TEAMCENTER

Rich Client Customization

Teamcenter 2412

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a global leader in the growing field of product lifecycle management (PLM), manufacturing operations management (MOM), and electronic design automation (EDA) software, hardware, and services. Siemens works with more than 100,000 customers, leading the digitalization of their planning and manufacturing processes. At Siemens Digital Industries Software, we blur the boundaries between industry domains by integrating the virtual and physical, hardware and software, design and manufacturing worlds. With the rapid pace of innovation, digitalization is no longer tomorrow’s idea. We take what the future promises tomorrow and make it real for our customers today. Where today meets tomorrow. Our culture encourages creativity, welcomes fresh thinking and focuses on growth, so our people, our business, and our customers can achieve their full potential.

Support Center: **support.sw.siemens.com**

Send Feedback on Documentation: **support.sw.siemens.com/doc_feedback_form**

Contents

Rich client customization using Eclipse plug-ins

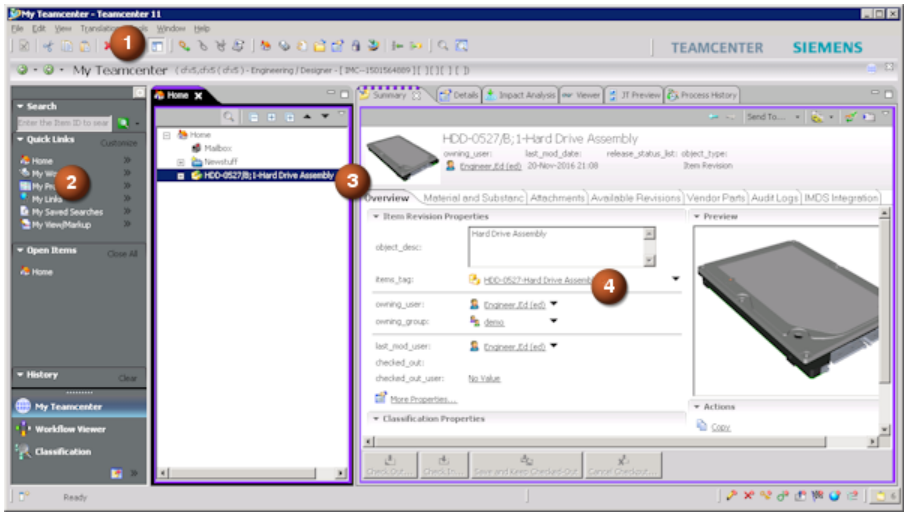
Rich client customization using Authorization

1. Getting started with rich client customization

Rich client customization types and requirements

The rich client uses several methods to control how it presents information. You need to be familiar with what you can customize, and which method controls each part of the interface.

Where can I customize?



	Location	Customization Method
1	Menu, Toolbar, and Context Menu	
	Hide or suppress	Application
	Create new	**Plug-in**
2	Left-hand Navigation (LHN) Panel	
	Search (Quick search)	Preference
	Quick Links	User
	Open Items	*None*
	History	*None*

	Location	Customization Method
	Perspectives (Applications)	User
3	Perspective	
	Modify existing	User
	Create new	**Plug-in**
4	View	**XRT**

What do I need?

In order to perform these customization methods, you need to know the following:

- **XML Rendering Templates (XRT)**

  XML rendering templates, also called style sheets, control the layout of properties within the windows. Style sheets:

  - Are easy to create and modify. (use XML)
  - Are stored in the database and do not need to be deployed.
  - Are the source for the organization of properties within view tabs as well as the view tabs themselves.
  - Do not require any additional software.

- **Custom Plug-ins**

  The core of the rich client is a set of plug-ins running on the Eclipse platform. The Eclipse platform controls the menuing, windowing, and all other general behind-the-scenes software requirements. Plug-in projects:

  - Are powerful (use Java)
  - Require the installation of the Eclipse IDE.
  - Require knowledge of the Eclipse platform, plug-in development, workbench, and the OSGi framework.
  - Are the source for modifying visibility of commands, menus, and toolbars.
  - Are the source of new command locations, views, perspectives.

- Need to be deployed to each rich client installation.

- **Preferences**

  These variables are stored within the Teamcenter database, and used to set configuration options. Teamcenter preferences:

  - Are simple text variables, similar to operating system environment variables.
  - Can be set using the rich client or command-line utility.
  - Have multiple layers of scope from user to site.
  - Do not require a deploy.

Siemens Digital Industries Software customization support

Siemens Digital Industries Software is committed to maintaining compatibility between Teamcenter product releases. If you customize functions and methods using published APIs and documented extension points, be assured that the next successive release will honor these interfaces. On occasion, it may become necessary to make behaviors more usable or to provide better integrity. Our policy is to notify customers at the time of the release *prior* to the one that contains a published interface behavior change.

Siemens Digital Industries Software does not support code extensions that use unpublished and undocumented APIs or extension points. All APIs and other extension points are unpublished unless documented in the official set of technical manuals and help files issued by Siemens Digital Industries Software.

The Teamcenter license agreements prohibit reverse engineering, including: decompiling Teamcenter object code or bytecode to derive any form of the original source code; the inspection of header files; and the examination of configuration files, database tables, or other artifacts of implementation. Siemens Digital Industries Software does not support code extensions made using source code created from such reverse engineering.

Syntax definitions

This manual uses a set of conventions to define the syntax of Teamcenter commands, functions, and properties. Following is a sample syntax format:

harvester_jt.pl [*bookmark-file-name bookmark-file-name* ...]
[*directory-name directory-name* ...]

The conventions are:

Bold	Bold text represents words and symbols you must type exactly as shown. In the preceding example, you type **harvester_jt.pl** exactly as shown.
Italic	Italic text represents values that you supply. In the preceding example, you supply values for *bookmark-file-name* and *directory-name*.
text-text	A hyphen separates two words that describe a single value. In the preceding example, *bookmark-file-name* is a single value.
[ ]	Brackets represent optional elements.
...	An ellipsis indicates that you can repeat the preceding element.

Following are examples of correct syntax for the **harvester_jt.pl:** command:

```
harvester_jt.pl
harvester_jt.pl assembly123.bkm
harvester_jt.pl assembly123.bkm assembly124.bkm assembly125.bkm
harvester_jt.pl AssemblyBookmarks
```

2. Rich client customization using style sheets

Introduction to style sheets

If you change style sheets, clear the client's cache to see the style sheet changes. Exit the rich client and restart it using the **-clean** command argument to remove the old configuration from cache.

If using the rich client to edit or register XML rendering templates (style sheets), you must have DBA privileges.

Why use style sheets?

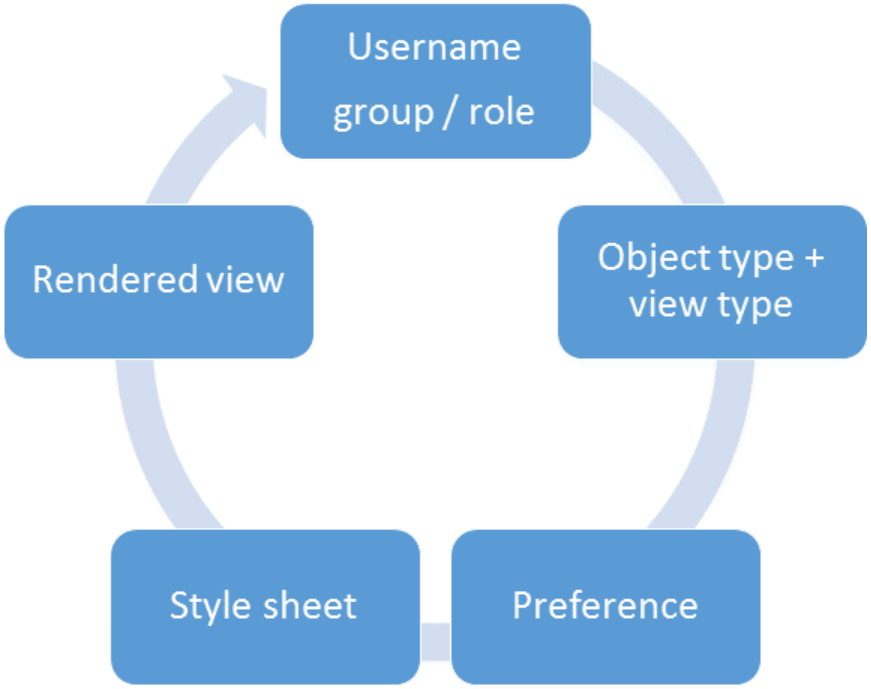
You can control the layout of the user interface based on the type of object being viewed and the credentials of the user.

- You can customize using configuration instead of coding.
- The style sheets are stored on the server. No deploy required.

Each commonly-used business object type (such as item revisions, folders, datasets, and so on) has a style sheet that defines the layout of its properties in the user interface.

How do they work?

Following are the steps Teamcenter takes to render a view using XRT.



1. The object type being viewed and the viewer being used determine the name of the preference that will be used. For example: if a **DocumentRevision** object is being viewed in the **Summary** viewer, the preference used is **DocumentRevision.SUMMARYRENDERING**

2. Since the preferences used for style sheet registration are hierarchical, a group, role, or user preference of the same name would override the site default. For example, if there is a role preference named **DocumentRevision.SUMMARYRENDERING**, it would override the site preference.

3. The appropriate preference is retrieved. It contains the name of the **XMLRenderingStylesheet** dataset that contains the rendering information.

4. The **XMLRenderingStylesheet** dataset is retrieved by name. Its contents are parsed and the view layout is generated.

5. The view is rendered and displayed to the user.

Where are they used?

Style sheets are used in the following locations.

- **Property** dialog box

  Preference syntax: `.RENDERING`

  Defines the layout of the **Properties** dialog box, and the **View** tab when showing properties.

- **Summary** view

  Preference syntax: `.SUMMARYRENDERING`

  Defines the layout of the **Summary** tab.

- **Create** dialog box

  Preference syntax: `.CREATERENDERING`

  Defines the layout of the dialog box used when the user chooses **File→New→Other** and some portions of the dialog box when the user chooses **File→New→***object*.

  To add a property to an operation dialog box, you must not only add the property to the registered style sheet but also to the operation input in the Business Modeler IDE.

- **Save As** dialog box

  Preference syntax: `.SAVEASRENDERING`

  Defines the layout of the **Save As** dialog box.

  To add a property to an operation dialog box, you must not only add the property to the registered style sheet but also to the operation input in the Business Modeler IDE.

- **Revise** dialog box

  Preference syntax: `.REVISERENDERING`

  Defines the layout of the **Revise** dialog box.

  To add a property to an operation dialog box, you must not only add the property to the registered style sheet but also to the operation input in the Business Modeler IDE.

- **Form** display

  Preference syntax: `.FORMRENDERING`

  Defines the layout of forms, such as the **Item Master** form or the **Item RevisionMaster** form.

How are they registered?

The preferences that register style sheets have the following generic syntax.

```
objectType.RENDERINGTYPE
```

objectType

Refers to the database (non-localized) name of the object, as found in the Business Modeler IDE. For example, the preference that controls the rendering for the Summary view when the user selects a Document Revision is **DocumentRevision.SUMMARYRENDERING**. If the object type selected does not have a preference, then Teamcenter will look for a preference for the object's parent in the POM. In this case, the parent of DocumentRevision is ItemRevision, so if **DocumentRevision.SUMMARYRENDERING** does not exist, then **ItemRevision.SUMMARYRENDERING** will be retrieved instead.

RENDERINGTYPE

One of the supported rendering locations.

What is the syntax of a style sheet?

Style sheets are XML documents stored in **XMLRenderingStylesheet** datasets. Even though they are commonly called style sheets, they are not CSS or XSL style sheets. They are more accurately called XML rendering templates (XRT), but both terms are used in this documentation. The XML code allows you to define a subset of properties to display, the display order, the user interface rendering components to be used, and more.

Which style sheets are provided?

To see all the available style sheets, search Teamcenter for **XMLRenderingStylesheet** datasets.

Registering a style sheet

Style sheet preferences are hierarchical preferences that take a single string value. You can set their value using any available method, they are no different than any other preference. Following are the most common:

Registering a style sheet using the command line

You can use the **preferences_manager** command line utility to import or modify style sheet preferences like any other preference. This utility provides all of the functionality you need when working with preferences, without the restrictions you may find in some of the UI helpers.

Registering a style sheet using the rich client preferences interface

You can use the **Edit→Options** interface to register style sheet preferences, just like any other preference.

Registering a style sheet using the rich client viewer tab

The rich client has a special interface for editing style sheet content that can also modify the registration information. When you have an **XMLRenderingTemplate** dataset highlighted, the **View** tab becomes

an IDE of sorts for editing the XML content. This interface saves the XML edits as well as creating or modifying the preference based on the settings.

An example of preference hierarchy

Everything in this example is based on a single preference, one which registers a style sheet to a business object for the summary view. It could be any preference as all preferences behave the same way. Since this preference definition's protection scope is **User**, you can create instances at the **Site**, **Group**, **Role**, and **User** location. This means you can control its value based on your users' current group, role, or even user name.

Example: I want the summary view's property layout for item revisions to depend on my users' login information

Following are the details of this example.

- You have three groups: Engineering, Manufacturing, and Testing.
  Each group has three roles: Manager, Designer, and Viewer.
- You want a default style sheet that everyone will use unless otherwise specified.
- Your technical users need an extended set of properties.
- Your managers need a page of workflow information.
- Your designers need classification information.
- You have users that just need a simplified layout for viewing.
- You have Conner. Conner is a power-user.
  Conner needs a special layout regardless of which group or role he's in.

Style sheet datasets

Five style sheet datasets are considered.

ItemRevSummary

Configured to be the default style sheet for the Item Revision summary page. This applies to everyone unless overridden.

IRSumTech

Configured to provides the extra properties for the Engineering and Manufacturing groups, but not for any other groups.

IRSumMgr

Configured to display workflow information for the Manager role, regardless of group.

IRSumDes

Configured to show the classification trace for the Designer role, regardless of group.

ConnersIRSum

Configured for Conner. Conner has his own requirements

Preference instances

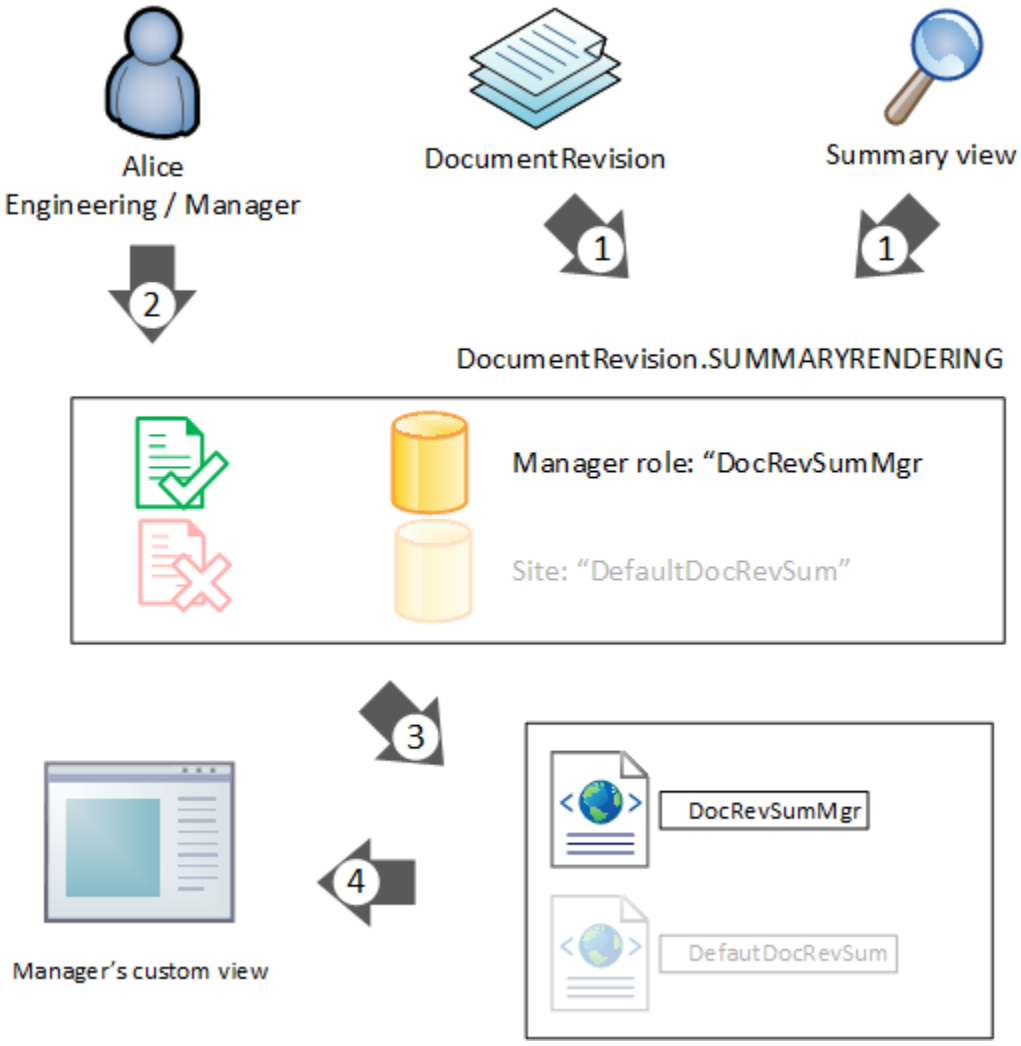
Assign the style sheets to the various groups and roles, and even users if desired, by creating each preference instance with the value pointing to the respective style sheet. In this example, there are 6 preference instances created.

User preferences	Conner: ConnersIRSum
Role preferences	Manager: IRSumMgr
	Designer: IRSumDes
Group preferences	Engineering: IRSumTech
	Manufacturing: IRSumTech
Site preference	*value:* ItemRevSum

The Viewer role and the Tester group have no preference instances created for their location.

How does Teamcenter choose which preference to use?

In this example, Alice selects a **DocumentRevision** business object and uses the **Summary** tab. When she does this, Teamcenter performs a few steps to determine which style sheet to use.



1. Based on the object type and the view location, the system knows the name of the preference instances to retrieve.

   In this example, **DocumentRevision.SUMMARYRENDERING**.

   There are two instances: one at the **Site** location, and one at the Manager **Role** location.

2. Based on the user's current session information, Teamcenter chooses the appropriate preference instance.

   Less specific locations are overridden by more specific locations.

3. The value of the chosen preference instance is read, providing the name of the style sheet to retrieve.

4. Teamcenter uses the style sheet to render the view.

Result

Your users see a different set of information based on what group or role they are in because the client uses different style sheets.

User - Group / Role	Preference instance build-up	Resulting style sheet
Alice — Engineering / Manager	Alice: *none* **Manager: IRSumMgr** Engineering: IRSumTech Site: ItemRevSum	IRSumMgr
Ted — Manufacturing / Manager	Ted: *none* **Manager: IRSumMgr** Manufacturing: IRSumTech Site: ItemRevSum	IRSumMgr
Sue — Testing / Manager	Sue: *none* **Manager: IRSumMgr** Testing: *none* Site: ItemRevSum	IRSumMgr
Bob — Engineering / Designer	Bob: *none* **Designer: IRSumDes** Engineering: IRSumTech Site: ItemRevSum	IRSumDes
Carol — Engineering / Viewer	Carol: *none* Viewer: *none* **Engineering: IRSumTech** Site: ItemRevSum	IRSumTech
Pat — Testing / Viewer	Pat: *none* Viewer: *none* Testing: *none* **Site: ItemRevSum**	ItemRevSum
Conner — Engineering / Manager	**Conner: ConnersIRSum** Manager: IRSumMgr Engineering: IRSumTech Site: ItemRevSum	ConnersIRSum
Conner — Testing / Viewer	**Conner: ConnersIRSum** Viewer: *none* Testing: *none* Site: ItemRevSum	ConnersIRSum

- Alice sees the style sheet for Managers because she does not have a user preference set to supersede it. The site preference is overridden by the Engineering group preference, which is overridden by the Manager role preference. Ted has the same result; the Manufacturing group preference is overridden by the Manager preference. Sue doesn't have a group preference, but she still gets the Manager role preference.

- Bob sees the style sheet for Designers because of his role, similar to the preceding example.

- Carol sees the tech style sheet because there is no role preference for Viewers.
- Pat's group and role do not have preferences associated with them, and neither does she have a user preference, so she gets the default style sheet defined by the site preference.
- Conner gets Conner's style sheet regardless of which group or role he's in, since a user preference supersedes all others.

How does the rich client style sheet viewer work?

The rich client's **Viewer** tab becomes an editor when the user selects an **XMLRenderingTemplate** dataset.

Rich client procedure

The rich client follows the following steps:

1. Search for a ***REGISTEREDTO** helper preference.

   These preferences are based on the name of the dataset, and the value tells the system to which object type the dataset is registered.

2. Use the helper preference's value to search for the corresponding ***RENDERING** preference.

3. Display the XML content of the dataset, and populate the two fields with the registration information.

   If none of the registration preferences were found, the rich client leaves them blank.

4. When the user chooses **Apply**, the rich client saves the dataset, and modifies or creates the appropriate registration and helper preference.

Style sheet registration helper preferences

Following are the helper preferences in the order they are searched.

- *{datasetName}*.**FORM_REGISTEREDTO**

  Corresponds to *{typeName}*.**FORMRENDERING**.

  The dialog box for a **Form** object.

- *{datasetName}*.**SUMMARY_REGISTEREDTO**

Corresponds to *{typeName}*.**SUMMARYRENDERING** .

The **Summary** view tab.

- *{datasetName}*.**CREATE_REGISTEREDTO**

  Corresponds to *{typeName}*.**CREATERENDERING**.

  The **Create** command panel.

- *{datasetName}*.**SAVEAS_REGISTEREDTO**

  Corresponds to *{typeName}*.**SAVEASRENDERING**.

  The **Save As** command panel.

- *{datasetName}*.**REVISE_REGISTEREDTO**

  Corresponds to *{typeName}*.**REVISERENDERING**.

  The **Revise** command panel.

- *{datasetName}*.**REGISTEREDTO**

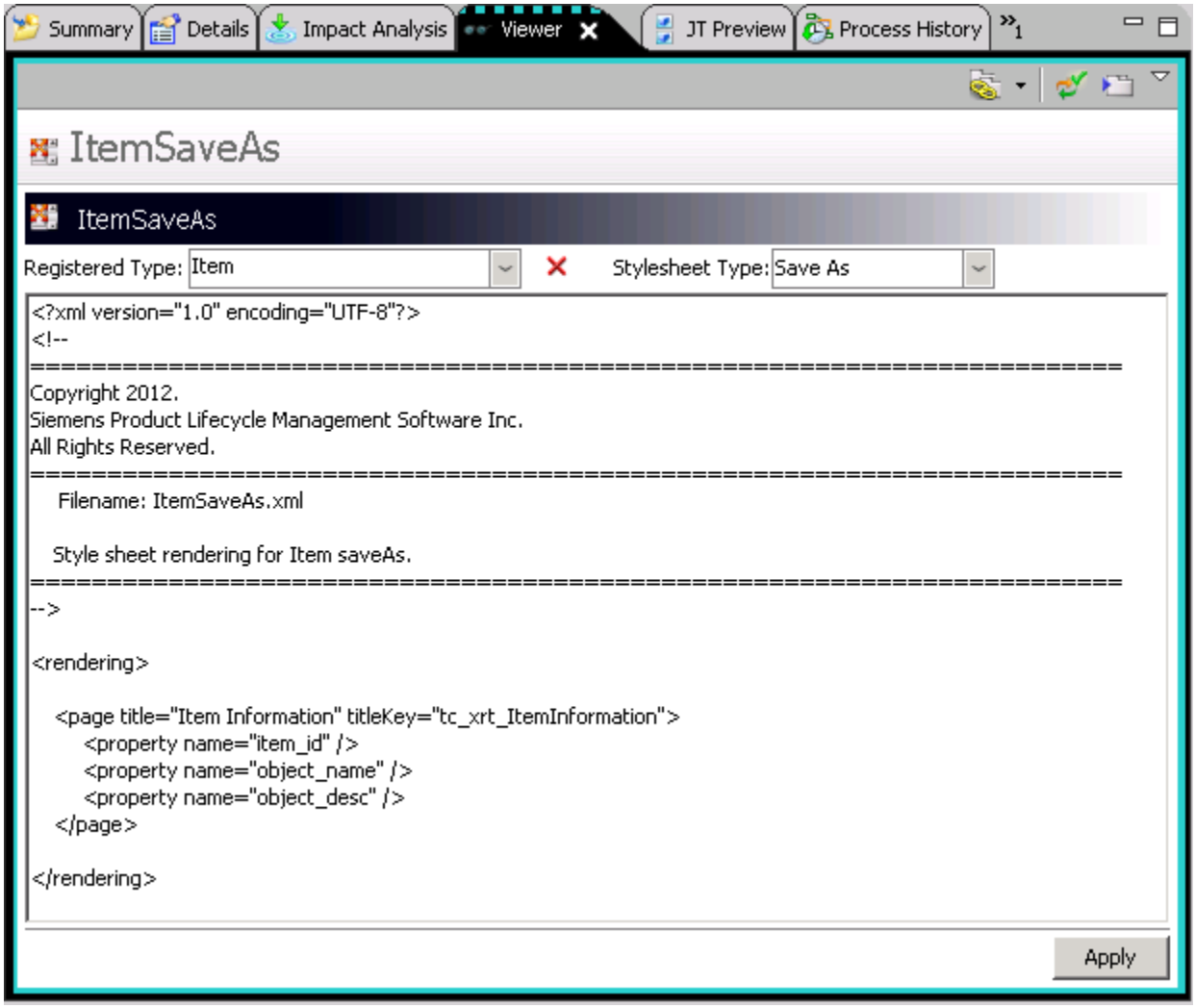
  Corresponds to *{typeName}*.**RENDERING**.

  The properties window. Either the **View Properties** dialog box, or when the **Viewer** tab shows object properties.

Example

For example, in a new installation of Teamcenter, the **ItemSaveAs** dataset might be registered to the **Save As** operation of the **Item**. If you highlight the **ItemSaveAs** dataset and switch to the **Viewer** tab, you will see its registration information in the **Registered Type** and **Stylesheet Type** fields.

When you select an **XMLRenderingStylesheet** dataset with the **Viewer** tab active, the rich client reads the *{datasetName}*.**REGISTEREDTO** preference to see



If you change the values in these fields and then click **Apply**, then the appropriate preference will be created or updated.

the value of the **Item.SAVEASRENDERING** preference. Its value will be **ItemSaveAs**.

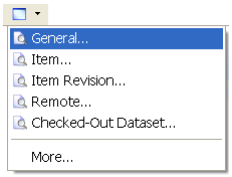
Sample customizations using style sheets

Search for style sheets

To find style sheets in the rich client, search for **XMLRenderingStylesheet** datasets.

1. Click the **Open Search View** button .

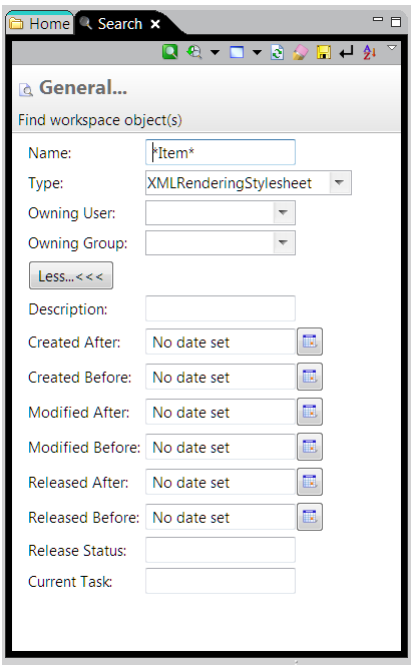
2. Click the arrow on the **Select a Search** button and choose **General**.



Starting the search for style sheets

3. In the **Type** box, type **XMLRenderingStylesheet**.

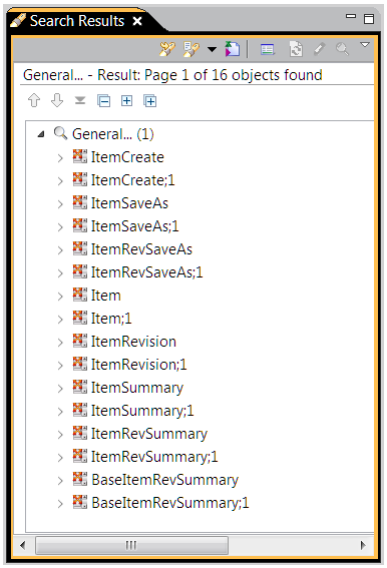
   If you are looking for style sheets for a particular kind of object, enter a string in the **Name** box to look for those kinds of style sheets. For example, if you want to find all style sheets for items or item revisions, type ***Item*** in the **Name** box.



Searching for XMLRenderingStylesheet datasets

4. Press the Enter key or click the **Execute the Search** button .

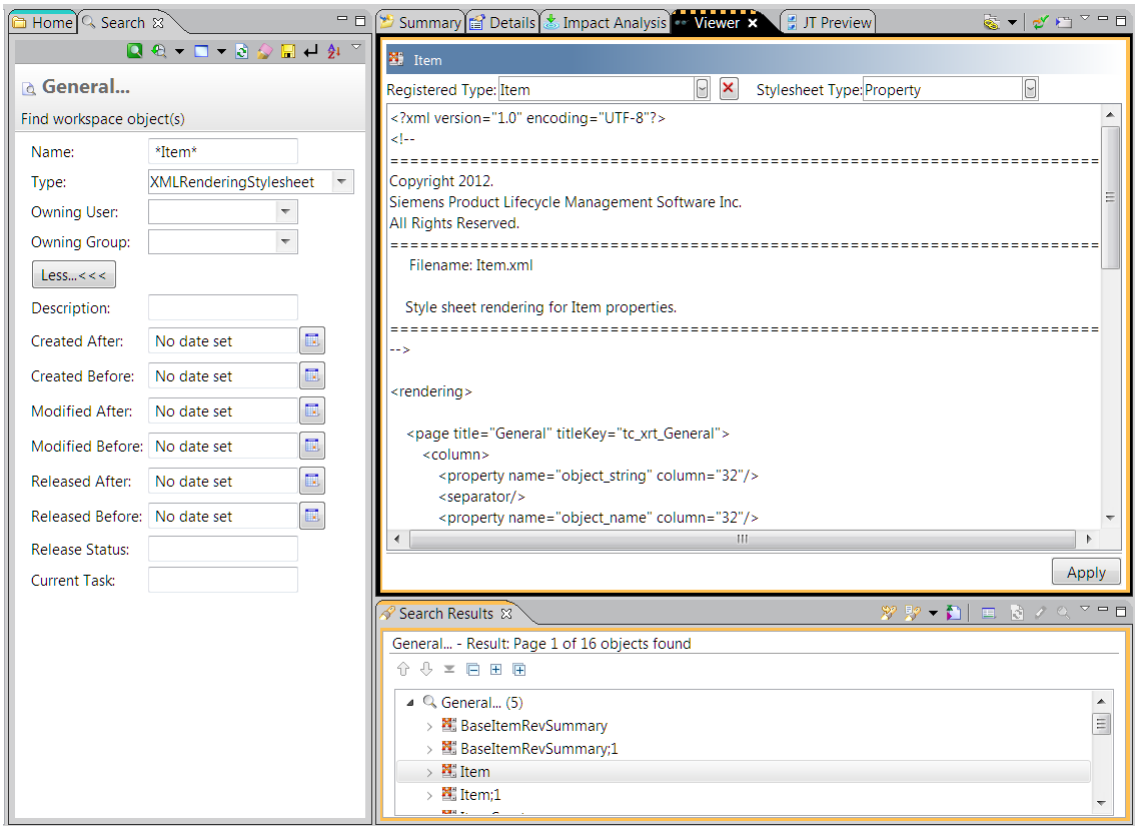
   The results are displayed in the **Search Results** view.



Viewing the search results for XMLRenderingStylesheet datasets

5. In the **Search Results** tab, select the style sheet you want to view. Click the **Viewer** tab to see the style sheet.

> Note:
>
> Do not double-click a style sheet (**XMLRenderingStylesheet** dataset file) in an attempt to open it. If you do, you receive an `Unable to open` error message. Instead, select the style sheet and view its contents in the **Viewer** view.



Viewing the style sheet contents in the rich client

Create a custom style sheet based on an existing style sheet

You can create your own custom style sheet.

For example, you create a custom business object in the Business Modeler IDE and install it to the rich client, and you want to create a unique style sheet to display the custom properties. To create the style sheet, in the rich client, search for **XMLRenderingStylesheet** datasets, save one as your own custom style sheet dataset, and then register the custom style sheet for use with the custom business object.

1. In the rich client, search for a style sheet you can base your new style sheet on.

2. In the **Search Results** view, select the style sheet you want to use, choose **File→Save As**, and rename it. For example, if you want to create a style sheet to be used with a custom **A5_MyItem** business object, you could name the style sheet **A5_MyItem**.

   The new style sheet dataset is saved in your **Newstuff** folder in the **Home** view and is still displayed in the **Viewer** tab.

3. Edit the style sheet.

a. Change the named reference for the file by selecting the style sheet dataset and choosing **View→Named References**. In the **Name** column in the **Named References** dialog box, change the old name of the file to the new save as file name.

b. In the **Viewer** tab, click the arrow in the **Registered Type** box and select the business object type you want to register it to. For example, if you have a custom **A5_MyItem** business object added to your server, select **A5_MyItem** from the list.

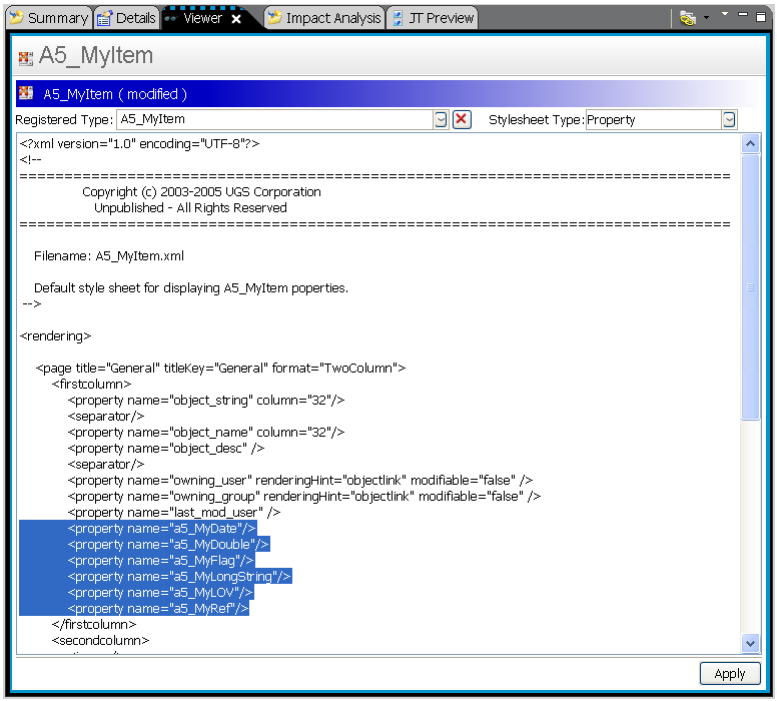
c. Edit the style sheet in the **Viewer** tab to include the elements you want displayed in the layout.

For example, if you want to display custom properties, add them where you want them to appear on the page, like this:

```
        <page title="General" titleKey="tc_xrt_General">
            <column>
                <property name="object_string" column="32"/>
                <separator/>
                <property name="object_name" column="32"/>
                <property name="object_desc" />
                <separator/>
                <property name="owning_user" renderingHint="objectlink"
modifiable="false" />
                <property name="owning_group" renderingHint="objectlink"
modifiable="false" />
                <property name="last_mod_user" />
                <property name="a5_MyDate"/>
             <property name="a5_MyDouble"/>
             <property name="a5_MyFlag"/>
             <property name="a5_MyLongString"/>
             <property name="a5_MyLOV"/>
             <property name="a5_MyRef"/>
            </column>
            <column>
                <image/>
            </column>
        </page>
```

d. To change the style sheet type, click the arrow in the **Stylesheet Type** box and choose the desired type.

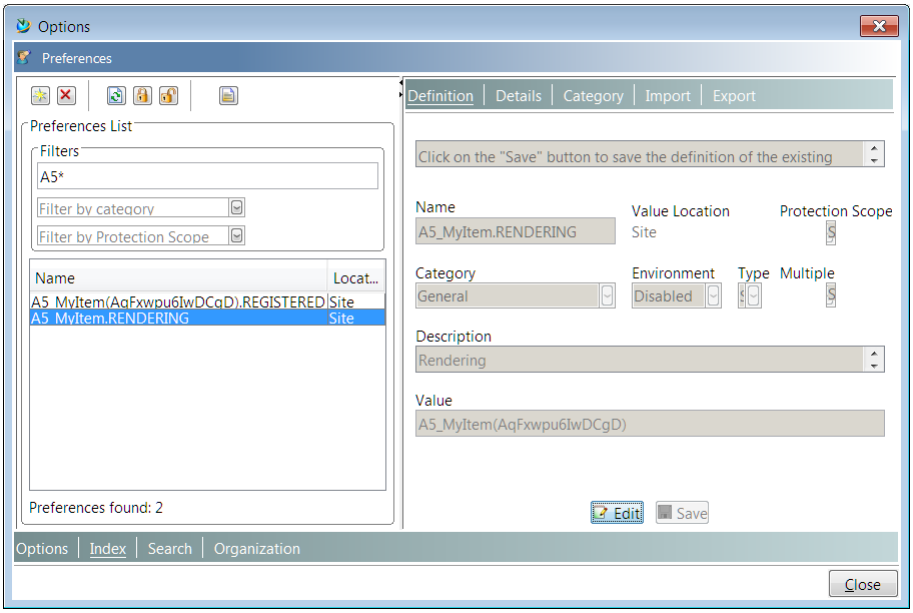
4. When you are done making changes, click the **Apply** button in the lower right corner of the view.



Create a custom style sheet

Because you used the **Registered Type** box on the **Viewer** tab to register the style sheet with a business object type, two new preferences are created (a **REGISTEREDTO** preference and a **RENDERING** preference). These preferences apply the style sheet to the business object so that the style sheet is displayed in the situation you set it for (for example, for display of the business object's property, summary, form, or create information).

5. To see the two new preferences, choose **Edit→Options** and at the bottom of the dialog box, click **Search**.



Viewing the <dataset_name>.REGISTEREDTO and <type_name>.RENDERING preferences

Notice how in the **Current Values** box of the *<type_name>***.RENDERING** preference there is a number in parentheses after the name of the business object. That is a GUID number that identifies that unique business object, and that GUID number is set in the *<dataset_name(dataset-UID)>***.REGISTEREDTO** preference. This ensures that the style sheet is applied to the correct business object.

6. To see the style sheet changes in the clients, clear the client's cache. Exit the rich client and restart it using the **-clean** command argument to remove the old configuration from cache.

   To make the new style sheets available for quick loading to clients, run the **generate_client_meta_cache** utility as a Teamcenter Administrator to add the new style sheets to client cache, for example:

   ```
   generate_client_meta_cache -u=... -p=... -g=... update stylesheets
   ```

Create a custom style sheet by importing an XML file

You can create a new style sheet by creating a new dataset and associating it with an XML file.

1. In the rich client My Teamcenter application, create a new dataset of type **XMLRenderingStylesheet**.

2. Import the XML file as a named reference to the dataset as follows:

   a. Right-click the dataset in the navigation tree and choose **Named References**.

      Teamcenter displays the **Named References** dialog box.

b. Click the **Add** button.

Teamcenter displays the **Add File** dialog box.

c. Locate and select the XML file in your operating system directory and click **Add**.

Teamcenter displays the XML file in the **Named References** dialog box.

d. Click **Close**.

3. Register the style sheet by setting the **Form.REGISTEREDTO** preference.

4. To see the style sheet changes in the clients, clear the client cache. Exit the rich client and restart it using the **-clean** command argument to remove the old configuration from cache.

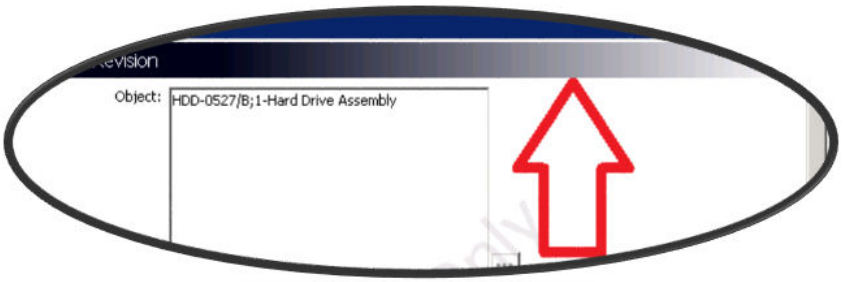
Localizing style sheets

Localization is the translation of text into the local language. XML rendering templates (XRTs), also known as style sheets, have their text localized by the TextServer. When you customize a style sheet with new text, you must enter the text to these file in the different languages to be displayed in the user interface. If you do not enter localized text strings, the unlocalized text appears in the user interface between exclamation marks (for example: **!MyText!**) except when in the **command** tag.

Use SWT instead of Swing

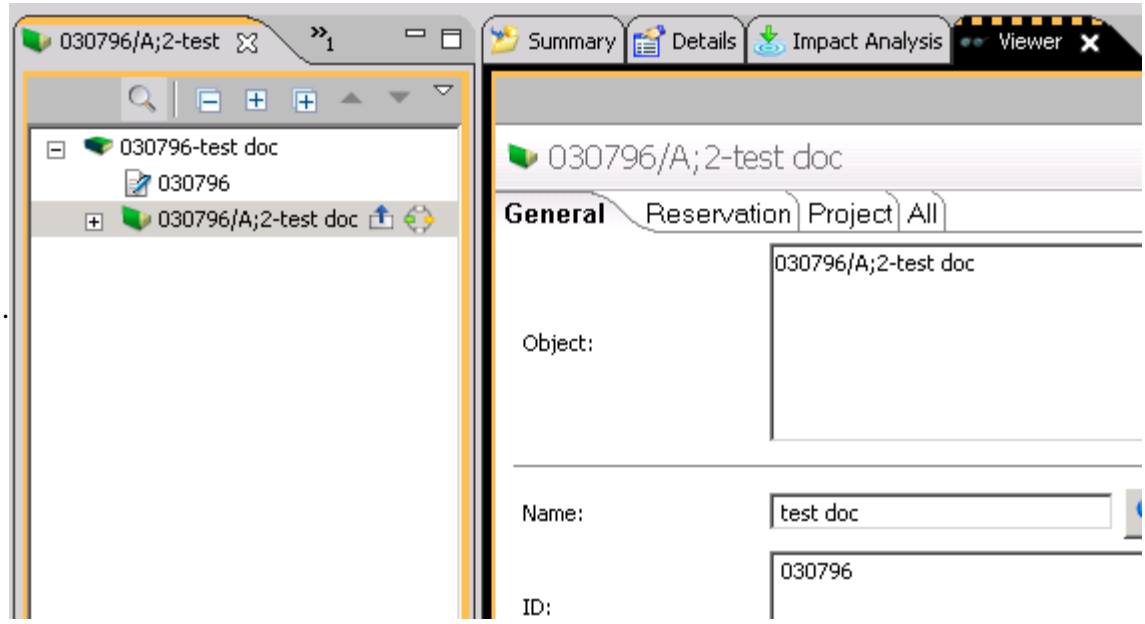
Teamcenter is moving toward SWT/JFace as the user interface toolkit and moving away from AWT and Swing. Siemens Digital Industries Software encourages you to customize Teamcenter using SWT/JFace components. Siemens Digital Industries Software will discontinue Swing/AWT support in a future version. Because of the need to support legacy functionality, you may still encounter some Swing panels when using the rich client

Swing panels can be identified by the graduated bar at the top of the panel.

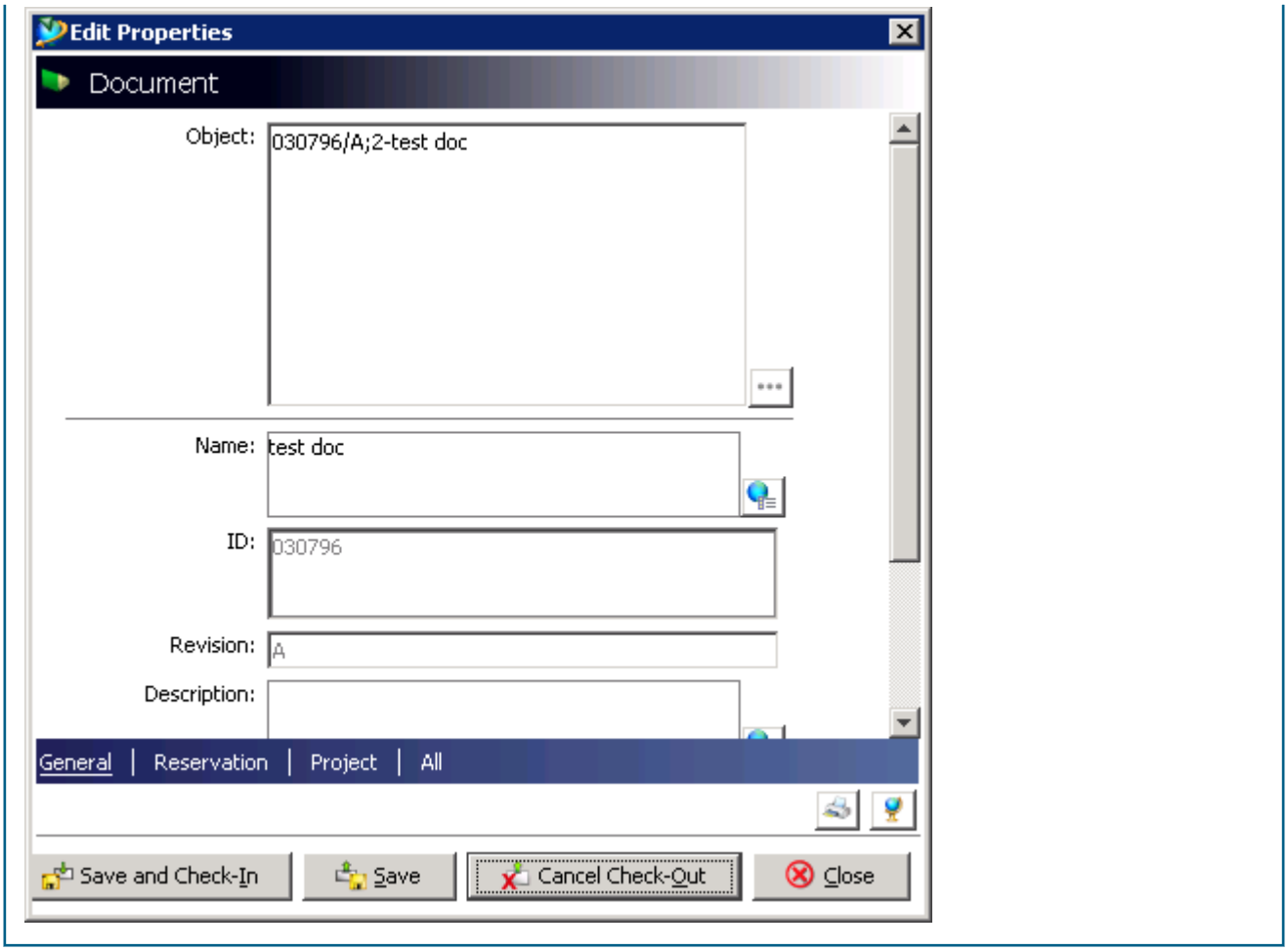


Example:

When viewing an object's properties, select it and then use the **Viewer** tab. The properties displayed in this view are rendered using SWT, but controlled by XML rendering templates (XRT), also known as style sheets. XRT is easily editable, does not need to be deployed, and can easily be tailored to any group, role, or user.



If you **View properties** using the context menu, the dialog box presented is controlled by the same XRT, but the underlying technology is Swing, and therefore limited in its functionality. Whenever possible, encourage your users to use the **Viewer** tab instead of the dialog box.



Style sheet definition

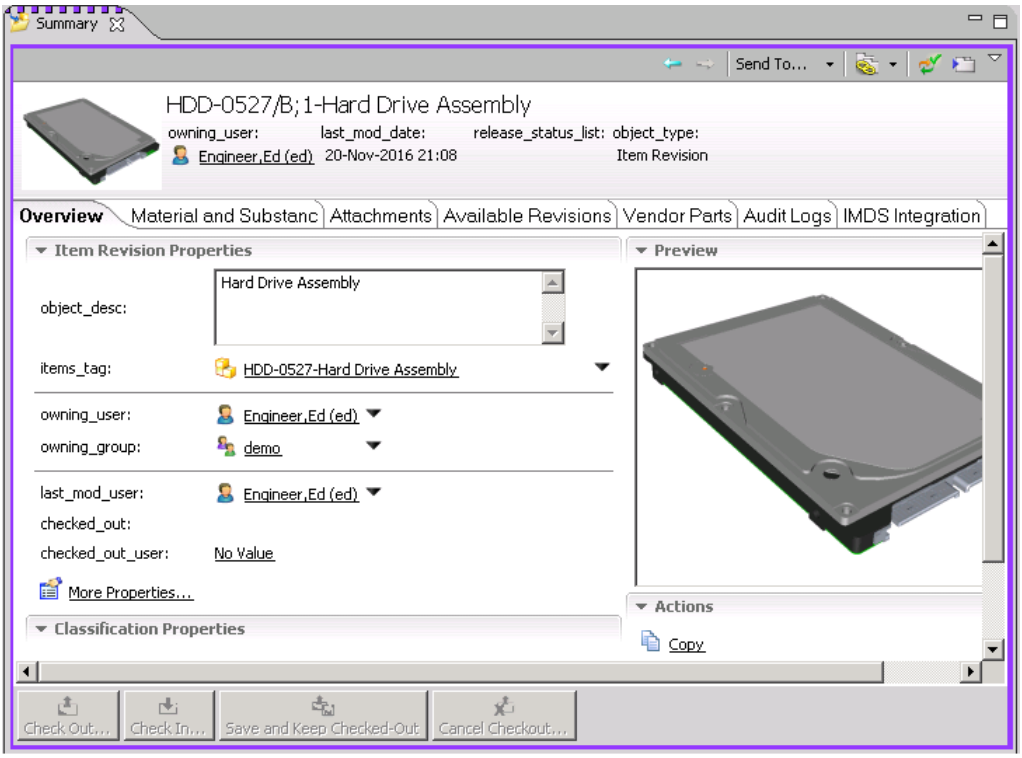
Rich client view rendering

In the Teamcenter rich client, the Eclipse view consists of several components controlled by an **XMLRenderingStylesheet** dataset. Following is a brief explanation of the various components of the rendering system.

View rendering

An individual view tab is defined by a `<rendering>` tag, and consists of two main components, the **header** and the **page**.

<rendering>

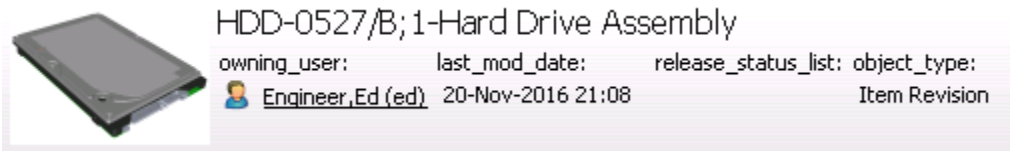



```
<rendering>
   <header>
   <page titleKey = ...>
   <page titleKey = ...>
   <page titleKey = ...>
```

View header

You control what is displayed in the header by modifying what is defined within the **<header>** tag.

<header>



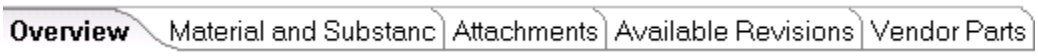

```
<header>
   <image source = "thumbnail"/>
   <property name = "owning_user"/>
   <property name = "last_mod_date"/>
   <property name = "release_status_list"/>
   <property name = "object_type"/>
```

View page

You control the number of pages within the view, their names, and when they appear using the **<page>** tag.

You can use **columns**, **sections**, **properties**, and **commands** within your page.

<page>

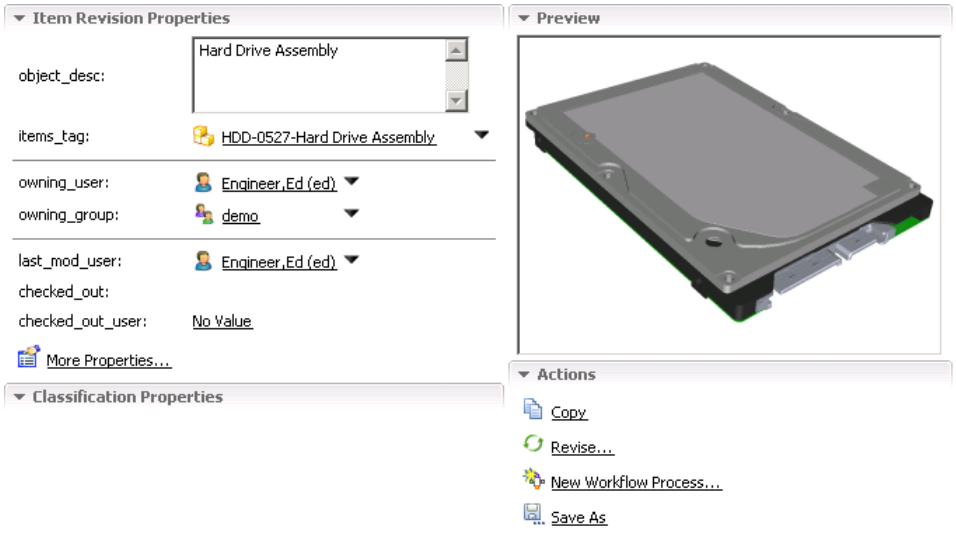



```
<page titleKey="tc_xrt_Overview">
<page titleKey = "tc_xrt_Mat1MaterialAndSubstanceInfo"
      title = "Material and Substance Information"
      visibleWhen = "Mat1UsesMaterial != null">
<page title = "Attachments" titleKey="tc_xrt_attachments">
<page titleKey="tc_xrt_AvailableRevisions">
<page title="Related Links" titleKey="tc_xrt_RelatedLinks"
      visibleWhen="{pref:LIS_RelatedLinkTabVisible}==true">
<page titleKey = "tc_xrt_VendorParts" title = "Vendor Parts  "
      visibleWhen = "VMRepresents==null">
<page titleKey = "tc_xrt_VendorParts*" title = "Vendor Parts *"
      visibleWhen = "VMRepresents!=null">
```

Page section

You may divide your page into sections. Their names and their order on the page is controlled using the **<section>** tag.

<section>




```
<section titleKey="tc_xrt_ItemRevProperties">
<section titleKey="tc_xrt_ClassificationProperties">
<section titleKey="tc_xrt_Preview">
<section titleKey="tc_xrt_actions" commandLayout="vertical">
```

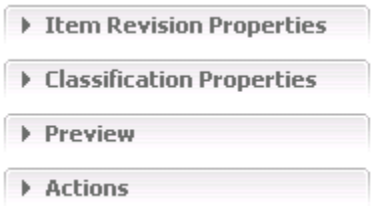
Page columns

You may divide your sections on your page into columns by wrapping them in <column> tags. If you choose to use columns, the recommended limit is two.

<column>


```
<page titleKey="tc_xrt_Overview">
   <column>
      <section titleKey="tc_xrt_ItemRevProperties">
      <section titleKey="tc_xrt_ClassificationProperties">
   <column>
      <section titleKey="tc_xrt_Preview">
      <section titleKey="tc_xrt_actions" commandLayout="vertical">
```

No columns

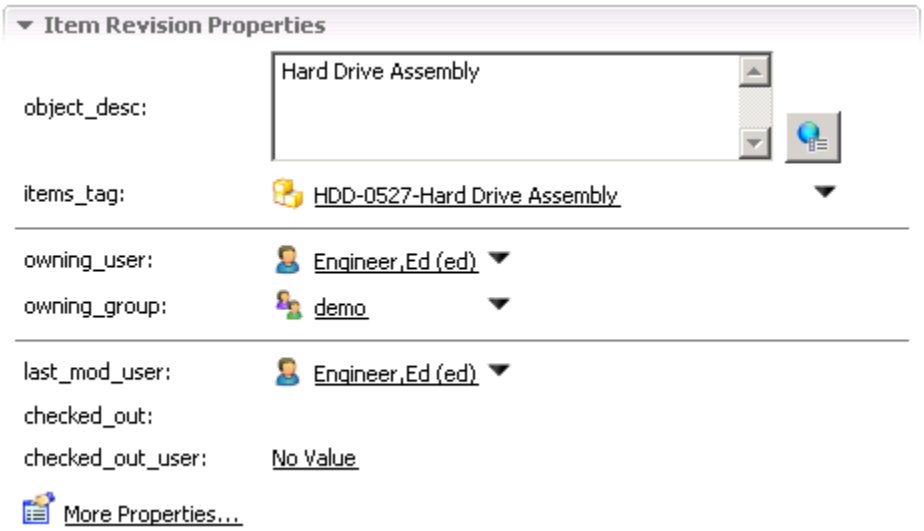



```
<page titleKey="tc_xrt_Overview">
   <section titleKey="tc_xrt_ItemRevProperties">
   <section titleKey="tc_xrt_ClassificationProperties">
   <section titleKey="tc_xrt_Preview">
   <section titleKey="tc_xrt_actions" commandLayout="vertical">
```

Section and header properties

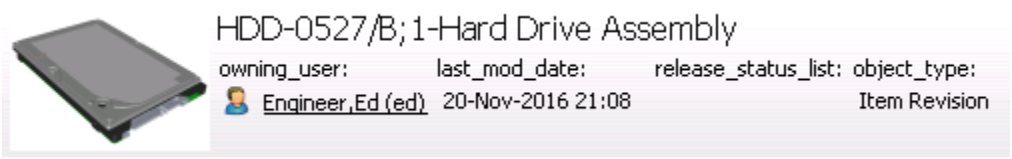
You control which properties are displayed within a section or header using the **<property>** tag.

<property> in a section




```
<section titleKey="tc_xrt_ItemRevProperties">
   <property name="object_desc"/>
   <property name="items_tag"/>
   <separator/>
   <property name = "owning_user" renderingHint = "objectlink"
             modifiable = "false"/>
   <property name = "owning_group" renderingHint = "objectlink"
             modifiable = "false"/>
   <separator/>
   <property name = "last_mod_user"/>
   <property name="checked_out"/>
   <property name = "checked_out_user"/>
   <command commandId="com.teamcenter.rac.properties"
            titleKey="tc_xrt_moreProperties"/>
</section>
```

<property> in a header

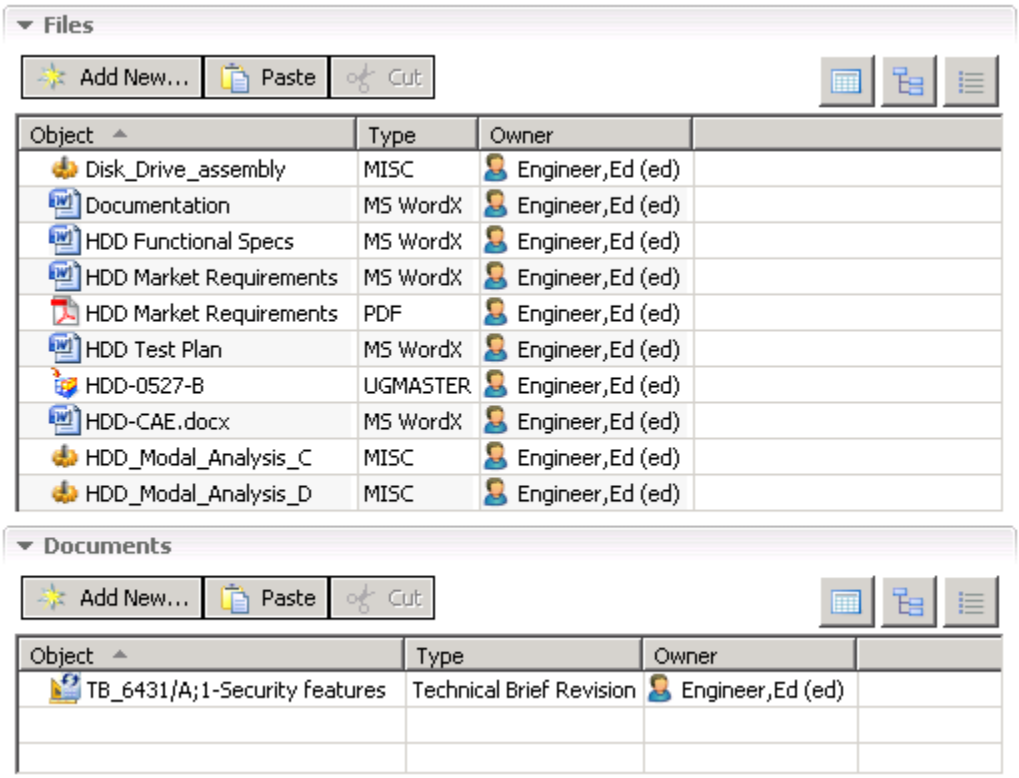



```
<header>
   <image source="thumbnail"/>
   <classificationTrace/>
   <property name = "owning_user"/>
   <property name = "last_mod_date"/>
   <property name="release_status_list"/>
   <property name = "object_type"/>
```

Property tables

You can display properties of related objects in an easy-to-read table.

<objectSet>

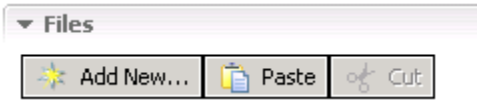



```
<page title = "Attachments" titleKey="tc_xrt_attachments">
   <section titleKey = "tc_xrt_Files" title = "Files">
      <objectSet source = "IMAN_specification.Dataset, IMAN_reference.Dataset, ...
   <section titleKey = "tc_xrt_Documents" title = "Documents">
      <objectSet source = "IMAN_specification.DocumentRevision" ...
```

Commands

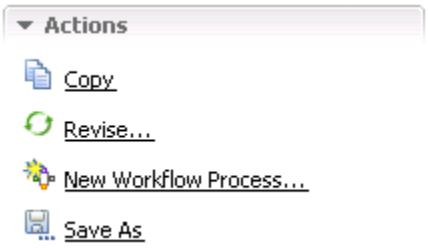
You can add commands to your sections or your tables.

<command> in table




```
<objectSet source = "IMAN_specification.Dataset" ...
    <tableDisplay ...
    <treeDisplay ...
    <thumbnailDisplay/>
    <listDisplay/>
    <command commandId = "com.teamcenter.rac.common.AddNew"
    <command commandId = "com.teamcenter.rac.viewer.pastewithContext"
    <command commandId = "org.eclipse.ui.edit.cut"
```

<command> in a section




```
<section titleKey="tc_xrt_actions" commandLayout="vertical">
   <command commandId = "com.teamcenter.rac.copy"/>
   <command commandId = "com.teamcenter.rac.revise"/>
   <command commandId="com.teamcenter.rac.newProcess" titleKey="tc_xrt_newProc"/>
   <command commandId="org.eclipse.ui.file.saveAs"/>
```

Other XML elements

You can use these elements to add additional information or visual input to your views.

<break>

You can use this element to add a horizontal space on the page. This is similar to the separator element, but with no visible line.

<classificationTrace>

You can add the classification category of an object. For example, **Home Electronics > Components > Hard Drives**.

<classificationProperties/>

You can add the classification properties of an object.

<label>

You can add static, formatted text to your page.

<customPanel>

You can add your own custom content.

<image>

You can add an image to the page.

XML elements

all

Lists all properties of the defined object.

ATTRIBUTES

type

Indicates whether to list all the object properties or only form properties. The valid values for this attribute are **property** and **form**.

SUPPORTED STYLE SHEETS

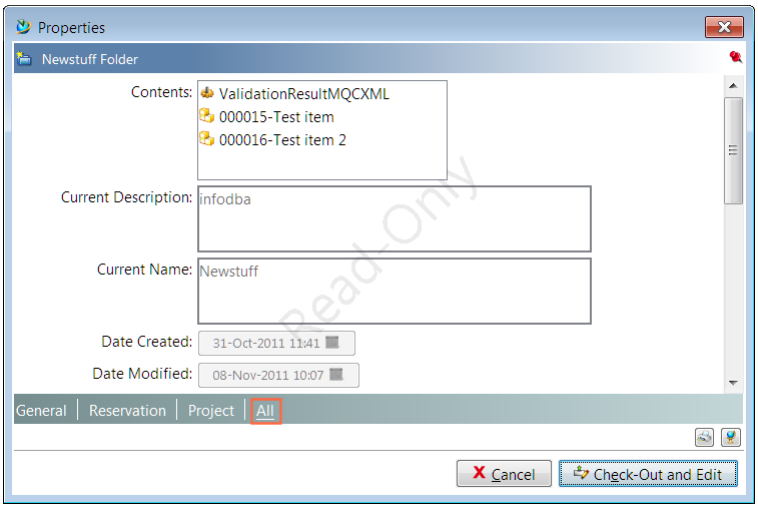
This tag can be used on the following types of style sheets (but not the New Business Object wizard):

Property
Summary
Form

EXAMPLE

Following is sample code from the **Folder.xml** XML rendering style sheet showing the **all** element:

```
<page title="All" titleKey="tc_xrt_All">
    <all type="property"/>
</page>
```



The all element in the rich client

Note:

The <**conditions**>, <**GoverningProperty**>, and <**Rules**> tags are not supported within the <**all**> tag.

break

Inserts a break in the pane.

ATTRIBUTES

None.

SUPPORTED STYLE SHEETS

This tag can be used on the following types of style sheets:

Property
Summary
Form
Create

EXAMPLE

```
<page titleKey="tc_xrt_General" title="General">
    <property name=.../>
    <property name=.../>
    <separator />
    <property name=.../>
    <property name=.../>
    <break />
</page>
```

classificationProperties

Specifies that the classification properties of the current object should be displayed. Properties and their values are rendered as name/value pairs in static text.

ATTRIBUTES

None.

SUPPORTED STYLE SHEETS

This tag can be used only on **Summary** style sheets.

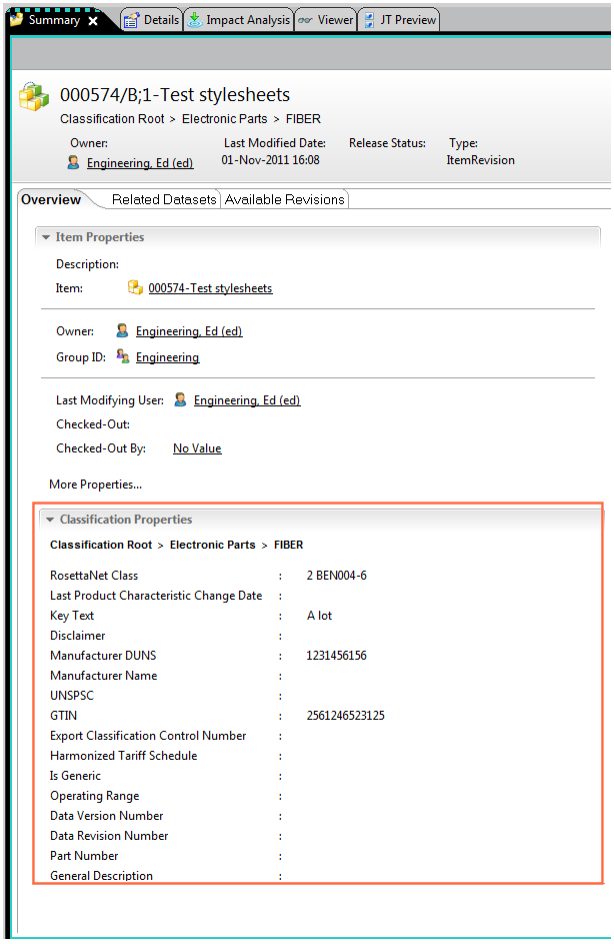
EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **classificationProperties** element:

```
    <page titleKey="tc_xrt_Overview">
        <column>
            <section titleKey="tc_xrt_AvailableRevisions">
                <objectSet source="revision_list.ItemRevision"
defaultdisplay="listDisplay"
sortdirection="descending" sortby="item_revision_id">
                    <tableDisplay>
                        <property name="object_string"/>
                        <property name="item_revision_id"/>
                        <property name="release_status_list"/>
                        <property name="last_mod_date"/>
                        <property name="last_mod_user"/>
                        <property name="checked_out_user"/>
                    </tableDisplay>
                    <thumbnailDisplay/>
                    <treeDisplay>
                        <property name="object_string"/>
                        <property name="item_revision_id"/>
                        <property name="release_status_list"/>
                        <property name="last_mod_date"/>
                        <property name="last_mod_user"/>
                        <property name="checked_out_user"/>
                    </treeDisplay>
                    <listDisplay/>
                </objectSet>
            </section>
            <section titleKey="tc_xrt_ItemProperties">
                <property name="object_desc"/>
                <separator/>
                <property name="owning_user" renderingHint="objectlink"
modifiable="false"/>
                <property name="owning_group" renderingHint="objectlink"
modifiable="false"/>
                <property name="last_mod_user"/>
                <separator/>
                <property name="checked_out"/>
```

```
                <property name="checked_out_user"/>
                <separator/>
                <command commandId="com.teamcenter.rac.properties"
titleKey="tc_xrt_moreProperties"/>
            </section>
            <section titleKey="tc_xrt_ClassificationProperties">
        <classificationProperties/>
      </section>
         </column>
```

Following is how classification properties are rendered.



classificationProperties

classificationTrace

Specifies that the classification traces of the item should be displayed, for example, **Home Care > Cleaners > Detergents**.

ATTRIBUTES

None.

SUPPORTED STYLE SHEETS

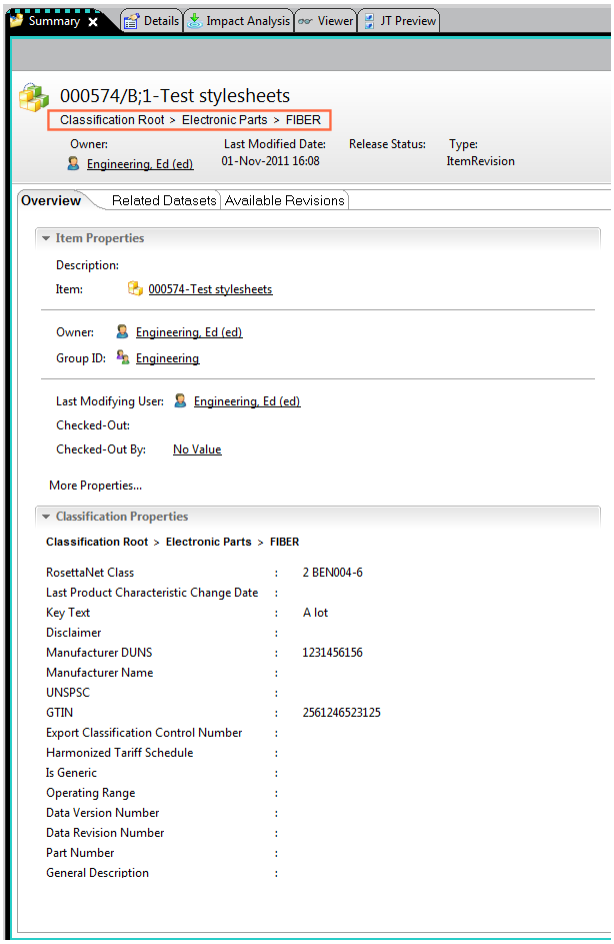
This tag can be used only on **Summary** style sheets.

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **classificationTrace** element:

```
<header>
    <image source="thumbnail"/>
    <classificationTrace/>
    <property name="owning_user"/>
    <property name="last_mod_date"/>
    <property name="release_status_list"/>
    <property name="object_type"/>
</header>
```

Following is how the classification trace is rendered



classificationTrace

column

Defines the layout of the column defined on a page.

ATTRIBUTES

None.

SUPPORTED STYLE SHEETS

This tag can be used on the following types of style sheets (but not the New Business Object wizard):

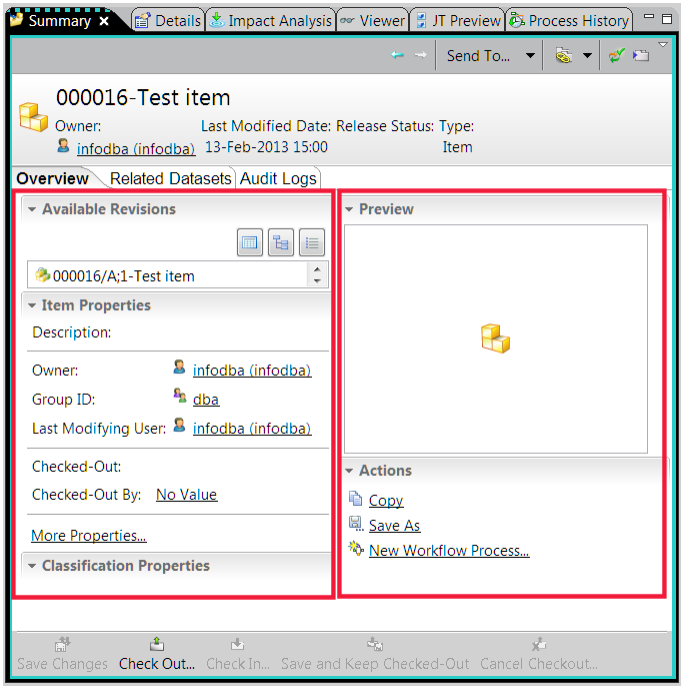
Property
Summary
Form

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **column** element:

```
<page titleKey="tc_xrt_Overview">
        <column>
            <section titleKey="tc_xrt_AvailableRevisions">
                <objectSet source="revision_list.ItemRevision"
defaultdisplay="listDisplay"
sortdirection="descending" sortby="item_revision_id">
                    <tableDisplay>
                        <property name="object_string"/>
                        <property name="item_revision_id"/>
                        <property name="release_status_list"/>
                        <property name="last_mod_date"/>
                        <property name="last_mod_user"/>
                        <property name="checked_out_user"/>
                    </tableDisplay>
                    <thumbnailDisplay/>
                    <treeDisplay>
                        <property name="object_string"/>
                        <property name="item_revision_id"/>
                        <property name="release_status_list"/>
                        <property name="last_mod_date"/>
                        <property name="last_mod_user"/>
                        <property name="checked_out_user"/>
                    </treeDisplay>
                    <listDisplay/>
                </objectSet>
            </section>
            <section titleKey="tc_xrt_ItemProperties">
                <property name="object_desc"/>
                <separator/>
                <property name="owning_user" renderingHint="objectlink"
modifiable="false"/>
                <property name="owning_group" renderingHint="objectlink"
```

```
modifiable="false"/>
                <property name="last_mod_user"/>
                <separator/>
                <property name="checked_out"/>
                <property name="checked_out_user"/>
                <separator/>
                <command commandId="com.teamcenter.rac.properties"
titleKey="tc_xrt_moreProperties"/>
            </section>
            <section titleKey="tc_xrt_ClassificationProperties">
                <classificationProperties/>
            </section>
        </column>
        <column>
            <section titleKey="tc_xrt_Preview">
                <image source="preview"/>
            </section>
            <section titleKey="tc_xrt_actions" commandLayout="vertical">
                <command actionKey="copyAction" commandId="com.teamcenter.rac.copy" />
                <command actionKey="saveAsAction"
commandId="org.eclipse.ui.file.saveAs" />
                <command actionKey="newProcessAction"
commandId="com.teamcenter.rac.newProcess"
titleKey="tc_xrt_newProc" />
            </section>
        </column>
</page>
```



column element in the rich client

command

Specifies a command representation to be displayed.

ATTRIBUTES

commandId

Specifies the command to be executed. The attribute value must be a key into a property file and must be a valid command ID. This is a string attribute that is required.

icon

Specifies the icon to be displayed before the command label. The attribute value must be a key into a property file. This is an optional attribute.

renderingHint

Specifies whether the command is rendered as a hyperlink or as a button. Valid values are **hyperlink** and **commandbutton**. This attribute is optional. If the attribute is not specified, the command is rendered as a hyperlink.

title

Specifies the default string of the title for this user interface element. This attribute is used when the string in the **titleKey** attribute is not found by the TextServer. This is an optional attribute.

titleKey

Specifies the key used by the TextServer to search for the title. If it is not defined, the string defined by the **title** attribute is used. This is an optional attribute.

tooltip

Specifies the tooltip for the command. The attribute value must be a key into a property file. This attribute is optional but is required if the **icon** attribute is specified.

SUPPORTED STYLE SHEETS

This tag can be used only on the **Summary** style sheets.

> Note:
>
> The **command** tag is ignored in the **header** section. Commands cannot be added to the toolbar.

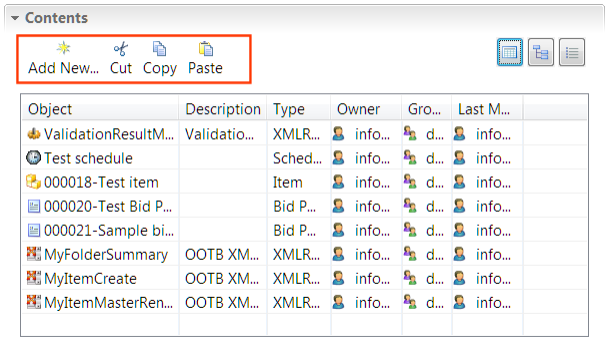
EXAMPLE

Following is sample code using the **command** element on an object set:

```
<objectSet source="contents.WorkspaceObject" defaultdisplay="thumbnailDisplay"
           sortby="object_name" sortdirection="ascending">
   <tableDisplay>
      <property name="object_string"/>
```

```
        ...
    </tableDisplay>
    <thumbnailDisplay/>
    <treeDisplay>
       <property name="object_string"/>
        ...
    </treeDisplay>
    <listDisplay/>
    <command commandId="com.teamcenter.rac.common.AddNew" renderingHint="commandbutton"/>
    <command commandId="org.eclipse.ui.edit.cut" renderingHint="commandbutton">
      <parameter name="localSelection" value="true"/>
    </command>
    <command commandId="com.teamcenter.rac.copy" renderingHint="commandbutton"/>
    <command commandId="com.teamcenter.rac.viewer.pastewithContext" renderingHint="commandbutton"/>
</objectSet>
```

In this example, the **command** element adds the **Add New**, **Cut**, **Copy**, and **Paste** buttons in the user interface.



command element

customPanel

Embeds a custom panel in a dialog box.

ATTRIBUTES

java

Specifies the fully qualified Java implementation class name responsible for building the custom user interface (for example, **com.teamcenter.rac.MyCustomPanel**). This is supported in the rich client only.

js

Specifies a JavaScript function that generates the HTML (for example, **<customPanel js=”MyCustomCalendar()” />**).

SUPPORTED STYLE SHEETS

This tag can be used on the **Create** style sheets.

EXAMPLE

Following is a custom style sheet that uses the **customPanel** element. (This example works in the rich client only.)

```
<rendering>
    <page title="General" titleKey="tc_xrt_General">
        <section title="Item Information" titleKey="tc_xrt_ItemInformation">
            <property name="item_id" />
            <property name="revision:item_revision_id" />
            <property name="object_name" />
            <property name="object_desc" />
            <separator/>
            <property name="uom_tag" />
          <separator/>
          <customPanel java="com.teamcenter.rac.ui.commands.newbo.mypanel.MyPanel" />

        <section title="Additional Item Information"
titleKey="tc_xrt_AdditionalItemInformation" initialstate="collapsed">
            <property name="IMAN_master_form:project_id" />
            <property name="IMAN_master_form:previous_item_id"  />
            <property name="IMAN_master_form:serial_number" />
            <property name="IMAN_master_form:item_comment" />
            <property name="IMAN_master_form:user_data_1"  />
            <property name="IMAN_master_form:user_data_2"  />
            <property name="IMAN_master_form:user_data_3"  />
        </section>

        <section title="Item Revision Information"
titleKey="tc_xrt_ItemRevisionInformation" initialstate="collapsed">
            <property name="revision:IMAN_master_form_rev:project_id" />
            <property name="revision:IMAN_master_form_rev:previous_version_id" />
            <property name="revision:IMAN_master_form_rev:serial_number" />
            <property name="revision:IMAN_master_form_rev:item_comment" />
```

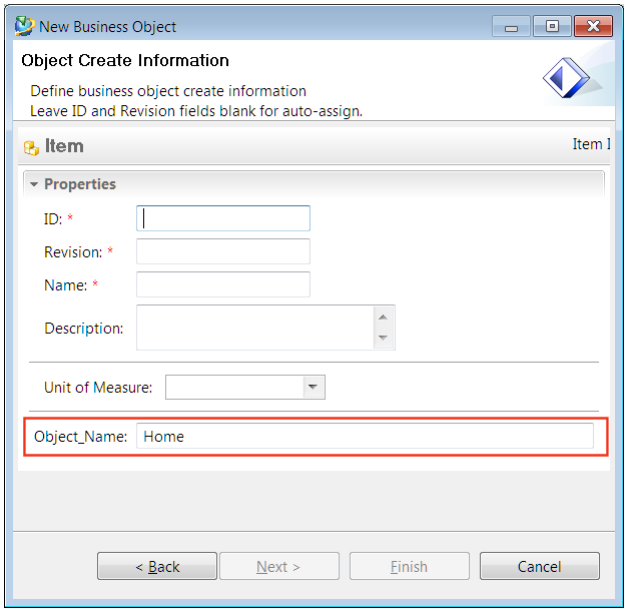
```
            <property name="revision:IMAN_master_form_rev:user_data_1"  />
            <property name="revision:IMAN_master_form_rev:user_data_2"  />
            <property name="revision:IMAN_master_form_rev:user_data_3"  />
        </section>
    </page>

</rendering>
```

You must create your own custom panel to pass to the **customPanel** tag. Following is the **MyPanel.java** file that defines the custom panel:

```
package com.teamcenter.rac.ui.commands.newbo.mypanel;

import com.teamcenter.rac.ui.commands.create.bo.NewBOWizard;
import com.teamcenter.rac.util.AbstractCustomPanel;
import com.teamcenter.rac.util.IPageComplete;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.forms.widgets.FormToolkit;
public class MyPanel extends AbstractCustomPanel implements IPageComplete
{
  private Composite composite;
  private Text text;

  public MyPanel()
  {
  }

  public MyPanel( Composite parent )
  {
      super( parent );
  }

  @Override
  public void createPanel()
  {
      FormToolkit toolkit = new FormToolkit( parent.getDisplay() );
      composite = toolkit.createComposite( parent );

      GridLayout gl = new GridLayout( 2, false );
      composite.setLayout( gl );

      GridData gd = new GridData( GridData.FILL_HORIZONTAL );
      gd.grabExcessHorizontalSpace = true;

      composite.setLayoutData( gd );

      GridData labelGD = new GridData( GridData.HORIZONTAL_ALIGN_END );
      Label label = toolkit.createLabel( composite, "Object_Name: " );
      label.setLayoutData( labelGD );

      GridData typeTextGd = new GridData( GridData.FILL_HORIZONTAL );
      text = toolkit.createText( composite, "" );
      text.setText( "This is my own panel" );
      text.setLayoutData( typeTextGd );
  }
```

```
    public boolean isPageComplete()
    {
        String txt = text.getText();
        return txt.length() == 0 ? false : true;
    }

    @Override
    public Composite getComposite()
    {
        return composite;
    }

    @Override
    public void updatePanel()
    {
        if( input != null )
        {
            NewBOWizard wizard = (NewBOWizard) input;
            String msg = "";
            if( wizard.model.getTargetArray()!= null )
            {
                try
                {
                    msg = wizard.model.getTargetArray()[0].getProperty(
                            "object_name" ).toString();
                }
                catch( Exception e )
                {
                    e.printStackTrace();
                }
            }
            else
            {
                msg = "Nothing is selected";
            }
            text.setText( msg );
        }
    }

    @Override
    public Object getUserInput()
    {
        return null;
    }
}
```

Following is the resulting custom panel added to the New Business Object wizard, which is run when you choose **File→New→Other** in the rich client.



customPanel example

header

Specifies that a header area must be displayed in the page.

ATTRIBUTES

None.

SUPPORTED STYLE SHEETS

This tag can be used on the **Summary** style sheets.

SUPPORTED ELEMENTS

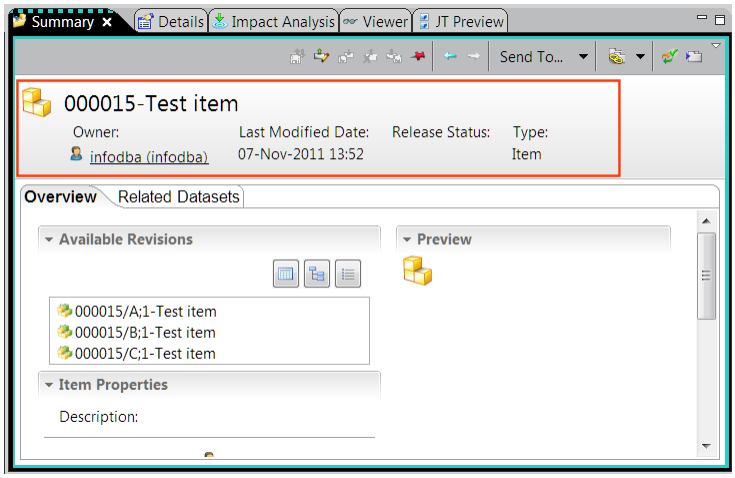
The following XML elements can be used within the header tag:

- `<classificationTrace>`
- `<image>`
- `<property>`

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **header** element:

```
<header>
   <image source="thumbnail"/>
   <classificationTrace/>
   <property name="owning_user"/>
   <property name="last_mod_date"/>
   <property name="release_status_list"/>
   <property name="object_type"/>
</header>
```



header element in the rich client

ADDITIONAL INFORMATION

The **<header>** element is optional. If it is not included, or if it does not contain any elements, the header is automatically be populated with the **object_string** as a label. This label is not selectable.

image

Specifies that an image is to be rendered. Image dimensions are always kept proportional when being scaled or resized.

ATTRIBUTES

maxheight

Specifies the maximum height in pixels to which the image should be scaled. This is a string attribute that is optional.

maxwidth

Specifies the maximum width in pixels to which the image should be scaled. This is a string attribute that is optional.

source

Specifies the source of the image to display. The attribute value can be a **thumbnail**, **preview**, or **type** keyword. This is a string attribute that is optional.

tooltip

Specifies the tooltip associated with the image. The attribute value must be a key into a property file. This is a string attribute that is optional.

> Note:
>
> For backward compatibility, if no attributes are specified and the current object type is an **Item**, **ItemRevision**, or **Dataset** business object, an attempt is made to find and render any preview image that is associated with the object.

SUPPORTED STYLE SHEETS

This tag can be used on the following types of style sheets:

Property
Summary
Form

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **image** element:

```
<header>
    <image source="thumbnail"/>
    <classificationTrace/>
    <property name="owning_user"/>
    <property name="last_mod_date"/>
    <property name="release_status_list"/>
```

```
    <property name="object_type"/>
</header>
```

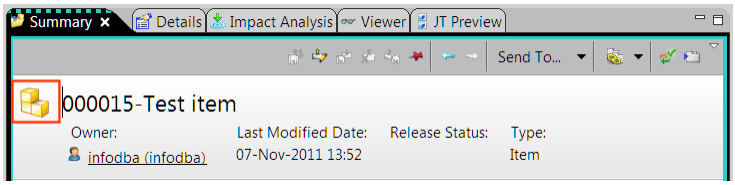


image element in the rich client

label

Specifies a label to be rendered.

ATTRIBUTES

style
Controls font style for the label text, including font size, weight, name, and style (such as italic). The format follows the CSS guideline, for example:

```
style="font-size:14pt;font-style:plain;
font-family:Tahoma;font-weight:bold"
```

textKey
Support is provided for localized values by using the TextServer. For example, the tagging **<label text="Hello World" />** displays text on the property page, and **<label textKey="k_version_name" />** displays the localized text in the provided property.

SUPPORTED STYLE SHEETS

This tag can be used on all style sheet types.

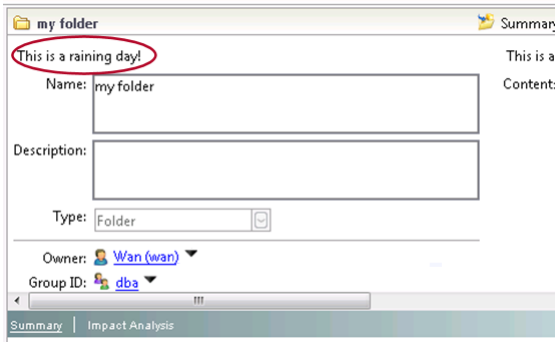
EXAMPLES

- Text

  Following is sample code for the **label text** element:

  ```
  <label text="This is a raining day!" />
  ```

  The following figure shows the text on the page.



Label tag example

- **style** attribute

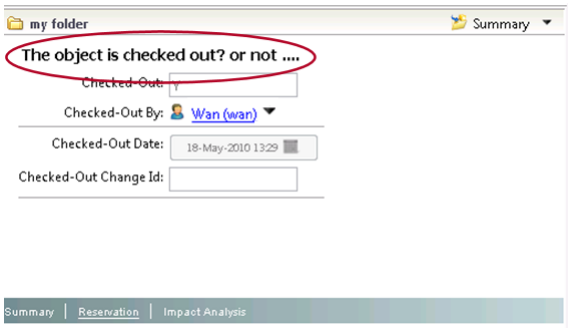
  The property style sheet page tagging includes the **style** attribute within the **label text** and **property** tags to control font style. Support is provided for font size, weight, name, and style

(such as italic). The format follows the CSS guideline, for example, **style="font-size:14pt;font-style:plain;font-family:Tahoma;font-weight:bold"**.

For example:

```
<page title="Reservation" titleKey="tc_xrt_Reservation"
visibleWhen="object_desc==Testing*">
<label text="The object is checked out? or not ...."
style="font-size:14pt; font-style:plain;font-family:Tahoma; font-weight:bold" />
...
</page>
```

This results in the font style shown in the following figure.



style tagging example

- URL rendering

  URL addresses included in the **label** and **property** tag are automatically rendered.

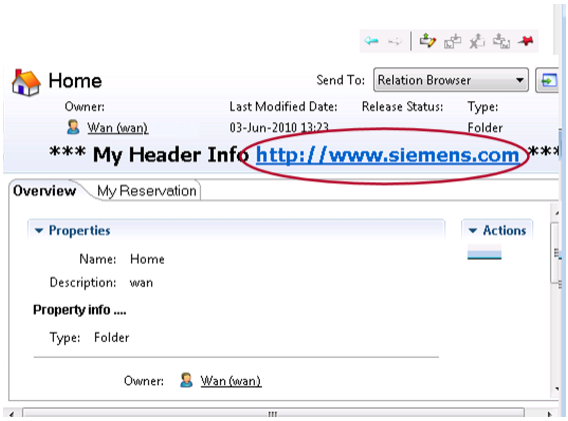
  For example:

  ```
  <label text="Press www.abcnews.com to view the latest headlines!"
  style=" font-size:14pt;font-style:plain;font-family:Tahoma;font-weight:normal" />
  ```

  Consider the following code:

  ```
  <label text="***My Header Info http://www.siemens.com ***" />
  ```

  Because a URL is included in the **label** tag, it is automatically rendered on the page, as shown in the following figure.



URL rendering example

listDisplay

Displays a set of objects in a list format.

ATTRIBUTES

None.

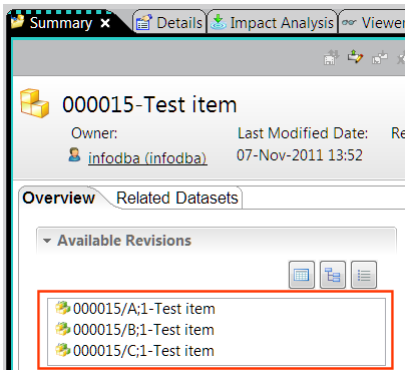
SUPPORTED STYLE SHEETS

This tag can be used on **Summary** style sheets.

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **listDisplay** element:

```
<section titleKey="tc_xrt_AvailableRevisions">
       <objectSet source="revision_list.ItemRevision" defaultdisplay="listDisplay"
sortdirection="descending" sortby="item_revision_id">
        <tableDisplay>
            <property name="object_string"/>
            <property name="item_revision_id"/>
            <property name="release_status_list"/>
            <property name="last_mod_date"/>
            <property name="last_mod_user"/>
            <property name="checked_out_user"/>
          </tableDisplay>
          <thumbnailDisplay/>
          <treeDisplay>
             <property name="object_string"/>
             <property name="item_revision_id"/>
             <property name="release_status_list"/>
             <property name="last_mod_date"/>
             <property name="last_mod_user"/>
             <property name="checked_out_user"/>
           </treeDisplay>
              <listDisplay/>
         </objectSet>
</section>
```

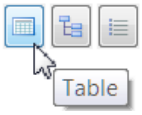


listDisplay element in the rich client

objectSet

Provides a set of display options for the selected object. This element is a replacement for the **attachments** element.

Click the appropriate button to view the object's characteristics in a table, list, or tree.



objectset buttons

The tree option is rendered as a thumbnail display.

ATTRIBUTES

defaultdisplay

Specifies the default format to use when displaying the set of objects. Valid values are **treeDisplay**, **tableDisplay**, **listDisplay**, or **thumbnailDisplay**. The default value is **listDisplay**. This is a string attribute that is optional.

sortby

Specifies the object property to sort the set of objects by prior to rendering. The default value is **object_string**. This is a string attribute that is optional.

sortdirection

Specifies the direction in which the set of objects should be sorted. Valid values are **ascending** or **descending**. The default value is **ascending**. This is a string attribute that is optional.

source

Specifies the comma-delimited set of run-time properties or relations that return the desired set of objects. The format for the attribute value is *property.type*, where *property* is the name of a relation, run-time, or reference property, and *type* represents the type of objects to be included. Multiple *property.type* values can be specified as a comma-separated list, such as **contents.Item** or **contents.ItemRevision**. When using a relation property, you must explicitly specify the relation. Subtypes are not traversed. This is a string attribute that is required.

SUPPORTED STYLE SHEETS

This tag can be used on the **Summary** style sheets.

EXAMPLE

Following is sample code using the **objectSet** tag:

```
  <objectSet source="contents.WorkspaceObject" defaultdisplay="thumbnailDisplay"
sortby="object_name" sortdirection="ascending">
     <tableDisplay>
         <property name="object_string"/>
           ...
     </tableDisplay>
        <thumbnailDisplay/>
        <treeDisplay>
          <property name="object_string"/>
            ...
        </treeDisplay>
          <listDisplay/>
            <command commandId="com.teamcenter.rac.common.AddNew"
renderingHint="commandbutton"/>
            <command commandId="org.eclipse.ui.edit.cut" renderingHint="commandbutton">
               <parameter name="localSelection" value="true"/>
            </command>
            <command acommandId="com.teamcenter.rac.copy" renderingHint="commandbutton"/>
            <command commandId="com.teamcenter.rac.viewer.pastewithContext"
renderingHint="commandbutton"/>
  </objectSet>
```

In an object set, you can use the **command** tag to add buttons for existing menu commands. For example, you can place cut, copy, and paste buttons on the object set.

Property tags used in an **objectSet** are able to display relation properties of the **source** objects. Use the *relationname.propertyname* format to access them. For example, to display the name of the relation type, use the following:

```
<property name = "TC_Attaches.type_string" />
```

You can use the **localSelection** parameter to specify the selected object. If the **localSelection** parameter is set to **true** for any command action, the action is performed on the object selected in the object set. For example, In case of the cut action, if the **localSelection** parameter is **true**, the cut action operates on the object selected in the object set list. If the **localSelection** parameter is **false** (or not set), the cut action operates on the object for which the summary is being shown and not on an object selected in the object set. Although the **localSelection** parameter works with all commands defined in the object set views, it only makes sense to use it with certain commands. For example, any commands such as **Revise**, **Cut**, and **Delete** that should operate on objects in the object set view should have the **localSelection** parameter set to **true**, and any commands such as **Add New** and **Paste** that do *not* operate on the object selected should not use this parameter.

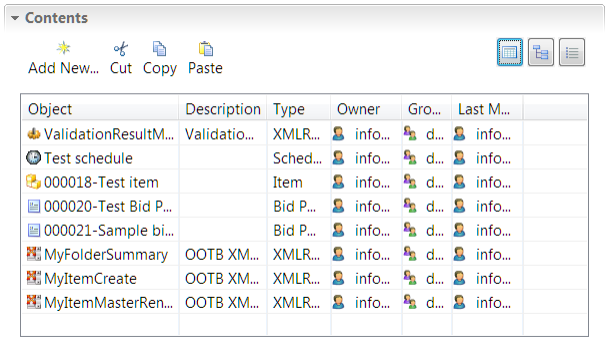
In Systems Engineering, to connect objects with any trace link relation (**FND_TraceLink** and its subtypes), you can use the runtime properties of the WorkSpace object:

- **fnd0defining_objects** (for defining objects)
- **fnd0complying_objects** (for complying objects)

These properties can be displayed using a style sheet with the same *property type* format. However, when you use the **Add New** command to add a new object, these properties are not set or modified.

See the Systems Engineering guide for information about configuring trace link features and configuring defining and complying object calculation.

Following is how the sample code is rendered in the user interface. This object set shows the contents of a folder.



Object set (with table button selected)

page

Presents a tab panel in a dialog box or view. If the **page** element is not defined in the XML file, a default page is created.

ATTRIBUTES

title

Specifies the default string of the title for this tab. This attribute is used when the string in the **titleKey** attribute is not found in the locale file. This is an optional attribute.

titleKey

Specifies the key used to search for the title in the locale file. If it is not defined, the string defined by the **title** attribute is used. This is an optional attribute.

visibleWhen

Defines the conditional display of a tab based on one of three types of expressions comparing a property or preference to a value. The value can be **null** or a string, including a string containing the ***** wildcard character. Multiple values can be checked with an array property or preference. When checking an array value, use a comma as a delimiter for the values. Any properties checked must be loaded on the page using the **<property>** element. The three types of expressions check the following:

- The value of a property on the selected object
- The value of a Teamcenter preference
- The value of a property on an object related to the selected object

- To check the value of a property on the selected object, use the real (database) name of the property in the expression.

  If you want to show a "**Test**" page if the **object_desc** property begins with the word **Testing**, use the following:

  ```
  <page title="Test" titleKey="tc_xrt_testpage"
                     visibleWhen="object_desc==Testing*">
  ```

  The following examples show "**My Page**" based on the value of a property called **myProp**.

  Display the page if **myProp** contains "test".

  ```
  <page titleKey="My Page" visibleWhen="myProp==*test*">
  ```

  Display the page if **myProp** does *not* contain "test".

  ```
  <page titleKey="My Page" visibleWhen="myProp!=*test*">
  ```

  Display the page if **myProp** contains no value (null).

```
<page titleKey="My Page" visibleWhen="myProp==null">
```

- To check the value of a Teamcenter preference, use **{pref:preference-name}** to differentiate it from a property-based expression. Following are some examples:

  Display the page when the **TC_Enable_MyPref** preference has no value.

  ```
  <page titleKey="My Page"
        visibleWhen="{pref:TC_Enable_MyPref}==null">
  ```

  Display the page when the **Item_ColumnPreferences** preference contains **object_string** and **object_type**, as the first two values.

  ```
  <page titleKey="My Page"
        visibleWhen="{pref:Item_ColumnPreferences}==object_string,object_type,*">
  ```

- To check a property on an object *related* to the selected object, you must include the reference or relation property name and the property name from the related object separated by a period.

  Display the page when the owner of the selected object is **user1**.

  ```
  <page title="Custom User Page"
        visibleWhen="owning_user.user_id==user1">
  ```

  Display the page when the status of the selected object is **TCM Released**.

  ```
  <page title="Custom Status Page"
        visibleWhen="release_status_list.name==TCM Released">
  ```

  Display the page when two specific statuses are present on the object — both **TCM Released** and **Approved**.

  ```
  <page title="Special Status Page"
        visibleWhen="release_status_list.name==TCM Released,Approved">
  ```

  Display the page when there is a PDF dataset attached with a specification relation.

  ```
  <page title="The PDF Page"
        visibleWhen="IMAN_specification.object_type==*PDF*">
  ```

Note:

If there is only one page, and the **visibleWhen** condition hides this page, the rich client ignores this condition and makes the page visible.

If you specify a reference property but you do not specify a property on the related object, the default value will be the secondary object's localized value - typically object_string.

SUPPORTED STYLE SHEETS

This tag can be used on the following types of style sheets, although the way pages appear may vary from one type to another.

Property
Summary
Form
Create (visibleWhen not supported)
SaveAs (visibleWhen not supported)

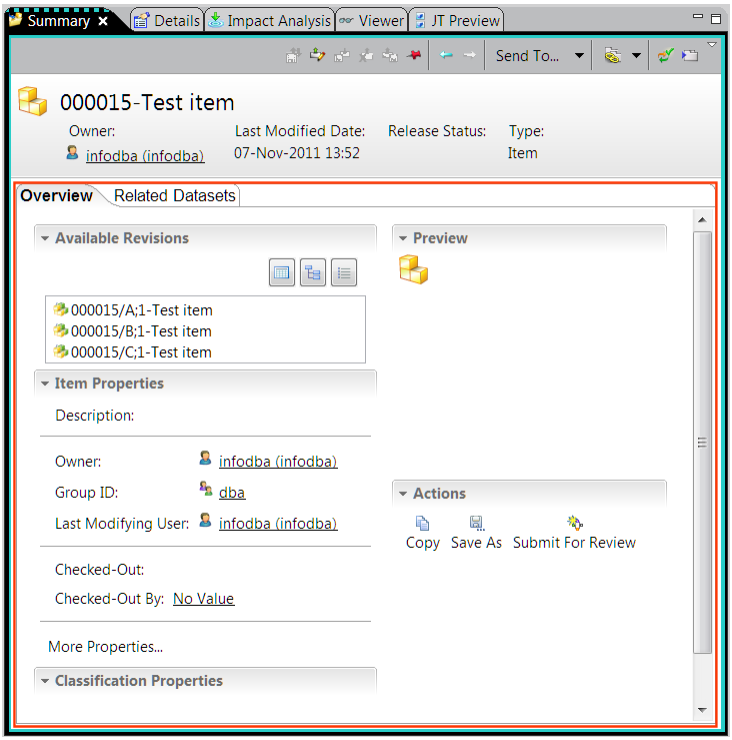
EXAMPLES

- **page** element

  Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **page** element:

```
<page titleKey="tc_xrt_Overview">
    <column>
        <section titleKey="tc_xrt_AvailableRevisions">
            <objectSet source="revision_list.ItemRevision"
defaultdisplay="listDisplay"
sortdirection="descending" sortby="item_revision_id">
                <tableDisplay>
                    <property name="object_string"/>
                    <property name="item_revision_id"/>
                    <property name="release_status_list"/>
                    <property name="last_mod_date"/>
                    <property name="last_mod_user"/>
                    <property name="checked_out_user"/>
                </tableDisplay>
                <thumbnailDisplay/>
                <treeDisplay>
                    <property name="object_string"/>
                    <property name="item_revision_id"/>
                    <property name="release_status_list"/>
                    <property name="last_mod_date"/>
                    <property name="last_mod_user"/>
                    <property name="checked_out_user"/>
                </treeDisplay>
                <listDisplay/>
            </objectSet>
        </section>
        <section titleKey="tc_xrt_ItemProperties">
            <property name="object_desc"/>
            <separator/>
            <property name="owning_user" renderingHint="objectlink"
modifiable="false"/>
            <property name="owning_group" renderingHint="objectlink"
modifiable="false"/>
            <property name="last_mod_user"/>
            <separator/>
            <property name="checked_out"/>
            <property name="checked_out_user"/>
```

```
                <separator/>
                <command commandId="com.teamcenter.rac.properties"
titleKey="tc_xrt_moreProperties"/>
            </section>
            <section titleKey="tc_xrt_ClassificationProperties">
                <classificationProperties/>
            </section>
        </column>
        <column>
            <section titleKey="tc_xrt_Preview">
                <image source="preview"/>
            </section>
            <section titleKey="tc_xrt_actions" commandLayout="vertical">
                <command actionKey="copyAction" commandId="com.teamcenter.rac.copy" />
                <command actionKey="saveAsAction"
commandId="org.eclipse.ui.file.saveAs" />
                <command actionKey="newProcessAction"
commandId="com.teamcenter.rac.newProcess" titleKey="tc_xrt_newProc" />
            </section>
        </column>
    </page>
```



page element in the rich client

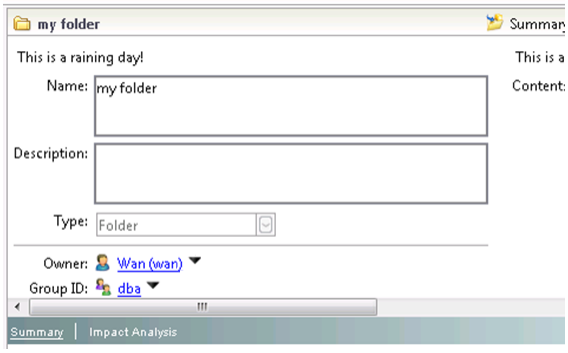
- **visibleWhen** attribute

  To specify a single conditional evaluation for the component property, include the **visibleWhen** parameter on the property style sheet page, for example, **visibleWhen="object_desc!=abc"**.

  Consider the following code:

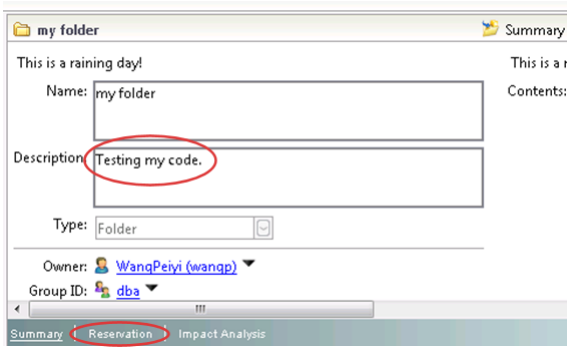
```
<page title="Reservation" titleKey="tc_xrt_Reservation"
visibleWhen="object_desc==Testing*">
...
</Page>
```

If the word **Testing** is used in the **object_desc** property, the **Reservation** link appears. In the following example, the **object_desc** property is blank, and therefore the **Reservation** link does not appear.



Condition is not met for the visibleWhen parameter

Now change the description value to **Testing my code**; the **Reservation** link appears after you save your changes, as shown in the following figure.



Condition is valid for the visibleWhen parameter

parameter

Passes in the name/value parameters to the parent command. This is a child element of the **command** element. An example of a parameter is **localSelection**.

ATTRIBUTES

name

Specifies the parameter name, for example, **searchName**. This is a required attribute.

value

Specifies the parameter value, for example, **CustomSearch**. This is a required attribute.

SUPPORTED STYLE SHEETS

This tag can be used on **Summary** style sheets.

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **paramter** element:

```
<command actionKey="pasteAction" commandId="com.teamcenter.rac.viewer.pastewithContext"
renderingHint="commandbutton"/>
<command actionKey="cutAction" commandId="org.eclipse.ui.edit.cut"
renderingHint="commandbutton">
        <parameter name="localSelection" value="true"/>
</command>
```

CUSTOM COMMANDS WITH PARAMETERS

When you define a custom command for the rich client that uses parameters, you must add the parameters with the **commandParameter** tag in the **plugin.xml** file of your custom plug-in, for example:

```
<extension
         id="com.myco.command.parameter.test.commands.category"
         name="Test Category"
         point="org.eclipse.ui.commands">
      <command
            name="Test command"
            categoryId="com.teamcenter.ddp.commands.category"
            id="com.myco.command.parameter.test.commands.testCommand">
             <commandParameter id="localStage1" name="localStage1" optional="true" />
             <commandParameter id="localStage2" name="localStage2" optional="true" />
             <commandParameter id="localStage3" name="localStage3" optional="true" />
             <commandParameter id="localStage4" name="localStage4" optional="true" />
             <commandParameter id="source" name="source" optional="true" />
      </command>
   </extension>
   <extension
```

```
         point="org.eclipse.ui.handlers">
      <handler
            commandId="com.myco.command.parameter.test.commands.testCommand"
            class="com.myco.command.parameter.test.handlerTest">
      </handler>
   </extension>
```

When you add the custom command to the style sheet, the parameters defined in the style sheet must match the **commandParameter** values defined in your plug-in, for example:

```
<command commandId="com.myco.command.parameter.test.commands.testCommand"
renderingHint="commandbutton">
        <parameter name="localStage1" value="1"/>
        <parameter name="localStage2" value="2"/>
        <parameter name="localStage3" value="3"/>
        <parameter name="localStage4" value="4"/>
</command>
```

If you add the parameters in the style sheet, but not in the **plugin.xml** file, then when you execute the command in the rich client interface you will receive an error that no parameters are found.

property

Specifies the real (database) name, not the **display name**, of the object property you want to display. You must include at least one property in the XML definition, otherwise, the system displays an empty panel.

> Note:
>
> You cannot add the same property multiple times in the same style sheet.

ATTRIBUTES

border

Determines whether the border is displayed. Valid values are **true** and **false**. This works only with the titled style.

column

Applies only to the **textfield** and **textarea** rendering hints. It sets the number of columns.

name

Specifies the database name of a property on the object. This is a required attribute.

When using this attribute on a **create** style sheet, which is used in the New Business Object wizard, there is additional functionality. You can specify a property from another object that is related to the original object by the **revision**, **IMAN_master_form**, or **IMAN_master_form_rev** relations. To do this, specify the relation trail followed by the name of the property on the destination related object, separated by a colon.

For example, if a **create** style sheet is registered for an item,

- to display the revision ID, you would use

  ```
  name=revision:item_revision_id
  ```

- to display the **project_id** property from the item's master form.

  ```
  name=IMAN_master_form:project_id
  ```

- to display the **serial_number** property from the item revision master form, you need to traverse from the item to the revision, and then to the revision's master form.

  ```
  name=revision:IMAN_master_form_rev:serial_number
  ```

renderingHint

Specifies the component used to render this property. This is an optional attribute. If not defined, the default renderer is used based on the property type.

renderingStyle

Defines the rendering style used in the rendering component. There are three styles: headed, headless, and titled.

- **Headed**

  This is the default rendering style. The property name is displayed on the left followed by the property value renderer.

- **Headless**

  This style renders only the property value without displaying the property name in front of it.

- **Titled**

  The property name is displayed on the top of the property value renderer.

> Tip:
>
> For consistency and to avoid possible display issues, use the same rendering style type for all properties within a section.

row

Applies only to **textarea** elements. It sets the number of the rows for the element.

style

This provides pass-through controls for the **font-style**, **font-size**, **font-weight**, and **font-name** settings. The values are not validated by the client and are simply passed through to the CSS processor. The **font-size** setting must only use **pt**, not **px**, or **%**, or any other unit.

> Example:
>
> ```
> style="font-size:14pt;font-style:plain;
> font-family:Tahoma;font-weight:bold"
> ```

visibleWhen

Defines the conditional display of a property based on one of two types of expressions comparing a property or preference to a value. The value can be **null** or a string, including a string containing wildcard characters. Multiple values can be checked with an array property or preference. When checking an array value, use a comma as a delimiter for the values. The two types of expressions check the following:

1. The value of a property on the selected object

```
<property name="p1" visibleWhen="<Property name>==<Some value>"/>
```

```
<property name="p1" visibleWhen="<Property name>!=<Some value>"/>

<property name="p1" visibleWhen="<Property name>==null"/>

<property name="p1" visibleWhen="<Property name>!=null"/>
```

2. The value of a Teamcenter preference

```
<property name="p1" visibleWhen="{pref:<Preference name>}==<Some value>"/>

<property name="p1" visibleWhen="{pref:<preference name>}!=<Some value>"/>
```

- To check the value of a property on the selected object, use the real (database) name of the property in the expression.

  If you want to show a "**myprop**" property only if the **object_desc** property begins with the word **Testing**, use the following:

  ```
  <property name="myprop" visibleWhen="object_desc==Testing*"/>
  ```

- To check the value of a Teamcenter preference, use **{pref:preference-name}** to differentiate it from a property-based expression. Following are some examples:

  Display a property when the **Cust_Enable_MyProp** preference is set to **true**.

  ```
  <property name="myprop" visibleWhen="{pref:Cust_Enable_MyProp}==true">
  ```

> Note:
>
> Using **visibleWhen** for properties is only for use in SWT-based style sheet views in the rich client. For example, **Summary** (when checked out), **Create**, and **SaveAs**.
>
> It will only work with SWT-based property beans, and will not work with Swing-based property beans.
>
> In the current implementation of SWT style sheet rendering in the **Summary** view, a plain **Label** instead of a **LabelPropertyBean** is used to display property values in read-only mode (**RenderFlat**). Since the **visibleWhen** framework implementation is based on **AbstractPropertyBeans** and its children, the **visibleWhen** feature will not be available in the **Summary** view in read-only mode. It will only be available when the object is checked-out.

SUPPORTED STYLE SHEETS

This tag can be used on the following types of style sheets:

Property
Summary
Form
Create

Caution:

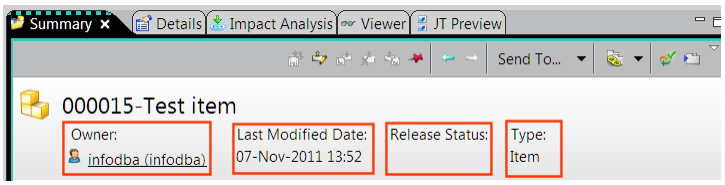
Only simple properties are supported on the create style sheet. More complex properties, like interdependent list of values (LoV), are not supported.

EXAMPLES

- **property** element

  Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **property** element:

```
 <header>
     <image source="thumbnail"/>
     <classificationTrace/>
     <property name="owning_user"/>
<property name="last_mod_date"/>
<property name="release_status_list"/>
<property name="object_type"/>
 </header>
```



property elements in the rich client

- URL rendering

  URL addresses in the **label text** and **property** tags are automatically rendered. (In the rich client, URL addresses are also automatically rendered for **textfield** and **textarea** rendering hints.)

  For an example, see *label*.

rendering

Root element

ATTRIBUTES

Version

Specifies the version of the XML schema. When an older version is detected, the program automatically converts the old scheme to the new one.

SUPPORTED STYLE SHEETS

This tag is required on all types of style sheets:

Property
Summary
Form
Create

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **rendering** element:

```
<rendering xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="XMLRenderingStylesheet_Schema.xsd">
    <header>
        <image source="thumbnail"/>
        <classificationTrace/>
        <property name="owning_user"/>
        <property name="last_mod_date"/>
        <property name="release_status_list"/>
        <property name="object_type"/>
    </header>
.
.
.
</rendering>
```

section

Defines a group of elements to be displayed together within a named section.

> Note:
>
> This element is a replacement for the **view** element.

ATTRIBUTES

commandLayout

Controls the layout of commands. Valid values are **horizontal** or **vertical**.

initialstate

Specifies whether the view or section should be expanded or collapsed on initial rendering. Valid values are **expanded** or **collapsed**. The default value is **expanded**. This attribute is optional.

SUPPORTED STYLE SHEETS

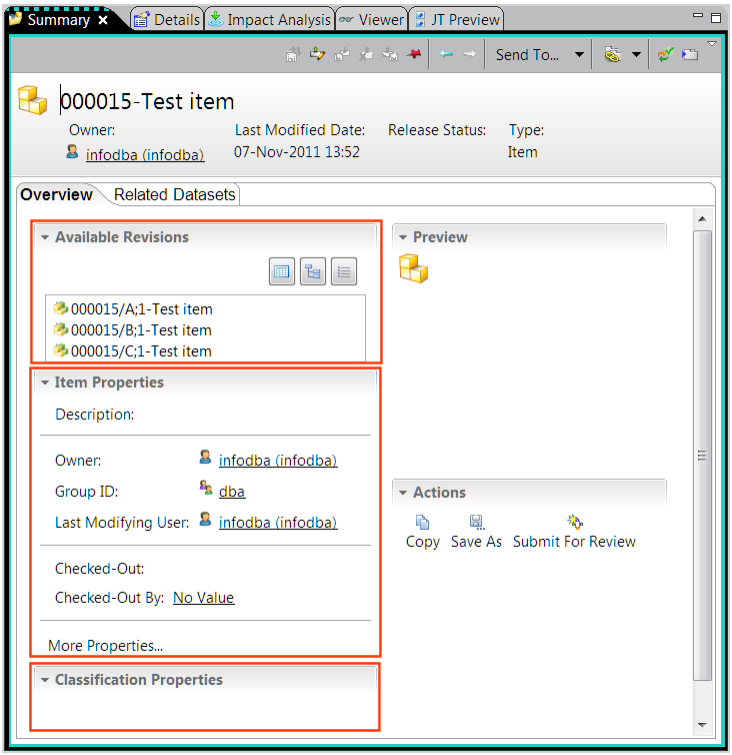
This tag can be used on the **Summary** style sheets.

EXAMPLE

- Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **section** element:

```
    <page titleKey="tc_xrt_Overview">
        <column>
            <section titleKey="tc_xrt_AvailableRevisions">
                <objectSet source="revision_list.ItemRevision"
defaultdisplay="listDisplay" sortdirection="descending"
sortby="item_revision_id">
                    <tableDisplay>
                        <property name="object_string"/>
                        <property name="item_revision_id"/>
                        <property name="release_status_list"/>
                        <property name="last_mod_date"/>
                        <property name="last_mod_user"/>
                        <property name="checked_out_user"/>
                    </tableDisplay>
                    <thumbnailDisplay/>
                    <treeDisplay>
                        <property name="object_string"/>
                        <property name="item_revision_id"/>
                        <property name="release_status_list"/>
                        <property name="last_mod_date"/>
                        <property name="last_mod_user"/>
                        <property name="checked_out_user"/>
                    </treeDisplay>
                    <listDisplay/>
                </objectSet>
            </section>
```

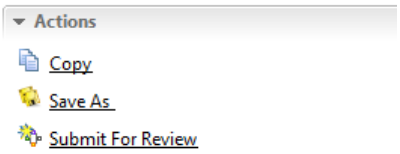
```
            <section titleKey="tc_xrt_ItemProperties">
                <property name="object_desc"/>
                <separator/>
                <property name="owning_user" renderingHint="objectlink"
modifiable="false"/>
                <property name="owning_group" renderingHint="objectlink"
modifiable="false"/>
                <property name="last_mod_user"/>
                <separator/>
                <property name="checked_out"/>
                <property name="checked_out_user"/>
                <separator/>
                <command commandId="com.teamcenter.rac.properties"
titleKey="tc_xrt_moreProperties"/>
            </section>
            <section titleKey="tc_xrt_ClassificationProperties">
                <classificationProperties/>
            </section>
        </column>
        <column>
            <section titleKey="tc_xrt_Preview">
                <image source="preview"/>
            </section>
            <section titleKey="tc_xrt_actions" commandLayout="vertical">
                <command commandId="com.teamcenter.rac.copy" />
                <command commandId="org.eclipse.ui.file.saveAs" />
                <command commandId="com.teamcenter.rac.newProcess"
titleKey="tc_xrt_newProc" />
            </section>
        </column>
    </page>
```



section elements in the rich client

- The following example demonstrates the use of the **commandLayout** attribute. The following code defines the command layout as vertical:

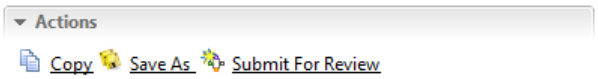
```
<section titleKey="tc_xrt_actions" commandLayout="vertical">
     <command commandId="com.teamcenter.rac.copy" />
     <command commandId="org.eclipse.ui.file.saveAs" />
     <command commandId="com.teamcenter.rac.newProcess" text="newProc" />
</section>
```



Vertical command layout

The following code defines the command layout as vertical:

```
<section titleKey="tc_xrt_actions" commandLayout="horizontal">
     <command commandId="com.teamcenter.rac.copy" />
     <command commandId="org.eclipse.ui.file.saveAs" />
     <command commandId="com.teamcenter.rac.newProcess" text="newProc" />
</section>
```



Horizontal command layout

separator

Adds a separator in the pane.

ATTRIBUTES

None.

SUPPORTED STYLE SHEETS

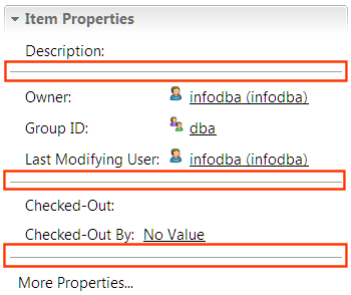
This tag can be used on the following types of style sheets and the New Business Object wizard:

Property
Summary
Form
Create

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **separator** element:

```
<section titleKey="tc_xrt_ItemProperties">
     <property name="object_desc"/>
     <separator/>
     <property name="owning_user" renderingHint="objectlink" modifiable="false"/>
     <property name="owning_group" renderingHint="objectlink" modifiable="false"/>
     <property name="last_mod_user"/>
     <separator/>
     <property name="checked_out"/>
     <property name="checked_out_user"/>
     <separator/>
         <command commandId="com.teamcenter.rac.properties"
titleKey="tc_xrt_moreProperties"/>
</section>
```



separator elements in the rich client

tableDisplay

Displays a set of objects in a table format.

ATTRIBUTES

minRowCount

Specifies the minimum number of rows to display.

The default is **1**.

maxRowCount

Specifies the maximum number of rows to display. Set to **-1** for an unlimited number of rows.

The default is **10**.

maxColumnCharCount

Specifies the maximum number of characters for a column. Set to **-1** for unlimited column size.

The default is **80**.

SUPPORTED STYLE SHEETS

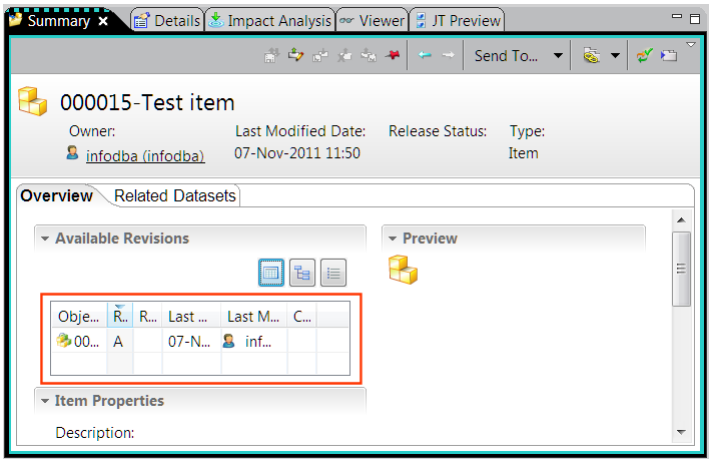
This tag can be used on the **Summary** style sheets.

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **tableDisplay** element:

```
<section titleKey="tc_xrt_AvailableRevisions">
     <objectSet source="revision_list.ItemRevision" defaultdisplay="listDisplay"
sortdirection="descending" sortby="item_revision_id">
    <tableDisplay>
      <property name="object_string"/>
      <property name="item_revision_id"/>
      <property name="release_status_list"/>
      <property name="last_mod_date"/>
      <property name="last_mod_user"/>
      <property name="checked_out_user"/>
    </tableDisplay>
            <thumbnailDisplay/>
        <treeDisplay>
             <property name="object_string"/>
             <property name="item_revision_id"/>
             <property name="release_status_list"/>
             <property name="last_mod_date"/>
             <property name="last_mod_user"/>
             <property name="checked_out_user"/>
        </treeDisplay>
        <listDisplay/>
```

```
    </objectSet>
</section>
```



tableDisplay element in the rich client

thumbnailDisplay

Displays a set of objects using thumbnails arranged in a grid.

ATTRIBUTES

None.

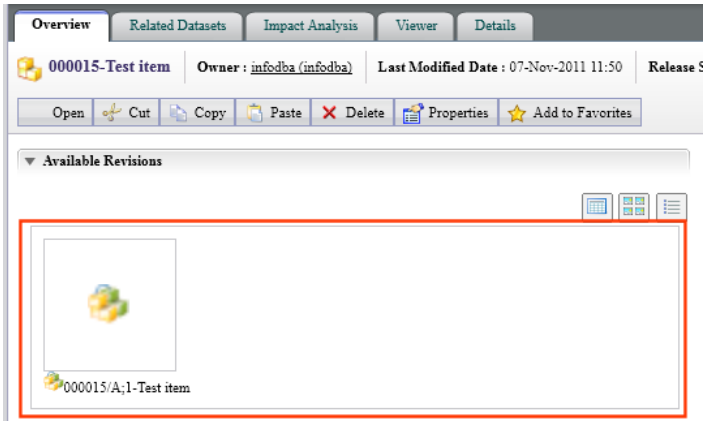
SUPPORTED STYLE SHEETS

This tag can be used on **Summary** style sheets.

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **thumbnailDisplay** element:

```
<section titleKey="tc_xrt_AvailableRevisions">
    <objectSet source="revision_list.ItemRevision" defaultdisplay="listDisplay"
sortdirection="descending" sortby="item_revision_id">
         <tableDisplay>
             <property name="object_string"/>
             <property name="item_revision_id"/>
             <property name="release_status_list"/>
             <property name="last_mod_date"/>
             <property name="last_mod_user"/>
             <property name="checked_out_user"/>
         </tableDisplay>
              <thumbnailDisplay/>
         <treeDisplay>
             <property name="object_string"/>
             <property name="item_revision_id"/>
             <property name="release_status_list"/>
             <property name="last_mod_date"/>
             <property name="last_mod_user"/>
             <property name="checked_out_user"/>
          </treeDisplay>
          <listDisplay/>
      </objectSet>
</section>
```



thumbnailDisplay

treeDisplay

Displays a set of objects in a tree format.

ATTRIBUTES

minRowCount

Specifies the minimum number of rows to display.

The default is **1**.

maxRowCount

Specifies the maximum number of rows to display. Set to **-1** for an unlimited number of rows.

The default is **10**.

maxColumnCharCount

Specifies the maximum number of characters for a column. Set to **-1** for unlimited column size.

The default is **80**.

SUPPORTED STYLE SHEETS

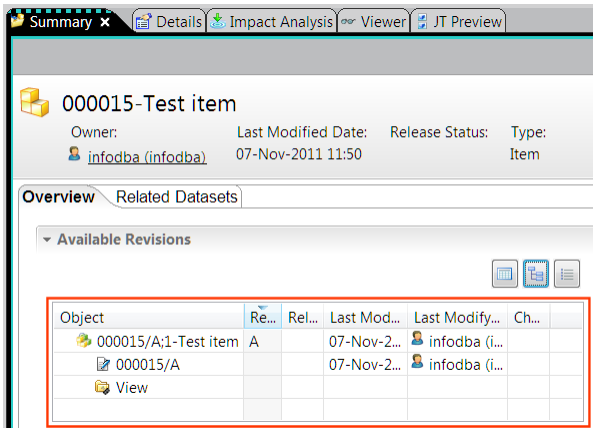
This tag can be used on **Summary** style sheets.

EXAMPLE

Following is sample code from the **ItemSummary.xml** XML rendering style sheet showing the **treeDisplay** element:

```
<section titleKey="tc_xrt_AvailableRevisions">
    <objectSet source="revision_list.ItemRevision" defaultdisplay="listDisplay"
sortdirection="descending" sortby="item_revision_id">
         <tableDisplay>
             <property name="object_string"/>
             <property name="item_revision_id"/>
             <property name="release_status_list"/>
             <property name="last_mod_date"/>
             <property name="last_mod_user"/>
             <property name="checked_out_user"/>
         </tableDisplay>
              <thumbnailDisplay/>
    <treeDisplay>
      <property name="object_string"/>
      <property name="item_revision_id"/>
      <property name="release_status_list"/>
      <property name="last_mod_date"/>
      <property name="last_mod_user"/>
      <property name="checked_out_user"/>
     </treeDisplay>
           <listDisplay/>
```

```
        </objectSet>
</section>
```



treeDisplay element

Rendering hints

Standard rendering hints

The **renderingHint** is an attribute on a **<property>** tag that allows you to specify which user interface widget to use to present the corresponding property. It is an optional attribute, and if not defined, the default rendering is used based on the property type.

The JavaBeans used depend on the specified rendering hint and rendering style. The following table lists the JavaBeans used based on the hint and style definition. All rendering hints are supported in the rich client. If the rendering style is not defined, the default style is **headed**. There is no titled style for SWT.

For standard rendering hints, see the **com.teamcenter.rac.viewer/plugin.xml** file for the SWT version of rendering hints, and see the **com.teamcenter.rac.common/plugin.xml** file for the Swing version of rendering hints. Siemens Digital Industries Software recommends you use SWT rather than Swing, because Swing is being phased out in favor of SWT.

Rendering hint	Use on property types	JavaBeans
array	Used for array only: character, date, double, float, integer, logical, short, string, note, typed reference, typed relation.	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyArray** Titled (Swing only)

Rendering hint	Use on property types	JavaBeans
		• **TitledPropertyArray**
checkbox	Logical	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyCheckbox** Titled (Swing only) • **TitledPropertyCheckbox**
checkboxoptionlov	Used with LOV only: character, date, double, float, integer, short, string, note, typed ref, typed relation.	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyCheckboxOptionLov** Titled (Swing only) • **TitledPropertyCheckboxOptionLov**
datebutton	Date	Headed or headless • SWT: **DateControlPropertyBean** • Swing: **PropertyDateButton** Titled (Swing only) • **TitledPropertyDateButton**
label	All types. Used for display only; you cannot change the value.	Headed or headless • SWT: **LabelPropertyBean**

Rendering hint	Use on property types	JavaBeans
		• Swing: **PropertyLabel** Titled (Swing only) • **TitledPropertyLabel**
localizablearray	String	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyLocalizableArray** Titled (Swing only) • Not applicable.
localizabletextarea	String	Headed or headless • SWT: **LocalizedTextAreaPropertyBean** • Swing: **PropertyLocalizableTextArea** Titled (Swing only) • Not applicable.
localizabletextfield	String	Headed or headless • SWT: **LocalizedTextfieldPropertyBean** • Swing: **PropertyLocalizableTextField** Titled (Swing only) • Not applicable.
localizablelongtextpanel	String, note	Headed or headless

Rendering hint	Use on property types	JavaBeans
		• SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyLocalizableLongTextPanel** Titled (Swing only) • Not applicable.
logical	Logical	Headed or headless • SWT: **LogicalPropertyBean** • Swing: **PropertyLogicalPanel** Titled (Swing only) • **TitledPropertyLogicalPanel**
longtext	Use only for string type when the maximum length is bigger than 2500.	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyLongText** Titled (Swing only) • **TitledPropertyLongText**
lovuicomp	Used with LOV only: character, date, double, float, integer, logical, short, string, note, typed ref, typed relation.	Headed or headless • SWT: **LOVUIComponentPropertyBean** • Swing: **PropertyLOVUIComponent** Titled (Swing only)

Rendering hint	Use on property types	JavaBeans
		• **TitledPropertyLOVUIComponent**
objectlink	Typed ref, typed relation	Headed or headless • SWT: **ObjectLinkPropertyBean** • Swing: **PropertyObjectLink** Titled (Swing only) • **TitledPropertyObjectLink**
panel	Unused	Headed or headless • SWT: Not applicable. • Swing: **PropertyPanel** Titled (Swing only) • **TitledPropertyPanel**
radiobutton	Character, date, double, float, integer, logic, short, string, note.	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyRadioButton** Titled (Swing only) • **TitledPropertyRadioButton**
radiobuttonoptionlov	Used with LOV only: character, date, double, float, integer, short, string, note, typed ref, typed relation.	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.)

Rendering hint	Use on property types	JavaBeans
		• Swing: **PropertyRadioButtonOptionLov** Titled (Swing only) • **TitledPropertyRadioButtonOptionLov**
slider	The Upper and Lower bounds as defined in the BMIDE control the slider minimum and maximum values. **SWT** Integer, short The SWT bean does not support other data types. **Swing** Character, double, float, integer, short, note, string. For types except integer and short, this attribute converts the property value to an integer. The value defaults to 0 if any errors occur.	Headed or headless • SWT: **SliderPropertyBean** • Swing: **PropertySlider** Titled (Swing only) • **TitledPropetySlider**
styledtextarea	Character, date, double, float, integer, logic, short, string, note.	Headed or headless • SWT: **StyledTextAreaPropertyBean**

Rendering hint	Use on property types	JavaBeans
		• Swing: **PropertyStyledTextArea** Titled (Swing only) • **TitledPropertyStyledTextArea**
styledtextfield	Character, date, double, float, integer, logic, short, string, note.	Headed or headless • SWT: **StyledTextfieldPropertyBean** • Swing: **PropertyStyledTextField** Titled (Swing only) • **TitledPropertyStyledTextField**
textarea	Character, date, double, float, integer, logic, short, string, note.	Headed or headless • SWT: **TextAreaPropertyBean** • Swing: **PropertyTextArea** Titled (Swing only) • **TitledPropertyTextArea**
textfield	Character, date, double, float, integer, logic, short, string, note.	Headed or headless • SWT: **TextfieldPropertyBean** • Swing: **PropertyTextField** Titled (Swing only) • **TitledPropertyTextField**
togglebutton	Character, date, double, float,	Headed or headless

Rendering hint	Use on property types	JavaBeans
	integer, logic, short, string, note.	• SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyToggleButton** Titled (Swing only) • **TitledPropertyToggleButton**
togglebuttonoptionlov	Used with LOV only: character, date, double, float, integer, short, string, note, typed ref, typed relation.	Headed or headless • SWT: Not applicable. (Currently using **LegacyPropertyBridgeBean**.) • Swing: **PropertyToggleButtonOptionLov** Titled (Swing only) • **TitledPropertyToggleButtonOptionLov**

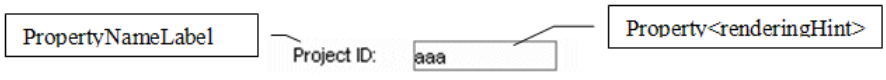
Rendering style

Each type of renderer supports three styles: headed, headless, and titled.

- Headed

  Displays the property name on the left followed by the property value renderer. This is the default rendering style.

  This style has two components, the **PropertyNameLabel** JavaBean for the property name and the **Property***renderingHint* JavaBean for the renderer (for example, **PropertyTextField** or **PropertyTextArea**).



Headed rendering style

- Headless

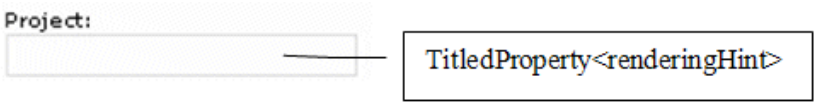
Renders only the property value without displaying the property name.

This style contains only one JavaBean, **Property***renderingHint*.

- Titled

  Displays the property name above the property value renderer.

  This style uses only the **TitledProperty***renderingHint* JavaBean, for example **TitledPropertyTextField** or **TitledPropertyTextArea**.



Titled rendering style

Default renderers

The following table displays the default renderer for each type. If the rendering hint is not provided, the default renderer is used.

Type	Default renderer
string	**textfield** if size < 60 **textarea** if 60 < size < 2500 **longtext** if size >= 2500
char	**textfield**
double	**textfield**
float	**textfield**
int	**textfield**
short	**textfield**
long	**textfield**
date	**datebutton**
logical	**logical**
note	**textfield** if size < 60 **textarea** if size > 60
TypedReference/ UntypedReference	**objectlink**
All array types	**array**

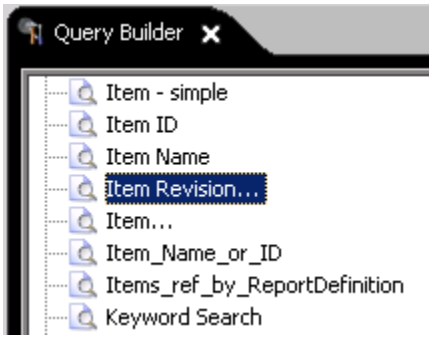
Type	Default renderer
	Array properties *must* use the array renderer. All other renderers are for single-value properties.
Any property with an LOV attached	**lovuicomp** In turn, the **lovuicomp** renderer uses the **lovcombox** renderer if the number of list of values does not exceed 500, and if there are more, the **lovpopupbutton** renderer is used.

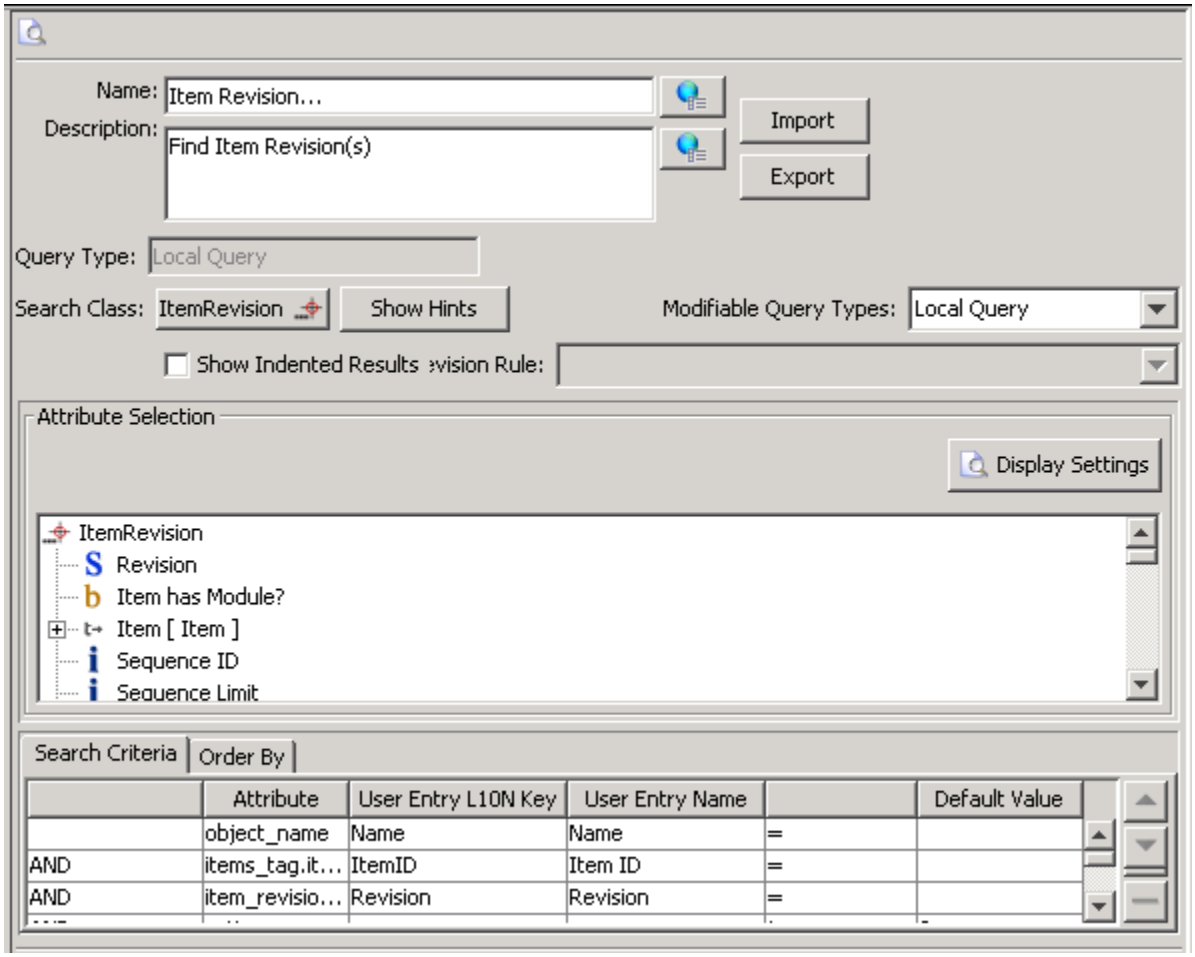
3. Rich client customization using preferences and properties files

Add a quick search item

You can add an existing query to the quick search box at the top of the navigation pane. You can add any query from Query Builder, including queries you create yourself.

In the following example, you add the **Item Revision...** query to the quick search list, searching for the **object_name** attribute. All necessary information is found in the Query Builder.





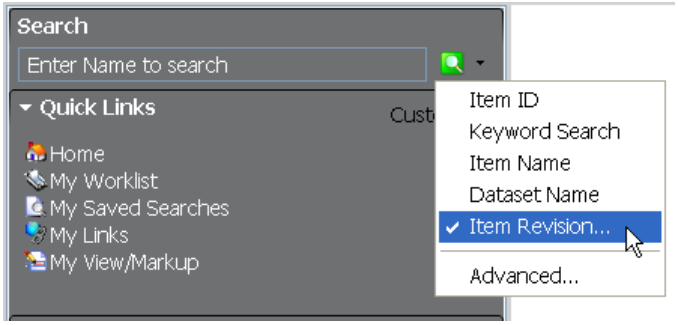
1. Edit the **Quick_Access_Queries** preference and add the name of the query to the list (for example, **Item Revision...**).

   ```
   Item Revision...
   ```

2. Edit the **Quick_Access_Queries_Attribute** preference and add the **User Entry L10N Key** for the search attribute to use with the query (for example, **Name**) for **object_name**:

   ```
   Item Revision..._SearchAttribute=Name
   ```

3. Test the new quick search by clicking the arrow to the right of the quick search.



Additional information

- The attribute to use for the search must exist in the query. Look at the query in Query Builder to see the attributes available.
- Any attributes used for quick search *must* have a localization (**L10N**) key registered in the TextServer. All OOTB attributes already have **L10N** keys. If you are searching with a custom attribute, you must add it to the TextServer definition.

Configuring the worklist using .properties files

You can modify properties files related to your worklist.

Worklist process and interface features that can be configured include the following:

- Application labels
- Task state labels
- System messages
- Buttons

Typically, the administrator checks first in the workflow **common_user.properties** file for the appropriate property.

- If the desired property is listed in the **common_user.properties** file, it is common to all workflow applications, and you can implement your change across all workflow applications simultaneously.
- If the desired property is not found in the **common_user.properties** file, it may be listed in the **inbox_user.properties** file. Changes made to the properties in this file are unique to the inbox.

Configuring forms

Master forms

Master forms are created and deleted when an item or item revision is created or deleted.

- Master forms display specific product information to the rest of the enterprise in a standardized format.
- When a new item is created, an **Item Master** form object is created automatically. Similarly, when a new item revision object is created, an **ItemRevision Master** form object is created automatically.

You can enter data in the item master and item revision master forms when you create an item or by opening an **Item Master** or **ItemRevision Master** form object.

Note:

- Master forms inherit access privileges from the parent item or item revision, so if you change access privileges to an item or item revision you affect the privileges on the master form.

  You can use the **TC_MASTERFORM_DELEGATE** environment variable to change this default behavior.
- An item can have only one **Item Master** form.
- An item revision can have only one **ItemRevision Master** form.

Configure edit and view for forms

For form objects, the **Form_double_click** preference value can be set to either **View** or **Edit** to cause the double-click action on a form to open that form in either edit or view mode.

Note:

To ensure consistent behavior on forms in both edit and view mode, set the **Configuration_shown_on_reservation_dialogs** value to **true**, and set the following preferences as required:

- **Confirm_cancel_checkout_suppressed**

  Set this preference to **true** to proceed without user input on the cancel checkout confirmation dialog box.
- **Confirm_checkin_suppressed**

Set this preference to **true** to proceed without user input on the checkin confirmation dialog box.

- **Confirm_checkout_suppressed**

  Set this preference to **true** to proceed without user input on the checkout confirmation dialog box.

Displaying files in the viewer

On Windows platforms, you can display Microsoft Word, PowerPoint, Excel, Portable Document Format (PDF), and text files in the **Viewer** panel. On Linux platforms, the properties of the file are displayed instead.

Microsoft Word, Excel, PowerPoint, PDF, and text files are displayed automatically when they are named references of corresponding datasets. No additional configuration required.

To display QAF files in the viewer if Teamcenter lifecycle visualization is not already installed:

1. In the **com.teamcenter.rac.common** plug-in, create an empty **com\teamcenter\rac\common\tcviewer\tcviewer_user.properties** file.

2. In the **com.teamcenter.rac.common** plug-in, open the **tcviewer\tcviewer.properties** file, copy the entire **DatasetViewer.VIEWSEARCHORDER** line, and paste it into the new **tcviewer_user.properties** file.

3. In the **tcviewer_user.properties** file, add **UG-QuickAccess-Binary** to the end of the **DatasetViewer.VIEWSEARCHORDER** line.

4. Add the following line to the **tcviewer_user.properties** file:

   ```
   UGMASTER.VIEWPANEL=com.teamcenter.rac.common.tcviewer.DatasetViewer
   ```

5. In the **com.teamcenter.rac.util** plug-in, create a **com\teamcenter\rac\util\viewer\viewer_user.properties** file, and add the following line:

   ```
   Imager.EXTENSION=gif,jpg,qaf
   ```

Note:

To disable the toolbar in the viewer, go to **com/teamcenter/rac/common/tcviewer**, create the **tcviewer_user.properties** file in a text editor, and add the following entry:

```
useNevaIEViewerToolBar=false
```

Working with themes

You can change the start-up theme for the rich client and also modify existing themes, by using a text editor.

How do I know what themes are available?

Using a text editor, examine the content of **TC_ROOT**/*portal/plugins/configuration/fragment.xml*. You will find each theme defined and its unique identifier (id)

Example:

In the following example, two themes are defined — **Classic** and **Light**:

```
<fragment>
   <extension
          point="org.eclipse.e4.ui.css.swt.theme">
       <theme
             basestylesheeturi="css/rac_classic.css"
             id="com.teamcenter.rac.classic"
             label="Classic">
       </theme>
       <theme
             basestylesheeturi="css/rac_lighttheme.css"
             id="com.teamcenter.rac.light"
             label="Light">
       </theme>
    </extension>
</fragment>
```

Your defined themes may differ.

How can I temporarily change which theme is used?

To launch the rich client with a specified theme once, launch the **TC_ROOT***portal**portal.bat* script specifying the **-theme=** argument followed by the **id** of the theme.

Example:

The following example will launch the rich client using the light theme this time only.

```
portal.bat -theme=com.teamcenter.rac.light
```

If you launch the rich client without the argument, it will use the default theme.

How can I permanently change which theme is used?

To launch the rich client with a specified theme every time, modify the **TC_ROOT***portal**portal.bat* script to include the **-theme=** argument followed by the **id** of the theme.

Example:

Before:

```
start Teamcenter.exe %* -vm "JRE_HOME...
```

After:

```
start Teamcenter.exe %* -theme=com.teamcenter.rac.light -vm "JRE_HOME...
```

How do I change theme colors?

The rich client applies a theme in two parts. Depending on which widget you want to modify, you need to modify the corresponding file.

- The colors for SWT widgets are defined by ***.CSS** files in the following directory:

  ```
  TC_ROOT\portal\plugins\configuration\css
  ```

  After making your changes, delete the rich client cache and restart the rich client.

  If your changes do not appear, clear the cache by deleting the **Teamcenter** subdirectory in the user's home directory on the client. This directory is automatically created again when the user starts the rich client.

  On a Windows client, it is typically the **%HOMEDRIVE%%HOMEPATH%\Teamcenter** directory.

  On a Linux client, it is typically the **$HOME/Teamcenter** directory.

  When you delete this directory, the last state of the rich client is lost, and the user interface appears as it does at initial startup.

- The colors for Swing widgets are defined by Java registry-style ***.properties** files in the following directory:

  ```
  TC_ROOT\portal\plugins\configuration\com\teamcenter\rac\themeName
  ```

  Remember, after modifying any registry files, regenerate the registry and restart the rich client.

  *TC_ROOT***portal\registry\genregxml.bat**

Registry

What is the registry?

The registry plays a vital role in the Teamcenter rich client. The registry uses the properties files to define classes, icons, internationalized text, inheritance, search order, and appearance. Our registry is fundamentally a subclass of the **ResourceBundle** object found in Java and uses the properties files to store replaceable information that can be tailored based on need. Many basic customization tasks can be accomplished using the registry.

> Note:
>
> This registry is not the Windows registry.

The key contained within the registry is contained within the product code, but the value can be modified. Every key/value pair that appears within the registry falls into one of the following categories: appearance, structural, instantiation, or localization.

- Appearance

  Keys that define colors, fonts, and sizes.

- Structural

  Keys that are defined (usually at the top) and are called import statements. Importing allows the nesting of property files for searching. This allows you to define a key in one place within the rich client that can be reused without being duplicated. For instance, the **OK** button text must be localized in the rich client user interface based on the current locale. Therefore, the text for **OK** is placed within the **aif_locale.properties** file.

- Instantiation

  Keys used to construct Java objects. Teamcenter rich client makes use of reflection and dynamic instantiation within the Java language to construct objects by their string name rather than hard-coding them within the source code.

  For example, each command that appears in the menu bars and toolbars is dynamically constructed and run. You can subclass a command and replace it with your own by replacing an entry within a properties file to tell the rich client to use your command rather than the base command.

- Localization

  Keys allow localization of text and messages to be flexibly obtained based on the current locale of the OS where rich client is running. Teamcenter rich client is delivered with locales that support English, German, French, Spanish, Italian, Czech, Russian, Korean, Japanese, and Chinese. Each of

these separate property files is based on the convention that JavaSoft has established. Localization keys are those for which the **ResourceBundle** class was first designed.

In any given package (directory) that uses registries, there are three property files: base, locale, and user. These properties files contain structural, instantiation, and appearance keys.

- Base properties file

  Contains non locale-specific information and is the only property required in a package.

- Locale properties file

  Contains only localization keys. The reason for having the localization separate from the base is that when keys are to be localized within the rich client, it is easy to separate the localization keys from the rest. In addition, if you are developing customization to be deployed in more than one locale, this provides a convenient separation between the two and allows for easier localization that could be outsourced to third parties for completion.

User properties files

These files contains properties that can be modified by customers. When you make modifications to the rich client, always make them in the user properties file. This protects the base system from user mistakes. It also enables a backup approach in the event that the change breaks the rich client, in which case you can reload or obtain a copy from the reference directory.

User property files are always consulted first for registry lookups. The rich client is completely loaded on demand. Therefore, when it starts up, the applications listed on the left side may or may not exist. This is not known until the user clicks on the application for the first time. In addition, commands are never loaded until requested by the user. This loading method provides a small memory footprint and enables the system to only use resources that the user has requested.

Supported types

Primitive types are supported in the registry as well. The following types are supported: color, font, integer, float, double, char, boolean, string, string array, and images. Some of the types use special formats, as follows:

- Color

  red, *green*, *blue*

  ```
  255,255,255
  ```

- Font

  font-name, *font-style*, *font-size*

```
“Times Roman, Font.ITALIC, 12”
```

- String array

  one,*two*,*three*

Registry keys

If a key must contain a blank or space, the space follows the backslash (\) character. If the name of the type contains a space, such as that found in the **Adobe Acrobat** type, the entry would be:

```
Adobe\ Acrobat=images/acrobat.gif
```

Java treats a space like an equal sign (=). Therefore, if the entry is:

```
Adobe Acrobat=images/acrobat.gif
```

Java interprets the key as **Adobe** and the value as **Acrobat=images/acrobat.gif**.

Edit rich client registry file

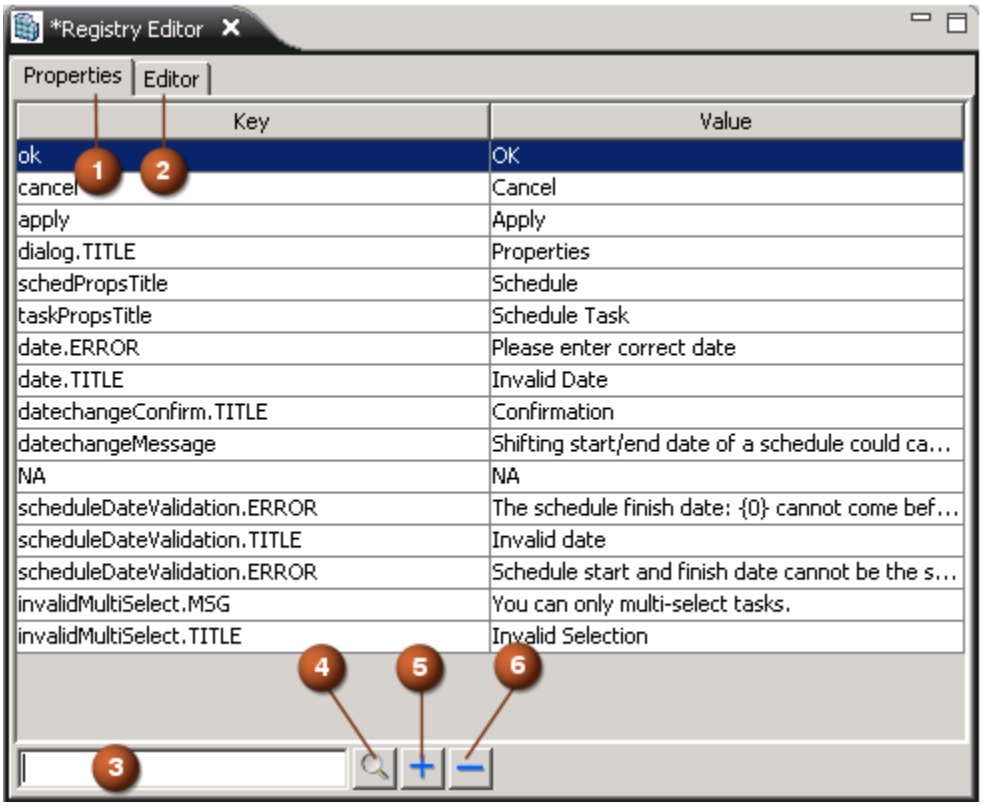
What is Registry Editor?

Registry Editor is a rich client application for editing registry files of applications registered in the Teamcenter framework. Such registry files may need modification for purposes of internationalization, dynamic class invocation, or configuration. While administrators can use any standard text editor to edit a registry file, Registry Editor simplifies the tasks of editing the registry files.

When a property in a registry file must be modified for a site, a system administrator locates the file in the rich client and opens the file using Registry Editor. The system administrator navigates to the desired key name and changes the value. If a new property is required, the administrator inserts a blank line and enters the new key name and key value.

Registry Editor does not need to be enabled before use, but the feature must be selected during Teamcenter installation.

Registry Editor interface



1	**Properties** tab	The **Properties** pane displays a two-column table populated with key names and key values contained in the currently open **.properties** file. You can edit key names and values in this pane.
2	**Editor** tab	The **Editor** pane displays the full text within the currently open **.properties** file, including comments. You can directly edit the text. You may want to use the **Editor** pane so that you can add your own comments about any changes you make.
3	**Find** box	Field for entering a **Key** name search string.
4	**Find**	Starts the search operation. If the registry file contains a Key whose name exactly matches the string entered in the **Find** box, then clicking **Find** (or pressing **Enter** while focus is on the **Find** box) highlights the row containing the key. If the row is not already visible, the dialog box scrolls to display the row.

5	**Create** +	Adds a row to the properties table for a new key/value pair.
6	**Remove** —	Deletes the selected row.

All Registry Editor menus are standard Teamcenter rich client menus.

Registry files in the rich client

The rich client uses *registry* files named with the extension **.properties**. A registry file contains user-defined configuration settings (keys/values) that influence how the application appears and performs in the interface. Property files are located in various **.jar** files.

- A *key* is a field in a record that contains unique data and identifies the record in the file or database. Each key value must be unique in each record.
- A *value* is the content of a field or variable. It can refer to alphabetic, numeric, or alphanumeric data.
- *Properties* are the keys and values in a registry file that specify the configuration settings for an application.

The following figures contain examples of registry files, as viewed in the Registry Editor **Editor** pane. These files are contained in the **com.teamcenter.rac.aifrcp_1.0.0.jar** file.

```
import=com.teamcenter.rac.aif.aif
com.teamcenter.rac.aif.registryeditor.RegistryEditorApplication.
   PANEL=com.teamcenter.rac.aif.registryeditor.RegistryEditorApplicationPanel
com.teamcenter.rac.aif.registryeditor.RegistryEditorApplication.
   MENUBAR=com.teamcenter.rac.aif.registryeditor.RegistryEditorApplicationMenuBar
com.teamcenter.rac.aif.registryeditor.RegistryEditorApplication.
   TOOLBAR=com.teamcenter.rac.aif.registryeditor.RegistryEditorApplicationToolBar
### helpPage address ###
helpPage=/registry_editor/WebHelp/WHStart.htm
# New Key
# ------------------------------
newKey.ICON=images/add_16.png
# Remove a Key
# ------------------------------
removeKey.ICON=images/remove_16.png
```

registryeditor.properties file

```
# Find in Display Keys
# ------------------------------
findInDisplay.TIP=Find In Display
# Read-Only Text
# ------------------------------
readOnly=(Read-Only)
# New Key
# ------------------------------
newKey.TIP=Create a New Key
# Remove a Key
# ------------------------------
removeKey.TIP=Remove a Key
# File Type description key.
# ------------------------------
RegistryFiles=AIF Registry Files
# Unable to Save keys.
# ------------------------------
UnableToSave.MESSAGE=Unable to Save...File is Read Only!
UnableToSave.TITLE=Unable to Save
# No File Open
# ------------------------------
NoFileOpen.MESSAGE=Can't Save...No file open!
NoFileOpen.TITLE=Can't Save
# Overwrite File
# ------------------------------
Overwrite.TITLE=File Already Exists
Overwrite.MESSAGE=Overwrite the existing file?
# About Dialog keys
# ------------------------
application.TITLE=Registry Editor
application.DESCRIPTION=Registry Editor is used to edit
   Portal Registry files. This application is used\only
   for editing Registry files that are used for internationalization,
   dynamic\nclass invocation, and configuration in the Portal.
#  Title for the Save-As dialog
# ------------------------------
saveAs.TITLE=Save-As
open.TITLE=Open
untitled.FILE=Untitled
saveModified.TITLE=Save Modified File
saveModified.MESSAGE=Should the changes be saved before closing?
saveModified.TIP=Save modified file.
openFile.TIP=Open registry file.
```

registryeditor_locale.properties file

Open a rich client registry file

1. In the rich client, open the **Registry Editor** application.
2. Choose **File→Open**.
3. Browse to and select a registry file, and then click **Open**.

Modify a rich client registry file

1. Open the registry file.
2. In the **Properties** pane, modify the file:

To perform this action	Do this
Change an existing value	Double-click the value to change and update the value.
Add a new key and value	Click **Create Key** + and enter a new key/value pair.
Remove a key and value	Click the key/value pair in the row that you want to remove and click **Remove Key** —.

   Alternatively, you can click the **Editor** tab and directly edit the registry file text in the **Editor** pane. You may want to use the **Editor** pane so that you can add your own comments about any changes you make.

3. Choose either **File→Save** or **File→Save-As**.

   If you choose **File→Save-As**, Teamcenter appends the **.properties** extension to the file name.

Customizing tabs

Customizing the data tabs display

If you have administrator privileges, you can customize how data tabs appear in Multi-Structure Manager and other applications. When a user runs one of these applications and selects the data panel, several tabs appear in the panel. You can change the tabs that appear by editing the application properties file.

There are two types of tabs in each application:

- Tabs that always appear for a particular parent panel regardless of whether anything is selected.

- Tabs that appear only when particular components are selected in the parent panel.

You can customize how the second, selection-specific group of tabs is displayed.

To determine the tabs to display, the system checks four criteria:

- The class type of the selected display component, for example:

```
BOMLine
CfgAttachmentLine
TcItemBOPLine
```

- The subtype of the selected display component, which is generally the same as the class type. However, for BOM lines, it is the occurrence type and for attachment lines it is the relation to the parent.

- The class type of the underlying component, such as **ItemRevision**.

- The subtype of the underlying component (the component type name).

For each selection, the system checks for six properties and adds all the tabs found. You can edit these properties to change the tabs that are presented to the user:

Display-component-classtype.TABS
Display-component-subtype.TABS
Display-component-classtype.underlying component classtype.TABS
Display-component-classtype.underlying component subtype.TABS
Display-component-subtype.underlying component classtype.TABS
Display-component-subtype.underlying component subtype.TABS

For example, in the Multi-Structure Manager application, the default properties are:

```
BOMLine.TABS=Referencers, Variant, Attachments, InClassAtt, CMEViewer, Report,
IncrementalChangeInfo
TcItemBOPLine.TABS=Referencers, Variant, Attachments, InClassAtt, CMEViewer,
Report, IncrementalChangeInfo
AppGroupBOPLine.TABS=Referencers, Attachments, CMEViewer, IncrementalChangeInfo
GDELine.TABS=Referencers, InClassAtt, CMEViewer, Report, IncrementalChangeInfo
GDELinkLine.TABS=Referencers, InClassAtt, CMEViewer, Report,
IncrementalChangeInfo
MEAppearanceLine.TABS=Referencers, Attachments, CMEViewer, IncrementalChangeInfo
CfgAttachmentLine.TABS=Referencers, CMEViewer, IncrementalChangeInfo, Report
BOMLine.ItemRevision.TABS=ProductAppearance
TcItemBOPLine.ItemRevision.TABS=ProductAppearance
CfgAttachmentLine.ItemRevision.TABS=InClassAtt
```

You can add or delete the names of tabs that are displayed for each data panel in this file.

Edit a custom properties file to display tabs

To display tabs, you can place the changes in a **_user.properties** file to override the default **.properties** file.

1. Extract the *application*.**properties** file from the appropriate **com.teamcenter.rac.***component*.**jar** file using the **jar xf** command. For example, if you want to extract the **mpp.properties** file for Part Planner, the command looks like this:

   ```
   jar xf com.teamcenter.rac.cme.legacy.jar com/teamcenter/rac/cme/mpp/mpp.properties
   ```

   > Note:
   >
   > For more information about the **jar** command, see Oracle's Java documentation.

2. Save the extracted file as the *application*_**user.properties** file.

3. In the custom properties file, edit the **.TABS** line to include the tab you want.

4. Insert the custom properties file into your own custom plug-in.

Sample tab customization

The following example is for creating new tabs for Manufacturing Process Planner.

1. In the Business Modeler IDE, create a new type as a child of the **Process** business object.

   When the object is deployed to the rich client and displayed in Manufacturing Process Planner, the tabs in the data pane are different than the tabs for the **Process** business object. You must customize the tabs for the new business object so that they match tabs for the parent.

2. Open the *installation-location***portal****plugins****com.teamcenter.rac.cme.legacy** JAR file and find the **mpp.properties** file in the following directory:

   com\teamcenter\rac\cme\mpp\mpp.properties

3. Make a copy of the **mpp.properties** file and rename it into a customer-specific *application*_**user.properties** file.

4. In the custom properties file, create **.TABS** entries for your custom business object.

   Manufacturing Process Planner accepts the following definitions in the properties files:

   line-type.**TABS=** *tab-1*, *tab-2*, *tab-n*

line-subtype.**TABS=***tab-1*, *tab-2*, *tab-n*
line-type.*underlying-type*.**TABS=** *tab-1*, *tab-2*, *tab-n*
line-type.*underlying-subtype*.**TABS=** *tab-1*, *tab-2*, *tab-n*
line-subtype.*underlying-type*.**TABS=** *tab-1*, *tab-2*, *tab-n*
line-subtype.*underlying-subtype*.**TABS=** *tab-1*, *tab-2*, *tab-n*

line-type is the type of the BOM line, for example, **BOMLine**, **ImanItemBOPLine**, or **Mfg0BvrProcess**. *line-subtype* is the subtype of a line and it can be an occurrence type or a relation type, for example, **MEConsumed** (in some cases, it is equal the line type). *underlying-type* is the type of the underlying component; a **BOPLine** can have the underlying **OperationRevision** type, **MrocessRevision** type, or other types. *underlying-subtype* is the subtype of the underlying component; like the line subtype, the underlying subtype can also be the same as the underlying type.

If a **BOMLine** type matches more than one definition, the result is the sum of tabs from all matched definitions. For example, an item name **I1** is assigned to an operation as **MEConsumed** type. The following tab lines are defined:

```
ImanItemBOPLine.TABS= Variant
ImanItemBOPLine.ItemRevision.TABS= CMEViewer
BOMLine.TABS= Referencers
BOMLine.ItemRevision.TABS= ProductAppearance
```

Selecting the **I1** item in the process structure (below the operation) matches **ImanItemBOPLine.TABS** and **ImanItemBOPLine.ItemRevision.TABS**, and as a result, the system shows the **Variant** and **CMEViewer** tabs. But selecting **I1** in the BOM structure matches **BOMLine.TABS** and **BOMLine.ItemRevision.TABS**; therefore, the system shows the **References** and **ProductAppearance** tabs.

> Note:
>
> The tabs are also defined in the **mpp.properties** file:
>
> *tab*.**CLASS**
> *tab*.**ICON**
>
> The tab label and tooltip are defined in the **mpp_locale.properties** file:
>
> *tab*.**TABLABEL**
> *tab*.**TOOLTIP**

5. Insert the custom properties file into your own custom plug-in.

Add a column to view occurrence notes

By default, you cannot view occurrence notes on intermediate data capture (IDC) objects in the Multi-Structure Manager. However, you can view the occurrence notes if you can add a column to

display them. To accomplish this, you must define the alias for the occurrence note type in the *TC_DATA***structure_alias.xml** file, and then add the column for the occurrence note type in the **IDCStructureColumnsShownPref** preference.

1. Create the occurrence note on the IDC object.

   a. Create a structure in Multi-Structure Manager.

   b. In the data pane to the right, select an occurrence (item revision) in the structure.

   c. Right-click a column heading and add a column for an occurrence note type (for example, **UG NAME**).

   d. For the selected occurrence, double-click in the empty cell in that column and type in your note (for example, type **This is my occurrence note**).

   e. In the left pane, select that occurrence in the structure (the item revision) and choose **Tools→Intermediate Data Capture**. For our example, choose the **Transfer Mode Name** of **tcm_export** because it transfers the **UG NAME** type of occurrence note.

      When the IDC appears on the tab, there is no content in the data panel, and there is no occurrence note column (for example, no **UG NAME** column). That's because the occurrence note type column must be added to this IDC structure view using a customization.

   f. Exit the rich client.

2. Open the *TC_DATA***structure_alias.xml** file and add code.

   a. Add the following under the **Alias name="BOM.Property"** node:

```
<Alias name="UG NAME" default=" " type="string" size="0"/>
```

   b. Add the following under the **AliasAlternate name="BOM.Property"** node:

```
<Alias name="UG NAME">
   <Path depth="0..0">*:*:@instancedRef</Path>
   <Xpath>/UserData/UserValue[@title='UG NAME']/@value</Xpath>
</Alias>
```

3. Regenerate the caches and restart the rich client.

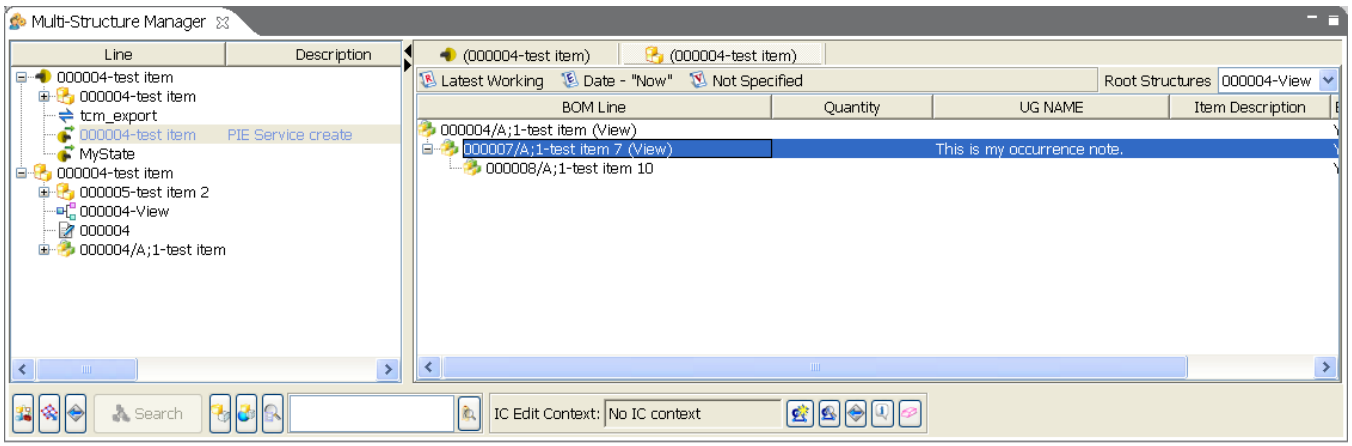
> Note:
>
> `generate_metadata_cache -force`

```
generate_client_meta_cache -log update all
```

4. Add the **UG NAME** value to the **IDCStructureColumnsShownPref** preference, and add a value for the new column's width to the **IDCStructureShownColumnWidthsPref** preference.

> Note:
>
> Since these are interdependent preferences, you cannot use the Options dialog box to make changes.

5. Select the item structure in My Teamcenter and send it to Multi-Structure Manager. The occurrence note displays on the IDC in the right data panel under the new column (for example, the **UG NAME** column).



New column for occurrence notes

Customize the rich client properties files

You can customize the appearance of applications in the rich client by creating user properties files that override the default properties files. The default properties files, with a **.properties** extension, are located in the various **com.teamcenter.rac.***component***.jar** files in the *TC_ROOT***portal\plugins** directory. You override the default file by creating a **_user.properties** file and wrapping it in an Eclipse plug-in. For example, if you want to override a property in Manufacturing Process Planner, you must override the **mpp.properties** file located in the **com.teamcenter.rac.cme.legacy.jar** file with the **mpp_user.properties** file.

1. Create the Java project.

   a. In Eclipse, choose **File→New→Project**.

   b. In the **New Project** dialog box, select **Plug-in Project**. Then click **Next**.

c. In the **Project name** box, type the project name, which should be in the form of **com.mycom.***project-name*. For example, type **com.mycom.propertiesfile**. Click **Next**.

d. Under **Options**, ensure the **Generate an activator** and **This plug-in will make contributions to the UI** check boxes are selected. Click **Next**.

e. Clear the **Create a plug-in using one of these templates** check box. Click **Finish**.

2. Modify the properties file.

a. Navigate to the *TC_ROOT***portal\plugins** directory, and open the JAR file that contains the properties file you want to override. The file name has the form *property***.properties**. For example, to override the **mpp.properties** file, open the **com.teamcenter.rac.cme.legacy.jar** file.

b. Extract the properties file you want to override from the JAR file. For example, extract the **mpp.properties** file.

> Note:
>
> A JAR file uses ZIP compression and can be opened using standard unjar or unzip tools.

c. In a text editor, open the properties file to determine the syntax of the property you want to modify.

d. Create a new *property*_**user.properties** in a text editor. For example, if you open the **mpp.properties** file, create the **mpp_user.properties** file.

e. Type the entries you want to use to override the defaults.

f. Save and exit the file.

3. Update the project tabs.

a. In Eclipse, click your project tab and click its **Dependencies** tab.

b. Under **Required Plug-ins**, click the **Add** button.

c. Select the following plug-ins from the list by holding down the Ctrl key while you click them:

- **com.teamcenter.rac.aifrcp**
- **com.teamcenter.rac.common**
- **com.teamcenter.rac.external**

- **com.teamcenter.rac.kernel**
- **com.teamcenter.rac.tcapps**
- **com.teamcenter.rac.util**

d. Click **OK**.

e. Click your project's **Runtime** tab.

f. Under **Exported Packages**, click the **Add** button.

g. Select your project and click **OK**.

h. Click your project's **Extensions** tab.

i. Click the **Add** button.

j. Select the **com.teamcenter.rac.util.tc_properties** extension point.

k. Click **Finish**.

l. Click the **plugin.xml** tab. Type text in the file so it looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.6"?>
<plugin>
   <extension
          point="com.teamcenter.rac.util.tc_properties">
   </extension>
</plugin>
```

If the text is different, edit it in the tab to resemble the example.

m. Save the project by choosing **File→Save All**.

4. Create a package.

a. In the **Package Explorer** view, right-click your project and choose **New→Package**.

b. In the **Name** box, type the path name where the properties file was originally. For example, if you originally extracted the **mpp.properties** file, the package name should be **com.teamcenter.rac.cme.mpp**.

c. Click **Finish**.

d. From Windows Explorer, drag your modified *property*_**user.properties** file and drop it on the package you created in **Package Explorer**.

e. In **Package Explorer**, select **plugin.xml**.

f. Click the **MANIFEST.MF** tab and type your new package at the end of the **Export-Package** line. For example, if your project name is **com.mycom.propertiesfile** and the new package is called **com.teamcenter.rac.cme.mpp**, the line should read:

```
Export-Package: com.mycom.propertiesfile, com.teamcenter.rac.cme.mpp
```

The **MANIFEST.MF** file should look similar to the following:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Propertiesfile
.
.
.
Bundle-ActivationPolicy: lazy
Export-Package: com.mycom.propertiesfile, com.teamcenter.rac.cme.mpp
```

g. Click your project's **Runtime** tab. If one or more of the items listed in the **Export-Package** line in the previous step are missing, add the missing ones by clicking the **Add** button, selecting the missing packages, and clicking **OK**.

5. Save and export your changes.

6. Debug in Eclipse.

a. In Eclipse, choose **Run→Debug Configurations**.

b. In the **Debug** dialog box, under **Java Application** on the left-hand side, select the configuration you want to debug.

c. At the bottom of the dialog box, click the **Debug** button.

This launches the application in debug mode. To change the perspective to the debug perspective, choose **Window→Open Perspective→Debug**.

You can then debug your application.

7. Run the **genregxml** script.

If you make changes to any of the **.properties** files, or you add new plug-ins or change plug-in content, you must run the **genregxml** script to ensure your changes are included when the rich client starts. This enhances performance because it caches the properties so they can be loaded when the rich client starts. The script takes no arguments and generates a **RegistryLoader** file for

each locale in the **portal\Registry** directory. The **RegistryLoader** file is added during rich client startup.

a. Delete the old registry loader file:

*TC_ROOT***portal\registry\RegistryLoader.xml.gz**

b. Run the following script:

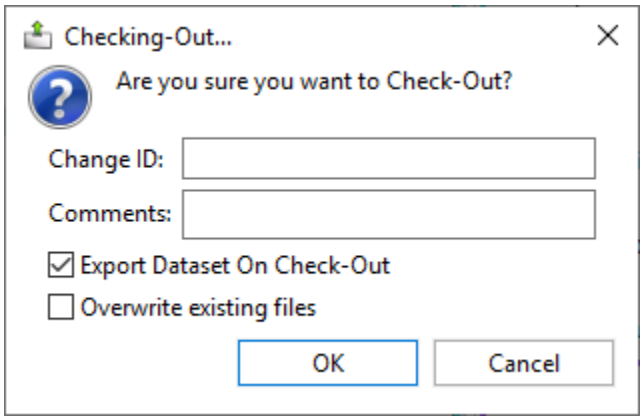
*TC_ROOT***portal\registry\genregxml**

When the script is finished, a new **RegistryLoader.xml.gz** file is created.

8. Verify the customization.

Modify the rich client's export on checkout

You can control the default behavior of the **Export Dataset On Check-Out** option from the rich client.



Export location

The location to which the file is exported is controlled by the **TC_check_out_dir** preference.

However, if the **TC_check_out_dir** preference contains no value, then the rich client will look for the **TcExportDir** setting in the *client_specific.properties* file in **%TC_ROOT%***portal\plugins\configuration*.

Related preferences

- **TC_remove_file_on_check_in**
- **TC_checkout_and_export**
- **Tc_Auto_Checkout**

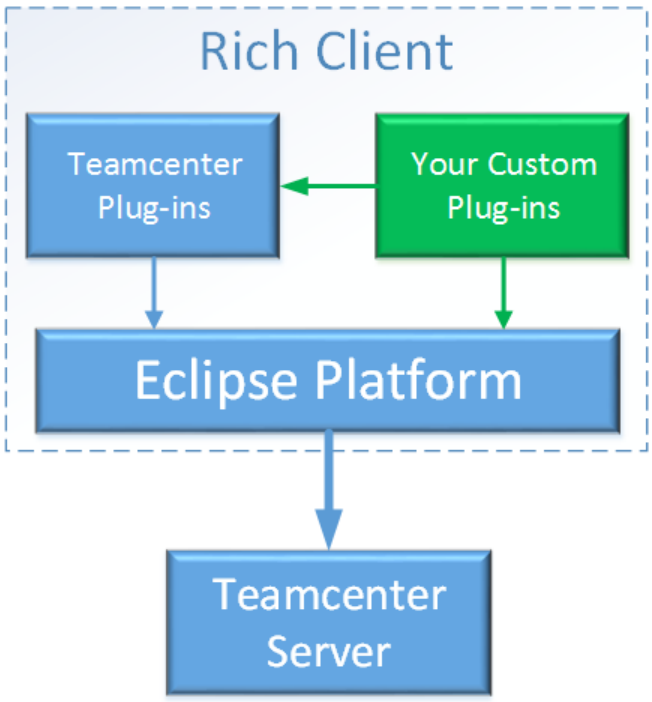
4. Rich client customization using Eclipse plug-ins

Basic concepts about rich client customization

Understanding the Eclipse rich client platform framework

When the rich client is installed, Java archive (JAR) files are installed in the *TC_ROOT***portal\plugins** directory. These files comprise the rich client and the resources required to run it. When you add a new custom plug-in, you deploy it into this directory.

The Teamcenter rich client is hosted within the Eclipse rich client platform (RCP) framework. The RCP is a general purpose application framework which provides strong support for modular and extensible component-based development through the use of plug-ins.



You can find more information about the RCP's features, advantages, and use at the Eclipse Web site:

http://www.eclipse.org/

For more information about using Eclipse, see the *Platform Plug-in Developer Guide*, which can be found at the following location:

https://www.eclipse.org/documentation/

To customize the rich client UI beyond the capabilities of style sheets, you need to use the Eclipse IDE.

What reference material is available?

There are two major sources of reference material available for rich client customization.

- The rich client JavaDoc API reference.

  You can download the rich client JavaDoc from the Teamcenter **Downloads** area of Support Center. It is with the files in the **Major Releases** section.

- The **sample plugin project** provides working examples of the most common types of customization.

What are perspectives and views?

Within the rich client user interface, application functionality is provided in *perspectives* and *views*.

View	The basic display component that displays related information in a UI window.
Perspective	A collection of one or more views and their layout.

Some applications use a perspective with multiple views to arrange how functionality is presented. Other applications use a perspective with a single view.

You can use the **HiddenPerspectives** preference to prevent the display of some Teamcenter perspectives in the rich client.

If your site has online help installed, you can access the application and view help from the rich client **Help** menu.

Process for creating rich client customizations

When you create a rich client customization, follow this general process to create, test, and distribute the customization:

1. Set up your Eclipse environment for rich client customization.
2. Create the desired customization.
3. If you create a custom rich client plug-in, export the plug-in to the *TC_ROOT***portal\plugins** directory or a shared directory.
4. Clear the cache and register any new plug-in to ensure the customization appears in the rich client.
5. Run the rich client to verify the customization.

6. After testing is successful, distribute the customization to your organization.

Introduction to SWT

The Standard Widget Toolkit (SWT), maintained by the Eclipse Foundation, is a graphical toolkit for use with the Java platform. It is an alternative to the Abstract Window Toolkit (AWT) and Swing Java toolkits.

For more information about SWT, see the following URL:

http://www.eclipse.org/swt/

Because the rich client already uses SWT, the necessary files are already loaded into your Eclipse environment from *TC_ROOT*\portal\plugins when you defined the target platform. Therefore, when you create a Java file, you can use the **import org.eclipse.swt.widgets** command to include SWT in your coding.

> Note:
>
> Teamcenter is moving toward SWT/JFace as the user interface toolkit and moving away from AWT and Swing. Siemens Digital Industries Software encourages you to customize Teamcenter using SWT/JFace components. Siemens Digital Industries Software will discontinue Swing/AWT support in a future version.

To become familiar with SWT, you can download SWT samples and run the SWT tutorial in Eclipse.

Create the Eclipse rich client development environment

You must use the Eclipse plug-in development environment in order to do advanced rich client customization. The prerequisites are:

- The Teamcenter rich client must be installed.
- Java must be installed.

The following steps are required to customize the rich client with Eclipse:

1. Set up Eclipse.
2. Set the Eclipse preferences.
3. Launch the rich client from Eclipse.

For the supported versions of Eclipse and Java, see the Hardware and Software Certifications knowledge base article on Support Center.

Tip:

You must use the same version of Eclipse that the rich client uses. From the rich client, choose **Help → About**, and then choose **Installation Details**. On the **Features** tab, verify the version of **Eclipse RCP**.

Set up Eclipse

1. Install the Eclipse IDE with the Plug-in Development functionality.

2. Create a batch file that sets the Teamcenter environment, starts the server, and launches Eclipse using the JDK command line parameters.

   Use the following template to create your batch file:

   ```
   set FMS_HOME=TC_ROOT\tccs
   set JAVA_HOME=jre-install-directory
   set JRE_HOME=jre-install-directory
   set CLASSPATH=TC_ROOT\portal
   set PATH=%FMS_HOME%\bin;%FMS_HOME%\lib;TC_ROOT\portal;%PATH%
   start "Start FCC" /min cmd /c "%TC_ROOT%\tccs\bin\fccstat.exe -restart"
   Eclipse-install-directory\eclipse.exe -vm jdk-install-directory\bin\javaw
   ```

   You can use the *TC_ROOT***portal\portal.bat** file as an example for this batch file.

3. Run the batch file you just created to launch Eclipse in a Teamcenter environment.

Set the Eclipse preferences

1. Verify that the supported Java Runtime Environment (JRE) version is listed and checked.

   Find this option in **Window→Preferences** under the **Java** node, in the **Installed JREs** section.

2. Add the Teamcenter rich client as a target platform.

   Find this option in **Window→Preferences** under the **Plug-in Development** node, in the **Target Platform**section.

   Choose the *TC_ROOT***portal** as the target directory.

   In the **Target Platform** dialog box, ensure the new target is checked.

Launch the rich client from Eclipse

1. Configure the **Debug Configurations** to launch the rich client.

2. Add the **VM arguments** for memory in the **Arguments** tab.

   ```
   -Xmx1g -XX:MaxPermSize=512m
   ```

3. After applying the changes, launch the rich client.

   In the **Debug Configurations** window, click **Debug**.

The rich client should launch to the **Login** page, and you should be able to log in normally.

Eclipse source files

Eclipse provides three files with the source for various extensions. When you're using these extension points with the Eclipse RCP package as the target, there are no issues. But when you switch the target to the rich client, warnings may appear related to these extension points. To avoid these warnings, locate the following files in the Eclipse RCP software (normally in the *eclipse\plugins* folder) and copy them into the rich client's *portal\plugins* folder.

`org.eclipse.core.commands.source_{versionNumber}.jar`
`org.eclipse.core.expressions.source_{versionNumber}.jar`
`org.eclipse.e4.core.commands.source_{versionNumber}.jar`

Ensure your customizations appear

You can run the rich client from Eclipse, and thereby test if your customizations appear in the rich client.

But running your customization from Eclipse is not a true test of whether your customizations will work in an end user's rich client installation. After you create a customization, you must perform the following steps to ensure it appears in the rich client when it is run outside the Eclipse IDE.

You must perform these steps whenever you create a new plug-in to add to the rich client, which should be in most cases of rich client customization (with the exception of style sheets). You must always wrap your rich client customizations in a plug-in to ensure that the out-of-the-box rich client is unchanged, thereby maintaining its integrity the next time you upgrade to a newer version of Teamcenter.

> Note:
>
> If you customize the out-of-the-box rich client files, you can lose those customizations the next time you upgrade.

1. **Export your custom plug-in** from Eclipse to the rich client.

2. From the *TC_ROOT* directory, run the **portal\registry\genregxml** script to register new plugins and other changes with the rich client.

   When the script is finished, a new **RegistryLoader.xml.gz** file is created for each locale.

   > Note:
   >
   > If you make changes to any of the **.properties** files, or you add new plug-ins or change plug-in content, you must run the **genregxml** script to ensure your changes are included when the rich client starts. This enhances performance because it caches the properties so they can be loaded at startup. The script takes no arguments and generates a **RegistryLoader** file for each locale in the **portal\Registry** directory.

3. Ensure that the **-clean** and **-initialize** arguments are added to the **portal.bat** file (between the **start Teamcenter.exe** command and **%***).

   Run the **portal.bat** file to launch the rich client to verify that your customizations appear. To confirm that the plug-in is registered in the rich client, choose **Help→About→Installation Details** and search the list to see the registered plug-ins.

4. If your customizations still do not appear, to clear the cache, delete the **Teamcenter** subdirectory in the user's home directory on the client. This directory is automatically created again when the user starts the rich client.

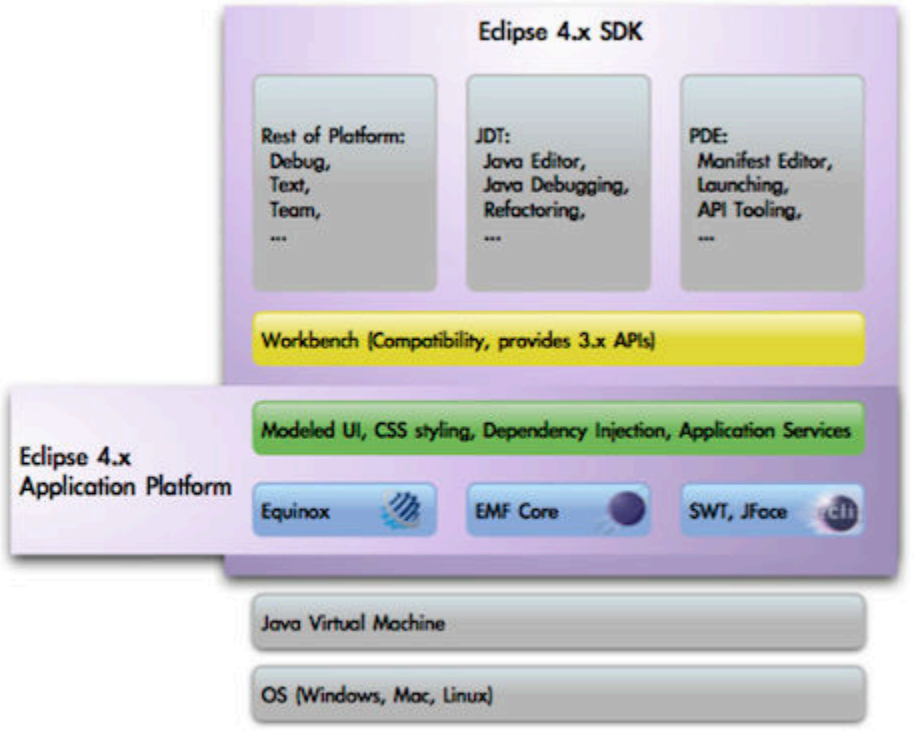
   On a Windows client, it is typically the **%HOMEDRIVE%%HOMEPATH%\Teamcenter** directory. On a Linux client, it is typically the **$HOME/Teamcenter** directory.

   If you delete this directory, the last state of the rich client is lost, and the user interface appears as it does at initial startup.

The rich client and Eclipse 4 migration

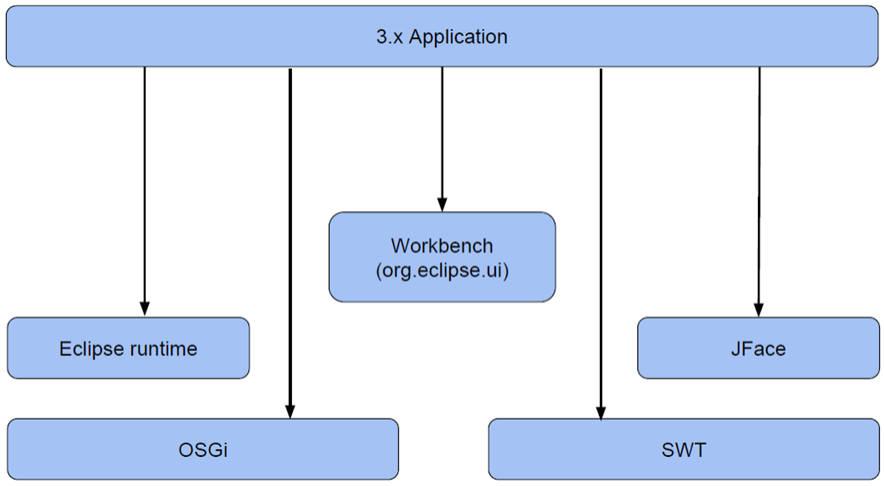
Eclipse 4 architecture overview

Following is a high-level overview of the Eclipse 4.x SDK architecture. Most of the functionality is very similar to the Eclipse 3.x platform.

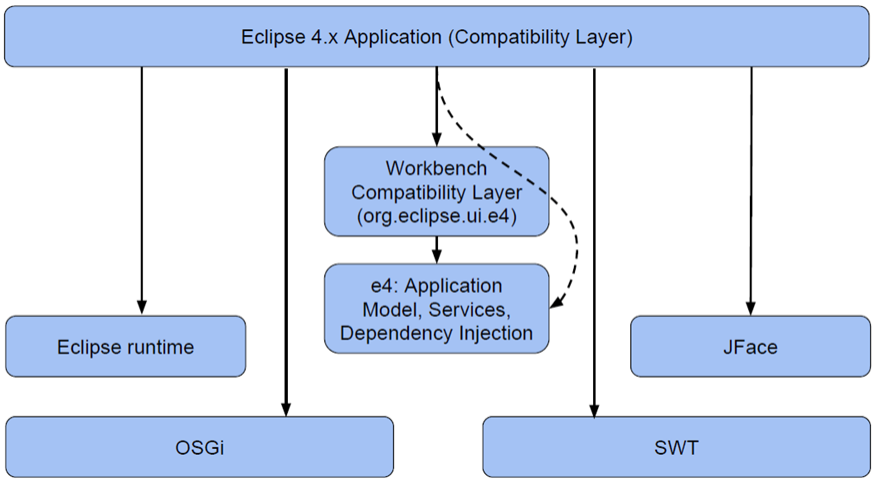


Comparing Eclipse 3 to Eclipse 4

The Eclipse 4 Application Platform (E4AP) differs from the Eclipse 3.x platform in the implementation of the Workbench (**org.eclipse.ui.workbench plugin** for example), and the technologies this new implementation is based on.



On top of E4AP, the 4.0 Workbench offers an implementation of the 3.x Workbench APIs, called the compatibility layer, to provide backwards compatibility for existing well-behaved Eclipse 3.x applications.



Teamcenter rich client and Eclipse 4 strategy

Compatibility layer

Due to the length of time the Teamcenter rich client has been available and the volume of customer customizations based on it, the transition from E3 to E4 will take advantage of E4's Compatibility Layer. One important criteria for existing applications to work well on the compatibility layer is that they should not use any internal workbench API. Aside from this, there should be no source code changes required. This is especially true for existing applications, as it requires almost no migration effort. Additionally, the Eclipse IDE itself is still based on the E3.x API, so if you extend it, you automatically need to use the compatibility layer. If your application uses 3.x internal API, you may have to adjust your code as this code might have been changed.

Use the E4 application model

You still might want to use some of the benefits of the Eclipse 4 programming model, especially for newly developed components and migrate existing Eclipse 3.x RCP application to E4 in stages. This approach allows you to develop new parts of the application using the benefits of the Eclipse 4 programming model as well as reuse existing components. The Teamcenter rich client integrates Eclipse 4 components to create a rich client specific application model used by the compatibility layer, register it as the application model of the rich client, and add new E4 components to it. This allows contributing E4 API-based views, menus, and toolbars to the rich client application. The relevant model in the Teamcenter rich client, application.e4xmi, can be found in the `configuration_12000.0.0` libraries. There are some limitation to do so from rich client customizations due to mixed hybrid application, but you still can leverage E4 application model to learn and evaluate E4, planning your migration for a future Teamcenter release. Furthermore, your newly developed E4 components based upon the Teamcenter 12 application model can be integrated into a pure E4 application without any adaptations.

No impacts on extension point

Teamcenter 12 uses the 3.x compatibility layer along with the new E4 application model. Technically, there is no impact on existing customization code to use Eclipse 3.X extension point to define perspective, view, command, command expression, handler, menucontribution. The extension points introduced by previous versions of Teamcenter still work in Teamcenter 12.0. In addition, you can contribute new views, commands, handlers through model fragment using the new E4 way or by migrating your old extension points to model fragments.

Eclipse internal code change

- **The presentation layer has been removed**

  The package `org.eclipse.ui.presentations` in `org.eclipse.ui.workbench plugin` and the `org.eclipse.ui.presentationFactories` extension point have been removed in Eclipse 4.6.

  Per Eclipse, using the presentation API and extension point to customize the workbench appearance will no longer have any effect. Clients are encouraged to try out the provisional new API in Eclipse 4.X for performing equivalent workbench customization. Complete rendering control can be achieved by supplying an `org.eclipse.e4.ui.workbench.IPresentationEngine`. Customization of fonts, spacing, and color can be achieved by supplying custom CSS style sheets via the `org.eclipse.e4.ui.css.swt.theme` extension point. Customers using internal `org.eclipse.ui.*` APIs need to refactor code to remove API dependency and find replacement. Other internal APIs may not be guaranteed to work in Eclipse 4.X platform. Test your code before refactoring.

- **Certain Elicpse plug-ins have been removed**

  The following Eclipse plugins have been removed in Eclipse 4.6:
  `org.eclipse.core.runtime.compatibility`
  `org.eclipse.update.core`
  `org.eclipse.update.ui`
  See complete porting guides from Eclipse 3.7 to Eclipse 4.6 for API deprecations removal information.

Important changes in Eclipse 4

For convenience, flowing is a list of commonly used patterns that have changed.

visibleWhen checkEnabled

In Teamcenter 11.X based upon Eclipse 3.8 RCP, we could use `checkEnabled="true"` in the `<visibleWhen>` clause for `<menuContribution>` to determine the visibility based upon enabled state of the command, for example `handler.isEnabled()`. This is specific to Eclipse 3.X-based API and is no longer supported in Eclipse 4.X.

This will always return false in an Eclipse 4 application with the compatibility layer. In E4, handlers can be installed on parts, windows, as well as globally on the MApplication. Enablement expressions are now handled by the handler class itself through `@CanExecute` methods. To workaround this issue in the compatibility layer, please add `enabledWhen` or `activeWhen` conditions in your handler extension point in the `<visableWhen>` block.

See example Customer Command 2 in the examples plugin for reference. Use Case: We contribute "Custom Command 2" to File menu. The command should be visible and enabled only when Customer List View is selected and in Customer Example perspective. If Customer Overview is selected, this command should be removed from File menu.

Fast view has been removed

Eclipse 4.x no longer supports the Fast View functionality, so the **Show View as Fast View** button has been removed from Teamcenter 12. Views can still be opened using **Window→Show View**.

Save Perspective As event

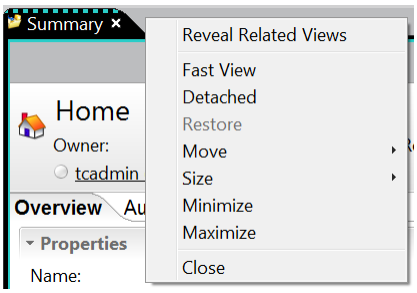
In Teamcenter rich client based upon Eclipse 3.8 RCP, Eclipse fires the **org/eclipse/ui/Perspective/SavedAs** event for **Save Perspective As**. Now in the Teamcenter rich client based upon Eclipse4.6, it now fires the **org/eclipse/e4/ui/LifeCycle/perpSaved** event.

The **org/eclipse/ui/Perspective/SavedAs** event will not work in Teamcenter 12. If you are doing some customization for save perspective, use the **org/eclipse/e4/ui/LifeCycle/perpSaved** listen event instead.

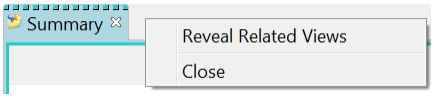
Context Menus on the View Tab

Eclipse 4 removes many of the context menus from the view tabs.

Rich client on Eclipse 3



Rich client on Eclipse 4



Following are the menus removed and the Eclipse 4 alternative, if available:

Fast View

No Alternative.

Detached

Drag and drop the view outside the application.

Move

Drag and drop the view inside the application.

Size

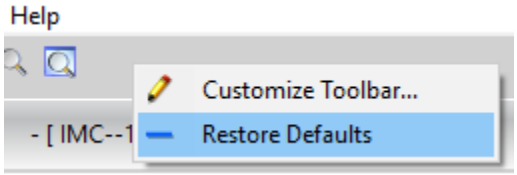
No Alternative.

Minimize, Maximize, and Restore

Use the view control buttons in the top right of the view.

Coolbar is not supported

The CoolbarManager2 from Eclipse 3 has been replaced by the CoolbarToTrimManager in Eclipse 4. Creation of the context menu and adding at a particular location is now done through the EModel Service.



CommandContributionItem

In Eclipse 3, the Menu Manager returned a CommandContributionItem. In Eclipse 4, it returns a HandledContributionItem instead. An example of this, named **HandledContribution Item Example**, is in the **provided sample customization**.

Creating UI components using Eclipse 4 model fragments

Beginning with Teamcenter 12, the rich client runs on the Eclipse 4.6 platform utilizing the Eclipse 3.x compatibility layer. The rich client registers an application model in order to leverage some E4 features. Following are the new programming models used in the rich client.

The E4 application model

The E4 application model is used to describe the structure of an application. This is similar to the DOM of a web page that describes the layout and structure of the user interface. This application model contains the visual elements as well as some non-visual elements of the application.

Components

The visual components are windows, parts, menus, toolbars, and so on. Non-visual components are handlers, commands, key bindings and expression.

Model elements

Each model element has attributes which describe its current state, like the size and the position of a window. The application model also expresses the relationship of the model elements via a hierarchy.

Widgets

The individual user interface widgets, which are displayed in a part, are not defined via the application model. The content of the part is still defined by your source code.

Model

The model itself is created and maintained using EMF, and uses EMF-style patterns for creation and adding elements to containers. The base of the application model is typically defined as a static file. By default, it is called **Application.e4xmi** and is located in the main directory of the plug-in which defines the product extension. This file is read at application startup and is used to construct the initial application model. Changes made by the user are persisted and re-applied at startup.

E4 tools

The E4 tools provide UI tooling to develop and Eclipse 4 RCP application. Parts, commands, and handlers can be added into the application model through the *Application Model Editor*. After opening eclipse, use the update manager to install **Eclipse e4 Tools Developer Resources**, once you double click the application model, the Application Model Editor will be opened in your IDE.

Model fragments

Siemens Digital Industries Software does not recommend modifying the OOTB Teamcenter Application Model to add a new customized view implemented in custom plugin.

In order to statically contribute a new part using the E4 pattern, you contribute to the Application Model in a custom plugin using a *model fragment*. A model fragment statically specifies model elements and the location in the application model to which they should be contributed. In these fragments you contribute to an existing model element using model fragment extension points. A model fragment file typically uses the **.e4xmi** extension. In the model fragment editor, specify the ID of the element in the application model you are contributing to.

Dependency injection

Dependency injection separates the configuration of an object from its behavior.

The Eclipse 4 Application Platform has adopted the use of Dependency Injection (DI) to circumvent several issues:

- Code frequently used global singleton accessors, like **Platform** and **PlatformUI**, or required navigating a deep chain of dependencies.
- Client code needed to know the internals of the Eclipse code base.
- The use of singletons tightly coupled the consumers of *things* to their producer or provider, which inhibited reuse.

Rather than require client code to know how to access a service, the client instead describes the service required, and the platform is responsible for configuring the object with an appropriate service. DI shields the client code from knowing the origin of the injected objects. With DI, one object supplies the dependencies of another object.
A dependency is an object that can be used (a service).
An injection is the passing of a dependency to a dependent object (a client) that would use it.

Example:

In Eclipse 3, a view accessed the Eclipse Selection Service using the PlatformUI singleton.

```
public class DetailsView extends ViewPart {

  @Override
  public void createPartControl(Composite parent) {
    //create tableViewer...

    final ISelectionService ss =
getSite().getWorkbenchWindow().getSelectionService();
    ss.addPostSelectionListener(myListener);
    myListener.selectionChanged(null, ss.getSelection());
  }

  protected void setSelection(Contact contact) {
    // ... do something with the selection
  }

  private final ISelectionListener myListener = new ISelectionListener() {
    @Override
    public void selectionChanged(IWorkbenchPart part, ISelection selection) {
      if (!(selection instanceof IStructuredSelection)) {
        setSelection(null);
        return;
      }

      final IStructuredSelection ss = (IStructuredSelection) selection;
      if (ss.isEmpty() || !(ss.getFirstElement() instanceof Contact)) {
        setSelection(null);
        return;
      }
      setSelection((Contact) ss.getFirstElement());
    }
  };
}
```

Example:

In Eclipse 4, you can get the activeSelection without accessing the selectionService by defining a **DetailsView**.

```
public class DetailsView  {

    @Inject
    public DetailsView(Composite parent) {
        //create tableViewer
```

```
            ...
        }

        @Inject
        public void setSelection( @Optional
                                  @Named(IServiceConstants.ACTIVE_SELECTION)
                                  Contact contact) {
            // ... do something with the selection
        }
    }
```

In Eclipse 3.x you have a Java method chain.
`getSite().getWorkbenchWindow().getSelectionService()`

E4 uses a magic class or string.
`@Optional @Named(IServiceConstants.ACTIVE_SELECTION).`

Annotations

Some of E4's injectors are based upon standard JSR 330 annotations. Additional annotations, such as **@Optional** or **@Persist**, are specific for Eclipse 4.

To get an overview of commonly used annotations, the following list shows all described annotations with the bundle defining them. If you use any of these annotations, you will need a dependency or a package import to these bundles.

Annotation	Bundle
@Active	org.eclipse.e4.core.contexts
@Creatable	org.eclipse.e4.core.di.annotations
@CanExecute	org.eclipse.e4.core.di.annotations
@Execute	org.eclipse.e4.core.di.annotations
@Inject	javax.inject
@Named	javax.inject
@Optional	org.eclipse.e4.core.di.annotations
@Persist	org.eclipse.e4.ui.di
@PersistState	org.eclipse.e4.ui.di
@PostConstruct	javax.annotation
@ProcessAdditions	org.eclipse.e4.ui.workbench.lifecycle
@ProcessRemovals	org.eclipse.e4.ui.workbench.lifecycle
@PostContextCreate	org.eclipse.e4.ui.workbench.lifecycle
@PreDestroy	javax.annotation
@PreSave	org.eclipse.e4.ui.workbench.lifecycle
@Singleton	javax.inject

Annotation	Bundle
@Focus	org.eclipse.e4.ui.di
@AboutToShow, @AboutToHide	org.eclipse.e4.ui.di
@EventTopic	org.eclipse.e4.core.di.extensions
@UIEventTopic	org.eclipse.e4.ui.di

CSS

Eclipse 4 provides an easy way to style the UI widgets using CSS. The new css declarative styling support provides developers with the flexibility of styling their user interface based on a set of properties defined within a CSS stylesheet. CSS selectors used in Eclipse are identifiers, which relate to widgets or other elements, for example predefined pseudo classes. E4 CSS isn't HTML CSS, E4AP has its own CSS rendering engine which can parse eclipse specific CSS Selector.

CSS style sheets can be used to modify SWT widget properties. The SWT widget class is used as an element type, such as **Shell**, **Button**, **Table**. Eclipse publishes a **table** to show the mapping between CSS properties defined for SWT widgets and SWT widget calls.

Model elements of your Eclipse application, like **MPartStack**, **MPart**, or **MTrimbar**, can also be used as selectors for styling. The CSS selectors are based on the Java classes for the model elements.

Example:

You can hide the minimize and maximize button of a **MPartStack** via the following CSS rule.

```
.MPartStack {
swt-maximize-visible: false;
swt-minimize-visible: false;
}
```

Example:

You can assign a CSS ID tag on SWT widget class directly. In CSS, define the **RACPerspectiveHeader** selector.

```
#RACPerspectiveHeader {
    color: #004664;
    background-color: #FFFFFF;
}
```

Example:

In the **RACPerspectiveHeader.java**, set this selector codefuly.

```
public class RACPerspectiveHeader
    extends Composite
{
```

```
        public RACPerspectiveHeader( final Composite parent, final int style )
        {
            super( parent, style );

            m_parentComposite = parent;

            GridLayout gridLayout = new GridLayout( 4, false );
            gridLayout.marginBottom = 1;
            gridLayout.marginHeight = 0;
            setLayout( gridLayout );

            GridData gd = new GridData();

            gd.minimumWidth = 2048;

            setLayoutData( gd );

            setData( ICSSConstants.ECLIPSE_CSS_ID,
    ICSSConstants.ID_RACPerspectiveHeader );

            init();

            ...
        }
    }
```

Example:

Some SWT widget states are captured in pseudo selectors.

```
Button:checked {
    background-color: #FF0000;
}
```

Example:

The style bits, normally passed through the constructor, are available through the style attribute.

```
Button[style~='SWT.CHECK']{
    color: #E2E2E2;
}
```

Customizing Command Suppression

Introduction to customizing Command Suppression

You can suppress menu commands using the Command Suppression application in Teamcenter. If you want to make your custom plug-ins conform to the Command Suppression application, you must add the proper coding to the plug-ins.

The rich client uses Eclipse declarative commands, menu contributions, and handlers to define the vast majority of menu bar and toolbar items. Eclipse uses a Boolean expression based syntax to

allow control over visibility of any specific command in a menu using the **visibleWhen** expression. Just like Eclipse provides some sources like **activeContexts**, **activePartId**, and so on, a rich client **rac_command_suppression** source is defined.

Every command contribution in a custom **plugin.xml** file must have a **visibleWhen** expression consuming the **rac_command_suppression** source using the **with** expression. (That is, those command contributions using the **org.eclipse.ui.menus** extension point to contribute a command using the **menuContribution** source.) Using the **rac_command_suppression** source ensures that the command is visible only when it is not suppressed.

The **rac_command_suppression** source gets called whenever the workbench state changes, for example, when new commands are suppressed, when switching between perspectives, and so on.

The **with** expression (also referred to as the *Command Suppression expression*) consuming the **rac_command_suppression** source must be specified in the **visibleWhen** expression of every command contribution. A template of this Command Suppression expression follows:

```
<with variable="rac_command_suppression">
    <not>
        <iterate operator="or">
            <equals value="command_ID_of_the_command_contribution"/>
        </iterate>
    </not>
</with>
```

Replace *command_ID_of_the_command_contribution* with the command ID of the respective command contribution.

Using the Command Suppression expression in the plugin.xml file

Using the Command Suppression expression in a rich client **plugin.xml** file varies based on the following scenarios:

- The command contribution does not contain a **visibleWhen** expression.

  Example before changes:

  ```
  <menuContribution locationURI="menu:edit?after=cut.ext">
      <command commandId="org.eclipse.ui.edit.cut" id="org.eclipse.ui.edit.cut"
  label="%cutAction.NAME">
      </command>
  </menuContribution>
  ```

  Example after changes:

  ```
  <menuContribution locationURI="menu:edit?after=cut.ext">
      <command commandId="org.eclipse.ui.edit.cut" id="org.eclipse.ui.edit.cut"
  label="%cutAction.NAME">
          <visibleWhen>
              <with variable="rac_command_suppression">
  ```

```
                <not>
                    <iterate operator="or">
                        <equals value="org.eclipse.ui.edit.cut"/>
                    </iterate>
                </not>
            </with>
        </visibleWhen>
    </command>
</menuContribution>
```

- The command contribution contains a **visibleWhen** expression with a nested **and** expression.

  Example before changes:

```
<menuContribution locationURI="popup:org.eclipse.ui.popup.any
?after=org.eclipse.ui.edit.paste">
    <command commandId="com.teamcenter.rac.pasteDuplicate">
        <visibleWhen>
            <and>
                <reference definitionId="com.teamcenter.rac.cme.mpp.inMainView"/>
                <reference definitionId="com.teamcenter.rac.tcapps.
isPasteDuplicateAllowed"/>
            </and>
        </visibleWhen>
    </command>
</menuContribution>
```

  Example after changes:

```
<menuContribution locationURI="popup:org.eclipse.ui.popup.any
?after=org.eclipse.ui.edit.paste">
    <command commandId="com.teamcenter.rac.pasteDuplicate">
        <visibleWhen>
            <and>
                <reference definitionId="com.teamcenter.rac.cme.mpp.inMainView"/>
                <reference definitionId="com.teamcenter.rac.tcapps.
isPasteDuplicateAllowed"/>
                <with variable="rac_command_suppression">
                    <not>
                        <iterate operator="or">
                            <equals value="com.teamcenter.rac.pasteDuplicate"/>
                        </iterate>
                    </not>
                </with>
            </and>
        </visibleWhen>
    </command>
</menuContribution>
```

- The command contribution contains a **visibleWhen** expression with nested expressions (not the **and** expression).

  Example before changes:

```
<menuContribution locationURI="menu:file?after=save.ext">
    <command commandId="com.teamcenter.rac.importAMRuleTree"
icon="icons/importamruletree_16.png" mnemonic="%importAMRuleTreeAction.MNEMONIC">
        <visibleWhen>
            <reference definitionId="com.teamcenter.rac.accessmanager.inMainView"/>
        </visibleWhen>
    </command>
</menuContribution>
```

Example after changes:

```
<menuContribution locationURI="menu:file?after=save.ext">
    <command commandId="com.teamcenter.rac.importAMRuleTree"
icon="icons/importamruletree_16.png" mnemonic="%importAMRuleTreeAction.MNEMONIC">
        <visibleWhen>
            <and>
                <reference definitionId="com.teamcenter.rac.accessmanager.inMainView"/>
                <with variable="rac_command_suppression">
                    <not>
                        <iterate operator="or">
                            <equals value="com.teamcenter.rac.importAMRuleTree"/>
                        </iterate>
                    </not>
                </with>
            </and>
        </visibleWhen>
    </command>
</menuContribution>
```

Command Suppression constraints

Following are constraints on Command Suppression customization:

- The Command Suppression application cannot suppress commands that are dynamic contributions using the **<dynamic> </dynamic>** tag in the **menuContribution** section of the **org.eclipse.ui.menus** extension point.
- Any contributions that are done statically in the code (for example, the **Window** menu) cannot be suppressed.
- Any contributions that are done using Eclipse actions cannot be suppressed.

Naming convention for extensions and Command Suppression

Every command contribution in a plug-in has the **visibleWhen** expression containing the **rac_command_suppression** source provider. The naming convention of the extensions in a plug-in should always start with the bundle symbolic name to clearly indicate which plug-in is providing the contribution.

Assume that the *com.mycom.myapp* plug-in has the *com.mycom.myapp* bundle symbolic name in the **META-INF/MANIFEST.MF**. Assume that this plug-in creates the **Sample Perspective** perspective

and makes contributions to the global menu bar and toolbar that are visible in the perspective. The perspective ID starts with the bundle symbolic name, for example:

```
com.mycom.myapp.perspectives.samplePerspectiveId
```

Because the command contributions from the *com.mycom.myapp* plug-in should be visible only in the **Sample Perspective** perspective, a reference definition can be defined and associated on all command contributions, for example:

```
<definition id="com.mycom.myapp.inMainPerspective">
    <with variable="activeContexts">
        <iterate operator="or">
            <or>
                <equals value="com.mycom.myapp.perspectives.samplePerspectiveId"/>
            </or>
        </iterate>
    </with>
</definition>
```

Assume that the plug-in contributes a sample command. This means that the plug-in must define a command ID using the **org.eclipse.ui.commands** extension point adhering to the naming convention, for example:

```
com.mycom.myapp.sampleCommand
```

To make this sample command visible only in the **Sample Perspective** perspective, and if it is not suppressed from the **Command Suppression** perspective, use a combination of this reference definition and the command suppression source provider, for example:

```
<command commandId="com.mycom.myapp.sampleCommand" tooltip="%sampleCommand.TIP">
    <visibleWhen>
        <and>
            <reference definitionId="com.mycom.myapp.inMainPerspective"/>
            <with variable="rac_command_suppression">
                <not>
                    <iterate operator="or">
                        <equals value="com.mycom.myapp.sampleCommand"/>
                    </iterate>
                </not>
            </with>
        </and>
    </visibleWhen>
</command>
```

Common Teamcenter command IDs

Teamcenter command IDs have many uses. The most common being to add a command to a rich client menu or toolbar, or adding a command to a page or objectSet using an XML style sheet.

- The preferred method to locate commandIDs is to use the rich client command-line utility DumpCMSConfigInfo. It will generate a **CSV** file containing an accurate listing of command IDs for

your specific rich client configuration, as well as one for your view IDs, and one for your application context IDs as well.

```
teamcenter -application com.teamcenter.rac.util.DumpCMSConfigInfo
```

- When a command to create a new object is added within the **objectSet** tag, only commands for **New Item**, **New Dataset**, **New Part**, and **New Other** actions paste the new object with the relation specified in the **<objectSet source="***relation.business-object***">** tag. All other new object types use the default paste relation.

  The following code example shows creating a new object with the **com.teamcenter.rac.newDesign** command ID. However, the default paste relation is used instead of the **IMAN_specification** relation in the **objectSet source** tag:

```
<objectSet source = "IMAN_specification.Design Revision "
defaultdisplay = "tableDisplay" sortby = "object_string" sortdirection = "ascending">
.
.
.
<command actionKey = "newDesignAction" commandId = "com.teamcenter.rac.NewDesign"
renderingHint = "commandbutton"/>
```

The following tables show a few common examples of a menu command, and their associated command Ids.

My Teamcenter File→New menu commands

Menu command	Command ID
File→New→Item	**com.teamcenter.rac.newItem**
File→New→Folder	**com.teamcenter.rac.newFolder**

My Teamcenter Edit menu commands

Menu command	Command ID
Edit→Cut	**org.eclipse.ui.edit.cut**
Edit→Copy	**org.eclipse.ui.edit.copy**
Edit→Paste	**org.eclipse.ui.edit.paste**

Hiding commands in Rich Client

Controlling command visibility

As an administrator, you can limit the Teamcenter application commands that appear in the rich client for members of specific groups or roles. Control of command visibility in other clients, such as Active Workspace, is described in documentation for those clients.

Suppressing commands for specific groups or roles provides at least two benefits:

- Suppressed commands are not available to unauthorized group or role members; you control access.
- Members of the group or role are not distracted by commands that they do not want or need.

You have three methods to limit commands that appear in the rich client:

Method	Effect	Comments
Manually create a command suppression preference.	In the specified application or perspective, for the specified group(s) or role(s), the commands for specified command IDs are suppressed from drop-down menus, toolbars, and context menus.	You can use a manually created preference to suppress a command for a role in all groups where the role appears.
Use the **Suppress Commands** view in the **Command Suppression** application to automatically create a command suppression preference.	In the selected perspective, for the selected group or role within a group, the selected command is suppressed from drop-down menus, toolbars, and context menus. Note: It is possible that a context menu command may have the same name as a command that appears on a drop-down menu or toolbar, but actually refer to a different command ID. In that case, it would not be suppressed.	You can choose to apply the suppression to either • all perspectives in the application to which the selected perspective belongs (default mode) • only the selected perspective. You can use a role preference to show on menus and toolbars the commands that are hidden by a group preference. Commands hidden for a group are always hidden on context menus for all the group's roles.
Use the **Suppress Context Menus** view in the **Command Suppression** application to	In the selected perspective, for the selected group or role within a group, the selected command is suppressed from context menus that appear for objects. Suppressions defined by this means are in addition to suppressions that apply	You can choose to apply the suppression to the context menu for an object selected in either • all views in the selected perspective

Method	Effect	Comments
suppress commands on context menus.	because of settings defined using the **Suppress Commands** view.	• only a selected view and to either • all object types • only a selected object type.

Example: Suppress display of a command for an entire group

If a group does not use Multi-Site Collaboration, then for the My Teamcenter perspective, for all members of the group, you may want to suppress the Multi-Site Synchronization commands.

Example: Suppress display of a command for selected roles in a group

If a business process expects only managers in the logistics group to be able to import and export data, then for all member roles in the **Logistics** group except the **Manager** role, you could suppress the **Import** and **Export** commands.

Example: Suppress display of a command in a perspective

If a perspective is intended for focusing on a particular application task, then you might suppress the **Open** command in the **Manufacturing – Work Instructions** perspective, while the **Open** command is available in the **Manufacturing – BOM Reconciliation** perspective.

Limits to command suppression

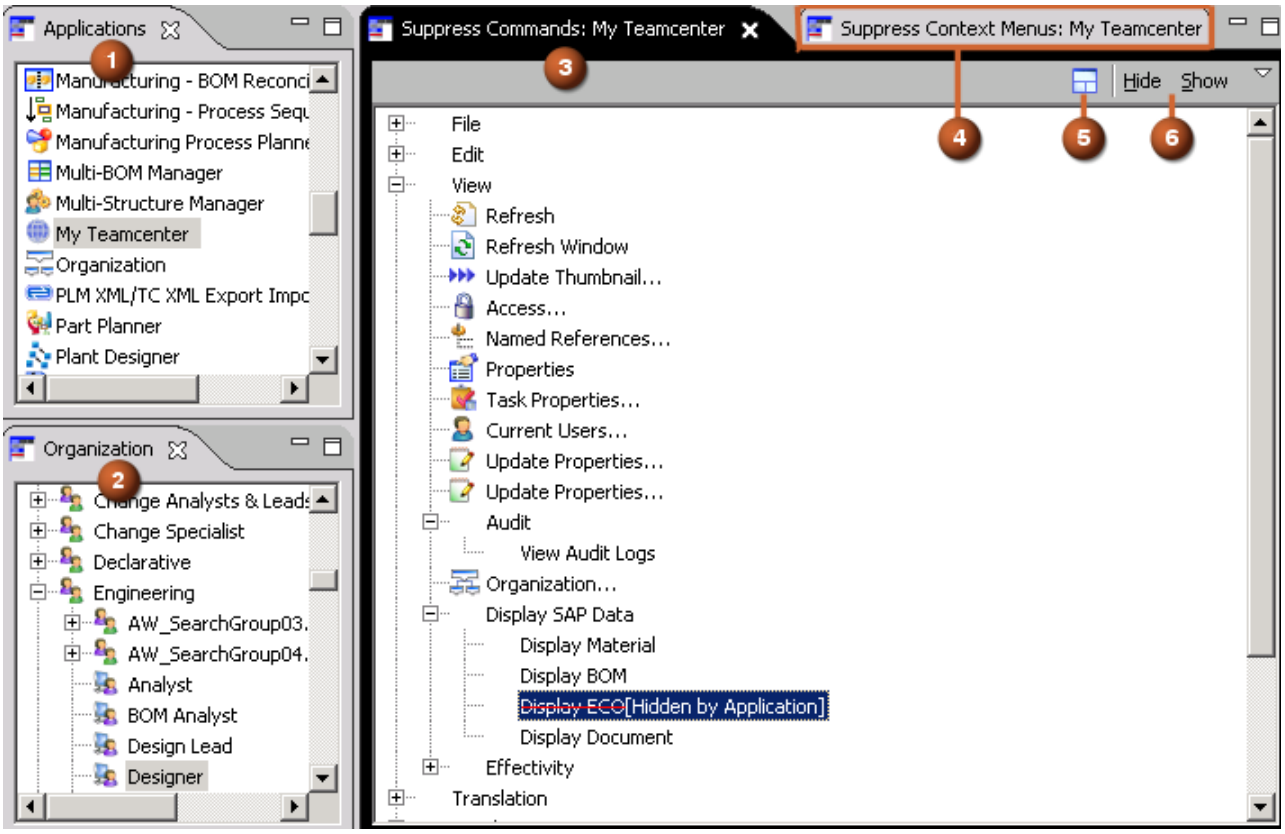
You cannot limit visibility of commands that are:

- Contributed statically in the code (for example, **Window** menu), with the exception of **savePerspective**, **resetPerspective**, and **closePerspective** in the **Window** menu.
- Contributed using Eclipse actions.
- Contributed dynamically. If there are static and dynamic commands within the same parent group, suppressing the parent group also suppresses any contained static commands. It does not suppress the dynamic commands, and the parent group still appears.

  To determine whether a command is dynamic, use the **DumpCMSConfigInfo** utility.

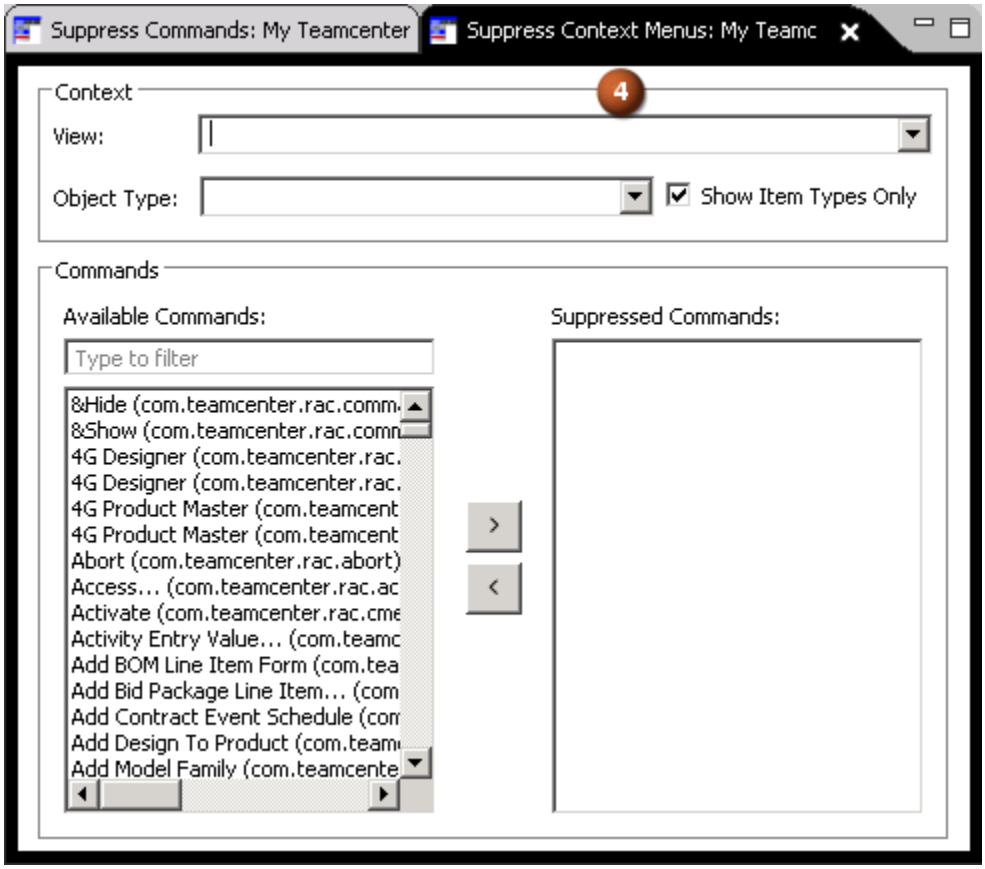
> Tip:
>
> To hide an entire perspective in the rich client, find the perspective's row in the DumpCMSConfigInfo output *activityxxx.csv* file and add the value in the **ID** column to the Teamcenter preference **HiddenPerspectives**.

Command Suppression interface



1	**Applications** view	Displays a list of application perspectives. Select a perspective in the list to display its application's commands in the **Suppress Commands** and **Suppress Context Menus** views.
2	**Organization** view	Displays a tree of the groups and roles that you can select when suppressing commands.
3	**Suppress Commands** view	Displays a tree of the menus, submenus, and commands for the application of the selected perspective. To expand a tree node and display its submenus and commands, click the plus symbol ⊞ next to the node. In the example above, **View** is a menu, **Audit** is a sub-menu, and **View Audit Logs** is a command.
4	**Suppress Context Menus** view	Displays a set of controls for suppressing application commands that might be found on context menus for objects shown in a perspective view. The list of available application commands depends on the perspective selected in the **Applications** view. The **View** and **Object Type** values can be used to limit command suppression to a certain view within the perspective, and to a certain type of object.

5	**Perspective Mode** button	The **Perspective Mode** button is displayed and applicable when the **Suppress Commands** view is active. The button can be set to one of two states, Off (default) or On . The active mode at the time a command is hidden determines whether the command is hidden for all of the application's perspectives (Off), or only the selected perspective of the application (On).
6	**Hide** and **Show** buttons	The **Hide** and **Show** buttons are displayed when the **Suppress Commands** view is active. You can click a button to set the respective status for the menu or command that is currently selected.



Suppress menu bar commands

1. In the rich client, open the Command Suppression perspective.
2. Click the **Suppress Commands** view to make it active.
3. In the **Applications** view, select the perspective containing the commands that you want to suppress.

Many applications have only one perspective. Some have multiple perspectives. For example, **Manufacturing – BOM Reconciliation** and **Manufacturing – Process Sequencing** are perspectives of the Manufacturing Process Planner application. By default, new command suppression settings apply to all perspectives of an application; but, if **Perspective Mode** is on, then new command suppression settings apply only to the selected perspective.

4. Select a group, subgroup, or role from the **Organization** tree.

   For information about how command suppression for group and role interact, see **Manually create preferences to suppress commands in the rich client**.

5. If you want to limit new command suppression settings to only the selected perspective, then turn **Perspective Mode** on.

Off 	This is the default mode. In this mode, when you hide menus and commands, they are suppressed for all perspectives of the application to which the perspective selected in the **Applications** view belongs. Menus and commands hidden while in this mode are struck through with a red line and annotated with **[Hidden by Application]**.
On 	In this mode, when you hide menus and commands, they are suppressed only in the selected perspective. Menus and commands hidden while in this mode are struck through with a red line and annotated with **[Hidden by Perspective]**.

6. In the **Suppress Commands** view, select a menu, submenu, or command from the tree.

7. Click **Hide**.

   The menu, submenu, or command name nodes selected for suppression are struck through with a red line and an annotation identifying the suppression as perspective- or application-based is added.

   To remove the suppression, click **Show**.

8. Choose **File→Save**.

Suppress context menu commands

Preferences that hide commands on drop-down menus and toolbars also hide commands on context menus, if the command ID is the same. You can use the following procedure to hide commands on context menus in addition to those hidden by preferences.

Suppressions defined using the following procedure do not hide commands on drop-down menus and toolbars.

1. In the rich client, open the Command Suppression perspective.

2. Click the **Suppress Context Menus** view to make it active.

3. In the **Applications** view, select the perspective in which you want to suppress context menu commands.

4. Select a group, subgroup, or role from the **Organization** tree.

   For information about how command suppression for group and role interact, see **Manually create preferences to suppress commands in the rich client**.

5. If you want to limit command suppression to a particular view in the perspective, then from the **View** list, select a view.

   If no view is selected, command suppression applies to all views in the perspective.

6. If you want to limit command suppression to a particular item type, then from the **Object Type** list, select an object type.

   If no object type is selected, the command suppression applies to all object types.

   By default, only children of the **item** object type are included in the **Object Type** list. If you want to select an object type other than one based on the **item** type, then clear the **Show Item Types Only** check box.

7. If you want to filter the list of available commands, then enter a filter pattern in the box above the list of commands.

   The list of commands available for a perspective is often very long. Entering a few characters of the command name can make it much easier to find and select the command you are interested in.

8. In the **Available Commands** list, select a command from the list and click › to move it to the **Suppressed Commands** list.

   > Note:
   >
   > Inherited suppressions are not shown in the list.

9. Choose **File→Save**.

Inheritance of group command suppression

The potential for command suppression inheritance differs, depending on whether a command suppression preference has been defined for the role, and on the command location in the user interface.

For these UI locations	Command suppression preference for the role exists?	Then group command suppressions
drop-down menus and toolbars	No	Do apply
drop-down menus and toolbars	Yes	Do not apply
context menus	either No or Yes	Do apply

Note that even if the command suppression preference defined for a role does not contain any values, the command suppression preference for the role must be deleted in order to restore the role's inheritance of group command suppression on drop-down menus and toolbars.

Suppressions defined using the **Suppress Context Menus** view in the **Command Suppression** application do not affect drop-down menus and toolbars.

Example interactions of group and role suppressions on command display

Consider the commands **Aaaa**, **Bbbb**, and **Cccc**. The following table shows the interaction of a command suppression preference defined for *Group* and a command suppression preference defined for *Role A* within *Group*. In this example, no additional context menu command suppression has been defined.

Command suppression preference values	Command suppression preference values	Commands shown on menus and toolbars when logged in as	Commands shown on context menus when logged in as	Commands shown on menus and toolbars when logged in as
Group	*Role A*	*Group / Role A*	*Group / Role A*	*Group / Role B*
Aaaa	No preference defined	Bbbb Cccc	Bbbb Cccc	Bbbb Cccc
Aaaa	Aaaa Bbbb	Cccc	Cccc	Bbbb Cccc
Aaaa Cccc	Cccc	Aaaa Bbbb	Bbbb	Bbbb
No preference defined	Aaaa	Bbbb Cccc	Bbbb Cccc	Aaaa Bbbb Cccc

Command suppression preference values	Command suppression preference values	Commands shown on menus and toolbars when logged in as	Commands shown on context menus when logged in as	Commands shown on menus and toolbars when logged in as
Group	*Role A*	*Group / Role A*	*Group / Role A*	*Group / Role B*
Bbbb	Aaaa	Bbbb Cccc	Cccc	Aaaa Cccc
Bbbb	Preference exists, but no value defined	Aaaa Bbbb Cccc	Aaaa Cccc	Aaaa Cccc

Manually create preferences to suppress commands in the rich client

While the Command Suppression application provides an easy way to hide a command for all members of a group, or for a role within a group, you may want to suppress an application command for a member role in all the groups where the role appears. You can manually create a preference to accomplish this general role-based suppression.

Many examples of such command suppression preferences can be found in an Administration Data Documentation Report of Teamcenter preferences.

> Note:
>
> You cannot suppress a command for all groups and all roles.

The following example procedure shows how to suppress commands in the Navigator (My Teamcenter) perspective for a role in all groups.

1. In My Teamcenter, choose **Edit→Options**.
2. Click **Filters**.
3. Click **Create new preference definition** .
4. In the **Create new preference** panel, type the following statement in the **Name** box:

   ```
   com.teamcenter.rac.ui.perspectives.navigatorPerspective\\*\\[role]\\HIDDEN_COMMANDS
   ```

   Replace [role] with the name of the role for which you want to suppress commands.
5. In the **Create new preference** panel, in the **Values** box, type a comma-separated list of command IDs for the commands that you want to suppress.

6. Click **Create**.

Example:

Following are three examples of using a preference to suppress commands.

To suppress the **Cut**, **Copy**, and **Paste** commands for the My Teamcenter perspective for all groups and the **DBA** role only:

```
com.teamcenter.rac.ui.perspectives.navigatorPerspective\\*\\DBA\\HIDDEN_COMMANDS
```

Values: **cutAction,copyAction,pasteAction**

Note that these command names are unusually simple. See command IDs for examples of typical Teamcenter application commands.

To suppress the **Cut**, **Copy**, and **Paste** actions for the My Teamcenter perspective for the **dba** group only and all roles defined in the group:

```
com.teamcenter.rac.ui.perspectives.navigatorPerspective\\dba\\*\\HIDDEN_COMMANDS
```

Values: **cutAction,copyAction,pasteAction**

To suppress the **Cut** action for the Structure Manager application for all groups and the **DBA** role only:

```
com.teamcenter.rac.pse.PSEApplication\\*\\DBA\\HIDDEN_COMMANDS
```

Values: **cutAction**

Enable Command Suppression for a custom application

You can make custom Teamcenter application perspectives available in the Command Suppression application.

1. Copy the **TC_CS.APPLICATIONS** key and its value from the **portal.properties** file to the **portal_user.properties** file.

2. Append the path name of the new application to the value.

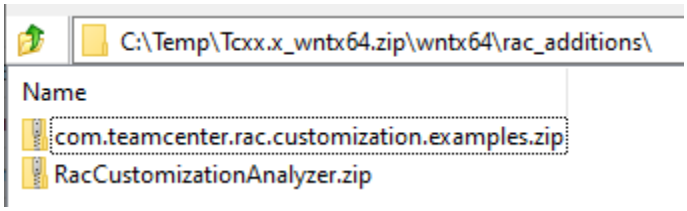
Sample rich client customizations

Provided rich client customization examples

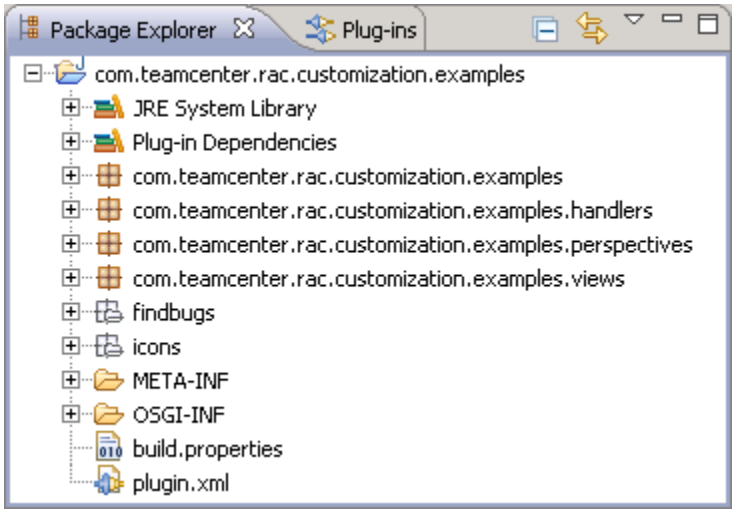
You can use the provided rich client customization example project as a guide to how typical customizations are accomplished.

com.teamcenter.rac.customization.examples.zip

This project is found in the Teamcenter installation zip file specific to your operating system, *Tcxx.x_***wntx64***.zip* or *Tcxx.x_***lnx64***.zip*.



Import this project into your Eclipse rich client platform (RCP) plugin development environment to see the examples in action.



The core of any plugin is the **plugin.xml** file. This contains the information needed to connect the custom code into Teamcenter.

Example listing

Following are some of the examples provided:

- Create a custom perspective (application).
- Create custom views and hide views.
- Add custom menu contributions including **File→New**.
- Add an entry to the **Sent To** menu.
- Permanently remove an existing menu contribution.
- See Teamcenter's use of OSGI services.

Example: Important files and the viewer

The dynamic nature of Eclipse plugins is controlled by several important files. The sample plugin contains the following:

- MANIFEST.MF
- plugin.xml
- build.properties
- component_overview.xml
- OpenServiceComponent.xml

To examine any of these files, double-click them to open the viewer. Opening the **plugin.xml** file in the viewer displays several tabs, showing the majority of the configuration for the plugin on a single location. Changes made on the **Extensions** tab, for example are reflected in the **plugin.xml** file, and vice versa. Using the viewer instead of manually editing is a good way to help validate your configuration.

Take the time to examine the **Extensions** and **Dependencies** tabs, as they contain the most information.

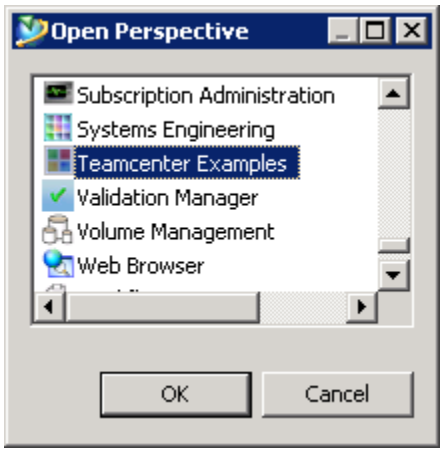
Example: Create a custom perspective

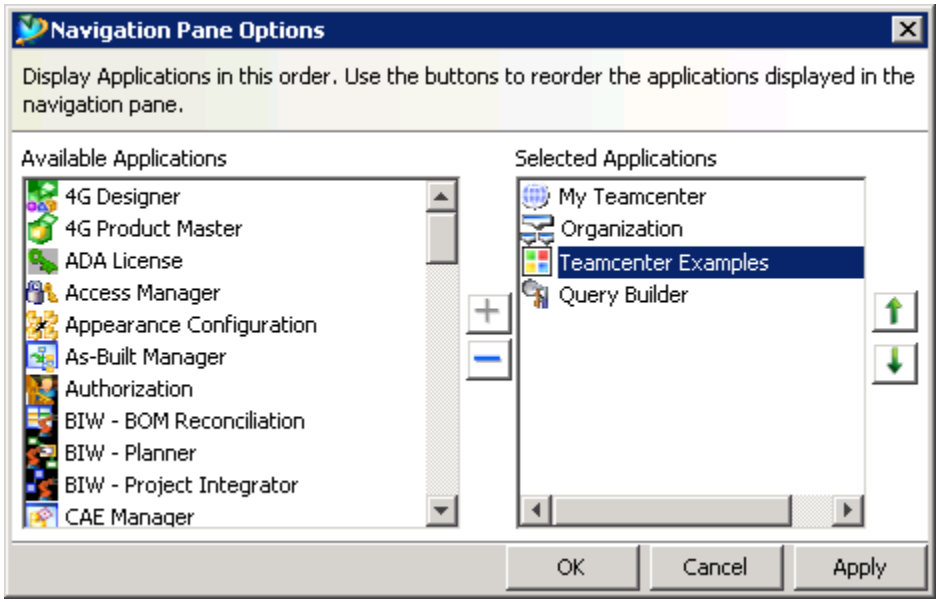
You might want to create your own perspective to the rich client. A perspective is also called an application, and is a collection of related views. When referring to an existing view, you must not use viewDef, since the view is already defined, use viewRef instead.

Included with the provided sample project is the **perspectives** package. This package contains a custom perspective which is the focus of many of the other customizations provided.

com.teamcenter.rac.customization.examples.perspectives

The definition for a custom perspective, **Teamcenter Examples**, is contained in the *TeamcenterExamplePerspective.java* class. You can examine it in the rich client by choosing **Window→Open Perspective** or by opening it with the navigation pane.





This custom perspective implements two layout interfaces and extends the **AbstractRACPerspective** class. It also loads custom views.

Examine the source code to see more details.

Example: Create a custom view

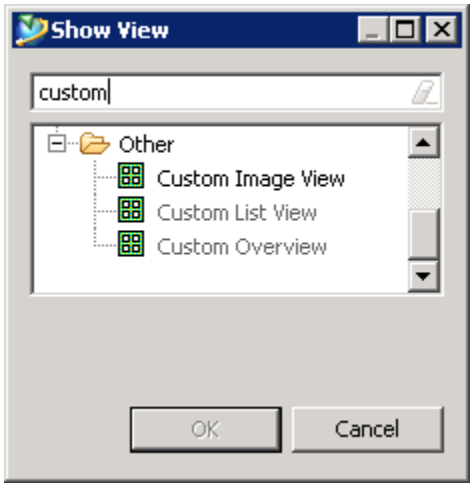
You might want to create your own views in the rich client. Included with the provided sample project is the **views** package. This package contains custom views which demonstrate the various ways to display information and also how to assign icons to the views.

com.teamcenter.rac.customization.examples.views

The definitions for several custom views are contained in the following classes.

- CustomImageView.java
- CustomListView.java
- CustomOverview.java

You can examine them in the rich client by choosing **Window→Show View** or by opening the provided perspective.



Custom Image View	Demonstrates a custom image viewer, which extends the *Thumbnail2DFileView* class.
Custom List View	Demonstrates a custom list viewer, which extends the *AbstractContentViewPart*, and implements the *IContentView* interface.
Custom Overview	Demonstrates a custom list viewer, which extends the *AbstractContentViewPart*, and implements the *IContentView* interface.

Examine the source code to see more details.

> Tip:
>
> Due to changes in Eclipse, if you want CTRL-C / CTRL-V functionality in text fields, you must add `SWTUIUtilities.addFocusListener` to your view. The *CustomListView.java* sample code contains this listener for your reference.

Example: Allow an object type to be opened in an application

Normal behavior for the rich client is to open an object in its associated application (also known as a perspective). For example, if you have another application open and you select **My Worklist** from the **Quick Links**, the rich client switches to the **My Teamcenter** application, because that is the application associated with the inbox.

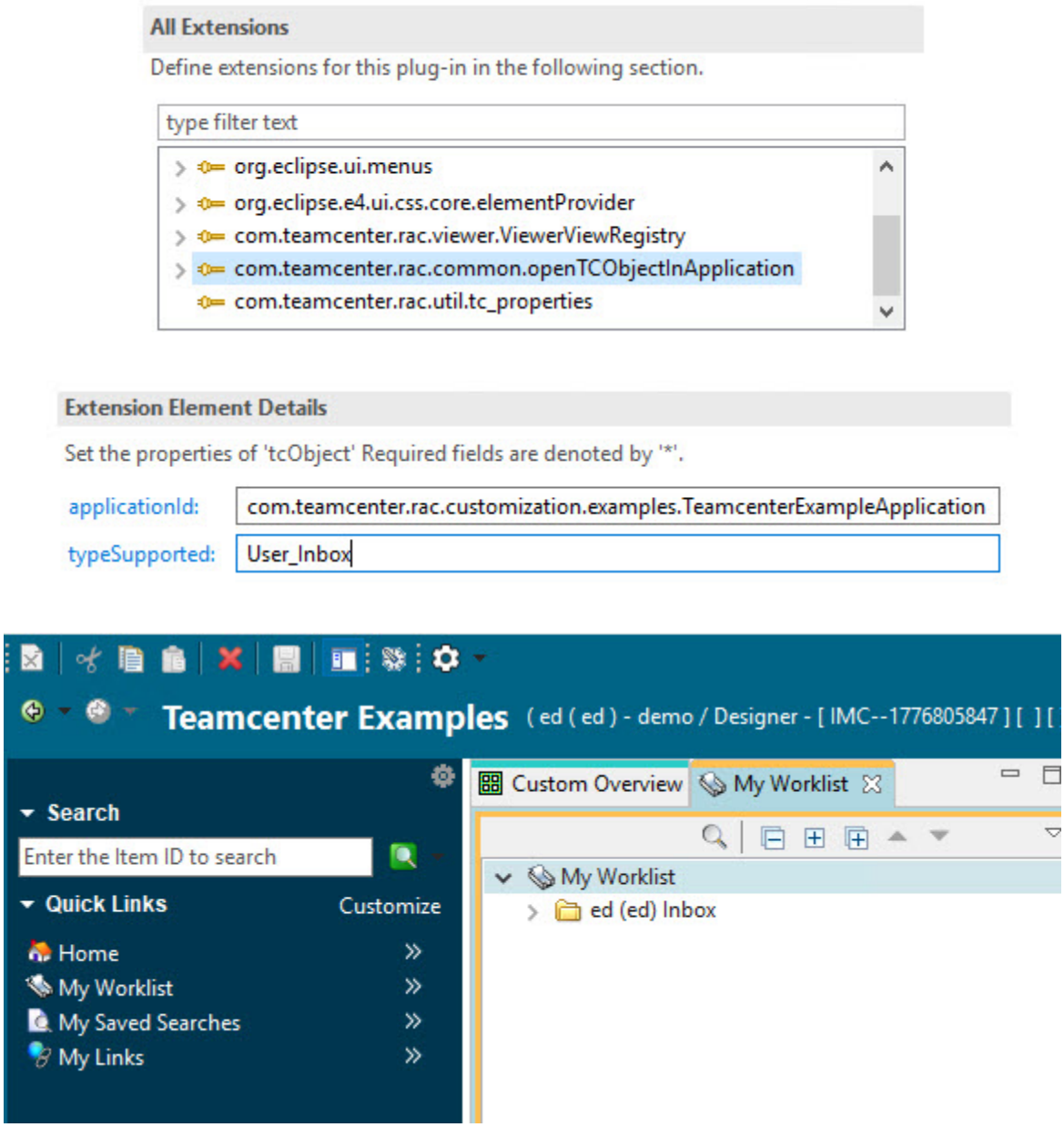
If you wish to change this behavior and allow an object to be opened in other applications, you must modify the *plugin.xml* file.

com.teamcenter.rac.common.openTCObjectInApplication

This extension point allows you to specify an object type and an application in which it can be opened.

Example:

In this example, the inbox is added to the example application. This allows the example application to remain open when you select the inbox.



Example: Use logging in your code

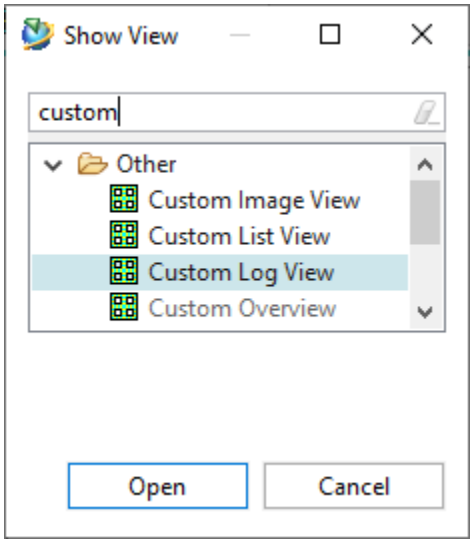
If you want to add logging to your rich client Java code, see the provided example view. The rich client uses the log4j2 logger.

com.teamcenter.rac.customization.examples.views

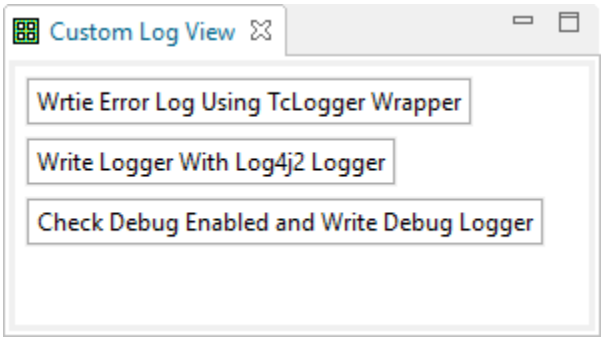
You can find the definition for a custom view in the following class.

- CustomLog4j2View.java

You can use this view in the rich client by choosing **Window→Show View**.



With this view, you can choose the buttons to output messages to the log file.



Examine the source code to see more details.

Example: Command handlers

com.teamcenter.rac.customization.examples.handlers

When creating custom commands, you also need custom handlers. Several handlers have been provided as examples. Match them with the commands that have been provided.

Some of the examples extend the **AbstractHandler** class, and some extend the **NewBOHandler** class.

The abstract handler

The basic Eclipse handler class that you extend for nearly all commands.

The business object handler

A Teamcenter provided handler class specifically for working with the New Business Object wizard.

Example: OSGi

An OSGI (Open Service Gateway Initiative) service is a Java object. It is registered in the Service Registry under the name of a Java interface. Unlike extensions, which are scanned and registered during start-up, services are not automatically registered. To register a service, you must create a Component Definition in the project's OSGI folder, and then make a call to the OSGI API to register the object as a service.

You will find several OSGI service examples in the sample plugin project.

component_overview.xml

Implemented in the **DisplayObjectInOverview** class. See that source file for details.

OpenServiceComponent.xml

Implemented in the **CustomizationExampleOpenService** class. See that source file for details.

These files define the services used in the sample plugin project.

Example: Localize your customizations

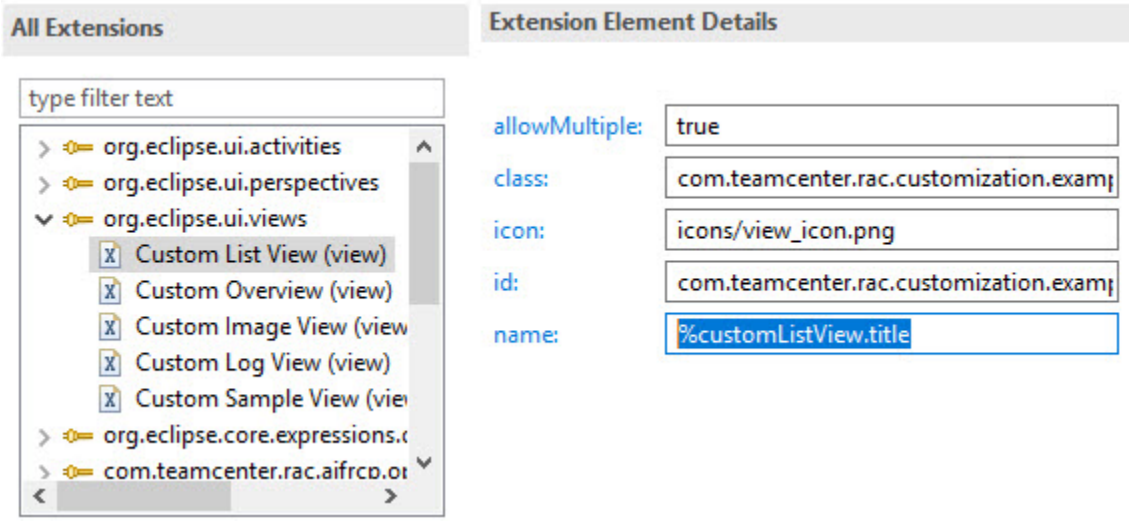
Add localization to your customization by creating or modifying *.properties* files in your plugin project.

For more information about localizing Eclipse plug-ins, see the Eclipse documentation. Also, you can unzip any of the Teamcenter plug-ins to see how the rich client uses localization. The **plugin.properties** and **plugin.xml** files in the **aifrcp** plug-in are good examples.

> Tip:
>
> When you deploy your plugin, remember to regenerate the registry XML files and clear the rich client cache.

com.teamcenter.rac.customization.examples

The *plugin.properties* files located at the root of the plugin project contain the localization information for use within the *plugin.xml* file. They are retrieved by using a percent sign (%) and the name of the localization key.



com.teamcenter.rac.customization.examples.handlers

The *messages_locale.properties* files located in this package contain the localization information for use within the *.java* files here. They are retrieved by using the `.getString` method of the `com.teamcenter.rac.ui.Messages` class.

com.teamcenter.rac.customization.examples.handlers.param

The *param_locale.properties* files located in this package contain the localization information for use within the *.java* files here. They are retrieved by using the `.getString` method of the `com.teamcenter.rac.util.Registry` class.

Regenerate the registry XML files

Any time you make changes to the *.properties* files, add new plugins, or modify existing plugins, you must regenerate the registry files for your rich client. In the **TC_ROOT** directory for your rich client, run the following script:

```
portal\registry\genregxml
```

When the script is finished, new *RegistryLoader.xml.gz* files are created, one for each locale.

Clear the rich client cache

Manually deleting the cache directory will force the rich client to recreate it. To clear the cache, delete the *RAC* directory in the user's home directory on the client. This directory is automatically created again when the user starts the rich client.

On a Windows client, it is typically under the **%HOMEDRIVE%%HOMEPATH%\Teamcenter** directory.

On a Linux client, it is typically under the **$HOME/Teamcenter** directory.

Example: Other contents in the sample plugin project

You must examine the plugin project to get the full details of all the samples provided. Many of them will only appear when you have the **Teamcenter Examples** perspective open.

Command contributions	Menu, toolbar, and popup examples are given including commands with parameters.
Custom new business object wizard	Two custom new business object wizards are available at **File→New** when the **Custom Overview** is active.
Permanently hidden existing command	The **Help→Help Library** is permanently removed from the menu.
Custom theme	A new custom theme
Style sheets	Several example style sheets and a batch file to import them are included in the *Stylesheets* directory.

Hide a view

You can hide views based upon user context using the **Command Suppression perspective** in the rich client. However, in order to hide a view from all users at all times, you must modify the **plugin.xml** file of your Eclipse rich client plug-in project. The visibility is controlled using the **org.eclipse.ui.activities** extension point. By binding an empty activity to the view, the view is suppressed regardless of user context.

In the following example, an empty activity, **Hide Custom List View**, is created, and then attached to a **Custom List View** view.

```
<extension point="org.eclipse.ui.activities">
    <activity
            id="com.teamcenter.rac.customization.examples.hidecustomlistview"
            name="Hide Custom List View">
    </activity>
    <activityPatternBinding
            activityId="com.teamcenter.rac.customization.examples.hidecustomlistview"
            isEqualityPattern="true"
            pattern="com.teamcenter.rac.customization.examples/
                    com.teamcenter.rac.customization.examples.view.CustomListView">
    </activityPatternBinding>
</extension>
```

The actual **name** and **id** used for the empty activity are not relevant, however it is recommended to choose something descriptive.

Distributing rich client customizations

Export your custom plug-in to the rich client

To test your custom plug-in, export it as a JAR file to the rich client *TC_ROOT***portal\plugins** directory.

1. In Eclipse, right-click the customization project and choose **Export**.
2. In the **Export** dialog box, choose **Plug-in Development→Deployable plug-ins and fragments**.
3. Click **Next**.
4. Click the **Browse** button to the right of the **Directory** dialog box and browse to the *TC_ROOT***portal** directory.
5. Click **Finish**.

   The JAR file is automatically generated into the *TC_ROOT***portal\plugins** directory.

Before running the rich client to verify the customization, run the *TC_ROOT***portal\registry\genregxml** file to register the plug-in with the rich client, and clear the rich client cache.

genregxml

This script regenerates the rich client's registry cache. This binary cache file is used for fast access to the rich client's configuration information instead of the individual text-based properies files, xml files, and so on.

From the *TC_ROOT* directory, run the **portal\registry\genregxml** script to register new plugins and other changes with the rich client.

When the script is finished, a new **RegistryLoader.xml.gz** file is created for each locale.

If you make changes to any of the **.properties** files, or you add new plug-ins, or change plug-in content, or any **JAR** file is added or modified, you must run the **genregxml** script to ensure your changes are included in the cache. This enhances performance because it caches the properties so they can be loaded at startup. The script takes no arguments and generates a **RegistryLoader** file for each locale in the **portal\Registry** directory.

```
RegistryLoader_de.xml.gz
RegistryLoader_en.xml.gz
RegistryLoader_es.xml.gz
RegistryLoader_fr.xml.gz
```

```
RegistryLoader_it.xml.gz
.
.
.
```

Process for distributing customizations to the rich client

The basic customization technique is to create a plug-in that contains the customizations and deploy the custom plug-in JAR file to the rich client installation's *TC_ROOT***portal\plugins** directory. This way, when you upgrade to a newer version of Teamcenter, you can simply install the custom plug-in without having to alter files in the Teamcenter installation.

Perform the following steps to distribute your rich client customizations created in Eclipse:

1. Package your custom files.

2. Create a feature file to install the custom files.

Package custom rich client files

When you create customizations to the rich client, typically they are comprised of JAR files exported from Eclipse that must be installed to the *TC_ROOT***portal\plugins** directory. Package these JAR files into a ZIP file so you can install them using Teamcenter Environment Manager (TEM).

1. To create a plug-in JAR file of your customization, in Eclipse right-click the project and choose **Export→Plug-in Development→Deployable plug-ins and fragments**.

2. Zip your custom JAR file into a **rac_***feature-name***.zip** file (for example, **rac_samplecust.zip**) with the root path for the JAR file set to **\plugins**.

   The use of **rac_** in the name signifies that the ZIP file contains a rich client feature. For examples of these files, see the **portal\compressed_files** directory on the Teamcenter installation source.

3. Create a directory structure to hold the ZIP files, for example, **portal\compressed_files**.

   You can use any directory structure you want. For convenience, this structure is the same used on the Teamcenter installation source.

4. Copy your ZIP files to the **portal\compressed_files** directory.

5. Create a feature file so you can install the custom files using TEM.

Create a feature file for rich client customizations

After you have packaged your rich client customization files into a ZIP file, you can create a feature file so that you can install them using Teamcenter Environment Manager (TEM).

1. Package your rich client customization files JAR files into a ZIP file.

2. In a text editor, create an XML feature file that points to the ZIP file, for example, **feature_rac_mycust.xml**.

> Tip:
>
> For examples of **feature_rac_***feature-name***.xml** files, see the **install\modules** directory on the installation source.

Following is an example of a feature file:

```
<?xml version="1.0" encoding="UTF-8"?>
<feature>
   <name value="Rich Client Customizations"/>
   <size value="21"/>
   <os value="windows" version="5.1"/>
   <os value="sunos" version="5.10"/>
   <os value="hp-ux"   version="11.11"/>
   <root value="true"/>
   <group value="package"/>
   <singular value="true"/>
   <guid value="7560C67ED5B01B65C39B8DF7364066C8"/>
   <relation>
      <depends>
         <or value="FF18D25DA73019F880BCFFBC0029CA28"/>
         <or value="E2564104E1B964BB23D78078DBA34EEA"/>
         <or value="A2564824E1B434AC23D70178DBA34BCA"/>
      </depends>
   </relation>
   <files>
      <code>
         <unzip file="portal/compressed_files/rac_samplecust.zip"
            todir="portal"/>
      </code>
   </files>
</feature>
```

In the feature file, change the following lines:

- In the **size** tag, change the **value** text to the approximate size (in megabytes) of your customization files.
- Add or remove **os** tags to match the platforms where you use rich client.
- In the **guid** tag, change the **value** text to a unique value exactly 32 characters in length and composed of any combination of the numbers 0–9 and the letters A–F.
- In the **unzip** tag, change the **file** text to the name of the ZIP file that contains your customizations.

To ensure that the **todir="portal"** line correctly unzips to the **portal\plugins** directory, ensure that **\plugins** is set as the location in the ZIP file.

Tip:

If your customizations include a packaged Business Modeler IDE template project, you can edit the packaged project's feature file to include the new rich client feature by adding a node similar to the following. This is not required; you can still have a separate feature file for the rich client customizations.

```
<feature>
   <name value="feature-display-name"/>
   <property name="feature_id" value="rac"/>
   <property name="bmide_optional" value="true"/>
   <relation>
      <depends>
         <or value="FF18D25DA73019F880BCFFBC0029CA28"/>
         <or value="E2564104E1B964BB23D78078DBA34EEA"/>
         <or value="A2564824E1B434AC23D70178DBA34BCA"/>
      </depends>
   </relation>
   <size value="0"/>
   <singular value="true"/>
   <guid value="85B3827CD5A0FA61220398F6C60C21AB"/>
   <files>
      <code>
         <unzip file="portal/compressed_files/rac_feature-name.zip"
            todir="portal"/>
      </code>
   </files>
</feature>
```

3. Copy the XML feature file to the *TC_ROOT***install\install\modules** directory on the installation source.

4. Open the *TC_ROOT***install** directory and run Teamcenter Environment Manager (TEM).

5. Click **Configuration Manager** and click **Next**.

6. Click **Perform maintenance on an existing configuration** and click **Next**.

7. Ensure the correct configuration is selected and click **Next**.

8. Ensure **Add/Remove Features** is selected and click **Next**.

9. Under the **Extensions** node, select the check box for your new files. The name of the box is the one you gave it in the feature file. Click **Next**.

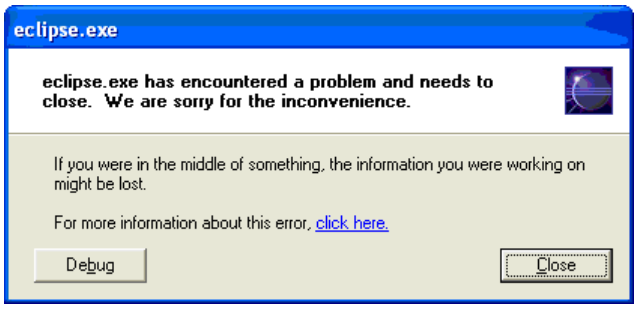
10. In the **Confirm Selections** panel, click **Next** to start installing your customization files.

Troubleshooting rich client customization

Common problems in rich client customization

Eclipse startup error

If you get the error shown in the following figure during Eclipse's startup, either the Java Runtime Environment is not installed or the **PATH** statement does not contain the *JDK-installation-directory***bin** directory.



Customizations from a new plug-in do not appear

Teamcenter has a base set of Eclipse plug-ins for the rich client. They are located in the *TC_ROOT***portal\plugins** directory, and the file names start with **com.teamcenter**. When you add a new plug-in, you deploy it into this directory.

If you add or remove a plug-in when customizing the rich client and your changes do not appear, delete the **Teamcenter** subdirectory in the user's home directory on the client. This clears the cache.

On a Windows client, it is typically the **%HOMEDRIVE%%HOMEPATH%\Teamcenter** directory. On a Linux client, it is typically the **$HOME/Teamcenter/** directory.

Unable to load application error

When you start the rich client, you may get the following error:

```
Unable to Load Application: application-name
Details java.lang.reflect.InvocationTargetException
```

This error can occur when you expose custom classes using Eclipse but do not include the class information in the **MANIFEST.MF** file. To correct this error, ensure that you have added your plug-in on the **Runtime** tab in the **Exported Packages** pane in Eclipse. This adds class information to the **MANIFEST.MF** file.

Previous to Teamcenter 2007.2, the classpath alone was sufficient to handle custom class information. In later Teamcenter versions, you must add your plug-in to the **Exported Packages** pane on the project **Runtime** tab to ensure that the **MANIFEST.MF** file is correctly updated.

Rich client debugging

Rich client debugging tools

You can debug your customizations to the rich client by using the following tools:

- **Print Object** view

  Displays the selected object's internal attribute values. Use it to determine if there is an incorrect value for an attribute.

- **Communication Monitor** view

  Shows you the calls between the rich client and server. Use it to determine if your data is being correctly exchanged between the server and client.

- **Performance Monitor** tool

  Tracks the calls between the server and the database. Use it to determine how fast the server and the database are communicating.

- **DB Walker** view

  Examines the database. This view is for Siemens Digital Industries Software internal use only.

There are also standard Eclipse views that can help you debug your customization.

Debug using the Print Object view

You can use the **Print Object** view in the rich client to find objects or retrieve information about objects in the Teamcenter database.

Retrieve an object by UID

Enter an object's tag (UID) here to retrieve that object from the database. Once retrieved, you can copy a URL link to that object, and then paste it within the rich client or into an email, for example.

Copy a link to an object

Use the **Copy** button  to copy a URL link to this object onto the rich client and Windows clipboards.

See previously referenced objects

Use the drop-down list at the top-right of the view to switch between previously viewed objects.

Filter which attributes you see

Use the two drop-down lists at the bottom of the view to control which attributes are displayed.

Save attributes to a file

Use the **SAVE** button to save the shown attributes to a file for future reference.

Debug using the Communication Monitor view

1. In the rich client, choose **Window→Show View→Communication Monitor**.

   The values appear in the **Communication Monitor** pane at the bottom of the rich client.

2. To choose what you want to see in the monitor, click the **Menu** button in the **Communication Monitor** pane and choose one or more of the following:

   - **Show Server Calls**

     Displays an entry for each call to the server.

   - **Show Stack Trace**

     Displays the call stack trace for each call to the server.

   - **Show Request**

     Displays the XML request sent to the server.

   - **Show Response**

     Displays the XML response returned by the server.

   - **Show Timing**

     Displays the length of the server call in seconds.

3. If you want to clear the data, click the **Clear** button in the pane.

4. If you want to save the data to a file, click the **Save As** button in the pane.

Debug using the Performance Monitor tool

1. Set the **TC_PERFORMANCE_MONITOR** environment variable to **0**.

   The tool appears the next time you open the rich client.

2. To see the data from the server and client, click the **Report** button. The data is also logged along with the text in the **Log comment** box. It also resets the counters.

   - The SQL and server CPU statistics are retrieved from the server.
   - **Wallclock time since reset** is the time since the last reset.

     Times are shown in milliseconds. If the top of the Performance Monitor states that the Hi-Res timer is in use, times are accurate to 1 millisecond. Otherwise the standard operating system clock is in use; Microsoft Windows uses a 60 Hertz clock, so times on Windows are accurate to about 16 milliseconds.
   - **Client CPU time** is available only with the Hi-Res timer.
   - **Server calls made** is a count of all calls made to the server, not including the call to get the SQL statistics.

   > Note:
   >
   > If you select the **Reset on first server call** check box, the Performance Monitor is reset after the next server call after you click the **Report** button.

3. To clear the data and reset all counters, click the **Reset** button.

   > Note:
   >
   > If you select the **Reset on first server call** check box, the Performance Monitor is reset after the next server call after you click the **Reset** button.

4. To save the data to a file, click the **Save** button.

Debug using Eclipse views

Some Eclipse views may be helpful when debugging your customizations:

- **Console** view

  Shows the standard output, standard error, and standard input for your program. Used to debug problems with customizations and to monitor rich client activity.

- **Progress** view

  Shows the progress of background jobs. You can connect this view to your customizations if you want to see the progress when your customizations run.

For more information, see the following URL in the Eclipse help:

https://www.eclipse.org/documentation/

Rich client customization analyzer

The rich client customization analyzer overview

You can create a report which lists your rich client customizations.

The rich client customization analyzer (**RacCustomizationAnalyzer**) examines the content of the *plugin.xml* files contained within the **jar** files that you specify. It creates a report of the contributions that it finds.

Limitations

This utility only analyzes programmatically added custom Java classes and declared extension-point contributions.

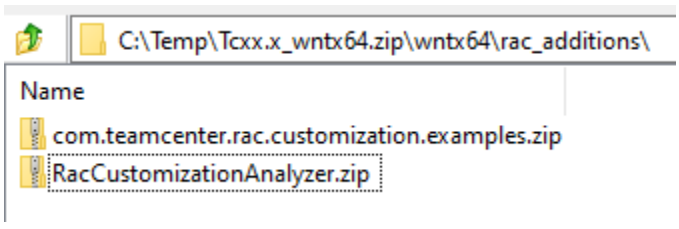
Only the directory specified will be scanned. Subdirectories are not recursively scanned.

The script cannot find any customizations that modify *.properties* files or other OOTB files, custom style sheets, modified Teamcenter preferences, deployed BMIDE template, and so on.

The rich client customization analyzer install

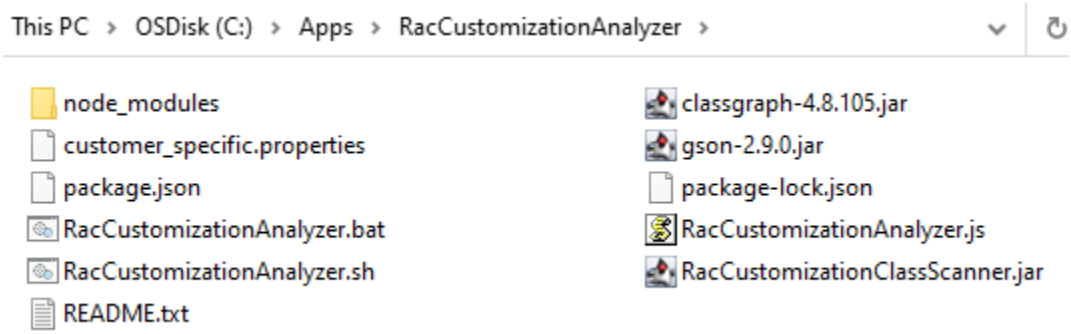
The analyzer script is located within the Teamcenter installation zip file specific to your operating system, **Tcxx.x_wntx64.zip** or **Tcxx.x_lnx64.zip**, in the *rac_additions* folder.

Locate the *RacCustomizationAnalyzer.zip* and unzip it to a local folder.



Example:

In this example, the analyzer is unzipped to the *C:\Apps\RacCustomizationAnalyzer* folder.



RacCustomizationAnalyzer syntax

This utility scans the provided rich client portal root directory to find all custom jar files that match the provided name pattern. It extracts the plugin.xml file from each matching jar, analyzes it, and then writes the extension contribution information to an Excel file in the provided destination directory. The utility also locates and analyzes custom Java classes. Use the *customer_specific.properties* file to ignore specific **JAR** files.

Syntax

RacCustomizationAnalyzer.bat --tc_portal_root=*path to portal root* **--destdir=***output path* **--jar_name_pattern=***jar file patterns*

Arguments

--tc_portal_root (required)

Where to scan for customization jars. Usually points to the rich client portal root..

--destdir (optional)

Where to create the analysis results. If no value is provided, it will default to .\ *RACCustomizationAnalysisResults*.

--jar_name_pattern (optional)

Custom jar file name pattern. Use commas to separate multiple patterns. If no jar name pattern is provided, all jar files will be processed from the tc_portal_root directory except known jars like com.teamcenter.*, org.apache.*, org.eclipse.*, com.google.*, and so on.

-h or --help

Print usage information.

Environment

- Must be run from a Teamcenter command prompt as specified in .
- Node.js must be installed.

Files

- An Excel spreadsheet is created in the destination directory as the output report.
- The *customer_specific.properties* file is read when the utility runs. This file contains the **EXCLUDE_JAR_NAME_LIST** property, which is a list of **JAR** file names that will not be processed. By default, this list is empty.

Restrictions

None.

Examples

In these examples, the rich client is installed at *C:\Apps\tc\clients\4tier\portal*, this is the **tc_portal_root**. Each example must be a single line, but are shown here as multiple lines for clarity.

- Process all jar files found.

```
RacCustomizationAnalyzer.bat --
tc_portal_root="C:\Apps\tc\clients\4tier\portal"
```

- Process all jar files found and create the output in the *C:\Temp\AnalysisResults* directory.

```
RacCustomizationAnalyzer.bat --
tc_portal_root="C:\Apps\tc\clients\4tier\portal"
     --destdir="C:\Temp\AnalysisResults"
```

- Process only the **com.test.xyz** jar file and create the output in the *C:\Temp\AnalysisResults* directory.

```
RacCustomizationAnalyzer.bat --
tc_portal_root="C:\Apps\tc\clients\4tier\portal"
     --destdir="C:\Temp\AnalysisResults"
     --jar_name_pattern="com.test.xyz"
```

- Process the **com.test.xyz** and the **com.abc.def** jar files and create the output in the *C:\Temp\AnalysisResults* directory.

```
RacCustomizationAnalyzer.bat --
tc_portal_root="C:\Apps\tc\clients\4tier\portal"
```

```
--destdir="C:\Temp\AnalysisResults"
--jar_name_pattern="com.test.xyz,com.abc.def"
```

Analyze your rich client installation

Generate a report of extension use.

Prerequisites

- **NodeJS** must be installed.
- **RacCustomizationAnalyzer** must be installed.

Procedure

1. Open a Teamcenter command prompt.

   Use the *server-side environment*.

2. Navigate to the *RacCustomizationAnalyzer* directory.

   Example:

   ```
   cd /d c:\Apps\RacCustomizationAnalyzer
   ```

   You must have write access to this directory.

3. Run the script, specifying which jar files to process.

   In this example, you examine the OOTB customization example jar file.

   ```
   RacCustomizationAnalyzer.bat
   --tc_portal_root="C:\Apps\tc\tcnnnn\Clients\4TierRichClient\portal"
   --destdir="C:\Temp\RACCustomizationAnalysisResults"
   --jar_name_pattern="teamcenter.rac.customization.examples_nnnn"
   ```

4. Observe the results

   Note:

   ```
   --- The provided TC Portal Root:
   C:\Apps\tc\tcnnnn\Clients\4TierRichClient\portal,
       the program will analyze custom jars located in
   C:\Apps\tc\tcnnnn\Clients\4TierRichClient\portal\plugins
   --- User provided destination directory:
   ```

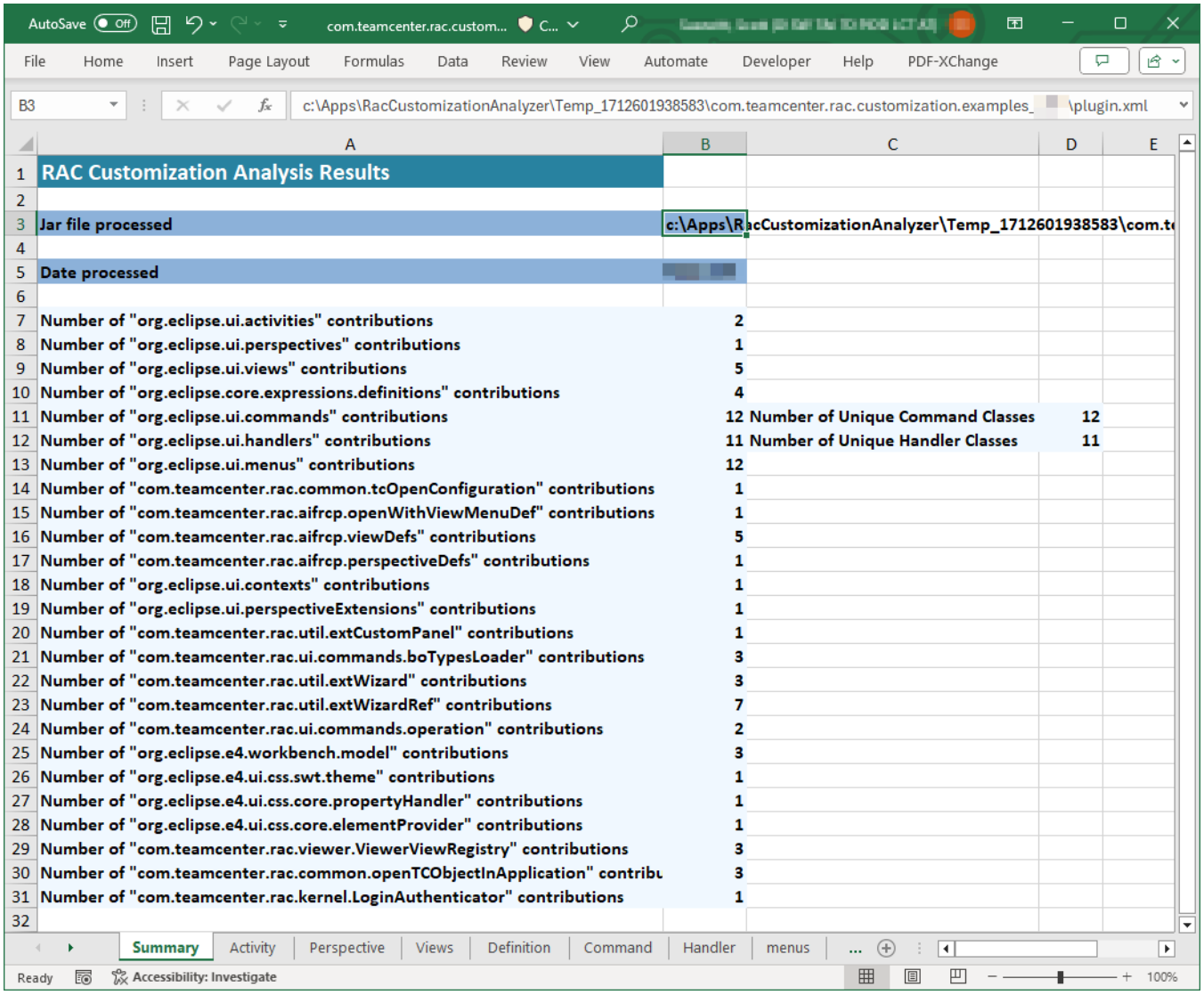
```
C:\Temp\RACCustomizationAnalysisResults
--- User provided jar file name pattern:
"teamcenter.rac.customization.examples_nnnn"
--- Start analyzing customization jars
--- Found 1 customization jars:
         com.teamcenter.rac.customization.examples_nnnn.jar
--- Start extracting
com.teamcenter.rac.customization.examples_nnnn.jar
--- Finish extracting
com.teamcenter.rac.customization.examples_nnnn.jar
--- Number of plugin.xml files found: 1
--- Begin processing
com.teamcenter.rac.customization.examples_nnnn\plugin.xml
        Found customization on extensionPoint =
org.eclipse.ui.activities
        Found customization on extensionPoint =
org.eclipse.ui.perspectives
        Found customization on extensionPoint = org.eclipse.ui.views
        Found customization on extensionPoint =
org.eclipse.core.expressions.definitions
.
.
.
--- Finish processing
com.teamcenter.rac.customization.examples_nnnn\plugin.xml
--> Completed analyzing
com.teamcenter.rac.customization.examples_nnnn.jar, please see
results file:

C:\Temp\RACCustomizationAnalysisResults\com.teamcenter.rac.customiza
tion.examples_nnnn_1712601939508_.xlsx
```

Understanding the rich client customization analyzer report

The analyzer creates a multi-tabbed Excel file report.

The **Summary** tab displays the number of each type of contribution found.

Each type of customization is represented by the other tabs. A list of the individual contributions for each type is displayed on the other tabs.



	A	B	C	D	E
1	**RAC Customization Analysis Results**				
2					
3	**Jar file processed**	**c:\Apps\RacCustomizationAnalyzer\Temp_1712601938583\com.te**			
4					
5	**Date processed**				
6					
7	**Number of "org.eclipse.ui.activities" contributions**	**2**			
8	**Number of "org.eclipse.ui.perspectives" contributions**	**1**			
9	**Number of "org.eclipse.ui.views" contributions**	**5**			
10	**Number of "org.eclipse.core.expressions.definitions" contributions**	**4**			
11	**Number of "org.eclipse.ui.commands" contributions**	**12**	**Number of Unique Command Classes**	**12**	
12	**Number of "org.eclipse.ui.handlers" contributions**	**11**	**Number of Unique Handler Classes**	**11**	
13	**Number of "org.eclipse.ui.menus" contributions**	**12**			
14	**Number of "com.teamcenter.rac.common.tcOpenConfiguration" contributions**	**1**			
15	**Number of "com.teamcenter.rac.aifrcp.openWithViewMenuDef" contributions**	**1**			
16	**Number of "com.teamcenter.rac.aifrcp.viewDefs" contributions**	**5**			
17	**Number of "com.teamcenter.rac.aifrcp.perspectiveDefs" contributions**	**1**			
18	**Number of "org.eclipse.ui.contexts" contributions**	**1**			
19	**Number of "org.eclipse.ui.perspectiveExtensions" contributions**	**1**			
20	**Number of "com.teamcenter.rac.util.extCustomPanel" contributions**	**1**			
21	**Number of "com.teamcenter.rac.ui.commands.boTypesLoader" contributions**	**3**			
22	**Number of "com.teamcenter.rac.util.extWizard" contributions**	**3**			
23	**Number of "com.teamcenter.rac.util.extWizardRef" contributions**	**7**			
24	**Number of "com.teamcenter.rac.ui.commands.operation" contributions**	**2**			
25	**Number of "org.eclipse.e4.workbench.model" contributions**	**3**			
26	**Number of "org.eclipse.e4.ui.css.swt.theme" contributions**	**1**			
27	**Number of "org.eclipse.e4.ui.css.core.propertyHandler" contributions**	**1**			
28	**Number of "org.eclipse.e4.ui.css.core.elementProvider" contributions**	**1**			
29	**Number of "com.teamcenter.rac.viewer.ViewerViewRegistry" contributions**	**3**			
30	**Number of "com.teamcenter.rac.common.openTCObjectInApplication" contribu**	**3**			
31	**Number of "com.teamcenter.rac.kernel.LoginAuthenticator" contributions**	**1**			
32					

Changing the logging level and location

The Teamcenter rich client uses the **log4j2** mechanism for logging.

Since the log4j2 libraries are Java-based, they can not be used directly as a plugin bundle for an Eclipse RCP application. Siemens Digital Industries Software created a wrapper class, **TcLogger**, which is designed to abstract the mechanics of the logger in the rich client. We recommend using the wrapper class to help protect your code from future changes in logger libraries.

There are two ways to work with logging:

- You can change logging parameters in the **log4j2.xml** file, located in the *TC_ROOT***portal\plugins\configuration** directory, to specify the level and scope of logging.

- You can use logging functionality from within your rich client customization code. You can examine the **sample logger customization** as a new rich client view.

Listener leaks

Events in Java are fired by means of listeners. An object registers interest with a target object, so that when an event occurs the listening object is notified. For this relationship to be maintained, the target object must maintain a reference to the listening object. The Java memory management facilities look only to delete objects from virtual memory when the objects are no longer referenced by any other Java object. The problem at hand is the removal of listeners.

Rich client does not currently have a system in place to facilitate the removal of listeners. This creates two issues:

- It causes a memory leak.

  The memory leak issue exists because references are maintained to Java objects that are no longer needed. This creates the situation in which the garbage collector runs but is unable to remove the old objects because they are still tied as listeners. Under this condition, the virtual memory of the Java VM eventually runs out.

- It begins to impact performance of the UI, because old components are being needlessly updated.

  The performance issue is more prevalent than the memory leak. System performance begins to deteriorate quickly under certain UI conditions. The use of the viewer illustrates this, because as new viewers are displayed, they add their components to the session, attached as listeners. The UI appears sluggish and eventually becomes unusable.

The **InterfaceSignalOnClose** and **SignalOnClose** classes are used to remedy listener leaks:

- **InterfaceSignalOnClose**

  The **InterfaceSignalOnClose** class requires the implementation of the **closeSignaled()** method. This interface is designed to signify the desire to be notified when closure is to commence. The **closeSignaled()** method is designed to remove any listeners that were created during the life of the object.

  This interface signifies that the implementing class registers interest to be known when closure occurs. The implementing class is required to implement the **closeSignaled()** method. The **closeSignaled()** method is invoked when closure is commencing (as shown in the following code):

```
public interface InterfaceSignalOnClose
{
    public void closeSignaled();
}
Example: An implementation of the closeSignaled() method can look like the following:
    public void closeSignaled()
    {
```

```
        TCSession session = (TCSession) application.getSession();
        if (session != null)
        {
            session.removeAIFComponentEventListener( this );
        }
    }
```

- **SignalOnClose**

  The **SignalOnClose** class is designed to signal the processing of the components to detach themselves from listeners and prepare to be closed. This class contains a single method, **close()**, which is designed to be passed a reference to a **Container** object. The **Container** object is the start of a recursive walk down the component tree to look for instances of **InterfaceSignalOnClose** classes. If instances are found, the classes are notified that closure is commencing. At this point, it is the responsibility of the implementing class to take appropriate action.

5. Rich client customization using Authorization

Using Authorization to control access to applications and utilities

The Authorization application enables you to control access to Teamcenter administrative applications and utilities based on users' group membership or their role in a group.

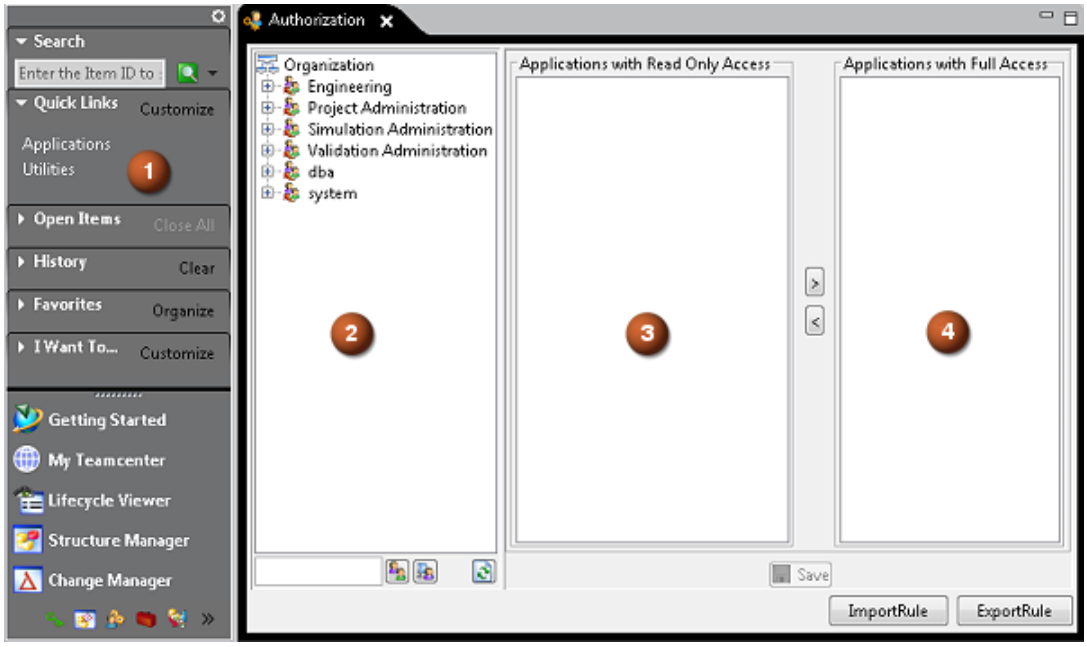
For example, you can:

- Grant all members of the **DBA Lite** group access to the Organization application, regardless of their role within the group.
- Grant users who occupy the **importer** role in the **DBA Lite** group access to the PLM XML/TC XML Export Import Administration application.

The Authorization application works in conjunction with other Teamcenter applications to control access to product features and data, as follows:

- Access to product features is controlled using the Command Suppression application.
- Access to operations on objects, such as delete, copy, and change ownership, is controlled by configuring rules in Access Manager.

Authorization interface



1	**Quick Links**	Click either the **Applications** link or **Utilities** link to set access level.
2	**Organization** tree	Displays the groups and roles in your organization. From the **Organization** tree, you can choose a group or for a selected role.
3	**Applications with Read Only Access**	For the selected group or role in group, the **Applications with Read Only Access** list contains the administrative utilities or applications that are shown in the interface with read only access to the functionality.
4	**Applications with Full Access**	For the selected group or role in group, the **Applications with Full Access** list contains the administrative utilities or applications that are shown in the interface with full access to the functionality.

How Authorization works with group hierarchies

Groups within the organization tree can be configured into one or more hierarchies. Each group has exactly one parent group (unless it is at the root of the hierarchy, when it has no parent group), and each group can have one or more child groups (subgroups).

Authorization rules are inherited within the group hierarchy, as follows:

- Rules defined for a parent group are inherited by all subgroups of the parent group.
- Rules defined at the subgroup level apply only to that subgroup.

> Note:
>
> In the event that two subgroups of different parentage share the same name, rules defined for one parent group are not inherited by the same-name subgroup of the other parent group. For example, if both the **Manufacturing** group and the **Design** group have a **Validation** subgroup, authorization rules defined for the **Manufacturing** group apply only to the **Validation** subgroup that is directly related to the **Manufacturing** group. Likewise, authorization rules defined for the **Design** group apply only to the **Validation** subgroup that is directly related to the **Design** group.

Default authorization rules and Authorization

System-level authorization rules are those rules delivered as part of your standard Teamcenter installation that govern access to administrative applications and utilities. By default, Teamcenter supplies two groups for administrative purposes, the **Project Administration** group and the **dba** group.

Project Administration group members only have access to the Project application, which allows them to create, delete, modify, and add users to or remove users from projects. **dba** group members are granted access to all Teamcenter administrative applications and utilities.

Often, administrative tasks are assigned at a functional level corresponding to your business practices. For example, responsibility for administering user data such as personal and organization information may be assigned to one group, while a different group may be responsible for designing workflow processes. In such cases, **dba** group privileges are more broad and powerful than is necessary or desirable. Authorization enables you to create authorization rules to model access to administrative tools to your business processes.

Where can I limit access using Authorization?

The following applications are supported for access configuration using Authorization:

Access Manager	Organization
ADA License	PLM XML/TC XML Export Import Administration
Audit Manager	Project
Authorization	Setup Wizard
Business Modeler IDE	Subscription Administration
Classification Admin	Workflow Designer

You can configure access to these applications by group or by role in group.

You can also set the **TC_authorization_mode** preference to specify whether to evaluate all the group memberships of users and their role in those groups when authorizing access to an application or to evaluate their current group logon and role in that group.

Example:

To access the Organization application, a user must have **dba** privileges (be a member of **dba** with the role of **DBA**). If the user is a member of both the **dba** and the Engineering groups and logs on under the Engineering group, the user may or may not have access to the Organization application depending on how **TC_authorization_mode** is set:

- If set to **on** (evaluate all group memberships and the roles in the groups), the user has administration privileges based on membership in the **dba** group, even though the user is not currently logged on under that group. The user can access the Organization application.
- If set to **off** (evaluate only the current logon group and the role in that group), the user does not have administration privileges through the Engineering group. Therefore, the user cannot access the Organization application.

The following utilities are supported for access configuration using Authorization:

data_share	dsa_util	purge_invalid_subscriptions
data_sync	export_recovery	update_project_data
database_verify	fscadmin	

You can configure access to these utilities by group or by role in group.

Create Authorization rules for applications and utilities

1. Click the **Applications** link in the **Quick Links** section of the navigation pane.
2. Expand the **Organization** tree and click the group or role to whom you want to grant or deny application access.
3. Select the application from the **Applications with Read Only Access** list to grant access to the group or role in group. Click the right-arrow button to move the application to the **Applications with Full Access** list.

   > Tip:
   >
   > If the **Applications with Read Only Access** list is empty, click any group or role symbol in the **Organization** tree to refresh the list.

4. Click **Save**.

Configure access to utilities by group or by role in group

1. Click the **Utilities** link in the **Quick Links** section of the navigation pane.
2. In the Authorization application pane, expand the **Organization** tree and click the group or role to whom you want to grant or deny utility access.
3. Select the utility from the **Applications with Read Only Access** list to grant access to the group or role in group. Click the right-arrow button to move the utility to the **Applications with Full Access** list.

   > Tip:
   >
   > If the **Applications with Read Only Access** list is empty, click any group or role symbol in the **Organization** tree to refresh the list.

4. Click **Save**.

Sharing authorization rules with other Teamcenter sites

Importing and exporting Authorization rules

Authorization rules can be exported to an operating system directory as an XML file that can then be imported at another Teamcenter site, allowing you to synchronize authorization rules between sites that share data.

Export authorization rules

1. In the Authorization application pane, click the **exportRule** button.
2. In the **exportRule** dialog box, navigate to the directory location where you want to save the rule file.
3. Type a name for the file in the **File name** box.

   > Note:
   >
   > The file is output in XML format; therefore, the file name must end in **.xml**.

4. Click the **exportRule** button.

   The authorization rule file is saved in the operating system directory that you specified in step 2.

Import authorization rules

1. In the Authorization application pane, click the **importRule** button.
2. In the **importRule** dialog box, navigate to the directory containing the authorization rule file that you want to import.

   > Note:
   >
   > Rule files are XML files.

3. Select the authorization rule file.
4. Click the **importRule** button.

   The authorization rule file is imported in to Teamcenter.