare the components tag library that contains the necessary tags for constructing the wizard and wizard steps.

```
<%@ taglib prefix="jca" uri="http://www.ptc.com/windchill/taglib/components" %>
```

2. Include files used by the wizard infrastructure:

```
<%@ include file="/netmarkets/jsp/components/beginWizard.jspf"%>
<%@ include file="/netmarkets/jsp/components/includeWizBean.jspf"%>
```

3. Define the wizard using the wizard tag. All the steps for this wizard are defined inside the <wizard> tag using wizardStep tags.

```
<jca:wizard >
 <jca:wizardStep action="defineItemWizStep" type="object"/>
 <jca:wizardStep action="setAttributesWizStep" type="object"/>
 <jca:wizardStep action="affectedAndResultingItemsStep" type="changeTask"/>
</jca:wizard>
```

4. Finally, include the *end.jspf* fragment:

```
<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

Specify Localized Strings and Properties

Localized strings and properties can be defined in either *action.properties* file or resource bundles. Additionally, the height and width of a popup windows — commonly used for wizards — can be specified. For a given type and action, the resource bundle entry would look like:
<TYPE>.<ACTION>.moreurlinfo.value=width=800,height=700

Note: It is recommended to specify localized data in resource bundles.

Customization Points

The following items can be customized in wizards:

- Adding a help icon
- Providing user-defined buttons to Wizard
- Providing user-defined form processor controller
- Providing server-side validation before and/or after processing a wizard step
- Loading the wizard step content when it is visited
- Marking a wizard step as “required”
- Hiding a wizard step
- Displaying a hidden or dynamic step at runtime
- Providing user defined SUBMIT function
- Providing client side validations before a wizard step is displayed

:

Action Attributes

The following attributes can be set for all actions:

- *id <type>.<action>* overrides the id of the wizard step; the default is “type.action”.
- *afterJS* is a JavaScript function name to invoke client side validation for a wizard step when step is finished.
- *beforeJS* is a JavaScript function name to invoke client side validation for a wizard step when step is loaded.
- *beforeVK* is a server-side validator name to invoke server side validation for a wizard step when step is loaded.
- *afterVK* is a server-side validator name to invoke server side validation for a wizard step when step is finished.
- *preloadWizardPage* (false / **true**) specifies that wizard step is to be downloaded when wizard is launched.
- *hidden* (**false** / true) specifies that wizard step is to be hidden at first, or for the action to be rendered as non-selectable.
- *required* (**false** / true) specifies that the wizard step is required.

Providing user defined buttons to Wizard

Buttons that are present at the bottom of the wizard (the navigation area) .

The wizard framework provides a default button model, which consists of buttons like **Next**, **Previous**, **OK**, **Apply** and **Cancel**.

The Wizard framework also provides the ability to specify a custom set of buttons with user-defined functionality. The jca:wizard tag would specify a custom *buttonList* attribute as follows

```
<jca:wizard buttonList="PreferenceWizardButtons">
</jca:wizard>
```

There must be a corresponding entry in an **actionModels.xml* file:

```
<actionmodels>
  <model name="PreferenceWizardButtons">
    <action name="revertButton" type="object"/>
    <action name="okButton" type="object"/>
    <action name="cancelButton" type="object"/>
  </model>
</actionmodels>
```

Every button in the action model should have a corresponding action in *actions.xml* file.

If there is a separate Java class written to render that particular button, than specify the name of that class and its method (which contains the rendering code) using the class and method attributes of the command element. For example:

```
<action name="revertButton" id="PJL_wizard_revert_to_default">
 <command class="" method="" windowType="page" url="javascript:revertToDefault()"/>
</action>
```

User-defined Form Processor Controller

- Custom form processing can be specified with a form processor controller.
- It would be executed when the wizard is completed and the user selects the **OK**, **Finish** or **Apply** buttons.
- The form processor controller gets the data specified in all the wizard steps and processes it.
- After processing the data, a form processor controller returns an object of FormResult, along with the status. Based on the status returned (success or failure), the wizard framework will determine the next action.
- An example of calling a form processor:

```
<jca:wizard formProcessorController=
  "com.ptc.[...].controllers.EffectivityAwareIframeFormProcessorController">
</jca:wizard>
```

Server-side Validation

Server-side validations using specific Java classes and configuring the wizard to execute them before / after processing a wizard step.

Achieve this by using “beforeVK” and “afterVK” attributes of the action tag.

Provide a key as the value for these attributes. The corresponding key should be declared in any <*-service>.properties file specifying the Java class which will perform the validations and return the result. For example:

```
<action
name="setAttributesWizStepForCreateMultiPart" beforeVK="nameNumberValidation">
 <command windowType="wizard_step"/>
</action>
```

Or:

```
<action
name="setAttributesWizStepForCreateMultiPart" afterVK="nameNumberValidation">
 <command windowType="wizard_step"/>
</action>
```

Declaration in a *services.properties* file:

```
<Service context="default" name="com.ptc.core.ui.validation.UIComponentValidator">
```

```
  <Option requestor="null" selector="nameNumberValidation"
      serviceClass="com.ptc.windchill.enterprise.part.validator.CreateMultiPartNameNumberValidator"/>
</Service>
```

Loading the Wizard Step Content when it is Visited

By default, all of the wizard steps are downloaded at wizard execution time.

However, the wizard can be configured so each wizard step is loaded only when visiting that step.

This feature may be needed when a step is dependant on information gathered from a previous step.

To configure loading the step only when the step is visited, set the *preloadWizardPage* attribute of the action element to *false*:

```
<action name="setClassificationAttributesWizStep" preloadWizardPage="false">
 <command windowType="wizard_step"/>
</action>
```

Marking a Wizard Step as “required”

A “required” wizard step means the wizard cannot be submitted unless and until all required steps have been visited at least once.

Submit buttons (such as **OK**, **Finish** or **Apply**) would be enabled only after all required steps are visited at least once. Example code:

```
<action name="setClassificationAttributesWizStep" required="true">
 <command windowType="wizard_step"/>
</action>
```

A wizard step can also be marked as “required” at runtime.

Use a JavaScript function setStepRequired (defined in the *wizard.js* file).

Pass the *id* of the step, where the *id* format is “<type>.<action>”.

Use the value of the *id* attribute if it is defined explicitly while defining wizard step action. Example:

```
<script src="/netmarkets/javascript/components/wizard.js">
</script>
<table border="0">
  <tr>
    <td align="left" valign="top" NOWRAP>
      <w:radioButton id="copy"
         name="<%=NmObjectHelper.CHOICE%>"
         value="<%=NmObjectHelper.CB_COPY%>"
         checked="true"
         onclick="removeStep('defineItemWizStep ');"/>
    </td>
  </tr>
</table>
```

Hiding a Wizard Step

There are two ways to hide a wizard step:

1. hide a wizard step initially when the wizard is launched
2. hide the wizard step at run time depending on certain criteria or certain user actions

In both the cases, it is mandatory to register the wizard step while defining the wizard.

In the wizard JSP page:

```
<jca:wizard>
 <jca:wizardStep action="defineItemWizStep" id=”defineItemWizStep” type="object"/>
 <jca:wizardStep action="setClassificationAttributesWizStep" type="object"/>
 <jca:wizardStep action="affectedAndResultingItemsStep" type="changeTask" />
</jca:wizard>
```

Hide Wizard Step when Wizard is Launched

Use the *hidden* attribute of the action element. Example *actions.xml* file:

```
<action name="setClassificationAttributesWizStep" hidden="true">
 <command windowType="wizard_step"/>
```

```
</action>
```

Hide Wizard Step at Runtime

Use a JavaScript function called removeStep which is defined in the *wizard.js* file.

Pass the *id* of the step that needs to be removed as an argument.

The default id of the step is in the format “<type>.<action>” format.

Use the value of the *id* attribute if it is defined explicitly while defining wizard step action. If the wizard step JSP page has the following code:

```
<jca:wizard >
 <jca:wizardStep action="defineItemWizStep" id=” defineItemWizStep” type="object"/>
 <jca:wizardStep action="setClassificationAttributesWizStep" type="object" />
 <jca:wizardStep action="affectedAndResultingItemsStep" type="changeTask" />
</jca:wizard>
```

In *actions.xml*:

```
<action name="setClassificationAttributesWizStep" hidden="true">
 <command windowType="wizard_step"/>
</action>
```

The wizard step JSP page:

```
<script src="/netmarkets/javascript/components/wizard.js">
</script>
<table border="0">
 <tr>
  <td align="left" valign="top" NOWRAP>
   <w:radioButton id="copy" name="<%=NmObjectHelper.CHOICE%>"
     value="<%=NmObjectHelper.CB_COPY%>"
     checked="true"
   onclick="insertStep('defineItemWizStep');"/>
  </td>
 </tr>
</table>
```

Providing User-defined “Submit” Function

User-defined JavaScript functions can be called when the wizard is submitted.

1. Write a custom JavaScript submit function.
2. Make the JavaScript code must be accessible to the wizard (inside the wizard or as an include directive).
3. Pass the name of that function as an argument to setUserSubmitFunction (provided by wizard framework).

By doing this, the wizard framework is prevented from calling the default submit function.

Based on the return value (true / false), the wizard framework will determine the next action.

Example JavaScript code:

```
<script language="javascript">
  function user_validate() {
    // Do something
    return true;
  }
  setUserSubmitFunction(user_validate);
</script>
```

Client-side Validations Before a Wizard Step is Displayed

User-defined JavaScript functions can be called when a wizard step is loaded, but before it is displayed.

1. Write a custom JavaScript function.
2. Make the JavaScript code must be accessible to the wizard (inside the wizard or as an include directive).

3. specify the name of function as the value for the attribute beforeJS while defining the action for that particular step.

This particular step will only be displayed if the specified function returns “true”, otherwise the wizard will move to the previous step. Example *actions.xml*:

```
<action name="setClassificationAttributesWizStep" afterJS ="validateStep">
 <command windowType="wizard_step"/>
</action>
```

Example JavaScript code:

```
<script language="javascript">
  function validateStep() {
    // Do something
    return true;
  }
</script>
```

Exercise 10-1: Build a Create Wizard

Objectives

- Update an existing sub tab to implement a table that displays only a soft type of Parts.
- Add a custom tool bar with a custom icon.
- Create a **Create** wizard launched from the custom tool bar.

Scenario

A customer has requested a **Create** wizard for a specific soft type.

Detailed Description

1. There will be a sub tab to display only objects of the soft type. This table will :
   a. have a tool bar with one icon to launch a **Create** wizard
   b. display a soft attribute of the type
   c. show all objects only of a specific soft type
2. The create wizard will have three steps/screens:
   a. General Instructions
   b. Context Selection
   c. Attribute Definition

Step 1. Create a String IBA named **`MfgPlant`**.

a. Open **Site > Utilities > Type and Attribute Manager**.
b. Select the **Attribute Definition Manager** tab.
c. If an Organizer named “Business” does not exist, create it.
d. Under the **Business** attribute organizer, create a soft attribute named **`MfgPlant`** with **Type** equal to **String**.

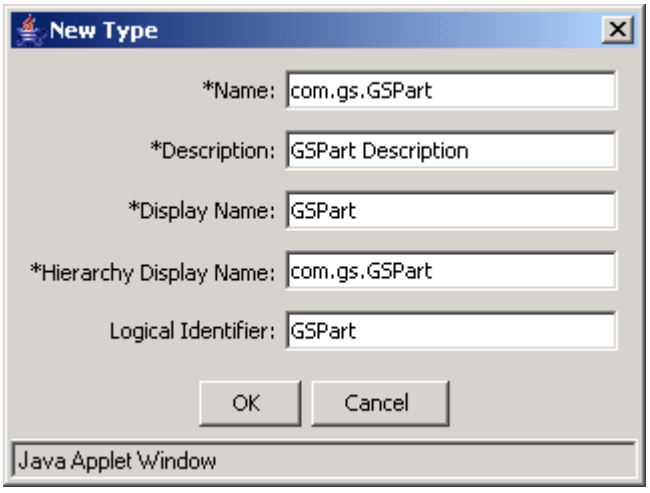
Example:

e. The *logical id* should be the same as the attribute name.
f. Select **OK**.

Step 2. Create a custom Part.

a. Open **Site > Utilities > Type and Attribute Manager**.
b. Select the **Type Manager** tab.
c. Highlight **Part** and select the create icon.
d. For **Name**, enter **`GSPart`**.
e. The package name will vary depending upon the organization; be consistent throughout the remainder of the exercise to use the package appropriate for your installation/organization.

Example:



Create GSPart

f. Add the **MfgPlant** IBA to the **Template > Attribute Root**.
g. Add a **Discrete Set Constraint** to **MftPlant** with three value: **`USA`**, **`China`** and **`Taiwan`**.
h. Add a **Value Required Constraint** to **MftPlant**.
i. Set the **Value** to **`USA`**.
j. Save and check in **GSPart**.

> If at any time in the UI GSPart instances are not visible — check the view configuration.

Step 3. Create an Info*Engine Task and a corresponding Task Delegate to search for a GSPart objects.

a. Copy *WT_HOME/tasks/com/gsdev/client/demo-search.xml* to *WT_HOME/tasks/com/gsdev/client/demo-searchGSParts.xml*.
b. Edit *demo-searchGSParts.xml* to search for all GSPart objects by logical id (GSPart) and return all attributes:

Example:

```
<?xml version="1.0" standalone="yes"?>
<%@taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<ie:webject name="Search-Objects" type="OBJ">
  <ie:param name="ATTRIBUTE" data="name,number,MfgPlant" delim=","/>
  <ie:param name="INSTANCE"  data="local.ptc.windchill.Windchill"/>
  <ie:param name="WHERE"     data="()"/>
  <ie:param name="TYPE"      data="GSPart"/>
</ie:webject>
```

c. Make sure the *INSTANCE* parameter is correct for your Windchill installation.
d. Save the file and test it in a browser (http://localhost/Windchill/servlet/IE/tasks/com/gsdev/client/demo-searchGSParts.xml).

Step 4. Implement a Task Delegate.

a. Select **Site > Utilities > Task Delegate Administrator**.
b. Enter the LDAP Administrator credentials. Ask the instructor for help, if necessary.
c. Select **Create Delegate**.
d. For **Repository Type**, select **com.ptc.windchill**.
e. For **Name**, enter **`demo-searchGSParts`**.
f. For **Source URL**, enter **`com/gsdev/client/demo-searchGSParts.xml`**.
g. For **Type Identifier**, select **WCTYPE|wt.fc.Persistable**.
h. Select **Create**.

Step 5. Create the navigation model for the tool bar.

a. There will be only one action, for now — **Create**.
b. Add the following code in the *custom-actionModels.xml* file

Example:

```
<model name="custom tool bar">
  <action name="doCustomCreate" type="GSPart"/>
</model>
```

Step 6. Register the actions.

a. Edit *custom-actions.xml*.
b. Create an objecttype element for GSPart objects.

Example:

```
<objecttype name="GSPart" class="wt.part.WTPart|com.gs.GSPart">

</objecttype>
```

c. Add an action to represent the create icon.
Example:

```
<objecttype name="GSPart" class="wt.part.WTPart|com.gs.GSPart">
  <action name="doCustomCreate">
    <command windowType="popup"
             class="com.ptc.windchill.enterprise.part.forms.CreatePartFormProcessor"
             method="execute"/>
  </action>
</objecttype>
```

d. Add actions to represent the steps in the wizard.
Example:

```
<objecttype name="GSPart" class="wt.part.WTPart|com.gs.GSPart">
  <action name="doCustomCreate">
    <command windowType="popup"
             class="com.ptc.windchill.enterprise.part.forms.CreatePartFormProcessor"
             method="execute"/>
  </action>
  <action name="firstPage">
    <command windowType="wizard_step"/>
  </action>
  <action name="secondPage">
    <command windowType="wizard_step" url="/netmarkets/jsp/object/setContextWizStep.jsp"/>
  </action>

</objecttype>
```

The second page will reference an existing OOTB component.

e. Add an icon for the create action.
f. Save the file.
g. Reload the action and model configuration.

Step 7. Create an icon for the **Create** action.
a. Copy *WT_HOME/codebase/wtcore/images/part_create.gif* to *WT_HOME/codebase/com/gsdev/client/images/gspart_create.gif*.

Step 8. Generate the resource bundle.
a. Create *WT_HOME/src/com/gsdev/client/GSPart.rbInfo*:
Example:

```
ResourceInfo.class=wt.tools.resource.StringResourceInfo
ResourceInfo.customizable=true
ResourceInfo.deprecated=false

GSPart.doCustomCreate.description.value=Create GSPart
GSPart.doCustomCreate.icon.value=../../com/gsdev/client/images/gspart_create.gif
GSPart.firstPage.description.value=First Page
GSPart.secondPage.description.value=Second Page
```

b. Build the resource bundle.
c. Restart the Method Server and Tomcat.

Step 9. Create the sub tab JSP.
a. The sub tab will open a table of custom parts.

b. Edit *WT_HOME/codebase/netmarkets/newTab/list6.jsp*.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>

<jca:describeTable var="tableDescriptor" id="custom.gspart.list" label="GSParts"
                   configurable="false" toolbarName="custom tool bar">
  <jca:describeColumn id="name" isInfoPageLink="true" sortable="true"/>
  <jca:describeColumn id="number" isInfoPageLink="true" sortable="true"/>
  <jca:describeColumn id="infoPageAction" sortable="false"/>
  <jca:describeColumn id="nmActions" sortable="false"/>
  <jca:describeColumn id="thePersistInfo.modifyStamp" sortable="true"/>
  <jca:describeColumn id="thePersistInfo.createStamp" sortable="true"/>
  <jca:describeColumn id="MfgPlant" need="MfgPlant" label="Plant" sortable="true"/>
</jca:describeTable>

<jca:getIeModel var="tableModel" descriptor="${tableDescriptor}"
                action="demo-searchGSParts"/>

<jca:renderTable model="${tableModel}"/>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

c. How many labels are explicitly defined in this file? What should be used in production?

d. Save the file.

e. There is no wizard yet, but the basic configuration can be tested — test it.

Step 10. Create the Wizard JSP.

a. Based on the *doCustomCreate* action definition, where is the page located?

b. Edit *WT_HOME/codebase/netmarkets/jsp/GSPart/doCustomCreation.jsp*

Example:

```
<%@ taglib prefix="jca" uri="http://www.ptc.com/windchill/taglib/components"%>

<%@ include file="/netmarkets/jsp/components/beginWizard.jspf"%>
<%@ include file="/netmarkets/jsp/components/includeWizBean.jspf"%>

<jca:initializeItem operation="${createBean.create}"
                    baseTypeName="WCTYPE|wt.part.WTPart|com.gs.GSPart"/>

<jca:wizard buttonList="DefaultWizardButtonsNoApply">
  <jca:wizardStep action="firstPage"  type="GSPart" label="1st Page"/>
  <jca:wizardStep action="secondPage" type="GSPart" label="2nd Page: Select Context"/>
</jca:wizard>

<%@include file="/netmarkets/jsp/util/end.jspf"%>
```

c. Save the file.

Step 11. Create the Wizard Step JSP.

a. Create *WT_HOME/codebase/netmarkets/GSPart/firstPage.jsp*.

b. Add the following content:

Example:

```
<%@ taglib uri="http://www.ptc.com/windchill/taglib/components" prefix="jca"%>
<%@ taglib prefix="wctags" tagdir="/WEB-INF/tags"%>

<%@ include file="/netmarkets/jsp/util/begin.jspf"%>
<%@ include file="/netmarkets/jsp/components/includeWizBean.jspf"%>

<jca:describeAttributesTable var="attributesTableDescriptor"
   id="createSetAttributes" mode="CREATE"
      componentType="WIZARD_ATTRIBUTES_TABLE"
   type="WCTYPE|wt.part.WTPart|com.gs.GSPart" label="Set Attributes"
     scope="request">
  <jca:describeProperty id="number"/>
  <jca:describeProperty id="name"/>
  <jca:describeProperty id="ALL_SOFT_SCHEMA_ATTRIBUTES"/>
</jca:describeAttributesTable>

<%@ include file="/netmarkets/jsp/components/setAttributesWizStep.jspf"%>

<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

c. Each JSP file for any step would be a similar format.

d. Save the file.

Step 12. Test it.

Wizard Processing

Overview

You have created a JSP wizard to gather information from a user about one or more object(s). You now need to create the code to process that information and perform a database operation(s) on the object(s).

If your wizard uses one of the built-in button sets, when a user clicks the Finish, Apply, Save, or Check In button to submit the form, a javascript function is called that invokes the doPost() method of the WizardServlet. The WizardServlet loads the HTTP form data and other wizard context information, such as where the wizard was launched, into a NmCommandBean. It then passes the NmCommandBean to the FormDispatcher class. The FormDispatcher performs an RMI call to a FormProcessorController class in the MethodServer.

The FormProcessorController partitions the form data into ObjectBeans. One ObjectBean is created for each target object of the wizard. It contains all the form data specific to that object and any form data that is common to all objects. The FormProcessorController then passes the ObjectBeans to classes called ObjectFormProcessors that perform the tasks appropriate to the wizard — for example, creating a new object in the database, updating an object in the database, or checking in an object. ObjectFormProcessors, in turn, can call classes called ObjectFormProcessorDelegates to perform one or more subtasks.

If your wizard is performing an operation on a single object you may need to create your own ObjectFormProcessor and/or ObjectFormProcessorDelegates to perform the tasks specific to your wizard. However, if your wizard is creating or editing an object, you may be able to take advantage of some processors that are delivered with the product for those purposes.

If your wizard has multiple target objects you may or may not also need to create your own FormProcessorController to control the order in which objects are processed.

The solution has the following elements:

- *WizardServlet* - Java class that uns in the servlet container
  - This is the class to which wizard form data gets posted and which sends the response page sent back to the browser after processing completes.
  - Runtime location:
    `<WT_HOME>/srclib/CommonComponents-web.jar`
- *FormProcessorController* Java interface which when implemented runs in the Method Server
  - Classes implementing this interface instantiate and call ObjectFormProcessor(s) to execute wizard tasks.
  - Runtime location: *<WT_HOME>/srclib/CommonComponents.jar*
- *ObjectFormProcessorDelegate* Java interface
  - Classes implementing this interface are called by ObjectFormProcessors to perform processing subtasks. Multiple ObjectFormProcessorDelegates may be called by one processor and the same delegate may be used by multiple processors to handle a task common to multiple wizards.
  - Runtime location: *<WT_HOME>/srclib/CommonComponents.jar*
- *DefaultObjectFormProcessorDelegate* A default implementation of ObjectFormProcessorDelegate.
  - This provides no-op behavior for methods a subclass may not need to implement. This is the base class that should be extended by task-specific delegates.
  - Runtime location: *<WT_HOME>/srclib/CommonComponents.jar*

- *ObjectBean* A container for the form data specific to a specific target object and the data common to all objects
  - Provides methods to retrieve the form data for a given object
  - Runtime location: *<WT_HOME>/srclib/CommonComponents.jar*

- *ProcessorBean* A container for ObjectBeans that knows which ObjectBeans should be processed by the same processor instance and the order in which they should be processed.
  – Runtime location: *<WT_HOME>/srclib/CommonComponents.jar*
- *FormResult* Class used to pass method results between server methods and from the server to the WizardServet
  – Runtime location: *<WT_HOME>/srclib/CommonComponents.jar*

The relationship between the the main Java classes in the wizard processing framework is shown in the UML diagram below.

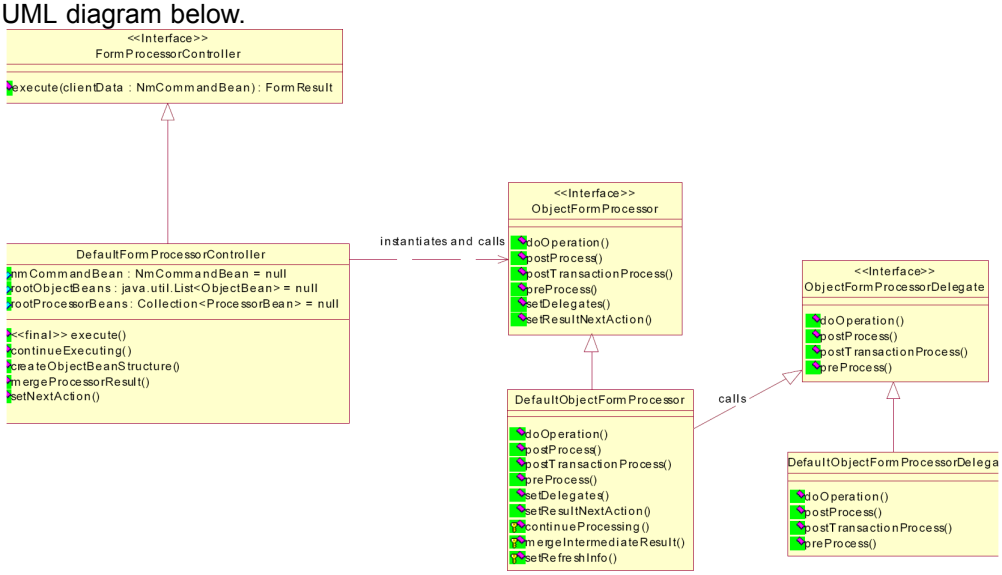


Figure 10-1: UML Model of the Form Processing Framework

Form Processing Task Flow

These typically follow the following sequence:

1. Preprocessing
2. Start a database transaction block
3. Do the database operation.
4. Postprocessing
5. End the database transaction block
6. Post-transaction processing

The tasks in each processing phase are performed by ObjectFormProcessors and ObjectFormProcessorDelegates. These classes have preProcess(), doOperation(), postProcess(), and postTransactionProcess() methods for this purpose. Not every wizard will have all of the above tasks, so it is not necessary that every processor and every delegate implement all of these methods, if they extend from the default classes DefaultObjectFormProcessor and DefaultObjectFormProcessorDelegate, respectively. If an ObjectFormProcessor does implement one of these methods, it should call the super() method of the DefaultObjectFormProcessor, which will handle the calling of ObjectFormProcessorDelegates.

Note: ObjectFormProcessors should not open/commit additional transaction blocks in steps 3 or 4 as nesting of transactions is not recommended.

The HTML form data will be passed to ObjectFormProcessors and ObjectFormProcessorDelegates in a list of ObjectBeans. An ObjectBean contains the data specific to one target object and the data common to all the target objects. For wizards with a single target object the list will contain only one ObjectBean. For wizards with multiple target objects the list will contain one ObjectBean for each target object and the ObjectBeans may be organized into a tree structure representative

of the relationship between the objects and the order in which they should be processed. The creation of ObjectBeans is handled by the FormProcessorController.

The FormProcessorController also handles the opening, closing, and, if necessary, rollback of the database transaction and the calling of the ObjectFormProcessors. For the latter, the DefaultFormProcessorController uses objects called ProcessorBeans. Target objects with the same parent object, which are of the same type, and which have the same ObjectFormProcessorDelegates are placed into the same ProcessorBean. Each ProcessorBean also will have it's own instances of the ObjectFormProcessor and ObjectFormProcessorDelegates for the ObjectBeans it contains. ProcessorBeans may be organized into a tree structure to control the order in which objects are processed. (Wizards with a single target object will have only one ProcessorBean.)

The task flow of the DefaultFormProcessorController is as follows:

1. Call the preProcess() method of ObjectFormProcessor for the root ProcessorBean, passing it the ObjectBeans in the ProcessorBean. Then call the preProcess() method of the processors for the children of the root ProcessorBean, in the order in which they appear in the child list. Then call the preProcess() method for the children of the children, and so forth.
2. Do Transaction.start()
3. Call the doOperation() method of the ObjectFormProcessor for the root ProcessorBean and its children in the same way as preProcess().
4. Call the postProcess() method of the ObjectFormProcessor for the root ProcessorBean and its children in the same way as preProcess().
5. If steps 1-4 are successful, do Transaction.commit()
6. Call the postTransactionProcess() method of the ObjectFormProcessor for the root ProcessorBean and its children in the same way as preProcess().

If any method returns a status of FormProcessingStatus.FAILURE, the controller will call the setResultNextAction() method of the ObjectFormProcessor that failed so it can set information needed for the HTML response page.

Wizards with Multiple Target Objects

Two types of multiple-object wizards are supported:

- Multiple unrelated target objects e.g.: create multiple parts
- Tree of related target objects e.g.: create a change notice and related change tasks

This is covered in detail in the Customizer's Guide.

Procedure - Creating Your Wizard Processing Code

For Wizards with a Single Target Object the process consists of:

1. Create your processor class
2. Specify the processor class for the wizard on the wizard action
3. Create any necessary ObjectFormProcessorDelegate classes for your wizard
4. Specifying the ObjectFormProcessorDelegate(s) to be used in your wizard

Note: The direction by Product Management is to avoid using Rational Rose for modelling UI components. Extensions described here should be performed manually when creating descendant classes.

Create your processor class

To help create your processor class, three processors for handling object creation and editing wizards are delivered with the product:

1. com.ptc.core.components.forms.CreateObjectFormProcessor
2. com.ptc.core.components.forms.DefaultEditFormProcessor
3. com.ptc.core.components.forms.EditWorkableFormProcessor

If your wizard is not an object creation or editing wizard you will need to create your own ObjectFormProcessor. ObjectFormProcessors should extend the DefaultObjectFormProcessor class.

You should place your form processing logic into the preProcess(), doOperation(), postProcess(), and postTransactionProcess() methods of the processor, as appropriate. Your methods should call the corresponding super method of the DefaultObjectFormProcessor, which will handle the calling of ObjectFormProcessorDelegates. These methods will be passed a single ObjectBean, which will contain all the form data from the wizard. The form data can be accessed using the getter methods of that object. The following getter methods are commonly used:

```
public Map<String,List<String>> getChangedComboBox()
public Map<String,String> getChangedRadio()
public Map<String,String> getChangedText()
public Map<String,String> getChangedTextArea()
public Map<String,List<String>> getChecked()
public Map<String,List<String>> getUnChecked()
public List getRemovedItemsByName(String paramName)
public List getAddedItemsByName(String paramName)
public String getTextParameter(String key)
public String[] getTextParameterValues String key)
```

The "object" attribute of the ObjectBean represents an instance of the target object . Where appropriate, it should be set by one of your processor methods (most likely preProcess()) for use by downstream methods. Other information can be passed from one method to another using processor instance variables.

The NmCommandBean object passed to these methods contains information about the page from which the wizard was launched and the object selected on the parent page when the wizard was launched. It also contains all the HTML form data but you should use the methods on the ObjectBean rather than the NmCommandBean to access that data. See the javadoc for NmCommandBean for more information.

You pass the outcome of the preProcess(), doOperation(), postProcess(), and postTransactionProcess() methods back to the DefaultFormProcessorController using the com.ptc.core.component.FormResult object. Before returning, each of these methods should call FormResult.setStatus() to return the processing status. Three options are available:

- FormProcessingStatus.SUCCESS - if the method executed without error
- FormProcessingStatus.FAILURE - if the method encountered a fatal errors FormProcessingStatus.NON_FATAL_ERROR - if the method succeeded but encountered one or more problems that should be reported to the user.

The DefaultFormProcessorController passes the returned FormResult to its continueExecuting() method to determine whether processing should continue to the next phase or be aborted. By default, it will abort only if the status is FormProcessingStatus.FAILURE. If processing is to be aborted or after all processing competes successfully, the controller will call the setResultNextAction() method of the ObjectFormProcessor to set information in the FormResult needed to construct the response page sent back to the browser.

This method should convey the following information:

- Whether the wizard window should be closed. Determined from the status and nextAction variables.
- Whether the window from which the wizard was launched should be refreshed. Determined from the nextAction variable
- The javascript you would like executed in the response, if any. Determined from the nextAction and javascript variables.
- Whether a new URL should be loaded into the launch window and, if so, what that URL should be loaded. Determined from the nextAction and URL variables
- What feedback messages should be displayed to the user, if any. Determined from the feedbackMessages and exceptions variables.

Feedback messages, if any, are displayed before executing any window operations or provided javascript

Exeception messages are only displayed if status is FormProcessingStatus.FAILURE or FormProcessingStatus.NON_FATAL_ERROR.

See the javadoc for the FormResult, FormProcessingStatus, and FormResultAction classes for more information.

It is also possible for your ObjectFormProcessor's preProcess(), doOperation(), postProcess(), and postTransactionProcess() methods to throw exceptions. In that case, control will be returned to the WizardServlet which will set the variables of the FormResult as follows:

- status - FormProcessingStatus.FAILURE
- nextAction - FormResultAction.NONE
- exceptions - the thrown Exception

This will cause the response page to display the exception message in an alert window and then close the wizard window.

Specify the processor class for the wizard on the wizard action

You specify the ObjectFormProcessor class in the <command> subtag of the <action> tag for the wizard. Your action tag will be contained in a *actions.xml file. Here is an example of how you would specify the CreateDocFormProcessor class as your processor.:

```
<action name="create">
 <command
    class="com.ptc.core.components
        .forms.CreateObjectFormProcessor"
    windowType="popup" />
</action>
```

Create any necessary ObjectFormProcessorDelegate classes for your wizard

ObjectFormProcessorDelegates are used to carry out one or more subtasks in your wizard processing. Because the same delegate class may be called by multiple ObjectFormProcessors, they are typically used for tasks that are needed by multiple wizards. Here are some examples of how ObjectFormProcessorDelegates are used in the delivered product:

- Several object creation wizards have a checkbox named "Keep checked out after checkin." The ObjectFormProcessors for all these wizards call the same ObjectFormProcessorDelegate class to handle this checkbox and checkout the object after it is created if the box is checked.
- Wizards to create objects that are ContentHolders typically have a Set Attachments step to specific the documents that should be attached to the object. All these wizards call the same ObjectFormProcessorDelegate class to handle the persistence of the attachments.
- Many object creation wizards have an input field for a Location attribute to specify the object's folder. Because the processing of this input field is complex all these wizards call the same ObjectFormProcessorDelegate to set the folder on the object being created.

As shown in the examples above, a ObjectFormProcessorDelegate can handle the processing of one or many input fields. If your wizard does not have any HTML elements that are used in multiple wizards you may not need any delegates. However, they can be useful for modularizing your processing code also.

ObjectFormProcessorDelegates should extend the class DefaultObjectFormProcessorDelegate. ObjectFormProcessorDelegates are instantiated by the DefaultFormProcessorController and passed to the ObjectFormProcessor.

ObjectFormProcessorDelegates have preProcess(), doOperation(), postProcess(), and postTransactionProcess() methods just like ObjectFormProcessors. The delegates registered for the wizard will be called by the DefaultObjectFormProcessor during the different processing phases..

The outcome of an ObjectFormProcessorDelegate method is passed back to the ObjectFormProcessor using a FormResult, just as the ObjectFormProcessor methods pass back their results to the FormProcessorController. Typically, a delegate will only pass back a status and, possibly, feedback messages in the FormResult. The ObjectFormProcessor will set and pass

back the nextAction in the FormResult it sends to the FormProcessorController since it knows the launch context of the specific wizard in which the delegate is being used

Like ObjectFormProcessor methods, ObjectFormProcessorDelegate methods can throw exceptions. How these are handled depends on the ObjectFormProcessor calling it.

Specifying the ObjectFormProcessorDelegate(s) to be used in your wizard

The names of the ObjectFormProcessorDelegate classes to be instantiated by the DefaultObjectFormProcessor, if any, are communicated in hidden input fields as follows:

```
<input
  name="FormProcessorDelegate"
  value="com.ptc.core.components.
       forms.NumberPropertyProcessor"
  type="hidden"
>
```

These hidden input fields can be created in several ways:

- If you have a delegate associated with a specific wizard step, the delegate can be specified in the command subtag of in the wizard step action. The wizard framework will then generate a hidden FormProcessorDelegate field in the wizard for any wizard using this step. If you have specified an object handle on the wizard step action, the name attribute of the hidden field will include the object handle so that the delegate will be called only for the target object associated with the step.

```
<action
   name="attachmentsWizStep"
   postloadJS="preAttachmentsStep"
   preloadWizardPage="false"
 <command
    class="com.ptc.windchill.
          enterprise. attachments.forms.
          SecondaryAttachmentsSubFormProcessor"
    windowType="wizard_step"/>
</action>
```

- If you have a specific input field that requires a delegate you can generate the hidden field in the data utility that creates the gui component for the input field. It is recommended that the data utility return a subclass of AbstractGuiComponent to take advantage of the addHiddenField() method and the AbstractRendererAfter creating the gui component in the data utility, call the method addHiddenField() in the AbstractGuiComponent class. as shown in the example below. The hidden input field will begenerated for you by the AbstractRenderer automatically. If the field is associated with a step or table row that has an object handle, that object handle will be embedded in the HTML name attribute of the hidden field. If you choose to return a gui component that does not extend Abstract GuiComponent, your gui component and renderer would have to know how to render the necessary hidden field.

```
LocationInputGuiComponent guiComponent =
   new LocationInputComponent(…);
guiComponent.addHiddenField(
   CreateAndEditWizBean.FORM_PROCESSOR_DELEGATE,
   "com.ptc.windchill.enterprise.folder.LocationPropertyProcessor");
```

- You can include a hidden field for your delegate directly in your JSP file. However, one of the first two methods is preferred because it encapsulates the hidden field with its associated HTML input fields.

Working with Multiple Target Objects

This process consists of the following steps:

1. Create your processor class
2. Specify the processor class for the wizard on the wizard action

3. Create any necessary ObjectFormProcessorDelegate classes for your wizard
4. Specify the ObjectFormProcessorDelegate(s) to be used in your wizard
5. Create a FormProcessorController, if needed
6. Specify the FormProcessorController to be used in your wizard, if needed

Refer to the Customizer's Guide for more detail on this topic.

Customization

Limitations exist on the customizations that can be created:

- Only one ObjectFormProcessor class per wizard is supported at this time
- The framework does not support having data for multiple objects in the same wizard step unless it is in tabular format. It also does not support having data common to all the objects on the same step as object-specific data.

Implementing Ajax in Windchill

1. Define and configure an action to perform an Ajax function.
   - Specify whether the Ajax refresh is against the row, table or partial pages.
   - Specify additional validation such as “confirmation prompts” prior to the action.
2. Create a delegate (essentially a Java program) that performs the database transaction.
   - It’s also possible to reference existing classes and methods.
3. Configure the JCA components in JSP files to partially refresh the page (row, table or portion of page).
   - Most existing JCA components will automatically understand using Ajax — it’s “built-in”.

Action Definition

To enable Ajax functions in Windchill, the action definition must be configured for Ajax.

- To refresh a row in a table, the action definition XML file will include an Ajax definition of ajax="row"

```
<action name="customAction" ajax="row">
  <command class="customClass" method="customRowMethod">
</action>
```

- To refresh the whole table, the action definition XML file will include an Ajax definition of ajax="component"

```
<action name="customAction" ajax="component">
  <command class="customClass" method="customMethod">
</action>
```

- To get the enter third-level navigation section to refresh when the action completes, the action will include an Ajax definition ajax=thirdLevelNav.

```
<action name="customAction" ajax="thirdLevelNav">
  <command class="customClass" method="customMethod">
</action>
```

Delegate

Existing delegates can be used, if appropriate, for common functions:

- Delete and refresh table
- Create and add row

Finding examples:

- Refer to the OOTB functions, using “jcaDebug=1” on the URLs, to identify existing actions to use as examples.
- Browse Carambola examples

Once an action is found, the Action Report page can identify the details of the action

- class
- method
- additional configuration such as confirmation prompts

Exercise 10-2: Implementing Ajax in Windchill

Objectives

- Add a custom action defined as an Ajax component refresh.
- Add the action to a custom table.
- Define the delegate to process the deletion.

Scenario

The customer requires a deletion action in a custom table.

Step 1. Test delete from the existing custom table.

a. Default row-specific actions were added to the table, including **Delete**.
b. This function will call the super class (wt.part.WTPart) delete.
c. After deleting a custom GSPart, does the table automatically update?
d. Do the surrounding tabs update?

Step 2. Find an appropriate delete command OOTB.

a. Open the search page.
b. Select the **Search** button.
c. Add **`&jcaDebug=1`** to the URL.
d. As you mouse-over the delete icon, the action displayed is search.list_delete.
e. There are 2 options at this point: either use the OOTB action search.list_delete directly, or identify it's properties and create a custom action with similar properties.

Step 3. Use the Action Report to get information about "search.list_delete".

a. Open the browser to the URL **`Windchill/netmarkets/jsp/carambola/tools/actionReport/action/jsp`**.
b. Based on the action format, the object type is "search" and the name is "list_delete".
c. For **Object Type:**, enter **`search`**.
d. For **Action Name:**, enter **`list_delete`**.
e. Select **Search**.
f. Open the properties for the action.
g. Note the class and method inside the command element.

Step 4. Add a custom action defined as an Ajax component refresh.

a. Edit *custom-actions.xml*.
b. For the *objecttype* named *GSPart*, add an action named **`deleteGSPart`**.
c. Specify **`ajax="component"`**. Inside the command element, for **class**, enter **`com.ptc.netmarkets.search.NmSearchCommands`**.
d. For **method**, enter **`list_delete`**.
e. Save the file.

Example:

```
<objecttype name="GSPart" class="wt.part.WTPart|com.gs.GSPart" resourceBundle="com.gsdev.client.GSP

  [...]

  <action name="deleteGSPart" ajax="component">
    <command class="com.ptc.netmarkets.search.NmSearchCommands" method="list_delete" />
  </action>
</objecttype>
```

Step 5. Add the icon and label for the delete action.

a. The delete action was added to an action definition that gets labels from a resource bundle.
b. Edit the resource bundle (*WT_HOME/src/com/gsdev/client/*) to include a label and action.

Example:

```
GSPart.deleteGSPart.description.value=Delete GSPart
GSPart.deleteGSPart.icon.value=../../com/gsdev/client/images/gspart_delete.gif
```

c. Create a delete icon or copy a similar icon from *WT_HOME/codebase/netmarkets/images*. For example, copy *WT_HOME/codebase/netmarkets/images/delete.gif* to *WT_HOME/codebase/com/gsdev/client/gspart_delete.gif*

d. Rebuild the resource bundle.

e. Restart Tomcat and the Method Server.

Step 6. Add the action to a custom table.

a. Edit *custom-actionModels.xml* :

Example:

```
<model name="custom tool bar">
  <action name="doCustomCreate" type="GSPart"/>
  <action name="deleteGSPart" type="GSPart"/>
</model>
```

b. Save the file.

c. Reload the action configuration.

Step 7. Test it.

a. Open the custom GSPart table. A second action should appear.

b. Does the table have the capability to select objects by row?

Step 8. Add the ability to “select by row” to the custom table.

a. If the capability does not already exist, update the JSP file that defines the custom table to include “select by row” functions.

b. Open *WT_HOME/codebase/netmarkets/newTab/list6.jsp*.

c. Add **`<jca:setComponentProperty key="selectable" value="true"/>`** to the **jca:describeTable** element.

d. Save the file.

e. Test again.

Step 9. Test again.

a. The delete should function.

b. There should NOT be a prompt to confirm the delete.

Step 10. Challenge: review the properties for the search.list_delete action and modify the deleteGSPart action to prompt for confirmation.

Step 11. Challenge: define a custom delegate to process the deletion.

a. Create *WT_HOME/codebase/com/gsdev/client/DeleteProcessor.java*..

b. Add a custom method named **`processDeleteGSParts()`**.

c. Use the method list_delete in the class *WT_HOME/codebase/com/ptc/netmarkets/search/NmSearchCommands* as an example.

d. Compile the class.

e. Modify the deleteGSPart action to use the custom class and method.

f. Reload the action configuration.

Summary

After completing this module, you should be able to:

- Create wizards
- Add custom validation to wizards.

Module

11

Information Pages

In this module, you will learn how to register default information pages for your custom objects

Objectives

Upon successful completion of this module, you will be able to:

- Create an object information page specific to a given business object

Lecture Notes

You may use the space below to take your own notes.

Design And Structure

- There are many types of business objects in Windchill, plus potentially many soft types or modeled object types introduced by customers.
- Each type of object can potentially have its own associated information page, a page that displays information around a selected instance of that type.
- While the information associated with different object types can vary widely, it is important to maintain a common look and feel among the multitude of information pages.
- The **Info Page** component was developed to give the Windchill products consistency for displaying an object details page.
- The component is also intended to make developing an Info Page much easier.
- Using the component, a developer only needs to configure certain parts of the page (list of actions, which attributes to show, etc..) while the main layout of the page is provided by the component.
- The **Info Page** component provides a common layout.
- Using the **Info Page** component will provide consistency between all objects and Windchill products.
- The specific content that is displayed for any given object type will be configured by overriding certain elements in the context of the specific object types.
- Configurable portions of the page include things like which action list is used and which attributes are shown in the top attribute panel.

Figure 11-1: Info Page Example

Info Page Characteristics

- Combination of a JSP page, bean, taglib, renderers and service call(s)
- Adhere to user interface standards
- Section 508 compliant per requirements
- Supports customization requirements
- Supports soft types and soft attributes
- Follows a standard format
- Supports the Object Icon including support for soft types and part sub-types
- Supports third-level Navigation stickiness (Windchill stores the last third-level navigation table a user was viewing)

Configurable Aspects of Info Page

In the default information page, only the name of the object, the object icon and a link to the action list would appear.

The configurable portions of the page are:

- Action list displayed in the top attributes panel
- Visualization area
- Status glyphs

- Help context
- Attributes displayed in the top attributes panel
- Third level navigation menus
- Actions contained in each third-level navigation menu
- Default third-level content table and table view

Implementation

Create the JSP

The custom Info page JSP file will have a describeInfoPage tag, a describePropertyPanel tag, and an include of the info page component, *infoPage.jspf*. Based on the configuration in the describeInfoPage and describePropertyPanel tags, the *infoPage.jspf* component will render the object icon, object identification, status glyphs, help link, action list, attributes panel (property panel), visualization component, third-level navigation and third-level navigation content. Before the include of infoPage.jspf, your jsp can set up what information is displayed on the page. (e.g. what attributes to show, whether or not visualization applies, what action model to use for third-level navigation content, etc.)

```
<jca:describePropertyPanel var="propertyPanelDescriptor">
  <jca:describeProperty id="name"/>
  <jca:describeProperty id="number"/>
</jca:describePropertyPanel>

<jca:describeInfoPage showVisualization="true"
   helpContext="part_details_help"
   navBarName="third_level_nav_part"
   propertyPanel="${propertyPanelDescriptor}">
  <jca:describeStatusGlyph id="statusFamily_Share"/>
  <jca:describeStatusGlyph id="statusFamily_General"/>
  <jca:describeStatusGlyph id="statusFamily_Change"/>
</jca:describeInfoPage>
```

Register the JSP

For the info page servlet to know which jsp to forward to for a custom type, an entry for the custom jsp needs to be specified in a properties file that is used by the type based service. The default properties file for this is *typedservices.properties* which can be found directly in codebase. The entries should be in the following format:

```
wt.services/rsc/default/com.ptc.netmarkets.util.misc.FragmentFactory/InfoPage/java.lang.Object/0
          =/netmarkets/jsp/object/info.jsp
wt.services/rsc/default/com.ptc.netmarkets.util.misc.FragmentFactory/InfoPage/<CLASS.PATH>/0
          =<jsp path relative to codebase>.jsp
```

xconf format

```
<Resource context="default"  name="com.ptc.netmarkets.util.misc.FilePathFactory">
  <Option requestor="<fully qualified class path>"
          resource="<jsp path relative to codebase>"
          selector="InfoPage"/>
</Resource>
```

Specify Status Glyphs to show in the Title Bar

```
<jca:describeInfoPage>
 <jca:describeStatusGlyph id="<Status Glyph Family Id>"/>
</comp:describeInfoPage>
```

Specify the Attributes to include

```
<jca:describePropertyPanel>
 <jca:describePropertyPanel var="propertyPanelDescriptor">
 <jca:describeProperty id="<attribute name | data utility id | logical form>" />
 <jca:describeProperty id="<attribute name | data utility id | logical form>" />
</jca:describePropertyPanel>

<jca:describeInfoPage propertyPanel="${propertyPanelDescriptor}">

</jca:describeInfoPage>
```

Note: To determine the set of available properties that can be displayed for an object type, use the property report

Exercise 11-1: Implement a Custom Info Page

Objectives

- Create **Info Page**.
- Implement **Info Page** for a custom soft type.

Scenario

Add a custom **Info Page** for GSParts.

Step 1. Register a custom **Info Page**.

a. Edit *custom-service.properties.xconf* to add an entry for a soft type **Info** page.

Example:

```
<Resource context="default" name="com.ptc.netmarkets.util.misc.FilePathFactory">
  <Option requestor="wt.part.WTPart|com.gs.GSPart"
          resource="/netmarkets/jsp/GSPart/info.jsp" selector="InfoPage"/>
</Resource>
```

b. Save the file.

c. Restart the Method Server and Tomcat.

Step 2. Create the **Info Page**.

a. Copy *WT_HOME/codebase/netmarkets/jsp/part/info.jsp* to *WT_HOME/codebase/netmarkets/jsp/GSPart/info.jsp*.

Step 3. Edit the custom **Info Page**.

a. Edit *WT_HOME/codebase/netmarkets/jsp/GSPart/info.jsp*.

b. Add a text tag to ensure that the correct page is opening.

Example:

```
<%@ include file="/netmarkets/jsp/util/begin.jspf" %>

<%@ taglib prefix="fmt" uri="http://www.ptc.com/windchill/taglib/fmt" %>
<%@ taglib prefix="jca" uri="http://www.ptc.com/windchill/taglib/components" %>

<%@ page import="com.ptc.windchill.enterprise.part.partResource" %>
<%@ page import="com.ptc.windchill.enterprise.object.dataUtilities.dataUtilitiesResource"
%>

THIS FILE IS GSPart/info.jsp

...
```

c. Add the soft attribute display to the describePropertyPanel tag.and disable the visualization panel.

> Although the example is truncated, the correct area to edit is at the bottom of the describePropertyPanel and should be clear from the example.

Example:

```
...

<!-- Define the contents of the part property panel. -->
<jca:describePropertyPanel var="propertyPanelDescriptor">
   <jca:describeProperty id="name" />
   <jca:describeProperty id="supplierId" />
   <jca:describeProperty id="lifeCycleState" />
   <jca:describeProperty id="checkoutInfo" />
   <jca:describeProperty id="pdmCheckoutStatus" need="containerName" isInfoPageLink  [...]
   <jca:describeProperty id="pdmCheckoutVersion" need="version" label="${pdmCheckedOutVersion}"/>
   <jca:describeProperty id="resultingPDMVersion" need="version" label="${resultingPDMVersion}"/>
   <jca:describeProperty id="modifyStamp" need="thePersistInfo.modifyStamp" label="$  [...]
   <jca:describeProperty id="compositePath" label="${locationLabel}">
       <jca:setComponentProperty key="COMPOSITE_PATH"  value="${compositePath}"/>
   </jca:describeProperty>
   <jca:describeProperty id="classification" label="${classificationLabel}" />
   <jca:describeProperty id="endItem" />
    <jca:describeProperty id="defaultTraceCode"/>
   <jca:describeProperty id="genericType" />
   <jca:describeProperty id="templateOfGeneric" />
   <jca:describeProperty id="MfgPlant" />
</jca:describePropertyPanel>

...
```

d. Disable the visualization panel.

Example:

```
...
<!-- Define the contents of the part info page. -->
<jca:describeInfoPage showVisualization="false"
                      helpContext="part.view"
                      navBarName="third_level_nav_part"
                      propertyPanel="${propertyPanelDescriptor}">
   <jca:describeStatusGlyph id="statusFamily_Share" />
   <jca:describeStatusGlyph id="statusFamily_General" />
   <jca:describeStatusGlyph id="statusFamily_Change" />
</jca:describeInfoPage>

<%@ include file="/netmarkets/jsp/components/infoPage.jspf" %>
<%@ include file="/netmarkets/jsp/util/end.jspf" %>
```

More Info Page Features

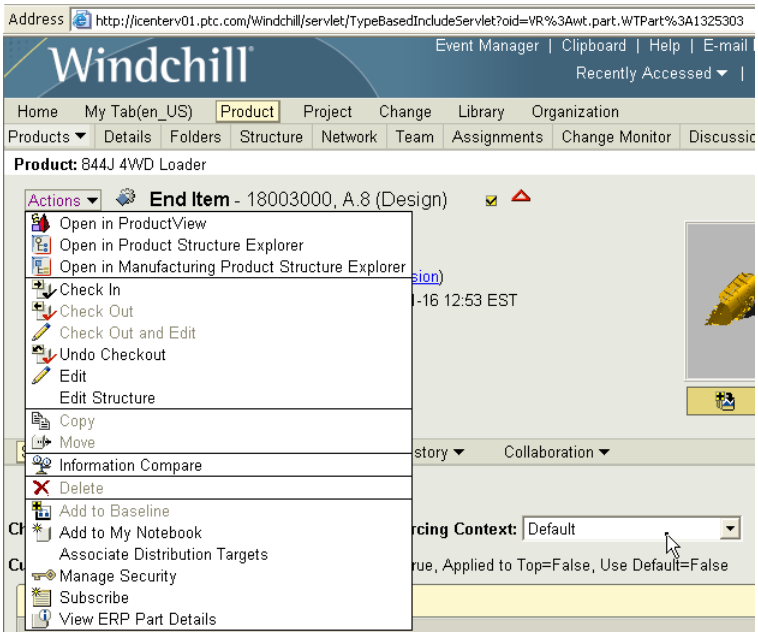


Figure 11-2: Info Page Actions List

Configure Action List Actions

The actions that apply for a particular object type are configured in *actionModels.xml*.

To specify which action model to use as the actions list for a specific class or soft type, use the attribute *menufor*:

```
<model
name="An_Action_Model_Name>" menufor="Fully_Qualified_Type_Name">
  <action name="view" type="object"/>
</model>
```

Multiple classes or soft types can be in a comma separated list:

```
<model name="more meeting
actions" menufor="wt.meeting.TraditionalMeeting,wt.meeting.MeetingCenterMeeting">
 <action name="host" type="meeting"/>
  <action name="join" type="meeting"/>
  <action name="add_minutes" type="meeting"/>
</model>
```

Info Page Overriding Default Action Menu

An **Info Page** can override the default registered in *actionModels.xml* by configuring the describeInfoPage tag with a different *actionListName* :

```
<jca:describeInfoPage actionListName="<action model name>">
</jca:describeInfoPage>
```

If there should be no action list, set actionListName equal to an empty string.

Hide/Show Visualization

Visualization display can be configured in an**Info Page** with the *showVisualization* element:

```
<jca:describeInfoPage showVisualization="true" >
</jca:describeInfoPage>
```

Specify a Third-level Navigation Menu Bar

The third-level navigation bar is defined by *navBarName* attribute of the describeInfoPage element:

```
<jca:describeInfoPage navBarName="<action model name>" >
…
</jca:describeInfoPage>
```

Configure the Third-level Navigation Menus

The following is an example configuration of a third-level navigation menu for the Part **Info Page**:

```
<model name="third_level_nav_part">
 <action name="productStructure" type="object"/> <!-- Structure -->
 <submodel name="general"/> <!-- General -->
 <submodel name="relatedItems"/> <!-- Related Objects -->
 <submodel name="history"/> <!-- History -->
 <submodel name="collaboration"/> <!-- Collaboration -->
</model>
```

Third-level navigation menus are either:

- actions
  - An action in the third-level navigation bar will be displayed as just a link. (e.g., **Product Structure** browser).
- “submodels”
  - A submodel in the third-level navigation bar will be displayed as a drop down menu (e.g., **Related Items** menu).
  - Submodels are defined in *Navigaton-actionmodels.xml*.

The **General**, **Related Items**, **History** and **Collaboration** menus are meant to be used by all object types; actions that do not apply for certain object types will be removed via action validation.

Summary

After completing this module, you should be able to:

- Create an object information page specific to a given business object

Module

12

Incorporating Pickers

In this module, “pickers” are introduced as a way to enter information, e.g., during object creation.

Objectives

Upon successful completion of this module, you will be able to:

- Configure an OOTB property picker in some wizard step.
- Manage interaction between that page and the picker.
- Configure a context picker to pick context(s) to be used in your application.
- Configure an Item Picker.
- Configure an Organization Picker.
- Configure a User Picker.
- Configure a Participant Picker.
- Add new customized type to search item picker.
- Display a picker for configuration specifications.

Lecture Notes

You may use the space below to take your own notes.

Picker Interaction

There are many types of business objects in Windchill, plus potentially many soft types or modeled object types introduced by the user. These types would predominantly would have some relationship with the other types. There are numerous instances when user creates or updates objects of such types using an user interface, he would need to specify the property value(s) derived from the other available objects of the same or different type. e.g. The business object like Part or Document are owned by a specific organization and hence at the time of their creation user needs to be able to specify an owning organization.

This sort of user interaction is achieved through the use of property picker. The property picker simply aids in picking up a property from an existing business object. The user would pick an object by launching the picker and have it's pro properties used to update visible and/or non-visible properties on the page. The following sketch illustrates this sort of interaction pictorially.

Assume you have a customized type <myObjectType> object needing to be created. On the first step of the create wizard, you need to define a relation ship with an object of type <someObjectType>. This spawns a need to let user select the object he desires from a picker and have the attributes of these objects used to populate some content on the first step of the create wizard.

Picker Solution Elements

- picker.tld - Tag Library Descriptor (TLD) file, which contains Picker related Tag definitions.
  - The 'propertyPicker, populate and pickerParam are the ones needed to be used to implement this design concept.
  - Run time Location: *WT_HOME/codebase/WEB-INF/tlds*
- create_step_<step no>.jsp - JSP providing the content for the specified step of the Create Wizard .
  - This step would include the controls to launch a picker and populate values from the picked object
  - This JSP could as well render a javascript picker callback function to be executed once the picker 'OK' action is executed
  - Run time Location: *WT_HOME/codebase/netmarkets/ <myObjectType>/create_step_<stepno>.jsp*
- <someObjectType>_picker.jsp JSP - JSP defining a single step wizard with 'OK' and 'Cancel' actions for the wizard
  - The wizard step would render a content that presents a list of <someObject> in table or tree component
  - The user would select the object of interest by ticking the check boxes to select row(s) in the table
  - Run time Location: WT_HOME/codebase/netmarkets/< someObjectType >/<someObjectType>_picker.jsp
- <someObjectType>_picker_step.jsp JSP defining the content for the picker wizard step.
  - This step can contain a table listing the objects of <someObjectType> or search criteria along with
  - Run time Location: WT_HOME/ codebase/netmarkets/< someObjectType >/ <someObjectType>_picker_step.jsp

- Actions.xml - XML file with definitions of all the actions.
  - The action that launches picker need to get defined for the type <myObjectType>
  - The picker wizard step actions as well get defined in this file
- picker.jspf - JSP fragment to be always included by the picker implementation
  - i.e. in <someObjectType>_picker_step.jsp
  - Run time location: *WT_HOME/codebase/netmarkets/components/ picker.jspf*

- defaultPickedDataFetcher.jsp - Default Picker Data Fetcher JSP available in the product
  - uses the ‘oids’ passed to get the specified attributes
  - The ‘oid’ and attribute information is compiled into a string as JSON (http://en.wikipedia.org/wiki/JSON) representation to be sent back as http response
  - Run time location: *WT_HOME/codebase/netmarkets/components/defaultPickedDataFetcher.jsp*

Default Data Fetcher

A data fetcher is responsible for retrieving data from the picker.

The default data fetcher is implemented in the form of a ‘jsp’ page invoked using the Ajax asynchronous request. This page looks for the pickedOids and attributes query string parameters in the request. The page then invokes an Info*Engine webject to extract the attribute values for the specified Oid. The Oid could be in the form of Object Reference or Version Reference or UFID. The page returns the extracted information as a single string from which one can easily construct the javascript object. Such a string would look like

```
{"pickedObject":
 [ { "oid":
    "uid=wcadmin,
       ou=people,
       cn=koakx10-44,
       dc=ptcnet,
       dc=ptc,
       dc=com",
   "class":
    "wt.org.WTUser",
   "name":
    "Administrator"},
   { "oid":
    "uid=demo,
       ou=people,
       cn=koakx10-44,
       dc=ptcnet,
       dc=ptc,
       dc=com",
   "class":
    "wt.org.WTUser",
   "name":
    "demo"
  }
 ]
}
```

Default Callback Function

The Callback function is responsible for bringing the data from the picker to the elements of the JSP

The default callback function is authored using the ‘populate’ tag information, it does the task of updating the html form input text fields specified by the ‘id’ using the attribute information for the object picked which is made available in the form of a javascript object. The default callback function would be of the following form and would be unique for each of the picker occurrence

```
function jcaPickerCallBack0(objects)
{
 document.getElementById("b1").setAttribute
    ("value",objects.pickedObject[0].name);
 document.getElementById("b2").setAttribute
    ("value",objects.pickedObject[0].oid);
}
function jcaPickerCallBack1(objects)
{
 document.getElementById("c1").setAttribute
    ("value",objects.pickedObject[0].name);
 document.getElementById("c2").setAttribute
    ("value",objects.pickedObject[0].oid);
}
```

Use Existing Picker

To use the existing picker, the user would have the following procedures to be performed

1. Define the action to launch a picker
2. Create the HTML form elements to be updated using the picker
3. Use the tags

Define Action to Launch a Picker

The action needs to be defined in *actions.xml* file:

```
<objecttype name="<CustomObjectType>" class="<>">
  <action name="launch<SomeObjectType>Picker">
    <command windowType="popup"/>
  </action>
</objecttype>
```

Create the HTML form elements to be updated using the picker

Create the visible input fields that need to be updated as a part of the picker interaction using the AbstractGuiComponent to the client page i.e. create_step_<step no>.jsp. The following describes how to add a text field:

```
<%@ taglib prefix="c-rt" uri="http://java.sun.com/jstl/core_rt"; %>
<%
  TextBox<SomeObjectType> name = new TextBox();
  tab_tbox.setId("<SomeObjectType>Name_tbox_id");
  tab_tbox.setName("<SomeObjectType>Name_tbox_id");
  RenderingContext tab_rcontext = new
  RenderingContext();
%>
<c-rt:set var="tbox" value="<%=<SomeObjectType> name %>"/>
<c-rt:set var="tboxId" value="<%=<SomeObjectType> name.getId()%>"/>
```

Create the non-visible input fields that would be as well updated as a part of the picker interaction in the client page mentioned above.

```
<input id="<SomeObjectType>Reference" type="hidden" value=" " />
```

Note: Some tags don't need tlds. Anything defined in the tags directory has an automatically generated TLD file in the servlet container. In Windchill the standard is to use *wctags* as the prefix for this tags and you can reference them as follows

```
<%@ taglib prefix="wctags" tagdir="/WEB-INF/tags" %>
```

Use the tags

Use the 'PropertyPickerTag'in client page, i.e. create_step_<step no>.jsp, as follows

```
<p:propertyPicker
   propertyLabel="<SomeObject>"
   propertyField="${tbox}"
   action="launch<SomeObjectType>Picker"
   type="<myObjectType>"
  >
 <p:populate
    from="name"
    to="${ tboxId }" />
 <p:populate
    from="oid"
    to="<SomeObjectType>Reference 2" />
 <p:pickerParam
    name="pickerId"
    value="userPicker1" />
```

```
 <p:pickerParam
    name="objectType"
    value="wt.org.WTUser" />
 <p:pickerParam
    name="componentId"
    value="pickerSearch" />
</p:propertyPicker>
```

Note: The propertyPicker is a common component that requires that is usually given the picker prefix.

```
<%@ taglib prefix="picker" uri="http://www.ptc.com/windchill/taglib/picker" %>
```

The above usage supposes that user would be interested in using the ‘name’ and ‘oid’ attributes from the information extracted for the pickedObject.

The pickerProperty attributes are:

- label - String which serves as the localized display label of the property the value of which is to be populated using a picker. field
- AbstractGuiComponent - Used to render the form field that would be populated from picker object
- name String which serves as the name of the action as it exists in ‘actions.xml’
- type - ‘type’ of the action for which the definition exists in ‘actions.xml’ that would launch the picker.
- renderingContext - Rendering context to be used for the AbstractGuiComponent specified through a ‘field’ attribute

This populate tag configures source attribute of the picked object to be used to populate a target property on the client page. The tag would be always encapsulated in the propertyPicker tag. The attributes are:

- from - Specifies the name of the property (attribute) of the picked object
- to - HTML ’id’ of the form input element to update on the parent page.

The pickParam tag configures the “name=value” URL query string parameters to drive the actual picker content. The tag would be always encapsulated in the propertyPicker tag. The attributes are:

- name - Name of the query string parameter to be passed to the popup picker
- value - Value of the URL query string parameter to be passed to the popup picker

Available Pickers

- itempicker.tag
- genericpicker.tag
- grouppicker.tag
- userpicker.tag

Exercise 12-1: Employ userPicker on the Notify Now Table Action

Objectives

- Using existing pickers
- Customizing the Callback Function
- Customizing the default Data Fetcher

Scenario

In the previous exercises we have been typing e-mail addresses manually. In this exercise we add an e-mail address to the "Notify Now" string using the userPicker in our JSP for the "Notify Now" Table

Step 1. Update the JSP to use the userPicker

Step 2. Map a callback function to update the display attribute to usrename

Step 3. Map a callback function to update the hidden attribute for the long list of addresses

Implementing New Picker Interaction

This section defines the exact steps to implement a picker

Overview

The overall process looks like this:

1. Create *_Picker.jsp which is the main JSP to call for the wizard
2. Create a step that the *_Picker.jsp will call
3. Incorporate a Javascript to be invoked on button press

Author a JSP page named as <someObjectType>_picker.jsp. This JSP page must include a single step wizard using the wizard component of the Windchill Client Infrastructure.

```
<%@ page errorPage="/netmarkets/jsp/util/error.jsp"%>
<%@ page import=" ... %>
<jsp:useBean ... scope="request" >
<!—As the wizard step content might have a flexibility in
presenting itself and can be controlled by the picker parameters,
these parameters could be passed to the step using a bean
with request scope -->
<%@ include file="/netmarkets/util/beginPopup.jspf"%>
<%@ include file=”/netmarkets/components/picker.jspf”%>
...
...
...
<script language="JavaScript1.2">
<!—sets a callback to javascript function on hitting ‘ok’ action -->
 setUserSubmitFunction(onOK);
</script>
<%--> Wizard Step. <--%>
<jca:wizard
   buttonList="OkCancelWizardButtons">
 <jca:wizardStep
    action="<someObjectType>_picker_step "
    type="<someObjectType>"/>
</jca:wizard>
<!—Javascript callback function for wizard ‘OK’ action goes here -->
<!- OK callback definition start -->
...
<!—OK callback definition end -->
<%@ include file="/netmarkets/jsp/util/end.jspf"%>
```

The ‘picker.jspf’ is needed to be included in the picker page always. It defines the following javasccript window variables to track the pickerCallback and jcaPickerCallback functions. These windows variables are needed to be used in the ‘OK’ action javascript user submit function .

The ‘setUserSubmitFunction’ is needed to be invoked to configure wizard ‘OK’ action to invoke a javascript function.

The javascript callback definition would be as well included in this page, the definition of this callback and any other definitions of functions the callback calls need to be placed preferably before the ‘end.jspf’ is included.

Implement Picker Step

The picker step content would get defined by the codebase/netmarkets/< someObjectType >/<someObjectType>_picker_step.jsp. This step would render a search client or just a listing table to allow the user select the objects from. The details on how this could be done can be found out in the design pattern for Search Picker OR table component.

Provide definition of the javascript function to be invoked on ‘OK’ action

The javascript callback definition would be as well included in this page, the definition of this callback and any other definitions of functions the callback calls need to be placed preferably before the ‘end.jspf’ is included. This java script function has to fulfill the following responsibilities

- It would not have any arguments to it
- It scans the page and sifts through the table or tree data to identify what objects have been selected, if any are selected builds an array of these objects
- Invokes a pickedDataFetcher uri using Ajax and the array of objects selected passed over query string parameters as ‘oid=<oid_value>’ parameters that the pickedDataFetcher URI can understand.
- The Ajax request returns the response

Exercise 12-2: Create a new userPicker

Objectives

- Creating a New picker
- Augmenting existing pickers

Scenario

The existing userPicker has the limitation of forcing people to choose an e-mail address that is a member of the context. However, in this special situation the person running the function is an Administrator that would like to use the same screen as notify now for creating users on the fly if they don't exist.

Step 1. Create *_Picker.jsp which is the main JSP to call for the wizard

Step 2. Create a step that the *_Picker.jsp will call

Step 3. Incorporate a Javascript to be invoked on button press

Context Picker

Item Picker

Organization Picker

User Picker

Participant Picker

Summary

After completing this module, you should be able to:

- Configure an OOTB property picker in some wizard step.
- Manage interaction between that page and the picker.
- Configure a context picker to pick context(s) to be used in your application.
- Configure an Item Picker.
- Configure an Organization Picker.
- Configure a User Picker.
- Configure a Participant Picker.
- Add new customized type to search item picker.
- Display a picker for configuration specifications.

Module

13

Customizing the Product Structure Explorer (PSE)

The Produce Structure Explorer configuration is driven by XML files. In this module, you will learn how to modify those files to customize the display of the PSE.

Objectives

Upon successful completion of this module, you will be able to:

- Change the content of a PSE menu, toolbar or popup menu
- Change the default display order of columns in a PSE table, whether a specific column is mandatory or optional, or whether a column is frozen
- Change the display soft types of WTPart and WTPartUsageLink attributes in the PSE UI
- Customize the PSE so that it will fully support your modeled subclass
- Customize the PSE New Query dialog to include soft types of WTPart and allow queries to be created referencing IBA's of the soft type

Lecture Notes

You may use the space below to take your own notes.

Customizing the Product Structure Explorer

The PSE window display is configured in XML files.

These files are located under *WT_HOME/codebase/config/logicrepositoryxml/explorer*.

Changes to the XML files require restarting the Method Server and also clearing the client's Java cache.

The DTD files for the configuration are under *WT_HOME/codebase/config/logicrepositorydtd*.

PSE Modes

- The PSE has 3 modes:
  1. **Edit**
  2. **Draft**
  3. **Annotation**
- The mode is displayed in the PSE header.
- Toggle between **Edit** mode and **Draft** mode using the **Edit** menu.
- Activate the **Annotate** menu by creating an annotation set (**File > New > Annotation Set...** or opening an existing annotation set.

Configuring the Product Structure Explorer

Edit the PSE user interface by editing the XML configuration files.

The *WCCustomizersGuide.pdf* has an extenisve chapter on typical UI edits.

Exercise 13-1: Modify the Attributes Displayed in the PSE

Objectives

- Modify the attributes displayed in the PSE.

Scenario

Modify the attributes displayed in the PSE.

Step 1. Backup the PSE configuration files.

a. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure* to *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure_bak*.

b. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer* to *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer_bak*.

Step 2. Open the PSE.

a. From any Part page (including GSParts, which are Parts, as well), select **Actions > Open in Product Structure Explorer**.

b. View the available attributes on the **Information** sub tab.

c. Close the PSE.

Step 3. Modify displayed attributes.

a. Edit *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer/ExplorerForTablesAndPanels.xml* to modify displayed attributes.

b. Find the element ExplorerElementGroup with a LogicContext equal to "ptc.wnc.StructureExplorer" and dataType equal to "wt.part.WTPart".

Example:

```
<ExplorerElementGroup>
  <LogicContext application="ptc.wnc.StructureExplorer"
    dataType="wt.part.WTPart"/>
```

c. Find the nested AttributeGroup element with the id equal to "ptc.wnc.exp.ViewPropertiesPanel".

Example:

```
<AttributeGroup id="ptc.wnc.exp.ViewPropertiesPanel" displayMode="view">
   <CellDefinition id="number">
      <AttributeDefinition attributeId="number"/>
   </CellDefinition>
   <CellDefinition id="organizationIdentifier">
      <AttributeDefinition attributeId="organizationIdentifier"/>
   </CellDefinition>
   <CellDefinition id="name">
      <AttributeDefinition attributeId="name"/>
   </CellDefinition>
   <CellDefinition id="versionIterationView">
      <AttributeDefinition attributeId="versionIterationView"/>
   </CellDefinition>
```

d. Comment out “name” and add “Cost”.

Example:

```
<AttributeGroup id="ptc.wnc.exp.ViewPropertiesPanel" displayMode="view">
   <CellDefinition id="number">
      <AttributeDefinition attributeId="number"/>
   </CellDefinition>
   <CellDefinition id="organizationIdentifier">
      <AttributeDefinition attributeId="organizationIdentifier"/>
   </CellDefinition>
   <!-- COMMENT OUT NAME: BEGIN
   <CellDefinition id="name">
      <AttributeDefinition attributeId="name"/>
   </CellDefinition>
   COMMENT OUT NAME: END -->
   <CellDefinition id="Cost">
      <AttributeDefinition attributeId="Cost"/>
   </CellDefinition>
   <CellDefinition id="versionIterationView">
      <AttributeDefinition attributeId="versionIterationView"/>
   </CellDefinition>
```

e. Save the file.

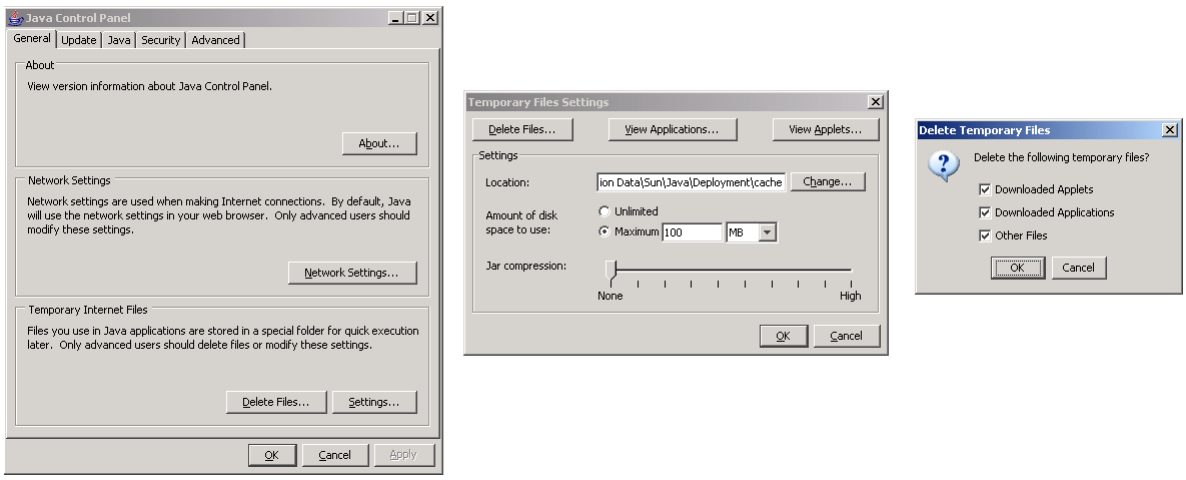
Step 4. Clear the client Java cache.

a. Close the browser.

b. The exact steps will vary depending on the configuration of the computer. Select **Start > Settings > Control Panel > Java**

c. From the General tab, select **Settings...**. From the **Temporary Files Settings** window, select **Delete Files...**. From the **Delete Temporary Files** window, select **OK**. Select **OK**. Select **OK**.

Example:



Delete Temporary Java Files

Step 5. Restart the Method Server.

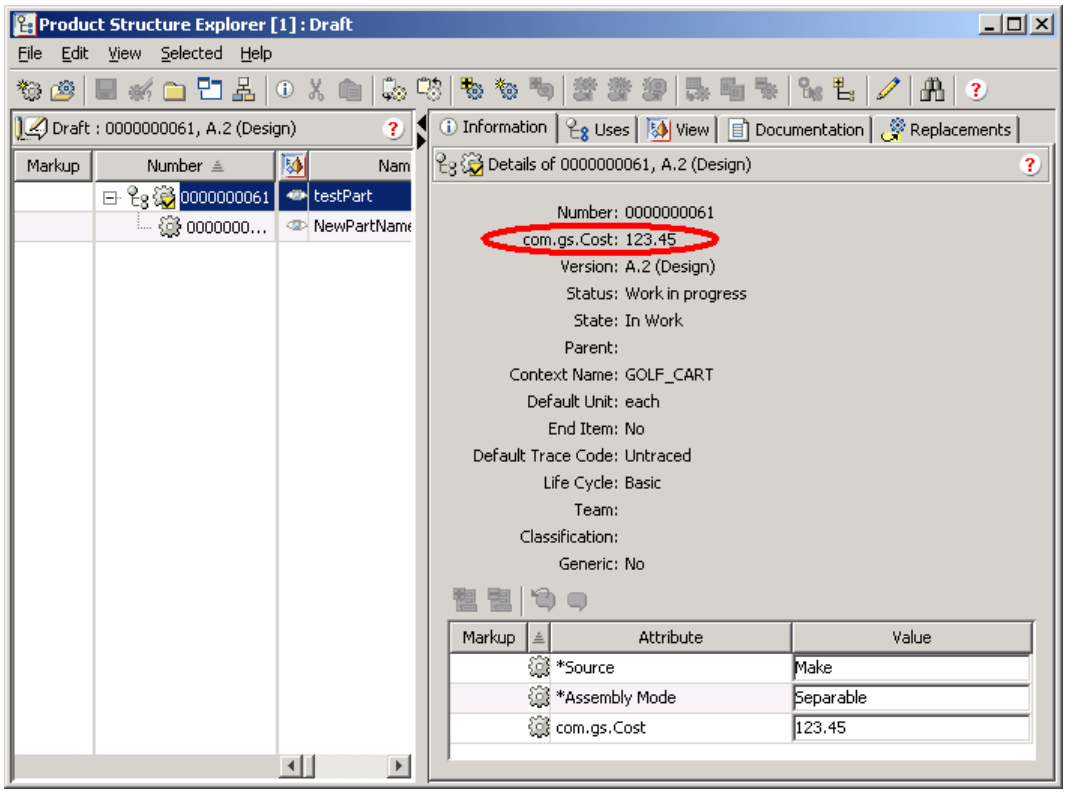
Step 6. Test it.

a. Open the browser.

b. Open a Part in PSE.

c. The attribute panel should hide **name** and display the IBA **Cost**.

Result:



Property Panel Displays Cost but not Name

Exercise 13-2: Modify the Menu Bar Actions Displayed in PSE

Objectives

- Modify the menu bar actions displayed in PSE.

Scenario

Modify the menu bar actions displayed in PSE.

Step 1. Backup the PSE configuration files.

a. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure* to *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure_bak*.

b. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer* to *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer_bak*.

Step 2. Open the PSE.

a. From any Part page (including GSParts, which are Parts, as well), select **Actions > Open in Product Structure Explorer**.

b. Write down the mode (**Draft** or **Edit**) displayed in the PSE header.

c. Note the menu bar actions.

d. Toggle the mode of the PSE (from either **Draft** to **Edit**, or from **Edit** to **Draft**) by selecting the Edit menu.

e. Note the menu bar actions.

f. Close the PSE.

Step 3. Review the configuration file for menus.

a. Edit *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure/PDMLinkExplorerMenus.xml*.

b. Note that this file has one ExplorerElementGroup for the application "ptc.pdm.ProductStructureExplorer".

Example:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE LogicRepository SYSTEM "/config/logicrepository/dtd/LogicRepository.dtd">
<LogicRepository>

  <ExplorerElementGroup>
    <LogicContext application="ptc.pdm.ProductStructureExplorer"/>
```

c. Further, there is only one element ActionAccess with an id equal to "ptc.pdm.pse.ExplorerTreeAA". It contains one MenuBar element and three ModeToolBar elements.

- To remove or move elements in the **File**, **Edit**, **View** and **Selected** menus, edit the MenuBar element.
- The MenuBar element contains separate MenuBar elements for **File**, **Edit**, **View** and **Selected**.
- This is reference information not applied in this exercise.
-

d. As noted previously, there are three ModeToolBar elements. ToolBarA is for **Edit** mode; ToolBarB is for **Draft** mode and ToolBarC is for **Annotate** mode. For example, the ToolBarA element begins as follows:

Example:

```
<ModeToolBar id="ToolBarA">
  <Import id="ptc.wnc.exp.EditAppMode"/>
```

Step 4. Edit the **Edit** mode menu bar.

a. Comment out **Cut**.

Example:

```
<ModeToolBar id="ToolBarA">
 <Import id="ptc.wnc.exp.EditAppMode"/>
 <MenuItemIdentifier id="FileNewPartMI"/>
 <MenuItemIdentifier id="FileOpenPartMI"/>
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.CloseMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.LaunchNewPSEMI"/>
 <MenuItemIdentifier id="SetConfigSpecMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.RefreshMI"/>
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.CheckOutMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CheckInMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.UndoCheckoutMI"/>
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.InfoPageMI"/>
 <!-- REMOVE CUT FROM EDIT MODE MENU BAR
 <MenuItemIdentifier id="ptc.wnc.exp.CutMI"/>
 -->
```

b. Save the file.

Step 5. Edit the **Draft** mode menu bar.

a. Comment out **Copy**.

Example:

```
<ModeToolBar id="ToolBarB">
 <Import id="ptc.wnc.exp.DraftAppMode"/>
 <MenuItemIdentifier id="FileNewPartMI"/>
 <MenuItemIdentifier id="FileOpenPartMI"/>
 <Separator/>
 <MenuItemIdentifier id="FileSaveChangesMI"/>
 <MenuItemIdentifier id="FileAnnotationValidationMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CloseMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.LaunchNewPSEMI"/>
 <MenuItemIdentifier id="SetConfigSpecMI"/>
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.InfoPageMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CutMI"/>
 <!-- REMOVE COPY FROM DRAFT MODE MENU BAR
 <MenuItemIdentifier id="ptc.wnc.exp.CopyMI"/>
 -->
```

b. Save the file.

Step 6. Challenge: edit the **Annotate** mode menu bar.

a. Comment out **Paste**.

Example:

```
<ModeToolBar id="ToolBarC">
 <Import id="ptc.wnc.exp.AnnotateAppMode"/>
 <Import id="ptc.wnc.exp.ReadOnlyAppMode"/>
 <MenuItemIdentifier id="FileSaveChangesMI"/>
 <MenuItemIdentifier id="FileAnnotationValidationMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CloseMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.LaunchNewPSEMI"/>
 <MenuItemIdentifier id="SetConfigSpecMI"/>
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.InfoPageMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CutMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CopyMI"/>
 <!-- REMOVE PASTE FROM ANNOTATE MODE MENU BAR
 <MenuItemIdentifier id="ptc.wnc.exp.PasteMI"/>
 -->
```

b. Save the file.

Step 7. Clear the client Java cache.

a. Close the browser.

b. The exact steps will vary depending on the configuration of the computer. Select **Start > Settings > Control Panel > Java**

c. From the General tab, select **Settings...**. From the **Temporary Files Settings** window, select **Delete Files...**. From the **Delete Temporary Files** window, select **OK**. Select **OK**. Select **OK**.

Step 8. Restart the Method Server.

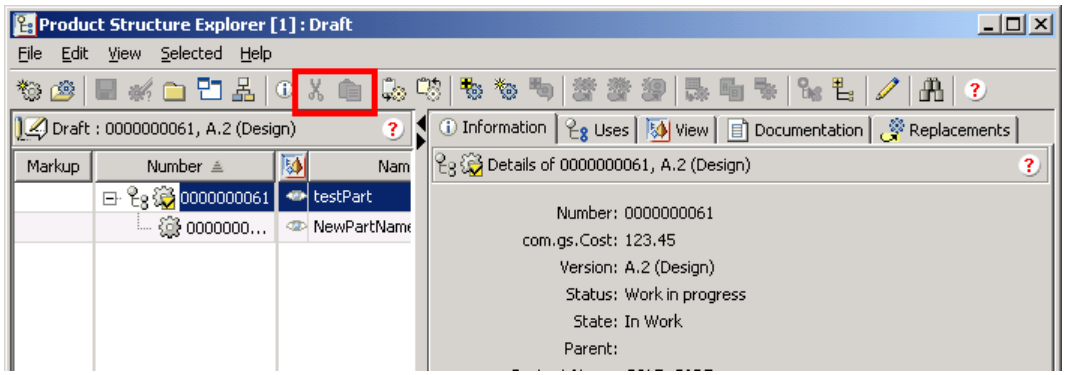
Step 9. Test it.

a. Open the browser.

b. Open a Part in PSE.

c. Switch the modes between **Edit** and **Draft**. In the appropriate mode, **Cut** or **Copy** will be hidden.

Result:



Draft Mode with only Cut and Paste

d. Open or create an annotation. **Paste** will be hidden.

Exercise 13-3: Modify the Popup Actions Displayed in PSE

Objectives

- Modify the pop up actions displayed in the PSE.

Scenario

Modify the pop up actions displayed in the PSE.

Step 1. Backup the PSE configuration files.

a. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure* to *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure_bak*.

b. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer* to *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer_bak*.

Step 2. Open the PSE.

a. From any Part page (including GSParts, which are Parts, as well), select **Actions > Open in Product Structure Explorer**.

b. Write down the mode (**Draft** or **Edit**) displayed in the PSE header.

c. Right-click in the @TODO panel and note the pop up menu actions.

d. Toggle the mode of the PSE (from either **Draft** to **Edit**, or from **Edit** to **Draft**) by selecting the Edit menu.

e. Right-click in the @TODO panel and note the pop up menu actions.

f. Close the PSE.

Step 3. Review the configuration file for menus.

a. Edit *PDMLinkExplorerMenus.xml* in *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure*.

b. Note that this file has one ExplorerElementGroup for the application "ptc.pdm.ProductStructureExplorer".

Example:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE LogicRepository SYSTEM "/config/logicrepository/dtd/LogicRepository.dtd">
<LogicRepository>

  <ExplorerElementGroup>
    <LogicContext application="ptc.pdm.ProductStructureExplorer"/>
```

c. There is only one element ActionAccess with an id equal to "ptc.pdm.pse.ExplorerTreeAA". It contains two ModePopupMenu elements.

d. PopupMenuA is for **Edit** mode, and PopupMenuB is for **Draft** mode.

Example:

```
<ModePopupMenu id="PopupMenuA">
 <Import id="ptc.wnc.exp.EditAppMode"/>
```

Step 4. Edit the **Edit** mode pop up menu.

a. Add a second **Check In** action to the pop up menu.

Example:

```
<ModePopupMenu id="PopupMenuA">
 <Import id="ptc.wnc.exp.EditAppMode"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CutMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CopyMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.PasteMI"/>
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.CheckOutMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CheckInMI"/>
 <!-- ADD 2ND CHECK IN TO EDIT POPUP -->
 <MenuItemIdentifier id="ptc.wnc.exp.CheckInMI"/>
 <!-- -->
```

b. Save the file.

Step 5. Edit the **Draft** mode pop up menu.

a. Add a second **Revise** action to the pop up menu.

Example:

```
<ModePopupMenu id="PopupMenuB">
 <Import id="ptc.wnc.exp.DraftAppMode"/>
 <Import id="ptc.wnc.exp.AnnotateAppMode"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CutMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CopyMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.PasteMI"/>
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.ReviseMI"/>
 <!-- ADD 2ND REVISE TO DRAFT POPUP -->
 <MenuItemIdentifier id="ptc.wnc.exp.ReviseMI"/>
 <!-- -->
```

b. Save the file.

Step 6. Clear the client Java cache.

a. Close the browser.

b. The exact steps will vary depending on the configuration of the computer. Select **Start > Settings > Control Panel > Java**

c. From the General tab, select **Settings...**. From the **Temporary Files Settings** window, select **Delete Files...**. From the **Delete Temporary Files** window, select **OK**. Select **OK**. Select **OK**.

Step 7. Restart the Method Server.

Step 8. Test it.

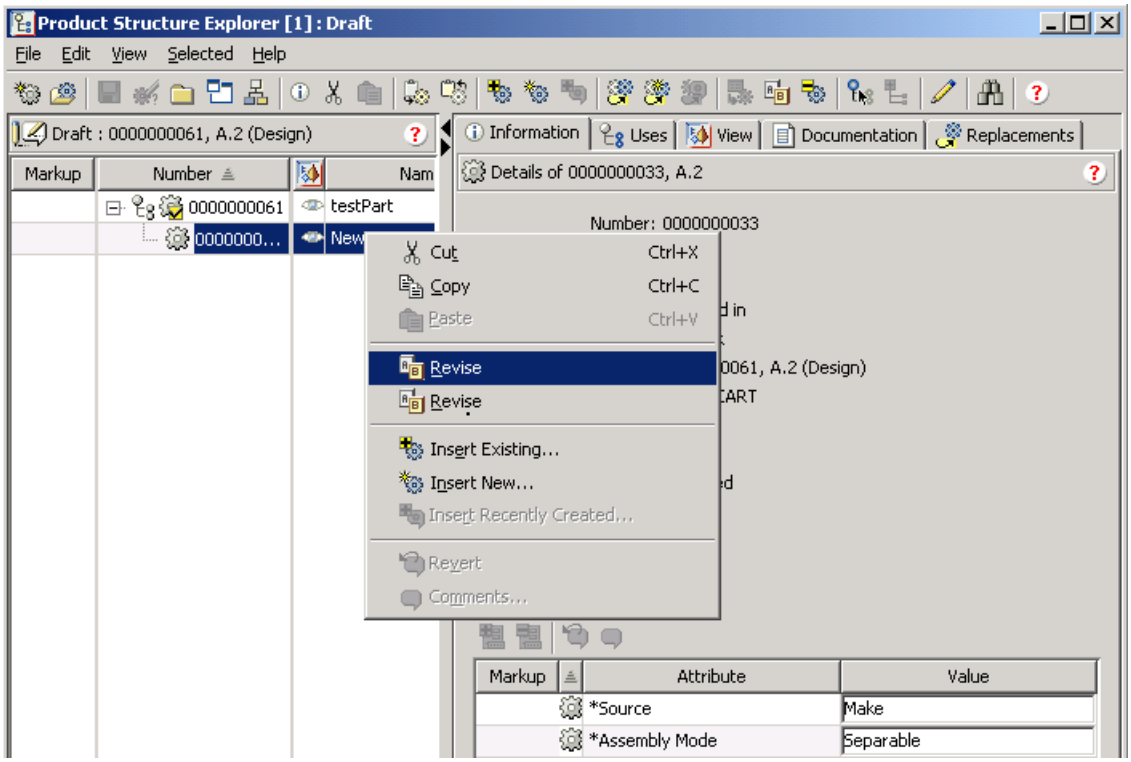
a. Open the browser.

b. Open a Part in PSE.

c. Switch the modes between **Edit** and **Draft**

d. In the appropriate mode, either two **Check In** or two **Revise** actions will appear in the pop up menu.

Example:



Draft Mode Pop Up Menu with Two Revise Actions

Exercise 13-4: Modify the Sub Tab Pop Up Menus Displayed in PSE

Objectives

- Modify the sub tab Actions Displayed in PSE.

Scenario

Modify the sub tab pop up menus displayed in PSE.

Step 1. Backup the PSE configuration files.

a. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure* to *WT_HOME/codebase/config/logicrepository/xml/explorer/productstructure_bak*.

b. Copy the directory *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer* to *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer_bak*.

Step 2. Open the PSE.

a. From any Part page (including GSParts, which are Parts, as well), select **Actions > Open in Product Structure Explorer**.

b. Write down the mode (**Draft** or **Edit**) displayed in the PSE header.

c. Open the **Uses** tab.

d. Note the menu of actions insides the **Uses** tab itself.

e. Right click inside the **Uses** tab and note the actions displayed.

f. Toggle the mode of the PSE (from either **Draft** to **Edit**, or from **Edit** to **Draft**) by selecting the Edit menu.

g. Note the menu of actions insides the **Uses** tab itself.

h. Right click inside the **Uses** tab and note the actions displayed.

i. Close the PSE.

Step 3. Review the configuration file for menus.

a. Edit *WT_HOME/codebase/config/logicrepository/xml/explorer/structureexplorer/ExplorerMenusForUsesTab.xml.xml*.

b. This file has two ActionAccess elements.

c. Edit the ActionAccess for “ptc.wnc.exp.PartUsesLinkAssocTableAA”. This represents the **Uses** sub tab panel.

d. Note that the sub tab has it’s own menu bar, tool bar and pop up menu definition elements.

e. There are two ModePopUp elements. PopupMenuA is for **Edit** mode; PopupMenuB is for **Draft** mode. For example, the PopupMenuA element begins as follows:

Example:

```
<ModePopupMenu id="PopupMenuA">
 <Import id="ptc.wnc.exp.EditAppMode"/>
```

Step 4. Edit the **Edit** mode pop up menu.

a. Duplicate the **Cut**, **Copy** and **Paste** actions.

Example:

```
<ModePopupMenu id="PopupMenuA">
 <Import id="ptc.wnc.exp.EditAppMode"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CutAssocMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CopyAssocMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.PasteAssocMI"/>
 <!-- EDIT MODE USES SUB TAB POP UP MENU HAS EXTRA CUT, COPY, PASTE -->
 <Separator/>
 <MenuItemIdentifier id="ptc.wnc.exp.CutAssocMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.CopyAssocMI"/>
 <MenuItemIdentifier id="ptc.wnc.exp.PasteAssocMI"/>
```

b. Save the file.

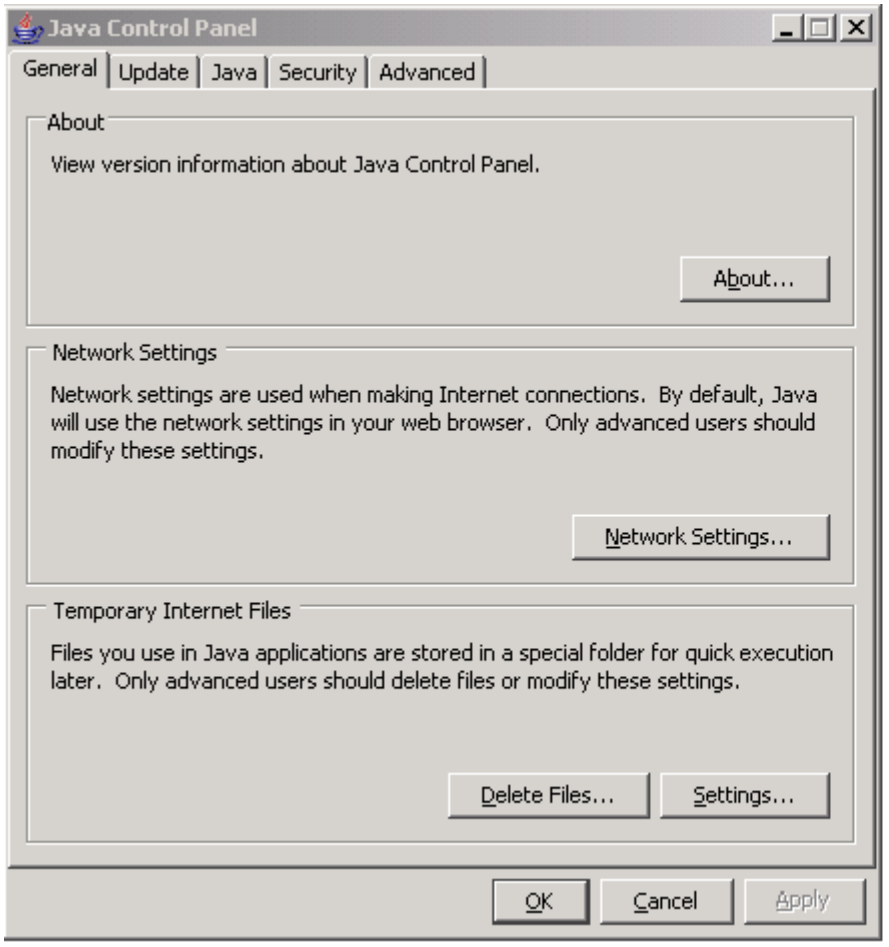
Step 5. Clear the client Java cache.

a. Close the browser.

b. The exact steps will vary depending on the configuration of the computer. Select **Start > Settings > Control Panel > Java**

c. From the General tab, select **Settings...**. From the **Temporary Files Settings** window, select **Delete Files...**. From the **Delete Temporary Files** window, select **OK**. Select **OK**. Select **OK**.

Example:



Delete Temporary Java Files

Step 6. Restart the Method Server.

Step 7. Test it.

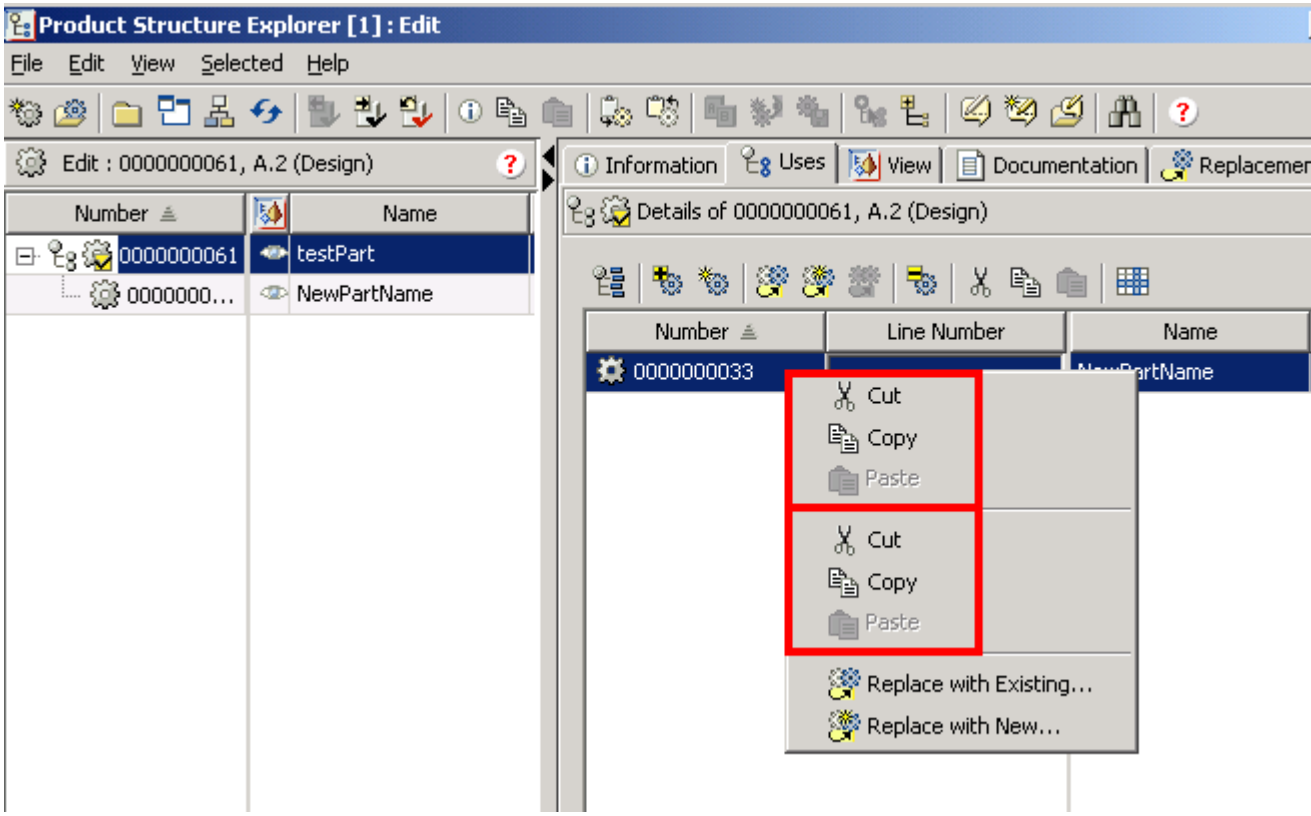
a. Open the browser.

b. Open a Part in PSE.

c. Switch to **Edit** mode and open the **Uses** sub tab.

d. The right-click menu actions should include two sets of **Cut** / **Copy** / **Paste** actions.

Result:



Edit Mode : Uses Sub Tab

Summary

After completing this module, you should be able to:

- Change the content of a PSE menu, toolbar or popup menu
- Change the default display order of columns in a PSE table, whether a specific column is mandatory or optional, or whether a column is frozen
- Change the display soft types of WTPart and WTPartUsageLink attributes in the PSE UI
- Customize the PSE so that it will fully support your modeled subclass
- Customize the PSE New Query dialog to include soft types of WTPart and allow queries to be created referencing IBA's of the soft type

Copyright

Third-Party Technology Information

Certain PTC software products contain licensed third-party technology:
Rational Rose and Rational ClearCase are copyrighted software of IBM Corp
RetrievalWare is copyrighted software of Convera Corporation.
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.
I Run and ISOGEN are copyrighted software of Alias Ltd.
Xdriver is copyrighted software of 3Dconnexion, Inc, a Logitech International S.A. company.
G POST is copyrighted software and a registered trademark of Intercim.
VERICUT is copyrighted software and a registered trademark of CGTech.
FLEXnet Publisher is copyrighted software of Macrovision Corporation
Pro/PLASTIC ADVISOR is powered by Moldflow technology.

Fatigue Advisor nCode libraries from nCode International.

TetMesh GHS3D provided by Simulog Technologies, a business unit of Simulog S.A.

MainWin Dedicated Libraries are copyrighted software of Mainsoft Corporation.

DFORMD.DLL is copyrighted software from Compaq Computer Corporation and may not be distributed.

LightWork Libraries are copyrighted by LightWork Design 19902001

Visual Basic for Applications and Internet Explorer is copyrighted software of Microsoft Corporation.

Parasolid is UGS Corp.

TECHNOMATIX is copyrighted software and contains proprietary information of Technomatix Technologies Ltd.

TIBCO ActiveEnterprise, TIBCO Designer, TIBCO Enterprise Message Service, TIBCO Rendezvous, TIBCO TurboXML, and TIBCO BusinessWorks are provided by TIBCO Software Inc.

DataDirect Connect is copyrighted software of DataDirect Technologies

Technology "Powered by Groove" is provided by Groove Networks, Inc.

Technology "Powered by WebEx" is provided by WebEx Communications, Inc.

Oracle 8i run time, Oracle 9i run time, and Oracle 10g run time are Copyright 20022004 Oracle Corporation. Oracle programs provided herein are subject to a restricted use license and can only be used in conjunction with the PTC software they are provided with. Adobe Acrobat Reader and Adobe Distiller are copyrighted software of Adobe Systems Inc. and are subject to the Adobe End User License Agreement as provided by Adobe with those products.

Certain license management is based on Elan License Manager 1989-1999 Rainbow Technologies, Inc. All rights reserved. Portions compiled from Microsoft Developer Network Redistributable Sample Code, Copyright 1998 by Microsoft Corporation. The CD-ROM Composer and CD-ROM Consumer are based on Vivace CD-Web Composer Integrator 1996-1997 KnowledgeSet Corporation. All rights reserved.

Larson CGM Engine 9.4, Copyright 1992-2008 Larson Software Technology, Inc. All rights reserved.

Certain graphics-handling portions are based on the following technologies:

GIF: Copyright 1989, 1990 Kirk L. Johnson. The author disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall the author be liable for any special, indirect, or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence, or other tortious action, arising out of or in connection with the use or performance of this software.

JPEG: This software is based in part on the work of the Independent JPEG Group.

PNG: Copyright 2004-2008 Glenn Randers-Pehrson

TIFF: Copyright 1988-1997 Sam Leffler, Copyright 1991-1997 Silicon Graphics, Inc. The software is provided AS IS and without warranty of any kind, express, implied, or otherwise, including without limitation, any warranty of merchantability or fitness for a particular purpose. In no event shall Sam Leffler or Silicon Graphics be liable for any special, incidental, indirect, or consequential damages of any kind, or any damages whatsoever resulting from loss of use, data or profits, whether or not advised of the possibility of damage, or on any theory of liability, arising out of or in connection with the use or performance of this software. XBM, Sun Raster, and Sun Icon: Copyright,1987, Massachusetts Institute of Technology.

ZLIB: Copyright 1995-2004 Jean-loup Gailly and Mark Adler.

PDFlib software is copyright 1997-2005 PDFlib GmbH. All rights reserved. PStill software is copyright Dipl.- Ing. Frank Siegert, 1996-2005

Proximity Linguistic Technology provides Spelling Check/Thesaurus portions of certain software products, including: The Proximity/Bertelsmann Lexikon Verlag Database. Copyright 1997 Bertelsmann Lexikon Verlag. Copyright 1997, All Rights Reserved, Proximity Technology, Inc.; The Proximity/C.A. Strombertg AB Database. Copyright 1989 C.A. Strombertg AB. Copyright 1989, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Editions Fernand Nathan Database. Copyright 1984 Editions Fernand Nathan. Copyright 1989, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Espasa-Calpe Database. Copyright 1990 Espasa-Calpe. Copyright 1990, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Dr. Lluis de Yzaguirre i Maura Database. Copyright 1991 Dr. Lluis de Yzaguirre i Maura Copyright 1991, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Franklin Electronic Publishers, Inc. Database. Copyright 1994 Franklin Electronic Publishers, Inc. Copyright 1994, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Hachette Database. Copyright 1992 Hachette. Copyright 1992, All Rights Reserved, Proximity Technology, Inc.; The Proximity/IDE a.s. Database. Copyright 1989, 1990 IDE a.s. Copyright 1989, 1990, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Merriam-Webster, Inc. Database. Copyright 1984, 1990 Merriam-Webster, Inc. Copyright 1984, 1990, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Merriam-Webster, Inc./Franklin Electronic Publishers, Inc. Database. Copyright 1990 Merriam-Webster Inc. Copyright 1994 Franklin Electronic Publishers, Inc. Copyright 1994, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Munksgaard International Publishers Ltd. Database. Copyright 1990 Munksgaard International Publishers Ltd. Copyright 1990, All Rights Reserved, Proximity Technology, Inc.; The Proximity/S. Fischer Verlag Database. Copyright 1983 S. Fischer Verlag. Copyright 1997, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Van Dale Lexicografie by Database. Copyright 1995, 1997 Van Dale Lexicografie by. Copyright 1996, 1997, All Rights Reserved, Proximity Technology, Inc.; The Proximity/William Collins Sons & Co. Ltd. Database. Copyright 1984, 1990 William Collins Sons & Co. Ltd. Copyright 1988, 1990, All Rights Reserved, Proximity Technology, Inc.; The Proximity/Zanichelli Database. Copyright 1989 Zanichelli. Copyright 1989, All Rights Reserved, Proximity Technology, Inc.

CimPro, IGES/Pro, and PS/Pro software are provided by CADCAM-E, Inc.

The Arbortext Import/Export feature includes components that are licensed and copyrighted by CambridgeDocs LLC (2002-2005 CambridgeDocs LLC). This functionality:

Includes software developed by the Apache Software Foundation (http://www.apache.org).

Redistributes JRE from Sun Microsystems. The Redistributable is complete and unmodified, and only bundled as part of the product. CambridgeDocs is not distributing additional software intended to supersede any component(s) of the Redistributable, nor has CambridgeDocs removed or altered any proprietary legends or notices contained in or on the Redistributable. CambridgeDocs is only distributing the Redistributable pursuant to a license agreement that protects Suns interests consistent with the terms contained in the Agreement. CambridgeDocs agrees to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys fees) incurred in connection with any claim, lawsuit, or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software. This product includes code licensed from RSA Security, Inc. ICU4J portions licensed from IBM (see terms at http://www-128.ibm.com/developerworks/opensource/).

Redistributes the Saxon XSLT Processor from Michael Kay, more information, including source code is available at http://saxon.sourceforge.net/.

Uses cxImage, an open source image conversion library that follows the zlib license. cxImage further uses the following images libraries which also ship (statically linked) with cxLib: zLib, LibTIFF, LibPNG, LibJPEG, JBIG-Kit, JasPer, LibJ2K. See http://www.xdp.it/cximage.htm.

Includes software developed by Andy Clark, namely Neko DTD. NekoDTD is Copyright 2002, 2003, Andy Clark. All rights reserved. For more information, visit http://www.apache.org/~andyc/neko/doc/index.html. Includes code which was developed and copyright by Steven John Metsker, and shipped with Building Parsers with Java, from Addison Wesley.

Uses controls from Infragistics NetAdvantage 2004, Volume 3, Copyright 2004 Infragistics.

Word, FrameMaker, and Interleaf filters. Copyright 2000 Blueberry Software. All rights reserved.

Portions of software documentation are used with the permission of the World Wide Web Consortium. Copyright 19942008 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal. Such portions are indicated at their points of use. Copyright and ownership of certain software components is with YARD SOFTWARE SYSTEMS LIMITED, unauthorized use and copying of which is hereby prohibited. YARD SOFTWARE SYSTEMS LIMITED 1987. (Lic. #YSS:SC:9107001) Certain business intelligence reporting functionality is powered by Cognos.

MKM (Mathsoft Kernel Maple) 1994 Waterloo Maple Software.

Microsoft Internet Explorer 1995-2005 Microsoft Corporation.

Web Help 2004 and Macromedia RoboHelp are copyrighted software of Adobe Systems Incorporated.

Portions of the Mathcad Solver 1990-2002 by Frontline Systems, Inc.

Sentry Spelling-Checker Engine copyright 1994-2003 Wintertree Software, Inc.

File Filters 1986-2002 Circle Systems, Inc.

PDEFIT 1995-2002 Dr. Klaus Schittkowski

Hyphenation Copyright 1986-1999, Computer Hyphenation Ltd. All rights reserved.

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.

Certain software components licensed in connection with the Apache Software Foundation, all rights reserved, and use is subject to the terms and limitations (and license agreement) at http://www.apache.org. Apache software is provided by its Contributors AS IS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, and any expressed or implied warranties, including, but not limited to, the implied warranties of title non-infringement, merchantability and fitness for a particular purpose are disclaimed. In no event shall the Apache Software Foundation or its Contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage. Apache software includes:

Apache Server, Tomcat, Xalan, Xerces, and Jakarta, Jakarta POI, Jakarta Regular Expression, Commons-FileUpload, and XML Beans

IBM XML Parser for Java Edition, the IBM SaxParser and the IBM Lotus XSL Edition

DITA-OT - Apache License Version

JakartaORO (as used with Jena Software)

NekoHTML software developed by Andy Clark Copyright 2002-2005, Andy Clark. All rights reserved.

Lucene (http://lucene.apache.org)

IzPack: Java-based Software Installers Generator (http://www.izforge.com/izpack/start)

Pop up calendar components Copyright 1998 Netscape Communications Corporation. All Rights Reserved. The following software is distributed under GNU Lesser General Public License (LGPL) which is at http://www.gnu.org/copyleft/lesser.html and is provided AS IS by authors with no warranty therefrom without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE (see the GNU LGPL for more details). Upon request PTC will provide the source code for such software for a charge no more than the cost of performing this distribution:

GTK+ - The GIMP Toolkit is licensed under the GNU Library General Public License (LGPL). You may obtain a copy of the source code at http://www.gtk.org/, which is likewise provided under the GNU LGPL.

OmniORB is distributed under the terms and conditions of the GNU General Public License The OmniORB Libraries are released under the GNU LGPL.

Java Port copyright 1998 by Aaron M. Renn (arenn@urbanophile.com), is redistributed under the GNU LGPL. You may obtain a copy of the source code at http://www.urbanophile.com/arenn/hacking/download.html. The source code is likewise provided under the GNU LGPL.

JFreeChart is licensed under the GNU LGPL and can be found at http://www.jfree.org

eXist, an Open Source Native XML Database, is redistributed under the GNU LGPL. You may obtain a copy of the source code at http://exist.sourceforge.net/index.html. The source code is likewise provided under the GNU LGPL.

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 http://www.info-zip.org/doc/LICENSE. "Info-ZIP" is defined as the following set of individuals: Mark Adler, John Bush, Karl Davis, Harald Denker, Jean-Michel Dubois, Jean-loup Gailly, Hunter Goatley, Ian Gorman, Chris Herborth, Dirk Haase, Greg Hartwig, Robert Heath, Jonathan Hudson, Paul Kienitz, David Kirschbaum, Johnny Lee, Onno van der Linden, Igor Mandrichenko, Steve P. Miller, Sergio Monesi, Keith Owens, George Petrov, Greg

Perl support was developed with the aid of Perl Kit, Version 5.0. Copyright 1989-2002, Larry Wall. All rights reserved. Regular Expressions support was derived from copyrighted software written by Henry Spencer, Copyright 1986 by University of Toronto.

SGML parser: Copyright 1994, 1995, 1996, 1997, 1998 James Clark, 1999 Matthias Clasen.

XML parser and XSLT processing was developed using Libxml and Libxslt by Daniel Veillard, Copyright 2001.

Curl software, Copyright 1996 - 2005, Daniel Stenberg, (daniel@haxx.se). All rights reserved. Software is used under the following permissions: Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. THE SOFTWARE IS PROVIDED AS IS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use, or other dealings.

The cad2eda program utilizes wxWidgets (formerly wxWindows) libraries for its cross-platform UI API, which is licensed under the wxWindows Library License at http://www.wxwindows.org.

LAPACK libraries used are freely available at http://www.netlib.org/ (authors are Anderson, E. and Bai, Z. and Bischof, C. and Blackford, S. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Sorensen, D.). The following software, which is provided with and called by certain PTC software products, is licensed under the GNU General Public License (http://www.gnu.org/licenses/gpl.txt) and is provided AS IS by the authors with no warranty therefrom without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE (see the GNU GPL for more details). Upon request PTC will provide the source code for such software for a charge no more than the cost of performing this distribution:

Ghost Script (http://www.cs.wisc.edu/~ghost);

The PJA (Pure Java AWT) Toolkit library (http://www.eteks.com/pja/en).

Launch4j: This program is free software licensed under the GPL license, the head subproject (the code which is attached to the wrapped jars) is licensed under the LGPL license. Launch4j may be used for wrapping closed source, commercial applications.

The following unmodified libraries distributed under the GNU-GPL: libstdc and #ziplib (each are provided pursuant to an exception that permits use of the library in proprietary applications with no restrictions provided that the library is not modified).

Rhino JavaScript engine, distributed with a form of the Mozilla Public License (MPL).

Java Advanced Imaging (JAI) is provided pursuant to the Sun Java Distribution License (JDL) at http://www.jai.dev.java.net. The terms of the JDL shall supersede any other licensing terms for PTC software with respect to JAI components.

May include Jena Software Copyright 2000, 2001, 2002, 2003, 2004, 2005 Hewlett-Packard Development Company, LP. THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Jena includes:

JakartaORO software developed by the Apache Software Foundation (described above).

ICU4J software Copyright 1995-2003 International Business Machines Corporation and others All rights reserved. Software is used under the following permissions: Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and in supporting documentation. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

May contain script.aculo.us (built on prototype.conio.net). Copyright 2005 Thomas Fuchs (http://script.aculo.us, http://mir.aculo.us). Software is used under the following permissions: Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER

DEALINGS IN THE SOFTWARE.

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England. This software is based in part on the work of the Independent JPEG Group. antlr-2.7.0.zip - ANTLR parser and lexer generator, version 2.7.0; Provided pursuant to: ANTLR 2 License http://www.antlr.org/license.html.

jpeg-6b.zip - JPEG image compression library, version 6.2. Used to create images for HTML output; Provided pursuant to: http://www.faqs.org/faqs/jpeg-faq/part2.

lpng120.zip - PNG image library version 1.2.0. http://www.ijg.org/; Provided pursuant to: http://www.libpng.org/pub/png/src/libpng-LICENSE.txt.

pcre-4.3-2-src.zip - Perl Compatible Regular Expression Library version 4.3. http://www.pcre.org/; Provided pursuant to: PCRE License tiff-v3.4-tar.gz - Libtiff File IO Library version 3.4: (see also http://www.libtiff.org/ ftp://ftp.sgi.com/graphics/tiff) Used by the image EFI library; Provided pursuant to: http://www.libtiff.org/misc.html.

zlib-1.2.1.tar.gz - Zip compression library version 1.2.1. http://www.gzip.org/zlib; Provided pursuant to: Zlib.h License.

The Boost Library - Misc. C++ software from http://www.boost.org/; Provided pursuant to: Boost Software License http://www.boost.org/more/license_info.html and http://www.boost.org/LICENSE_1_0.txt.

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) (OCT95) or DFARS 227.7202-1(a) and 227.7202-3(a) (JUN95), and are provided to the US Government under a limited commercial license only. For procurements predating the above clauses, use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227 7013 (OCT88) or Commercial Computer Software-Restricted Rights at FAR 52.227 19(c)(1)-(2) (JUN87), as applicable. 071906

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